Toga Documentation

Release 0.4.9.dev589+g564c79446

Russell Keith-Magee

Feb 13, 2025

CONTENTS

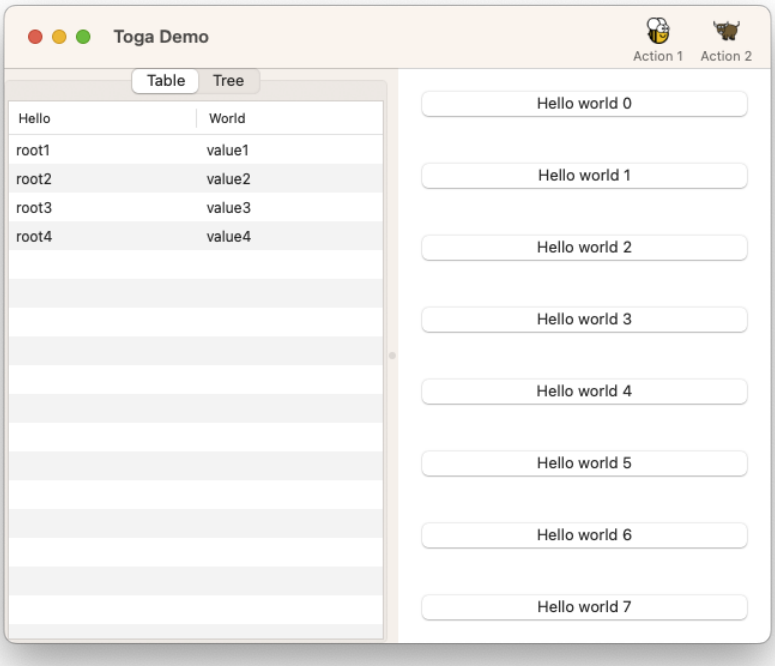
Toga is a Python native, OS native, cross platform GUI toolkit. Toga consists of a library of base components with a shared interface to simplify platform-agnostic GUI development.

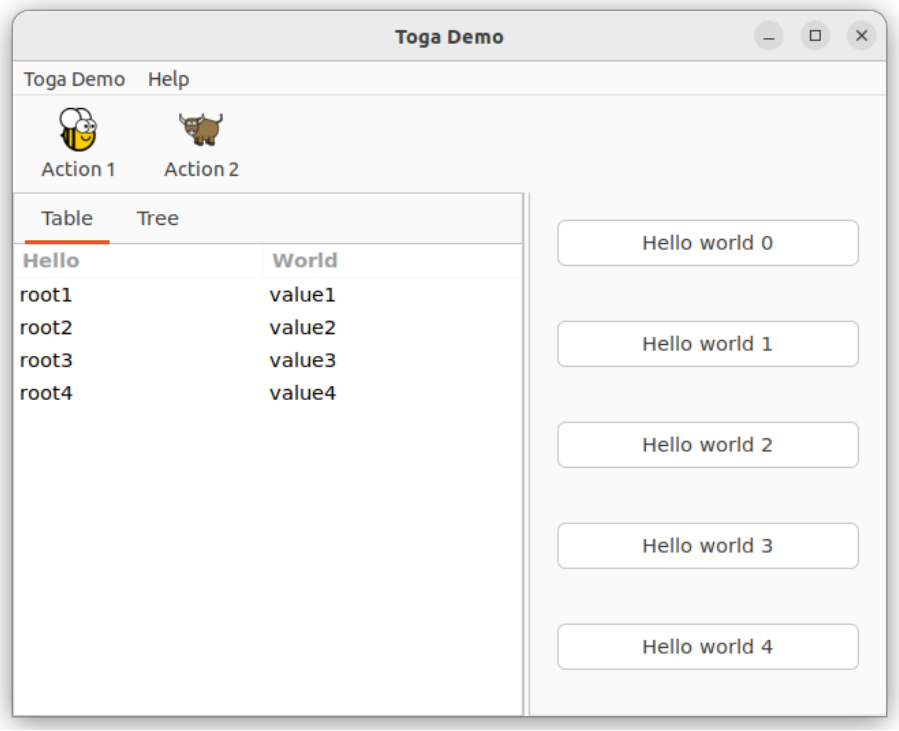
Toga is available on macOS, Windows, Linux (GTK), Android, iOS, for single-page web apps, and console apps.

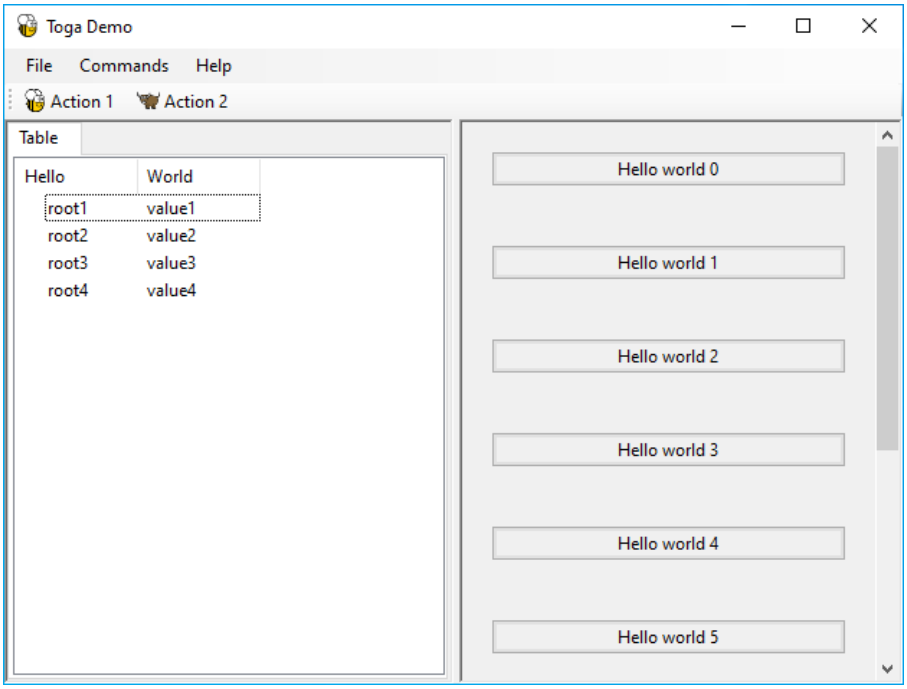
macOS

Linux

Windows




Toga Demo
Toga Demo
Help
Action 1
Action 2
Table
Tree
Hello
World
root1
value1
root2
value2
root3
value3
root4
value4
Hello world 0
Hello world 1
Hello world 2
Hello world 3
Hello world 4


Toga Demo
File
Commands
Help
Action 1
Action 2
Table
Hello
World
root1
value1
root2
value2
root3
value3
root4
value4
Hello world 0
Hello world 1
Hello world 2
Hello world 3
Hello world 4
Hello world 5

CHAPTER

ONE

TUTORIAL

CHAPTER

TWO

REFERENCE

- *API reference*
- *Toga's platform support*
- *Widget support by platform*
- *Managing style with Toga*
- *Toga's plugin interfaces*

CHAPTER

THREE

HOW-TO

- *Topic guides*
- *Contribute to Toga*
- *Internal Toga processes*

CHAPTER

FOUR

BACKGROUND

- *Learn more about Toga*
- *Contacting the BeeWare community*
- *Toga's architecture and other internals*

4.1 Tutorials

Note

Is this the tutorial you're looking for?

If this is your first time using BeeWare, we suggestion you start with the BeeWare tutorial. This tutorial only covers BeeWare's GUI toolkit, Toga, and doesn't cover any of the details of getting your code running on specific hardware platforms. Once you've completed the BeeWare tutorial, this tutorial will introduce more details about Toga's capabilities as a GUI toolkit.

4.1.1 Toga quick start

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

```
$ python -m pip install toga-demo
$ toga-demo
```

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

If you want to see how to write a Toga app of your own, you can *start the tutorial*.

4.1.2 Your first Toga app

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

Set up your development environment

If you haven't got Python 3 installed, you can do so via the official installer, or using your operating system's package manager.

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, open a fresh terminal session, and run:

macOS

Linux

Windows

```
$ mkdir toga-tutorial
$ cd toga-tutorial
$ python3 -m venv venv
$ source venv/bin/activate
```

```
$ mkdir toga-tutorial
$ cd toga-tutorial
$ python3 -m venv venv
$ source venv/bin/activate
```

```
C:\...>mkdir toga-tutorial
C:\...>cd toga-tutorial
C:\...>py -m venv venv
C:\...>venv\Scripts\activate
```

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

Next, install Toga into your virtual environment:

macOS

Linux

Windows

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

Before you install Toga, you'll need to install some system packages.

These instructions are different on almost every version of Linux and Unix; here are some of the common alternatives:

Ubuntu / Debian

```
(venv) $ sudo apt update
(venv) $ sudo apt install git build-essential pkg-config python3-dev libgirepository1.0-
↪dev libcairo2-dev gir1.2-gtk-3.0 libcanberra-gtk3-module
```

Fedora

```
(venv) $ sudo dnf install git gcc make pkg-config python3-devel gobject-introspection-
↪devel cairo-gobject-devel gtk3 libcanberra-gtk3
```

Arch / Manjaro

```
(venv) $ sudo pacman -Syu git base-devel pkgconf python3 gobject-introspection cairo␣
↪gtk3 libcanberra
```

OpenSUSE Tumbleweed

```
(venv) $ sudo zypper install git patterns-devel-base-devel_basis pkgconf-pkg-config␣
↪python3-devel gobject-introspection-devel cairo-devel gtk3 'typelib(Gtk)=3.0'␣
↪libcanberra-gtk3-module
```

FreeBSD

```
(venv) $ sudo pkg update
(venv) $ sudo pkg install git gcc cmake pkgconf python3 gobject-introspection cairo gtk3␣
↪libcanberra-gtk3
```

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

In addition to the dependencies above, if you would like to help add additional support for GTK4, you need to also install `gir1.2-gtk-4.0` on Ubuntu/Debian, or `gtk4` on Fedora or Arch. For other distributions, consult your distributions's platform documentation.

Some widgets (most notably, the *WebView* and *MapView* widgets) have additional system requirements. Likewise, certain hardware features (*Location*) have system requirements.

See the documentation of those widgets and hardware features for details.

Then, install Toga:

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

If you get an error when installing Toga, please ensure that you have fully installed all the platform prerequisites.

Confirm that your system meets the *Windows prerequisites*; then run:

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

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

Write the app

Create a new file called `helloworld.py` in your `toga-tutorial` directory, 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.margin = 50
    button.style.flex = 1
    box.add(button)

    return box


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

(continues on next page)

(continued from previous page)

```
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 a margin 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.margin = 50
```

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

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

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

The `flex` attribute specifies how a widget is sized with respect to other widgets along its direction. The default direction is row (horizontal) and since the button is the only widget 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.toga.tutorial", startup=build)
```

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

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

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

Running the app

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

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

macOS

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

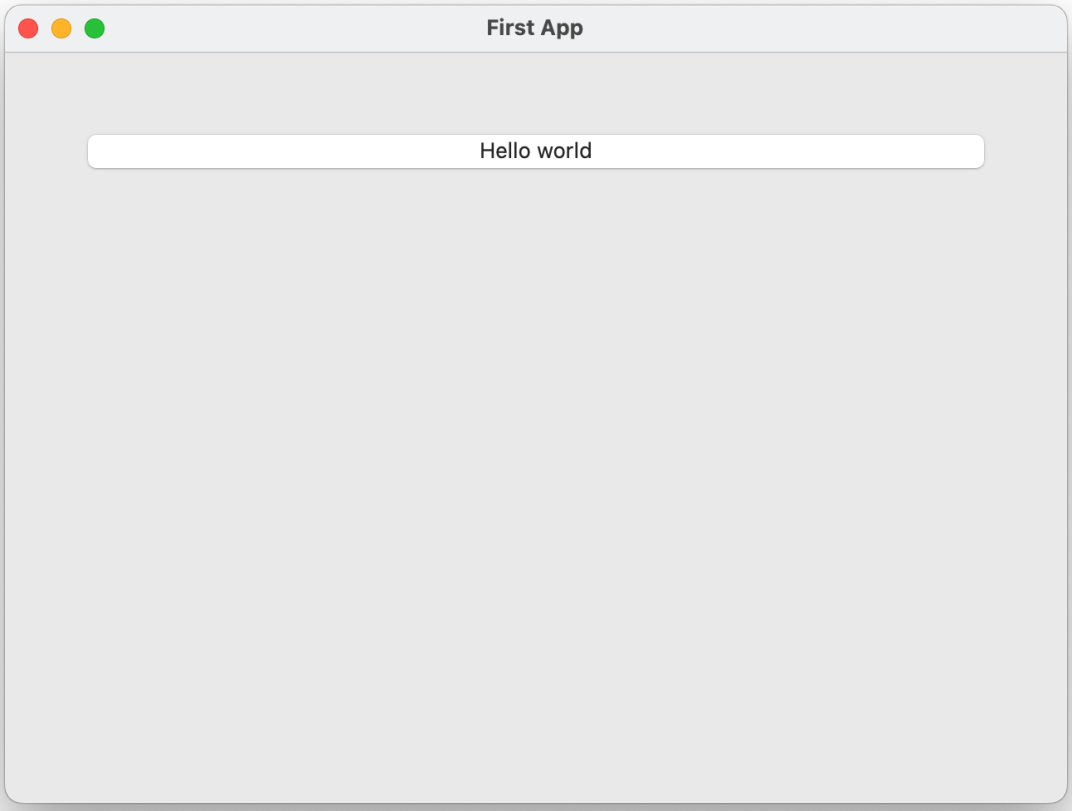
Linux

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

Windows

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

This should pop up a window with a button:



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

Troubleshooting issues

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

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

```
$ python -m pip install toga
```

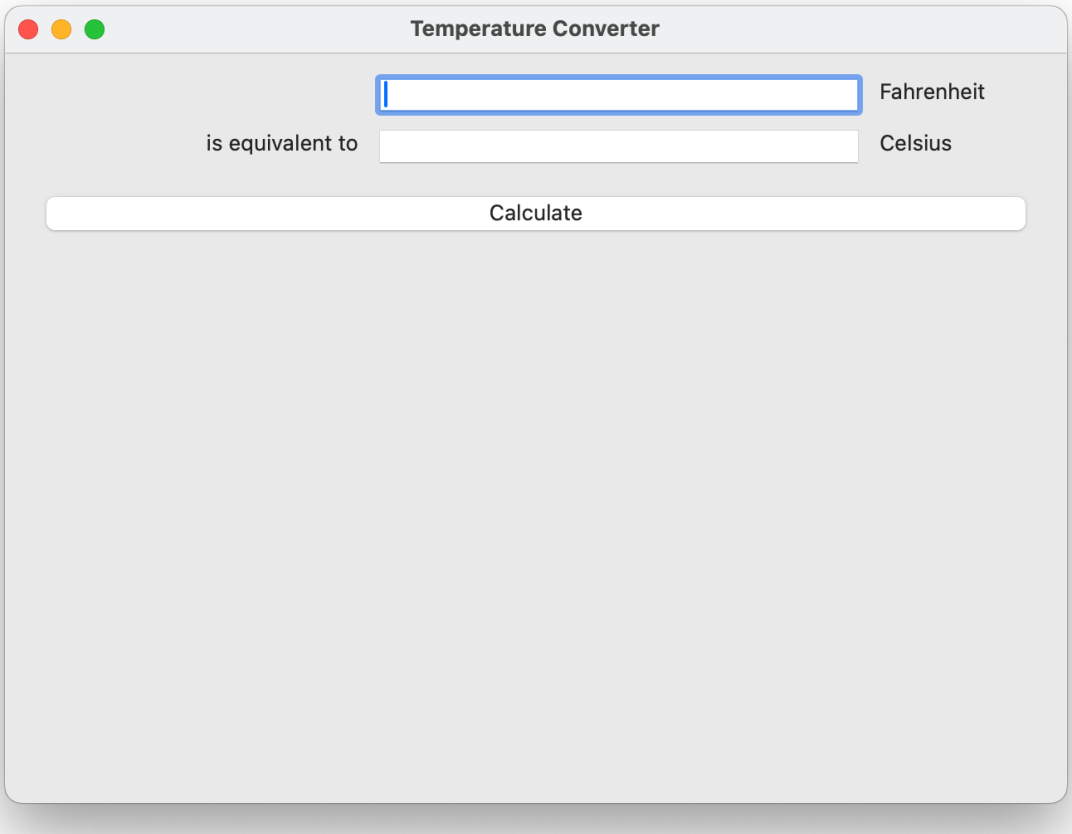
We strongly suggest that you **don't** do this. We'd suggest creating a virtual environment first, and installing toga in that virtual environment as directed at the top of this guide.

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

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

4.1.3 A slightly less toy example

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



Here's the source code:

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


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

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

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

    box.style.update(direction=COLUMN, margin=10, gap=10)
    f_box.style.update(direction=ROW, gap=10)
    c_box.style.update(direction=ROW, gap=10)

    c_input.style.update(flex=1)
    f_input.style.update(flex=1, margin_left=210)
    c_label.style.update(width=100)
    f_label.style.update(width=100)
    join_label.style.update(width=200)

    button.style.update(margin_top=5)

    return box


def main():
    return toga.App(
        "Temperature Converter",
        "org.beeware.toga.examples.tutorial",
        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.

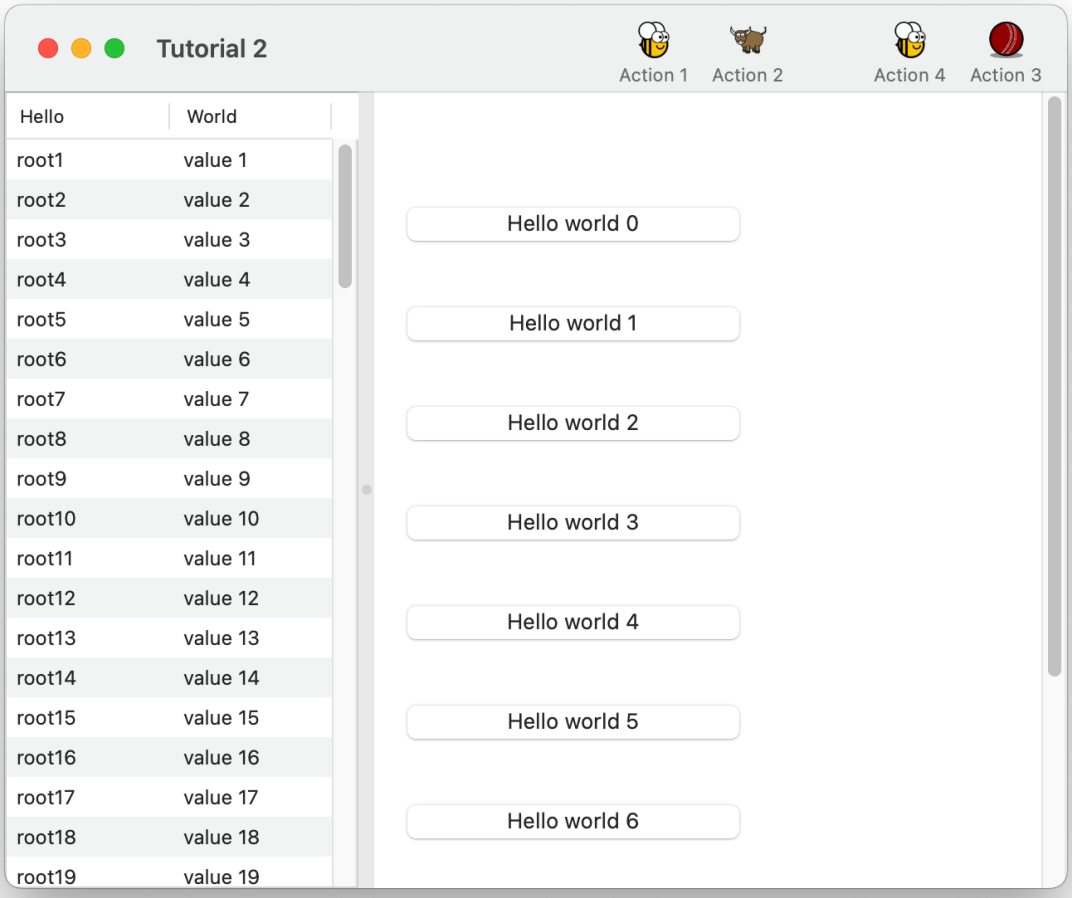
Since there's no width styling on the horizontal boxes, they'll try to fit the widgets they contain into the available space. The `TextInput` widgets have a style of `flex=1`, but the `Label` widgets have a fixed width; as a result, the `TextInput` widgets will be stretched to fit the available horizontal space. The `margin_left` style ensures that the `TextInputs` are aligned, and the `gap` style adds spacing between all the widgets.

4.1.4 You put the box inside another box...

If you've done any GUI programming before, you will know that one of the biggest problems that any widget toolkit solves is how to put widgets on the screen in the right place. Different widget toolkits use different approaches - constraints, packing models, and grid-based models are all common. Toga'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 asyncio

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

```

(continues on next page)

(continued from previous page)

```
async def button_handler(widget):
    print("button handler")
    for i in range(0, 10):
        print("hello", i)
        await asyncio.sleep(1)
    print("done", i)


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


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


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


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


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


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


class Tutorial2App(toga.App):
    def startup(self):
        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, margin_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, margin=20),
                )
            )
```

(continues on next page)

(continued from previous page)

```
        right_container = toga.ScrollContainer(horizontal=False)

        right_container.content = right_content

        split = toga.SplitContainer()

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

        # Create a "Things" menu group to contain some of the commands.
        # No explicit ordering is provided on the group, so it will appear
        # after application-level menus, but *before* the Command group.
        # Items in the Things group are not explicitly ordered either, so they
        # will default to alphabetical ordering within the group.
        things = toga.Group("Things")
        cmd0 = toga.Command(
            action0,
            text="Action 0",
            tooltip="Perform action 0",
            icon=brutus_icon,
            group=things,
        )
        cmd1 = toga.Command(
            action1,
            text="Action 1",
            tooltip="Perform action 1",
            icon=brutus_icon,
            group=things,
        )
        cmd2 = toga.Command(
            action2,
            text="Action 2",
            tooltip="Perform action 2",
            icon=toga.Icon.DEFAULT_ICON,
            group=things,
        )

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

(continues on next page)

(continued from previous page)

```
            order=3,
        )

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

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

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

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

        self.main_window = toga.MainWindow()
        # Command 2 has not been *explicitly* added to the app. Adding it to
        # a toolbar implicitly adds it to the app.
        self.main_window.toolbar.add(cmd1, cmd3, cmd2, cmd4)
        self.main_window.content = split

        self.main_window.show()


def main():
    return Tutorial2App("Tutorial 2", "org.beeware.toga.examples.tutorial")
```

(continues on next page)

(continued from previous page)

```
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.

You'll also see that we're not creating a *`toga.App`* directly. Instead, we're declaring a subclass of *toga.App*, and instantiating that class. This also changes the startup sequence of the app - instead of a function called `build()`, the app invokes a method on the app class named `startup()`. This method behaves slightly differently to our `build()` method - whereas previously the `build()` method returned the content that we wanted to put into our main window, the `startup()` method is responsible for creating and showing the main window of the app.

4.1.5 Let's build a browser!

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

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

Here's the source code:

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


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

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

        box = toga.Box(
            children=[
                toga.Box(
                    children=[
                        self.url_input,
                        toga.Button(
                            "Go",
                            on_press=self.load_page,
                            style=Pack(width=50, margin_left=5),
                        ),
                    ],
                    style=Pack(
                        direction=ROW,
                        align_items=CENTER,
                        margin=5,
                    ),
                ),
                self.webview,
            ],
            style=Pack(direction=COLUMN),
        )

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

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

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

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


def main():
    return Graze("Graze", "org.beeware.toga.examples.tutorial")
```

(continues on next page)

(continued from previous page)

```
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.

This is a simple example of something that *looks* a lot like a browser, but it's a long way from being a replacement for Chrome, Firefox or Safari. As a result, it's missing a lot of features. For example, it requires a full URL, starting with `http://` or `https://`. If you omit this prefix, and press the "Go" button, you'll see an error in the terminal where you started the app.

If you're feeling adventurous, you might try modifying the code to fix this. When the "Go" button is pressed, the `on_press` handler will sets URL of the web view to the content of the text view. Try modifying the `on_press` handler to check if the TextView content starts with a URL-like prefix - and if it doesn't, add `https://` to the start of the string that is used as the web view URL.

Another feature that is missing is that you have to press the button to make the URL load, pressing "enter" after typing a URL won't do anything. However, if you add an `on_confirm` handler to the text input, that handler will be invoked when you press enter. See if you can define an `on_confirm` handler that will do the same thing as the `on_press` handler of the button.

4.1.6 Let's draw on a canvas!

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.constants import Baseline
from toga.fonts import SANS_SERIF
from toga.style import Pack


class StartApp(toga.App):
    def startup(self):
        self.main_window = toga.MainWindow(size=(150, 250))

        # Create empty canvas
        self.canvas = toga.Canvas(
            style=Pack(flex=1),
            on_resize=self.on_resize,
            on_press=self.on_press,
        )
        box = toga.Box(children=[self.canvas])
```

(continues on next page)

(continued from previous page)

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

        # Draw tiberius on the canvas
        self.draw_tiberius()

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

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

    def stroke_head(self):
        with self.canvas.Stroke(line_width=4.0) as head_stroker:
            with head_stroker.ClosedPath(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)

        # Draw eyes separately to avoid miter join
        with self.canvas.Stroke(line_width=4.0) as eye_outline:
            eye_outline.arc(58, 92, 15)
        with self.canvas.Stroke(line_width=4.0) as eye_outline:
            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)
```

(continues on next page)

(continued from previous page)

```
                r_horn_stroker.quadratic_curve_to(130, 60, 109, 75)

        with self.canvas.Context() as l_horn:
            with l_horn.Fill(color=rgb(212, 212, 212)) as l_horn_filler:
                l_horn_filler.move_to(35, 99)
                l_horn_filler.quadratic_curve_to(2, 65, 6, 36)
                l_horn_filler.quadratic_curve_to(17, 60, 37, 75)
            with l_horn.Stroke(line_width=4.0) as l_horn_stroker:
                l_horn_stroker.move_to(35, 99)
                l_horn_stroker.quadratic_curve_to(2, 65, 6, 36)
                l_horn_stroker.quadratic_curve_to(17, 60, 37, 75)

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

    def draw_text(self):
        font = toga.Font(family=SANS_SERIF, size=20)
        self.text_width, text_height = self.canvas.measure_text("Tiberius", font)

        x = (150 - self.text_width) // 2
        y = 175

        with self.canvas.Stroke(color="REBECCAPURPLE", line_width=4.0) as rect_stroker:
            self.text_border = rect_stroker.rect(
                x - 5,
                y - 5,
                self.text_width + 10,
                text_height + 10,
            )
        with self.canvas.Fill(color=rgb(149, 119, 73)) as text_filler:
            self.text = text_filler.write_text("Tiberius", x, y, font, Baseline.TOP)

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

    def on_resize(self, widget, width, height, **kwargs):
        # On resize, center the text horizontally on the canvas. on_resize will be
        # called when the canvas is initially created, when the drawing objects won't
```

(continues on next page)

(continued from previous page)

```
        # exist yet. Only attempt to reposition the text if there's context objects on
        # the canvas.
        if widget.context:
            left_pad = (width - self.text_width) // 2
            self.text.x = left_pad
            self.text_border.x = left_pad - 5
            widget.redraw()

    async def on_press(self, widget, x, y, **kwargs):
        await self.main_window.dialog(
            toga.InfoDialog("Hey!", f"You poked the yak at ({x}, {y})")
        )


def main():
    return StartApp("Tutorial 4", "org.beeware.toga.examples.tutorial")


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

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

4.1.7 A quick test drive

Before you run through the tutorial, *install a Toga app* to see what Toga looks like.

4.1.8 Tutorial 0 - your first Toga app

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

4.1.9 Tutorial 1 - a slightly less toy example

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

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

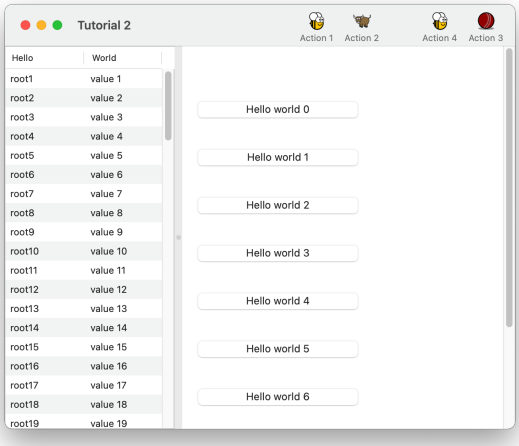
In *You put the box inside another box...*, you will discover how to use the `SplitContainer` widget to display some components, a toolbar and a table.

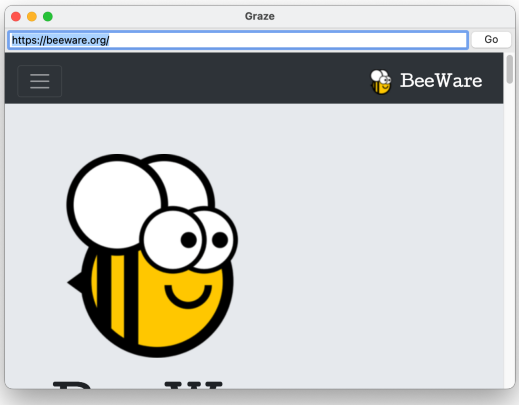
4.1.11 Tutorial 3 - let's build a browser!

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

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

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


Tutorial 2
Action 1
Action 2
Action 4
Action 3
Hello
World
root1
value 1
root2
value 2
root3
value 3
root4
value 4
root5
value 5
root6
value 6
root7
value 7
root8
value 8
root9
value 9
root10
value 10
root11
value 11
root12
value 12
root13
value 13
root14
value 14
root15
value 15
root16
value 16
root17
value 17
root18
value 18
root19
value 19
Hello world 0
Hello world 1
Hello world 2
Hello world 3
Hello world 4
Hello world 5
Hello world 6


Graze
https://beeware.org/
Go
BeeWare

4.2 Reference

4.2.1 API Reference

Core application components

Component	Description
App	The top-level representation of an application.
Window	An operating system-managed container of widgets.
MainWindow	A window that can use the full set of window-level user interface elements.
DocumentWindow	A window that can be used as the main interface to a document-based app.

General widgets

Component	Description
ActivityIndicator	A small animated indicator showing activity on a task of indeterminate length, usually rendered as a "spinner" animation.
Button	A button that can be pressed or clicked.
Canvas	A drawing area for 2D vector graphics.
DateInput	A widget to select a calendar date
DetailedList	An ordered list of content where each item has an icon, a main heading, and a line of supplementary text.
Divider	A separator used to visually distinguish two sections of content in a layout.
ImageView	Image Viewer
Label	A text label for annotating forms or interfaces.
MapView	A zoomable map that can be annotated with location pins.
MultilineTextInput	A scrollable panel that allows for the display and editing of multiple lines of text.
NumberInput	A text input that is limited to numeric input.
PasswordInput	A widget to allow the entry of a password. Any value typed by the user will be obscured, allowing the user to see the number of characters they have typed, but not the actual characters.
ProgressBar	A horizontal bar to visualize task progress. The task being monitored can be of known or indeterminate length.
Selection	A widget to select an single option from a list of alternatives.
Slider	A widget for selecting a value within a range. The range is shown as a horizontal line, and the selected value is shown as a draggable marker.
Switch	A clickable button with two stable states: True (on, checked); and False (off, unchecked). The button has a text label.
Table	A widget for displaying columns of tabular data.
TextInput	A widget for the display and editing of a single line of text.
TimeInput	A widget to select a clock time
Tree	A widget for displaying a hierarchical tree of tabular data.
WebView	An embedded web browser.
Widget	The abstract base class of all widgets. This class should not be be instantiated directly.

Layout widgets

Usage	Description
Box	A generic container for other widgets. Used to construct layouts.
ScrollContainer	A container that can display a layout larger that the area of the container, with overflow controlled by scroll bars.
SplitContainer	A container that divides an area into two panels with a movable border.
OptionContainer	A container that can display multiple labeled tabs of content.

Resources

Component	Description
App Paths	A mechanism for obtaining platform-appropriate file system locations for an application.
Command	A representation of app functionality that the user can invoke from menus or toolbars.
Dialogs	A short-lived window asking the user for input.
Document	A representation of a file on disk that will be displayed in one or more windows
Font	A representation of a Font
Icon	An icon for buttons, menus, etc
Image	An image
Source	A base class for data source implementations.
Status Icons	Icons that appear in the system tray for representing app status while the app isn't visible.
ListSource	A data source describing an ordered list of data.
TreeSource	A data source describing an ordered hierarchical tree of data.
ValueSource	A data source describing a single value.
Validators	A mechanism for validating that input meets a given set of criteria.

Hardware

Usage	Description
Camera	A sensor that can capture photos and/or video.
Location	A sensor that can capture the geographical location of the device.
Screen	A representation of a screen attached to a device.

Other

Component	Description
Constants	Symbolic constants used by various APIs.
Keys	Symbolic representation of keys used for keyboard shortcuts.
Types	Utility data structures used by Toga APIs.

App

The top-level representation of an application.

Table 1: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web	Terminal

Usage

The App class is the top level representation of all application activity. It is a singleton object - any given process can only have a single App. That application may manage multiple windows, but it will generally have at least one window (called the `main_window`).

The application is started by calling `main_loop()`. This will invoke the `startup()` method of the app.

```
import toga

app = toga.App("Simplest App", "com.example.simplest")
app.main_loop()
```

You can populate an app's main window by passing a callable as the `startup` argument to the *toga.App* constructor. This `startup` method must return the content that will be added to the main window of the app.

```
import toga

def create_content(app):
    return toga.Box(children=[toga.Label("Hello!")])

app = toga.App("Simple App", "com.example.simple", startup=create_content)
app.main_loop()
```

This approach to app construction is most useful with simple apps. For most complex apps, you should subclass *toga.App*, and provide an implementation of *startup()*. This implementation *must* assign a value to *main_window* for the app. The possible values are *discussed below*; most apps will assign an instance of *toga.MainWindow*:

```
import toga

class MyApp(toga.App):
    def startup(self):
        self.main_window = toga.MainWindow()
        self.main_window.content = toga.Box(children=[toga.Label("Hello!")])
        self.main_window.show()

if __name__ == '__main__':
    app = MyApp("Realistic App", "org.beeware.realistic")
    app.main_loop()
```

Every app must have a formal name (a human readable name), and an app ID (a machine-readable identifier - usually a reversed domain name). In the examples above, these are provided as constructor arguments. However, you can also provide these details, along with many of the other constructor arguments, as packaging metadata in a format compatible with `importlib.metadata`. If you deploy your app with Briefcase, this will be done automatically.

A Toga app will install a number of default commands to reflect core application functionality (such as the Quit/Exit menu item, and the About menu item). The IDs for these commands are defined as constants on the *Command* class. These commands are automatically installed *before* *startup()* is invoked. If you wish to customize the menu items exposed by your app, you can add or remove commands in your *startup()* implementation.

Assigning a main window

An app *must* assign `main_window` as part of the startup process. However, the value that is assigned as the main window will affect the behavior of the app.

Window

Most apps will assign an instance of *toga.Window* (or a subclass, such as *toga.MainWindow*) as the main window. This window will control the life cycle of the app. When the window assigned as the main window is closed, the app will exit.

If you create an `App` by passing a `startup` argument to the constructor, a *MainWindow* will be automatically created and assigned to `main_window`.

None

If your app doesn't have a single "main" window, but instead has multiple windows that are equally important (e.g., a document editor, or a web browser), you can assign a value of `None` to *main_window*. The resulting behavior is slightly different on each platform, reflecting platform differences.

On macOS, the app is allowed to continue running without having any open windows. The app can open and close windows as required; the app will keep running until explicitly exited. If you give the app focus when it has no open windows, a file dialog will be displayed prompting you to select a file to open. If the file is already open, the existing representation for the document will be given focus.

On Linux and Windows, when an app closes the last window it is managing, the app will automatically exit. Attempting to close the last window will trigger any app-level *on_exit()* handling in addition to any window-specific *on_close()* handling.

Mobile, web and console platforms *must* define a main window.

BACKGROUND

Assigning a value of *toga.App.BACKGROUND* as the main window will allow your app to persist even if it doesn't have any open windows. It will also hide any app-level icon from your taskbar.

Background apps are not supported on mobile, web and console platforms.

Life cycle of an app

Regardless of what an application does, every application goes through the same life cycle of starting, running, and shutting down.

Application startup is handled by the *startup()* method described above. *startup()* *cannot* be an asynchronous method, as it runs *before* the App's event loop is started.

All other events in the life cycle of the app can be managed with event handlers. *toga.App* defines the following event handlers:

- *on_running()* occurs as soon as the app's event loop has started.
- *on_exit()* occurs when the user tries to exit. The handler for this event must return a Boolean value: `True` if the app is allowed to exit; `False` otherwise. This allows an app to abort the exit process (for example, to prevent exit if there are unsaved changes).

Event handlers can be defined by subclassing *toga.App* and overriding the event handler method, by assigning a value to the event handler when the app instance is constructed, or by assigning the event handler attribute on an existing app instance. When the event handler is set by assigning a value to the event handler, the handler method must accept an `app` argument. This argument is not required when subclassing, as the app instance can be implied. Regardless of how they are defined, event handlers *can* be defined as `async` methods.

Managing documents

When you create an App instance, you can declare the type of documents that your app is able to manage by providing a value for `document_types`. When an app declares that it can manage document types, the app will automatically create file management menu items (such as New, Open and Save), and the app will process command line arguments, creating a *toga.Document* instance for each argument matching a registered document type.

For details on how to define and register document types, refer to *the documentation on document handling*.

Notes

- On macOS, menus are tied to the app, not the window; and a menu is mandatory. Therefore, a macOS app will *always* have a menu with the default menu items, regardless of the window being used as the main window.
- Apps executed under Wayland on Linux environment may not show the app's formal name correctly. Wayland considers many aspects of app operation to be the domain of the windowing environment, not the app; as a result, some API requests will be ignored under a Wayland environment. Correctly displaying the app's formal name requires the use of a desktop metadata that Wayland can read. Packaging your app with Briefcase is one way to produce this metadata.

Reference

class toga.**App**(*formal_name=None*, *app_id=None*, *app_name=None*, ***, *icon=None*, *author=None*, *version=None*, *home_page=None*, *description=None*, *startup=None*, *document_types=None*, *on_running=None*, *on_exit=None*)

Create a new App instance.

Once the app has been created, you should invoke the *main_loop()* method, which will start the event loop of your App.

Parameters

- **formal_name** (*str* | *None*) – The human-readable name of the app. If not provided, the metadata key `Formal-Name` must be present.
- **app_id** (*str* | *None*) – The unique application identifier. This will usually be a reversed domain name, e.g. `org.beeware.myapp`. If not provided, the metadata key `App-ID` must be present.
- **app_name** (*str* | *None*) – The name of the distribution used to load metadata with `importlib.metadata`. If not provided, the following will be tried in order:
  1. If the `__main__` module is contained in a package, that package's name will be used.
  2. If the `app_id` argument was provided, its last segment will be used. For example, an `app_id` of `com.example.my-app` would yield a distribution name of `my-app`.
  3. As a last resort, the name `toga`.
- **icon** (*IconContentT* | *None*) – The *icon* for the app. Defaults to *toga.Icon.APP_ICON*.
- **author** (*str* | *None*) – The person or organization to be credited as the author of the app. If not provided, the metadata key `Author` will be used.
- **version** (*str* | *None*) – The version number of the app. If not provided, the metadata key `Version` will be used.
- **home_page** (*str* | *None*) – The URL of a web page for the app. Used in auto-generated help menu items. If not provided, the metadata key `Home-page` will be used.
- **description** (*str* | *None*) – A brief (one line) description of the app. If not provided, the metadata key `Summary` will be used.
- **startup** (`AppStartupMethod` | *None*) – A callable to run before starting the app.
- **on_running** (`OnRunningHandler` | *None*) – The initial *on_running* handler.
- **on_exit** (`OnExitHandler` | *None*) – The initial *on_exit* handler.
- **document_types** (*list[type[Document]]* | *None*) – A list of *Document* classes that this app can manage.

BACKGROUND: str = 'background app'

A constant that can be used as the main window to indicate that an app will run in the background without a main window.

about()

Display the About dialog for the app.

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

Return type
None

add_background_task(*handler*)

DEPRECATED – Use `asyncio.create_task`, or override/assign *on_running()*.

Parameters
handler (BackgroundTask)

Return type
None

app: *App* | None = None

The currently running *App*. Since there can only be one running Toga app in a process, this is available as a class property via `toga.App.app`. If no app has been created yet, this is set to `None`.

property app_id: str

The unique application identifier (read-only). This will usually be a reversed domain name, e.g. `org.beeware.myapp`.

property app_name: str

The name of the distribution used to load metadata with `importlib.metadata` (read-only).

property author: str | None

The person or organization to be credited as the author of the app (read-only).

beep()

Play the default system notification sound.

Return type
None

property camera: *Camera*

A representation of the device's camera (or cameras).

property commands: *CommandSet*

The commands available in the app.

property current_window: *Window* | None

Return the currently active window.

property dark_mode: bool | None

Whether the user has dark mode enabled in their environment (read-only).

Returns
A Boolean describing if the app is in dark mode; `None` if Toga cannot determine if the app is in dark mode.

property description: str | None

A brief (one line) description of the app (read-only).

async dialog(*dialog*)

Display a dialog to the user in the app context.

Param
The *dialog* to display to the user.

Returns
The result of the dialog.

Parameters
dialog (Dialog)

Return type
Coroutine[None, None, Any]

property documents: *DocumentSet*

The list of documents associated with this app.

enter_presentation_mode(*windows*)

Enter into presentation mode with one or more windows on different screens.

Presentation mode is not the same as "Full Screen" mode; presentation mode is when window borders, other window decorations, app menu and toolbars are no longer visible.

Parameters
windows (*list[Window] | dict[Screen, Window]*) – A list of windows, or a dictionary mapping screens to windows, to go into presentation, in order of allocation to screens. If the number of windows exceeds the number of available displays, those windows will not be visible. The windows must have a content set on them.

Raises
ValueError – If the presentation layout supplied is not a list of windows or a dict mapping windows to screens, or if any window does not have content.

Return type
None

exit()

Unconditionally exit the application.

This *does not* invoke the on_exit handler; the app will be immediately and unconditionally closed.

Return type
None

exit_full_screen()

DEPRECATED – Use *App.exit_presentation_mode()*.

Return type
None

exit_presentation_mode()

Exit presentation mode.

Return type
None

property formal_name: str

The human-readable name of the app (read-only).

hide_cursor()

Hide cursor from view.

Return type
None

property home_page: str | None

The URL of a web page for the app (read-only). Used in auto-generated help menu items.

property icon: *Icon*

The Icon for the app.

Can be specified as any valid *icon content*.

property in_presentation_mode: bool

Is the app currently in presentation mode?

property is_bundled: bool

Has the app been bundled as a standalone binary, or is it running as a Python script?

property is_full_screen: bool

DEPRECATED – Use *App.in_presentation_mode*.

property location: *Location*

A representation of the device's location service.

property loop: AbstractEventLoop

The event loop of the app's main thread (read-only).

main_loop()

Start the application.

On desktop platforms, this method will block until the application has exited. On mobile and web platforms, it returns immediately.

Return type
None

property main_window: *Window* | str | None

The main window for the app.

See *the documentation on assigning a main window* for values that can be used for this attribute.

on_exit()

The event handler that will be invoked when the app is about to exit.

The return value of this method controls whether the app is allowed to exit. This can be used to prevent the app exiting with unsaved changes, etc.

If necessary, the overridden method can be defined as an `async` coroutine.

Returns
`True` if the app is allowed to exit; `False` if the app is not allowed to exit.

Return type
bool

on_running()

The event handler that will be invoked when the app's event loop starts running.

If necessary, the overridden method can be defined as an `async` coroutine.

Return type
None

property paths: *Paths*

Paths for platform-appropriate locations on the user's file system.

Some platforms do not allow access to any file system location other than these paths. Even when arbitrary file access is allowed, there are preferred locations for each type of content.

request_exit()

Request an exit from the application.

This method will call the *on_exit()* handler to confirm if the app should be allowed to exit; if that handler confirms the action, the app will exit.

property screens: list[*Screen*]

Returns a list of available screens.

set_full_screen(windows*)**

DEPRECATED – Use *App.enter_presentation_mode()* and *App.exit_presentation_mode()*.

Parameters
windows (Window)

Return type
None

show_cursor()

Make the cursor visible.

Return type
None

startup()

Create and show the main window for the application.

Subclasses can override this method to define customized startup behavior; however, any override *must* ensure the *main_window* has been assigned before it returns.

Return type
None

property status_icons: *StatusIconSet*

The status icons displayed by the app.

property version: str | None

The version number of the app (read-only).

visit_homepage()

Open the application's *home_page* in the default browser.

This method is invoked as a handler by the "Visit homepage" default menu item. If the *home_page* is None, this is a no-op, and the default menu item will be disabled.

Return type
None

property widgets: WidgetRegistry

The widgets managed by the app, over all windows.

Can be used to look up widgets by ID over the entire app (e.g., `app.widgets["my_id"]`).

Only returns widgets that are currently part of a layout. A widget that has been created, but not assigned as part of window content will not be returned by widget lookup.

property windows: *WindowSet*

The windows managed by the app. Windows are automatically added to the app when they are created, and removed when they are closed.

protocol toga.app.**AppStartupMethod**

typing.Protocol.

Classes that implement this protocol must have the following methods / attributes:

__call__(*app*, ***kwargs*)

The startup method of the app.

Called during app startup to set the initial main window content.

Parameters

- **app** (App) – The app instance that is starting.
- **kwargs** (*Any*) – Ensures compatibility with additional arguments introduced in future versions.

Returns
The widget to use as the main window content.

Return type
Widget

protocol toga.app.**BackgroundTask**

typing.Protocol.

Classes that implement this protocol must have the following methods / attributes:

__call__(*app*, ***kwargs*)

Code that should be executed as a background task.

Parameters

- **app** (App) – The app that is handling the background task.
- **kwargs** (*Any*) – Ensures compatibility with additional arguments introduced in future versions.

Return type
object

protocol toga.app.**OnRunningHandler**

typing.Protocol.

Classes that implement this protocol must have the following methods / attributes:

__call__(*app*, ***kwargs*)

A handler to invoke when the app event loop is running.

Parameters

- **app** (App) – The app instance that is running.
- **kwargs** (*Any*) – Ensures compatibility with additional arguments introduced in future versions.

Return type
None

protocol toga.app.**OnExitHandler**

typing.Protocol.

Classes that implement this protocol must have the following methods / attributes:

__call__(*app*, ***kwargs*)

A handler to invoke when the app is about to exit.

The return value of this callback controls whether the app is allowed to exit. This can be used to prevent the app exiting with unsaved changes, etc.

Parameters

- **app** (App) – The app instance that is exiting.
- **kwargs** (*Any*) – Ensures compatibility with additional arguments introduced in future versions.

Returns
`True` if the app is allowed to exit; `False` if the app is not allowed to exit.

Return type
bool

Window

An operating system-managed container of widgets.

macOS

Linux

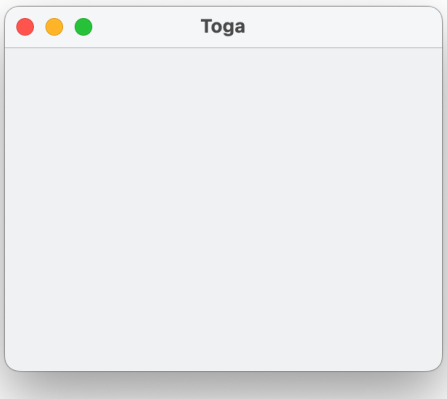
Windows

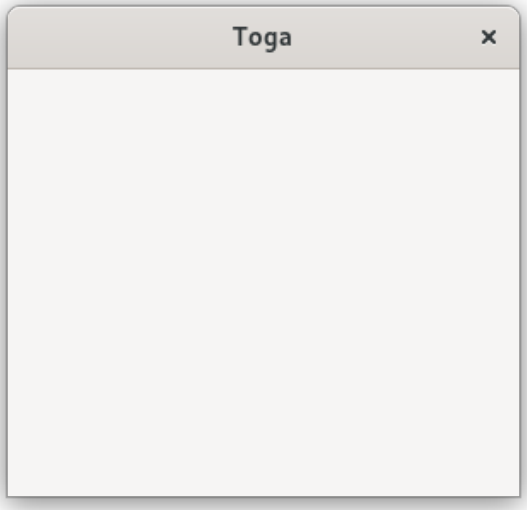
Android

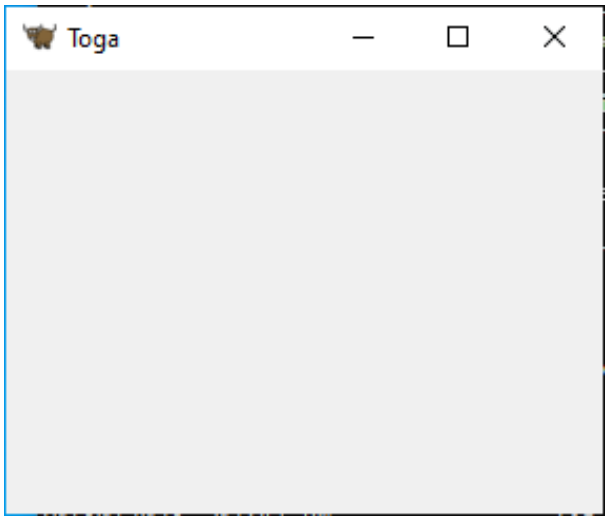
iOS

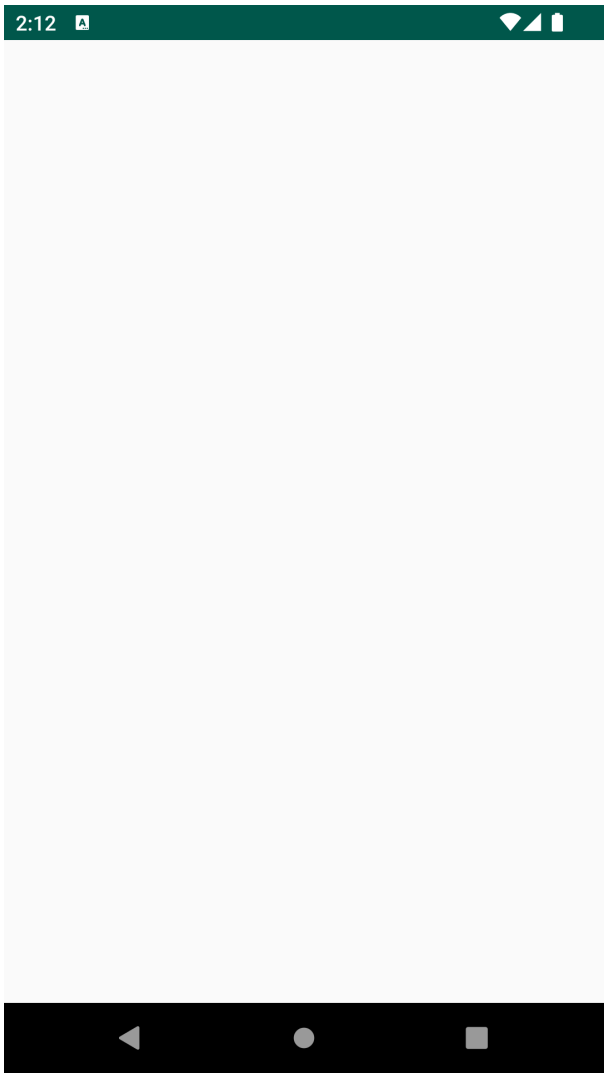
Web

Textual




Toga


Toga


2:12


2:13

Screenshot not available

Screenshot not available

Usage

A window is the top-level container that the operating system uses to display widgets. On desktop platforms, an instance of *Window* will have a title bar, but will not have a menu or toolbar. On mobile, web and console platforms, *Window* is a bare container with no other decoration. Subclasses of *Window* (such as *MainWindow*) add other decorations.

When first created, a window is not visible. To display it, call the *show()* method. The title of the window will default to the formal name of the app.

The window has content, which will usually be a container widget of some kind. The content of the window can be changed by re-assigning its `content` attribute to a different widget.

