LINUX VOICE TUTORIAL

URWID: CREATE TEXT MODE INTERFACES

VALENTINE SINITSYN

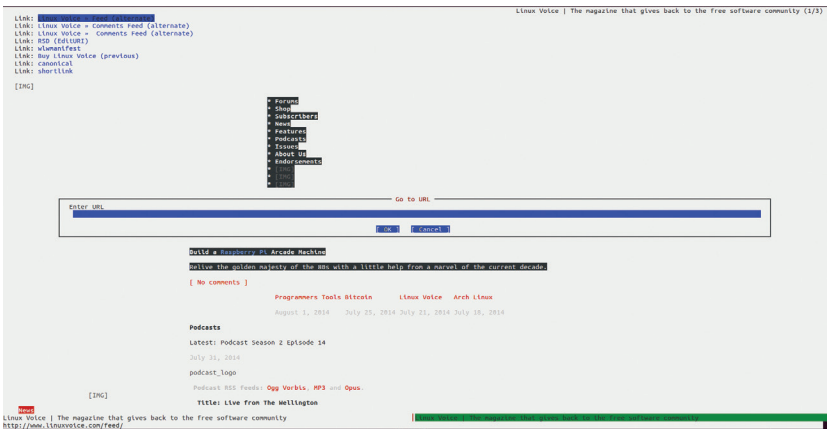
Text-mode user interfaces do not belong to museums yet – find out why and craft one yourself.

WHY DO THIS?

- Create easy to use, lightweight interfaces.
- Rewrite dialog(1)-based shell scripts in Python.
- Learn Linux beyond the desktop.

Today, one can hardly imagine the PC without a graphical desktop. Even the smallest computers such as the Raspberry Pi have an HDMI port and a CPU powerful enough for a graphical environment. Text (or console) user interfaces (TUI) may feel like a weird artefact from ye olden days that fit a museum stand better than your monitor. Sure, you are unlikely to use a terminal to chat on Facebook (although you can surf the web with the *Links* browser if you wish), or write a report (*Latex* can award you with state-of-the-art documents). Nevertheless, console-based programs come in handy where you don't have graphics configured (in installers or setup tools) or work on slow connections (say, you SSH into your Raspberry Pi-based sensor somewhere in countryside available over a 2.75G cellular network only). Text interfaces are also often preferable for specialised applications, like point-of-sale terminals.

This tutorial is about making console interfaces in Python with the *Urwid* library. If you've ever done any programming with *Qt*, *GTK* or any other toolkit, you will find many concepts similar, but not the same. That's because *Urwid* is, strictly speaking, not a widget toolkit. It's a widget construction toolkit, and this subtle difference sometimes matters. It provides the elements of a user interface that you'd expect, like buttons or text input boxes. But many advanced widgets, say dialogs or drop-down menus, are missing (you do them yourself, and we'll show you how in a minute). There is also no straightforward way to set the "tab order" (ie how the focus moves with Tab key). This doesn't mean that *Urwid* is limited or primitive – it's a full-fledged library with mouse support, third-party IO loop integration and other services that you might expect from a mature toolkit – but it's a peculiarity to keep in mind when you program with it.



There are TUI eqiuvalents for many graphical programs, including browsers.

Widget types

One task that a widget toolkit performs is calculating positions and screen space for widgets. This is not as simple as it may sound, and there's no one-size-fits-all recipe either. Some older libraries tended to avoid this job altogether, so if a label was too long to display, it was simply cut off.

Urwid's approach is to introduce three types of widgets. The first one, "box", takes as much space as its container allocates; a top-level widget in *Urwid* application is always a box one. Flow widgets are given a number of columns to occupy, and are responsible for calculating the number of screen rows they need (as we are working in text mode, units are characters, and widget size is measured in rows and columns, not pixels). Fixed widgets are, er, fixed: they always occupy the same screen space regardless what is available, and they decide on their size themselves. A typical example of a flow widget is Text; common boxed widget is SolidFill, which fills an area with the given character and is useful for backgrounds. Fixed widgets are rare, and we won't discuss them.

There are also "decoration widgets" that wrap other widgets and alter their appearance or behaviour. In this way, flow widgets can be made boxed (for

In a timely manner

The main loop is not only the dispatcher of events, but also a timer. These two roles may seem distant, but they are closely related if you descend to the system calls level.

We won't go that deep here, but instead will see how to use timers in *Urwid*. Actually, it's quite simple, and the API resembles JavaScript's **window.setTimeout()**:

```
def callback(main_loop, user_data):
        # I'm to be called in 10 seconds
handle = main_loop.set_alarm_in(10,
    callback, user_data=[])
```

user_data is for passing arbitrary values to your callback; if you don't need it, simply omit the argument. There is also **set_alarm_at()**, which schedules an alarm at the given moment. If you don't need an alarm anymore, you can remove it with:

```
main_loop.remove_alarm(handle)
```

Alarms in *Urwid* are not periodic, so there is no need to remove the alarm that was already triggered.

instance, with Filler, which fills rows left unused by its child) or vice versa (see BoxAdapter). All of these types are visually summarised in the "Included Widgets" section of the *Urwid* manual (**http://urwid.org/manual**).