```
import toga

window = toga.Window()
window.content = toga.Box(children=[...])
window.show()

# Change the window's content to something new
window.content = toga.Box(children=[...])
```

If the user attempts to close the window, Toga will call the `on_close` handler. This handler must return a `bool` confirming whether the close is permitted. This can be used to implement protections against closing a window with unsaved changes.

Once a window has been closed (either by user action, or programmatically with *close()*), it *cannot* be reused. The behavior of any method on a *Window* instance after it has been closed is undefined.

Notes

- The operating system may provide controls that allow the user to resize, reposition, minimize, maximize or close the window. However, the availability of these controls is entirely operating system dependent.
- While Toga provides methods for specifying the size and position of windows, these are ultimately at the discretion of the OS (or window manager). For example, on macOS, depending on a user's OS-level settings, new windows may open as tabs on the main window; on Linux, some window managers (e.g., tiling window managers) may not honor an app's size and position requests. You should avoid making UI design decisions that are dependent on specific size and placement of windows.
- A mobile application can only have a single window (the *main_window*), and that window cannot be moved, resized, hidden, or made full screen. Toga will raise an exception if you attempt to create a secondary window on a mobile platform. If you try to modify the size, position, or visibility of the main window, the request will be ignored.
- On mobile platforms, a window's state cannot be *WindowState.MINIMIZED* or *WindowState.MAXIMIZED*. Any request to move to these states will be ignored.
- On Linux, when using Wayland, a request to put a window into a *WindowState.MINIMIZED* state, or to restore from the *WindowState.MINIMIZED* state, will be ignored, and any associated events like *on_hide()* and *on_show()*, will not be triggered. This is due to limitations in window management features that Wayland allows apps to use.

Reference

class toga.**Window**(*id=None*, *title=None*, *position=None*, *size=Size(640, 480)*, *resizable=True*, *closable=True*, *minimizable=True*, *on_close=None*, *on_gain_focus=None*, *on_lose_focus=None*, *on_show=None*, *on_hide=None*, *content=None*)

Create a new Window.

Parameters

- **id** (*str | None*) – A unique identifier for the window. If not provided, one will be automatically generated.
- **title** (*str | None*) – Title for the window. Defaults to the formal name of the app.
- **position** (*PositionT | None*) – Position of the window, as a *toga.Position* or tuple of `(x, y)` coordinates, in *CSS pixels*.
- **size** (*SizeT*) – Size of the window, as a *toga.Size* or tuple of `(width, height)`, in *CSS pixels*.
- **resizable** (*bool*) – Can the window be resized by the user?
- **closable** (*bool*) – Can the window be closed by the user?
- **minimizable** (*bool*) – Can the window be minimized by the user?
- **on_close** (*OnCloseHandler | None*) – The initial *on_close* handler.
- **content** (*Widget | None*) – The initial content for the window.
- **on_gain_focus** (*OnGainFocusHandler | None*)
- **on_lose_focus** (*OnLoseFocusHandler | None*)
- **on_show** (*OnShowHandler | None*)
- **on_hide** (*OnHideHandler | None*)

property **app**: *App*

The *App* that this window belongs to (read-only).

New windows are automatically associated with the currently active app.

as_image(*format=Image*)

Render the current contents of the window as an image.

Parameters

format (*type[ImageT]*) – Format to provide. Defaults to *Image*; also supports `PIL.Image.Image` if Pillow is installed, as well as any image types defined by installed *image format plugins*.

Returns

An image containing the window content, in the format requested.

Return type

ImageT

property **closable**: bool

Can the window be closed by the user?

close()

Close the window.

This *does not* invoke the `on_close` handler. If the window being closed is the app's main window, it will trigger `on_exit` handling; otherwise, the window will be immediately and unconditionally closed.

Once a window has been closed, it *cannot* be reused. The behavior of any method or property on a *Window* instance after it has been closed is undefined, except for *closed* which can be used to check if the window was closed.

Returns
True if the window was actually closed; False if closing the window triggered `on_exit` handling.

Return type
None

property closed: bool

Whether the window was closed.

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

DEPRECATED - await *dialog()* with a *ConfirmDialog*

Parameters

- **title** (*str*)
- **message** (*str*)
- **on_result** (DialogResultHandler[*bool*] | *None*)

Return type
Dialog

property content: Widget | None

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

async dialog(*dialog*)

Display a dialog to the user, modal to this window.

Param
The *dialog* to display to the user.

Returns
The result of the dialog.

Return type
Coroutine[None, None, *Any*]

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

DEPRECATED - await *dialog()* with an *ErrorDialog*

Parameters

- **title** (*str*)
- **message** (*str*)
- **on_result** (DialogResultHandler[*None*] | *None*)

Return type
Dialog

property full_screen: bool

DEPRECATED – Use *Window.state*.

hide()

Hide the window. If the window is already hidden, this method has no effect.

Raises
ValueError – If the window is currently in a minimized, full screen or presentation state.

Return type
None

property id: str

A unique identifier for the window (read-only).

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

DEPRECATED - await *dialog()* with an *InfoDialog*

Parameters

- **title** (*str*)
- **message** (*str*)
- **on_result** (DialogResultHandler[None] | None)

Return type
Dialog

property minimizable: bool

Can the window be minimized by the user?

property on_close: OnCloseHandler | None

The handler to invoke if the user attempts to close the window.

property on_gain_focus: callable

The handler to invoke if the window gains input focus.

property on_hide: callable

The handler to invoke if the window is hidden from a visible state.

property on_lose_focus: callable

The handler to invoke if the window loses input focus.

property on_show: callable

The handler to invoke if the window is shown from a hidden state.

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

DEPRECATED - await *dialog()* with an *OpenFileDialog*

Parameters

- **title** (*str*)
- **initial_directory** (*Path | str | None*)
- **file_types** (*list[str] | None*)
- **multiple_select** (*bool*)
- **on_result** (DialogResultHandler[list[Path]] | DialogResultHandler[Path] | DialogResultHandler[None] | None)

Return type
Dialog

property position: *Position*

Absolute position of the window, in *CSS pixels*.

The origin is the top left corner of the primary screen.

Raises

RuntimeError – If position change is requested while in *WindowState.FULLSCREEN* or *WindowState.PRESENTATION*.

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

DEPRECATED - await *dialog()* with a *QuestionDialog*

Parameters

- **title** (*str*)
- **message** (*str*)
- **on_result** (DialogResultHandler[*bool*] | *None*)

Return type

Dialog

property resizable: bool

Can the window be resized by the user?

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

DEPRECATED - await *dialog()* with a *SaveFileDialog*

Parameters

- **title** (*str*)
- **suggested_filename** (*Path* | *str*)
- **file_types** (*list*[*str*] | *None*)
- **on_result** (DialogResultHandler[*Path* | *None*] | *None*)

Return type

Dialog

property screen: *Screen*

Instance of the `toga.Screen` on which this window is present.

property screen_position: *Position*

Position of the window with respect to current screen, in *CSS pixels*.

Raises

RuntimeError – If position change is requested while in *WindowState.FULLSCREEN* or *WindowState.PRESENTATION*.

select_folder_dialog(*title*, *initial_directory=None*, *multiple_select=False*, *on_result=None*)

DEPRECATED - await *dialog()* with a *SelectFolderDialog*

Parameters

- **title** (*str*)
- **initial_directory** (*Path* | *str* | *None*)
- **multiple_select** (*bool*)
- **on_result** (DialogResultHandler[*list*[*Path*]] | DialogResultHandler[*Path*] | DialogResultHandler[*None*] | *None*)

Return type
Dialog

show()

Show the window. If the window is already visible, this method has no effect.

Raises
ValueError – If the window is currently in a minimized, full screen or presentation state.

Return type
None

property size: *Size*

Size of the window, in *CSS pixels*.

Raises
RuntimeError – If resize is requested while in *WindowState.FULLSCREEN* or *WindowState.PRESENTATION*.

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

DEPRECATED - await *dialog()* with a *StackTraceDialog*

Parameters

- **title** (*str*)
- **message** (*str*)
- **content** (*str*)
- **retry** (*bool*)
- **on_result** (DialogResultHandler[*bool*] | DialogResultHandler[*None*] | *None*)

Return type
Dialog

property state: *WindowState*

The current state of the window.

When the window is in transition, then this will return the state it is transitioning towards, instead of the actual instantaneous state.

Raises

- **RuntimeError** – If state change is requested while the window is hidden.
- **ValueError** – If any state other than *WindowState.MINIMIZED* or *WindowState.NORMAL* is requested on a non-resizable window.

property title: str

Title of the window. If no title is provided, the title will default to “Toga”.

property visible: bool

Is the window visible?

property widgets: FilteredWidgetRegistry

The widgets contained in the window.

Can be used to look up widgets by ID (e.g., `window.widgets["my_id"]`).

class toga.app.**WindowSet**(*app*)

Bases: MutableSet[*Window*]

A collection of windows managed by an app.

A window is automatically added to the app when it is created, and removed when it is closed. Adding a window to an App's window set automatically sets the *app* property of the Window.

Parameters
app (App)

add(*window*)

Add an element.

Parameters
window (Window)

Return type
None

discard(*window*)

Remove an element. Do not raise an exception if absent.

Parameters
window (Window)

Return type
None

protocol toga.window.**Dialog**

typing.Protocol.

Classes that implement this protocol must have the following methods / attributes:

RESULT_TYPE: str = 'dialog'

protocol toga.window.**OnCloseHandler**

typing.Protocol.

Classes that implement this protocol must have the following methods / attributes:

__call__(*window*, ***kwargs*)

A handler to invoke when a window is about to close.

The return value of this callback controls whether the window is allowed to close. This can be used to prevent a window closing with unsaved changes, etc.

Parameters

- **window** (Window) – The window instance that is closing.
- **kwargs** (*Any*) – Ensures compatibility with arguments added in future versions.

Returns
`True` if the window is allowed to close; `False` if the window is not allowed to close.

Return type
bool

protocol toga.window.**DialogResultHandler**

typing.Protocol.

Classes that implement this protocol must have the following methods / attributes:

__call__(*window*, *result*, ***kwargs*)

A handler to invoke when a dialog is closed.

Parameters

- **window** (*Window*) – The window that opened the dialog.
- **kwargs** (*Any*) – Ensures compatibility with arguments added in future versions.
- **result** (*_DialogResultT*) – The result returned by the dialog.

Return type
object

MainWindow

A window that can use the full set of window-level user interface elements.

macOS

Linux

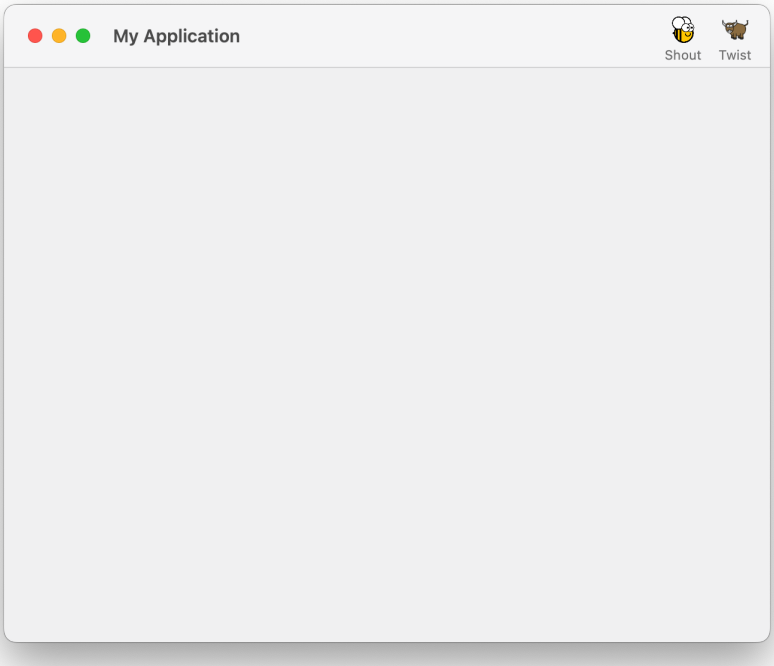
Windows

Android

iOS

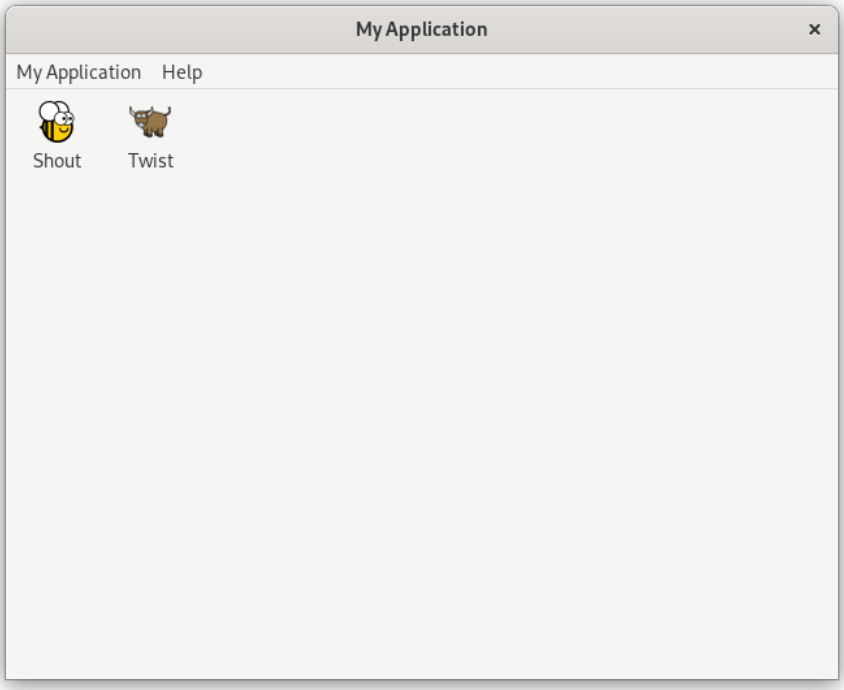
Web

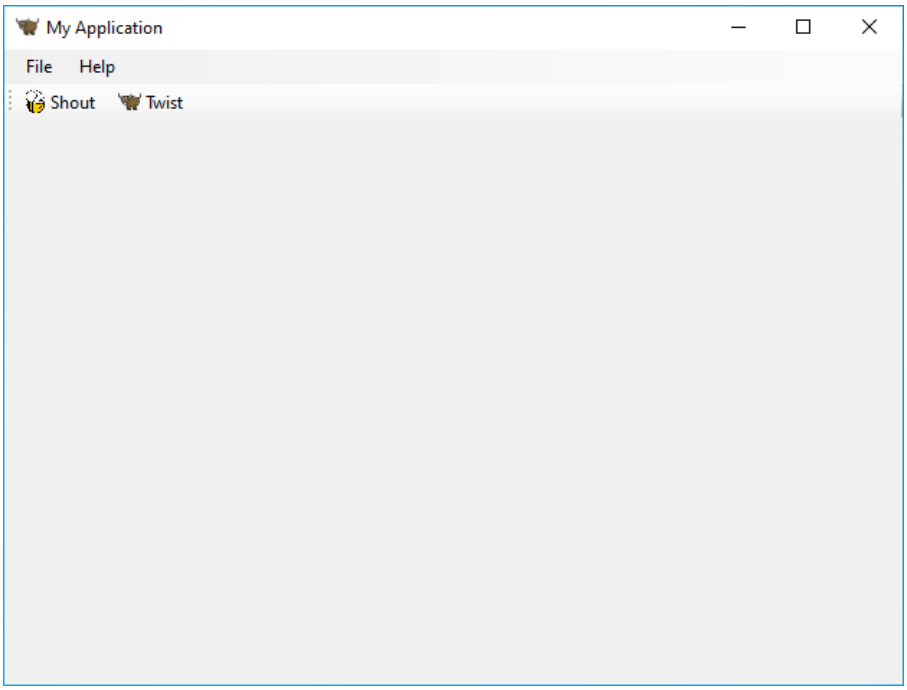
Textual

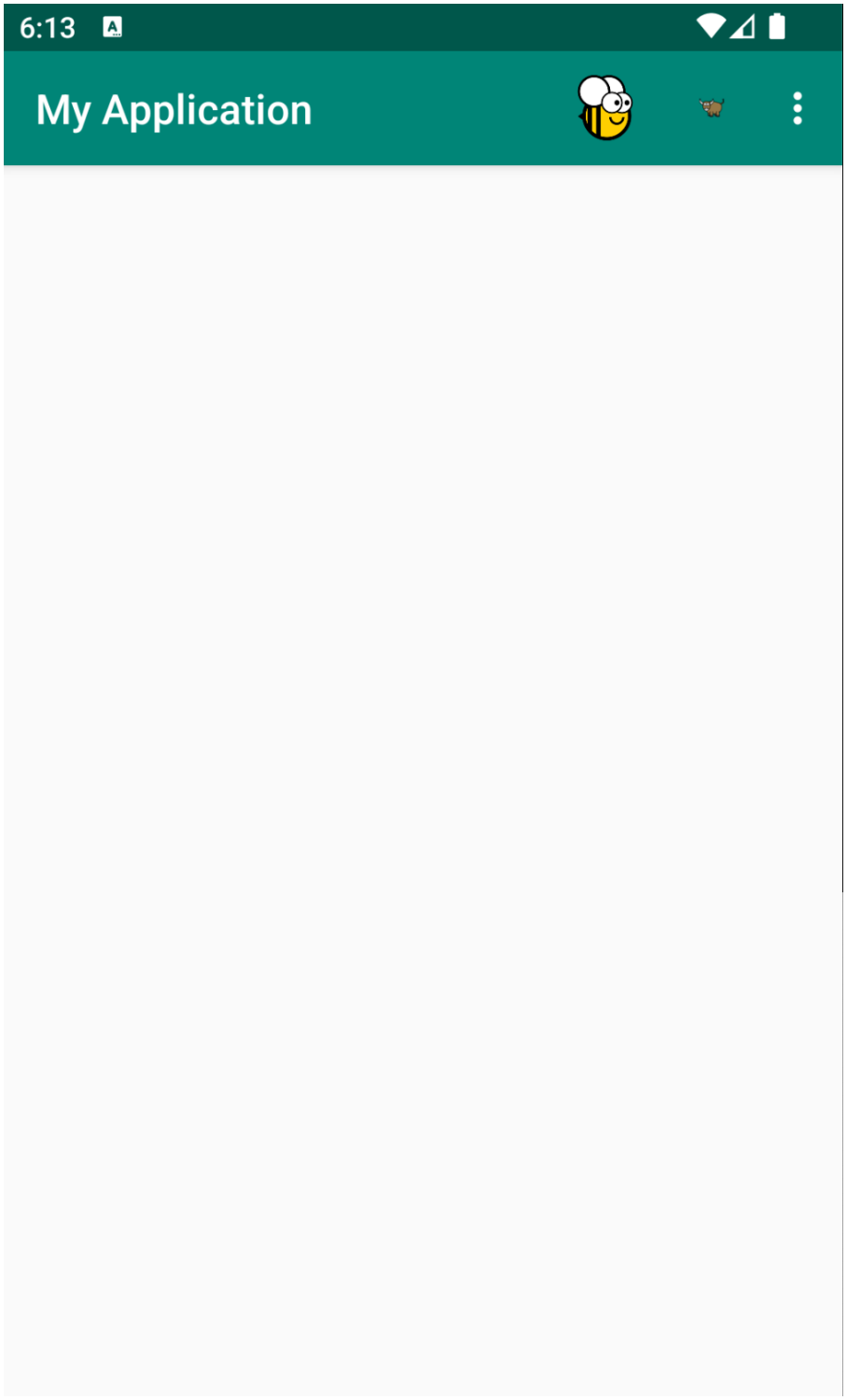


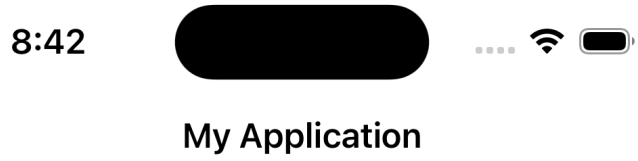
Screenshot not available

Screenshot not available


My Application
My Application
Help
Shout
Twist


My Application
File
Help
Shout
Twist


6:13
My Application


8:42
My Application

Usage

A *toga.MainWindow* is a *toga.Window* that can serve as the main interface to an application. A *toga.MainWindow* may optionally have a toolbar. The presentation of *toga.MainWindow* is platform dependent:

- On desktop platforms that place menus inside windows (e.g., Windows, and most Linux window managers), a *toga.MainWindow* instance will display a menu bar that contains the defined app *commands*.
- On desktop platforms that use an app-level menu bar (e.g., macOS, and some Linux window managers), the window will not have a menu bar; all menu items will be displayed in the app bar.
- On mobile, web and console platforms, a *toga.MainWindow* will include a title bar that can contain both menus and toolbar items.

Toolbar items can be added by adding them to *toolbar*; any command added to the toolbar will be automatically added to the App's commands as well.

```
import toga

main_window = toga.MainWindow(title='My Application')

self.toga.App.main_window = main_window
main_window.show()
```

Reference

class toga.**MainWindow**(**args*, ***kwargs*)

Bases: *Window*

Create a new Main Window.

Accepts the same arguments as *Window*.

property toolbar: *CommandSet*

Toolbar for the window.

DocumentWindow

A window that can be used as the main interface to a document-based app.

macOS

Linux

Windows

Android ×

iOS ×

Web ×

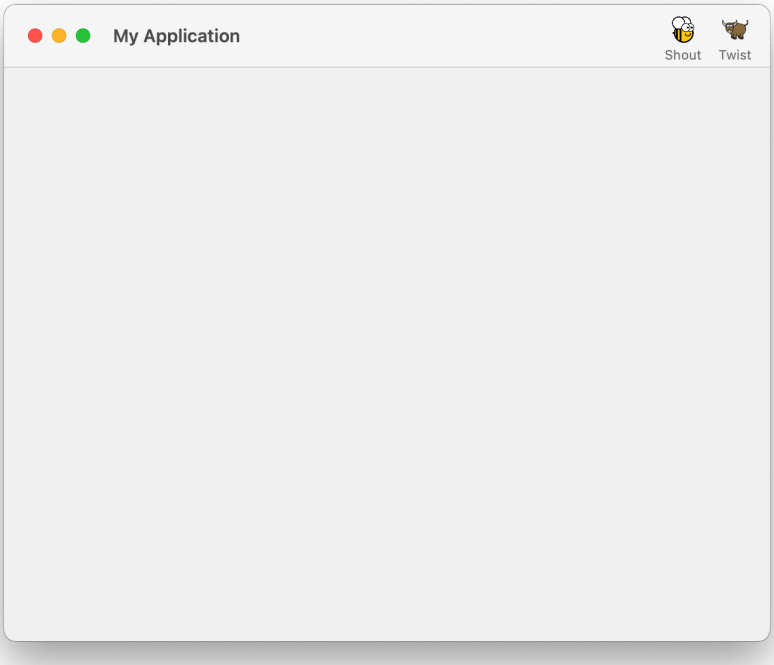
Textual ×

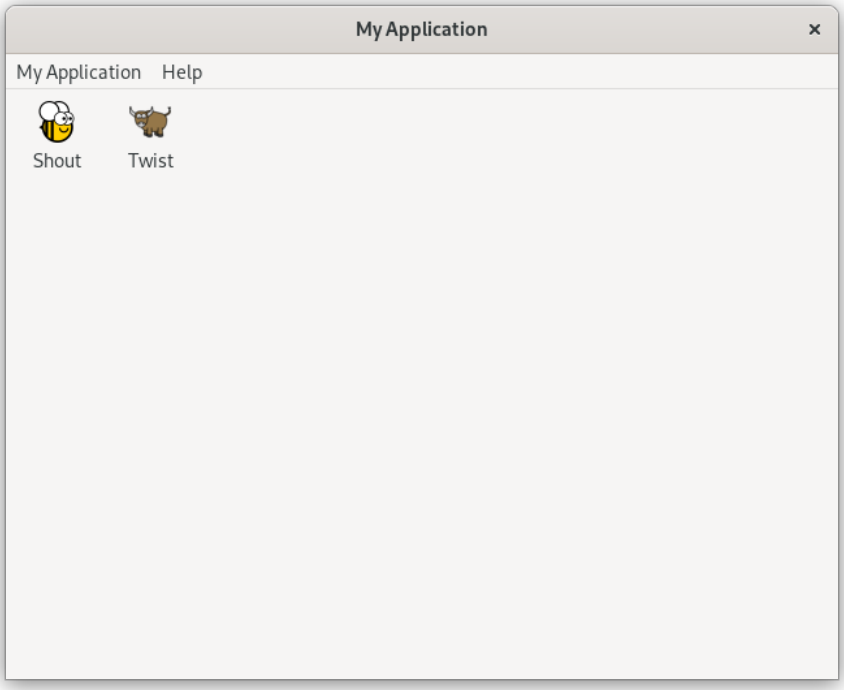
Not supported

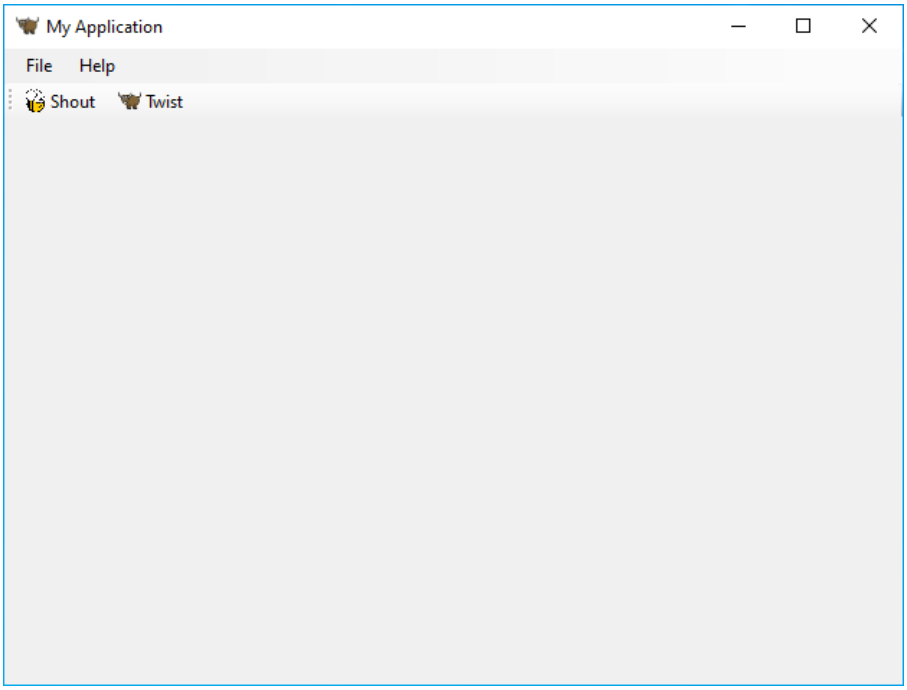
Not supported

Not supported

Not supported


My Application
Shout
Twist


My Application
My Application
Help
Shout
Twist



Usage

A DocumentWindow is the same as a *toga.MainWindow*, except that it is bound to a *toga.Document* instance, exposed as the *toga.DocumentWindow.doc* attribute.

Instances of *toga.DocumentWindow* should be created as part of the *create()* method of an implementation of *toga.Document*.

Reference

class toga.**DocumentWindow**(*doc*, **args*, ***kwargs*)

Bases: *MainWindow*

Create a new document Window.

A document window is a MainWindow (so it will have a menu bar, and *can* have a toolbar), bound to a document instance.

In addition to the required `doc` argument, accepts the same arguments as *Window*.

The default `on_close` handler will use the document's modification status to determine if the document has been modified. It will allow the window to close if the document is fully saved, or the user explicitly declines the opportunity to save.

Parameters
doc (`Document`) – The document being managed by this window

property doc: *Document*

The document displayed by this window.

async save()

Save the document associated with this window.

If the document associated with a window hasn't been saved before, the user will be prompted to provide a filename.

Returns
True if the save was successful; False if the save was aborted, or the document type doesn't define a *write()* method.

async save_as()

Save the document associated with this window under a new filename.

The default implementation will prompt the user for a new filename, then save the document with that new filename.

Returns
True if the save was successful; False if the save was aborted, or the document type doesn't define a *write()* method.

Containers

Box

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

Table 2: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web	Terminal

Usage

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

```
import toga

box = toga.Box()

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

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

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

```
import toga

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

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

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

Reference

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

Bases: *Widget*

Create a new Box container widget.

Parameters

- **id** (*str | None*) – The ID for the widget.
- **style** (*StyleT | None*) – A style object. If no style is provided, a default style will be applied to the widget.
- **children** (*Iterable[Widget] | None*) – An optional list of children for to add to the Box.
- **kwargs** – Initial style properties.

property enabled: bool

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

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

focus()

No-op; Box cannot accept input focus.

Return type
None

toga.**Row**(**args*, ***kwargs*)

Shorthand for *Box* with its *direction* set to “row”.

toga.**Column**(**args*, ***kwargs*)

Shorthand for *Box* with its *direction* set to “column”.

OptionContainer

A container that can display multiple labeled tabs of content.

macOS

Linux

Windows

Android

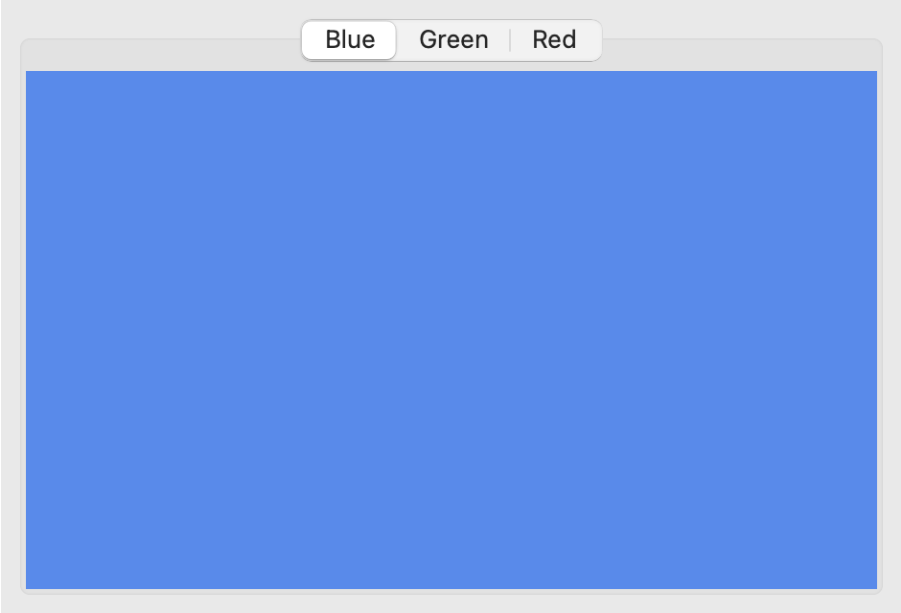
iOS

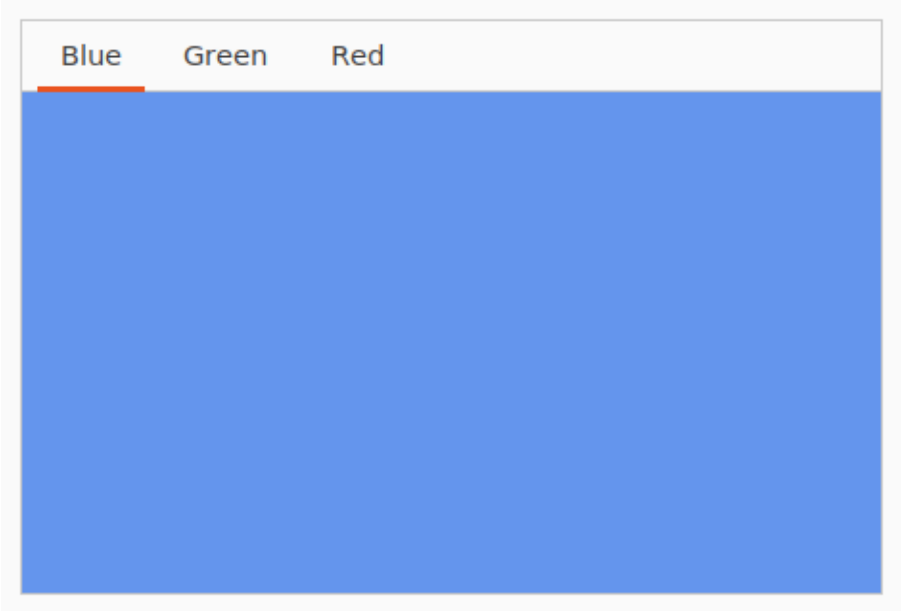
Web ×

Textual ×

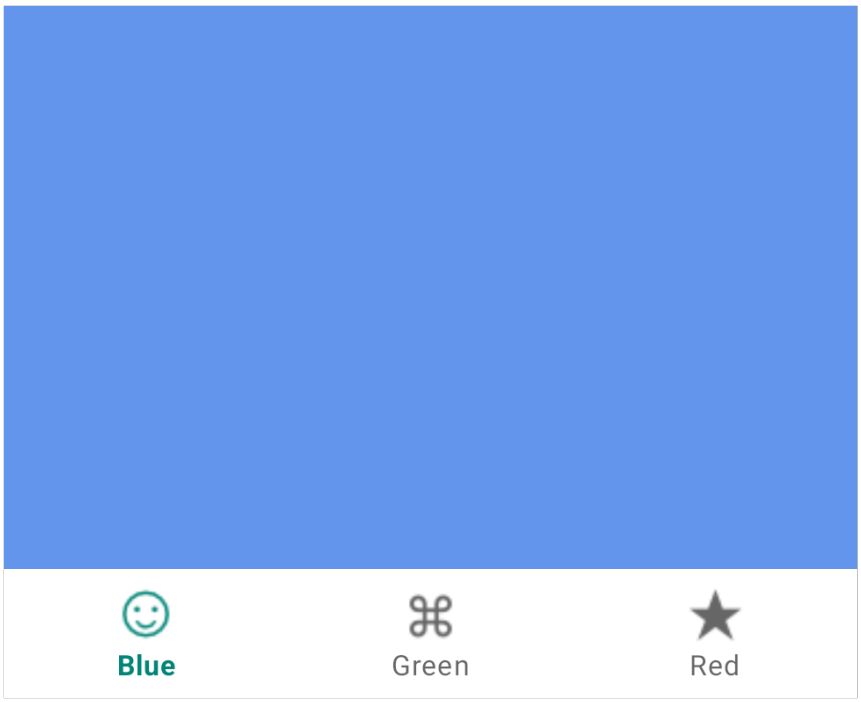
Not supported

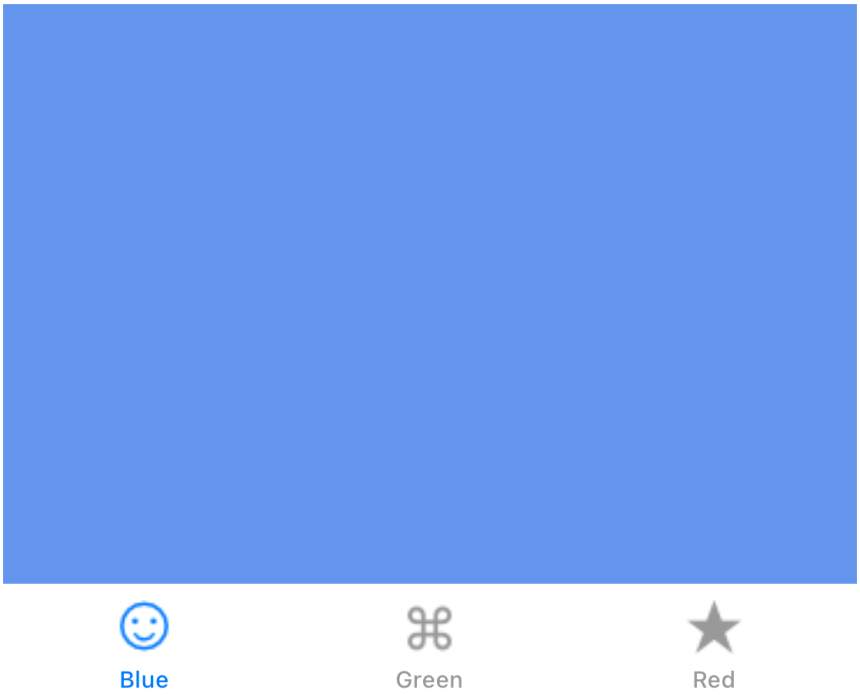
Not supported


Blue
Green
Red


Blue
Green
Red


Blue
Green
Red


Blue
Green
Red



Usage

The content of an OptionContainer is a list of widgets that will form discrete tabs in the display. Each tab can be identified by a label, and, optionally, an icon. This list of content can be modified after initial construction:

```
import toga

pizza = toga.Box()
pasta = toga.Box()

# Create 2 initial tabs; one with an icon, and one without.
container = toga.OptionContainer(
    content=[("Pizza", pizza), ("Pasta", pasta, toga.Icon("pasta"))]
)

# Add another tab of content, without an icon.
salad = toga.Box()
container.content.append("Salad", salad)

# Add another tab of content, with an icon
icecream = toga.Box()
container.content.append("Ice Cream", icecream, toga.Icon("icecream"))
```

OptionContainer content can also be specified by using *toga.OptionItem* instances instead of tuples. This enables you to be explicit when setting an icon or enabled status; it also allows you to set the initial enabled status *without* setting an icon:

```
import toga
```

(continues on next page)

(continued from previous page)

```
pizza = toga.Box()
pasta = toga.Box()

# Create 2 initial tabs; one with an icon, and one without.
container = toga.OptionContainer(
    content=[
      toga.OptionItem("Pizza", pizza),
      toga.OptionItem("Pasta", pasta, icon=toga.Icon("pasta"))
    ]
)

# Add another tab of content, initially disabled, without an icon.
salad = toga.Box()
container.content.append(toga.OptionItem("Salad", salad, enabled=False))
```

When retrieving or deleting items, or when specifying the currently selected item, you can specify an item using:

- The index of the item in the list of content:

```
# Insert a new second tab
container.content.insert(1, "Soup", toga.Box())
# Make the third tab the currently active tab
container.current_tab = 2
# Delete the second tab
del container.content[1]
```

- The string label of the tab:

```
# Insert a tab at the index currently occupied by a tab labeled "Pasta"
container.content.insert("Pasta", "Soup", toga.Box())
# Make the tab labeled "Pasta" the currently active tab
container.current_tab = "Pasta"
# Delete tab labeled "Pasta"
del container.content["Pasta"]
```

- A reference to an *toga.OptionItem*:

```
# Get a reference to the "Pasta" tab
pasta_tab = container.content["Pasta"]
# Insert content at the index currently occupied by the pasta tab
container.content.insert(pasta_tab, "Soup", toga.Box())
# Make the pasta tab the currently active tab
container.current_tab = pasta_tab
# Delete the pasta tab
del container.content[pasta_tab]
```

System requirements

- Using OptionContainer on Android requires the Material package in your project's Gradle dependencies. Ensure your app declares a dependency on `com.google.android.material:material:1.12.0` or later.

Notes

- The use of icons on tabs varies between platforms. If the platform requires icons, and no icon is provided, a default icon will be used. If the platform does not support icons, any icon provided will be ignored, and requests to retrieve the icon will return `None`.
- The behavior of disabled tabs varies between platforms. Some platforms will display the tab, but put it in an unselectable state; some will hide the tab. A hidden tab can still be referenced by index - the tab index refers to the logical order, not the visible order.
- iOS can only display 5 tabs. If there are more than 5 visible tabs in an OptionContainer, the last item will be converted into a "More" option that will allow the user to select the additional items. While the "More" menu is displayed, the current tab will return as `None`.
- Android can only display 5 tabs. The API will allow you to add more than 5 tabs, and will allow you to programmatically control tabs past the 5-item limit, but any tabs past the limit will not be displayed or be selectable by user interaction. If the OptionContainer has more than 5 tabs, and one of the visible tabs is removed, one of the previously unselectable tabs will become visible and selectable.
- iOS allows the user to rearrange icons on an OptionContainer. When referring to tabs by index, user re-ordering is ignored; the logical order as configured in Toga itself is used to identify tabs.
- Icons for iOS OptionContainer tabs should be 25x25px alpha masks.
- Icons for Android OptionContainer tabs should be 24x24px alpha masks.

Reference

type **OptionContainerContentT**

An item of *OptionContainer* content can be:

- a 2-tuple, containing the title for the tab, and the content widget;
- a 3-tuple, containing the title, content widget, and *icon* for the tab;
- a 4-tuple, containing the title, content widget, *icon* for the tab, and enabled status; or
- an *toga.OptionItem* instance.

class toga.**OptionContainer**(*id=None*, *style=None*, *content=None*, *on_select=None*, ***kwargs*)

Bases: *Widget*

Create a new OptionContainer.

Parameters

- **id** (*str | None*) – The ID for the widget.
- **style** (*StyleT | None*) – A style object. If no style is provided, a default style will be applied to the widget.
- **content** (*Iterable[OptionContainerContentT] | None*) – The initial *OptionContainer content* to display in the OptionContainer.
- **on_select** (*toga.widgets.optioncontainer.OnSelectHandler | None*) – Initial *on_select* handler.
- **kwargs** – Initial style properties.

property content: *OptionList*

The tabs of content currently managed by the OptionContainer.

property current_tab: *OptionItem* | None

The currently selected tab of content, or None if there are no tabs, or the OptionContainer is in a state where no tab is currently selected.

This property can also be set with an int index, or a str label.

property enabled: bool

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

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

focus()

No-op; OptionContainer cannot accept input focus.

Return type
None

property on_select: *OnSelectHandler*

The callback to invoke when a new tab of content is selected.

class toga.**OptionItem**(*text*, *content*, *, *icon=None*, *enabled=True*)

A tab of content in an OptionContainer.

Parameters

- **text** (*str*) – The text label for the new tab.
- **content** (Widget) – The content widget to use for the new tab.
- **icon** (*IconContentT | None*) – The *icon content* to use to represent the tab.
- **enabled** (*bool*) – Should the new tab be enabled?

property content: *Widget*

The content widget displayed in this tab of the OptionContainer.

property enabled: bool

Is the panel of content available for selection?

property icon: *Icon*

The Icon for the tab of content.

Can be specified as any valid *icon content*.

If the platform does not support the display of icons, this property will return None regardless of any value provided.

property index: int | None

The index of the tab in the OptionContainer.

Returns None if the tab isn't currently part of an OptionContainer.

property interface: *OptionContainer*

The OptionContainer that contains this tab.

Returns None if the tab isn't currently part of an OptionContainer.

property text: str

The label for the tab of content.

class toga.widgets.optioncontainer.**OptionList**(*interface*)

Parameters
interface (*Any*)

__delitem__(*index*)

Same as *remove*.

Parameters
index (*int* | *str* | OptionItem)

Return type
None

__getitem__(*index*)

Obtain a specific tab of content.

Parameters
index (*int* | *str* | OptionItem)

Return type
OptionItem

append(*text_or_item:* OptionItem) → None

append(*text_or_item: str*, *content:* Widget, *, *icon: IconContentT | None = None*, *enabled: bool | None = True*) → None

Add a new tab of content to the OptionContainer.

The new tab can be specified as an existing *OptionItem* instance, or by specifying the full details of the new tab of content. If an *OptionItem* is provided, specifying `content`, `icon` or `enabled` will raise an error.

Parameters

- **text_or_item** – An *OptionItem*; or, the text label for the new tab.
- **content** – The content widget to use for the new tab.
- **icon** – The *icon content* to use to represent the tab.
- **enabled** – Should the new tab be enabled? (Default: `True`)

index(*value*)

Find the index of the tab that matches the given value.

Parameters
value (*str* | *int* | OptionItem) – The value to look for. An integer is returned as-is; if an *OptionItem* is provided, that item's index is returned; any other value will be converted into a string, and the first tab with a label matching that string will be returned.

Raises
ValueError – If no tab matching the value can be found.

Return type
int

insert(*index: int | str |* OptionItem, *text_or_item:* OptionItem) → None

insert(*index: int | str |* OptionItem, *text_or_item: str*, *content:* Widget, *, *icon: IconContentT | None = None*, *enabled: bool | None = True*) → None

Insert a new tab of content to the OptionContainer at the specified index.

The new tab can be specified as an existing *OptionItem* instance, or by specifying the full details of the new tab of content. If an *OptionItem* is provided, specifying `content`, `icon` or `enabled` will raise an error.

Parameters

- **index** – The index where the new tab should be inserted.
- **text_or_item** – An *OptionItem*; or, the text label for the new tab.
- **content** – The content widget to use for the new tab.
- **icon** – The *icon content* to use to represent the tab.
- **enabled** – Should the new tab be enabled? (Default: `True`)

remove(*index*)

Remove the specified tab of content.

The currently selected item cannot be deleted.

Parameters
index (*int | str | OptionItem*) – The index where the new tab should be inserted.

Return type
None

class toga.widgets.optioncontainer.**OptionItem**(*text*, *content*, *, *icon=None*, *enabled=True*)

A tab of content in an OptionContainer.

Parameters

- **text** (*str*) – The text label for the new tab.
- **content** (*Widget*) – The content widget to use for the new tab.
- **icon** (*IconContentT | None*) – The *icon content* to use to represent the tab.
- **enabled** (*bool*) – Should the new tab be enabled?

property content: *Widget*

The content widget displayed in this tab of the OptionContainer.

property enabled: bool

Is the panel of content available for selection?

property icon: *Icon*

The Icon for the tab of content.

Can be specified as any valid *icon content*.

If the platform does not support the display of icons, this property will return `None` regardless of any value provided.

property index: int | None

The index of the tab in the OptionContainer.

Returns `None` if the tab isn't currently part of an OptionContainer.

property interface: *OptionContainer*

The OptionContainer that contains this tab.

Returns `None` if the tab isn't currently part of an OptionContainer.

property text: *str*

The label for the tab of content.

protocol toga.widgets.optioncontainer.**OnSelectHandler**

typing.Protocol.

Classes that implement this protocol must have the following methods / attributes:

__call__(*widget*, ***kwargs*)

A handler that will be invoked when a new tab is selected in the OptionContainer.

Parameters

- **widget** (OptionContainer) – The OptionContainer that had a selection change.
- **kwargs** (*Any*) – Ensures compatibility with arguments added in future versions.

Return type

None

ScrollContainer

A container that can display a layout larger than the area of the container, with overflow controlled by scroll bars.

macOS

Linux

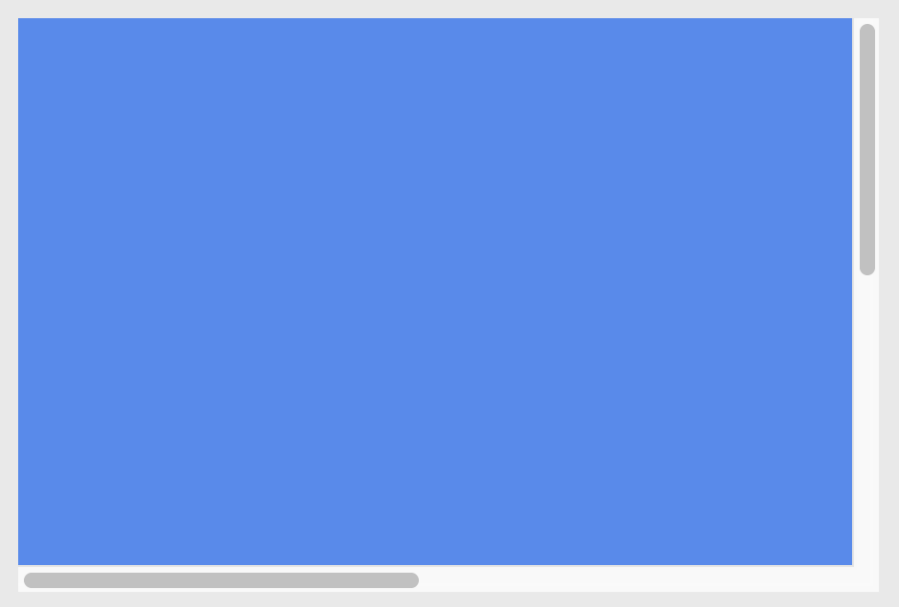
Windows

Android

iOS

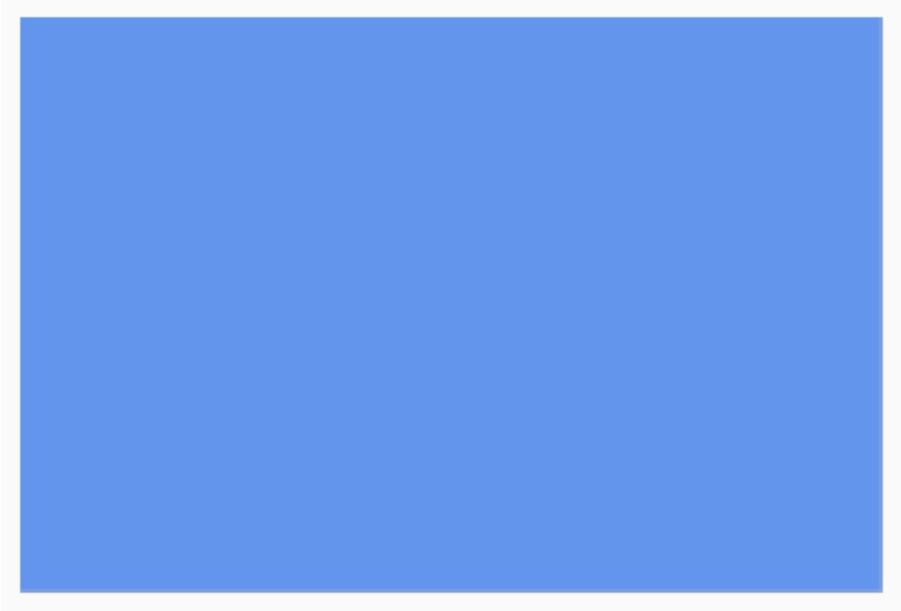
Web ×

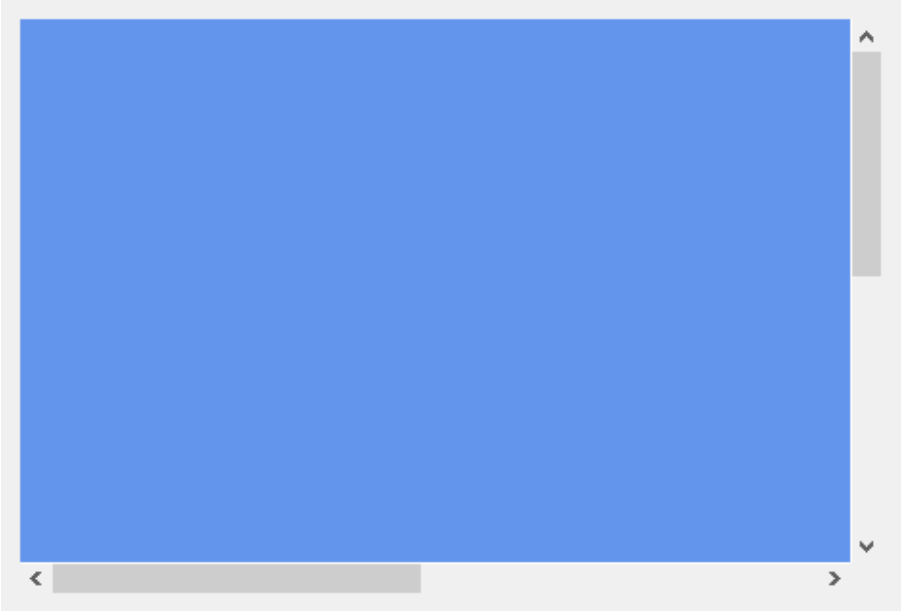
Textual ×



Not supported

Not supported





Usage

```
import toga

content = toga.Box(children=[...])

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

Reference

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

Bases: *Widget*

Create a new Scroll Container.

Parameters

- **id** (*str | None*) – The ID for the widget.
- **style** (*StyleT | None*) – A style object. If no style is provided, a default style will be applied to the widget.
- **horizontal** (*bool*) – Should horizontal scrolling be permitted?
- **vertical** (*bool*) – Should horizontal scrolling be permitted?
- **on_scroll** (*OnScrollHandler | None*) – Initial *on_scroll* handler.
- **content** (*Widget | None*) – The content to display in the scroll window.
- **kwargs** – Initial style properties.

property content: *Widget* | None

The root content widget displayed inside the scroll container.

property enabled: Literal[True]

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

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

focus()

No-op; ScrollContainer cannot accept input focus.

Return type
None

property horizontal: bool

Is horizontal scrolling enabled?

property horizontal_position: int

The current horizontal scroll position.

If the value provided is negative, or greater than the maximum horizontal position, the value will be clipped to the valid range.

Returns
The current horizontal scroll position.

Raises
ValueError – If an attempt is made to change the horizontal position when horizontal scrolling is disabled.

property max_horizontal_position: int

The maximum horizontal scroll position (read-only).

property max_vertical_position: int

The maximum vertical scroll position (read-only).

property on_scroll: *OnScrollHandler*

Handler to invoke when the user moves a scroll bar.

property position: *Position*

The current scroll position.

If the value provided for either axis is negative, or greater than the maximum position in that axis, the value will be clipped to the valid range.

If scrolling is disabled in either axis, the value provided for that axis will be ignored.

property vertical: bool

Is vertical scrolling enabled?

property vertical_position: int

The current vertical scroll position.

If the value provided is negative, or greater than the maximum vertical position, the value will be clipped to the valid range.

Returns
The current vertical scroll position.

Raises
ValueError – If an attempt is made to change the vertical position when vertical scrolling is disabled.

protocol toga.widgets.scrollcontainer.**OnScrollHandler**

typing.Protocol.

Classes that implement this protocol must have the following methods / attributes:

__call__(*widget*, ***kwargs*)

A handler to invoke when the container is scrolled.

Parameters

- **widget** (ScrollContainer) – The ScrollContainer that was scrolled.
- **kwargs** (*Any*) – Ensures compatibility with arguments added in future versions.

Return type
object

SplitContainer

A container that divides an area into two panels with a movable border.

macOS

Linux

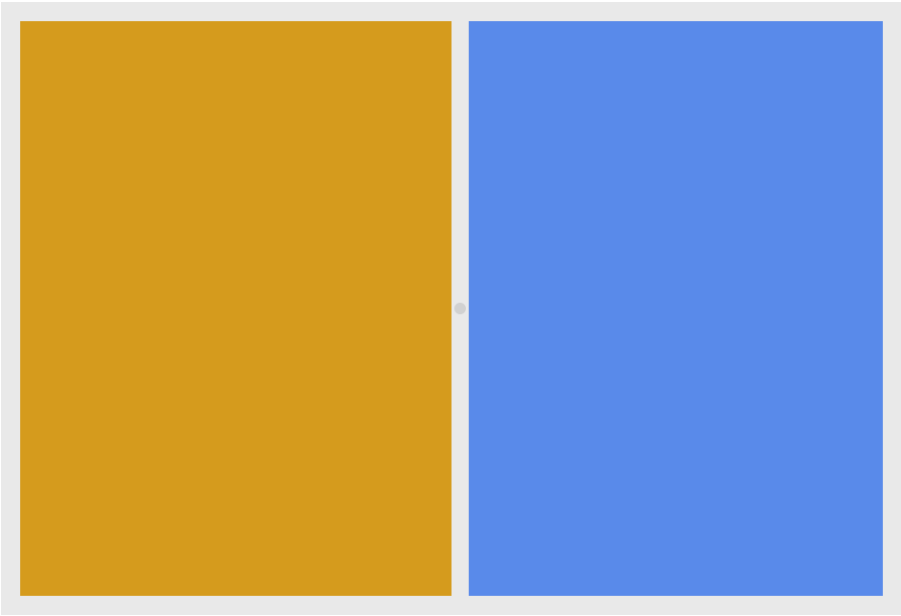
Windows

Android ×

iOS ×

Web ×

Textual ×

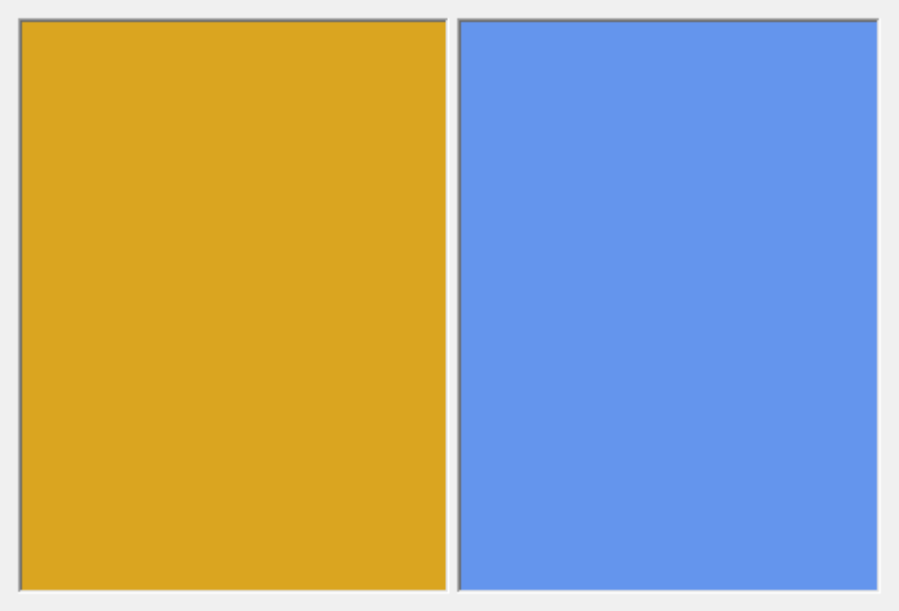


Not supported

Not supported

Not supported

Not supported



Usage

```
import toga

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

split = toga.SplitContainer(content=[left_container, right_container])
```

Content can be specified when creating the widget, or after creation by assigning the `content` attribute. The direction of the split can also be configured, either at time of creation, or by setting the `direction` attribute:

```
import toga
from toga.constants import Direction

split = toga.SplitContainer(direction=Direction.HORIZONTAL)

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

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

By default, the space of the SplitContainer will be evenly divided between the two panels. To specify an uneven split, you can provide a flex value when specifying content. In the following example, there will be a 60/40 split between the left and right panels.

```
import toga

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

split.content = [(left_container, 3), (right_container, 2)]
```

This only specifies the initial split; the split can be modified by the user once it is displayed.

Reference

type **SplitContainerContentT**

An item of *SplitContainer* content can be:

- a *Widget*; or
- a 2-tuple, containing a *Widget*, and an `int` flex value

class toga.**SplitContainer**(*id=None*, *style=None*, *direction=Direction.VERTICAL*, *content=None*, ***kwargs*)

Bases: *Widget*

Create a new SplitContainer.

Parameters

- **id** (*str | None*) – The ID for the widget.
- **style** (*StyleT | None*) – A style object. If no style is provided, a default style will be applied to the widget.
- **direction** (Direction) – The direction in which the divider will be drawn. Either *HORIZONTAL* or *VERTICAL*; defaults to *VERTICAL*
- **content** (*Sequence[SplitContainerContentT] | None*) – Initial *SplitContainer content* of the container. Defaults to both panels being empty.
- **kwargs** – Initial style properties.

property content: `list`[SplitContainerContentT]

The widgets displayed in the SplitContainer.

This property accepts a sequence of exactly 2 elements, each of which can be either:

- A *Widget* to display in the panel.
- `None`, to make the panel empty.
- A tuple consisting of a Widget (or `None`) and the initial flex value to apply to that panel in the split, which must be greater than 0.

If a flex value isn't specified, a value of 1 is assumed.

When reading this property, only the widgets are returned, not the flex values.

property direction: *Direction*

The direction of the split.

property enabled: `bool`

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

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

focus()

No-op; SplitContainer cannot accept input focus.

Return type
None

Hardware

Camera

A sensor that can capture photos and/or video.

Table 3: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web	Terminal

Usage

Cameras attached to a device running an app can be accessed using the *camera* attribute. This attribute exposes an API that allows you to check if you have have permission to access the camera device; and if permission exists, capture photographs.

The Camera API is *asynchronous*. This means the methods that have long-running behavior (such as requesting permissions and taking photographs) must be `await`-ed, rather than being invoked directly. This means they must be invoked from inside an asynchronous handler:

```
import toga

class MyApp(toga.App):
    ...
    async def time_for_a_selfie(self, widget, **kwargs):
        photo = await self.camera.take_photo()
```

Most platforms will require some form of device permission to access the camera. The permission APIs are paired with the specific actions performed on those APIs - that is, to take a photo, you require *Camera.has_permission*, which you can request using *Camera.request_permission()*.

Toga will confirm whether the app has been granted permission to use the camera before invoking any camera API. If permission has not yet been granted, the platform *may* request access at the time of first camera access; however, this is not guaranteed to be the behavior on all platforms.

Notes

- Apps that use a camera must be configured to provide permission to the camera device. The permissions required are platform specific:
  - iOS: `NSCameraUsageDescription` must be defined in the app's `Info.plist` file.
  - macOS: The `com.apple.security.device.camera` entitlement must be enabled, and `NSCameraUsageDescription` must be defined in the app's `Info.plist` file.
  - Android: The `android.permission.CAMERA` permission must be declared.
- The iOS simulator implements the iOS Camera APIs, but is not able to take photographs. To test your app's Camera usage, you must use a physical iOS device.

Reference

class toga.hardware.camera.**Camera**(*app*)

Parameters
app (App)

property app: *App*

The app with which the camera is associated

property devices: list[*CameraDevice*]

The list of available camera devices.

property has_permission: bool

Does the user have permission to use camera devices?

If the platform requires the user to explicitly confirm permission, and the user has not yet given permission, this will return `False`.

request_permission()

Request sufficient permissions to capture photos.

If permission has already been granted, this will return without prompting the user.

This is an asynchronous method. If you invoke this method in synchronous context, it will start the process of requesting permissions, but will return *immediately*. The return value can be awaited in an asynchronous context, but cannot be used directly.

Returns
An asynchronous result; when awaited, returns True if the app has permission to take a photo; False otherwise.

Return type
PermissionResult

take_photo(*device=None*, *flash=FlashMode.AUTO*)

Capture a photo using one of the device's cameras.

If the platform requires permission to access the camera, and the user hasn't previously provided that permission, this will cause permission to be requested.

This is an asynchronous method. If you invoke this method in synchronous context, it will start the process of taking a photo, but will return *immediately*. The return value can be awaited in an asynchronous context, but cannot be used directly.

Parameters

- **device** (CameraDevice | *None*) – The initial camera device to use. If a device is *not* specified, a default camera will be used. Depending on the hardware available, the user may be able to change the camera used to capture the image at runtime.
- **flash** (FlashMode) – The initial flash mode to use; defaults to "auto". Depending on the hardware available, this may be modified by the user at runtime.

Returns
An asynchronous result; when awaited, returns the *toga.Image* captured by the camera, or `None` if the photo was cancelled.

Raises
PermissionError – if the app does not have permission to use the camera.

Return type
PhotoResult

class toga.hardware.camera.**CameraDevice**(*impl*)

Parameters
impl (*Any*)

property has_flash: bool

Does the device have a flash?

property id: str

A unique identifier for the device

property name: str

A human-readable name for the device

Location services

A sensor that can capture the geographical location of the device.

Table 4: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web	Terminal

Usage

The location services of a device can be accessed using the *toga.App.location* attribute. This attribute exposes an API that allows you to check if you have have permission to access location services; if permission exists, you can capture the current location of the device, and/or set a handler to be notified when position changes occur.

The Location API is *asynchronous*. This means the methods that have long-running behavior (such as requesting permissions and requesting a location) must be `await`-ed, rather than being invoked directly. This means they must be invoked from inside an asynchronous handler:

```
import toga

class MyApp(toga.App):
    ...
    async def determine_location(self, widget, **kwargs):
        location = await self.location.current_location()
```

All platforms require some form of permission to access the location service. To confirm if you have permission to use the location service while the app is running, you can call *Location.has_permission*; you can request permission using *Location.request_permission()*.

If you wish to track the location of the user while the app is in the background, you must make a separate request for background location permissions using *Location.request_background_permission()* . This request must be made *after* foreground permissions have been requested and confirmed. To confirm if you have permission to use location while the app is in the background, you can call *Location.has_background_permission*.

Toga will confirm whether the app has been granted permission to use Location services before invoking any location API. If permission has not yet been granted, or if permission has been denied by the user, a `PermissionError` will be raised.

To continuously track location, add an `on_change` handler to the location service, then call *Location.start_tracking()*. The handler will be invoked whenever a new location is obtained:

```
class MyApp(toga.App):
    ...
    async def location_update(self, location, altitude, **kwargs):
        print(f"You are now at {location}, with altitude {altitude} meters")
```

(continues on next page)

(continued from previous page)

```
    def start_location(self):
        # Install a location handler
        self.location.on_change = self.location_update
        # Start location updates. This assumes permissions have already been
        # requested and granted.
        try:
            self.location.start_tracking()
        except PermissionError:
            print("User has not permitted location tracking.")
```

If you no longer wish to receive location updates, call *Location.stop_tracking()*.

System requirements

- Using location services on Linux requires that the user has installed the system packages for GeoClue2, plus the GObject Introspection bindings for GeoClue2. The name of the system package required is distribution dependent:
  - Ubuntu and Debian: `gir1.2-geoclue-2.0`
  - Fedora: `geoclue2-libs`
  - Arch/Manjaro: `geoclue`
  - OpenSUSE Tumbleweed: `geoclue2 typelib(geoclue2)`
  - FreeBSD: `geoclue`
- The GeoClue service must be enabled for Toga GTK location services to work. Some distributions are pre-configured with GeoClue and require no action from users to enable location services. Others, for example, Ubuntu, have special controls for managing location services, which must be turned on before GeoClue will function. Refer to your distribution's documentation on GeoClue and location services for details on how to manage and configure the GeoClue service.

Notes

- Apps that use location services must be configured to provide permissions to access those services. The permissions required are platform specific:
  - iOS: `NSLocationWhenInUseUsageDescription` must be defined in the app's `Info.plist` file. If you want to track location while the app is in the background, you must also define `NSLocationAlwaysAndWhenInUseUsageDescription`, and add the `location` and `processing` values to `UIBackgroundModes`.
  - macOS: The `com.apple.security.personal-information.location` entitlement must be enabled, and `NSLocationUsageDescription` must be defined in the app's `Info.plist` file.
  - Android: At least one of the permissions `android.permission.ACCESS_FINE_LOCATION` or `android.permission.ACCESS_COARSE_LOCATION` must be declared; if only one is declared, this will impact on the precision available in location results. If you want to track location while the app is in the background, you must also define the permission `android.permission.ACCESS_BACKGROUND_LOCATION`.
- On macOS and GTK, there is no distinction between "background" permissions and "while-running" permissions for location tracking.
- On Linux, there are no reliable permission controls for non-sandboxed applications. Sandboxed applications (e.g., Flatpak apps) request location information via the XDG Portal Location API, which has coarse grained

permissions allowing users to reliably disallow location access on a per-app basis. However, Linux users should be aware of the limitations of location privacy for non-sandboxed applications. This applies to all Linux applications, not just ones using Toga GTK's Location implementation.

- On iOS, if the user has provided "allow once" permission for foreground location tracking, requests for background location permission will be rejected.
- On Android prior to API 34, altitude is reported as the distance above the WGS84 ellipsoid datum, rather than Mean Sea Level altitude.

Reference

class toga.hardware.location.**Location**(*app*)

Parameters
app (App)

property app: *App*

The app with which the location service is associated

current_location()

Obtain the user's current location using the location service.

If the app hasn't requested and received permission to use location services, a PermissionError will be raised.

This is an asynchronous method. If you call this method in a synchronous context, it will start the process of requesting the user's location, but will return *immediately*. The return value can be awaited in an asynchronous context, but cannot be used directly.

If location tracking is enabled, and an *on_change* handler is installed, requesting the current location may also cause that handler to be invoked.

Returns
An asynchronous result; when awaited, returns the current *toga.LatLng* of the device.

Raises
PermissionError – If the app has not requested and received permission to use location services.

Return type
LocationResult

property has_background_permission: bool

Does the app have permission to use location services in the background?

If the platform requires the user to explicitly confirm permission, and the user has not yet given permission, this will return False.

property has_permission: bool

Does the app have permission to use location services?

If the platform requires the user to explicitly confirm permission, and the user has not yet given permission, this will return False.

property on_change: *OnLocationChangeHandler*

The handler to invoke when an update to the user's location is available.

`request_background_permission()`

Request sufficient permissions to capture the user's location in the background.

If permission has already been granted, this will return without prompting the user.

Before requesting background permission, you must first request and receive foreground location permission using *Location.request_permission*. If you ask for background permission before receiving foreground location permission, a PermissionError will be raised.

This is an asynchronous method. If you invoke this method in synchronous context, it will start the process of requesting permissions, but will return *immediately*. The return value can be awaited in an asynchronous context, but cannot be used directly.

Returns
An asynchronous result; when awaited, returns True if the app has permission to capture the user's location while running in the background; False otherwise.

Raises
PermissionError – If the app has not already requested and received permission to use location services.

Return type
PermissionResult

`request_permission()`

Request sufficient permissions to capture the user's location.

If permission has already been granted, this will return without prompting the user.

This method will only grant permission to access location services while the app is in the foreground. If you want your application to have permission to track location while the app is in the background, you must call this method, then make an *additional* permission request for background permissions using *Location.request_background_permission()*.

This is an asynchronous method. If you invoke this method in synchronous context, it will start the process of requesting permissions, but will return *immediately*. The return value can be awaited in an asynchronous context, but cannot be used directly.

Returns
An asynchronous result; when awaited, returns True if the app has permission to capture the user's location; False otherwise.

Return type
PermissionResult

`start_tracking()`

Start monitoring the user's location for changes.

An *on_change* callback will be generated when the user's location changes.

Raises
PermissionError – If the app has not requested and received permission to use location services.

Return type
None

`stop_tracking()`

Stop monitoring the user's location.

Raises
PermissionError – If the app has not requested and received permission to use location services.

Return type
None

protocol toga.hardware.location.**OnLocationChangeHandler**

typing.Protocol.

Classes that implement this protocol must have the following methods / attributes:

__call__(*, *service*, *location*, *altitude*, ***kwargs*)

A handler that will be invoked when the user's location changes.

Parameters

- **service** (Location) – the location service that generated the update.
- **location** (LatLng) – The user's location as (latitude, longitude).
- **altitude** (*float | None*) – The user's altitude in meters above mean sea level. Returns None if the altitude could not be determined.
- **kwargs** (*Any*) – Ensures compatibility with arguments added in future versions.

Return type
object

Screen

A representation of a screen attached to a device.

Table 5: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web	Terminal

Usage

An app will always have access to at least one screen. The *toga.App.screens* attribute will return the list of all available screens; the screen at index 0 will be the "primary" screen. Screen sizes and positions are given in CSS pixels.

```
# Print the size of the primary screen.
print(my_app.screens[0].size)

# Print the identifying name of the second screen
print(my_app.screens[1].name)
```

Notes

- When using the GTK backend under Wayland, the screen at index 0 may not be the primary screen. This because the separation of concerns enforced by Wayland makes determining the primary screen unreliable.

Reference

class toga.screens.**Screen**(*_impl*)

Parameters
_impl (*Any*)

as_image(*format=Image*)

Render the current contents of the screen as an image.

Parameters
format (*type[ImageT]*) – Format to provide. Defaults to *Image*; also supports PIL.Image.Image if Pillow is installed, as well as any image types defined by installed *image format plugins*.

Returns
An image containing the screen content, in the format requested.

Return type
ImageT

property name: str

Unique name of the screen.

property origin: *Position*

The absolute coordinates of the screen's origin, in *CSS pixels*.

property size: *Size*

The size of the screen, in *CSS pixels*.

Resources

App Paths

A mechanism for obtaining platform-appropriate file system locations for an application.

Table 6: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web	Terminal

Usage

When Python code executes from the command line, the working directory is a known location - the location where the application was started. However, when executing GUI apps, the working directory varies between platforms. As a result, when specifying file paths, relative paths cannot be used, as there is no location to which they can be considered relative.

Complicating matters further, operating systems have conventions (and in some cases, hard restrictions) over where certain file types should be stored. For example, macOS provides the `~/Library/Application Support` folder; Linux encourages use of the `~/.config` folder (amongst others), and Windows provides the `AppData/Local` folder in the user's home directory. Application sandbox and security policies will sometimes prevent reading or writing files in any location other than these pre-approved locations.

To assist with finding an appropriate location to store application files, every Toga application instance has a *paths* attribute that returns an instance of *Paths*. This object provides known file system locations that are appropriate for storing files of given types, such as configuration files, log files, cache files, or user data.

Each location provided by the *Paths* object is a `pathlib.Path` that can be used to construct a full file path. If required, additional sub-folders can be created under these locations.

You should not assume that any of these paths already exist. The location is guaranteed to follow operating system conventions, but the application is responsible for ensuring the folder exists prior to writing files in these locations.

Reference

class toga.paths.**Paths**

property app: Path

The path of the folder that contains the definition of the app class.

This path should be considered read-only. You should not attempt to write files into this path.

property cache: Path

The platform-appropriate location for storing cache files associated with this app.

It should be assumed that the operating system will purge the contents of this directory without warning if it needs to recover disk space.

property config: Path

The platform-appropriate location for storing user configuration files associated with this app.

property data: Path

The platform-appropriate location for storing user data associated with this app.

property logs: Path

The platform-appropriate location for storing log files associated with this app.

property toga: Path

The path that contains the core Toga resources.

This path should be considered read-only. You should not attempt to write files into this path.

Dialogs

A short-lived window asking the user for input.

Table 7: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web	Terminal

Usage

A dialog is a short-lived window that requires the user to provide acknowledgement, respond to a question, or provide information to the application.

A dialog can be presented relative to a specific window (a window-modal dialog), and relative to the entire app (an app-modal dialog). When presented as a window-modal dialog, the user will *not* be able to interact with anything in the window until the dialog is dismissed. When presented as an app-modal dialog, the user will be restricted from interacting with the rest of the app (there are some platform-specific variations in this behavior; see the *platform notes* for details).

When a dialog is presented, the app's event loop will continue to run, and the content of the app windows will redraw if requested by the event loop. For this reason, dialogs are implemented *asynchronously* - that is, they must be `await`-ed

in the context of an `async` method. The value returned by the `await` is the response of the dialog; the return type will vary depending on the type of dialog being displayed.

To display a dialog, create an instance of the dialog type you want to display, and `await` the `dialog()` method in the context that you want to display the dialog (either *toga.Window.dialog()* or *toga.App.dialog()*). In the following example, `my_handler` is an asynchronous method defined on an *App* subclass that would be installed as an event handler (e.g., as an *on_press()* handler on a *Button*). The dialog is displayed as window-modal against the app's main window; the dialog returns `True` or `False` depending on the user's response:

```
async def my_handler(self, widget, **kwargs):
    ask_a_question = toga.QuestionDialog("Hello!", "Is this OK!")

    if await self.main_window.dialog(ask_a_question):
        print("The user said yes!")
    else:
        print("The user said no.")
```

When this handler is triggered, the dialog will be displayed, but a `print` statement will not be executed until the user's response has been received. To convert this example into an app-modal dialog, you would use `self.dialog(ask_a_question)`, instead of `self.main_window.dialog(ask_a_question)`.

If you need to display a dialog in a synchronous context (i.e., in a normal, non-`async` event handler), you must create a `asyncio.Task` for the dialog, and install a callback that will be invoked when the dialog is dismissed:

```
def my_sync_handler(self, widget, **kwargs):
    ask_a_question = toga.QuestionDialog("Hello!", "Is this OK!")

    task = asyncio.create_task(self.main_window.dialog(ask_a_question))
    task.add_done_callback(self.dialog_dismissed)
    print("Dialog has been created")

def dialog_dismissed(self, task):
    if task.result():
        print("The user said yes!")
    else:
        print("The user said no.")
```

In this example, when `my_sync_handler` is triggered, a dialog will be created, the display of that dialog will be scheduled as an asynchronous task, and a message will be logged saying the dialog has been created. When the user responds, the `dialog_dismissed` callback will be invoked, with the dialog task provided as an argument. The result of the task can then be interrogated to handle the response.

Notes

- On macOS, app-modal dialogs will *not* prevent the user from interacting with the rest of the app.

Reference

class toga.**InfoDialog**(*title*, *message*)

Bases: `Dialog`

Ask the user to acknowledge some information.

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

Returns a response of `None`.

Parameters

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

class toga.**QuestionDialog**(*title*, *message*)

Bases: Dialog

Ask the user a yes/no question.

Presents as a dialog with "Yes" and "No" buttons.

Returns a response of `True` when the "Yes" button is pressed, `False` when the "No" button is pressed.

Parameters

- **title** (*str*) – The title of the dialog window.
- **message** (*str*) – The question to be answered.

class toga.**ConfirmDialog**(*title*, *message*)

Bases: Dialog

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

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

Returns a response of `True` when the "OK" button is pressed, `False` when the "Cancel" button is pressed.

Parameters

- **title** (*str*) – The title of the dialog window.
- **message** (*str*) – A message describing the action to be confirmed.

class toga.**ErrorDialog**(*title*, *message*)

Bases: Dialog

Ask the user to acknowledge an error state.

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

Returns a response of `None`.

Parameters

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

class toga.**StackTraceDialog**(*title*, *message*, *content*, *retry=False*)

Bases: Dialog

Open a dialog to display a large block of text, such as a stack trace.

If `retry` is true, returns a response of `True` when the user selects "Retry", and `False` when they select "Quit".

If `retry` is `False`, returns a response of `None`.

Parameters

- **title** (*str*) – The title of the dialog window.
- **message** (*str*) – Contextual information about the source of the stack trace.
- **content** (*str*) – The stack trace, pre-formatted as a multi-line string.

- **retry** (*bool*) – If true, the user will be given options to "Retry" or "Quit"; if false, a single option to acknowledge the error will be displayed.

class toga.**SaveFileDialog**(*title*, *suggested_filename*, *file_types=None*)

Bases: Dialog

Prompt the user for a location to save a file.

This dialog is not currently supported on Android or iOS.

Returns a path object for the selected file location, or None if the user cancelled the save operation.

If the filename already exists, the user will be prompted to confirm they want to overwrite the existing file.

Parameters

- **title** (*str*) – The title of the dialog window
- **suggested_filename** (*Path | str*) – The initial suggested filename
- **file_types** (*list[str] | None*) – The allowed filename extensions, without leading dots. If not provided, any extension will be allowed.

class toga.**OpenFileDialog**(*title*, *initial_directory=None*, *file_types=None*, *multiple_select=False*)

Bases: Dialog

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

This dialog is not currently supported on Android or iOS.

If multiple_select is True, returns a list of Path objects.

If multiple_select is False, returns a single Path.

Returns None if the open operation is cancelled by the user.

Parameters

- **title** (*str*) – The title of the dialog window
- **initial_directory** (*Path | str | None*) – The initial folder in which to open the dialog. If None, use the default location provided by the operating system (which will often be the last used location)
- **file_types** (*list[str] | None*) – The allowed filename extensions, without leading dots. If not provided, all files will be shown.
- **multiple_select** (*bool*) – If True, the user will be able to select multiple files; if False, the selection will be restricted to a single file.

class toga.**SelectFolderDialog**(*title*, *initial_directory=None*, *multiple_select=False*)

Bases: Dialog

Prompt the user to select a directory (or directories).

This dialog is not currently supported on Android or iOS.

If multiple_select is True, returns a list of Path objects.

If multiple_select is False, returns a single Path.

Returns None if the select operation is cancelled by the user.

Parameters

- **title** (*str*) – The title of the dialog window

- **initial_directory** (*Path | str | None*) – The initial folder in which to open the dialog. If None, use the default location provided by the operating system (which will often be "last used location")
- **multiple_select** (*bool*) – If True, the user will be able to select multiple directories; if False, the selection will be restricted to a single directory. This option is not supported on WinForms.

Font

A font for displaying text.

Table 8: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web	Terminal

Usage

For most widget styling, you do not need to create instances of the Font class. Fonts are applied to widgets using style properties:

```
import toga
from toga.style.pack import pack, SERIF, BOLD

# Create a bold label in the system's serif font at default system size.
my_label = toga.Label("Hello World", style=Pack(font_family=SERIF, font_weight=BOLD))
```

Toga provides a number of *built-in system fonts*. Font sizes are specified in *CSS points*; the default size depends on the platform and the widget.

If you want to use a custom font, the font file must be provided as part of your app's resources, and registered before first use:

```
import toga

# Register the user font with name "Roboto"
toga.Font.register("Roboto", "resources/Roboto-Regular.ttf")

# Create a label with the new font.
my_label = toga.Label("Hello World", style=Pack(font_family="Roboto")
```

When registering a font, if an invalid value is provided for the style, variant or weight, NORMAL will be used.

When a font includes multiple weights, styles or variants, each one must be registered separately, even if they're stored in the same file:

```
import toga
from toga.style.pack import BOLD

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

# Register a single font file that contains both a regular and bold weight
```

(continues on next page)

(continued from previous page)

```
Font.register("Bahnschrift", "resources/Bahnschrift.ttf")
Font.register("Bahnschrift", "resources/Bahnschrift.ttf", weight=BOLD)
```

A small number of Toga APIs (e.g., *Context.write_text*) *do* require the use of `Font` instance. In these cases, you can instantiate a Font using similar properties to the ones used for widget styling:

```
import toga
from toga.style.pack import BOLD

# Obtain a 14 point Serif bold font instance
my_font = toga.Font(SERIF, 14, weight=BOLD)

# Use the font to write on a canvas.
canvas = toga.Canvas()
canvas.context.write_text("Hello", font=my_font)
```

Notes

- iOS and macOS do not support the use of variant font files (that is, fonts that contain the details of multiple weights/variants in a single file). Variant font files can be registered; however, only the "normal" variant will be used.
- Android and Windows do not support the oblique font style. If an oblique font is specified, Toga will attempt to use an italic style of the same font.
- Android and Windows do not support the small caps font variant. If a Small Caps font is specified, Toga will use the normal variant of the same font.

Reference

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

Constructs a reference to a font.

This class should be used when an API requires an explicit font reference (e.g. *Context.write_text*). In all other cases, fonts in Toga are controlled using the style properties linked below.

Parameters

- **family** (*str*) – The *font family*.
- **size** (*int* | *str*) – The *font size*.
- **weight** (*str*) – The *font weight*.
- **style** (*str*) – The *font style*.
- **variant** (*str*) – The *font variant*.

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

Registers a file-based font.

Note: This is not currently supported on macOS or iOS.

Parameters

- **family** (*str*) – The *font family*.
- **path** (*str* | *Path*) – The path to the font file. This can be an absolute path, or a path relative to the module that defines your *App* class.

- **weight** (*str*) – The *font weight*.
- **style** (*str*) – The *font style*.
- **variant** (*str*) – The *font variant*.

Return type
None

Command

A representation of app functionality that the user can invoke from menus or toolbars.

Table 9: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web	Terminal

Usage

Aside from event handlers on widgets, most GUI toolkits also provide other ways for the user to give instructions to an app. In Toga, these UI patterns are supported by the *Command* class.

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.

Adding commands

Commands are added to an app using the properties *toga.App.commands* and *toga.MainWindow.toolbar*. Toga then takes control of ensuring that the command is exposed to the user in a way that they can access. On desktop platforms, this may result in a command being added to a menu.

Commands can be organized into a *Group* of similar commands. Groups are hierarchical, so a group can contain a sub-group, which can contain a sub-group, and so on. Inside a group, commands can be organized into sections.

For example:

```
import toga

def callback(sender, **kwargs):
    print("Command activated")

stuff_group = toga.Group('Stuff', order=40)

cmd1 = toga.Command(
    callback,
    text='Example command',
    tooltip='Tells you when it has been activated',
    shortcut=toga.Key.MOD_1 + 'k',
    icon='icons/pretty.png',
    group=stuff_group,
    section=0
)
cmd2 = toga.Command(
    ...
```

(continues on next page)

(continued from previous page)

```
)
...

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 display 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.

Removing commands

Commands can be removed using set-like and dictionary-like APIs. The set-like APIs use the command instance; the dictionary-like APIs use the command ID:

```
# Remove the app using the instance
app.commands.remove(cmd_1)

# Remove a command by ID
del app.commands["Some-Command-ID"]
```

Standard commands

Each command has an *id* attribute. This is set when the command is defined; if no ID is provided, a random ID will be generated for the Command. This identifier can be used to retrieve a command from *toga.App.commands* and *toga.MainWindow.toolbar*.

These command IDs are also used to create *standard* commands. These are commands that are expected functionality in most applications, such as *ABOUT* and *EXIT*, as well as document management commands such as *NEW*, *OPEN* and *SAVE*.

These commands are automatically added to your app, depending on platform requirements and app definition. For example, mobile apps won't have an Exit command as mobile apps don't have a concept of "exiting". Document management commands will be automatically added if your app defines *document types*.

The label, shortcut, grouping and ordering of these commands is platform dependent. For example, on macOS, the *EXIT* command will be labeled "Quit My App", and have a shortcut of Command-q; on Windows, the command will be labeled "Exit", and won't have a keyboard shortcut.

Any automatically added standard commands will be installed *before* your app's *startup()* method is invoked. If you wish to remove or modify and a standard app command, you can use the standard command's ID to retrieve the command instance from *toga.App.commands*. If you wish to add or override a standard command that hasn't been installed by default (for example, to add an Open command without defining a document type), you can use the *toga.Command.standard()* method to create an instance of the standard command, and add that command to your app:

```
import toga
```

(continues on next page)

(continued from previous page)

```
class MyApp(toga.app):
    def startup(self):
        ...
        # Delete the default Preferences command
        del self.commands[toga.Command.PREFERENCES]

        # Modify the text of the "About" command
        self.commands[toga.Command.ABOUT].text = "I'm Customized!!"

        # Add an Open command
        custom_open = toga.Command.standard(
            self,
            toga.Command.OPEN,
            action=self.custom_open
        )

        self.commands.add(custom_open)
```

Reference

class toga.**Command**(*action*, *text*, ***, *shortcut=None*, *tooltip=None*, *icon=None*, *group=Group.COMMANDS*, *section=0*, *order=0*, *enabled=True*, *id=None*)

Create a new Command.

Commands may not use all the arguments - for example, on some platforms, menus will contain icons; on other platforms they won't.

Parameters

- **action** (*ActionHandler | None*) – A handler to invoke when the command is activated. If this is None, the command will be disabled.
- **text** (*str*) – A label for the command.
- **shortcut** (*str | Key | None*) – A key combination that can be used to invoke the command.
- **tooltip** (*str | None*) – A short description of what the command will do.
- **icon** (*IconContentT | None*) – The *icon content* that can be used to decorate the command if the platform requires.
- **group** (*Group*) – The group to which this command belongs.
- **section** (*int*) – The section where the command should appear within its group.
- **order** (*int*) – The position where the command should appear within its section. If multiple items have the same group, section and order, they will be sorted alphabetically by their text.
- **enabled** (*bool*) – Is the Command currently enabled?
- **id** (*str*) – A unique identifier for the command.

ABOUT: str = 'about'

An identifier for the standard "About" menu item. This command is always installed by default. Uses *toga.App.about()* as the default action.

EXIT: str = 'request_exit'

An identifier for the standard "Exit" menu item. This command may be installed by default, depending on platform requirements. Uses *toga.App.request_exit()* as the default action.

NEW: str = 'documents.new'

An identifier for the standard "New" menu item. This constant will be used for the default document type for your app; if you specify more than one document type, the command for the subsequent commands will have a colon and the first extension for that data type appended to the ID. Uses *toga.documents.DocumentSet.new()* as the default action.

OPEN: str = 'documents.request_open'

An identifier for the standard "Open" menu item. This command will be automatically installed if your app declares any document types. Uses *toga.documents.DocumentSet.request_open()* as the default action.

PREFERENCES: str = 'preferences'

An identifier for the standard "Preferences" menu item. The Preferences item is not installed by default. If you install it manually, it will attempt to use `toga.App.preferences()` as the default action; your app will need to define this method, or provide an explicit value for the action.

SAVE: str = 'documents.save'

An identifier for the standard "Save" menu item. This command will be automatically installed if your app declares any document types. Uses *toga.documents.DocumentSet.save()* as the default action.

SAVE_ALL: str = 'documents.save_all'

An identifier for the standard "Save All" menu item. This command will be automatically installed if your app declares any document types. Uses *toga.documents.DocumentSet.save_all()* as the default action.

SAVE_AS: str = 'documents.save_as'

An identifier for the standard "Save As…" menu item. This command will be automatically installed if your app declares any document types. Uses *toga.documents.DocumentSet.save_as()* as the default action.

VISIT_HOMEPAGE: str = 'visit_homepage'

An identifier for the standard "Visit Homepage" menu item. This command may be installed by default, depending on platform requirements. Uses *toga.App.visit_homepage()* as the default action.

property action: *ActionHandler* | None

The Action attached to the command.

property enabled: bool

Is the command currently enabled?

property icon: *Icon* | None

The Icon for the command.

Can be specified as any valid *icon content*.

property id: str

A unique identifier for the command.

classmethod standard(*app*, *id*, ***kwargs*)

Create an instance of a standard command for the provided app.

The default action for the command will be constructed using the value of the command's ID as an attribute of the app object. If a method or co-routine matching that name doesn't exist, a value of `None` will be used as the default action.

Parameters

- **app** (App) – The app for which the standard command will be created.
- **id** – The ID of the standard command to create.
- **kwargs** – Overrides for any default properties of the standard command. Accepts the same arguments as the *Command* constructor.

class toga.**Group**(*text*, *, *parent=None*, *section=0*, *order=0*, *id=None*)

A collection of commands to display together.

Parameters

- **text** (*str*) – A label for the group.
- **parent** (Group | *None*) – The parent of this group; use None to make a root group.
- **section** (*int*) – The section where the group should appear within its parent. A section cannot be specified unless a parent is also specified.
- **order** (*int*) – The position where the group should appear within its section. If multiple items have the same group, section and order, they will be sorted alphabetically by their text.
- **id** (*str* | *None*) – A unique identifier for the group.

APP: *Group* = <Group text='*' order=-100>

Application-level commands

COMMANDS: *Group* = <Group text='Commands' order=30>

Default group for user-provided commands

EDIT: *Group* = <Group text='Edit' order=-20>

Editing commands

FILE: *Group* = <Group text='File' order=-30>

File commands

HELP: *Group* = <Group text='Help' order=100>

Help commands

VIEW: *Group* = <Group text='View' order=-10>

Content appearance commands

WINDOW: *Group* = <Group text='Window' order=90>

Window management commands

property **id**: str

A unique identifier for the group.

is_child_of(*parent*)

Is this group a child of the provided group, directly or indirectly?

Parameters
parent (Group | *None*) – The potential parent to check

Returns
True if this group is a child of the provided parent.

Return type
bool

`is_parent_of`(*child*)

Is this group a parent of the provided group, directly or indirectly?

Parameters
child (Group | *None*) – The potential child to check

Returns
True if this group is a parent of the provided child.

Return type
bool

`property parent: Group | None`

The parent of this group; returns `None` if the group is a root group.

`property root: Group`

The root group for this group.

This will be `self` if the group *is* a root group.

`property text: str`

A text label for the group.

`class toga.command.CommandSet`(*on_change=None*, *app=None*)

Bases: MutableSet[*Command*], MutableMapping[str, *Command*]

A collection of commands.

This is used as an internal representation of Menus, Toolbars, and any other graphical manifestations of commands. You generally don't need to construct a CommandSet of your own; you should use existing app or window level CommandSet instances.

The `in` operator can be used to evaluate whether a *Command* is a member of the CommandSet, using either an instance of a Command, or the ID of a command.

Commands can be retrieved from the CommandSet using `[]` notation with the requested command's ID.

When iterated over, a CommandSet returns *Command* instances in their sort order, with *Separator* instances inserted between groups.

Parameters

- **on_change** (*CommandSetChangeHandler* | *None*) – A method that should be invoked when this CommandSet changes.
- **app** (App | *None*) – The app this CommandSet is associated with, if it is not the app's own CommandSet.

`add`(**commands*)

Add a collection of commands to the command set.

A command value of `None` will be ignored. This allows you to add standard commands to a command set without first validating that the platform provides an implementation of that command.

Parameters
commands (Command | *None*) – The commands to add to the command set.

`property app: App | None`

The app this CommandSet is associated with.

Returns None if this is the app's CommandSet.

clear()

Remove all commands from the command set.

Return type
None

discard(*command*)

Remove an element. Do not raise an exception if absent.

Parameters
command (Command)

class toga.command.**Separator**(*group=None*)

A representation of a separator between sections in a Group.

Parameters
group (Group | *None*) – The group that contains the separator.

protocol toga.command.**ActionHandler**

typing.Protocol.

Classes that implement this protocol must have the following methods / attributes:

__call__(*command*, ***kwargs*)

A handler that will be invoked when a Command is invoked.

Parameters

- **command** (Command) – The command that triggered the action.
- **kwargs** – Ensures compatibility with additional arguments introduced in future versions.

Return type
bool

Document

A representation of a file on disk that will be displayed in one or more windows.

Table 10: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web	Terminal

Usage

A common requirement for apps is to view or edit a particular type of file. In Toga, you define a *toga.Document* class to represent each type of content that your app is able to manipulate. This *Document* class is then registered with your app when the *App* instance is created.

The *toga.Document* class describes how your document can be read, displayed, and saved. It also tracks whether the document has been modified. In this example, the code declares an "Example Document" document type, which will create files with the extensions `.mydoc` and `.mydocument`; because it is listed first, the `.mydoc` extension will be the default for documents of this type. The main window for this document type contains a *MultilineTextInput*. Whenever the content of that widget changes, the document is marked as modified:

```
import toga

class ExampleDocument(toga.Document):
    description = "Example Document"
    extensions = ["mydoc", "mydocument"]

    def create(self):
        # Create the main window for the document. The window has a single widget;
        # when that widget changes, the document is modified.
        self.main_window = toga.DocumentMainWindow(
            doc=self,
            content=toga.MultilineTextInput(on_change=self.touch),
        )

    def read(self):
        # Read the content of the file represented by the document, and populate the
        # widgets in the main window with that content.
        with self.path.open() as f:
            self.main_window.content.value = f.read()

    def write(self):
        # Save the content currently displayed by the main window.
        with self.path.open("w") as f:
            f.write(self.main_window.content.value)
```

The document window uses the modification status to determine whether the window is allowed to close. If a document is modified, the user will be asked if they want to save changes to the document.

Registering document types

A document type is used by registering it with an app instance. The constructor for *toga.App* allows you to declare the collection of document types that your app supports. The first declared document type is treated as the default document type for your app; this is the type that will be connected to the keyboard shortcut of the *NEW* command.

After *startup()* returns, any filenames which were passed to the app by the operating system will be opened using the registered document types. If after this the app still has no windows, then:

- On Windows and GTK, an untitled document of the default type will be opened.
- On macOS, an Open dialog will be shown.

In the following example, the app will be able to manage documents of type `ExampleDocument` or `OtherDocument`, with `ExampleDocument` being the default content type. The app is configured *to not have a single "main" window*, so the life cycle of the app is not tied to a specific window.

```
import toga

class ExampleApp(toga.App):
    def startup(self):
        # The app does not have a single main window
        self.main_window = None

app = ExampleApp(
    "Document App",
    "com.example.documentapp",
```

(continues on next page)

(continued from previous page)

```
    document_types=[ExampleDocument, OtherDocument]
)

app.main_loop()
```

By declaring these document types, the app will automatically have file management commands (New, Open, Save, etc) added.

Reference

class toga.**Document**(*app*)

Bases: ABC

Create a new Document. Do not call this constructor directly - use *DocumentSet.new*, *DocumentSet.open* or *DocumentSet.request_open* instead.

Parameters
app (App) – The application the document is associated with.

property app: *App*

The app that this document is associated with (read-only).

abstract create()

Create the window (or windows) for the document.

This method must, at a minimum, assign the *main_window* property. It may also create additional windows or UI elements if desired.

Return type
None

description: str

A short description of the type of document (e.g., "Text document"). This is a class variable that subclasses should define.

extensions: list[str]

["doc", "txt"]). The list must have at least one extension; the first is the default extension for documents of this type. This is a class variable that subclasses should define.

focus()

Give the document focus in the app.

hide()

Hide the visual representation for this document.

Return type
None

property main_window: *Window* | None

The main window for the document.

property modified: bool

Has the document been modified?

open(*path*)

Open a file as a document.

Parameters
path (*str | Path*) – The file to open.

property path: Path

The path where the document is stored (read-only).

abstract read()

Load a representation of the document into memory from *path*, and populate the document window.

Return type
None

save(*path=None*)

Save the document as a file.

If a path is provided, the path for the document will be updated. Otherwise, the existing path will be used.

If the *write()* method has not been implemented, this method is a no-op.

Parameters
path (*str | Path | None*) – If provided, the new file name for the document.

show()

Show the visual representation for this document.

Return type
None

property title: str

The title of the document.

This will be used as the default title of a *toga.DocumentWindow* that contains the document.

touch(**args*, ***kwargs*)

Mark the document as modified.

This method accepts **args* and ***kwargs* so that it can be used as an `on_change` handler; these arguments are not used.

write()

Persist a representation of the current state of the document.

This method is a no-op by default, to allow for read-only document types.

Return type
None

class toga.documents.**DocumentSet**(*app*, *types*)

Bases: Sequence[*Document*], Mapping[Path, *Document*]

A collection of documents managed by an app.

A document is automatically added to the app when it is created, and removed when it is closed. The document collection will be stored in the order that documents were created.

Parameters

- **app** (App) – The app that this instance is bound to.
- **types** (*list[type[Document]]*) – The document types managed by this app.

new(*document_type*)

Create a new document of the given type, and show the document window.

Parameters
document_type (*type[Document]*) – The document type that has been requested.

Returns
The newly created document.

Return type
Document

open(*path*)

Open a document in the app, and show the document window.

If the provided path is already an open document, the existing representation for the document will be given focus.

Parameters
path (*Path | str*) – The path to the document to be opened.

Returns
The document that was opened.

Raises
ValueError – If the path describes a file that is of a type that doesn't match a registered document type.

Return type
Document

async request_open()

Present a dialog asking the user for a document to open, and pass the selected path to *open()*.

Returns
The document that was opened.

Raises
ValueError – If the path describes a file that is of a type that doesn't match a registered document type.

Return type
Document

async save()

Save the current content of an app.

If there isn't a current window, or current window doesn't define a `save()` method, the save request will be ignored.

async save_all()

Save the state of all content in the app.

This method will attempt to call `save()` on every window associated with the app. Any windows that do not provide a `save()` method will be ignored.

async save_as()

Save the current content of an app under a different filename.

If there isn't a current window, or the current window hasn't defined a `save_as()` method, the save-as request will be ignored.

property types: `list[type[Document]]`

The list of document types the app can manage.

The first document type in the list is the app's default document type.

Icon

A small, square image, used to provide easily identifiable visual context to a widget.

Table 11: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web	Terminal

Usage

Icons and Images are *not* the same!

Toga draws a distinction between an *Icon* and an *Image*. An `Icon` is small, square, and might vary between platforms. It is a visual element that is often used as part of an interactive element such as a button, toolbar item, or tab selector - but the Icon *itself* isn't an interactive element.

An `Image`, on the other hand, can have an arbitrary size or aspect ratio, and is *not* platform dependent - the same image will be used on *every* platform. An Image is *not* an interactive element, because there is no visual cue to the user that the image *can* be interacted with.

If you are looking for a widget that the user can click on, you're looking for a widget configured to use an Icon (probably `Button`), *not* an `on_press` handler on an `Image` or `ImageView`.

The filename specified for an icon should be specified *without* an extension; the platform will determine an appropriate extension, and may also modify the name of the icon to include a platform and/or size qualifier.

The following formats are supported (in order of preference):

- **Android** - PNG
- **iOS** - ICNS, PNG, BMP, ICO
- **macOS** - ICNS, PNG, PDF
- **GTK** - PNG, ICO, ICNS; 512, 256, 128, 72, 64, 32, and 16px variants of each icon can be provided;
- **Windows** - ICO, PNG, BMP

The first matching icon of the most specific platform, with the most specific size will be used. For example, on Windows, specifying an icon of `myicon` will cause Toga to look for (in order):

- `myicon-windows.ico`
- `myicon.ico`
- `myicon-windows.png`
- `myicon.png`
- `myicon-windows.bmp`
- `myicon.bmp`

On GTK, Toga will perform this lookup for each variant size, falling back to a name without a size specifier if a size-specific variant has not been provided. For example, when resolving the 32px variant, Toga will look for (in order):

- `myicon-linux-32.png`
- `myicon-32.png`
- `myicon-linux-32.ico`
- `myicon-32.ico`
- `myicon-linux-32.icns`
- `myicon-32.icns`
- `myicon.png`
- `myicon.ico`

Any icon that is found will be resized to the required size. Toga will generate any GTK icon variants that are not available from the highest resolution provided (e.g., if no 128px variant can be found, one will be created by scaling the highest resolution variant that *is* available).

An icon is **guaranteed** to have an implementation, regardless of the path specified. If you specify a path and no matching icon can be found, Toga will output a warning to the console, and return *DEFAULT_ICON*. The only exception to this is if an icon file is *found*, but it cannot be loaded (e.g., due to a file format or permission error). In this case, an error will be raised.

Reference

type **IconContentT**

When specifying an *Icon*, you can provide:

- a string specifying an absolute or relative path;
- an absolute or relative `pathlib.Path` object; or
- an instance of *toga.Icon*.

If a relative path is provided, it will be anchored relative to the module that defines your Toga application class.

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

Create a new icon.

Parameters

- **path** (*str | Path*) – Base filename for the icon. The path can be an absolute file system path, or a path relative to the module that defines your Toga application class. This base filename should *not* contain an extension. If an extension is specified, it will be ignored. If the icon cannot be found, the default icon will be *DEFAULT_ICON*. If an icon file is found, but it cannot be loaded (due to a file format or permission error), an exception will be raised.
- **system** (*bool*) – **For internal use only**

APP_ICON

The application icon.

The application icon will be loaded from `resources/<app name>` (where `<app name>` is the value of *toga.App.app_name*).

If this resource cannot be found, and the app has been packaged as a binary, the icon from the application binary will be used as a fallback.

Otherwise, *DEFAULT_ICON* will be used.

DEFAULT_ICON

The default icon used as a fallback - Toga's "Tiberius the yak" icon.

OPTION_CONTAINER_DEFAULT_TAB_ICON

The default icon used to decorate option container tabs.

Image

Graphical content of arbitrary size.

Table 12: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web	Terminal

Usage

Images and Icons are *not* the same!

Toga draws a distinction between an *Image* and an *Icon*. An *Image* can have an arbitrary size or aspect ratio, and is *not* platform dependent - the same image will be used on *every* platform. An Image is *not* an interactive element, because there is no visual cue to the user that the image *can* be interacted with.

An *Icon*, on the other hand, is small, square, and might vary between platforms. It is a visual element that is often used as part of an interactive element such as a button, a toolbar item, or a tab selector - but the Icon *itself* isn't an interactive element.

If you are looking for a widget that the user can click on, you're looking for a widget configured to use an Icon (probably *Button*), *not* an `on_press` handler on an *Image* or *ImageView*.

An image can be constructed from a *wide range of sources*:

```
from pathlib import Path
import toga

# Load an image in the same folder as the file that declares the App class
my_image = toga.Image("brutus.png")

# Load an image at an absolute path
my_image = toga.Image(Path.home() / "path/to/brutus.png")

# Create an image from raw data
with (Path.home() / "path/to/brutus.png").open("rb") as f:
    my_image = toga.Image(data=f.read())

# Create an image from a PIL image (if PIL is installed)
import PIL.Image
my_pil_image = PIL.Image.new("L", (30, 30))
my_toga_image = toga.Image(my_pil_image)
```

You can also tell Toga how to convert from (and to) other classes that represent images via *image format plugins*.

Notes

- PNG and JPEG formats are guaranteed to be supported. Other formats are available on some platforms:
  - GTK: BMP
  - macOS: GIF, BMP, TIFF
  - Windows: GIF, BMP, TIFF
- The native platform representations for images are:
  - Android: `android.graphics.Bitmap`
  - GTK: `GdkPixbuf.Pixbuf`
  - iOS: `UIImage`
  - macOS: `NSImage`
  - Windows: `System.Drawing.Image`
- If you subclass *`Image`*, you can supply that subclass as the requested format to any `as_format()` method in Toga, provided that your subclass has a constructor signature compatible with the base *`Image`* class.

Reference

type **ImageContentT**

When specifying content for an *`Image`*, you can provide:

- a string specifying an absolute or relative path to a file in a *known image format*;
- an absolute or relative `Path` object describing a file in a *known image format*;
- a "blob of bytes" data type (`bytes`, `bytearray`, or `memoryview`) containing raw image data in a *known image format*;
- an instance of *`toga.Image`*;
- if Pillow is installed, an instance of `PIL.Image.Image`;
- an image of a class registered via an *image format plugin* (or a subclass of such a class); or
- an instance of the *native platform image representation*.

If a relative path is provided, it will be anchored relative to the module that defines your Toga application class.

class toga.**Image**(*src=NOT_PROVIDED*, ***, *path=NOT_PROVIDED*, *data=NOT_PROVIDED*)

Create a new image.

Parameters

- **src** (*`ImageContentT`*) – The source from which to load the image. Can be any valid *`image content`* type.
- **path** (*`object`*) – **DEPRECATED** - Use `src`.
- **data** (*`object`*) – **DEPRECATED** - Use `src`.

Raises

- **`FileNotFoundError`** – If a path is provided, but that path does not exist.
- **`ValueError`** – If the source cannot be loaded as an image.

as_format(*format*)

Return the image, converted to the image format specified.

Parameters
format (*type[ImageT]*) – Format to provide. Defaults to *Image*; also supports PIL.Image.Image if Pillow is installed, as well as any image types defined by installed *image format plugins*.

Returns
The image in the requested format

Raises
TypeError – If the format supplied is not recognized.

Return type
ImageT

property data: bytes

The raw data for the image, in PNG format.

property height: int

The height of the image, in pixels.

property path: Path | None

The path from which the image was opened, if any (or None).

save(*path*)

Save image to given path.

The file format of the saved image will be determined by the extension of the filename provided (e.g `path/to/mypicture.png` will save a PNG file).

Parameters
path (*str | Path*) – Path to save the image to.

Return type
None

property size: tuple[int, int]

The size of the image, as a (width, height) tuple.

property width: int

The width of the image, in pixels.

Source

A base class for data source implementations.

Usage

Data sources are abstractions that allow you to define the data being managed by your application independent of the GUI representation of that data. For details on the use of data sources, see the *topic guide*.

Source isn't useful on its own; it is a base class for data source implementations. It is used by ListSource, TreeSource and ValueSource, but it can also be used by custom data source implementations. It provides an implementation of the notification API that data sources must provide.

Reference

class toga.sources.**Listener**(**args*, ***kwargs*)

Bases: Protocol

The protocol that must be implemented by objects that will act as a listener on a data source.

change(*item*)

A change has occurred in an item.

Parameters
item (*object*) – The data object that has changed.

Return type
object

clear()

All items have been removed from the data source.

Return type
object

insert(*index*, *item*)

An item has been added to the data source.

Parameters

- **index** (*int*) – The 0-index position in the data.
- **item** (*object*) – The data object that was added.

Return type
object

remove(*index*, *item*)

An item has been removed from the data source.

Parameters

- **index** (*int*) – The 0-index position in the data.
- **item** (*object*) – The data object that was added.

Return type
object

class toga.sources.**Source**

A base class for data sources, providing an implementation of data notifications.

add_listener(*listener*)

Add a new listener to this data source.

If the listener is already registered on this data source, the request to add is ignored.

Parameters
listener (Listener) – The listener to add

Return type
None

property listeners: list[*Listener*]

The listeners of this data source.

Returns

A list of objects that are listening to this data source.

notify(*notification*, ***kwargs*)

Notify all listeners an event has occurred.

Parameters

- **notification** (*str*) – The notification to emit.
- **kwargs** (*object*) – The data associated with the notification.

Return type

None

remove_listener(*listener*)

Remove a listener from this data source.

Parameters

listener (Listener) – The listener to remove.

Return type

None

ListSource

A data source describing an ordered list of data.

Usage

Data sources are abstractions that allow you to define the data being managed by your application independent of the GUI representation of that data. For details on the use of data sources, see the *topic guide*.

ListSource is an implementation of an ordered list of data. When a ListSource is created, it is given a list of `accessors` - these are the attributes that all items managed by the ListSource will have. The API provided by ListSource is `list`-like; the operations you'd expect on a normal Python list, such as `insert`, `remove`, `index`, and indexing with `[]`, are also possible on a ListSource:

```
from toga.sources import ListSource

source = ListSource(
    accessors=["name", "weight"],
    data=[
        {"name": "Platypus", "weight": 2.4},
        {"name": "Numbat", "weight": 0.597},
        {"name": "Thylacine", "weight": 30.0},
    ]
)

# Get the first item in the source
item = source[0]
print(f"Animal's name is {item.name}")

# Find an item with a name of "Thylacine"
item = source.find({"name": "Thylacine"})
```

(continues on next page)

(continued from previous page)

```
# Remove that item from the data
source.remove(item)

# Insert a new item at the start of the data
source.insert(0, {"name": "Bettong", "weight": 1.2})
```

The ListSource manages a list of *Row* objects. Each Row has all the attributes described by the source's `accessors`. A Row object will be constructed for each item that is added to the ListSource, and each item can be:

- A dictionary, with the accessors mapping to the keys in the dictionary.
- Any other iterable object (except for a string), with the accessors being mapped onto the items in the iterable in order of definition.
- Any other object, which will be mapped onto the *first* accessor.

Although Toga provides ListSource, you are not required to create one directly. A ListSource will be transparently constructed if you provide an iterable object to a GUI widget that displays list-like data (i.e., *toga.Table*, *toga.Selection*, or *toga.DetailedList*).

Custom List Sources

For more complex applications, you can replace ListSource with a *custom data source* class. Such a class must:

- Inherit from *Source*
- Provide the same methods as *ListSource*
- Return items whose attributes match the accessors expected by the widget
- Generate a `change` notification when any of those attributes change
- Generate `insert`, `remove` and `clear` notifications when items are added or removed

Reference

class toga.sources.**Row**(***data*)

Bases: Generic[T]

Create a new Row object.

The keyword arguments specified in the constructor will be converted into attributes on the new Row object.

When any public attributes of the Row are modified (i.e., any attribute whose name doesn't start with _), the source to which the row belongs will be notified.

Parameters
data (*T*)

__setattr__(*attr*, *value*)

Set an attribute on the Row object, notifying the source of the change.

Parameters

- **attr** (*str*) – The attribute to change.
- **value** (*T*) – The new attribute value.

Return type
None

class toga.sources.**ListSource**(*accessors*, *data=None*)

Bases: *Source*

A data source to store an ordered list of multiple data values.

Parameters

- **accessors** (*Iterable[str]*) – A list of attribute names for accessing the value in each column of the row.
- **data** (*Iterable | None*) – The initial list of items in the source. Items are converted as shown *above*.

__delitem__(*index*)

Deletes the item at position `index` of the list.

Parameters
index (*int*)

Return type
None

__getitem__(*index*)

Returns the item at position `index` of the list.

Parameters
index (*int*)

Return type
Row

__len__()

Returns the number of items in the list.

Return type
int

__setitem__(*index*, *value*)

Set the value of a specific item in the data source.

Parameters

- **index** (*int*) – The item to change
- **value** (*object*) – The data for the updated item. This data will be converted into a Row object.

Return type
None

append(*data*)

Insert a row at the end of the data source.

Parameters
data (*object*) – The data to append to the ListSource. This data will be converted into a Row object.

Returns
The newly constructed Row object.

Return type
Row

`clear`()

Clear all data from the data source.

Return type
None

`find`(*data*, *start=None*)

Find the first item in the data that matches all the provided attributes.

This is a value based search, rather than an instance search. If two Row instances have the same values, the first instance that matches will be returned. To search for a second instance, provide the first found instance as the `start` argument. To search for a specific Row instance, use the *index()*.

Parameters

- **data** (*object*) – The data to search for. Only the values specified in data will be used as matching criteria; if the row contains additional data attributes, they won't be considered as part of the match.
- **start** (Row | *None*) – The instance from which to start the search. Defaults to None, indicating that the first match should be returned.

Returns
The matching Row object

Raises
ValueError – If no match is found.

Return type
Row

`index`(*row*)

The index of a specific row in the data source.

This search uses Row instances, and searches for an *instance* match. If two Row instances have the same values, only the Row that is the same Python instance will match. To search for values based on equality, use *find()*.

Parameters
row (Row) – The row to find in the data source.

Returns
The index of the row in the data source.

Raises
ValueError – If the row cannot be found in the data source.

Return type
int

`insert`(*index*, *data*)

Insert a row into the data source at a specific index.

Parameters

- **index** (*int*) – The index at which to insert the item.
- **data** (*object*) – The data to insert into the ListSource. This data will be converted into a Row object.

Returns
The newly constructed Row object.

Return type
Row

remove(*row*)

Remove a row from the data source.

Parameters
row (Row) – The row to remove from the data source.

Return type
None

TreeSource

A data source describing an ordered hierarchical tree of values.

Usage

Data sources are abstractions that allow you to define the data being managed by your application independent of the GUI representation of that data. For details on the use of data sources, see the *topic guide*.

TreeSource is an implementation of an ordered hierarchical tree of values. When a TreeSource is created, it is given a list of `accessors` - these are the attributes that all items managed by the TreeSource will have. The API provided by TreeSource is `list`-like; the operations you'd expect on a normal Python list, such as `insert`, `remove`, `index`, and indexing with `[]`, are also possible on a TreeSource. These methods are available on the TreeSource itself to manipulate root nodes, and also on each node within the tree.

```
from toga.sources import TreeSource

source = TreeSource(
    accessors=["name", "height"],
    data={
        "Animals": [
            ({"name": "Numbat", "height": 0.15}, None),
            ({"name": "Thylacine", "height": 0.6}, None),
        ],
        "Plants": [
            ({"name": "Woollybush", "height": 2.4}, None),
            ({"name": "Boronia", "height": 0.9}, None),
        ],
    }
)

# Get the Animal group in the source.
# The Animal group won't have a "height" attribute.
group = source[0]
print(f"Group's name is {group.name}")

# Get the second item in the animal group
animal = group[1]
print(f"Animals's name is {animal.name}; it is {animal.height}m tall.")

# Find an animal with a name of "Thylacine"
row = source.find(parent=source[0], {"name": "Thylacine"})
```

(continues on next page)

(continued from previous page)

```
# Remove that row from the data. Even though "Thylacine" isn't a root node,
# remove will find it and remove it from the list of animals.
source.remove(row)

# Insert a new item at the start of the list of animals.
group.insert(0, {"name": "Bettong", "height": 0.35})

# Insert a new root item in the middle of the list of root nodes
source.insert(1, {"name": "Minerals"})
```

The TreeSource manages a tree of *Node* objects. Each Node has all the attributes described by the source's `accessors`. A Node object will be constructed for each item that is added to the TreeSource.

Each Node object in the TreeSource can have children; those children can in turn have their own children. A child that *cannot* have children is called a *leaf Node*. Whether a child *can* have children is independent of whether it *does* have children - it is possible for a Node to have no children and *not* be a leaf node. This is analogous to files and directories on a file system: a file is a leaf Node, as it cannot have children; a directory *can* contain files and other directories in it, but it can also be empty. An empty directory would *not* be a leaf Node.

When creating a single Node for a TreeSource (e.g., when inserting a new item), the data for the Node can be specified as:

- A dictionary, with the accessors mapping to the keys in the dictionary
- Any iterable object (except for a string), with the accessors being mapped onto the items in the iterable in order of definition.
- Any other object, which will be mapped onto the *first* accessor.

When constructing an entire TreeSource, the data can be specified as:

- A dictionary. The keys of the dictionary will be converted into Nodes, and used as parents; the values of the dictionary will become the children of their corresponding parent.
- Any other iterable object (except a string). Each value in the iterable will be treated as a 2-item tuple, with the first item being data for the parent Node, and the second item being the child data.
- Any other object will be converted into a single node with no children.

When specifying children, a value of `None` for the children will result in the creation of a leaf node. Any other value will be processed recursively - so, a child specifier can itself be a dictionary, an iterable of 2-tuples, or data for a single child, and so on.

Although Toga provides TreeSource, you are not required to create one directly. A TreeSource will be transparently constructed for you if you provide one of the items listed above (e.g. `list`, `dict`, etc) to a GUI widget that displays tree-like data (i.e., *toga.Tree*).

Custom TreeSources

For more complex applications, you can replace TreeSource with a *custom data source* class. Such a class must:

- Inherit from *Source*
- Provide the same methods as *TreeSource*
- Return items whose attributes match the accessors expected by the widget
- Generate a `change` notification when any of those attributes change
- Generate `insert`, `remove` and `clear` notifications when nodes are added or removed

Reference

class toga.sources.**Node**(***data*)

Bases: *Row*[T]

Create a new Node object.

The keyword arguments specified in the constructor will be converted into attributes on the new object.

When initially constructed, the Node will be a leaf node (i.e., no children, and marked unable to have children).

When any public attributes of the node are modified (i.e., any attribute whose name doesn't start with _), the source to which the node belongs will be notified.

Parameters
data (*T*)

__delitem__(*index*)

Parameters
index (*int*)

Return type
None

__getitem__(*index*)

Parameters
index (*int*)

Return type
Node[*T*]

__len__()

Return type
int

__setitem__(*index*, *data*)

Set the value of a specific child in the Node.

Parameters

- **index** (*int*) – The index of the child to change
- **data** (*object*) – The data for the updated child. This data will be converted into a Node object.

Return type
None

append(*data*, *children=None*)

Append a node to the end of the list of children of this node.

Parameters

- **data** (*object*) – The data to append as a child of this node. This data will be converted into a Node object.
- **children** (*object*) – The data for the children of the new child node.

Returns
The new added child Node object.

Return type
Node[*T*]

`can_have_children`()

Can the node have children?

A value of `True` does not necessarily mean the node *has* any children, only that the node is *allowed* to have children. The value of `len()` for the node indicates the number of actual children.

Return type
bool

`find`(*data*, *start=None*)

Find the first item in the child nodes of this node that matches all the provided attributes.

This is a value based search, rather than an instance search. If two Node instances have the same values, the first instance that matches will be returned. To search for a second instance, provide the first found instance as the `start` argument. To search for a specific Node instance, use the *index()*.

Parameters

- **data** (*object*) – The data to search for. Only the values specified in data will be used as matching criteria; if the node contains additional data attributes, they won't be considered as part of the match.
- **start** (Node*[T] | None*) – The instance from which to start the search. Defaults to `None`, indicating that the first match should be returned.

Returns
The matching Node object.

Raises

- **ValueError** – If no match is found.
- **ValueError** – If the node is a leaf node.

Return type
Node[*T*]

`index`(*child*)

The index of a specific node in children of this node.

This search uses Node instances, and searches for an *instance* match. If two Node instances have the same values, only the Node that is the same Python instance will match. To search for values based on equality, use *find()*.

Parameters
child (Node*[T]*) – The node to find in the children of this node.

Returns
The index of the node in the children of this node.

Raises
ValueError – If the node cannot be found in children of this node.

Return type
int

`insert`(*index*, *data*, *children=None*)

Insert a node as a child of this node a specific index.

Parameters

- **index** (*int*) – The index at which to insert the new child.
- **data** (*object*) – The data to insert into the Node as a child. This data will be converted into a Node object.
- **children** (*object*) – The data for the children of the new child node.

Returns
The new added child Node object.

Return type
Node[*T*]

remove(*child*)

Remove a child node from this node.

Parameters
child (*Node[T]*) – The child node to remove from this node.

Return type
None

class toga.sources.**TreeSource**(*accessors*, *data=None*)

Bases: *Source*

Parameters

- **accessors** (*Iterable[str]*)
- **data** (*object | None*)

__delitem__(*index*)

Parameters
index (*int*)

Return type
None

__getitem__(*index*)

Parameters
index (*int*)

Return type
Node

__len__()

Return type
int

__setitem__(*index*, *data*)

Set the value of a specific root item in the data source.

Parameters

- **index** (*int*) – The root item to change
- **data** (*object*) – The data for the updated item. This data will be converted into a Node object.

Return type
None

append(*data*, *children=None*)

Append a root node at the end of the list of children of this source.

If the node is a leaf node, it will be converted into a non-leaf node.

Parameters

- **data** (*object*) – The data to append onto the list of children of the given parent. This data will be converted into a Node object.
- **children** (*object | None*) – The data for the children to insert into the TreeSource.

Returns
The newly constructed Node object.

Raises
ValueError – If the provided parent is not part of this TreeSource.

Return type
Node

clear()

Clear all data from the data source.

Return type
None

find(*data*, *start=None*)

Find the first item in the child nodes of the given node that matches all the provided attributes.

This is a value based search, rather than an instance search. If two Node instances have the same values, the first instance that matches will be returned. To search for a second instance, provide the first found instance as the `start` argument. To search for a specific Node instance, use the *index()*.

Parameters

- **data** (*object*) – The data to search for. Only the values specified in data will be used as matching criteria; if the node contains additional data attributes, they won't be considered as part of the match.
- **start** (Node | *None*) – The instance from which to start the search. Defaults to None, indicating that the first match should be returned.

Returns
The matching Node object.

Raises

- **ValueError** – If no match is found.
- **ValueError** – If the provided parent is not part of this TreeSource.

Return type
Node

index(*node*)

The index of a specific root node in the data source.

This search uses Node instances, and searches for an *instance* match. If two Node instances have the same values, only the Node that is the same Python instance will match. To search for values based on equality, use *find()*.

Parameters
node (Node) – The node to find in the data source.

Returns
The index of the node in the child list it is a part of.

Raises
ValueError – If the node cannot be found in the data source.

Return type
int

insert(*index*, *data*, *children=None*)

Insert a root node into the data source at a specific index.

If the node is a leaf node, it will be converted into a non-leaf node.

Parameters

- **index** (*int*) – The index into the list of children at which to insert the item.
- **data** (*object*) – The data to insert into the TreeSource. This data will be converted into a Node object.
- **children** (*object*) – The data for the children to insert into the TreeSource.

Returns
The newly constructed Node object.

Raises
ValueError – If the provided parent is not part of this TreeSource.

Return type
Node

remove(*node*)

Remove a node from the data source.

This will also remove the node if it is a descendant of a root node.

Parameters
node (Node) – The node to remove from the data source.

Return type
None

ValueSource

A data source describing a single value.

Usage

Data sources are abstractions that allow you to define the data being managed by your application independent of the GUI representation of that data. For details on the use of data sources, see the *topic guide*.

ValueSource is an wrapper around a single atomic value.

```
from toga.sources import ValueSource

source = ValueSource(42)

# Get the value managed by the source
print(f"Meaning of life, the universe, and everything is {source.value}")
```

Custom ValueSources

A custom ValueSource has 3 requirements:

- It must have an `accessor` attribute that describes the name of the attribute that stores the data for the source.
- It must have an attribute matching the name of the accessor that can be used to set and retrieve and the value.
- When any change is made to the value, a `change` notification will be emitted.

Reference

class toga.sources.**ValueSource**(*value=None*, *accessor='value'*)

Bases: *Source*

Parameters

- **value** (*object*)
- **accessor** (*str*)

Status Icons

Icons that appear in the system tray for representing app status while the app isn't visible.

macOS

Linux

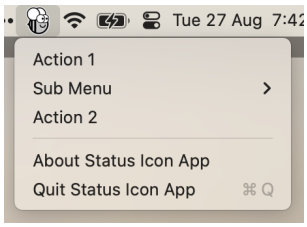
Windows

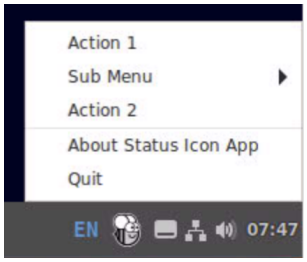
Android ×

iOS ×

Web ×

Textual ×

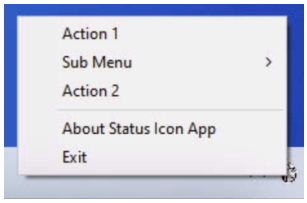




Not supported

Not supported

Not supported



Not supported

Usage

Although the usual user interface for an app is a window, some apps will augment - or even replace - a window-base interface with an icon in the system tray or status bar provided by the operating system. This is especially common for apps that primarily run in the background.

Toga supports two types of status icons - simple status icons, and menu status icons.

Simple status icons

A simple status icon is a bare icon in the status bar. You can set and change the icon as required to reflect changes in application state; by default, the status icon will use the app's icon. The text associated with the icon will be used as a tooltip; again, the app's formal name will be used as default text. The icon can respond to mouse clicks by defining an `on_press` handler.

To define a simple status icon, declare an instance of *toga.SimpleStatusIcon*, and add it to your app's *status_icons* set:

```
import toga

# Define a status icon that uses default values for icon and tooltip,
# and doesn't respond to mouse clicks.
status_icon_1 = toga.SimpleStatusIcon()

# Define a second status icon that provides explicit values for the id, icon and
# tooltip, and responds to mouse clicks.
def my_handler(widget, **kwargs):
    print("Second status icon pressed!")

status_icon_2 = toga.SimpleStatusIcon(
    id="second",
    text="Hello!",
    icon="icons/red",
    on_press=my_handler
)

# Add both status icons to the app
app.status_icons.add(status_icon_1, status_icon_2)
```

Once a status icon has been added to the app, it can be retrieved by ID or by index; and it can be removed from the app:

```
# Change the icon of the first status icon, retrieved by index:
app.status_icons[0].icon = "icons/green"

# Change the icon of the second status icon, retrieved by id:
```

(continues on next page)

(continued from previous page)

```
app.status_icons["second"].icon = "icons/blue"

# Remove the first status icon from the app:
app.status_icons.remove(status_icon_1)
```

Menu status icons

A menu-based status icon is defined by adding a *toga.MenuStatusIcon* instance. A *toga.MenuStatusIcon* behaves almost the same as *SimpleStatusIcon*, except that it *cannot* have an `on_click` handler - but it *can* be used to register Commands that will be displayed in a menu when the icon is clicked.

The *MenuStatusIcon* is a *Group* for command sorting purposes. To put a command in a menu associated with a *MenuStatusIcon*, set the `group` associated with that command, then add the command to the *CommandSet* associated with status icons. The following example will create a *MenuStatusIcon* that has a single top-level menu item, plus a sub-menu that itself has a single menu item:

```
# Create a MenuStatusIcon
status_icon = toga.MenuStatusIcon(icon="icons/blue")

# Create some commands that are associated with the menu status icon's group.
def callback(sender, **kwargs):
    print("Command activated")

cmd1 = toga.Command(
    callback,
    text='Example command',
    group=status_icon,
)

# Create a sub-group of the status icon. This will appear as a submenu.
stuff_group = toga.Group('Stuff', parent=status_icon)

cmd2 = toga.Command(
    callback,
    text='Stuff sub-command',
    group=stuff_group
)

# Add the status icon to the app
app.status_icons.add(status_icon)

# Add the commands to the status icons command set
app.status_icons.commands.add(cmd1, cmd2)
```

If you add at least one *MenuStatusIcon* instance to your app, Toga will add some standard commands to the app's status icon command set. These items will appear at the end of the first *MenuStatusIcon*'s menu. To remove these items, clear the app's status icon command set before adding your own commands.

If you add a command to the app's status icon command set that *doesn't* belong to a *MenuStatusIcon* group, or that belongs to a *MenuStatusIcon* group that hasn't been registered with the app as a status icon, a `ValueError` will be raised. An error will also be raised if you *remove* a status icon while there status icon commands referencing that command.

Notes

- Status icons on GTK are implemented using the XApp library. This requires that the user has installed the system packages for `libxapp`, plus the GObject Introspection bindings for that library. The name of the system package required is distribution dependent:
  - Ubuntu: `gir1.2-xapp-1.0`
  - Fedora: `xapps`
- The GNOME desktop environment does not provide built-in support for status icons. This is an explicit design decision on their part, and they advise against using status icons as part of your app design. However, there are GNOME shell extensions that can add support for status icons. Other GTK-based desktop environments (such as Cinnamon) *do* support status icons.

Reference

class toga.statusicons.**StatusIcon**(*icon=None*)

An abstract base class for all status icons.

Parameters
icon (*IconContentT | None*)

property icon: *Icon* | None

The Icon to display in the status bar.

When setting the icon, you can provide either an *Icon* instance, or a path that will be passed to the `Icon` constructor.

abstract property id: str

A unique identifier for the status icon.

abstract property text: str

A text label for the status icon.

class toga.**SimpleStatusIcon**(*id=None*, *icon=None*, *text=None*, *on_press=None*)

Bases: *StatusIcon*

An button in a status bar or system tray.

When pressed, the `on_press` handler will be activated.

Parameters

- **id** (*str | None*) – An identifier for the status icon.
- **icon** (*IconContentT | None*) – The icon, or icon resource, that will be displayed in the status bar or system tray.
- **text** (*str | None*) – A text label for the status icon. Defaults to the formal name of the app.
- **on_press** (`toga.widgets.button.OnPressHandler` | *None*) – The handler to invoke when the status icon is pressed.

property id: str

A unique identifier for the status icon.

property on_press: *toga.widgets.button.OnPressHandler*

The handler to invoke when the status icon is pressed.

property text: str

A text label for the status icon.

class toga.MenuStatusIcon(*id=None*, *icon=None*, *text=None*)

Bases: *Group*, *StatusIcon*

An item in a status bar or system tray that displays a menu when pressed.

A MenuStatusIcon can be used as a *Group* when defining *toga.Command* instances.

Parameters

- **id** (*str | None*) – An identifier for the status icon.
- **icon** (*IconContentT | None*) – The icon, or icon resource, that will be displayed in the status bar or system tray.
- **text** (*str | None*) – A text label for the status icon. Defaults to the formal name of the app.

class toga.statusicons.StatusIconSet

Bases: Sequence[*StatusIcon*], Mapping[str, *StatusIcon*]

An ordered collection of status icons.

The items in the set can be retrieved by instance, or by ID. When iterated, the items are returned in the order they were added.

add(**status_icons*)

Add one or more icons to the set.

Parameters

status_icons (StatusIcon) – The icon (or icons) to add to the set.

clear()

Remove all the icons from the set.

Raises

ValueError – If the status icon commands include any commands that reference an icon that has been removed.

remove(*status_icon*)

Remove a single icon from the set.

Parameters

status_icon (StatusIcon) – The status icon instance to remove.

Raises

ValueError – If the status icon commands include any commands that reference the icon that has been removed.

Validators

A mechanism for validating that input meets a given set of criteria.

Usage

A validator is a callable that accepts a string as input, and returns None on success, or a string on failure. If a string is returned, that string will be used as an error message. For example, the following example will validate that the user's input starts with the text "Hello":

```
def must_say_hello(value):
    if value.lower().startswith("hello"):
        return None
    return "Why didn't you say hello?"
```

Toga provides built-in validators for a range of common validation types, as well as some base classes that can be used as a starting point for custom validators.

A list of validators can then be provided to any widget that performs validation, such as the *TextInput* widget. In the following example, a `TextInput` will validate that the user has entered text that starts with "hello", and has provided at least 10 characters of input:

```
import toga
from toga.validators import MinLength

widget = toga.TextInput(validators=[
    must_say_hello,
    MinLength(10)
])
```

Whenever the input changes, all validators will be evaluated in the order they have been specified. The first validator to fail will put the widget into an "error" state, and the error message returned by that validator will be displayed to the user.

Reference

class toga.validators.**BooleanValidator**(*error_message*, *allow_empty=True*)

An abstract base class for defining a simple validator.

Subclasses should implement the `is_valid()` method

Parameters

- **error_message** (*str*) – The error to display to the user when the input isn't valid.
- **allow_empty** (*bool*) – Optional; Is no input considered valid? Defaults to `True`

abstract is_valid(*input_string*)

Is the input string valid?

Parameters
input_string (*str*) – The string to validate.

Returns
`True` if the input is valid.

Return type
bool

class toga.validators.**Contains**(*substring*, *count=None*, *error_message=None*, *allow_empty=True*)

Bases: *CountValidator*

A validator confirming that the string contains one or more copies of a substring.

Parameters

- **substring** (*str*) – The substring that must exist in the input.
- **count** (*int | None*) – Optional; The exact number of matches that are expected.

- **error_message** (*str | None*) – Optional; the error message to display when the input doesn't contain the substring (or the requested count of substrings).
- **allow_empty** (*bool*) – Optional; Is no input considered valid? Defaults to `True`

count(*input_string*)

Count the instances of content of interest in the input string.

Parameters
input_string (*str*) – The string to inspect for content of interest.

Returns
The number of instances of content that the validator is looking for.

Return type
int

class toga.validators.**ContainsDigit**(*count=None*, *error_message=None*, *allow_empty=True*)

Bases: *CountValidator*

A validator confirming that the string contains digits.

Parameters

- **count** (*int | None*) – Optional; if provided, the exact count of digits that must exist. If not provided, the existence of any digit will make the string valid.
- **error_message** (*str | None*) – Optional; the error message to display when the input doesn't contain digits (or the requested count of digits).
- **allow_empty** (*bool*) – Optional; Is no input considered valid? Defaults to `True`

count(*input_string*)

Count the instances of content of interest in the input string.

Parameters
input_string (*str*) – The string to inspect for content of interest.

Returns
The number of instances of content that the validator is looking for.

Return type
int

class toga.validators.**ContainsLowercase**(*count=None*, *error_message=None*, *allow_empty=True*)

Bases: *CountValidator*

A validator confirming that the string contains lower case letters.

Parameters

- **count** (*int | None*) – Optional; if provided, the exact count of lower case letters that must exist. If not provided, the existence of any lower case letter will make the string valid.
- **error_message** (*str | None*) – Optional; the error message to display when the input doesn't contain lower case letters (or the requested count of lower case letters).
- **allow_empty** (*bool*) – Optional; Is no input considered valid? Defaults to `True`

count(*input_string*)

Count the instances of content of interest in the input string.

Parameters
input_string (*str*) – The string to inspect for content of interest.

Returns
The number of instances of content that the validator is looking for.

Return type
int

class toga.validators.**ContainsSpecial**(*count=None*, *error_message=None*, *allow_empty=True*)

Bases: *CountValidator*

A validator confirming that the string contains "special" (i.e., non-alphanumeric) characters.

Parameters

- **count** (*int | None*) – Optional; if provided, the exact count of special characters that must exist. If not provided, the existence of any special character will make the string valid.
- **error_message** (*str | None*) – Optional; the error message to display when the input doesn't contain special characters (or the requested count of special characters).
- **allow_empty** (*bool*) – Optional; Is no input considered valid? Defaults to `True`

count(*input_string*)

Count the instances of content of interest in the input string.

Parameters
input_string (*str*) – The string to inspect for content of interest.

Returns
The number of instances of content that the validator is looking for.

Return type
int

class toga.validators.**ContainsUppercase**(*count=None*, *error_message=None*, *allow_empty=True*)

Bases: *CountValidator*

A validator confirming that the string contains upper case letters.

Parameters

- **count** (*int | None*) – Optional; if provided, the exact count of upper case letters that must exist. If not provided, the existence of any upper case letter will make the string valid.
- **error_message** (*str | None*) – Optional; the error message to display when the input doesn't contain upper case letters (or the requested count of upper case letters).
- **allow_empty** (*bool*) – Optional; Is no input considered valid? Defaults to `True`

count(*input_string*)

Count the instances of content of interest in the input string.

Parameters
input_string (*str*) – The string to inspect for content of interest.

Returns
The number of instances of content that the validator is looking for.

Return type
int

class toga.validators.**CountValidator**(*count*, *expected_existence_message*, *expected_non_existence_message*, *expected_count_message*, *allow_empty=True*)

Bases: object

An abstract base class for validators that are based on counting instances of some content in the overall content.

Subclasses should implement the `count()` method to identify the content of interest.

Parameters

- **count** (*int | None*) – Optional; The expected count.
- **expected_existence_message** (*str*) – The error message to show if matches are expected, but were not found.
- **expected_non_existence_message** (*str*) – The error message to show if matches were not expected, but were found.
- **expected_count_message** (*str*) – The error message to show if matches were expected, but a different number were found.
- **allow_empty** (*bool*) – Optional; Is no input considered valid? Defaults to `True`

abstract count(*input_string*)

Count the instances of content of interest in the input string.

Parameters
input_string (*str*) – The string to inspect for content of interest.

Returns
The number of instances of content that the validator is looking for.

Return type
int

class toga.validators.**Email**(*error_message=None*, *allow_empty=True*)

Bases: *MatchRegex*

A validator confirming that the string is an email address.

> **Note**
>
> It's impossible to do *true* RFC-compliant email validation with a regex. This validator does a "best effort" validation. It will inevitably allow some email addresses that aren't *technically* valid. However, it shouldn't *exclude* any valid email addresses.

Parameters

- **error_message** (*str | None*) – Optional; the error message to display when the input isn't a number.
- **allow_empty** (*bool*) – Optional; Is no input considered valid? Defaults to `True`

EMAIL_REGEX =
"^[a-zA-Z0-9.!#$%&'*+/=?^_`{|}~-]+@[a-zA-Z0-9-]+(?:\\.[a-zA-Z0-9-]+)*$"

class toga.validators.**EndsWith**(*substring*, *error_message=None*, *allow_empty=True*)

Bases: *BooleanValidator*

A validator confirming that the string ends with a given substring.

Parameters

- **substring** (*str*) – The substring that the input must end with.
- **error_message** (*str | None*) – Optional; the error message to display when the string doesn't end with the given substring.
- **allow_empty** (*bool*) – Optional; Is no input considered valid? Defaults to `True`

is_valid(*input_string*)

Is the input string valid?

Parameters
input_string (*str*) – The string to validate.

Returns
`True` if the input is valid.

Return type
bool

class toga.validators.**Integer**(*error_message=None*, *allow_empty=True*)

Bases: *BooleanValidator*

A validator confirming that the string is an integer.

Parameters

- **error_message** (*str | None*) – Optional; the error message to display when the input isn't an integer.
- **allow_empty** (*bool*) – Optional; Is no input considered valid? Defaults to `True`

is_valid(*input_string*)

Is the input string valid?

Parameters
input_string (*str*) – The string to validate.

Returns
`True` if the input is valid.

Return type
bool

class toga.validators.**LengthBetween**(*min_length*, *max_length*, *error_message=None*, *allow_empty=True*)

Bases: *BooleanValidator*

A validator confirming that the length of input falls in a given range.

Parameters

- **min_length** (*int | None*) – The minimum length of the string (inclusive).
- **max_length** (*int | None*) – The maximum length of the string (inclusive).
- **error_message** (*str | None*) – Optional; the error message to display when the length isn't in the given range.
- **allow_empty** (*bool*) – Optional; Is no input considered valid? Defaults to `True`

is_valid(*input_string*)

Is the input string valid?

Parameters
input_string (*str*) – The string to validate.

Returns
True if the input is valid.

Return type
bool

class toga.validators.**MatchRegex**(*regex_string*, *error_message=None*, *allow_empty=True*)

Bases: *BooleanValidator*

A validator confirming that the string matches a given regular expression.

Parameters

- **regex_string** (*str*) – A regular expression that the input must match.
- **error_message** (*str | None*) – Optional; the error message to display when the input doesn't match the provided regular expression.
- **allow_empty** (*bool*) – Optional; Is no input considered valid? Defaults to True

is_valid(*input_string*)

Is the input string valid?

Parameters
input_string (*str*) – The string to validate.

Returns
True if the input is valid.

Return type
bool

class toga.validators.**MaxLength**(*length*, *error_message=None*)

Bases: *LengthBetween*

A validator confirming that the length of input does not exceed a maximum size.

Parameters

- **length** (*int*) – The maximum length of the string (inclusive).
- **error_message** (*str | None*) – Optional; the error message to display when the string is too long.

class toga.validators.**MinLength**(*length*, *error_message=None*, *allow_empty=True*)

Bases: *LengthBetween*

A validator confirming that the length of input exceeds a minimum size.

Parameters

- **length** (*int*) – The minimum length of the string (inclusive).
- **error_message** (*str | None*) – Optional; the error message to display when the string isn't long enough.
- **allow_empty** (*bool*) – Optional; Is no input considered valid? Defaults to True

class toga.validators.**NotContains**(*substring*, *error_message=None*)

Bases: *Contains*

A validator confirming that the string does not contain a substring.

Parameters

- **substring** (*str*) – A substring that must not exist in the input.

- **error_message** (*str | None*) – Optional; the error message to display when the input contains the provided substring.

class toga.validators.**Number**(*error_message=None*, *allow_empty=True*)

Bases: *BooleanValidator*

A validator confirming that the string is a number.

Parameters

- **error_message** (*str | None*) – Optional; the error message to display when the input isn't a number.
- **allow_empty** (*bool*) – Optional; Is no input considered valid? Defaults to `True`

is_valid(*input_string*)

Is the input string valid?

Parameters
input_string (*str*) – The string to validate.

Returns
`True` if the input is valid.

Return type
bool

class toga.validators.**StartsWith**(*substring*, *error_message=None*, *allow_empty=True*)

Bases: *BooleanValidator*

A validator confirming that the input starts with a given substring.

Parameters

- **substring** (*str*) – The substring that the input must start with.
- **error_message** (*str | None*) – Optional; the error message to display when the string doesn't start with the given substring.
- **allow_empty** (*bool*) – Optional; Is no input considered valid? Defaults to `True`

is_valid(*input_string*)

Is the input string valid?

Parameters
input_string (*str*) – The string to validate.

Returns
`True` if the input is valid.

Return type
bool

Widgets

ActivityIndicator

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

macOS

Linux

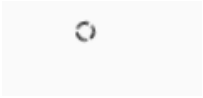
Windows ×

Android ×

iOS

Web

Textual ×



Not supported

Not supported

Screenshot not available

Not supported

Usage

```
import toga

indicator = toga.ActivityIndicator()

# Start the animation
indicator.start()

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

Notes

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

Reference

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

Bases: *Widget*

Create a new ActivityIndicator widget.

Parameters

- **id** (*str | None*) – The ID for the widget.
- **style** (*StyleT | None*) – A style object. If no style is provided, a default style will be applied to the widget.
- **running** (*bool*) – Describes whether the indicator is running at the time it is created.
- **kwargs** – Initial style properties.

property enabled: Literal[True]

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

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

focus()

No-op; ActivityIndicator cannot accept input focus.

Return type
None

property is_running: bool

Determine if the activity indicator is currently running.

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

True if this activity indicator is running; False otherwise.

start()

Start the activity indicator.

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

Return type
None

stop()

Stop the activity indicator.

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

Return type
None

Button

A button that can be pressed or clicked.

macOS

Linux

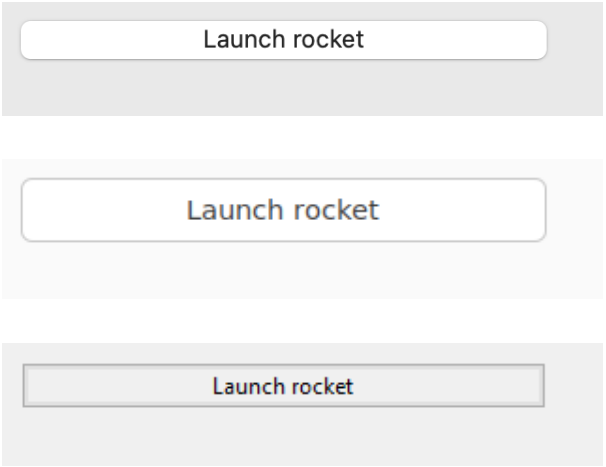
Windows

Android

iOS

Web

Textual



Screenshot not available

Screenshot not available

Usage

A button has a text label, or an icon (but not both). If an icon is specified, it will be resized to a size appropriate for the platform. A handler can be associated with button press events.

```
import toga

def my_callback(button):
    # handle event
    pass

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

icon_button = toga.Button(icon=toga.Icon("resources/my_icon"), on_press=my_callback)
```

Notes

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

Reference

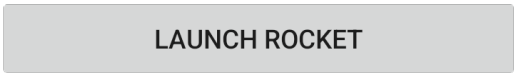
class toga.**Button**(*text=None*, *icon=None*, *id=None*, *style=None*, *on_press=None*, *enabled=True*, ***kwargs*)

Bases: *Widget*

Create a new button widget.

Parameters

- **text** (*str* | *None*) – The text to display on the button.



- **icon** (*IconContentT | None*) – The icon to display on the button. Can be specified as any valid *icon content*.
- **id** (*str | None*) – The ID for the widget.
- **style** (*StyleT | None*) – A style object. If no style is provided, a default style will be applied to the widget.
- **on_press** (*toga.widgets.button.OnPressHandler | None*) – A handler that will be invoked when the button is pressed.
- **enabled** (*bool*) – Is the button enabled (i.e., can it be pressed?). Optional; by default, buttons are created in an enabled state.
- **kwargs** – Initial style properties.

property icon: *Icon* | None

The icon displayed on the button.

Can be specified as any valid *icon content*.

If the button is currently displaying text, and an icon is assigned, the text will be replaced by the new icon.

If None is assigned as an icon, the button will become a text button with an empty label.

Returns None if the button is currently displaying text.

property on_press: *OnPressHandler*

The handler to invoke when the button is pressed.

property text: str

The text displayed on the button.

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

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

If the button is currently displaying an icon, and text is assigned, the icon will be replaced by the new text.

If the button is currently displaying an icon, the empty string will be returned.

protocol toga.widgets.button.**OnPressHandler**

typing.Protocol.

Classes that implement this protocol must have the following methods / attributes:

__call__(*widget*, ***kwargs*)

A handler that will be invoked when a button is pressed.

Parameters

- **widget** (Button) – The button that was pressed.
- **kwargs** (*Any*) – Ensures compatibility with arguments added in future versions.

Return type
object

Canvas

A drawing area for 2D vector graphics.

macOS

Linux

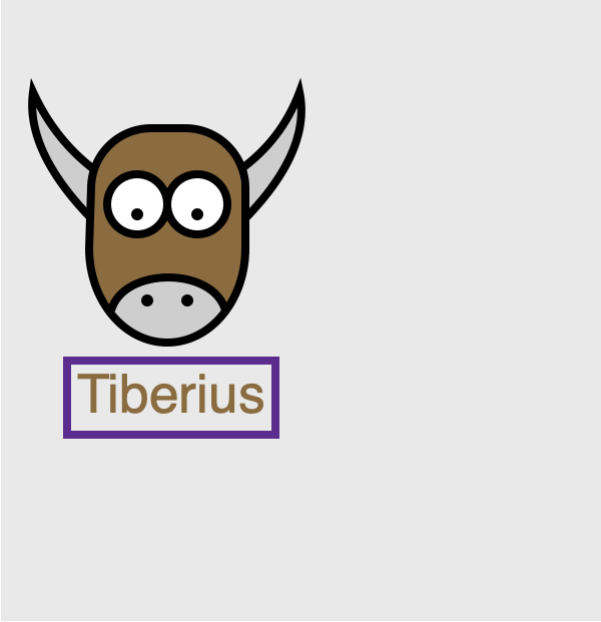
Windows

Android

iOS

Web ×

Textual ×



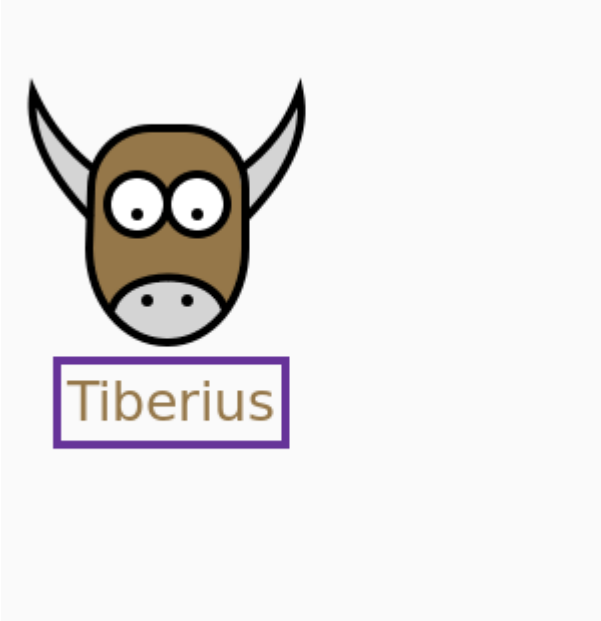
Not supported

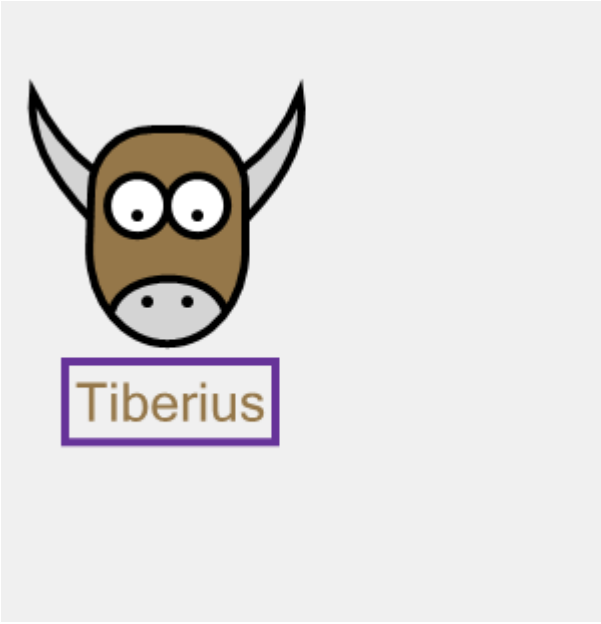
Not supported

Usage

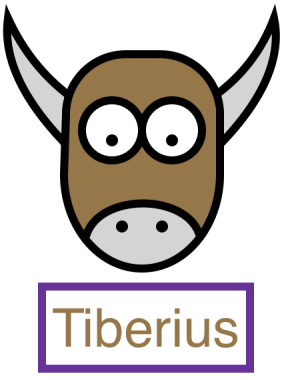
Canvas is a 2D vector graphics drawing area, whose API broadly follows the HTML5 Canvas API. The Canvas provides a drawing Context; drawing instructions are then added to that context by calling methods on the context. All positions and sizes are measured in *CSS pixels*.

For example, the following code will draw an orange horizontal line:


Tiberius


Tiberius


Tiberius


Tiberius

```
import toga
canvas = toga.Canvas()
context = canvas.context

context.begin_path()
context.move_to(20, 20)
context.line_to(160, 20)
context.stroke(color="orange")
```

Toga adds an additional layer of convenience to the base HTML5 API by providing context managers for operations that have a natural open/close life cycle. For example, the previous example could be replaced with:

```
import toga
canvas = toga.Canvas()

with canvas.context.Stroke(20, 20, color="orange") as stroke:
    stroke.line_to(160, 20)
```

Any argument provided to a drawing operation or context object becomes a property of that object. Those properties can be modified after creation, after which you should invoke *Canvas.redraw* to request a redraw of the canvas.

Drawing operations can also be added to or removed from a context using the `list` operations `append`, `insert`, `remove` and `clear`. In this case, *Canvas.redraw* will be called automatically.

For example, if you were drawing a bar chart where the height of the bars changed over time, you don't need to completely reset the canvas and redraw all the objects; you can use the same objects, only modifying the height of existing bars, or adding and removing bars as required.

In this example, we create 2 filled drawing objects, then manipulate those objects, requesting a redraw after each set of changes.

```
import toga

canvas = toga.Canvas()
with canvas.context.Fill(color="red") as fill:
    circle = fill.arc(x=50, y=50, radius=15)
    rect = fill.rect(x=50, y=50, width=15, height=15)

# We can then change the properties of the drawing objects.
# Make the circle smaller, and move it closer to the origin.
circle.x = 25
circle.y = 25
circle.radius = 5
canvas.redraw()

# Change the fill color to blue
fill.color = "blue"
canvas.redraw()

# Remove the rectangle from the canvas
fill.remove(rect)
```

For detailed tutorials on the use of Canvas drawing instructions, see the MDN documentation for the HTML5 Canvas API. Other than the change in naming conventions for methods - the HTML5 API uses `lowerCamelCase`, whereas the Toga API uses `snake_case` - both APIs are very similar.

Notes

- The Canvas API allows the use of handlers to respond to mouse/pointer events. These event handlers differentiate between "primary" and "alternate" modes of activation. When a mouse is in use, alternate activation will usually be interpreted as a "right click"; however, platforms may not implement an alternate activation mode. To ensure cross-platform compatibility, applications should not use the alternate press handlers as the sole mechanism for accessing critical functionality.

Reference

class toga.**Canvas**(*id=None*, *style=None*, *on_resize=None*, *on_press=None*, *on_activate=None*, *on_release=None*, *on_drag=None*, *on_alt_press=None*, *on_alt_release=None*, *on_alt_drag=None*, ***kwargs*)

Bases: *Widget*

Create a new Canvas widget.

Inherits from *toga.Widget*.

Parameters

- **id** (*str* | *None*) – The ID for the widget.
- **style** (*StyleT* | *None*) – A style object. If no style is provided, a default style will be applied to the widget.
- **on_resize** (OnResizeHandler | *None*) – Initial *on_resize* handler.
- **on_press** (OnTouchHandler | *None*) – Initial *on_press* handler.
- **on_activate** (OnTouchHandler | *None*) – Initial *on_activate* handler.
- **on_release** (OnTouchHandler | *None*) – Initial *on_release* handler.
- **on_drag** (OnTouchHandler | *None*) – Initial *on_drag* handler.
- **on_alt_press** (OnTouchHandler | *None*) – Initial *on_alt_press* handler.
- **on_alt_release** (OnTouchHandler | *None*) – Initial *on_alt_release* handler.
- **on_alt_drag** (OnTouchHandler | *None*) – Initial *on_alt_drag* handler.
- **kwargs** – Initial style properties.

ClosedPath(*x=None*, *y=None*)

Construct and yield a new *ClosedPathContext* context in the root context of this canvas.

Parameters

- **x** (*float* | *None*) – The x coordinate of the path's starting point.
- **y** (*float* | *None*) – The y coordinate of the path's starting point.

Yields
The new *ClosedPathContext* context object.

Return type
ContextManager[ClosedPathContext]

Context()

Construct and yield a new sub-*Context* within the root context of this Canvas.

Yields
The new *Context* object.

Return type
ContextManager[Context]

Fill(*x=None*, *y=None*, *color=BLACK*, *fill_rule=FillRule.NONZERO*)

Construct and yield a new *FillContext* in the root context of this canvas.

A drawing operator that fills the path constructed in the context according to the current fill rule.

If both an x and y coordinate is provided, the drawing context will begin with a `move_to` operation in that context.

Parameters

- **x** (*float | None*) – The x coordinate of the path's starting point.
- **y** (*float | None*) – The y coordinate of the path's starting point.
- **fill_rule** (FillRule) – *nonzero* is the non-zero winding rule; *evenodd* is the even-odd winding rule.
- **color** (*Color | str | None*) – The fill color.

Yields
The new *FillContext* context object.

Return type
ContextManager[FillContext]

Stroke(*x=None*, *y=None*, *color=BLACK*, *line_width=2.0*, *line_dash=None*)

Construct and yield a new *StrokeContext* in the root context of this canvas.

If both an x and y coordinate is provided, the drawing context will begin with a `move_to` operation in that context.

Parameters

- **x** (*float | None*) – The x coordinate of the path's starting point.
- **y** (*float | None*) – The y coordinate of the path's starting point.
- **color** (*Color | str | None*) – The color for the stroke.
- **line_width** (*float*) – The width of the stroke.
- **line_dash** (*list[float] | None*) – The dash pattern to follow when drawing the line. Default is a solid line.

Yields
The new *StrokeContext* context object.

Return type
ContextManager[StrokeContext]

as_image(*format=toga.Image*)

Render the canvas as an image.

Parameters
format (*type[ImageT]*) – Format to provide. Defaults to *Image*; also supports `PIL.Image.Image` if Pillow is installed, as well as any image types defined by installed *image format plugins*

Returns
The canvas as an image of the specified type.

Return type
ImageT

property context: *Context*

The root context for the canvas.

property enabled: Literal[True]

Is the widget currently enabled? i.e., can the user interact with the widget? ScrollContainer widgets cannot be disabled; this property will always return True; any attempt to modify it will be ignored.

focus()

No-op; ScrollContainer cannot accept input focus.

Return type
None

measure_text(*text*, *font=None*)

Measure the size at which *write_text()* would render some text.

Parameters

- **text** (*str*) – The text to measure. Newlines will cause line breaks, but long lines will not be wrapped.
- **font** (Font | *None*) – The font in which to draw the text. The default is the system font.

Returns
A tuple of `(width, height)`.

Return type
tuple[float, float]

property on_activate: OnTouchHandler

The handler invoked when the canvas is pressed in a way indicating the pressed object should be activated. When a mouse is in use, this will usually be a double click with the primary (usually the left) mouse button.

This event is not supported on Android or iOS.

property on_alt_drag: OnTouchHandler

The handler to invoke when the location of an alternate press changes.

This event is not supported on Android or iOS.

property on_alt_press: OnTouchHandler

The handler to invoke when the canvas is pressed in an alternate manner. This will usually correspond to a secondary (usually the right) mouse button press.

This event is not supported on Android or iOS.

property on_alt_release: OnTouchHandler

The handler to invoke when an alternate press is released.

This event is not supported on Android or iOS.

property on_drag: OnTouchHandler

The handler invoked when the location of a press changes.

property on_press: OnTouchHandler

The handler invoked when the canvas is pressed. When a mouse is being used, this press will be with the primary (usually the left) mouse button.

property on_release: OnTouchHandler

The handler invoked when a press on the canvas ends.

property on_resize: OnResizeHandler

The handler to invoke when the canvas is resized.

redraw()

Redraw the Canvas.

The Canvas will be automatically redrawn after adding or removing a drawing object, or when the Canvas resizes. However, when you modify the properties of a drawing object, you must call `redraw` manually.

Return type
None

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

Bases: *DrawingObject*

A drawing context for a canvas.

You should not create a *Context* directly; instead, you should use the *Context()* method on an existing context, or use *Canvas.context* to access the root context of the canvas.

Parameters

- **canvas** (toga.Canvas)
- **kwargs** (*Any*)

ClosedPath(*x=None*, *y=None*)

Construct and yield a new `ClosedPath` sub-context that will draw a closed path, starting from an origin.

This is a context manager; it creates a new path and moves to the start coordinate; when the context exits, the path is closed. For fine-grained control of a path, you can use *begin_path()* and *close_path()*.

Parameters

- **x** (*float | None*) – The x coordinate of the path's starting point.
- **y** (*float | None*) – The y coordinate of the path's starting point.

Yields
The *ClosedPathContext* context object.

Return type
Iterator[ClosedPathContext]

Context()

Construct and yield a new sub-*Context* within this context.

Yields
The new *Context* object.

Return type
Iterator[Context]

Fill(*x=None*, *y=None*, *color=BLACK*, *fill_rule=FillRule.NONZERO*)

Construct and yield a new `Fill` sub-context within this context.

This is a context manager; it creates a new path, and moves to the start coordinate; when the context exits, the path is closed with a fill. For fine-grained control of a path, you can use *begin_path*, *move_to*, *close_path* and *fill*.

If both an x and y coordinate is provided, the drawing context will begin with a `move_to` operation in that context.

Parameters

- **x** (*float | None*) – The x coordinate of the path's starting point.
- **y** (*float | None*) – The y coordinate of the path's starting point.
- **fill_rule** (FillRule) – *nonzero* is the non-zero winding rule; *evenodd* is the even-odd winding rule.
- **color** (*str*) – The fill color.

Yields
The new *FillContext* context object.

Return type
Iterator[FillContext]

Stroke(*x=None*, *y=None*, *color=BLACK*, *line_width=2.0*, *line_dash=None*)

Construct and yield a new `Stroke` sub-context within this context.

This is a context manager; it creates a new path, and moves to the start coordinate; when the context exits, the path is closed with a stroke. For fine-grained control of a path, you can use *begin_path*, *move_to*, *close_path* and *stroke*.

If both an x and y coordinate is provided, the drawing context will begin with a `move_to` operation in that context.

Parameters

- **x** (*float | None*) – The x coordinate of the path's starting point.
- **y** (*float | None*) – The y coordinate of the path's starting point.
- **color** (*str*) – The color for the stroke.
- **line_width** (*float*) – The width of the stroke.
- **line_dash** (*list[float] | None*) – The dash pattern to follow when drawing the line. Default is a solid line.

Yields
The new *StrokeContext* context object.

Return type
Iterator[StrokeContext]

__getitem__(*index*)

Returns the drawing object at the given index.

Parameters
index (*int*)

Return type
DrawingObject

__len__()

Returns the number of drawing objects that are in this context.

Return type
int

append(*obj*)

Append a drawing object to the context.

Parameters
obj (DrawingObject) – The drawing object to add to the context.

Return type
None

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

Draw a circular arc in the canvas context.

A full circle will be drawn by default; an arc can be drawn by specifying a start and end angle.

Parameters

- **x** (*float*) – The X coordinate of the circle's center.
- **y** (*float*) – The Y coordinate of the circle's center.
- **startangle** (*float*) – The start angle in radians, measured clockwise from the positive X axis.
- **endangle** (*float*) – The end angle in radians, measured clockwise from the positive X axis.
- **anticlockwise** (*bool*) – If true, the arc is swept anticlockwise. The default is clockwise.
- **radius** (*float*)

Returns
The Arc *DrawingObject* for the operation.

Return type
Arc

begin_path()

Start a new path in the canvas context.

Returns
The BeginPath *DrawingObject* for the operation.

Return type
BeginPath

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

Draw a Bézier curve in the canvas context.

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

Parameters

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

Returns
The `BezierCurveTo` *DrawingObject* for the operation.

Return type
BezierCurveTo

property canvas: *Canvas*

The canvas that is associated with this drawing context.

clear()

Remove all drawing objects from the context.

Return type
None

close_path()

Close the current path in the canvas context.

This closes the current path as a simple drawing operation. It should be paired with a *begin_path()* operation; or, to complete a complete closed path, use the *ClosedPath()* context manager.

Returns
The `ClosePath` *DrawingObject* for the operation.

Return type
ClosePath

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

Draw an elliptical arc in the canvas context.

A full ellipse will be drawn by default; an arc can be drawn by specifying a start and end angle.

Parameters

- **x** (*float*) – The X coordinate of the ellipse's center.
- **y** (*float*) – The Y coordinate of the ellipse's center.
- **radiusx** (*float*) – The ellipse's horizontal axis radius.
- **radiusy** (*float*) – The ellipse's vertical axis radius.
- **rotation** (*float*) – The ellipse's rotation in radians, measured clockwise around its center.
- **startangle** (*float*) – The start angle in radians, measured clockwise from the positive X axis.
- **endangle** (*float*) – The end angle in radians, measured clockwise from the positive X axis.
- **anticlockwise** (*bool*) – If true, the arc is swept anticlockwise. The default is clockwise.

Returns
The `Ellipse` *DrawingObject* for the operation.

Return type
Ellipse

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

Fill the current path.

The fill can use either the Non-Zero or Even-Odd winding rule for filling paths.

Parameters

- **fill_rule** (FillRule) – *nonzero* is the non-zero winding rule; *evenodd* is the even-odd winding rule.
- **color** (*str*) – The fill color.

Returns
The Fill *DrawingObject* for the operation.

Return type
Fill

insert(*index*, *obj*)

Insert a drawing object into the context at a specific index.

Parameters

- **index** (*int*) – The index at which the drawing object should be inserted.
- **obj** (DrawingObject) – The drawing object to add to the context.

Return type
None

line_to(*x*, *y*)

Draw a line segment ending at a point in the canvas context.

Parameters

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

Returns
The LineTo *DrawingObject* for the operation.

Return type
LineTo

move_to(*x*, *y*)

Moves the current point of the canvas context without drawing.

Parameters

- **x** (*float*) – The x coordinate of the new current point.
- **y** (*float*) – The y coordinate of the new current point.

Returns
The MoveTo *DrawingObject* for the operation.

Return type
MoveTo

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

Draw a quadratic curve in the canvas context.

A quadratic curve requires two points. The first point is a control point; the second is the end point. The starting point of the curve is the last point in the current path, which can be changed using moveTo() before creating the quadratic curve.

Parameters

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

- **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
The QuadraticCurveTo *DrawingObject* for the operation.

Return type
QuadraticCurveTo

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

Draw a rectangle in the canvas context.

Parameters

- **x** (*float*) – The horizontal coordinate of the left of the rectangle.
- **y** (*float*) – The vertical coordinate of the top of the rectangle.
- **width** (*float*) – The width of the rectangle.
- **height** (*float*) – The height of the rectangle.

Returns
The Rect *DrawingObject* for the operation.

Return type
Rect

redraw()

Calls *Canvas.redraw* on the parent Canvas.

Return type
None

remove(*obj*)

Remove a drawing object from the context.

Parameters
obj (DrawingObject) – The drawing object to remove.

Return type
None

reset_transform()

Reset all transformations in the canvas context.

Returns
A ResetTransform *DrawingObject*.

Return type
ResetTransform

rotate(*radians*)

Add a rotation to the canvas context.

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

Returns
The Rotate *DrawingObject* for the transformation.

Return type
Rotate

scale(*sx*, *sy*)

Add a scaling transformation to the canvas context.

Parameters

- **sx** (*float*) – Scale factor for the X dimension. A negative value flips the image horizontally.
- **sy** (*float*) – Scale factor for the Y dimension. A negative value flips the image vertically.

Returns
The `Scale` *DrawingObject* for the transformation.

Return type
Scale

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

Draw the current path as a stroke.

Parameters

- **color** (*str*) – The color for the stroke.
- **line_width** (*float*) – The width of the stroke.
- **line_dash** (*list[float] | None*) – The dash pattern to follow when drawing the line, expressed as alternating lengths of dashes and spaces. The default is a solid line.

Returns
The `Stroke` *DrawingObject* for the operation.

Return type
Stroke

translate(*tx*, *ty*)

Add a translation to the canvas context.

Parameters

- **tx** (*float*) – Translation for the X dimension.
- **ty** (*float*) – Translation for the Y dimension.

Returns
The `Translate` *DrawingObject* for the transformation.

Return type
Translate

write_text(*text*, *x=0.0*, *y=0.0*, *font=None*, *baseline=Baseline.ALPHABETIC*)

Write text at a given position in the canvas context.

Drawing text is effectively a series of path operations, so the text will have the color and fill properties of the canvas context.

Parameters

- **text** (*str*) – The text to draw. Newlines will cause line breaks, but long lines will not be wrapped.
- **x** (*float*) – The X coordinate of the text's left edge.
- **y** (*float*) – The Y coordinate: its meaning depends on `baseline`.
- **font** (`Font` | *None*) – The font in which to draw the text. The default is the system font.

- **baseline** (Baseline) – Alignment of text relative to the Y coordinate.

Returns
The WriteText *DrawingObject* for the operation.

Return type
WriteText

class toga.widgets.canvas.**DrawingObject**

Bases: ABC

A drawing operation in a *Context*.

Every context drawing method creates a DrawingObject, adds it to the context, and returns it. Each argument passed to the method becomes a property of the DrawingObject, which can be modified as shown in the *Usage* section.

DrawingObjects can also be created manually, then added to a context using the *append()* or *insert()* methods. Their constructors take the same arguments as the corresponding *Context* method, and their classes have the same names, but capitalized:

- *toga.widgets.canvas.Arc*
- *toga.widgets.canvas.BeginPath*
- *toga.widgets.canvas.BezierCurveTo*
- *toga.widgets.canvas.ClosePath*
- *toga.widgets.canvas.Ellipse*
- *toga.widgets.canvas.Fill*
- *toga.widgets.canvas.LineTo*
- *toga.widgets.canvas.MoveTo*
- *toga.widgets.canvas.QuadraticCurveTo*
- *toga.widgets.canvas.Rect*
- *toga.widgets.canvas.ResetTransform*
- *toga.widgets.canvas.Rotate*
- *toga.widgets.canvas.Scale*
- *toga.widgets.canvas.Stroke*
- *toga.widgets.canvas.Translate*
- *toga.widgets.canvas.WriteText*

class toga.widgets.canvas.**ClosedPathContext**(*canvas*, *x=None*, *y=None*)

Bases: *Context*

A drawing context that will build a closed path, starting from an origin.

This is a context manager; it creates a new path and moves to the start coordinate; when the context exits, the path is closed. For fine-grained control of a path, you can use *begin_path*, *move_to* and *close_path*.

If both an x and y coordinate is provided, the drawing context will begin with a move_to operation in that context.

You should not create a *ClosedPathContext* context directly; instead, you should use the *ClosedPath()* method on an existing context.

Parameters

- **canvas** (toga.Canvas)
- **x** (*float | None*)
- **y** (*float | None*)

class toga.widgets.canvas.**FillContext**(*canvas*, *x=None*, *y=None*, *color=BLACK*, *fill_rule=FillRule.NONZERO*)

Bases: *ClosedPathContext*

A drawing context that will apply a fill to any paths all objects in the context.

The fill can use either the Non-Zero or Even-Odd winding rule for filling paths.

This is a context manager; it creates a new path, and moves to the start coordinate; when the context exits, the path is closed with a fill. For fine-grained control of a path, you can use *begin_path*, *move_to*, *close_path* and *fill*.

If both an x and y coordinate is provided, the drawing context will begin with a `move_to` operation in that context.

You should not create a *FillContext* context directly; instead, you should use the *Fill()* method on an existing context.

Parameters

- **canvas** (toga.Canvas)
- **x** (*float | None*)
- **y** (*float | None*)
- **color** (*str*)
- **fill_rule** (FillRule)

property color: Color

The fill color.

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

Bases: *ClosedPathContext*

Construct a drawing context that will draw a stroke on all paths defined within the context.

This is a context manager; it creates a new path, and moves to the start coordinate; when the context exits, the path is drawn with the stroke. For fine-grained control of a path, you can use *begin_path*, *move_to*, *close_path* and *stroke*.

If both an x and y coordinate is provided, the drawing context will begin with a `move_to` operation in that context.

You should not create a *StrokeContext* context directly; instead, you should use the *Stroke()* method on an existing context.

Parameters

- **canvas** (toga.Canvas)
- **x** (*float | None*)
- **y** (*float | None*)
- **color** (*str | None*)
- **line_width** (*float*)
- **line_dash** (*list[float] | None*)

property color: Color

The color of the stroke.

protocol toga.widgets.canvas.**OnTouchHandler**

typing.Protocol.

Classes that implement this protocol must have the following methods / attributes:

__call__(*widget*, *x*, *y*, ***kwargs*)

A handler that will be invoked when a *Canvas* is touched with a finger or mouse.

Parameters

- **widget** (Canvas) – The canvas that was touched.
- **x** (*int*) – X coordinate, relative to the left edge of the canvas.
- **y** (*int*) – Y coordinate, relative to the top edge of the canvas.
- **kwargs** (*Any*) – Ensures compatibility with arguments added in future versions.

Return type
object

protocol toga.widgets.canvas.**OnResizeHandler**

typing.Protocol.

Classes that implement this protocol must have the following methods / attributes:

__call__(*widget*, *width*, *height*, ***kwargs*)

A handler that will be invoked when a *Canvas* is resized.

Parameters

- **widget** (Canvas) – The canvas that was resized.
- **width** (*int*) – The new width.
- **height** (*int*) – The new height.
- **kwargs** (*Any*) – Ensures compatibility with arguments added in future versions.

Return type
object

DateInput

A widget to select a calendar date.

macOS ×

Linux ×

Windows

Android

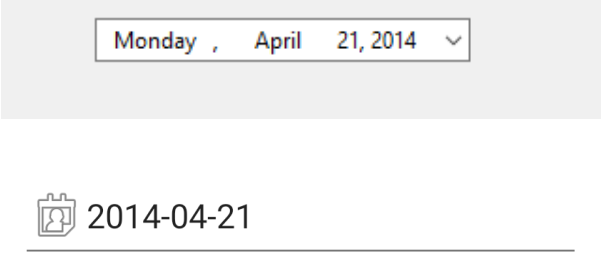
iOS ×

Web ×

Textual ×

Not supported

Not supported



Not supported

Not supported

Not supported

Usage

```
import toga

current_date = toga.DateInput()
```

Notes

- This widget supports years from 1800 to 8999 inclusive.
- Properties that return `datetime.date` objects can also accept:
  - `datetime.datetime`: The date portion will be extracted.
  - `str`: Will be parsed as an ISO8601 format date string (e.g., "2023-12-25").

Reference

class toga.**DateInput**(*id=None*, *style=None*, *value=None*, *min=None*, *max=None*, *on_change=None*, ***kwargs*)

Bases: *Widget*

Create a new DateInput widget.

Parameters

- **id** (*str | None*) – The ID for the widget.
- **style** (*StyleT | None*) – A style object. If no style is provided, a default style will be applied to the widget.
- **value** (*datetime.date | None*) – The initial date to display. If not specified, the current date will be used.
- **min** (*datetime.date | None*) – The earliest date (inclusive) that can be selected.
- **max** (*datetime.date | None*) – The latest date (inclusive) that can be selected.
- **on_change** (*toga.widgets.dateinput.OnChangeHandler | None*) – A handler that will be invoked when the value changes.
- **kwargs** – Initial style properties.

property max: *date*

The maximum allowable date (inclusive). A value of None will be converted into the highest supported date of 8999-12-31.

When setting this property, the current *value* and *min* will be clipped against the new maximum value.

Raises
ValueError – If set to a date outside of the supported range.

property min: *date*

The minimum allowable date (inclusive). A value of None will be converted into the lowest supported date of 1800-01-01.

When setting this property, the current *value* and *max* will be clipped against the new minimum value.

Raises
ValueError – If set to a date outside of the supported range.

property on_change: *OnChangeHandler*

The handler to invoke when the date value changes.

property value: *date*

The currently selected date. A value of None will be converted into today's date.

If this property is set to a value outside of the min/max range, it will be clipped.

protocol toga.widgets.dateinput.**OnChangeHandler**

typing.Protocol.

Classes that implement this protocol must have the following methods / attributes:

__call__(*widget*, ***kwargs*)

A handler that will be invoked when a change occurs.

Parameters

- **widget** (DateInput) – The DateInput that was changed.
- **kwargs** (*Any*) – Ensures compatibility with arguments added in future versions.

Return type
object

DetailedList

macOS

Linux

Windows

Android

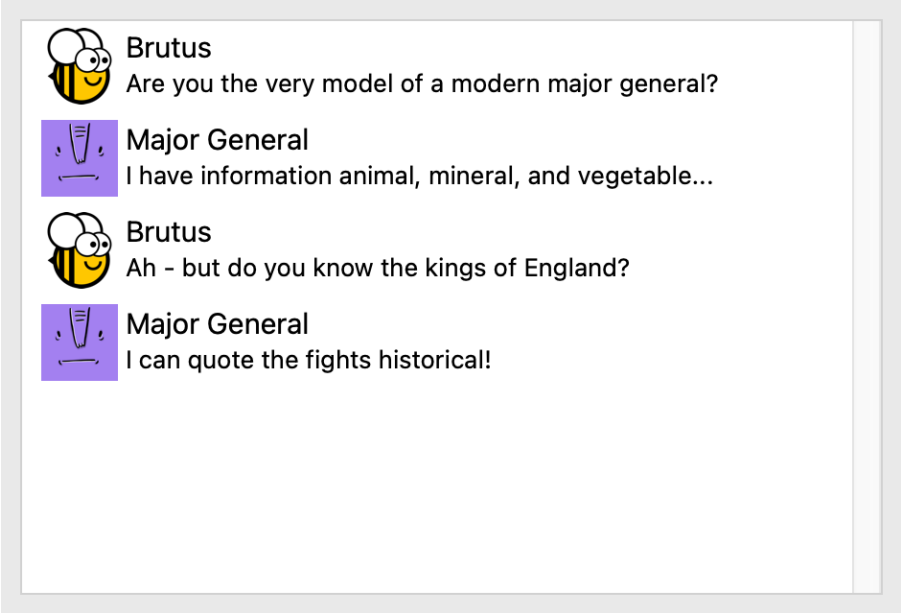
iOS

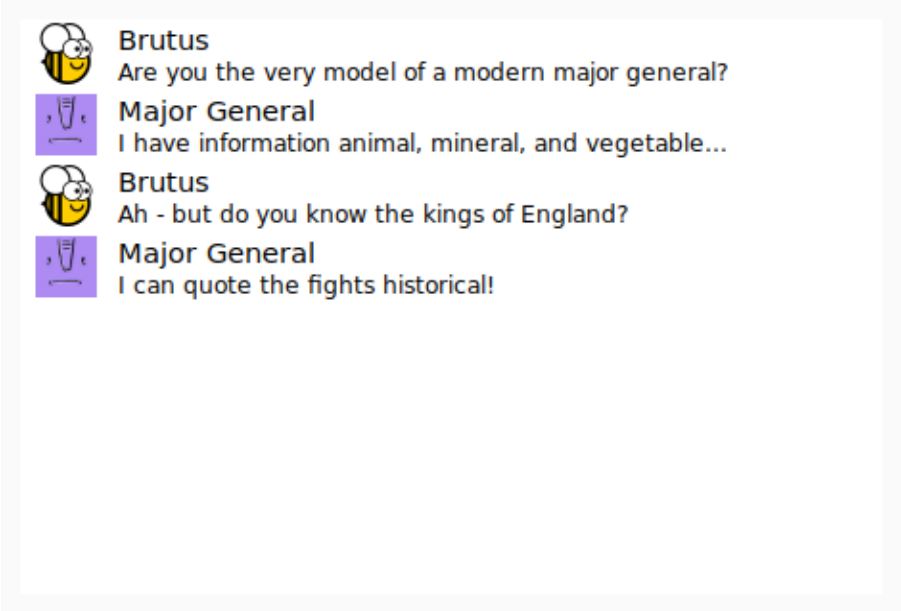
Web ×

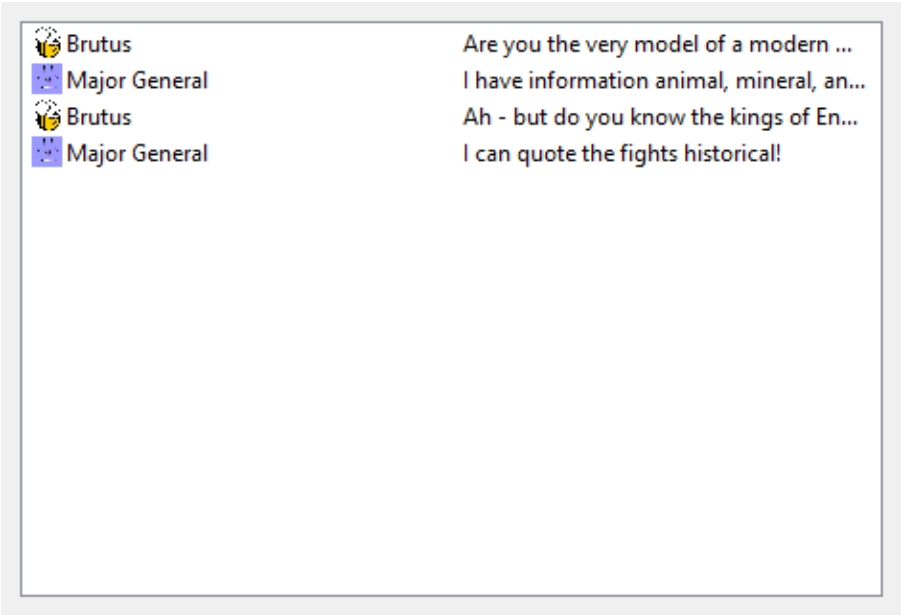
Textual ×

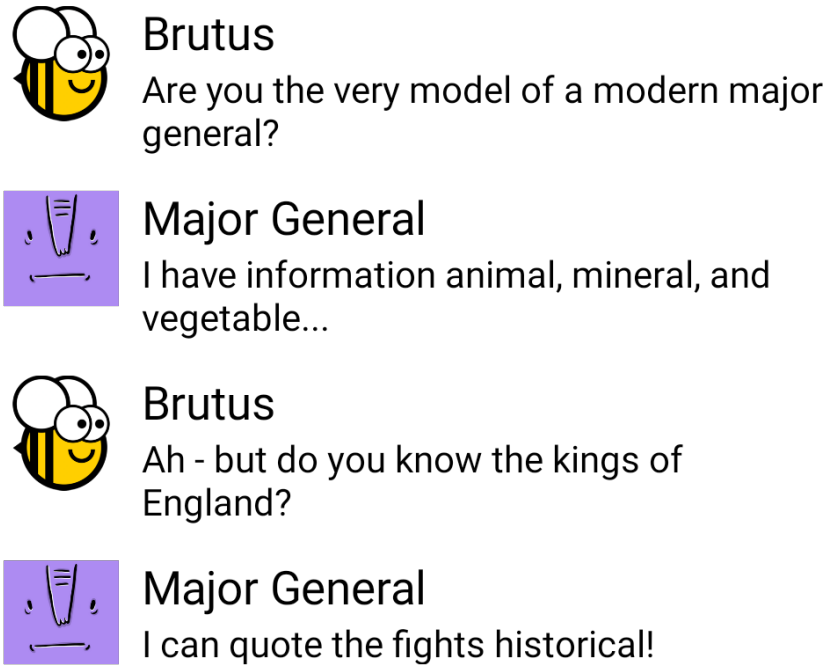
Not supported

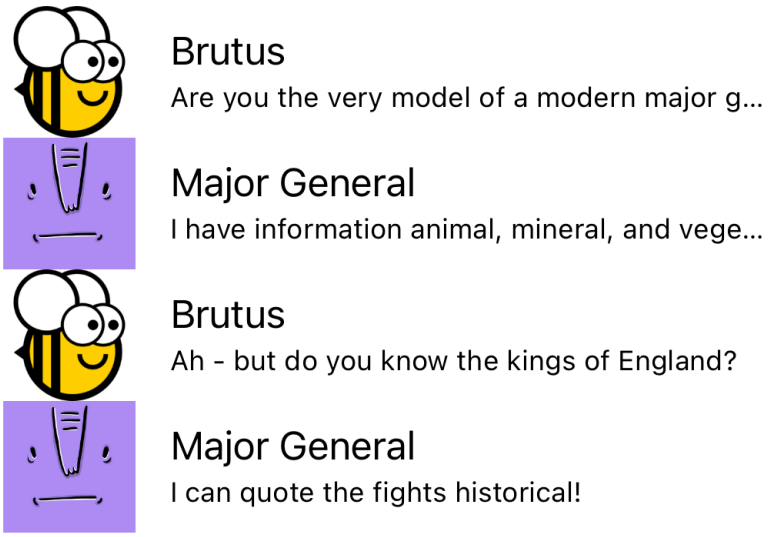
Not supported


Brutus
Are you the very model of a modern major general?
Major General
I have information animal, mineral, and vegetable...
Brutus
Ah - but do you know the kings of England?
Major General
I can quote the fights historical!


Brutus
Are you the very model of a modern major general?
Major General
I have information animal, mineral, and vegetable...
Brutus
Ah - but do you know the kings of England?
Major General
I can quote the fights historical!


Brutus
Are you the very model of a modern ...
Major General
I have information animal, mineral, an...
Brutus
Ah - but do you know the kings of En...
Major General
I can quote the fights historical!


Brutus
Are you the very model of a modern major general?
Major General
I have information animal, mineral, and vegetable...
Brutus
Ah - but do you know the kings of England?
Major General
I can quote the fights historical!



Usage

The simplest way to create a DetailedList is to pass a list of dictionaries, with each dictionary containing three keys: `icon`, `title`, and `subtitle`:

```
import toga

table = toga.DetailedList(
    data=[
        {
            "icon": toga.Icon("icons/arthur"),
            "title": "Arthur Dent",
            "subtitle": "Where's the tea?"
        },
        {
            "icon": toga.Icon("icons/ford"),
            "title": "Ford Prefect",
            "subtitle": "Do you know where my towel is?"
        },
        {
            "icon": toga.Icon("icons/tricia"),
            "title": "Tricia McMillan",
            "subtitle": "What planet are you from?"
        },
    ]
)
```

If you want to customize the keys used in the dictionary, you can do this by providing an `accessors` argument to the DetailedList when it is constructed. `accessors` is a tuple containing the attributes that will be used to provide the

icon, title, and subtitle, respectively:

```
import toga

table = toga.DetailedList(
    accessors=("picture", "name", "quote"),
    data=[
        {
            "picture": toga.Icon("icons/arthur"),
            "name": "Arthur Dent",
            "quote": "Where's the tea?"
        },
        {
            "picture": toga.Icon("icons/ford"),
            "name": "Ford Prefect",
            "quote": "Do you know where my towel is?"
        },
        {
            "picture": toga.Icon("icons/tricia"),
            "name": "Tricia McMillan",
            "quote": "What planet are you from?"
        },
    ]
)
```

If the value provided by the title or subtitle accessor is `None`, or the accessor isn't defined, the `missing_value` will be displayed. Any other value will be converted into a string.

The icon accessor should return an *Icon*. If it returns `None`, or the accessor isn't defined, then no icon will be displayed, but space for the icon will remain in the layout.

Items in a DetailedList can respond to a primary and secondary action. On platforms that use swipe interactions, the primary action will be associated with "swipe left", and the secondary action will be associated with "swipe right". Other platforms may implement the primary and secondary actions using a different UI interaction (e.g., a right-click context menu). The primary and secondary actions will only be enabled in the DetailedList UI if a handler has been provided.

By default, the primary and secondary action will be labeled as "Delete" and "Action", respectively. These names can be overridden by providing a `primary_action` and `secondary_action` argument when constructing the DetailedList. Although the primary action is labeled "Delete" by default, the DetailedList will not perform any data deletion as part of the UI interaction. It is the responsibility of the application to implement any data deletion behavior as part of the `on_primary_action` handler.

The DetailedList as a whole can also respond to a refresh UI action. This is usually implemented as a "pull down" action, such as you might see on a social media timeline. This action will only be enabled in the UI if an `on_refresh` handler has been provided.

Notes

- The iOS Human Interface Guidelines differentiate between "Normal" and "Destructive" actions on a row. Toga will interpret any action with a name of "Delete" or "Remove" as destructive, and will render the action appropriately.
- The WinForms implementation currently uses a column layout similar to *Table*, and does not support the primary, secondary or refresh actions.
- Using DetailedList on Android requires the AndroidX SwipeRefreshLayout widget in your

project's Gradle dependencies. Ensure your app declares a dependency on `androidx.swiperefreshlayout:swiperefreshlayout:1.1.0` or later.

Reference

class toga.**DetailedList**(*id=None*, *style=None*, *data=None*, *accessors=('title', 'subtitle', 'icon')*, *missing_value=''*, *primary_action='Delete'*, *on_primary_action=None*, *secondary_action='Action'*, *on_secondary_action=None*, *on_refresh=None*, *on_select=None*, ***kwargs*)

Bases: *Widget*

Create a new DetailedList widget.

Parameters

- **id** (*str | None*) – The ID for the widget.
- **style** (*StyleT | None*) – A style object. If no style is provided, a default style will be applied to the widget.
- **data** (*SourceT | Iterable | None*) – Initial *data* to be displayed in the list.
- **accessors** (*tuple[str, str, str]*) – The accessors to use to retrieve the data for each item, in the form (title, subtitle, icon).
- **missing_value** (*str*) – The text that will be shown when a row doesn't provide a value for its title or subtitle.
- **on_select** (toga.widgets.detailedlist.OnSelectHandler | *None*) – Initial *on_select* handler.
- **primary_action** (*str | None*) – The name for the primary action.
- **on_primary_action** (OnPrimaryActionHandler | *None*) – Initial *on_primary_action* handler.
- **secondary_action** (*str | None*) – The name for the secondary action.
- **on_secondary_action** (OnSecondaryActionHandler | *None*) – Initial *on_secondary_action* handler.
- **on_refresh** (OnRefreshHandler | *None*) – Initial *on_refresh* handler.
- **kwargs** – Initial style properties.

property accessors: `tuple[str, str, str]`

The accessors used to populate the list (read-only)

property data: `SourceT | ListSource`

The data to display in the table.

When setting this property:

- A *Source* will be used as-is. It must either be a *ListSource*, or a custom class that provides the same methods.
- A value of None is turned into an empty ListSource.
- Otherwise, the value must be an iterable, which is copied into a new ListSource. Items are converted as shown *here*.

property enabled: Literal[True]

Is the widget currently enabled? i.e., can the user interact with the widget? DetailedList widgets cannot be disabled; this property will always return True; any attempt to modify it will be ignored.

focus()

No-op; DetailedList cannot accept input focus.

Return type
None

property missing_value: str

The text that will be shown when a row doesn't provide a value for its title or subtitle.

property on_primary_action: *OnPrimaryActionHandler*

The handler to invoke when the user performs the primary action on a row of the DetailedList.

The primary action is "swipe left" on platforms that use swipe interactions; other platforms may manifest this action in other ways (e.g, a context menu).

If no on_primary_action handler is provided, the primary action will be disabled in the UI.

property on_refresh: *OnRefreshHandler*

The callback function to invoke when the user performs a refresh action (usually "pull down") on the DetailedList.

If no on_refresh handler is provided, the refresh UI action will be disabled.

property on_secondary_action: *OnSecondaryActionHandler*

The handler to invoke when the user performs the secondary action on a row of the DetailedList.

The secondary action is "swipe right" on platforms that use swipe interactions; other platforms may manifest this action in other ways (e.g, a context menu).

If no on_secondary_action handler is provided, the secondary action will be disabled in the UI.

property on_select: *OnSelectHandler*

The callback function that is invoked when a row of the DetailedList is selected.

scroll_to_bottom()

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

Return type
None

scroll_to_row(*row*)

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

Parameters
row (*int*) – 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).

Return type
None

scroll_to_top()

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

Return type
None

property selection: *Row* | None

The current selection of the table.

Returns the selected Row object, or None if no row is currently selected.

protocol toga.widgets.detailedlist.**OnPrimaryActionHandler**

typing.Protocol.

Classes that implement this protocol must have the following methods / attributes:

__call__(*widget*, *row*, ***kwargs*)

A handler to invoke for the primary action.

Parameters

- **widget** (DetailedList) – The DetailedList that was invoked.
- **row** (*Any*) – The current row for the detailed list.
- **kwargs** (*Any*) – Ensures compatibility with arguments added in future versions.

Return type
object

protocol toga.widgets.detailedlist.**OnSecondaryActionHandler**

typing.Protocol.

Classes that implement this protocol must have the following methods / attributes:

__call__(*widget*, *row*, ***kwargs*)

A handler to invoke for the secondary action.

Parameters

- **widget** (DetailedList) – The DetailedList that was invoked.
- **row** (*Any*) – The current row for the detailed list.
- **kwargs** (*Any*) – Ensures compatibility with arguments added in future versions.

Return type
object

protocol toga.widgets.detailedlist.**OnRefreshHandler**

typing.Protocol.

Classes that implement this protocol must have the following methods / attributes:

__call__(*widget*, ***kwargs*)

A handler to invoke when the detailed list is refreshed.

Parameters

- **widget** (DetailedList) – The DetailedList that was refreshed.
- **kwargs** (*Any*) – Ensures compatibility with arguments added in future versions.

Return type
object

protocol toga.widgets.detailedlist.**OnSelectHandler**

typing.Protocol.

Classes that implement this protocol must have the following methods / attributes:

__call__(*widget*, ***kwargs*)

A handler to invoke when the detailed list is selected.

Parameters

- **widget** (DetailedList) – The DetailedList that was selected.
- **kwargs** (*Any*) – Ensures compatibility with arguments added in future versions.

Return type
object

Divider

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

macOS

Linux

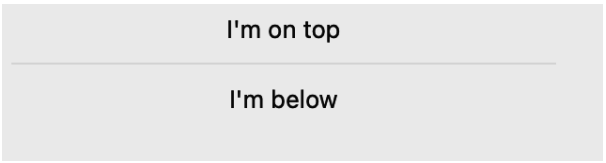
Windows

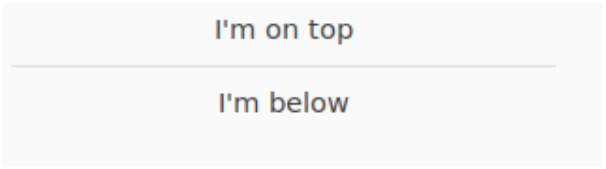
Android

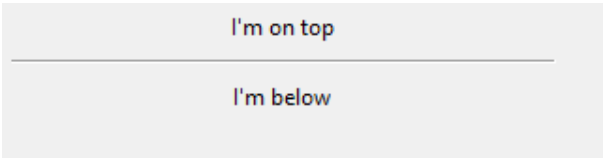
iOS

Web

Textual ×







Screenshot not available

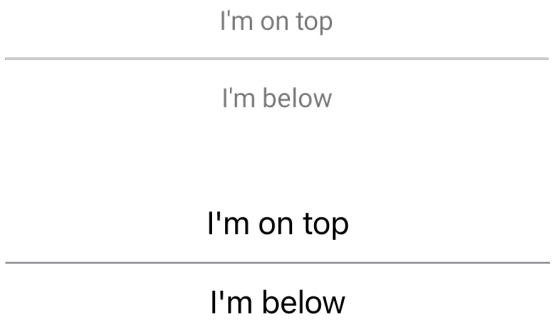
Not supported

Usage

To separate two labels stacked vertically with a horizontal line:

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

(continues on next page)



(continued from previous page)

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

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

Reference

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

Bases: *Widget*

Create a new divider line.

Parameters

- **id** (*str | None*) – The ID for the widget.
- **style** (*StyleT | None*) – A style object. If no style is provided, a default style will be applied to the widget.
- **direction** (Direction) – The direction in which the divider will be drawn. Either *HORIZONTAL* or *VERTICAL*; defaults to *HORIZONTAL*
- **kwargs** – Initial style properties.

HORIZONTAL = 0

VERTICAL = 1

property direction: *Direction*

The direction in which the visual separator will be drawn.

property enabled: Literal[True]

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

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

focus()

No-op; Divider cannot accept input focus.

Return type
None

ImageView

A widget that displays an image.

macOS

Linux

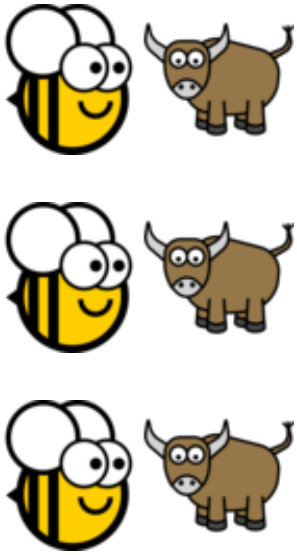
Windows

Android

iOS

Web ×

Textual ×



Not supported

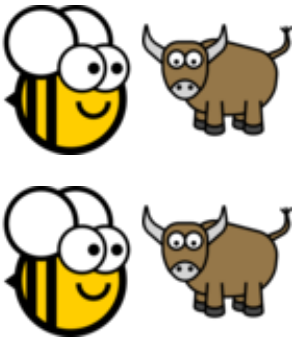
Not supported

Usage

An *ImageView* provides a mechanism to display an *Image* as part of an interface.

```
import toga

my_image = toga.Image(self.paths.app / "brutus.png")
view = toga.ImageView(my_image)
```



Notes

- An ImageView **is not** an interactive element - there is no `on_press` handler for ImageView. If you want a graphical element that can be clicked or pressed, try using a *toga.Button* that uses an *toga.Icon*.
- The default size of the view is the size of the image, or 0x0 if `image` is `None`.
- If an explicit width *or* height is specified, the size of the image will be fixed in that axis, and the size in the other axis will be determined by the image's aspect ratio.
- If an explicit width *and* height is specified, the image will be scaled to fill the described size without preserving the aspect ratio.
- If an ImageView is given a style of `flex=1`, and doesn't have an explicit size set along its container's main axis, it will be allowed to expand and contract along that axis, with the size determined by the flex allocation.
  - If the cross axis size is unspecified, it will be determined by applying the image's aspect ratio to the size allocated on the main axis.
  - If the cross axis has an explicit size, the image will be scaled to fill the available space so that the entire image can be seen, while preserving its aspect ratio. Any extra space will be distributed equally between both sides.

Reference

class toga.**ImageView**(*image=None*, *id=None*, *style=None*, ***kwargs*)

Bases: *Widget*

Create a new image view.

Parameters

- **image** (*ImageContentT | None*) – The image to display. Can be any valid *image content* type; or `None` to display no image.
- **id** (*str | None*) – The ID for the widget.
- **style** (*StyleT | None*) – A style object. If no style is provided, a default style will be applied to the widget.
- **kwargs** – Initial style properties.

as_image(*format=toga.Image*)

Return the image in the specified format.

Parameters

format (*type[ImageT]*) – Format to provide. Defaults to *Image*; also supports `PIL.Image.Image` if Pillow is installed, as well as any image types defined by installed *image format plugins*.

Returns
The image in the specified format.

Return type
ImageT

property enabled: Literal[True]

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

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

focus()

No-op; ImageView cannot accept input focus.

Return type
None

property image: *Image* | None

The image to display.

When setting an image, you can provide any valid *image content* type; or None to clear the image view.

Label

A text label for annotating forms or interfaces.

macOS

Linux

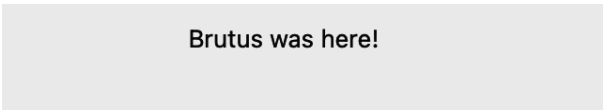
Windows

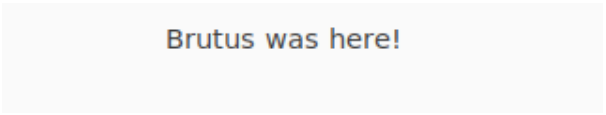
Android

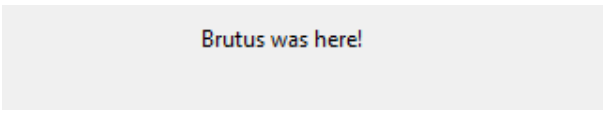
iOS

Web

Textual







Screenshot not available

Screenshot not available

Brutus was here!

Brutus was here!

Usage

```
import toga

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

Notes

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

Reference

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

Bases: *Widget*

Create a new text label.

Parameters

- **text** (*str*) – Text of the label.
- **id** (*str | None*) – The ID for the widget.
- **style** (*StyleT | None*) – A style object. If no style is provided, a default style will be applied to the widget.
- **kwargs** – Initial style properties.

focus()

No-op; Label cannot accept input focus.

Return type
None

property text: str

The text displayed by the label.

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

MapView

A zoomable map that can be annotated with location pins.

macOS

Linux

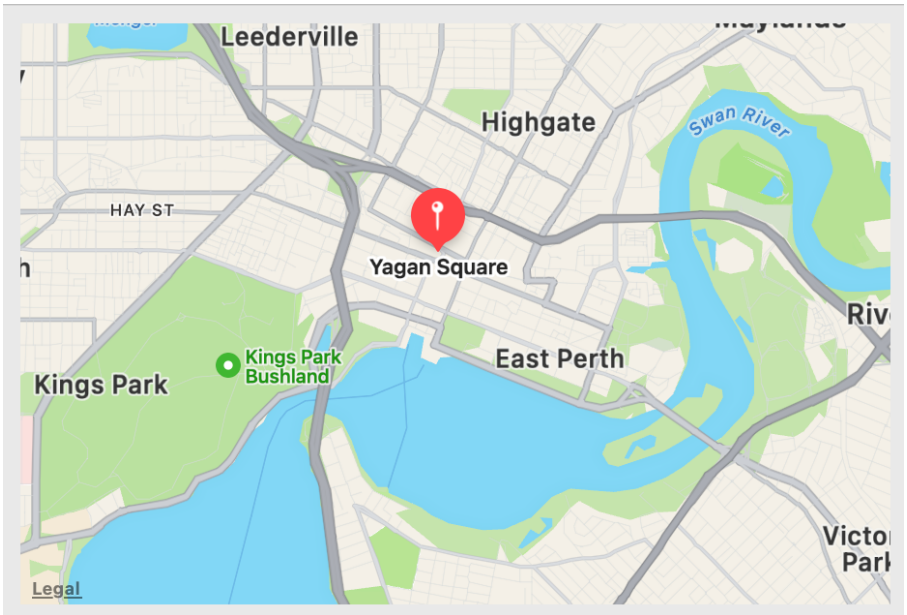
Windows

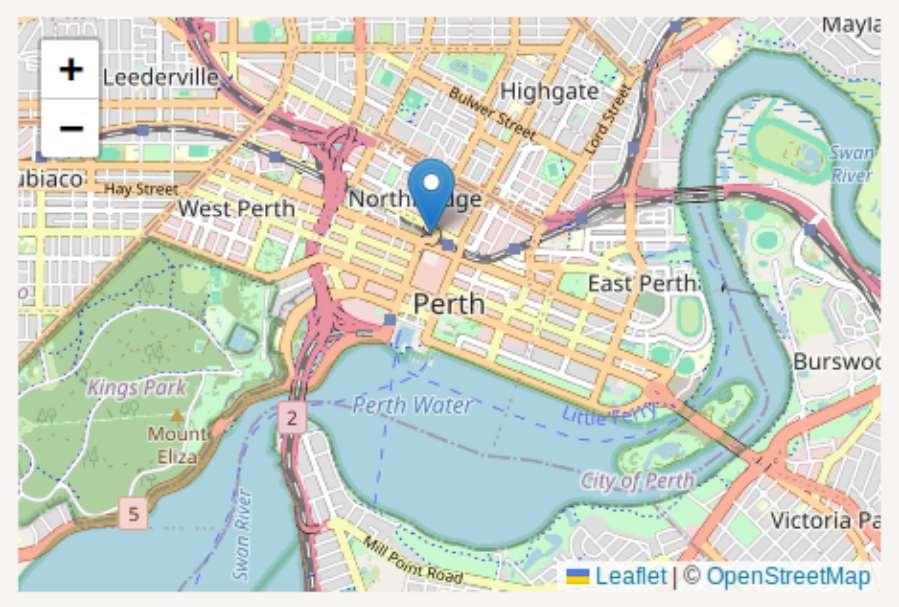
Android

iOS

Web ×

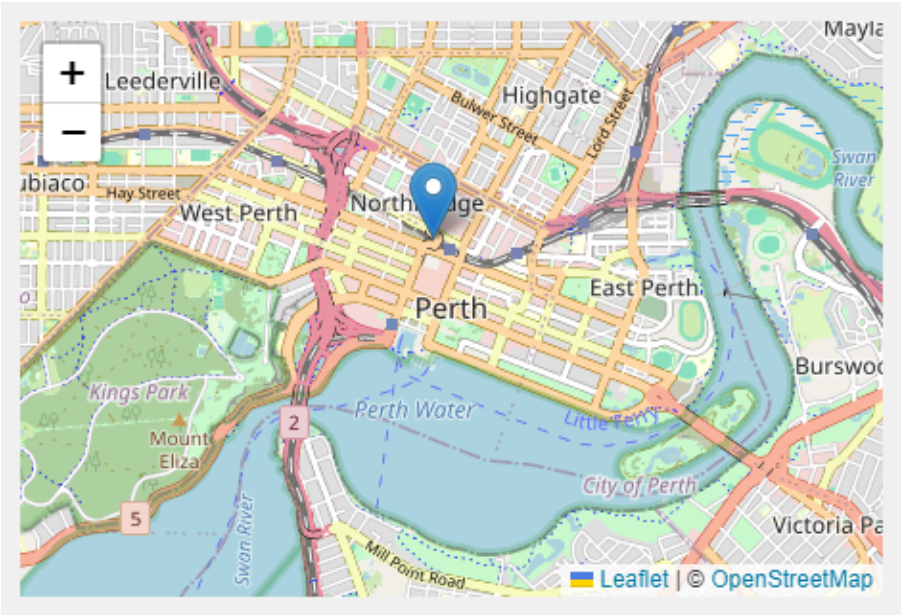
Textual ×

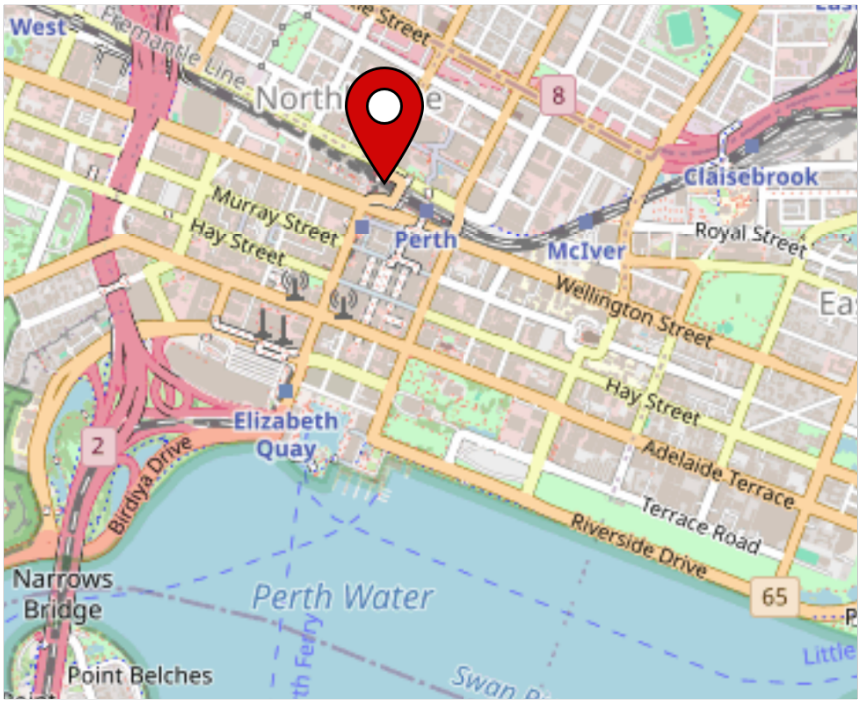


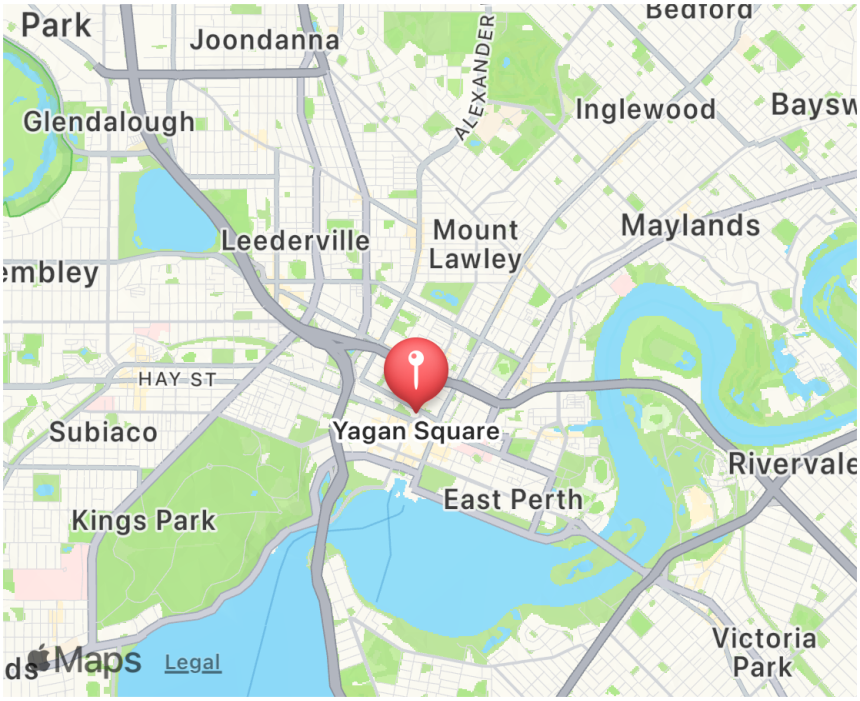


Not supported

Not supported


Leederville
Highgate
Hay Street
West Perth
East Perth
Perth
Kings Park
Mount Eliza
Perth Water
City of Perth
Swan River
Mill Point Road
Leaflet | © OpenStreetMap


West
Claisebrook
Murray Street
Hay Street
Perth
McIver
Royal Street
Wellington Street
Elizabeth Quay
Adelaide Terrace
Terrace Road
Riverside Drive
Narrows Bridge
Perth Water
Point Belches



Usage

A MapView is a scrollable area that can show a map at varying levels of detail, from nation-level to street level. The map can be centered at a given coordinate, and zoomed to the required level of detail using an integer from 0 (for global detail) to 20 (for building level detail):

```
import toga

# Create a map centered in London, UK.
mapview = toga.MapView(location=(51.507222, -0.1275))

# Center the map in Perth, Australia
mapview.location = (-31.9559, 115.8606)

# Zoom to show the map to show street level detail
mapview.zoom = 15
```

A map can also display pins. A map pin must have a title, and can optionally have a subtitle. Pins can be added at time of map construction, or can be dynamically added, updated and removed at runtime:

```
import toga

mapview = toga.MapView(
    pins=[
        toga.MapPin((-31.95064, 115.85889), title="Yagan Square"),
    ]
)

# Create a new pin, and add it to the map
```

(continues on next page)

(continued from previous page)

```
brutus = toga.MapPin((41.50375, -81.69475), title="Brutus was here")
mapview.pins.add(brutus)

# Update the pin label and position
brutus.location = (40.440831, -79.991162)
brutus.title = "Brutus will be here"

# Remove the Brutus pin
mapview.pins.remove(brutus)

# Remove all pins
mapview.pins.clear()
```

Pins can respond to being pressed. When a pin is pressed, the map generates an `on_select` event, which receives the pin as an argument.

System requirements

- Using MapView on Windows 10 requires that your users have installed the Edge WebView2 Evergreen Runtime. This is installed by default on Windows 11.
- Using MapView on Linux requires that the user has installed the system packages for WebKit2, plus the GObject Introspection bindings for WebKit2. The name of the system package required is distribution dependent:
  - Ubuntu 20.04; Debian 11: `gir1.2-webkit2-4.0`
  - Ubuntu 22.04+; Debian 12+: `gir1.2-webkit2-4.1`
  - Fedora: `webkit2gtk4.1`
  - Arch/Manjaro: `webkit2gtk-4.1`
  - OpenSUSE Tumbleweed: `libwebkit2gtk3 typelib(WebKit2)`
  - FreeBSD: `webkit2-gtk3`

  MapView is not fully supported on GTK4. If you want to contribute to the GTK4 MapView implementation, you will require v6.0 of the WebKit2 libraries. This is provided by `gir1.2-webkit-6.0` on Ubuntu/Debian, and `webkitgtk6.0` on Fedora; for other distributions, consult your distributions's platform documentation.
- Using MapView on Android requires the OSMDroid package in your project's Gradle dependencies. Ensure your app declares a dependency on `org\.osmdroid:osmdroid-android:6.1.20` or later.

Notes

- The Android, GTK and Winforms implementations of MapView use OpenStreetMap as a source of map tiles. OpenStreetMap is an open data project with its own copyright, license terms, and acceptable use policies. If you make use of MapView in your application, it is your responsibility to ensure that your app complies with these terms. In addition, we strongly encourage you to financially support the OpenStreetMap Foundation, as their work is what allows Toga to provide map content on these platforms.
- On macOS and iOS, MapView will not repeat map tiles if the viewable area at the given zoom level is bigger than the entire world. A zoom to a very low level will be clipped to the lowest level that allows displaying the map without repeating tiles.

Reference

class toga.**MapView**(*id=None*, *style=None*, *location=None*, *zoom=11*, *pins=None*, *on_select=None*, ***kwargs*)

Bases: *Widget*

Create a new MapView widget.

Parameters

- **id** (*str | None*) – The ID for the widget.
- **style** (*StyleT | None*) – A style object. If no style is provided, a default style will be applied to the widget.
- **location** (toga.LatLng | *tuple[float, float]* | *None*) – The initial latitude/longitude where the map should be centered. If not provided, the initial location for the map is Perth, Australia.
- **zoom** (*int*) – The initial zoom level for the map.
- **pins** (*Iterable[MapPin] | None*) – The initial pins to display on the map.
- **on_select** (toga.widgets.mapview.OnSelectHandler | *None*) – A handler that will be invoked when the user selects a map pin.
- **kwargs** – Initial style properties.

property location: *LatLng*

The latitude/longitude where the map is centered.

A tuple of `(latitude, longitude)` can be provided as input; this will be converted into a *toga.LatLng* object.

property on_select: *OnSelectHandler*

The handler to invoke when the user selects a pin on a map.

Note: This is not currently supported on GTK or Windows.

property pins: *MapPinSet*

The set of pins currently being displayed on the map

property zoom: int

Set the zoom level for the map.

The zoom level is an integer in the range 0-20 (inclusive). It can be used to set the number of degrees of longitude that will span a 256 *CSS pixel* region in the horizontal axis of the map, following the relationship:

```
longitude_per_256_pixels = 360 / (2**zoom)
```

In practical terms, this means a 256px square will cover:

- 0-2: Whole world
- 3-6: Large countries
- 7-8: Small countries, or a state in a large country
- 9-11: The extent of a city
- 12-14: Suburbs of a city, or small towns
- 15-17: Roads at the level useful for navigation
- 18-19: Individual buildings

- 20: A single building

These zoom levels use the same mathematical basis as the OpenStreetMap API. See OpenStreetMap's documentation on zoom levels for more details.

If the provided zoom value is outside the supported range, it will be clipped.

At very low zoom levels, some backends may constrain the viewable range to avoid repeating map tiles in the visible area. This effectively sets a minimum bound on the zoom level that can be requested. The value of this minimum varies depending on the size and aspect ratio of the map view.

class toga.**MapPin**(*location*, *, *title*, *subtitle=None*)

Create a new map pin.

Parameters

- **location** (*toga.LatLng | tuple[float, float]*) – A tuple describing the (latitude, longitude) for the pin.
- **title** (*str*) – The title to apply to the pin.
- **subtitle** (*str | None*) – A subtitle label to apply to the pin.

property location: *LatLng*

The (latitude, longitude) where the pin is located.

property subtitle: str | None

The subtitle of the pin.

property title: str

The title of the pin.

class toga.widgets.mapview.**MapPinSet**(*interface*, *pins*)

Parameters

- **interface** (MapView)
- **pins** (*Iterable[MapPin] | None*)

add(*pin*)

Add a new pin to the map.

Parameters
pin (MapPin) – The *toga.MapPin* instance to add.

Return type
None

clear()

Remove all pins from the map.

Return type
None

remove(*pin*)

Remove a pin from the map.

Parameters
pin (MapPin) – The *toga.MapPin* instance to remove.

Return type
None

protocol toga.widgets.mapview.**OnSelectHandler**

typing.Protocol.

Classes that implement this protocol must have the following methods / attributes:

__call__(*widget*, *, *pin*, ***kwargs*)

A handler that will be invoked when the user selects a map pin.

Parameters

- **widget** (MapView) – The MapView that was selected.
- **pin** (MapPin) – The pin that was selected.
- **kwargs** (*Any*) – Ensures compatibility with arguments added in future versions.

Return type
object

MultilineTextInput

A scrollable panel that allows for the display and editing of multiple lines of text.

macOS

Linux

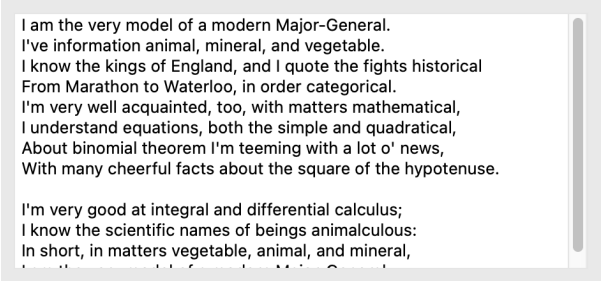
Windows

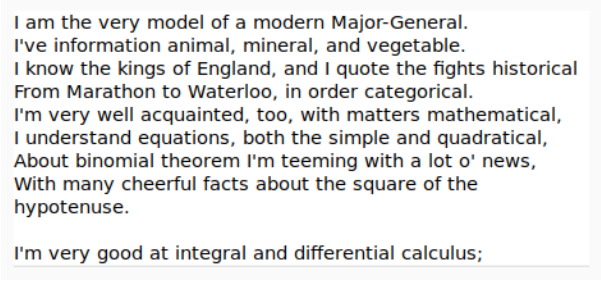
Android

iOS

Web ×

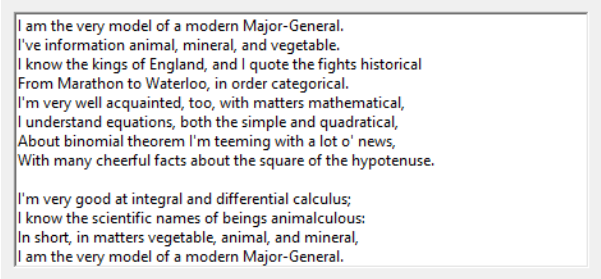
Textual ×

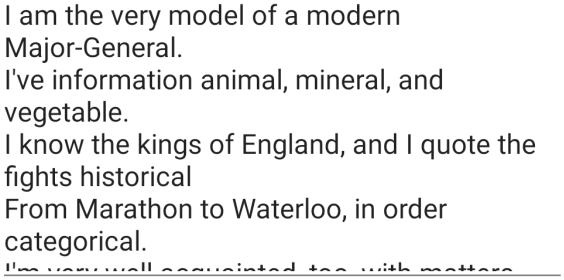


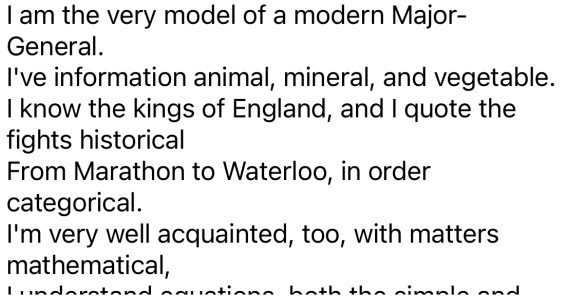


Not supported

Not supported


I am the very model of a modern Major-General.
I've information animal, mineral, and vegetable.
I know the kings of England, and I quote the fights historical
From Marathon to Waterloo, in order categorical.
I'm very well acquainted, too, with matters mathematical,
I understand equations, both the simple and quadratical,
About binomial theorem I'm teeming with a lot o' news,
With many cheerful facts about the square of the hypotenuse.
I'm very good at integral and differential calculus;
I know the scientific names of beings animalculous:
In short, in matters vegetable, animal, and mineral,
I am the very model of a modern Major-General.


I am the very model of a modern
Major-General.
I've information animal, mineral, and
vegetable.
I know the kings of England, and I quote the
fights historical
From Marathon to Waterloo, in order
categorical.


I am the very model of a modern Major-
General.
I've information animal, mineral, and vegetable.
I know the kings of England, and I quote the
fights historical
From Marathon to Waterloo, in order
categorical.
I'm very well acquainted, too, with matters
mathematical,

Usage

```
import toga

textbox = toga.MultilineTextInput()
textbox.value = "Some text.\nIt can be multiple lines of text."
```

The input can be provided a placeholder value - this is a value that will be displayed to the user as a prompt for appropriate content for the widget. This placeholder will only be displayed if the widget has no content; as soon as a value is provided (either by the user, or programmatically), the placeholder content will be hidden.

Notes

- Winforms does not support the use of partially or fully transparent colors for the MultilineTextInput background. If a color with an alpha value is provided (including `TRANSPARENT`), the alpha channel will be ignored. A `TRANSPARENT` background will be rendered as white.

Reference

class toga.**MultilineTextInput**(*id=None*, *style=None*, *value=None*, *readonly=False*, *placeholder=None*, *on_change=None*, ***kwargs*)

Bases: *Widget*

Create a new multi-line text input widget.

Parameters

- **id** (*str | None*) – The ID for the widget.
- **style** (*StyleT | None*) – A style object. If no style is provided, a default style will be applied to the widget.
- **value** (*str | None*) – The initial content to display in the widget.
- **readonly** (*bool*) – Can the value of the widget be modified by the user?
- **placeholder** (*str | None*) – The content to display as a placeholder when there is no user content to display.
- **on_change** (*toga.widgets.multilinetextinput.OnChangeHandler | None*) – A handler that will be invoked when the value of the widget changes.
- **kwargs** – Initial style properties.

property on_change: *OnChangeHandler*

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

property placeholder: *str*

The placeholder text for the widget.

A value of `None` will be interpreted and returned as an empty string. Any other object will be converted to a string using `str()`.

property readonly: *bool*

Can the value of the widget be modified by the user?

This only controls manual changes by the user (i.e., typing at the keyboard). Programmatic changes are permitted while the widget has `readonly` enabled.

scroll_to_bottom()

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

Return type
None

scroll_to_top()

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

Return type
None

property value: str

The text to display in the widget.

A value of None will be interpreted and returned as an empty string. Any other object will be converted to a string using str().

protocol toga.widgets.multilinetextinput.**OnChangeHandler**

typing.Protocol.

Classes that implement this protocol must have the following methods / attributes:

__call__(*widget*, ***kwargs*)

A handler to invoke when the value is changed.

Parameters

- **widget** (MultilineTextInput) – The MultilineTextInput that was changed.
- **kwargs** (*Any*) – Ensures compatibility with arguments added in future versions.

Return type
object

NumberInput

A text input that is limited to numeric input.

macOS

Linux

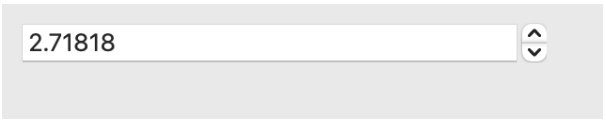
Windows

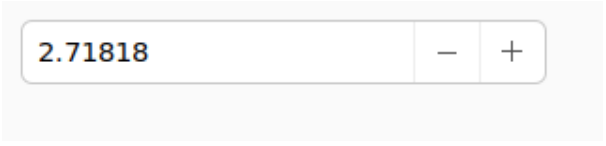
Android

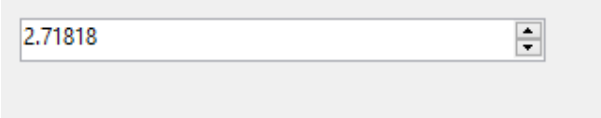
iOS

Web ×

Textual ×







Not supported

Not supported

Usage

```
import toga

widget = toga.NumberInput(min_value=1, max_value=10, step=0.001)
widget.value = 2.718
```

NumberInput's properties can accept `Decimal`, `int`, `float`, or `str` containing numbers, but they always return `Decimal` objects to ensure precision is retained.

Reference

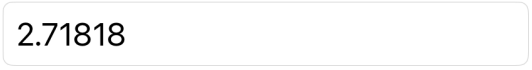
class toga.**NumberInput**(*id=None*, *style=None*, *step=1*, *min=None*, *max=None*, *value=None*, *readonly=False*, *on_change=None*, ***kwargs*)

Bases: *Widget*

Create a new number input widget.

Parameters

- **id** (*str | None*) – The ID for the widget.
- **style** (*StyleT | None*) – A style object. If no style is provided, a default style will be applied to the widget.
- **step** (*NumberInputT*) – The amount that any increment/decrement operations will apply to the widget's current value.
- **min** (*NumberInputT | None*) – If provided, *value* will be guaranteed to be greater than or equal to this minimum.
- **max** (*NumberInputT | None*) – If provided, *value* will be guaranteed to be less than or equal to this maximum.
- **value** (*NumberInputT | None*) – The initial value for the widget.
- **readonly** (*bool*) – Can the value of the widget be modified by the user?



- **on_change** (toga.widgets.numberinput.OnChangeHandler | *None*) – A handler that will be invoked when the value of the widget changes.
- **kwargs** – Initial style properties.

property max: Decimal | None

The maximum bound for the widget's value.

Returns None if there is no maximum bound.

When setting this property, the current *value* and *min* will be clipped against the new maximum value.

property min: Decimal | None

The minimum bound for the widget's value.

Returns None if there is no minimum bound.

When setting this property, the current *value* and *max* will be clipped against the new minimum value.

property on_change: *OnChangeHandler*

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

property readonly: bool

Can the value of the widget be modified by the user?

This only controls manual changes by the user (i.e., typing at the keyboard). Programmatic changes are permitted while the widget has `readonly` enabled.

property step: Decimal

The amount that any increment/decrement operations will apply to the widget's current value. (Not all backends provide increment and decrement buttons.)

property value: Decimal | None

Current value of the widget, rounded to the same number of decimal places as *step*, or None if no value has been set.

If this property is set to a value outside of the min/max range, it will be clipped.

While the widget is being edited by the user, it is possible for the UI to contain a value which is outside of the min/max range, or has too many decimal places. In this case, this property will return a value that has been clipped and rounded, and the visible text will be updated to match as soon as the widget loses focus.

protocol toga.widgets.numberinput.**OnChangeHandler**

typing.Protocol.

Classes that implement this protocol must have the following methods / attributes:

__call__(*widget*, ***kwargs*)

A handler to invoke when the value is changed.

Parameters

- **widget** (NumberInput) – The NumberInput that was changed.
- **kwargs** (*Any*) – Ensures compatibility with arguments added in future versions.

Return type
object

PasswordInput

A widget to allow the entry of a password. Any value typed by the user will be obscured, allowing the user to see the number of characters they have typed, but not the actual characters.

macOS

Linux

Windows

Android

iOS

Web ×

Textual ×

Not supported

Not supported

Usage

The `PasswordInput` is functionally identical to a *`TextInput`*, except for how the text is displayed. All features supported by *`TextInput`* are also supported by PasswordInput.

```
import toga

password = toga.PasswordInput()
```

Notes

- Winforms does not support the use of partially or fully transparent colors for the PasswordInput background. If a color with an alpha value is provided (including `TRANSPARENT`), the alpha channel will be ignored. A `TRANSPARENT` background will be rendered as white.
- On Winforms, if a PasswordInput is given an explicit height, the rendered widget will not expand to fill that space. The widget will have the fixed height determined by the font used on the widget. In general, you should avoid setting a `height` style property on PasswordInput widgets.

••••••

Reference

class toga.**PasswordInput**(*id=None*, *style=None*, *value=None*, *readonly=False*, *placeholder=None*, *on_change=None*, *on_confirm=None*, *on_gain_focus=None*, *on_lose_focus=None*, *validators=None*, ***kwargs*)

Bases: *TextInput*

Create a new password input widget.

Create a new single-line text input widget.

Parameters

- **id** (*str | None*) – The ID for the widget.
- **style** (*StyleT | None*) – A style object. If no style is provided, a default style will be applied to the widget.
- **value** (*str | None*) – The initial content to display in the widget.
- **readonly** (*bool*) – Can the value of the widget be modified by the user?
- **placeholder** (*str | None*) – The content to display as a placeholder when there is no user content to display.
- **on_change** (toga.widgets.textinput.OnChangeHandler | *None*) – A handler that will be invoked when the value of the widget changes.
- **on_confirm** (OnConfirmHandler | *None*) – A handler that will be invoked when the user accepts the value of the input (usually by pressing Return on the keyboard).
- **on_gain_focus** (OnGainFocusHandler | *None*) – A handler that will be invoked when the widget gains input focus.
- **on_lose_focus** (OnLoseFocusHandler | *None*) – A handler that will be invoked when the widget loses input focus.
- **validators** (*Iterable[Callable[[str], bool]] | None*) – A list of validators to run on the value of the input.
- **kwargs** – Initial style properties.

ProgressBar

A horizontal bar to visualize task progress. The task being monitored can be of known or indeterminate length.

macOS

Linux

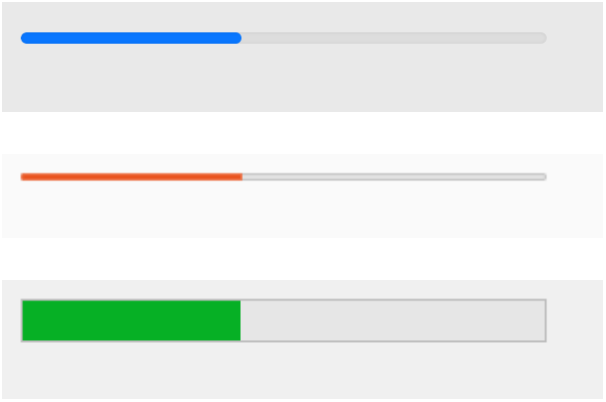
Windows

Android

iOS

Web

Textual ×



Screenshot not available

Not supported

Usage

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

```
import toga

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

# Start progress animation
progress.start()

# Update progress to 10%
progress.value = 10

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

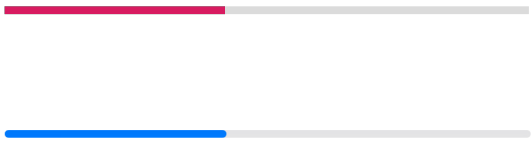
If a progress bar does *not* have a `max` value (i.e., `max == None`), it is an *indeterminate* progress bar. Any change to the value of an indeterminate progress bar will be ignored. When started, an indeterminate progress bar animates as a throbbing or "ping pong" animation.

```
import toga

progress = toga.ProgressBar(max=None)

# Start progress animation
progress.start()
```

(continues on next page)



(continued from previous page)

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

Notes

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

Reference

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

Bases: *Widget*

Create a new Progress Bar widget.

Parameters

- **id** (*str | None*) – The ID for the widget.
- **style** (*StyleT | None*) – A style object. If no style is provided, a default style will be applied to the widget.
- **max** (*str | SupportsFloat*) – The value that represents completion of the task. Must be > 0.0; defaults to 1.0. A value of `None` indicates that the task length is indeterminate.
- **value** (*str | SupportsFloat*) – The current progress against the maximum value. Must be between 0.0 and `max`; any value outside this range will be clipped. Defaults to 0.0.
- **running** (*bool*) – Describes whether the indicator is running at the time it is created. Default is False.
- **kwargs** – Initial style properties.

property enabled: `Literal[True]`

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

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

property is_determinate: `bool`

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

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

property is_running: `bool`

Describe if the activity indicator is currently running.

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

True if this activity indicator is running; False otherwise.

property max: `float` | `None`

The value indicating completion of the task being monitored.

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

start()

Start the progress bar.

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

Return type
None

stop()

Stop the progress bar.

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

Return type
None

property value: `float`

The current value of the progress indicator.

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

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

Selection

A widget to select a single option from a list of alternatives.

macOS

Linux

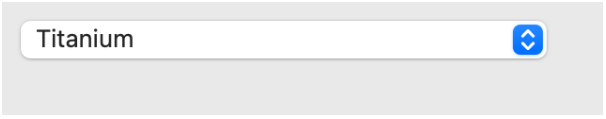
Windows

Android

iOS

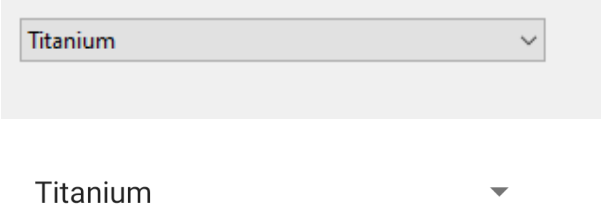
Web ×

Textual ×



Not supported

Not supported



Usage

The Selection uses a *ListSource* to manage the list of options. If `items` is not specified as a ListSource, it will be converted into a ListSource at runtime.

The simplest instantiation of a Selection is to use a list of strings. If a list of non-string objects are provided, they will be converted into a string for display purposes, but the original data type will be retained when returning the current value. If the string value contains newlines, only the substring up to the first newline will be displayed.

```
import toga

selection = toga.Selection(items=["Alice", "Bob", "Charlie"])

# Change the selection to "Charlie"
selection.value = "Charlie"

# Which item is currently selected? This will print "Charlie"
print(f"Currently selected: {selection.value}")
```

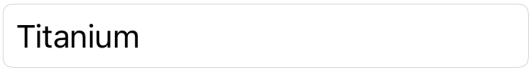
A Selection can also be used to display a list of dictionaries, with the `accessor` detailing which attribute of the dictionary will be used for display purposes. When the current value of the widget is retrieved, a Row object will be returned; this Row object will have all the original data as attributes on the Row. In the following example, the GUI will only display the names in the list of items, but the age will be available as an attribute on the selected item.

```
import toga

selection = toga.Selection(
    items=[
        {"name": "Alice", "age": 37},
        {"name": "Bob", "age": 42},
        {"name": "Charlie", "age": 24},
    ],
    accessor="name",
)

# Select Bob explicitly
selection.value = selection.items[1]
```

(continues on next page)



(continued from previous page)

```
# What is the age of the currently selected person? This will print 42
print(f"Age of currently selected person: {selection.value.age}")

# Select Charlie by searching
selection.value = selection.items.find(name="Charlie")
```

Notes

- On macOS and Android, you cannot change the font of a Selection.
- On macOS, GTK and Android, you cannot change the text color, background color, or text alignment of labels in a Selection.
- On GTK, a Selection widget with flexible sizing will expand its width (to the extent possible possible) to accommodate any changes in content (for example, to accommodate a long label). However, if the content subsequently *decreases* in width, the Selection widget *will not* shrink. It will retain the size necessary to accommodate the longest label it has historically contained.
- On iOS, the size of the Selection widget does not adapt to the size of the currently displayed content, or the potential list of options.

Reference

class toga.**Selection**(*id=None*, *style=None*, *items=None*, *accessor=None*, *value=None*, *on_change=None*, *enabled=True*, ***kwargs*)

Bases: *Widget*

Create a new Selection widget.

Parameters

- **id** (*str | None*) – The ID for the widget.
- **style** (*StyleT | None*) – A style object. If no style is provided, a default style will be applied to the widget.
- **items** (*SourceT | Iterable | None*) – Initial *items* to display for selection.
- **accessor** (*str | None*) – The accessor to use to extract display values from the list of items. See *items* and *value* for details on how `accessor` alters the interpretation of data in the Selection.
- **value** (*object | None*) – Initial value for the selection. If unspecified, the first item in `items` will be selected.
- **on_change** (*toga.widgets.selection.OnChangeHandler | None*) – Initial *on_change* handler.
- **enabled** (*bool*) – Whether the user can interact with the widget.
- **kwargs** – Initial style properties.

property items: SourceT | *ListSource*

The items to display in the selection.

When setting this property:

- A *Source* will be used as-is. It must either be a *ListSource*, or a custom class that provides the same methods.

- A value of None is turned into an empty ListSource.
- Otherwise, the value must be an iterable, which is copied into a new ListSource using the widget's accessor, or "value" if no accessor was specified. Items are converted as shown *here*.

property on_change: OnChangeHandler

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

property value: object | None

The currently selected item.

Returns None if there are no items in the selection.

If an `accessor` was specified when the Selection was constructed, the value returned will be Row objects from the ListSource; to change the selection, a Row object from the ListSource must be provided.

If no `accessor` was specified when the Selection was constructed, the value returned will be the value stored as the `value` attribute on the Row object. When setting the value, the widget will search for the first Row object whose `value` attribute matches the provided value. In practice, this means that you can treat the selection as containing a list of literal values, rather than a ListSource containing Row objects.

Slider

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

macOS

Linux

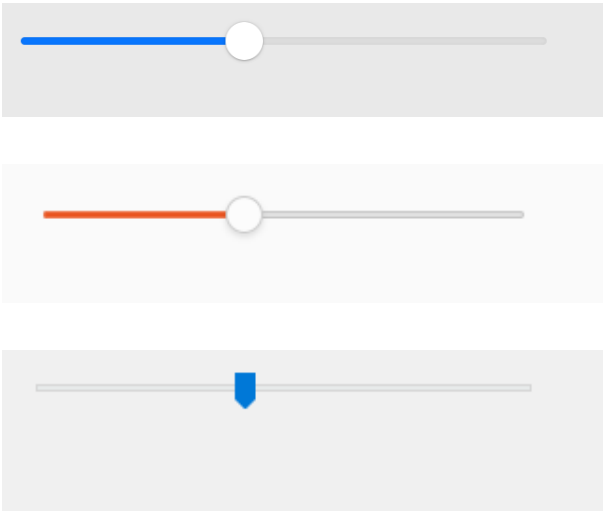
Windows

Android

iOS

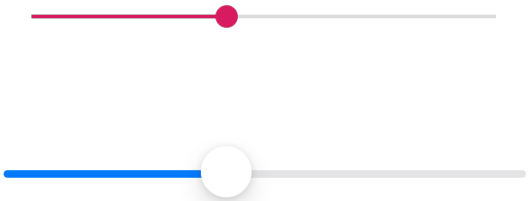
Web ×

Textual ×



Not supported

Not supported



Usage

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

```
import toga

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

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

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

Reference

class toga.**Slider**(*id=None*, *style=None*, *value=None*, *min=0.0*, *max=1.0*, *tick_count=None*, *on_change=None*, *on_press=None*, *on_release=None*, *enabled=True*, ***kwargs*)

Bases: *Widget*

Create a new Slider widget.

Parameters

- **id** (*str* | *None*) – The ID for the widget.
- **style** (*StyleT* | *None*) – A style object. If no style is provided, a default style will be applied to the widget.
- **value** (*float* | *None*) – Initial *value* of the slider. Defaults to the mid-point of the range.
- **min** (*float*) – Initial minimum value of the slider. Defaults to 0.
- **max** (*float*) – Initial maximum value of the slider. Defaults to 1.
- **tick_count** (*int* | *None*) – Initial *tick_count* for the slider. If None, the slider will be continuous.
- **on_change** (toga.widgets.slider.OnChangeHandler | *None*) – Initial *on_change* handler.
- **on_press** (toga.widgets.slider.OnPressHandler | *None*) – Initial *on_press* handler.
- **on_release** (OnReleaseHandler | *None*) – Initial *on_release* handler.
- **enabled** (*bool*) – Whether the user can interact with the widget.

- **kwargs** – Initial style properties.

property max: *float*

Maximum allowed value.

When setting this property, the current *value* and *min* will be clipped against the new maximum value.

property min: *float*

Minimum allowed value.

When setting this property, the current *value* and *max* will be clipped against the new minimum value.

property on_change: *OnChangeHandler*

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

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

property on_press: *OnPressHandler*

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

property on_release: *OnReleaseHandler*

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

property tick_count: *int* | *None*

Number of tick marks to display on the slider.

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

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

Raises

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

Note

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

property tick_step: *float* | *None*

Step between adjacent ticks.

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

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

property tick_value: *float* | *None*

Value of the slider, measured in ticks.

- If the slider is continuous, this property returns `None`.
- If the slider is discrete, it returns an integer between 1 (representing *min*) and *tick_count* (representing *max*).

Raises

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

property value: float

Current value.

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

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

protocol toga.widgets.slider.OnChangeHandler

typing.Protocol.

Classes that implement this protocol must have the following methods / attributes:

__call__(*widget*, ***kwargs*)

A handler to invoke when the value is changed.

Parameters

- **widget** (Slider) – The Slider that was changed.
- **kwargs** (*Any*) – Ensures compatibility with arguments added in future versions.

Return type
object

protocol toga.widgets.slider.OnPressHandler

typing.Protocol.

Classes that implement this protocol must have the following methods / attributes:

__call__(*widget*, ***kwargs*)

A handler to invoke when the slider is pressed.

Parameters

- **widget** (Slider) – The Slider that was pressed.
- **kwargs** (*Any*) – Ensures compatibility with arguments added in future versions.

Return type
object

protocol toga.widgets.slider.OnReleaseHandler

typing.Protocol.

Classes that implement this protocol must have the following methods / attributes:

__call__(*widget*, ***kwargs*)

A handler to invoke when the slider is pressed.

Parameters

- **widget** (Slider) – The Slider that was released.
- **kwargs** (*Any*) – Ensures compatibility with arguments added in future versions.

Return type
object

Switch

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

macOS

Linux

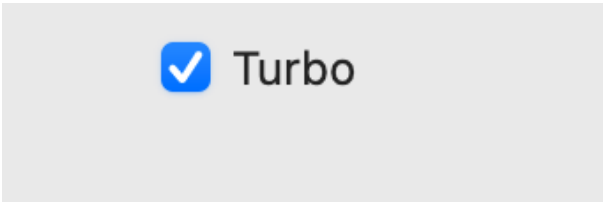
Windows

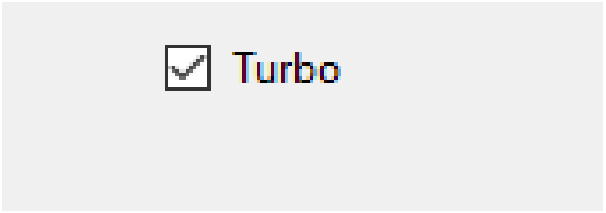
Android

iOS

Web

Textual ×





Screenshot not available

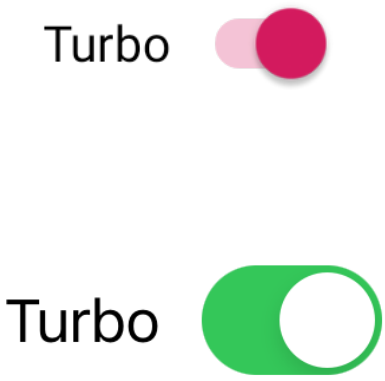
Not supported

Usage

```
import toga

switch = toga.Switch()

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



Notes

- The button and the label are considered a single widget for layout purposes.
- The visual appearance of a Switch is not guaranteed. On some platforms, it will render as a checkbox. On others, it will render as a physical "switch" whose position (and color) indicates if the switch is active. When rendered as a checkbox, the label will appear to the right of the checkbox. When rendered as a switch, the label will be left-aligned, and the switch will be right-aligned.
- You should avoid setting a `height` style property on Switch widgets. The rendered height of the Switch widget will be whatever the platform style guide considers appropriate; explicitly setting a `height` for the widget can lead to widgets that have a distorted appearance.
- On macOS, the text color of the label cannot be set directly; any `color` style directive will be ignored.

Reference

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

Bases: *Widget*

Create a new Switch widget.

Parameters

- **text** (*str*) – The text to display beside the switch.
- **id** (*str | None*) – The ID for the widget.
- **style** (*StyleT | None*) – A style object. If no style is provided, a default style will be applied to the widget.
- **value** (*bool*) – The initial value for the switch.
- **on_change** (*toga.widgets.switch.OnChangeHandler | None*) – A handler that will be invoked when the switch changes value.
- **enabled** (*bool*) – Is the switch enabled (i.e., can it be pressed?). Optional; by default, switches are created in an enabled state.
- **kwargs** – Initial style properties.

property on_change: *OnChangeHandler*

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

property text: str

The text label for the Switch.

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

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

toggle()

Reverse the current value of the switch.

Return type
None

property value: bool

The current state of the switch, as a Boolean.

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

protocol toga.widgets.switch.**OnChangeHandler**

typing.Protocol.

Classes that implement this protocol must have the following methods / attributes:

__call__(*widget*, ***kwargs*)

A handler to invoke when the value is changed.

Parameters

- **widget** (Switch) – The Switch that was changed.
- **kwargs** (*Any*) – Ensures compatibility with arguments added in future versions.

Return type
object

Table

A widget for displaying columns of tabular data. Scroll bars will be provided if necessary.

macOS

Linux

Windows

Android

iOS ×

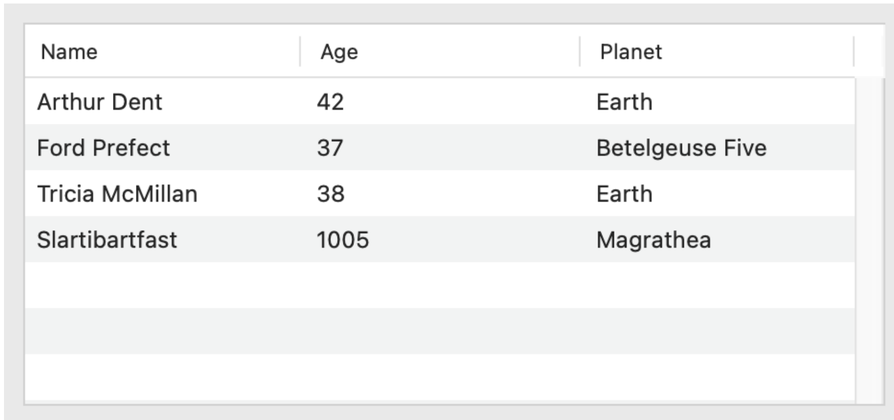
Web ×

Textual ×

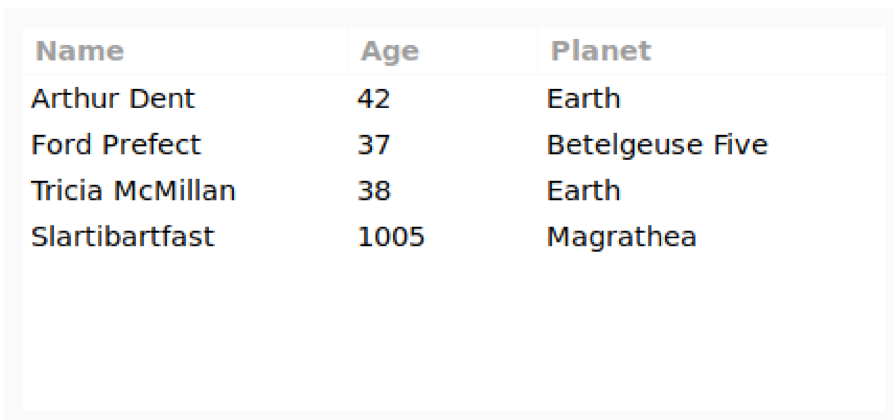
Not supported

Not supported

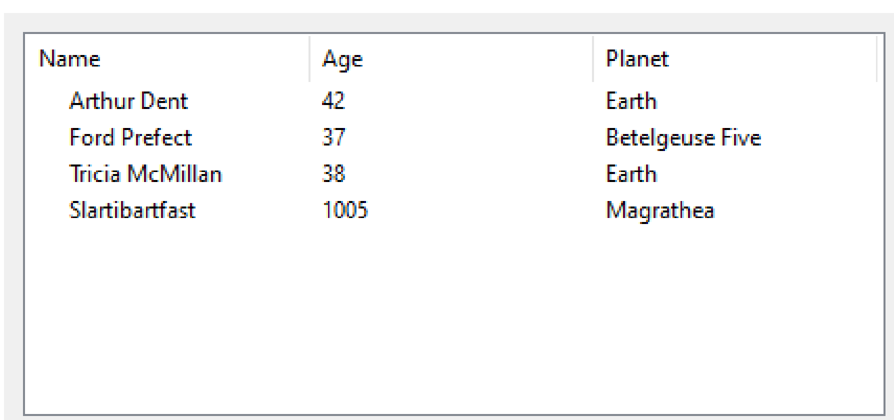
Not supported



Name	Age	Planet
Arthur Dent	42	Earth
Ford Prefect	37	Betelgeuse Five
Tricia McMillan	38	Earth
Slartibartfast	1005	Magrathea



Name	Age	Planet
Arthur Dent	42	Earth
Ford Prefect	37	Betelgeuse Five
Tricia McMillan	38	Earth
Slartibartfast	1005	Magrathea



Name	Age	Planet
Arthur Dent	42	Earth
Ford Prefect	37	Betelgeuse Five
Tricia McMillan	38	Earth
Slartibartfast	1005	Magrathea

Name	Age	Planet
Arthur Dent	42	Earth
Ford Prefect	37	Betelgeuse Five
Tricia McMillan	38	Earth
Slartibartfast	1005	Magrathea

Usage

The simplest way to create a Table is to pass a list of tuples containing the items to display, and a list of column headings. The values in the tuples will then be mapped sequentially to the columns.

In this example, we will display a table of 2 columns, with 3 initial rows of data:

```
import toga

table = toga.Table(
    headings=["Name", "Age"],
    data=[
        ("Arthur Dent", 42),
        ("Ford Prefect", 37),
        ("Tricia McMillan", 38),
    ]
)

# Get the details of the first item in the data:
print(f"{table.data[0].name} is age {table.data[0].age}")

# Append new data to the table
table.data.append(("Zaphod Beeblebrox", 47))
```

You can also specify data for a Table using a list of dictionaries. This allows you to store data in the data source that won't be displayed in the table. It also allows you to control the display order of columns independent of the storage of that data.

```
import toga

table = toga.Table(
    headings=["Name", "Age"],
    data=[
        {"name": "Arthur Dent", "age": 42, "planet": "Earth"},
        {"name": "Ford Prefect", "age": 37, "planet": "Betelgeuse Five"},
        {"name": "Tricia McMillan", "age": 38, "planet": "Earth"},
    ]
```

(continues on next page)

(continued from previous page)

```
)

# Get the details of the first item in the data:
row = table.data[0]
print(f"{row.name}, who is age {row.age}, is from {row.planet}")
```

The attribute names used on each row (called "accessors") are created automatically from the headings, by:

1. Converting the heading to lower case
2. Removing any character that can't be used in a Python identifier
3. Replacing all whitespace with _
4. Prepending _ if the first character is a digit

If you want to use different attributes, you can override them by providing an `accessors` argument. In this example, the table will use "Name" as the visible header, but internally, the attribute "character" will be used:

```
import toga

table = toga.Table(
    headings=["Name", "Age"],
    accessors={"Name", 'character'},
    data=[
        {"character": "Arthur Dent", "age": 42, "planet": "Earth"},
        {"character": "Ford Prefect", "age": 37, "planet": "Betelgeuse Five"},
        {"name": "Tricia McMillan", "age": 38, "planet": "Earth"},
    ]
)

# Get the details of the first item in the data:
row = table.data[0]
print(f"{row.character}, who is age {row.age}, is from {row.planet}")
```

The value provided by an accessor is interpreted as follows:

- If the value is a *Widget*, that widget will be displayed in the cell. Note that this is currently a beta API: see the Notes section.
- If the value is a `tuple`, it must have two elements: an icon, and a second element which will be interpreted as one of the options below.
- If the value is `None`, then `missing_value` will be displayed.
- Any other value will be converted into a string. If an icon has not already been provided in a tuple, it can also be provided using the value's `icon` attribute.

Icon values must either be an *Icon*, which will be displayed on the left of the cell, or `None` to display no icon.

Notes

- Widgets in cells is a beta API which may change in future, and is currently only supported on macOS.
- macOS does not support changing the font used to render table content.
- On Winforms, icons are only supported in the first column. On Android, icons are not supported at all.
- The Android implementation is not scalable beyond about 1,000 cells.

Reference

class toga.**Table**(*headings=None*, *id=None*, *style=None*, *data=None*, *accessors=None*, *multiple_select=False*, *on_select=None*, *on_activate=None*, *missing_value=''*, ***kwargs*)

Bases: *Widget*

Create a new Table widget.

Parameters

- **headings** (*Iterable[str] | None*) – The column headings for the table. Headings can only contain one line; any text after a newline will be ignored.

  A value of None will produce a table without headings. However, if you do this, you *must* give a list of accessors.

- **id** (*str | None*) – The ID for the widget.
- **style** (*StyleT | None*) – A style object. If no style is provided, a default style will be applied to the widget.
- **data** (*SourceT | Iterable | None*) – Initial *data* to be displayed in the table.
- **accessors** (*Iterable[str] | None*) – Defines the attributes of the data source that will be used to populate each column. Must be either:
  - None to derive accessors from the headings, as described above; or
  - A list of the same size as headings, specifying the accessors for each heading. A value of None will fall back to the default generated accessor; or
  - A dictionary mapping headings to accessors. Any missing headings will fall back to the default generated accessor.
- **multiple_select** (*bool*) – Does the table allow multiple selection?
- **on_select** (*toga.widgets.table.OnSelectHandler | None*) – Initial *on_select* handler.
- **on_activate** (*toga.widgets.table.OnActivateHandler | None*) – Initial *on_activate* handler.
- **missing_value** (*str*) – The string that will be used to populate a cell when the value provided by its accessor is None, or the accessor isn't defined.
- **kwargs** – Initial style properties.

property accessors: list[str]

The accessors used to populate the table (read-only)

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

Append a column to the end of the table.

Parameters

- **heading** (*str*) – The heading for the new column.
- **accessor** (*str | None*) – The accessor to use on the data source when populating the table. If not specified, an accessor will be derived from the heading.

Return type
None

property data: SourceT | *ListSource*

The data to display in the table.

When setting this property:

- A *Source* will be used as-is. It must either be a *ListSource*, or a custom class that provides the same methods.
- A value of None is turned into an empty ListSource.
- Otherwise, the value must be an iterable, which is copied into a new ListSource. Items are converted as shown *here*.

property enabled: Literal[True]

Is the widget currently enabled? i.e., can the user interact with the widget? Table widgets cannot be disabled; this property will always return True; any attempt to modify it will be ignored.

focus()

No-op; Table cannot accept input focus.

Return type
None

property headings: list[str] | None

The column headings for the table, or None if there are no headings (read-only)

insert_column(*index*, *heading*, *accessor=None*)

Insert an additional column into the table.

Parameters

- **index** (*int | str*) – The index at which to insert the column, or the accessor of the column before which the column should be inserted.
- **heading** (*str*) – The heading for the new column. If the table doesn't have headings, the value will be ignored.
- **accessor** (*str | None*) – The accessor to use on the data source when populating the table. If not specified, an accessor will be derived from the heading. An accessor *must* be specified if the table doesn't have headings.

Return type
None

property missing_value: str

The value that will be used when a data row doesn't provide a value for an attribute.

property multiple_select: bool

Does the table allow multiple rows to be selected?

property on_activate: *OnActivateHandler*

The callback function that is invoked when a row of the table is activated, usually with a double click or similar action.

property on_select: *OnSelectHandler*

The callback function that is invoked when a row of the table is selected.

remove_column(*column*)

Remove a table column.

Parameters
column (*int | str*) – The index of the column to remove, or the accessor of the column to remove.

Return type
None

`scroll_to_bottom()`

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

Return type
None

`scroll_to_row(row)`

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

Parameters
row (*int*) – 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).

Return type
None

`scroll_to_top()`

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

Return type
None

`property selection: list[Row] | Row | None`

The current selection of the table.

If multiple selection is enabled, returns a list of Row objects from the data source matching the current selection. An empty list is returned if no rows are selected.

If multiple selection is *not* enabled, returns the selected Row object, or `None` if no row is currently selected.

`protocol toga.widgets.table.OnSelectHandler`

`typing.Protocol`.

Classes that implement this protocol must have the following methods / attributes:

`__call__(widget, **kwargs)`

A handler to invoke when the table is selected.

Parameters
- **widget** (`Table`) – The Table that was selected.
- **kwargs** (*Any*) – Ensures compatibility with arguments added in future versions.

Return type
object

`protocol toga.widgets.table.OnActivateHandler`

`typing.Protocol`.

Classes that implement this protocol must have the following methods / attributes:

`__call__(widget, row, **kwargs)`

A handler to invoke when the table is activated.

Parameters

- **widget** (Table) – The Table that was activated.
- **row** (*Any*) – The Table Row that was activated.
- **kwargs** (*Any*) – Ensures compatibility with arguments added in future versions.

Return type
object

TextInput

A widget for the display and editing of a single line of text.

macOS

Linux

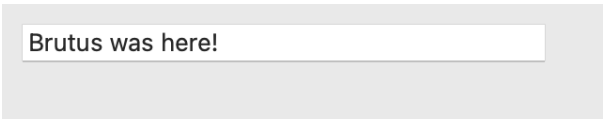
Windows

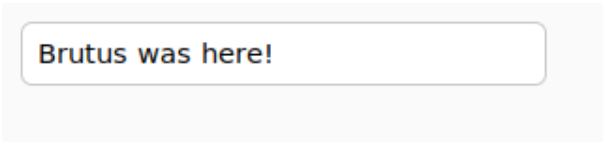
Android

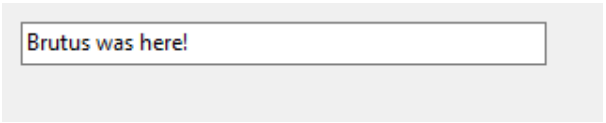
iOS

Web

Textual







Screenshot not available

Screenshot not available

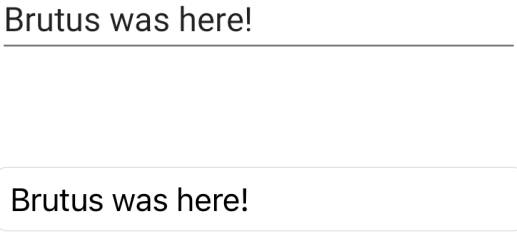
Usage

```
import toga

text_input = toga.TextInput()
text_input.value = "Jane Developer"
```

The input can be provided a placeholder value - this is a value that will be displayed to the user as a prompt for appropriate content for the widget. This placeholder will only be displayed if the widget has no content; as soon as a value is provided (either by the user, or programmatically), the placeholder content will be hidden.

The input can also be provided a list of *validators*. A validator is a function that will be invoked whenever the content of the input changes. The function should return `None` if the current value of the input is valid; if the current value is



invalid, it should return an error message. When `on_change` is invoked, the field will automatically be validated based on specified validators.

Notes

- Although an error message is provided when validation fails, Toga does not guarantee that this error message will be displayed to the user.
- Winforms does not support the use of partially or fully transparent colors for the TextInput background. If a color with an alpha value is provided (including `TRANSPARENT`), the alpha channel will be ignored. A `TRANSPARENT` background will be rendered as white.
- On Winforms, if a TextInput is given an explicit height, the rendered widget will not expand to fill that space. The widget will have the fixed height determined by the font used on the widget. In general, you should avoid setting a `height` style property on TextInput widgets.

Reference

class toga.**TextInput**(*id=None*, *style=None*, *value=None*, *readonly=False*, *placeholder=None*, *on_change=None*, *on_confirm=None*, *on_gain_focus=None*, *on_lose_focus=None*, *validators=None*, ***kwargs*)

Bases: *Widget*

Create a new single-line text input widget.

Parameters

- **id** (*str* | *None*) – The ID for the widget.
- **style** (*StyleT* | *None*) – A style object. If no style is provided, a default style will be applied to the widget.
- **value** (*str* | *None*) – The initial content to display in the widget.
- **readonly** (*bool*) – Can the value of the widget be modified by the user?
- **placeholder** (*str* | *None*) – The content to display as a placeholder when there is no user content to display.
- **on_change** (toga.widgets.textinput.OnChangeHandler | *None*) – A handler that will be invoked when the value of the widget changes.
- **on_confirm** (OnConfirmHandler | *None*) – A handler that will be invoked when the user accepts the value of the input (usually by pressing Return on the keyboard).
- **on_gain_focus** (OnGainFocusHandler | *None*) – A handler that will be invoked when the widget gains input focus.

- **on_lose_focus** (*OnLoseFocusHandler | None*) – A handler that will be invoked when the widget loses input focus.
- **validators** (*Iterable[Callable[[str], bool]] | None*) – A list of validators to run on the value of the input.
- **kwargs** – Initial style properties.

property is_valid: bool

Does the value of the widget currently pass all validators without error?

property on_change: *OnChangeHandler*

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

property on_confirm: *OnConfirmHandler*

The handler to invoke when the user accepts the value of the widget, usually by pressing return/enter on the keyboard.

property on_gain_focus: *OnGainFocusHandler*

The handler to invoke when the widget gains input focus.

property on_lose_focus: *OnLoseFocusHandler*

The handler to invoke when the widget loses input focus.

property placeholder: str

The placeholder text for the widget.

A value of `None` will be interpreted and returned as an empty string. Any other object will be converted to a string using `str()`.

property readonly: bool

Can the value of the widget be modified by the user?

This only controls manual changes by the user (i.e., typing at the keyboard). Programmatic changes are permitted while the widget has `readonly` enabled.

property validators: list[Callable[[str], bool]]

The list of validators being used to check input on the widget.

Changing the list of validators will cause validation to be performed.

property value: str

The text to display in the widget.

A value of `None` will be interpreted and returned as an empty string. Any other object will be converted to a string using `str()`.

Any newline (`\n`) characters in the string will be replaced with a space.

Validation will be performed as a result of changing widget value.

protocol toga.widgets.textinput.**OnChangeHandler**

typing.Protocol.

Classes that implement this protocol must have the following methods / attributes:

__call__(*widget*, ***kwargs*)

A handler to invoke when the text input is changed.

Parameters

- **widget** (TextInput) – The TextInput that was changed.

- **kwargs** (*Any*) – Ensures compatibility with arguments added in future versions.

Return type
object

protocol toga.widgets.textinput.**OnConfirmHandler**

typing.Protocol.

Classes that implement this protocol must have the following methods / attributes:

__call__(*widget*, ***kwargs*)

A handler to invoke when the text input is confirmed.

Parameters

- **widget** (TextInput) – The TextInput that was confirmed.
- **kwargs** (*Any*) – Ensures compatibility with arguments added in future versions.

Return type
object

protocol toga.widgets.textinput.**OnGainFocusHandler**

typing.Protocol.

Classes that implement this protocol must have the following methods / attributes:

__call__(*widget*, ***kwargs*)

A handler to invoke when the text input gains focus.

Parameters

- **widget** (TextInput) – The TextInput that gained focus.
- **kwargs** (*Any*) – Ensures compatibility with arguments added in future versions.

Return type
object

protocol toga.widgets.textinput.**OnLoseFocusHandler**

typing.Protocol.

Classes that implement this protocol must have the following methods / attributes:

__call__(*widget*, ***kwargs*)

A handler to invoke when the text input loses focus.

Parameters

- **widget** (TextInput) – The TextInput that lost focus.
- **kwargs** (*Any*) – Ensures compatibility with arguments added in future versions.

Return type
object

TimeInput

A widget to select a clock time.

macOS ×

Linux ×

Windows

Android

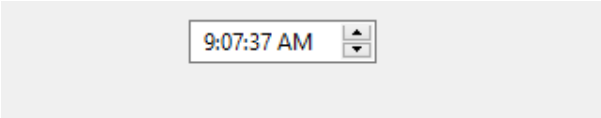
iOS ×

Web ×

Textual ×

Not supported

Not supported



Not supported

Not supported

Not supported

Usage

```
import toga

current_time = toga.TimeInput()
```

Notes

- This widget supports hours, minutes and seconds. Microseconds will always be returned as zero.
  - On Android, seconds will also be returned as zero.
- Properties that return `datetime.time` objects can also accept:
  - `datetime.datetime`: The time portion will be extracted.
  - `str`: Will be parsed as an ISO8601 format time string (e.g., “06:12”).

Reference

class toga.**TimeInput**(*id=None*, *style=None*, *value=None*, *min=None*, *max=None*, *on_change=None*, ***kwargs*)

Bases: *Widget*

Create a new TimeInput widget.

Parameters

- **id** (*str* | *None*) – The ID for the widget.

- **style** (*StyleT | None*) – A style object. If no style is provided, a default style will be applied to the widget.
- **value** (*datetime.time | None*) – The initial time to display. If not specified, the current time will be used.
- **min** (*datetime.time | None*) – The earliest time (inclusive) that can be selected.
- **max** (*datetime.time | None*) – The latest time (inclusive) that can be selected.
- **on_change** (toga.widgets.timeinput.OnChangeHandler | *None*) – A handler that will be invoked when the value changes.
- **kwargs** – Initial style properties.

property max: time

The maximum allowable time (inclusive). A value of None will be converted into 23:59:59.

When setting this property, the current *value* and *min* will be clipped against the new maximum value.

property min: time

The minimum allowable time (inclusive). A value of None will be converted into 00:00:00.

When setting this property, the current *value* and *max* will be clipped against the new minimum value.

property on_change: *OnChangeHandler*

The handler to invoke when the time value changes.

property value: time

The currently selected time. A value of None will be converted into the current time.

If this property is set to a value outside of the min/max range, it will be clipped.

protocol toga.widgets.timeinput.**OnChangeHandler**

typing.Protocol.

Classes that implement this protocol must have the following methods / attributes:

__call__(*widget*, ***kwargs*)

A handler to invoke when the time input is changed.

Parameters

- **widget** (TimeInput) – The TimeInput that was changed.
- **kwargs** (*Any*) – Ensures compatibility with arguments added in future versions.

Return type
object

Tree

A widget for displaying a hierarchical tree of tabular data. Scroll bars will be provided if necessary.

macOS

Linux

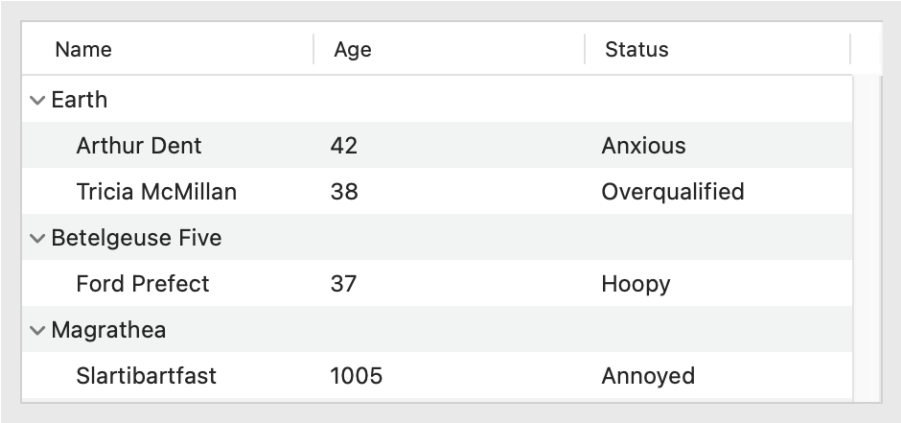
Windows ×

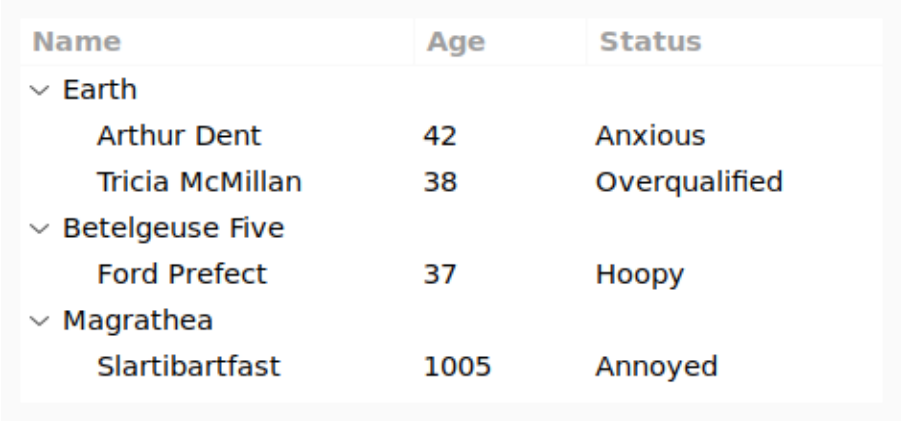
Android ×

iOS ×

Web ×

Textual ×





Not supported

Not supported

Not supported

Not supported

Not supported

Usage

The simplest way to create a Tree is to pass a dictionary and a list of column headings. Each key in the dictionary can be either a tuple, whose contents will be mapped sequentially to the columns of a node, or a single object, which will be mapped to the first column. And each value in the dictionary can be either another dictionary containing the children of that node, or `None` if there are no children.

In this example, we will display a tree with 2 columns. The tree will have 2 root nodes; the first root node will have 1 child node; the second root node will have 2 children. The root nodes will only populate the "name" column; the other column will be blank:

```
import toga

tree = toga.Tree(
    headings=["Name", "Age"],
```

(continues on next page)

(continued from previous page)

```
    data={
        "Earth": {
            ("Arthur Dent", 42): None,
        },
        "Betelgeuse Five": {
            ("Ford Prefect", 37): None,
            ("Zaphod Beeblebrox", 47): None,
        },
    }
)

# Get the details of the first child of the second root node:
print(f"{tree.data[1][0].name} is age {tree.data[1][0].age}")

# Append new data to the first root node in the tree
tree.data[0].append(("Tricia McMillan", 38))
```

You can also specify data for a Tree using a list of 2-tuples, with dictionaries providing data values. This allows you to store data in the data source that won't be displayed in the tree. It also allows you to control the display order of columns independent of the storage of that data.

```
import toga

tree = toga.Tree(
    headings=["Name", "Age"],
    data=[
        (
            {"name": "Earth"},
            [({"name": "Arthur Dent", "age": 42, "status": "Anxious"}, None)]
        ),
        (
            {"name": "Betelgeuse Five"},
            [
                ({"name": "Ford Prefect", "age": 37, "status": "Hoopy"}, None),
                ({"name": "Zaphod Beeblebrox", "age": 47, "status": "Oblivious"}, None),
            ]
        ),
    ]
)

# Get the details of the first child of the second root node:
node = tree.data[1][0]
print(f"{node.name}, who is age {node.age}, is {node.status}")
```

The attribute names used on each row (called "accessors") are created automatically from the headings, by:

1. Converting the heading to lower case
2. Removing any character that can't be used in a Python identifier
3. Replacing all whitespace with _
4. Prepending _ if the first character is a digit

If you want to use different attributes, you can override them by providing an `accessors` argument. In this example,

the tree will use "Name" as the visible header, but internally, the attribute "character" will be used:

```
import toga

tree = toga.Tree(
    headings=["Name", "Age"],
    accessors={"Name", 'character'},
    data=[
        (
            {"character": "Earth"},
            [({"character": "Arthur Dent", "age": 42, "status": "Anxious"}, None)]
        ),
        (
            {"character": "Betelgeuse Five"},
            [
                ({"character": "Ford Prefect", "age": 37, "status": "Hoopy"}, None),
                ({"character": "Zaphod Beeblebrox", "age": 47, "status": "Oblivious"},␣
↪None),
            ]
        ),
    ]
)

# Get the details of the first child of the second root node:
node = tree.data[1][0]
print(f"{node.character}, who is age {node.age}, is {node.status}")
```

The value provided by an accessor is interpreted as follows:

- If the value is a *Widget*, that widget will be displayed in the cell. Note that this is currently a beta API: see the Notes section.
- If the value is a `tuple`, it must have two elements: an icon, and a second element which will be interpreted as one of the options below.
- If the value is `None`, then `missing_value` will be displayed.
- Any other value will be converted into a string. If an icon has not already been provided in a tuple, it can also be provided using the value's `icon` attribute.

Icon values must either be an *Icon*, which will be displayed on the left of the cell, or `None` to display no icon.

Notes

- Widgets in cells is a beta API which may change in future, and is currently only supported on macOS.
- On macOS, you cannot change the font used in a Tree.

Reference

class toga.**Tree**(*headings=None*, *id=None*, *style=None*, *data=None*, *accessors=None*, *multiple_select=False*, *on_select=None*, *on_activate=None*, *missing_value=''*, ***kwargs*)

Bases: *Widget*

Create a new Tree widget.

Parameters

- **headings** (*Iterable[str] | None*) – The column headings for the tree. Headings can only contain one line; any text after a newline will be ignored.

  A value of None will produce a table without headings. However, if you do this, you *must* give a list of accessors.

- **id** (*str | None*) – The ID for the widget.
- **style** (*Pack | None*) – A style object. If no style is provided, a default style will be applied to the widget.
- **data** (*SourceT | object | None*) – Initial *data* to be displayed in the tree.
- **accessors** (*Iterable[str] | None*) – Defines the attributes of the data source that will be used to populate each column. Must be either:
  - None to derive accessors from the headings, as described above; or
  - A list of the same size as headings, specifying the accessors for each heading. A value of None will fall back to the default generated accessor; or
  - A dictionary mapping headings to accessors. Any missing headings will fall back to the default generated accessor.
- **multiple_select** (*bool*) – Does the tree allow multiple selection?
- **on_select** (toga.widgets.tree.OnSelectHandler | *None*) – Initial *on_select* handler.
- **on_activate** (toga.widgets.tree.OnActivateHandler | *None*) – Initial *on_activate* handler.
- **missing_value** (*str*) – The string that will be used to populate a cell when the value provided by its accessor is None, or the accessor isn't defined.
- **kwargs** – Initial style properties.

property accessors: list[str]

The accessors used to populate the tree (read-only)

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

Append a column to the end of the tree.

Parameters

- **heading** (*str*) – The heading for the new column.
- **accessor** (*str | None*) – The accessor to use on the data source when populating the tree. If not specified, an accessor will be derived from the heading.

Return type
None

collapse(*node=None*)

Collapse the specified node of the tree.

If no node is provided, all nodes of the tree will be collapsed.

If the provided node is a leaf node, or the node is already collapsed, this is a no-op.

Parameters
node (Node | *None*) – The node to collapse

Return type
None

property data: SourceT | *TreeSource*

The data to display in the tree.

When setting this property:

- A *Source* will be used as-is. It must either be a *TreeSource*, or a custom class that provides the same methods.
- A value of None is turned into an empty TreeSource.
- Otherwise, the value must be a dictionary or an iterable, which is copied into a new TreeSource as shown *here*.

property enabled: Literal[True]

Is the widget currently enabled? i.e., can the user interact with the widget? Tree widgets cannot be disabled; this property will always return True; any attempt to modify it will be ignored.

expand(*node=None*)

Expand the specified node of the tree.

If no node is provided, all nodes of the tree will be expanded.

If the provided node is a leaf node, or the node is already expanded, this is a no-op.

If a node is specified, the children of that node will also be expanded.

Parameters
node (Node | *None*) – The node to expand

Return type
None

focus()

No-op; Tree cannot accept input focus.

Return type
None

property headings: list[str] | None

The column headings for the tree (read-only)

insert_column(*index*, *heading*, *accessor=None*)

Insert an additional column into the tree.

Parameters

- **index** (*int* | *str*) – The index at which to insert the column, or the accessor of the column before which the column should be inserted.
- **heading** (*str*) – The heading for the new column. If the tree doesn't have headings, the value will be ignored.
- **accessor** (*str* | *None*) – The accessor to use on the data source when populating the tree. If not specified, an accessor will be derived from the heading. An accessor *must* be specified if the tree doesn't have headings.

Return type
None

property missing_value: str

The value that will be used when a data row doesn't provide a value for an attribute.

property multiple_select: *bool*

Does the tree allow multiple rows to be selected?

property on_activate: *OnActivateHandler*

The callback function that is invoked when a row of the tree is activated, usually with a double click or similar action.

property on_select: *OnSelectHandler*

The callback function that is invoked when a row of the tree is selected.

remove_column(*column*)

Remove a tree column.

Parameters

column (*int* | *str*) – The index of the column to remove, or the accessor of the column to remove.

Return type

None

property selection: *list[Node] | Node | None*

The current selection of the tree.

If multiple selection is enabled, returns a list of Node objects from the data source matching the current selection. An empty list is returned if no nodes are selected.

If multiple selection is *not* enabled, returns the selected Node object, or `None` if no node is currently selected.

protocol toga.widgets.tree.**OnSelectHandler**

typing.Protocol.

Classes that implement this protocol must have the following methods / attributes:

__call__(*widget*, ***kwargs*)

A handler to invoke when the tree is selected.

Parameters

- **widget** (Tree) – The Tree that was selected.
- **kwargs** (*Any*) – Ensures compatibility with arguments added in future versions.

Return type

object

protocol toga.widgets.tree.**OnActivateHandler**

typing.Protocol.

Classes that implement this protocol must have the following methods / attributes:

__call__(*widget*, ***kwargs*)

A handler to invoke when the tree is activated.

Parameters

- **widget** (Tree) – The Tree that was activated.
- **kwargs** (*Any*) – Ensures compatibility with arguments added in future versions.

Return type

object

WebView

An embedded web browser.

macOS

Linux

Windows

Android

iOS

Web ×

Textual ×


BeeWare


BeeWare

Not supported

Not supported

Usage

```
import toga

webview = toga.WebView()

# Request a URL be loaded in the webview.
webview.url = "https://beeware.org"

# Load a URL, and wait (non-blocking) for the page to complete loading
await webview.load_url("https://beeware.org")

# Load static HTML content into the wevbiew.
webview.set_content("https://example.com", "<html>...</html>")
```

System requirements

- Using WebView on Windows 10 requires that your users have installed the Edge WebView2 Evergreen Runtime. This is installed by default on Windows 11.
- Using WebView on Linux requires that the user has installed the system packages for WebKit2, plus the GObject Introspection bindings for WebKit2. The name of the system package required is distribution dependent:
  - Ubuntu 20.04; Debian 11: `gir1.2-webkit2-4.0`
  - Ubuntu 22.04+; Debian 12+: `gir1.2-webkit2-4.1`

- Fedora: `webkit2gtk4.1`
- Arch/Manjaro: `webkit2gtk-4.1`
- OpenSUSE Tumbleweed: `libwebkit2gtk3 typelib(WebKit2)`
- FreeBSD: `webkit2-gtk3`

WebView is not fully supported on GTK4. If you want to contribute to the GTK4 WebView implementation, you will require v6.0 of the WebKit2 libraries. This is provided by `gir1.2-webkit-6.0` on Ubuntu/Debian, and `webkitgtk6.0` on Fedora; for other distributions, consult your distributions's platform documentation.

Notes

- Due to app security restrictions, WebView can only display `http://` and `https://` URLs, not `file://` URLs. To serve local file content, run a web server on `localhost` using a background thread.
- On macOS 13.3 (Ventura) and later, the content inspector for your app can be opened by running Safari, enabling the developer tools, and selecting your app's window from the "Develop" menu.

  On macOS versions prior to Ventura, the content inspector is not enabled by default, and is only available when your code is packaged as a full macOS app (e.g., with Briefcase). To enable debugging, run:

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

  Substituting `com.example.appname` with the bundle ID for your packaged app.

Reference

class toga.**WebView**(*id=None*, *style=None*, *url=None*, *user_agent=None*, *on_webview_load=None*, ***kwargs*)

Bases: *Widget*

Create a new WebView widget.

Parameters

- **id** (*str | None*) – The ID for the widget.
- **style** (*StyleT | None*) – A style object. If no style is provided, a default style will be applied to the widget.
- **url** (*str | None*) – The full URL to load in the WebView. If not provided, an empty page will be displayed.
- **user_agent** (*str | None*) – The user agent to use for web requests. If not provided, the default user agent for the platform will be used.
- **on_webview_load** (*OnWebViewLoadHandler | None*) – A handler that will be invoked when the web view finishes loading.
- **kwargs** – Initial style properties.

property cookies: CookiesResult

Retrieve cookies from the WebView.

This is an asynchronous property. The value returned by this method must be awaited to obtain the cookies that are currently set.

Note: This property is not currently supported on Android or Linux.

Returns

An object that returns a CookieJar when awaited.

`evaluate_javascript`(*javascript*, *on_result=None*)

Evaluate a JavaScript expression.

This is an asynchronous method. There is no guarantee that the JavaScript has finished evaluating when this method returns. The object returned by this method can be awaited to obtain the value of the expression.

Note: On Android and Windows, *no exception handling is performed*. If a JavaScript error occurs, a return value of None will be reported, but no exception will be provided.

Parameters

- **javascript** (*str*) – The JavaScript expression to evaluate.
- **on_result** (*OnResultT | None*) – **DEPRECATED** `await` the return value of this method.

Return type
JavaScriptResult

`async load_url`(*url*)

Load a URL, and wait until the next *on_webview_load* event.

Note: On Android, this method will return immediately.

Parameters
url (*str*) – The URL to load.

Return type
Future

`property on_webview_load: OnWebViewLoadHandler`

The handler to invoke when the web view finishes loading.

Rendering web content is a complex, multi-threaded process. Although a page may have completed *loading*, there's no guarantee that the page has been fully *rendered*, or that the widget representation has been fully updated. The number of load events generated by a URL transition or content change can be unpredictable. An `on_webview_load` event should be interpreted as an indication that some change has occurred, not that a *specific* change has occurred, or that a specific change has been fully propagated into the rendered content.

Note: This is not currently supported on Android.

`set_content`(*root_url*, *content*)

Set the HTML content of the WebView.

Note: On Android and Windows, the `root_url` argument is ignored. Calling this method will set the `url` property to `None`.

Parameters

- **root_url** (*str*) – A URL which will be returned by the `url` property, and used to resolve any relative URLs in the content.
- **content** (*str*) – The HTML content for the WebView

Return type
None

`property url: str | None`

The current URL, or `None` if no URL is currently displayed.

After setting this property, it is not guaranteed that reading the property will immediately return the new value. To be notified once the URL has finished loading, use *load_url* or *on_webview_load*.

property user_agent: str

The user agent to use for web requests.

Note: On Windows, this property will return an empty string until the widget has finished initializing.

protocol toga.widgets.webview.OnWebViewLoadHandler

typing.Protocol.

Classes that implement this protocol must have the following methods / attributes:

__call__(*widget*, ***kwargs*)

A handler to invoke when the WebView is loaded.

Parameters

- **widget** (WebView) – The WebView that was loaded.
- **kwargs** (*Any*) – Ensures compatibility with arguments added in future versions.

Return type
object

Widget

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

Table 13: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web	Terminal

Reference

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

Bases: Node, *PackMixin*

Create a base Toga widget.

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

Parameters

- **id** (*str | None*) – The ID for the widget.
- **style** (*StyleT | None*) – A style object. If no style is provided, a default style will be applied to the widget.
- **kwargs** – Initial style properties.

add(**children*)

Add the provided widgets as children of this widget.

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

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

Raises
ValueError – If this widget cannot have children.

Return type
None

property app: *App* | None

The App to which this widget belongs.

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

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

property applicator

This node's applicator, which handles applying the style.

Assigning an applicator triggers an application of the node's style.

property can_have_children

Determine if the node can have children.

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

property children

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

Returns:
A list of the children for this widget.

clear()

Remove all child widgets of this node.

Refreshes the widget after removal if any children were removed.

Raises
ValueError – If this widget cannot have children.

Return type
None

property enabled: bool

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

focus()

Give this widget the input focus.

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

Return type
None

property id: str

A unique identifier for the widget (read-only).

index(*child*)

Get the index of a widget in the list of children of this widget.

Parameters
child (Widget) – The child widget of interest.

Raises
ValueError – If the specified child widget is not found in the list of children.

Returns
Index of specified child widget in children list.

Return type
int

insert(*index*, *child*)

Insert a widget as a child of this widget.

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

Parameters

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

Raises
ValueError – If this widget cannot have children.

Return type
None

property parent

The parent of this node.

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

refresh()

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

Return type
None

remove(**children*)

Remove the provided widgets as children of this node.

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

Refreshes the widget after removal if any children were removed.

Parameters
children (Widget) – The child nodes to remove.

Raises
ValueError – If this widget cannot have children.

Return type
None

replace(*old_child*, *new_child*)

Replace an existing child widget with a new child widget.

Parameters

- **old_child** (Widget) – The existing child widget to be replaced.
- **new_child** (Widget) – The new child widget to be included.

Return type
None

property root

The root of the tree containing this node.

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

property style

The node's style.

Assigning a style triggers an application of that style if an applicator has already been assigned.

property tab_index: int | None

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

> **Note**
>
> This is a beta feature. The `tab_index` API may change in the future.

property window: *Window* | None

The window to which this widget belongs.

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

If the widget has a value for *window*, it *must* also have a value for *app*.

class toga.widgets.base.**PackMixin**

Allows accessing the Pack *style properties* directly on the widget. For example, instead of `widget.style.color`, you can simply write `widget.color`.

Constants

class toga.constants.**Direction**(*value*, *names=_not_given*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

The direction a given property should act.

HORIZONTAL = 0

VERTICAL = 1

class toga.constants.**Baseline**(*value*, *names=_not_given*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

The meaning of a Y coordinate when drawing text.

ALPHABETIC = 1

Alphabetic baseline of the first line

TOP = 2

Top of text

MIDDLE = 3

Middle of text

BOTTOM = 4

Bottom of text

class toga.constants.**FillRule**(*value*, *names=_not_given*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

The rule to use when filling paths.

EVENODD = 0

NONZERO = 1

class toga.constants.**FlashMode**(*value*, *names=_not_given*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

The flash mode to use when capturing photos or videos.

AUTO = -1

OFF = 0

ON = 1

class toga.constants.**WindowState**(*value*, *names=_not_given*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

The possible window states of an app.

NOTE: Some platforms do not fully support all states; see the *toga.Window*'s platform notes for details.

NORMAL = 0

The NORMAL state represents the default state of the window or app when it is not in any other specific window state.

MINIMIZED = 1

MINIMIZED state is when the window isn't currently visible, although it will appear in any operating system's list of active windows.

MAXIMIZED = 2

The window is the largest size it can be on the screen with title bar and window chrome still visible.

FULLSCREEN = 3

FULLSCREEN state is when the window title bar and window chrome remain hidden; But app menu and toolbars remain visible.

PRESENTATION = 4

PRESENTATION state is when the window title bar, window chrome, app menu and toolbars all remain hidden.

A good example is a slideshow app in presentation mode - the only visible content is the slide.

Keys

A symbolic representation of keys used for keyboard shortcuts.

Most keys have a constant that matches the text on the key, or the name of the key if the text on the key isn't a legal Python identifier.

However, due to differences between platforms, there's no representation of "modifier" keys like Control, Command, Option, or the Windows Key. Instead, Toga provides three generic modifier constants, and maps those to the modifier keys, matching the precedence with which they are used on the underlying platforms:

Platform	*MOD_1*	*MOD_2*	*MOD_3*
Linux	Control	Alt	Tux/Windows/Meta
macOS	Command ()	Option	Control (^)
Windows	Control	Alt	Not supported

Key combinations can be expressed by combining multiple `Key` values with the + operator.

```
from toga import Key

just_an_a = Key.A
shift_a = Key.SHIFT + Key.A
# Windows/Linux - Control-Shift-A:
# macOS - Command-Shift-A:
modified_shift_a = Key.MOD_1 + Key.SHIFT + Key.A
```

The order of addition is not significant. `Key.SHIFT + Key.A` and `Key.A + Key.SHIFT` will produce the same key representation.

Reference

class toga.**Key**(*value*, *names=_not_given*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

An enumeration providing a symbolic representation for the characters on a keyboard.

A = 'a'

AMPERSAND = '&'

ASTERISK = '*'

AT = '@'

B = 'b'

BACKSLASH = '\\'

BACKSPACE = '<backspace>'

BACK_QUOTE = '`'

BEGIN = '<begin>'

C = 'c'

```
CAPSLOCK = '<caps lock>'

CARET = '^'

CLOSE_BRACE = '}'

CLOSE_BRACKET = ']'

CLOSE_PARENTHESIS = ')'

COLON = ':'

COMMA = ','

D = 'd'

DELETE = '<delete>'

DOLLAR = '$'

DOUBLE_QUOTE = '"'

DOWN = '<down>'

E = 'e'

EJECT = '<eject>'

END = '<end>'

ENTER = '<enter>'

EQUAL = '='

ESCAPE = '<esc>'

EXCLAMATION = '!'

F = 'f'

F1 = '<f1>'

F10 = '<f10>'

F11 = '<f11>'

F12 = '<f12>'

F13 = '<f13>'

F14 = '<f14>'

F15 = '<f15>'

F16 = '<f16>'

F17 = '<f17>'

F18 = '<f18>'

F19 = '<f19>'
```

```
    F2 = '<f2>'
    F3 = '<f3>'
    F4 = '<f4>'
    F5 = '<f5>'
    F6 = '<f6>'
    F7 = '<f7>'
    F8 = '<f8>'
    F9 = '<f9>'
    FULL_STOP = '.'
    G = 'g'
    GREATER_THAN = '>'
    H = 'h'
    HASH = '#'
    HOME = '<home>'
    I = 'i'
    INSERT = '<insert>'
    J = 'j'
    K = 'k'
    L = 'l'
    LEFT = '<left>'
    LESS_THAN = '<'
    M = 'm'
    MENU = '<menu>'
    MINUS = '-'
    MOD_1 = '<mod 1>'
    MOD_2 = '<mod 2>'
    MOD_3 = '<mod 3>'
    N = 'n'
    NUMLOCK = '<num lock>'
    NUMPAD_0 = 'numpad:0'
    NUMPAD_1 = 'numpad:1'
```

```
NUMPAD_2 = 'numpad:2'

NUMPAD_3 = 'numpad:3'

NUMPAD_4 = 'numpad:4'

NUMPAD_5 = 'numpad:5'

NUMPAD_6 = 'numpad:6'

NUMPAD_7 = 'numpad:7'

NUMPAD_8 = 'numpad:8'

NUMPAD_9 = 'numpad:9'

NUMPAD_CLEAR = 'numpad:clear'

NUMPAD_DECIMAL_POINT = 'numpad:.'

NUMPAD_DIVIDE = 'numpad:/'

NUMPAD_ENTER = 'numpad:enter'

NUMPAD_EQUAL = 'numpad:='

NUMPAD_MINUS = 'numpad:-'

NUMPAD_MULTIPLY = 'numpad:*'

NUMPAD_PLUS = 'numpad:+'

O = 'o'

OPEN_BRACE = '{'

OPEN_BRACKET = '['

OPEN_PARENTHESIS = '('

P = 'p'

PAGE_DOWN = '<pg dn>'

PAGE_UP = '<pg up>'

PAUSE = '<pause>'

PERCENT = '%'

PIPE = '|'

PLUS = '+'

Q = 'q'

QUESTION = '?'

QUOTE = "'"

R = 'r'
```

RIGHT = '<right>'

S = 's'

SCROLLLOCK = '<scroll lock>'

SEMICOLON = ';'

SHIFT = '<shift>'

SLASH = '/'

SPACE = ' '

T = 't'

TAB = '<tab>'

TILDE = '~'

U = 'u'

UNDERSCORE = '_'

UP = '<up>'

V = 'v'

W = 'w'

X = 'x'

Y = 'y'

Z = 'z'

is_printable()

Does pressing the key result in a printable character?

Return type
bool

Types

namedtuple toga.LatLng(*lat*, *lng*)

A geographic coordinate.

Fields

0) **lat** (float) – Latitude
1) **lng** (float) – Longitude

namedtuple toga.Position(*x*, *y*)

A 2D position.

Fields

0) **x** (int) – X coordinate, in CSS pixels.
1) **y** (int) – Y coordinate, in CSS pixels.

namedtuple toga.**Size**(*width*, *height*)

A 2D size.

Fields

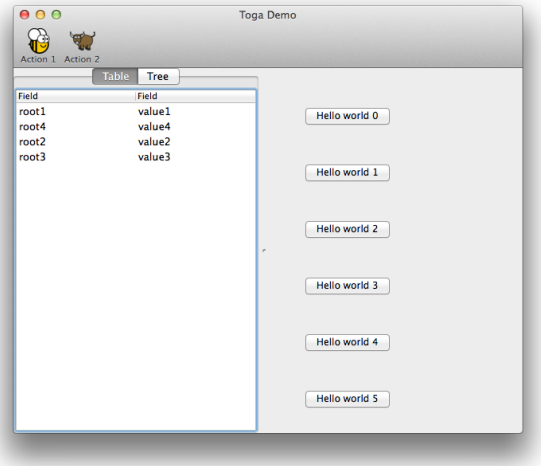
0) **width** (int) – Width, in CSS pixels.

1) **height** (int) – Height, in CSS pixels.

4.2.2 Supported platforms

Desktop

macOS



The Toga backend for macOS is toga-cocoa.

Prerequisites

toga-cocoa requires macOS 11 (Big Sur) or newer.

Installation

toga-cocoa is installed automatically on macOS machines (machines that report `sys.platform == 'darwin'`), or can be manually installed by running invoking:

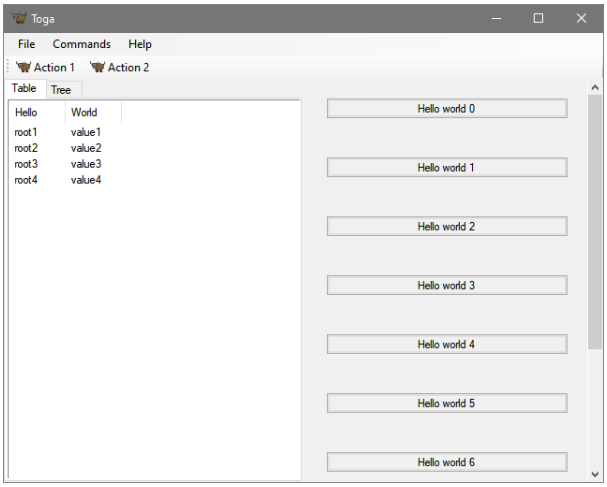
```
$ python -m pip install toga-cocoa
```

Implementation details

The toga-cocoa backend uses the AppKit Objective-C API, also known as Cocoa.

The native APIs are accessed using Rubicon Objective C.

Windows



The Toga backend for Windows is toga-winforms.

Prerequisites

`toga-winforms` requires Windows 10 or newer.

If you are using Windows 10 and want to use a WebView to display web content, you will also need to install the Edge WebView2 Evergreen Runtime. Windows 11 has this runtime installed by default.

Installation

`toga-winforms` is installed automatically on Windows machines (machines that report `sys.platform == 'win32'`), or can be manually installed by running:

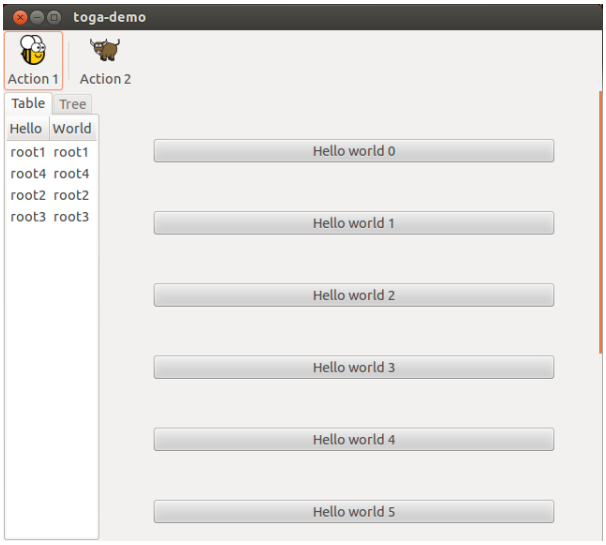
```
$ python -m pip install toga-winforms
```

Implementation details

The `toga-winforms` backend uses the Windows Forms API.

The native .NET APIs are accessed using Python.NET.

Linux/Unix



The Toga backend for Linux (and other Unix-like operating systems) is toga-gtk.

Qt support

Toga does not currently have a Qt backend for KDE-based desktops. However, we would like to add one; see this ticket for details. If you would like to contribute, please get in touch on that ticket, on Mastodon or on Discord.

GTK on Windows and macOS

Although GTK *can* be installed on Windows and macOS, and the `toga-gtk` backend *may* work on those platforms, this is not officially supported by Toga. We recommend using `toga-winforms` on *Windows*, and `toga-cocoa` on *macOS*.

Prerequisites

`toga-gtk` requires GTK 3.22 or newer. Most testing occurs with GTK 3.24 as this is the version that has shipped with all versions of Ubuntu since Ubuntu 20.04, and all versions of Fedora since Fedora 32.

The system packages that provide GTK must be installed manually:

These instructions are different on almost every version of Linux and Unix; here are some of the common alternatives:

Ubuntu / Debian

```
(venv) $ sudo apt update
(venv) $ sudo apt install git build-essential pkg-config python3-dev libgirepository1.0-
↪dev libcairo2-dev gir1.2-gtk-3.0 libcanberra-gtk3-module
```

Fedora

```
(venv) $ sudo dnf install git gcc make pkg-config python3-devel gobject-introspection-
↪devel cairo-gobject-devel gtk3 libcanberra-gtk3
```

Arch / Manjaro

```
(venv) $ sudo pacman -Syu git base-devel pkgconf python3 gobject-introspection cairo␣
↪gtk3 libcanberra
```

OpenSUSE Tumbleweed

```
(venv) $ sudo zypper install git patterns-devel-base-devel_basis pkgconf-pkg-config␣
↪python3-devel gobject-introspection-devel cairo-devel gtk3 'typelib(Gtk)=3.0'␣
↪libcanberra-gtk3-module
```

FreeBSD

```
(venv) $ sudo pkg update
(venv) $ sudo pkg install git gcc cmake pkgconf python3 gobject-introspection cairo gtk3␣
↪libcanberra-gtk3
```

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

In addition to the dependencies above, if you would like to help add additional support for GTK4, you need to also install `gir1.2-gtk-4.0` on Ubuntu/Debian, or `gtk4` on Fedora or Arch. For other distributions, consult your distributions's platform documentation.

Some widgets (most notably, the *WebView* and *MapView* widgets) have additional system requirements. Likewise, certain hardware features (*Location*) have system requirements.

See the documentation of those widgets and hardware features for details.

Toga does not currently support GTK 4.

Installation

`toga-gtk` is installed automatically on any Linux machine (machines that report `sys.platform == 'linux'`), or any FreeBSD machine (machines that report `sys.platform == 'freebsd*'`). It can be manually installed by running:

```
$ python -m pip install toga-gtk
```

Implementation details

The `toga-gtk` backend uses the GTK3 API.

The native APIs are accessed using the PyGObject binding.

Mobile

Android

The Toga backend for Android is toga-android.

Prerequisites

`toga-android` requires Android SDK 24 (Android 7 / Nougat) or newer.

Installation

`toga-android` must be manually installed into an Android project; The recommended approach for deploying `toga-android` is to use Briefcase to package your app.

Implementation details

The `toga-android` backend uses the Android Java API, with Material3 widgets. It uses Chaquopy to provide a bridge to the native Android Java libraries and implement Java interfaces from Python.

Platform-specific APIs

Activities and Intents

On Android, some interactions are managed using Activities, which are started using Intents.

Android's implementation of the *toga.App* class includes the method *start_activity()*, which can be used to start an activity.

`toga_android.App.`**`start_activity`**(*self*, *activity*, **options*, *on_complete=None*)

Start a native Android activity.

Parameters

- **activity** – The `android.content.Intent` instance to start
- **options** – Any additional arguments to pass to the native `android.app.Activity.startActivityForResult()` call.
- **on_complete** – A callback to invoke when the activity completes. The callback will be invoked with 2 arguments: the result code, and the result data.

To use this method, instantiate an instance of `android.content.Intent`; optionally, provide additional arguments, and a callback that will be invoked when the activity completes. For example, to dial a phone number with the `Intent.ACTION_DIAL` intent:

```
from android.content import Intent
from android.net import Uri

intent = Intent(Intent.ACTION_DIAL)
intent.setData(Uri.parse("tel:0123456789"))

def number_dialed(result, data):
    # result is the status code (e.g., Activity.RESULT_OK)
    # data is the value returned by the activity.
    ...

# Assuming your toga.App app instance is called `app`
app._impl.start_activity(intent, on_complete=number_dialed)
```

iOS

The Toga backend for iOS is toga-iOS.

Prerequisites

`toga-iOS` requires iOS 12 or newer.

Installation

`toga-iOS` must be manually installed into an iOS project; The recommended approach for deploying `toga-iOS` is to use Briefcase to package your app.

Implementation details

The `toga-iOS` backend uses the UIKit Objective C API.

The native APIs are accessed using Rubicon Objective C.

Other

Web

Toga is able to deploy apps as a single-page web app using the toga-web backend.

> **Note**
>
> The Web backend is currently proof-of-concept only. Most widgets have not been implemented.

Prerequisites

`toga-web` will run in any modern browser. It requires PyScript 2023.05.01 or newer, and Shoelace v2.3.

Installation

The recommended approach for deploying `toga-web` is to use Briefcase to package your app.

`toga-web` can be installed manually by adding `toga-web` to your `pyscript.toml` configuration file.

Implementation details

The `toga-web` backend is implemented using Shoelace web components.

The DOM is accessed using PyScript.

Terminal

The Toga backend for terminal applications is toga-textual.

Prerequisites

`toga-textual` should run on any terminal or command shell provided by macOS, Windows or Linux.

Installation

`toga-textual` must be manually installed by running:

```
$ python -m pip install toga-textual
```

If `toga-textual` is the only Toga backend that is installed, it will be picked up automatically on any desktop operating system. If you have another backend installed (usually, this will be the default GUI for your operating system), you will need to set the `TOGA_BACKEND` environment variable to `toga_textual` to force the selection of the backend.

Implementation details

The `toga-textual` backend uses the Textual API.

macOS Terminal.app limitations

There are some known issues with the default macOS Terminal.app. In some layouts, box outlines render badly; this can *sometimes* be resolved by altering the line spacing of the font used in the terminal. The default Terminal.app also has a limited color palette. The maintainers of Textual recommend using an alternative terminal application to avoid these problems.

Testing

Toga provides a toga-dummy backend that can be used for testing purposes. This backend implements the full interface required by a platform backend, but does not display any widgets visually. It provides an API that can be used to verify widget operation.

Prerequisites

The dummy backend has no prerequisites.

Installation

The dummy backend must be installed manually:

```
$ python -m pip install toga-dummy
```

To force Toga to use the dummy backend, it must either be the only backend that is installed in the current Python environment, or you must define the `TOGA_BACKEND` environment variable:

macOS

Linux

Windows

```
(venv) $ export TOGA_BACKEND=toga_dummy
```

```
(venv) $ export TOGA_BACKEND=toga_dummy
```

```
(venv) $ set TOGA_BACKEND=toga_dummy
```

Future Plans

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

- WinUI (for Modern Windows look and feel)
- Qt (for KDE-based Unix desktops)
- tvOS (for AppleTV devices)
- watchOS (for Apple Watch devices)

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

Unofficial support

At present, there are no known unofficial platform backends.

4.2.3 Toga APIs by platform

Key

Partly supported: functionality or testing is incomplete
Fully supported

Core Components

Component	macOS	GTK	Windows	iOS	Android	Web	Terminal
App							
Window							
DocumentWindow							
MainWindow							

General Widgets

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

Layout Widgets

Component	macOS	GTK	Windows	iOS	Android	Web	Terminal
Box							
ScrollContainer							
SplitContainer							
OptionContainer							

Hardware

Component	macOS	GTK	Windows	iOS	Android	Web	Terminal
Camera							
Location							
Screen							

Resources

Component	macOS	GTK	Windows	iOS	Android	Web	Terminal
Paths							
Command							
Dialogs							
Document							
Font							
Icon							
Image							
ListSource							
Source							
Status Icons							
TreeSource							
Validators							
ValueSource							

4.2.4 Style

The Pack Style Engine

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

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

Note

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

Pack style properties

display

Values: `pack` | `none`

Initial value: `pack`

Used to define how to display the widget. A value of `pack` will apply the pack layout algorithm to this node and its descendants. A value of `none` removes the widget from the layout entirely. Space will be allocated for the widget as if it were there, but the widget itself will not be visible.

visibility

Values: `hidden` | `visible`

Initial value: `visible`

Used to define whether the widget should be drawn. A value of `visible` means the widget will be displayed. A value of `hidden` removes the widget from view, but allocates space for the widget as if it were still in the layout.

Any children of a hidden widget are implicitly removed from view.

If a previously hidden widget is made visible, any children of the widget with a visibility of `hidden` will remain hidden. Any descendants of the hidden child will also remain hidden, regardless of their visibility.

direction

Values: `row` | `column`

Initial value: `row`

The packing direction for children of the box. A value of `column` indicates children will be stacked vertically, from top to bottom. A value of `row` indicates children will be packed horizontally; left-to-right if `text_direction` is `ltr`, or right-to-left if `text_direction` is `rtl`.

align_items

Values: `start` | `center` | `end`

Initial value: `start`

Aliases: `vertical_align_items` in a row, `horizontal_align_items` in a column

The alignment of this box's children along the cross axis. A row's cross axis is vertical, so `start` aligns children to the top, while `end` aligns them to the bottom. For columns, `start` is on the left if `text_direction` is `ltr`, and the right if `rtl`.

justify_content

Values: `start` | `center` | `end`

Initial value: `start`

Aliases: `horizontal_align_content` in a row, `vertical_align_content` in a column

The alignment of this box's children along the main axis. A column's main axis is vertical, so `start` aligns children to the top, while `end` aligns them to the bottom. For rows, `start` is on the left if `text_direction` is `ltr`, and the right if `rtl`.

This property only has an effect if there is some free space in the main axis. For example, if any children have a non-zero `flex` value, then they will consume all the available space, and `justify_content` will make no difference to the layout.

gap

Values: `<integer>`

Initial value: `0`

The amount of space to allocate between adjacent children, in *CSS pixels*.

width

Values: `<integer>` | `none`

Initial value: `none`

Specify a fixed width for the box, in *CSS pixels*.

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, in *CSS pixels*.

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.

margin_top

margin_right

margin_bottom

margin_left

Values: `<integer>`

Initial value: `0`

The amount of space to allocate outside the edge of the box, in *CSS pixels*.

margin

Values: <integer> or <tuple> of length 1-4

A shorthand for setting the top, right, bottom and left margin with a single declaration.

If 1 integer is provided, that value will be used as the margin for all sides.

If 2 integers are provided, the first value will be used as the margin 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 margin, the second for the left and right margin, and the third for the bottom margin.

If 4 integers are provided, they will be used as the top, right, bottom and left margin, 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).

Any other value will be checked against the family names previously registered with *Font.register*. If the name cannot be resolved, the system font will be used.

font_style

Values: normal | italic | oblique

Initial value: normal

The style of the font to be used.

Note: Windows and Android do not support the oblique font style. A request for an oblique font will be interpreted as italic.

font_variant

Values: normal | small_caps

Initial value: normal

The variant of the font to be used.

Note: Windows and Android do not support the small caps variant. A request for a small_caps font will be interpreted as normal.

font_weight

Values: normal | bold

Initial value: normal

The weight of the font to be used.

font_size

Values: <integer>

Initial value: System default

The size of the font to be used, in *CSS points*.

The relationship between Pack and CSS

Pack aims to be a functional subset of CSS. Any Pack layout can be converted into an equivalent CSS layout. After applying this conversion, the CSS layout should be considered a "reference implementation". Any disagreement between the rendering of a converted Pack layout in a browser, and the layout produced by the Toga implementation of Pack should be considered to be either a bug in Toga, or a bug in the mapping.

The mapping that can be used to establish the reference implementation is:

- The reference HTML layout document is rendered in no-quirks mode, with a default CSS stylesheet:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Pack layout testbed</title>
    <style>
      html, body {
```

(continues on next page)

(continued from previous page)

```
            height: 100%;
        }
        body {
            overflow: hidden;
            display: flex;
            margin: 0;
            white-space: pre;
        }
        div {
            display: flex;
            white-space: pre;
        }
    </style>
  </head>
  <body></body>
</html>
```

- The root widget of the Pack layout can be mapped to the <body> element of the HTML reference document. The rendering area of the browser window becomes the view area that Pack will fill.
- ImageViews map to <img> elements. The <img> element has an additional style of `object-fit: contain` unless *both* `height` and `width` are defined.
- All other widgets are mapped to elements.
- The following Pack declarations can be mapped to equivalent CSS declarations:

Pack property	CSS property
`direction: <str>`	`flex-direction: <str>`
`display: pack`	`display: flex`
`flex: <int>`	If `direction = row` and `width` is set, or `direction = column` and `height` is set, ignore. Otherwise, `flex: <int>` `0 auto`.
`font_size: <int>`	`font-size: <int>pt`
`height: <value>`	`height: <value>px` if value is an integer; `height: auto` if value is none.
`margin_top: <int>`	`margin-top: <int>px`
`margin_bottom: <int>`	`margin-bottom: <int>px`
`margin_left: <int>`	`margin-left: <int>px`
`margin_right: <int>`	`margin-right: <int>px`
`text_direction <str>`	`direction: <str>`
`width: <value>`	`width: <value>px` if value is an integer; `width: auto` if value is none.

- All other Pack declarations should be used as-is as CSS declarations, with underscores being converted to dashes (e.g., `background_color` becomes `background-color`).

4.2.5 Plugins

Image Format Plugins

Usage

Toga can be extended, via plugins, to understand externally defined image types, gaining the ability to convert them to and from its own *Image* class. Toga's Pillow support is, in fact, implemented as a plugin that's included as part of the core Toga package.

An image format plugin consists of two things:

- a converter class conforming to the *ImageConverter* protocol, with methods defining how to convert to and from your image class
- an entry point in the `toga.image_formats` group telling Toga the path to your converter class.

Let's say you want to tell Toga how to handle an image class called `MyImage`, and you're publishing your plugin as a package named `togax-myimage` (see *package prefixes*) that contains a `plugins.py` module that defines your `MyImageConverter` plugin class. Your `pyproject.toml` might include something like the following:

```
[project.entry-points."toga.image_formats"]
myimage = "togax_myimage.plugins.MyImageConverter"
```

The variable name being assigned to (`myimage` in this case) can be whatever you like (although it should probably have some relationship to the image format name) What matters is the string assigned to it, which represents where Toga can find (and import) your *ImageConverter* class.

Package prefixes

An image plugin can be registered from any Python module. If you maintain a package defining an image format, you could include a Toga converter plugin along with it. If you're publishing a plugin as a standalone package, you should title it with a `togax-` prefix, to indicate that it's an unofficial extension for Toga. Do *not* use the `toga-` prefix, as the BeeWare Project wishes to reserve that package prefix for "official" packages.

Reference

type **ExternalImageT**

Any class that represents an image.

protocol toga.images.**ImageConverter**

A class to convert between an externally defined image type and *toga.Image*.

Classes that implement this protocol must have the following methods / attributes:

static convert_from_format(*image_in_format*)

Convert from *image_class* to data in a *known image format*.

Will accept an instance of *image_class*, or subclass of that class.

Parameters
image_in_format (*ExternalImageT*) – An instance of *image_class* (or a subclass).

Returns
The image data, in a *known image format*.

Return type
BytesLikeT

static convert_to_format(*data*, *image_class*)

Convert from data to *image_class* or specified subclass.

Accepts a bytes-like object representing the image in a *known image format*, and returns an instance of the image class specified. This image class is guaranteed to be either the *image_class* registered by the plugin, or a subclass of that class.

Parameters

- **data** (*BytesLikeT*) – Image data in a *known image format*.
- **image_class** (*type[ExternalImageT]*) – The class of image to return.

Returns
The image, as an instance of the image class specified.

Return type
ExternalImageT

image_class: type[ExternalImageT]

The base image class this plugin can interpret.

4.3 How-to Guides

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

4.3.1 Topic guides

API design

Toga's API is structured around the following principles:

Coding style

The public API follows a "Pythonic" style, using Python language idioms (e.g., context managers and iterators) and naming conventions (e.g., `snake_case`, not `CamelCase`), even if the underlying platforms don't lean that way.

Names are spelled according to US English.

Properties

Wherever possible, Toga exposes an object's state using property notation (e.g. `widget.property`) rather than getter or setter methods.

Properties follow Postel's Law – for example, a widget's `text` property will accept any object when set, but will always return a string when when retrieved.

Constructors

Any of a class's writable properties can be initialized in its constructor by passing keyword arguments with the same names. The constructor may also accept read-only properties such as *Widget.id*, which cannot be changed later.

If a constructor has a single required argument, such as the text of a *Label*, it may be passed as a positional argument.

Events

Events are used to notify your app of user actions. To make your app handle an event, you can assign either a regular or async callable to an event handler property. These can be identified by their names, which always begin with `on_`.

Events are named for the general purpose of the interaction, not the specific mechanism. For example, a *Button*'s event is called `on_press`, not `on_click`, because "click" implies a mouse is used.

When the event occurs, your handler will be passed the widget as a positional argument, and other event-specific information as keyword arguments. For forward compatibility with arguments added in the future, handlers should always declare a `**kwargs` argument.

If an event is triggered by a change in a property:

- The new value of the property will be visible within the event handler.
- Setting the property programmatically will also generate an event, unless the property is set to its existing value, in which case whether it generates an event is undefined.

Common names

When a widget allows the user to control a simple value (e.g. the `str` of a *TextInput*, or the `bool` of a *Switch*), then its property is called `value`, and the corresponding event is called `on_change`.

When a widget has a non-editable caption, (e.g. a *Button* or *Switch*), then its property is called `text`.

Ranges of numbers are expressed as separate `min` and `max` properties.

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 window needs to perform a layout, the layout engine asks the style object for the container to lay out its contents within the space that the container has available. This will calculate a size and position for all the widgets in the tree.

Every window has a container representing the total viewable area of the window. However, some widgets (those with "Container" in their name) establish sub-containers. When a refresh is requested on a container, any sub-containers will also be refreshed.

Length units

Toga uses CSS units in its public API. Their physical size depends on the device type:

- A CSS pixel is about 1/96 of an inch (0.26 mm) on a desktop screen, and about 1/160 of an inch (0.16 mm) on a phone screen.

- A CSS point is 1.33 CSS pixels.

Toga only uses points to measure font sizes. All other lengths are expressed as pixels.

For a full explanation of CSS units, see this article.

Implementation notes

- On macOS and iOS, one CSS pixel equals one "point", which is 1, 2 or 3 linear physical pixels, depending on the device.
- On Windows, one CSS pixel equals one physical pixel at 100% scale, and is adjusted as necessary at higher scale factors.
- On Android, one CSS pixel equals one dp.

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 storage and manipulation of 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 through the use of data sources. Instead of directly telling a widget to display a particular value (or collection of values), you should define a **data source**, and then attach a widget to that source. The data source is responsible for tracking the data that is in the source; the widget responds to those changes in the data, providing an appropriate visualization.

Built-in data sources

There are three built-in data source types in Toga:

- *Value Sources*: For managing a single value. A ValueSource has a single attribute, (by default, `value`), which is what 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`*.

Although Toga provides these built-in data sources, in general, *you shouldn't use them directly*. Toga's data sources are wrappers around Python's primitive collection types - `list`, `dict`, and so on. While this is useful for quick demonstrations, or to visualize simple data, more complex applications should define their own *custom data sources*.

Listeners

Data sources communicate using a *`Listener`* interface. When any significant event occurs to the data source, all listeners will be notified. This includes:

- Adding a new item
- Removing an existing item
- Changing an attribute of an existing item
- Clearing an entire data source

If any attribute of a *ValueSource*, *Row* or *Node* is modified, the source will generate a change event.

When you create a widget like Selection or Table, and provide a data source for that widget, the widget is automatically added as a listener on that source.

Although widgets are the obvious listeners for a data source, *any* object can register as a listener. For example, a second data source might register as a listener to an initial source to implement a filtered source. When an item is added to the first data source, the second data source will be notified, and can choose whether to include the new item in it's own data representation.

Custom 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 file system. However, you might have a `create_empty_file()` method that creates a new file in the file system and adds a representation to the data 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. Toga provides a *Source* base class for custom data source implementations. This base class implements the notification API.

4.3.2 Contributing to Toga

Toga is an open source project, and actively encourages community contributions. The following guides will help you get started contributing to Toga.

Contributing code to Toga

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

Set up your development environment

First, ensure that you have Python 3 and pip installed. To do this, run:

macOS

Linux

Windows

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

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

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

Create a virtual environment

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

macOS

Linux

Windows

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

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

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

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

Install system dependencies

Next, install any additional dependencies for your operating system:

macOS

Linux

Windows

No additional dependencies

These instructions are different on almost every version of Linux and Unix; here are some of the common alternatives:

Ubuntu / Debian

```
(venv) $ sudo apt update
(venv) $ sudo apt install git build-essential pkg-config python3-dev libgirepository1.0-
↪dev libcairo2-dev gir1.2-gtk-3.0 libcanberra-gtk3-module
```

Fedora

```
(venv) $ sudo dnf install git gcc make pkg-config python3-devel gobject-introspection-
↪devel cairo-gobject-devel gtk3 libcanberra-gtk3
```

Arch / Manjaro

```
(venv) $ sudo pacman -Syu git base-devel pkgconf python3 gobject-introspection cairo␣
↪gtk3 libcanberra
```

OpenSUSE Tumbleweed

```
(venv) $ sudo zypper install git patterns-devel-base-devel_basis pkgconf-pkg-config␣
↪python3-devel gobject-introspection-devel cairo-devel gtk3 'typelib(Gtk)=3.0'␣
↪libcanberra-gtk3-module
```

FreeBSD

```
(venv) $ sudo pkg update
(venv) $ sudo pkg install git gcc cmake pkgconf python3 gobject-introspection cairo gtk3␣
↪libcanberra-gtk3
```

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

In addition to the dependencies above, if you would like to help add additional support for GTK4, you need to also install `gir1.2-gtk-4.0` on Ubuntu/Debian, or `gtk4` on Fedora or Arch. For other distributions, consult your distributions's platform documentation.

Some widgets (most notably, the *WebView* and *MapView* widgets) have additional system requirements. Likewise, certain hardware features (*Location*) have system requirements.

See the documentation of those widgets and hardware features for details.

No additional dependencies

Clone the Toga repository

Next, go to the Toga page on GitHub, 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

Linux

Windows

Fork the Toga repository, and then:

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

(substituting your GitHub username)

Fork the Toga repository, and then:

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

(substituting your GitHub username)

Fork the Toga repository, and then:

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

(substituting your GitHub username)

Install Toga

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 resolve the dependencies to source packages, so we have to manually install each package:

macOS

Linux

Windows

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

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

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

Pre-commit automatically runs during the commit

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

macOS

Linux

Windows

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

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

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

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

macOS

Linux

Windows

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

reformatted some/interesting_file.py

All done!
1 file reformatted.

flake8...................................................................Passed
check toml...........................................(no files to check)Skipped
check yaml...........................................(no files to check)Skipped
check for case conflicts.................................................Passed
```

(continues on next page)

(continued from previous page)

```
check docstring is first.....................................................Passed
fix end of files.............................................................Passed
trim trailing whitespace.....................................................Passed
isort........................................................................Passed
pyupgrade....................................................................Passed
docformatter.................................................................Passed
```

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

reformatted some/interesting_file.py

All done!
1 file reformatted.

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

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

reformatted some\interesting_file.py

All done!
1 file reformatted.

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

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

macOS

Linux

Windows

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

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

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

Now you are ready to start hacking on Toga!

What should I do?

Depending on your level of expertise, or areas of interest, there are a number of ways you can contribute to Toga's code.

Fix a bug in an existing widget

Toga's issue tracker logs the known issues with existing widgets. Any of these issues are candidates to be worked on. This list can be filtered by platform, so you can focus on issues that affect the platforms you're able to test on. There's also a filter for good first issues . These have been identified as problems that have a known cause, and we believe the fix *should* be relatively simple (although we might be wrong in our analysis).

We don't have any formal process of "claiming" or "assigning" issues; if you're interested in a ticket, leave a comment that says you're working on it. If there's an existing comment that says someone is working on the issue, and that comment is recent, then leave a comment asking if they're still working on the issue. If you don't get a response in a day or two, you can assume the issue is available. If the most recent comment is more than a few weeks old, it's probably safe to assume that the issue is still available to be worked on.

If an issue is particularly old (more than 6 months), it's entirely possible that the issue has been resolved, so the first step is to verify that you can reproduce the problem. Use the information provided in the bug report to try and reproduce the problem. If you can't reproduce the problem, report what you have found as a comment on the ticket, and pick another ticket.

If a bug report has no comments from anyone other than the original reporter, the issue needs to be triaged. Triaging a bug involves taking the information provided by the reporter, and trying to reproduce it. Again, if you can't reproduce the problem, report what you have found as a comment on the ticket, and pick another ticket.

If you can reproduce the problem - try to fix it! Work out what combination of core and backend-specific code is implementing the feature, and see if you can work out what isn't working correctly. You may need to refer to platform specific documentation (e.g., the Cocoa AppKit, iOS UIKit, GTK, Winforms, Android, Shoelace or Textual API documentation) to work out why a widget isn't behaving as expected.

If you're able to fix the problem, you'll need to add tests for *the core API* and/or *the testbed backend* for that widget, depending on whether the fix was in the core API or to the backend (or both).

Even if you can't fix the problem, reporting anything you discover as a comment on the ticket is worthwhile. If you can find the source of the problem, but not the fix, that knowledge will often be enough for someone who knows more about a platform to solve the problem. Even a good reproduction case (a sample app that does nothing but reproduce the problem) can be a huge help.

Contribute improvements to documentation

We've got a *separate contribution guide* for documentation contributions. This covers how to set up your development environment to build Toga's documentation, and separate ideas for what to work on.

Implement a platform native widget

If the core library already specifies an interface for a widget, but the widget isn't implemented on your platform of choice, implement that interface. The *supported widgets by platform* table can show you the widgets that are missing on various platforms. You can also look for log messages in a running app (or the direct `factory.not_implemented()` function calls that produce those log messages). At present, the Web and Textual backends have the most missing widgets. If you have web skills, or would like to learn more about PyScript and Shoelace, the web backend could be a good place to contribute; if you'd like to learn more about terminal applications and the or Textual API, contributing to the Textual backend could be a good place for you to contribute.

Alternatively, if there's a widget that doesn't exist, propose an interface design, and implement it for at least one platform. You may find this presentation by BeeWare emeritus team member Dan Yeaw helpful. This talk gives an architectural overview of Toga, as well as providing a guide to the process of adding new widgets.

If you implement a new widget, don't forget you'll need to write tests for the new core API. If you're extending an existing widget, you may need to *add a probe for the backend*.

Add a new feature to an existing widget

Can you think of a feature than an existing widget should have? Propose a new API for that widget, and provide a sample implementation. If you don't have any ideas of your own, the Toga issue tracker has some existing feature suggestions that you could try to implement.

Again, you'll need to add unit tests and/or backend probes for any new features you add.

Contribute to the GTK4 update

Toga's GTK support is currently based on the GTK3 API. This API works, and ships with most Linux distributions, but is no longer maintained by the GTK team. We're in the process of adding GTK4 support to Toga's GTK backend. You can help with this update process.

GTK4 support can be enabled by setting the `TOGA_GTK=4` environment variable. To contribute to the update, pick a widget that currently has GTK3 support, and try updating the widget's API to support GTK4 as well. You can identify a widget that hasn't been ported by looking at the *GTK probe for the widget* - widgets that aren't ported yet will have an "if GTK4, skip" block at the top of the probe definition.

The code needs to support both GTK3 and GTK4; if there are significant differences in API, you can add conditional branches based on the GTK version. See one of the widgets that *has* been ported (e.g., Label) for examples of how this can be done.

Implement an entirely new platform backend

Toga currently has support for 7 backends - but there's room for more! In particular, we'd be interested in seeing a Qt-based backend to support KDE-based Linux desktops.

The first steps of any new platform backend are always the same:

1. Implement enough of the Toga Application and Window classes to allow you to create an empty application window, integrated with the Python `asyncio` event loop.
2. Work out how to use native platform APIs to position a widget at a specific position on the window. Most widget frameworks will have some sort of native layout scheme; we need to replace that scheme with Toga's layout algorithm. If you can work out how to place a button with a fixed size at a specific position on the screen, that's usually sufficient.
3. Get Tutorial 0 working. This is the simple case of a single box that contains a single button. To get this tutorial working, you'll need to implement the factory class for your new backend so that Toga can instantiate widgets on your new backend, and connect the Toga style applicator methods on the base widget that sets the position of widgets on the screen.

Once you have those core features in place, you can start implementing widgets and other Toga features (like fonts, images, and so on).

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.

Running the core test suites

Toga uses tox to manage the testing process.

Testing Core

To run the core test suite:

macOS

Linux

Windows

```
(venv) $ tox -m test-core
```

```
(venv) $ tox -m test-core
```

```
(venv) C:\...>tox -m test-core
```

You should get some output indicating that tests have been run. You may see `SKIPPED` tests, but 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!

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

```
Name    Stmts   Miss Branch BrPart   Cover   Missing
-----------------------------------------------------
TOTAL    4345      0   1040      0  100.0%
```

This tells us that the test suite has executed every possible branching path in the `toga-core` library. This isn't a 100% guarantee that there are no bugs, but it does mean that we're exercising every line of code in the core API.

If you make changes to the core API, it's possible you'll introduce a gap in this coverage. When this happens, the coverage report will tell you which lines aren't being executed. For example, lets say we made a change to `toga/window.py`, adding some new logic. The coverage report might look something like:

```
Name                  Stmts   Miss Branch BrPart  Cover   Missing
-----------------------------------------------------------------
src/toga/window.py      186      2     22      2  98.1%   211, 238-240
-----------------------------------------------------------------
TOTAL                  4345      2   1040      2  99.9%
```

This tells us that line 211, and lines 238-240 are not being executed by the test suite. You'll need to add new tests (or modify an existing test) to restore this coverage.

Testing Travertino

In addition to the core library, the Toga repository also includes Travertino, a package that defines the lower-level layout mechanisms and style definitions which core then builds on. Its test suite can be run just like that of core:

macOS

Linux

Windows

```
(venv) $ tox -m test-trav
```

```
(venv) $ tox -m test-trav
```

```
(venv) C:\...>tox -m test-trav
```

Just as with core, this should report 100% test coverage.

You can run both the core and Travertino tests with one command:

macOS

Linux

Windows

```
(venv) $ tox -m test
```

```
(venv) $ tox -m test
```

```
(venv) C:\...>tox -m test
```

This will run both test suites, and report the two coverage results one after the other.

Run a subset of tests

When you're developing your new test, it may be helpful to run *just* that one test. To do this, you can pass in the name of a specific test file (or a specific test, using pytest specifiers):

macOS

Linux

Windows

```
(venv) $ tox -e py -- tests/path_to_test_file/test_some_test.py
```

```
(venv) $ tox -e py -- tests/path_to_test_file/test_some_test.py
```

```
(venv) C:\...>tox -e py -- tests/path_to_test_file/test_some_test.py
```

These test paths are relative to the `core` directory. To run a Travertino test instead, add `-trav`:

macOS

Linux

Windows

```
(venv) $ tox -e py-trav -- tests/path_to_test_file/test_some_test.py
```

```
(venv) $ tox -e py-trav -- tests/path_to_test_file/test_some_test.py
```

```
(venv) C:\...>tox -e py-trav -- tests/path_to_test_file/test_some_test.py
```

Either way, you'll still get a coverage report when running a part of the test suite - but the coverage results will only report the lines of code that were executed by the specific tests you ran.

Running the test suites for multiple Python versions

Tox can also run the test suites for all supported version of Python. This requires that each version of Python is available from `Path`.

macOS

Linux

Windows

```
(venv) $ tox
```

```
(venv) $ tox
```

```
(venv) C:\...>tox
```

Running CI checks

Tox can also be used to run many of the same checks that run in CI; this is most useful prior to committing and pushing your changes.

macOS

Linux

Windows

```
(venv) $ tox -m ci
```

```
(venv) $ tox -m ci
```

```
(venv) C:\...>tox -m ci
```

Running the testbed

The above test suites exercise `toga-core` and `travertino` - but what about the backends? To verify the behavior of the backends, Toga has a testbed app. This app uses the core API to exercise all the behaviors that the backend APIs need to perform - but uses an actual platform backend to implement that behavior.

To run the testbed app, install Briefcase, and run the app in developer test mode:

macOS

Linux

Windows

```
(venv) $ python -m pip install briefcase
(venv) $ cd testbed
(venv) $ briefcase dev --test
```

```
(venv) $ python -m pip install briefcase
(venv) $ cd testbed
(venv) $ briefcase dev --test
```

```
(venv) C:\...>python -m pip install briefcase
(venv) C:\...>cd testbed
(venv) C:\...>briefcase dev --test
```

This will display a Toga app window, which will flash as it performs all the GUI tests. You'll then see a coverage report for the code that has been executed.

If you want to run a subset of the entire test suite, Briefcase honors pytest specifiers) in the same way as the main test suite.

The testbed app provides one additional feature that the core tests don't have – slow mode. Slow mode runs the same tests, but deliberately pauses for 1 second between each GUI action so that you can observe what is going on.

So - to run *only* the button tests in slow mode, you could run:

macOS

Linux

Windows

```
(venv) $ briefcase dev --test -- tests/widgets/test_button.py --slow
```

```
(venv) $ briefcase dev --test -- tests/widgets/test_button.py --slow
```

```
(venv) C:\...>briefcase dev --test -- tests/widgets/test_button.py --slow
```

This test will take a lot longer to run, but you'll see the widget (Button, in this case) go through various color, format, and size changes as the test runs. You won't get a coverage report if you run a subset of the tests, or if you enable slow mode.

Developer mode is useful for testing desktop platforms (Cocoa, Winforms and GTK); but if you want to test a mobile backend, you'll need to use `briefcase run`.

macOS

Linux

Windows

To run the Android test suite:

```
(venv) $ briefcase run android --test
```

To run the iOS test suite:

```
(venv) $ briefcase run iOS --test
```

To run the Android test suite:

```
(venv) $ briefcase run android --test
```

iOS tests can't be executed on Linux.

To run the Android test suite:

```
(venv) C:\...>briefcase run android --test
```

iOS tests can't be executed on Windows.

You can also use slow mode or pytest specifiers with `briefcase run`, using the same `--` syntax as you used in developer mode.

Finally, if you would like to run the tests against GTK4 on Linux, set the environmental variable `TOGA_GTK=4`. This is experimental and only partially implemented, but we would greatly appreciate your help translating widgets from GTK3 to GTK4.

How the testbed works

The testbed works by providing a generic collection of behavioral tests on a live app, and then providing an API to instrument the live app to verify that those behaviors have been implemented. That API is then implemented by each backend.

The implementation of the generic behavioral tests is contained in the tests folder of the testbed app. These tests use the public API of a widget to exercise all the corner cases of each implementation. Some of the tests are generic (for example, setting the background color of a widget) and are shared between widgets, but each widget has its own set of specific tests. These tests are all declared `async` because they need to interact with the event loop of a running application.

Each test will make a series of calls on a widget's public API. The public API is used to verify the behavior that an end user would experience when programming a Toga app. The test will *also* make calls on the *probe* for the widget.

The widget probe provides a generic interface for interacting with the internals of widget, verifying that the implementation is in the correct state as a result of invoking a public API. The probes for each platform are implemented in the `tests_backend` folder of each backend. For example, the Cocoa tests backend and probe implementations can be found here.

The probe for each widget provides a way to manipulate and inspect the internals of a widget in a way that may not be possible from a public API. For example, the Toga public API doesn't provide a way to determine the physical size of a widget, or interrogate the font being used to render a widget; the probe implementation does. This allows a testbed test case to verify that a widget has been laid out correctly inside the Toga window, is drawn using the right font, and has any other other appropriate physical properties or internal state.

The probe also provides a programmatic interface for interacting *with* a widget. For example, in order to test a button, you need to be able to press that button; the probe API provides an API to simulate that press. This allows the testbed to verify that the correct callbacks will be invoked when a button is pressed. These interactions are performed by generating events in the GUI framework being tested.

The widget probe also provides a `redraw()` method. GUI libraries don't always immediately apply changes visually, as graphical changes will often be batched so that they can be applied in a single redraw. To ensure that any visual changes have been applied before a test asserts the properties of the app, a test case can call `await probe.redraw()`. This guarantees that any outstanding redraw events have been processed. These `redraw()` requests are also used to implement slow mode - each redraw is turned into a 1 second sleep.

If a widget doesn't have a probe for a given widget, the testbed should call `pytest.skip()` for that platform when constructing the widget fixture (there is a `skip_on_platforms()` helper method in the testbed method to do this). If a widget hasn't implemented a specific probe method that the testbed required, it should call `pytest.skip()` so that the backend knows to skip the test.

If a widget on a given backend doesn't support a given feature, it should use `pytest.xfail()` (expected failure) for the probe method testing that feature. For example, Cocoa doesn't support setting the text color of buttons; as a result, the Cocoa implementation of the `color` property of the Button probe performs an `xfail` describing that limitation.

Submitting a pull request

Before you submit a pull request, there's a few bits of housekeeping to do.

Submit from a feature branch, not your `main` branch

Before you start working on your change, make sure you've created a branch. By default, when you clone your repository fork, you'll be checked out on your `main` branch. This is a direct copy of Toga's `main` branch.

While you *can* submit a pull request from your `main` branch, it's preferable if you *don't* do this. If you submit a pull request that is *almost* right, the core team member who reviews your pull request may be able to make the necessary changes, rather than giving feedback asking for a minor change. However, if you submit your pull request from your `main` branch, reviewers are prevented from making modifications.

Instead, you should make your changes on a *feature branch*. A feature branch has a simple name to identify the change that you've made. For example, if you've found a bug in Toga's layout algorithm, you might create a feature branch `fix-layout-bug`. If your bug relates to a specific issue that has been reported, it's also common to reference that issue number in the branch name (e.g., `fix-1234`).

To create a feature branch, run:

macOS

Linux

Windows

```
(venv) $ git checkout -b fix-layout-bug
```

```
(venv) $ git checkout -b fix-layout-bug
```

```
(venv) C:\...>git checkout -b fix-layout-bug
```

Commit your changes to this branch, then push to GitHub and create a pull request.

Add change information for release notes

Before you submit this change as a pull request, you need to add a *change note*. Toga uses towncrier to automate building the release notes for each release. Every pull request must include at least one file in the `changes/` directory that provides a short description of the change implemented by the pull request.

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

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

The corresponding change note would read something like:

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

The change note should be in ReST format, in a file that has name of the format `<id>.<fragment type>.rst`. If the change you are proposing will fix a bug or implement a feature for which there is an existing issue number, the ID will be the number of that ticket. If the change has no corresponding issue, the PR number can be used as the ID. You won't know this PR number until you push the pull request, so the first CI pass will fail the Towncrier check; add the change note and push a PR update and CI should then pass.

There are five allowed fragment types:

- `feature`: The PR adds a new behavior or capability that wasn't previously possible (e.g., adding a new widget, or adding a significant capability to an existing widget);
- `bugfix`: The PR fixes a bug in the existing implementation;
- `doc`: The PR is an significant improvement to documentation;
- `removal`; The PR represents a backwards incompatible change in the Toga API; or
- `misc`; A minor or administrative change (e.g., fixing a typo, a minor language clarification, or updating a dependency version) that doesn't need to be announced in the release notes.

Some PRs will introduce multiple features and fix multiple bugs, or introduce multiple backwards incompatible changes. In that case, the PR may have multiple change note files. If you need to associate two fragment types with the same ID, you can append a numerical suffix. For example, if PR 789 added a feature described by ticket 123, closed a bug described by ticket 234, and also made two backwards incompatible changes, you might have 4 change note files:

- `123.feature.rst`
- `234.bugfix.rst`
- `789.removal.1.rst`
- `789.removal.2.rst`

For more information about Towncrier and fragment types see News Fragments. You can also see existing examples of news fragments in the `changes` directory of the Toga repository. If this folder is empty, it's likely because Toga has recently published a new release; change note files are deleted and combined to update the *release notes* with each release. You can look at that file to see the style of comment that is required; you can look at recently merged PRs to see how to format your change notes.

It's not just about coverage!

Although we're always trying to improve test coverage, the task isn't *just* about increasing the numerical coverage value. 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.

Waiting for feedback

Once you've written your code, test, and change note, you can submit your changes as a pull request. One of the core team will review your work, and give feedback. If any changes are requested, you can make those changes, and update your pull request; eventually, the pull request will be accepted and merged. Congratulations, you're a contributor to Toga!

What next?

Rinse and repeat! If you've improved coverage by one line, go back and do it again for *another* coverage line! If you've implemented a new widget, implement *another* widget!

Most importantly - have fun!

Contributing to Toga's documentation

You might have the best software in the world - but if nobody knows how to use it, what's the point? Documentation can always be improved - and we need need your help!

Toga's documentation is written using Sphinx and reStructuredText. We aim to follow the Diataxis framework for structuring documentation.

Building Toga's documentation

To build Toga's documentation, start by *setting up a development environment*. You **must** have a Python 3.12 interpreter installed and available on your path (i.e., `python3.12` must start a Python 3.12 interpreter).

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

macOS

Linux

Windows

Enchant can be installed using Homebrew:

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

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

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

Enchant can be installed as a system package:

Ubuntu / Debian

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

Fedora

```
$ sudo dnf install enchant
```

Arch / Manjaro

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

OpenSUSE Tumbleweed

```
$ sudo zypper install enchant
```

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

Build documentation locally

Once your development environment is set up, run:

macOS

Linux

Windows

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

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

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

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

Live documentation preview

To support rapid editing of documentation, Toga also has a "live preview" mode:

macOS

Linux

Windows

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

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

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

This will build the documentation, start a web server to serve the build documentation, and watch the file system for any changes to the documentation source. If a change is detected, the documentation will be rebuilt, and any browser viewing the modified page will be automatically refreshed.

Live preview mode will only monitor the `docs` directory for changes. If you're updating the inline documentation associated with Toga source code, you'll need to use the `docs-live-src` target to build docs:

macOS

Linux

Windows

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

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

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

This behaves the same as `docs-live`, but will also monitor any changes to the `core/src` folder, reflecting any changes to inline documentation. However, the rebuild process takes much longer, so you may not want to use this target unless you're actively editing inline documentation.

Documentation linting

The build process will identify reStructuredText problems, but Toga performs some additional "lint" checks. To run the lint checks:

macOS

Linux

Windows

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

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

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

This will validate the documentation does not contain:

- dead hyperlinks
- misspelled words

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

- We prefer US spelling, with some liberties for programming-specific colloquialism (e.g., "apps") and verbing of nouns (e.g., "scrollable")
- Any reference to a product name should use the product's preferred capitalization. (e.g., "macOS", "GTK", "pytest", "Pygame", "PyScript").
- If a term is being used "as code", then it should be quoted as a literal rather than being added to the dictionary.

Rebuilding all documentation

To force a rebuild for all of the documentation:

macOS

Linux

Windows

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

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

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

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

What to work on?

If you're looking for specific areas to improve, there are tickets tagged "documentation" in Toga's issue tracker.

However, you don't need to be constrained by these tickets. If you can identify a gap in Toga's documentation, or an improvement that can be made, start writing! Anything that improves the experience of the end user is a welcome change.

Submitting a pull request

Before you submit a pull request, there's a few bits of housekeeping to do. See the section on submitting a pull request in the *code contribution guide* for details on our submission process.

4.3.3 Internal How-to guides

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

How to cut a Toga release

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

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

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

The procedure for cutting a new release is as follows:

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

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

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

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

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

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

   ```
   $ tox -e towncrier
   ```

   to generate the updated release notes.

3. Build the documentation to ensure that the new release notes don't include any spelling errors or markup problems:

   ```
   $ tox -e docs-lint,docs
   ```

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

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

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

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

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

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

6. Create a clean virtual environment, install the new release from Test PyPI, and perform any pre-release testing that may be appropriate:

```
$ python3 -m venv testvenv
$ . ./testvenv/bin/activate
(testvenv) $ pip install --extra-index-url https://test.pypi.org/simple/ toga==1.2.3
(testvenv) $ toga-demo
(testvenv) $ #... any other manual checks you want to perform ...
```

7. Log into ReadTheDocs, visit the Versions tab, and activate the new version. Ensure that the build completes; if there's a problem, you may need to correct the build configuration, roll back and re-tag the release.
8. Edit the GitHub release to add release notes. You can use the text generated by Towncrier, but you'll need to update the format to Markdown, rather than ReST. If necessary, check the pre-release checkbox.
9. Double check everything, then click Publish. This will trigger a publication workflow on GitHub.
10. Wait for the packages to appear on PyPI (toga-core, toga-cocoa, etc.).

Congratulations, you've just published a release!

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

4.4 Background

4.4.1 About Toga

Why Toga?

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

Native widgets - not themes

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

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

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

Abstract the broad concepts

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

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

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

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

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

We follow the 80% rule for abstractions - the goal is to provide an API that makes 80% of use cases possible, with enough internals exposed so that advanced users can get to the other 20% if they need to. We don't consider 100% coverage of all possible features to be desirable (or even achievable in many cases).

Python native

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

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

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

pip install and nothing more

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

Embrace mobile

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

FAQ

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.

Success Stories

Want to see examples of Toga in use? Here's some:

- Travel Tips is an app in the iOS App Store that uses Toga to describe it's user interface.
- Eddington is a data fitting tool based on *Toga* and *Briefcase*
- taRpnCalcTG is a Toga based calculator for Android, Windows and MacOS which is extensible with Python scripts.
- pyPlayground is a Toga based app for Android and Windows which can be modified to try Toga without additional tool chain.
- taAppLister is a Toga based Android app for listing and exporting all installed apps.
- RemoteCommand is a Toga based app for synchronizing the clipboard between Windows and MacOS.
- ChariotGazer is a Toga based app for Android and Windows, which provides detailed information about UK registered vehicles.
- Patent Toolkit is a Toga based app for Windows and MacOS, which contains a suite of tools for patent professionals.

Release History

0.4.9 (2025-02-07)

This release contains no new features. The primary purpose of this release is to add an upper version pin to Toga's Travertino requirement, protecting against the upcoming Toga 0.5.0 release that will include backwards incompatible changes in Travertino. (#3167)

Bugfixes

- The testbed app can now be run on *any* supported Python version. (#2883)
- App.app is now set to an initial value of `None`, before an app instance is created. This avoids a potential `AttributeError` when the test suite finishes. (#2918)

Misc

- #2476, #2913

0.4.8 (2024-10-16)

Bugfixes

- On macOS, apps that specify both *document_types* and a *main_window* no longer display the document selection dialog on startup. (#2860)
- The integration with Android's event loop has been updated to support Python 3.13. (#2907)

Backward Incompatible Changes

- Toga no longer supports Python 3.8. (#2888)
- Android applications should update their Gradle requirements to use version 1.12.0 of the Material library (`com.google.android.material:material:1.12.0`). (#2890)
- Android applications should update their Gradle requirements to use version 6.1.20 of the OSMDroid library (`org.osmdroid:osmdroid-android:6.1.20`). (#2890)

Misc

- #2868, #2869, #2870, #2876, #2877, #2884, #2885, #2886, #2887, #2893, #2897, #2898, #2899, #2900, #2901, #2902, #2903, #2904, #2905, #2906, #2912

0.4.7 (2024-09-18)

Features

- The GTK backend was modified to use PyGObject's native asyncio handling, instead of GBulb. (#2550)
- The ActivityIndicator widget is now supported on iOS. (#2829)

Bugfixes

- Windows retain their original size after being unminimized on Windows. (#2729)
- DOM storage is now enabled for WebView on Android. (#2767)
- A macOS app in full-screen mode now correctly displays the contents of windows that use a `toga.Box()` as the top-level content. (#2796)
- Asynchronous tasks are now protected from garbage collection while they are running. This could lead to asynchronous tasks terminating unexpectedly with an error under some conditions. (#2809)
- When a handler is a generator, control will now always be released to the event loop between iterations, even if no sleep interval or a sleep interval of 0 is yielded. (#2811)
- When the X button is clicked for the About dialog on GTK, it is now properly destroyed. (#2812)
- The Textual backend is now compatible with versions of Textual after v0.63.3. (#2822)
- The event loop is now guaranteed to be running when your app's `startup()` method is invoked. This wasn't previously the case on macOS and Windows. (#2834)
- iOS apps now correctly account for the size of the navigation bar when laying out app content. (#2836)
- A memory leak when using Divider or Switch widgets on iOS was resolved. (#2849)
- Apps bundled as standalone frozen binaries (e.g., POSIX builds made with PyInstaller) no longer crash on startup when trying to resolve the app icon. (#2852)

Misc

- #2088, #2708, #2715, #2792, #2799, #2802, #2803, #2804, #2807, #2823, #2824, #2825, #2826, #2846, #2847, #2848

0.4.6 (2024-08-28)

Features

- Toga can now define apps that persist in the background without having any open windows. (#97)
- Apps can now add items to the system tray. (#97)
- It is now possible to use an instance of Window as the main window of an app. This allows the creation of windows that don't have a menu bar or toolbar decoration. (#1870)
- The initial position of each newly created window is now different, cascading down the screen as windows are created. (#2023)

- The API for Documents and document types has been finalized. Document handling behavior is now controlled by declaring document types as part of your `toga.App` definition. (#2209)
- Toga can now define an app whose life cycle isn't tied to a single main window. (#2209)
- The Divider widget was implemented on iOS. (#2478)
- Commands can now be retrieved by ID. System-installed commands (such as "About" and "Visit Homepage") are installed using a known ID that can be used at runtime to manipulate those commands. (#2636)
- A `MainWindow` can now have an `on_close` handler. If a request is made to close the main window, the `on_close` handler will be evaluated; app exit handling will only be processed if the close handler allows the close to continue. (#2643)
- Dialogs can now be displayed relative to an app, in addition to be being modal to a window. (#2669)
- An `on_running` event handler was added to `toga.App`. This event will be triggered when the app's main loop starts. (#2678)
- The `on_exit` handler for an app can now be defined by overriding the method on the `toga.App` subclass. (#2678)
- CommandSet now exposes a full set and dictionary interface. Commands can be added to a CommandSet using `[]` notation and a command ID; they can be removed using set-like `remove()` or `discard()` calls with a Command instance, or using dictionary-like `pop()` or `del` calls with the command ID. (#2701)
- WebView2 on Winforms now uses the v1.0.2592.51 WebView2 runtime DLLs. (#2764)

Bugfixes

- The order of creation of system-level commands is now consistent between platforms. Menu creation is guaranteed to be deferred until the user's startup method has been invoked. (#2619)
- The type of SplitContainer's content was modified to be a list, rather than a tuple. (#2638)
- Programmatically invoking `close()` on the main window will now trigger `on_exit` handling. Previously `on_exit` handling would only be triggered if the close was initiated by a user action. (#2643)
- GTK apps no longer have extra padding between the menu bar and the window content when the app does not have a toolbar. (#2646)
- On Winforms, the window of an application that is set as the main window is no longer shown as a result of assigning the window as `App.main_window`. (#2653)
- Menu items on macOS are now able to correctly bind to the arrow and home/end/delete keys. (#2661)
- On GTK, the currently selected tab index on an `OptionContainer` can now be retrieved inside an `on_select` handler. (#2703)
- The WebView can now be loaded when using Python from the Windows Store. (#2752)
- The WebView and MapView widgets now log an error if initialization fails. (#2779)

Backward Incompatible Changes

- The `add_background_task()` API on `toga.App` has been deprecated. Background tasks can be implemented using the new `on_running` event handler, or by using `asyncio.create_task`. (#2099)
- The API for Documents and Document-based apps has been significantly modified. Unfortunately, these changes are not backwards compatible; any existing Document-based app will require modification.

  The `DocumentApp` base class is no longer required. Apps can subclass `App` directly, passing the document types as a `list` of `Document` classes, rather than a mapping of extension to document type.

  The API for `Document` subclasses has also changed:

  - A path is no longer provided as an argument to the Document constructor;
  - The `document_type` is now specified as a class property called `description`; and
  - Extensions are now defined as a class property of the `Document`; and
  - The `can_close()` handler is no longer honored. Documents now track if they are modified, and have a default `on_close` handler that uses the modification status of a document to control whether a document can close. Invoking `touch()` on document will mark a document as modified. This modification flag is cleared by saving the document. (#2209)
- It is no longer possible to create a toolbar on a `Window` instance. Toolbars can only be added to `MainWindow` (or subclass). (#2646)
- The default title of a `toga.Window` is now the name of the app, rather than `"Toga"`. (#2646)
- The APIs on `Window` for displaying dialogs (`info_dialog()`, `question_dialog()`, etc) have been deprecated. They can be replaced with creating an instance of a `Dialog` class (e.g., `InfoDialog`), and passing that instance to `window.dialog()`. (#2669)

Documentation

- Building Toga's documentation now requires the use of Python 3.12. (#2745)

Misc

- #2382, #2635, #2640, #2647, #2648, #2654, #2657, #2660, #2665, #2668, #2675, #2676, #2677, #2682, #2683, #2684, #2689, #2693, #2694, #2695, #2696, #2697, #2698, #2699, #2709, #2710, #2711, #2712, #2722, #2723, #2724, #2726, #2727, #2728, #2733, #2734, #2735, #2736, #2739, #2740, #2742, #2743, #2755, #2756, #2757, #2758, #2760, #2771, #2775, #2776, #2777, #2783, #2788, #2789, #2790

0.4.5 (2024-06-11)

Features

- The typing for Toga's API surface was updated to be more precise. (#2252)
- APIs were added for replacing a widget in an existing layout, and for obtaining the index of a widget in a list of children. (#2301)
- The content of a window can now be set when the window is constructed. (#2307)
- Size and position properties now return values as a `Size` and `Position` `namedtuple`, respectively. `namedtuple` objects support addition and subtraction operations. Basic tuples can still be used to *set* these properties. (#2388)
- Android deployments no longer require the SwipeRefreshLayout component unless the app uses the Toga DetailedList widget. (#2454)

Bugfixes

- Invocation order of TextInput on_change and validation is now correct. (#2325)
- Dialog windows are now properly modal when using the GTK backend. (#2446)
- The Button testbed tests can accommodate minor rendering differences on Fedora 40. (#2583)
- On macOS, apps will now raise a warning if camera permissions have been requested, but those permissions have not been declared as part of the application metadata. (#2589)

Documentation

- The instructions for adding a change note to a pull request have been clarified. (#2565)
- The minimum supported Linux release requirements were updated to Ubuntu 20.04 or Fedora 32. (#2566)
- The first-time contributor README link has been updated. (#2588)
- Typos in the usage examples of `toga.MapPin` were corrected. (#2617)

Misc

- #2567, #2568, #2569, #2570, #2571, #2576, #2577, #2578, #2579, #2580, #2593, #2600, #2601, #2602, #2604, #2605, #2606, #2614, #2621, #2625, #2626, #2627, #2629, #2631, #2632

0.4.4 (2024-05-08)

Bugfixes

- The mechanism for loading application icons on macOS was corrected to account for how Xcode populates `Info.plist` metadata. (#2558)

Misc

- #2555, #2557, #2560

0.4.3 (2024-05-06)

Features

- A MapView widget was added. (#727)
- Toga apps can now access details about the screens attached to the computer. Window position APIs have been extended to allow for placement on a specific screen, and positioning relative to a specific screen. (#1930)
- Key definitions were added for number pad keys on GTK. (#2232)
- Toga can now be extended, via plugins, to create Toga Images from external image classes (and vice-versa). (#2387)
- Non-implemented features now raise a formal warning, rather than logging to the console. (#2398)
- Support for Python 3.13 was added. (#2404)
- Toga's release processes now include automated testing on ARM64. (#2404)
- An action for a Toga command can now be easily modified after initial construction. (#2433)
- A geolocation service was added for Android, iOS and macOS. (#2462)
- When a Toga app is packaged as a binary, and no icon is explicitly configured, Toga will now use the binary's icon as the app icon. This means it is no longer necessary to include the app icon as data in a `resources` folder if you are packaging your app for distribution. (#2527)

Bugfixes

- Compatibility with macOS 14 (Sonoma) was added. (#2188, #2383)
- Key handling for Insert, Delete, NumLock, ScrollLock, and some other esoteric keys was added for GTK and Winforms. Some uses of bare Shift on GTK were also improved. (#2220)
- A crash observed on iOS devices when taking photographs has been resolved. (#2381)

- Key shortcuts for punctuation and special keys (like Page Up and Escape) were added for GTK and Winforms. (#2414)
- The placement of menu items relative to sub-menus was corrected on GTK. (#2418)
- Tree data nodes can now be modified prior to tree expansion. (#2439)
- Some memory leaks associated with macOS Icon and Image storage were resolved. (#2472)
- The stack trace dialog no longer raises an `asyncio.TimeoutError` when displayed. (#2474)
- The integration of the `asyncio` event loop was simplified on Android. As a result, `asyncio.loop.run_in_executor()` now works as expected. (#2479)
- Some memory leaks associated with the macOS Table, Tree and DetailedList widgets were resolved. (#2482)
- Widget IDs can now be reused after the associated widget's window is closed. (#2514)
- *`WebView`* is now compatible with Linux GTK environments only providing WebKit2 version 4.1 without version 4.0. (#2527)

Backward Incompatible Changes

- The macOS implementations of `Window.as_image()` and `Canvas.as_image()` APIs now return images in native device resolution, not CSS pixel resolution. This will result in images that are double the previous size on Retina displays. (#1930)

Documentation

- The camera permission requirements on macOS apps have been clarified. (#2381)
- Documentation for the class property `toga.App.app` was added. (#2413)
- The documentation landing page and some documentation sections were reorganized. (#2463)
- The README badges were updated to display correctly on GitHub. (#2491)
- The links to ReadTheDocs were updated to better arbitrate between linking to the stable version or the latest version. (#2510)
- An explicit system requirements section was added to the documentation for widgets that require the installation of additional system components. (#2544)
- The system requirements were updated to be more explicit and now include details for OpenSUSE Tumbleweed. (#2549)

Misc

- #2153, #2372, #2389, #2390, #2391, #2392, #2393, #2394, #2396, #2397, #2400, #2403, #2405, #2406, #2407, #2408, #2409, #2422, #2423, #2427, #2440, #2442, #2445, #2448, #2449, #2450, #2457, #2458, #2459, #2460, #2464, #2465, #2466, #2467, #2470, #2471, #2476, #2487, #2488, #2498, #2501, #2502, #2503, #2504, #2509, #2518, #2519, #2520, #2521, #2522, #2523, #2532, #2533, #2534, #2535, #2536, #2537, #2538, #2539, #2540, #2541, #2542, #2546, #2552

0.4.2 (2024-02-06)

Features

- Buttons can now be created with an icon, instead of a text label. (#774)
- Widgets and Windows can now be sorted. The ID of the widget is used for the sorting order. (#2190)

- The main window generated by the default `startup()` method of an app now has an ID of `main`. (#2190)
- A cross-platform API for camera access was added. (#2266, #2353)
- An OptionContainer widget was added for Android. (#2346)

Bugfixes

- New widgets with an ID matching an ID that was previously used no longer cause an error. (#2190)
- `App.current_window` on GTK now returns `None` when all windows are hidden. (#2211)
- Selection widgets on macOS can now include duplicated titles. (#2319)
- The padding around DetailedList on Android has been reduced. (#2338)
- The error returned when an Image is created with no source has been clarified. (#2347)
- On macOS, `toga.Image` objects can now be created from raw data that didn't originate from a file. (#2355)
- Winforms no longer generates a system beep when pressing Enter in a TextInput. (#2374)

Backward Incompatible Changes

- Widgets must now be added to a window to be available in the widget registry for lookup by ID. (#2190)
- If the label for a Selection contains newlines, only the text up to the first newline will be displayed. (#2319)
- The internal Android method `intent_result` has been deprecated. This was an internal API, and not formally documented, but it was the easiest mechanism for invoking Intents on the Android backend. It has been replaced by the synchronous `start_activity` method that allows you to register a callback when the intent completes. (#2353)

Documentation

- Initial documentation of backend-specific features has been added. (#1798)
- The difference between Icon and Image was clarified, and a note about the lack of an `on_press` handler on ImageView was added. (#2348)

Misc

- #2298, #2299, #2302, #2312, #2313, #2318, #2331, #2332, #2333, #2336, #2337, #2339, #2340, #2357, #2358, #2359, #2363, #2367, #2368, #2369, #2370, #2371, #2375, #2376

0.4.1 (2023-12-21)

Features

- Toga images can now be created from (and converted to) PIL images. (#2142)
- A wider range of command shortcut keys are now supported on WinForms. (#2198)
- Most widgets with flexible sizes now default to a minimum size of 100 CSS pixels. An explicit size will still override this value. (#2200)
- OptionContainer content can now be constructed using `toga.OptionItem` objects. (#2259)
- An OptionContainer widget was added for iOS. (#2259)
- Apps can now specify platform-specific icon resources by appending the platform name (e.g., `-macOS` or `-windows`) to the icon filename. (#2260)

- Images can now be created from the native platform representation of an image, without needing to be transformed to bytes. (#2263)

Bugfixes

- TableViews on macOS will no longer crash if a drag operation is initiated from inside the table. (#1156)
- Separators before and after command sub-groups are now included in menus. (#2193)
- The web backend no longer generates a duplicate title bar. (#2194)
- The web backend is now able to display the About dialog on first page load. (#2195)
- The testbed is now able to run on macOS when the user running the tests has the macOS display setting "Prefer tabs when opening documents" set to "Always". (#2208)
- Compliance with Apple's HIG regarding the naming and shortcuts for the Close and Close All menu items was improved. (#2214)
- Font handling on older versions of iOS has been corrected. (#2265)
- ImageViews with `flex=1` will now shrink to fit if the image is larger than the available space. (#2275)

Backward Incompatible Changes

- The `toga.Image` constructor now takes a single argument (`src`); the `path` and `data` arguments are deprecated. (#2142)
- The use of Caps Lock as a keyboard modifier for commands was removed. (#2198)
- Support for macOS release prior to Big Sur (11) has been dropped. (#2228)
- When inserting or appending a tab to an OptionContainer, the `enabled` argument must now be provided as a keyword argument. The name of the first argument has been also been renamed (from `text` to `text_or_item`); it should generally be passed as a positional, rather than keyword argument. (#2259)
- The use of synchronous `on_result` callbacks on dialogs and `Webview.evaluate_javascript()` calls has been deprecated. These methods should be used in their asynchronous form. (#2264)

Documentation

- Documentation for `toga.Key` was added. (#2199)
- Some limitations on App presentation imposed by Wayland have been documented. (#2255)

Misc

- #2201, #2204, #2215, #2216, #2219, #2222, #2224, #2226, #2230, #2235, #2240, #2246, #2249, #2256, #2257, #2261, #2264, #2267, #2269, #2270, #2271, #2272, #2283, #2284, #2287, #2294

0.4.0 (2023-11-03)

Features

- The Toga API has been fully audited. All APIs now have 100% test coverage, complete API documentation (including type annotations), and are internally consistent. (#1903, #1938, #1944, #1946, #1949, #1951, #1955, #1956, #1964, #1969, #1984, #1996, #2011, #2017, #2025, #2029, #2044, #2058, #2075)
- Headings are no longer mandatory for Tree widgets. If headings are not provided, the widget will not display its header bar. (#1767)

- Support for custom font loading was added to the GTK, Cocoa and iOS backends. (#1837)
- The testbed app has better diagnostic output when running in test mode. (#1847)
- A Textual backend was added to support terminal applications. (#1867)
- Support for determining the currently active window was added to Winforms. (#1872)
- Programmatically scrolling to top and bottom in MultilineTextInput is now possible on iOS. (#1876)
- A handler has been added for users confirming the contents of a TextInput by pressing Enter/Return. (#1880)
- An API for giving a window focus was added. (#1887)
- Widgets now have a `.clear()` method to remove all child widgets. (#1893)
- Winforms now supports hiding and re-showing the app cursor. (#1894)
- ProgressBar and Switch widgets were added to the Web backend. (#1901)
- Missing value handling was added to the Tree widget. (#1913)
- App paths now include a `config` path for storing configuration files. (#1964)
- A more informative error message is returned when a platform backend doesn't support a widget. (#1992)
- The example apps were updated to support being run with `briefcase run` on all platforms. (#1995)
- Headings are no longer mandatory Table widgets. (#2011)
- Columns can now be added and removed from a Tree. (#2017)
- The default system notification sound can be played via `App.beep()`. (#2018)
- DetailedList can now respond to "primary" and "secondary" user actions. These may be implemented as left and right swipe respectively, or using any other platform-appropriate mechanism. (#2025)
- A DetailedList can now provide a value to use when a row doesn't provide the required data. (#2025)
- The accessors used to populate a DetailedList can now be customized. (#2025)
- Transformations can now be applied to *any* canvas context, not just the root context. (#2029)
- Canvas now provides more `list`-like methods for manipulating drawing objects in a context. (#2029)
- On Windows, the default font now follows the system theme. On most devices, this means it has changed from Microsoft Sans Serif 8pt to Segoe UI 9pt. (#2029)
- Font sizes are now consistently interpreted as CSS points. On Android, iOS and macOS, this means any numeric font sizes will appear 33% larger than before. The default font size on these platforms is unchanged. (#2029)
- MultilineTextInputs no longer show spelling suggestions when in read-only mode. (#2136)
- Applications now verify that a main window has been created as part of the `startup()` method. (#2047)
- An implementation of ActivityIndicator was added to the Web backend. (#2050)
- An implementation of Divider was added to the Web backend. (#2051)
- The ability to capture the contents of a window as an image has been added. (#2063)
- A PasswordInput widget was added to the Web backend. (#2089)
- The WebKit inspector is automatically enabled on all macOS WebViews, provided you're using macOS 13.3 (Ventura) or iOS 16.4, or later. (#2109)
- Text input widgets on macOS now support undo and redo. (#2151)
- The Divider widget was implemented on Android. (#2181)

Bugfixes

- The WinForms event loop was decoupled from the main form, allowing background tasks to run without a main window being present. (#750)
- Widgets are now removed from windows when the window is closed, preventing a memory leak on window closure. (#1215)
- Android and iOS apps no longer crash if you invoke `App.hide_cursor()` or `App.show_cursor()`. (#1235)
- A Selection widget with no items now consistently returns a selected value of `None` on all platforms. (#1723)
- macOS widget methods that return strings are now guaranteed to return strings, rather than native Objective C string objects. (#1779)
- WebViews on Windows no longer have a black background when they are resized. (#1855)
- The interpretation of `MultilineTextInput.readonly` was corrected iOS (#1866)
- A window without an `on_close` handler can now be closed using the window frame close button. (#1872)
- Android apps running on devices older than API level 29 (Android 10) no longer crash. (#1878)
- Missing value handling on Tables was fixed on Android and Linux. (#1879)
- The GTK backend is now able to correctly identify the currently active window. (#1892)
- Error handling associated with the creation of Intents on Android has been improved. (#1909)
- The DetailedList widget on GTK now provides an accurate size hint during layout. (#1920)
- Apps on Linux no longer segfault if an X Windows display cannot be identified. (#1921)
- The `on_result` handler is now used by Cocoa file dialogs. (#1947)
- Pack layout now honors an explicit width/height setting of 0. (#1958)
- The minimum window size is now correctly recomputed and enforced if window content changes. (#2020)
- The title of windows can now be modified on Winforms. (#2094)
- An error on Winforms when a window has no content has been resolved. (#2095)
- iOS container views are now set to automatically resize with their parent view (#2161)

Backward Incompatible Changes

- The `weight`, `style` and `variant` arguments for `Font` and `Font.register` are now keyword-only. (#1903)
- The `clear()` method for resetting the value of a MultilineTextInput, TextInput and PasswordInput has been removed. This method was an ambiguous override of the `clear()` method on Widget that removed all child nodes. To remove all content from a text input widget, use `widget.value = ""`. (#1938)
- The ability to perform multiple substring matches in a `Contains` validator has been removed. (#1944)
- The `TextInput.validate` method has been removed. Validation now happens automatically whenever the `value` or `validators` properties are changed. (#1944)
- The argument names used to construct validators have changed. Error message arguments now all end with `_message`; `compare_count` has been renamed `count`; and `min_value` and `max_value` have been renamed `min_length` and `max_length`, respectively. (#1944)
- The `get_dom()` method on WebView has been removed. This method wasn't implemented on most platforms, and wasn't working on any of the platforms where it *was* implemented, as modern web view implementations don't provide a synchronous API for accessing web content in this way. (#1949)

- The `evaluate_javascript()` method on WebView has been modified to work in both synchronous and asynchronous contexts. In a synchronous context you can invoke the method and use a functional `on_result` callback to be notified when evaluation is complete. In an asynchronous context, you can await the result. (#1949)
- The `on_key_down` handler has been removed from WebView. If you need to catch user input, either use a handler in the embedded JavaScript, or create a `Command` with a key shortcut. (#1949)
- The `invoke_javascript()` method has been removed. All usage of `invoke_javascript()` can be replaced with `evaluate_javascript()`. (#1949)
- The usage of local `file://` URLs has been explicitly prohibited. `file://` URLs have not been reliable for some time; their usage is now explicitly prohibited. (#1949)
- `DatePicker` has been renamed `DateInput`. (#1951)
- `TimePicker` has been renamed `TimeInput`. (#1951)
- The `on_select` handler on the Selection widget has been renamed `on_change` for consistency with other widgets. (#1955)
- The `_notify()` method on data sources has been renamed `notify()`, reflecting its status as a public API. (#1955)
- The `prepend()` method was removed from the `ListSource` and `TreeSource` APIs. Calls to `prepend(...)` can be replaced with `insert(0, ...)`. (#1955)
- The `insert()` and `append()` APIs on `ListSource` and `TreeSource` have been modified to provide an interface that is closer to that `list` API. These methods previously accepted a variable list of positional and keyword arguments; these arguments should be combined into a single tuple or dictionary. This matches the API provided by `__setitem__()`. (#1955)
- Images and ImageViews no longer support loading images from URLs. If you need to display an image from a URL, use a background task to obtain the image data asynchronously, then create the Image and/or set the ImageView `image` property on the completion of the asynchronous load. (#1956)
- A row box contained inside a row box will now expand to the full height of its parent, rather than collapsing to the maximum height of the inner box's child content. (#1958)
- A column box contained inside a column box will now expand to the full width of its parent, rather than collapsing to the maximum width of the inner box's child content. (#1958)
- On Android, the user data folder is now a `data` sub-directory of the location returned by `context.getFilesDir()`, rather than the bare `context.getFilesDir()` location. (#1964)
- GTK now returns `~/.local/state/appname/log` as the log file location, rather than `~/.cache/appname/log`. (#1964)
- The location returned by `toga.App.paths.app` is now the folder that contains the Python source file that defines the app class used by the app. If you are using a `toga.App` instance directly, this may alter the path that is returned. (#1964)
- On Winforms, if an application doesn't define an author, an author of `Unknown` is now used in application data paths, rather than `Toga`. (#1964)
- Winforms now returns `%USERPROFILE%/AppData/Local/<Author Name>/<App Name>/Data` as the user data file location, rather than `%USERPROFILE%/AppData/Local/<Author Name>/<App Name>`. (#1964)
- Support for SplitContainers with more than 2 panels of content has been removed. (#1984)
- Support for 3-tuple form of specifying SplitContainer items, used to prevent panels from resizing, has been removed. (#1984)
- The ability to increment and decrement the current OptionContainer tab was removed. Instead of *container.current_tab += 1*, use *container.current_tab = container.current_tab.index + 1* (#1996)

- `OptionContainer.add()`, `OptionContainer.remove()` and `OptionContainer.insert()` have been removed, due to being ambiguous with base widget methods of the same name. Use the `OptionContainer.content.append()`, `OptionContainer.content.remove()` and `OptionContainer.content.insert()` APIs instead. (#1996)
- The `on_select` handler for OptionContainer no longer receives the `option` argument providing the selected tab. Use `current_tab` to obtain the currently selected tab. (#1996)
- `TimePicker.min_time` and `TimePicker.max_time` has been renamed `TimeInput.min` and `TimeInput.max`, respectively. (#1999)
- `DatePicker.min_date` and `DatePicker.max_date` has been renamed `DateInput.min` and `DateInput.max`, respectively. (#1999)
- `NumberInput.min_value` and `NumberInput.max_value` have been renamed `NumberInput.min` and `NumberInput.max`, respectively. (#1999)
- `Slider.range` has been replaced by `Slider.min` and `Slider.max`. (#1999)
- Tables now use an empty string for the default missing value, rather than warning about missing values. (#2011)
- `Table.add_column()` has been deprecated in favor of `Table.append_column()` and `Table.insert_column()` (#2011)
- `Table.on_double_click` has been renamed `Table.on_activate`. (#2011, #2017)
- Trees now use an empty string for the default missing value, rather than warning about missing values. (#2017)
- The `parent` argument has been removed from the `insert` and `append` calls on `TreeSource`. This improves consistency between the API for `TreeSource` and the API for `list`. To insert or append a row in to a descendant of a TreeSource root, use `insert` and `append` on the parent node itself - i.e., `source.insert(parent, index, ...)` becomes `parent.insert(index, ...)`, and `source.insert(None, index, ...)` becomes `source.insert(index, ...)`. (#2017)
- When constructing a DetailedList from a list of tuples, or a list of lists, the required order of values has changed from (icon, title, subtitle) to (title, subtitle, icon). (#2025)
- The `on_select` handler for DetailedList no longer receives the selected row as an argument. (#2025)
- The handling of row deletion in DetailedList widgets has been significantly altered. The `on_delete` event handler has been renamed `on_primary_action`, and is now *only* a notification that a "swipe left" event (or platform equivalent) has been confirmed. This was previously inconsistent across platforms. Some platforms would update the data source to remove the row; some treated `on_delete` as a notification event and expected the application to handle the deletion. It is now the application's responsibility to perform the data deletion. (#2025)
- Support for Python 3.7 was removed. (#2027)
- `fill()` and `stroke()` now return simple drawing operations, rather than context managers. If you attempt to use `fill()` or `stroke()` on a context as a context manager, an exception will be raised; using these methods on Canvas will raise a warning, but return the appropriate context manager. (#2029)
- The `clicks` argument to `Canvas.on_press` has been removed. Instead, to detect "double clicks", you should use `Canvas.on_activate`. The `clicks` argument has also been removed from `Canvas.on_release`, `Canvas.on_drag`, `Canvas.on_alt_press`, `Canvas.on_alt_release`, and `Canvas.on_alt_drag`. (#2029)
- The `new_path` operation has been renamed `begin_path` for consistency with the HTML5 Canvas API. (#2029)
- Methods that generate new contexts have been renamed: `context()`, `closed_path()`, `fill()` and `stroke()` have become `Context()`, `ClosedPath()`, `Fill()` and `Stroke()` respectively. This has been done to make it easier to differentiate between primitive drawing operations and context-generating operations. (#2029)

- A Canvas is no longer implicitly a context object. The `Canvas.context` property now returns the root context of the canvas. If you were previously using `Canvas.context()` to generate an empty context, it should be replaced with `Canvas.Context()`. Any operations to `remove()` drawing objects from the canvas or `clear()` the canvas of drawing objects should be made on `Canvas.context`. Invoking these methods on `Canvas` will now call the base `Widget` implementations, which will throw an exception because `Canvas` widgets cannot have children. (#2029)
- The `preserve` option on `Fill()` operations has been deprecated. It was required for an internal optimization and can be safely removed without impact. (#2029)
- Drawing operations (e.g., `arc`, `line_to`, etc) can no longer be invoked directly on a Canvas. Instead, they should be invoked on the root context of the canvas, retrieved with via the *canvas* property. Context creating operations (`Fill`, `Stroke` and `ClosedPath`) are not affected. (#2029)
- The `tight` argument to `Canvas.measure_text()` has been deprecated. It was a GTK implementation detail, and can be safely removed without impact. (#2029)
- The `multiselect` argument to Open File and Select Folder dialogs has been renamed `multiple_select`, for consistency with other widgets that have multiple selection capability. (#2058)
- `Window.resizeable` and `Window.closeable` have been renamed `Window.resizable` and `Window.closable`, to adhere to US spelling conventions. (#2058)
- Windows no longer need to be explicitly added to the app's window list. When a window is created, it will be automatically added to the windows for the currently running app. (#2058)
- The optional arguments of `Command` and `Group` are now keyword-only. (#2075)
- In `App`, the properties `id` and `name` have been deprecated in favor of `app_id` and `formal_name` respectively, and the property `module_name` has been removed. (#2075)
- `GROUP_BREAK`, `SECTION_BREAK` and `CommandSet` were removed from the `toga` namespace. End users generally shouldn't need to use these classes. If your code *does* need them for some reason, you can access them from the `toga.command` namespace. (#2075)
- The `windows` constructor argument of `toga.App` has been removed. Windows are now automatically added to the current app. (#2075)
- The `filename` argument and property of `toga.Document` has been renamed `path`, and is now guaranteed to be a `pathlib.Path` object. (#2075)
- Documents must now provide a `create()` method to instantiate a `main_window` instance. (#2075)
- `App.exit()` now unconditionally exits the app, rather than confirming that the `on_exit` handler will permit the exit. (#2075)

Documentation

- Documentation for application paths was added. (#1849)
- The contribution guide was expanded to include more suggestions for potential projects, and to explain how the backend tests work. (#1868)
- All code blocks were updated to add a button to copy the relevant contents on to the user's clipboard. (#1897)
- Class references were updated to reflect their preferred import location, rather than location where they are defined in code. (#2001)
- The Linux system dependencies were updated to reflect current requirements for developing and using Toga. (#2021)

Misc

- #1865, #1875, #1881, #1882, #1886, #1889, #1895, #1900, #1902, #1906, #1916, #1917, #1918, #1926, #1933, #1948, #1950, #1952, #1954, #1963, #1972, #1977, #1980, #1988, #1989, #1998, #2008, #2014, #2019, #2022, #2028, #2034, #2035, #2039, #2052, #2053, #2055, #2056, #2057, #2059, #2067, #2068, #2069, #2085, #2090, #2092, #2093, #2101, #2102, #2113, #2114, #2115, #2116, #2118, #2119, #2123, #2124, #2127, #2128, #2131, #2132, #2146, #2147, #2148, #2149, #2150, #2163, #2165, #2166, #2171, #2177, #2180, #2184, #2186

0.3.1 (2023-04-12)

Features

- The Button widget now has 100% test coverage, and complete API documentation. (#1761)
- The mapping between Pack layout and HTML/CSS has been formalized. (#1778)
- The Label widget now has 100% test coverage, and complete API documentation. (#1799)
- TextInput now supports focus handlers and changing alignment on GTK. (#1817)
- The ActivityIndicator widget now has 100% test coverage, and complete API documentation. (#1819)
- The Box widget now has 100% test coverage, and complete API documentation. (#1820)
- NumberInput now supports changing alignment on GTK. (#1821)
- The Divider widget now has 100% test coverage, and complete API documentation. (#1823)
- The ProgressBar widget now has 100% test coverage, and complete API documentation. (#1825)
- The Switch widget now has 100% test coverage, and complete API documentation. (#1832)
- Event handlers have been internally modified to simplify their definition and use on backends. (#1833)
- The base Toga Widget now has 100% test coverage, and complete API documentation. (#1834)
- Support for FreeBSD was added. (#1836)
- The Web backend now uses Shoelace to provide web components. (#1838)
- Winforms apps can now go full screen. (#1863)

Bugfixes

- Issues with reducing the size of windows on GTK have been resolved. (#1205)
- iOS now supports newlines in Labels. (#1501)
- The Slider widget now has 100% test coverage, and complete API documentation. (#1708)
- The GTK backend no longer raises a warning about the use of a deprecated `set_wmclass` API. (#1718)
- MultilineTextInput now correctly adapts to Dark Mode on macOS. (#1783)
- The handling of GTK layouts has been modified to reduce the frequency and increase the accuracy of layout results. (#1794)
- The text alignment of MultilineTextInput on Android has been fixed to be TOP aligned. (#1808)
- GTK widgets that involve animation (such as Switch or ProgressBar) are now redrawn correctly. (#1826)

Improved Documentation

- API support tables now distinguish partial vs full support on each platform. (#1762)
- Some missing settings and constant values were added to the documentation of Pack. (#1786)
- Added documentation for `toga.App.widgets`. (#1852)

Misc

- #1750, #1764, #1765, #1766, #1770, #1771, #1777, #1797, #1802, #1813, #1818, #1822, #1829, #1830, #1835, #1839, #1854, #1861

0.3.0 (2023-01-30)

Features

- Widgets now use a three-layered (Interface/Implementation/Native) structure.
- A GUI testing framework was added.
- A simplified "Pack" layout algorithm was added.
- Added a web backend.

Bugfixes

- Too many to count!

0.2.15

- Added more widgets and cross-platform support, especially for GTK+ and Winforms

0.2.14

- Removed use of `namedtuple`

0.2.13

- Various fixes in preparation for PyCon AU demo

0.2.12

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

0.2.0 - 0.2.11

Internal development releases.

0.1.2

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

0.1.1

- Refactored code into multiple repositories, so that users of one backend don't have to carry the overhead of other installed platforms
- Corrected a range of bugs, mostly related to problems under Python 3.

0.1.0

Initial public release. Includes:

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

Toga's Road Map

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

Widgets

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

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

Input

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

Partially implemented widgets

- `Table`

  Mobile platforms don't provide a native "Table" widget; however, table-like data could be rendered by using a *`DetailedList`*, where the title and subtitle are the "important" columns selected by the user, and selecting a row on the table navigates to a sub-page showing the full row detail.

- `Tree`

  As with Table, mobile platforms don't provide a native "Tree" widget; however, we could use a similar approach to Table.

  On Windows, the native widget doesn't provide support for more than one column. This means we either need to make a special case on Windows for a "simple" tree; or we need to form a composite widget that pairs the scrolling of a table with a tree.

- `DateTimeInput` - A widget for selecting a date and a time.

  - Cocoa: `NSDatePicker`
  - GTK: `Gtk.Calendar` + ?
  - iOS: `UIDatePicker`

New widgets

- `RadioButton` - a set of mutually exclusive options.

  Functionally, the use case of "select one from a list of options" can be met with a *`Selection`*; however, from a UI design perspective, a radio button is a common design pattern.

  See this issue for discussion of how this widget may be implemented in a way that compliments existing widgets.

- `ComboBox` - A free entry text field that provides options (e.g., text with past choices)
  - Cocoa: `NSComboBox`
  - GTK: `Gtk.ComboBox.new_with_model_and_entry`
  - iOS: ?
  - Winforms: `ComboBox`
  - Android: `Spinner`
- `ColorInput` - A widget for selecting a color
  - Cocoa: `NSColorWell`
  - GTK: `Gtk.ColorButton` or `Gtk.ColorSelection`
  - iOS: ?
  - Winforms: ?
  - Android: ?
- `SearchInput` - A variant of `TextField` that is decorated as a search box.
  - Cocoa: `NSSearchField`
  - GTK: `Gtk.Entry`
  - iOS: `UISearchBar`?
  - Winforms: ?
  - Android: ?

Views

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

- `VideoView` - Display a video
  - Cocoa: `AVPlayerView`
  - GTK: Custom integration with `GStreamer`
  - iOS: `MPMoviePlayerController`
  - Winforms: ?
  - Android: ?
- `PDFView` - Display a PDF document

  - Cocoa: `PDFView`
  - GTK: ?
  - iOS: Integration with QuickLook?
  - Winforms: ?
  - Android: ?

Container widgets

Containers are widgets that can contain other widgets.

- `FormContainer` - A layout for a "key/value" or "label/widget" form
  - Cocoa: `NSForm`, or `NSView` with pre-set constraints.
  - GTK:
  - iOS:
  - Winforms: ?
  - Android: ?
- `NavigationContainer` - A container view that holds a navigable tree of sub-views

  Essentially a view that has a "back" button to return to the previous view in a hierarchy. Example of use: Top level navigation in the macOS System Preferences panel.
  - Cocoa: No native control
  - GTK: No native control; `Gtk.HeaderBar` in 3.10+
  - iOS: `UINavigationBar` + `NavigationController`
  - Winforms: ?
  - Android: ?

Other capabilities

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

- **Preferences** - Support for saving app preferences, and visualizing them in a platform native way.
- **Notification** - A mechanism to display popup "toast"-style notifications
- **System tray icons** - Presenting an icon and/or menu in the platform's system tray - possibly without having a main app window at all.
- **Licensing/registration** - Monetization is not a bad thing, and shouldn't be mutually exclusive with open source.
- **Audio** - The ability to play sound files, either once off, or on a loop.
- **Cloud data access** - Traditional apps store all their files locally; however, using cloud services or network resources is increasingly common, especially in mobile apps. However, accessing cloud-based files can be very complicated; Toga is in a position to provide a file-like API that abstracts accessing these APIs.
- **QR code scanning** - The ability to scan QR codes and bar codes in an app, and return the encoded data.

Platforms

Toga currently has good support for Cocoa on macOS, GTK on Linux, Winforms on Windows, iOS and Android. Proof-of-concept support exists for single page web apps and consoles. Support for a more modern Windows API would be desirable, as would support for Qt.

4.4.2 Community

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

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

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

4.4.3 Toga's internals

Architecture

Although Toga presents a single interface to the end user, there are three internal layers that make up every widget. They are:

- The **Interface** layer
- The **Implementation** layer
- The **Native** layer

Interface

The interface layer is the public, documented interface for each widget. Following *Toga's design philosophy*, these widgets reflect high-level design concepts, rather than specific common widgets. It forms the public API for creating apps, windows, widgets, and so on.

The interface layer is responsible for validation of any API inputs, and storage of any persistent values retained by a widget. That storage may be supplemented or replaced by storage on the underlying native widget (or widgets), depending on the capabilities of that widget.

The interface layer is also responsible for storing style and layout-related attributes of the widget.

The interface layer is defined in the `toga-core` module.

Implementation

The implementation layer is the platform-specific representation of each widget. Each platform that Toga supports has its own implementation layer, named after the widget toolkit that the implementation layer is wrapping – `toga-cocoa` for macOS (Cocoa being the name of the underlying macOS widget toolkit); `toga-gtk` for Linux (using the GTK+ toolkit); and so on. The implementation provides a private, internal API that the interface layer can use to create the widgets described by the interface layer.

The API exposed by the implementation layer is different to that exposed by the interface layer and is *not* intended for end-user consumption. It is a utility API, servicing the requirements of the interface layer.

Every widget in the implementation layer corresponds to exactly one widget in the interface layer. However, the reverse will not always be true. Some widgets defined by the interface layer are not available on all platforms.

An interface widget obtains its implementation when it is constructed, using the platform factory. Each platform provides a factory implementation. When a Toga application starts, it guesses its platform based on the value of `sys.platform`, and uses that factory to create implementation-layer widgets.

If you have an interface layer widget, the implementation widget can be obtained using the `_impl` attribute of that widget.

Native

The lowest layer of Toga is the native layer. The native layer represents the widgets required by the widget toolkit of your system. These are accessed using whatever bridging library or Python-native API is available on the implementation platform. This layer is usually provided by system-level APIs, not by Toga itself.

Most implementation widgets will have a single native widget. However, when a platform doesn't expose a single widget that meets the requirements of the Toga interface specification, the implementation layer will use multiple native widgets to provide the required functionality.

In this case, the implementation must provide a single "container" widget that represents the overall geometry of the combined native widgets. This widget is called the "primary" native widget. When there's only one native widget, the native widget is the primary native widget.

If you have an implementation widget, the interface widget can be obtained using the `interface` attribute, and the primary native widget using the `native` attribute.

If you have a native widget, the interface widget can be obtained using the `interface` attribute, and the implementation widget using the `impl` attribute.

An example

Here's how Toga's three-layer API works on the Button widget.

- `toga.Button` is defined in `core/src/toga/widgets/button.py`. This defines the public interface for the Button widget, describing (amongst other things) that there is an `on_click` event handler on a Button. It expects that there will be *an* implementation, but doesn't care which implementation is provided.
- `toga-gtk.widgets.Button` is defined in `gtk/src/toga-gtk/widgets/button.py`. This defines the Button at the implementation layer. It describes how to create a button on GTK, and how to connect the GTK `clicked` signal to the `on_click` Toga handler.
- `Gtk.Button` is the native GTK-Python widget API that implements buttons on GTK.

This three layered approach allows us to change the implementation of `Button` without changing the public API that end-users rely upon. For example, we could switch out `toga-gtk.widgets.Button` with `toga-cocoa.widgets.Button` to provide a macOS implementation of the Button without altering the API that end-users use to construct buttons.

The layered approach is especially useful with more complex widgets. Every platform provides a Button widget, but other widgets are more complex. For example, macOS doesn't provide a native DetailedList view, so it must be constructed out of a scroll view, a table view, and a collection of other pieces. The three layered architecture hides this complexity - the API exposed to developers is a single (interface layer) widget; the complexity of the implementation only matters to the maintainers of Toga.

Lastly, the layered approach provides a testing benefit. In addition to the Cocoa, GTK, and other platform implementations, there is a "dummy" implementation. This implementation satisfies all the API requirements of a Toga implementation layer, but without actually performing any graphical operations. This dummy API can be used to test code using the Toga interface layer.

PYTHON MODULE INDEX

t

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z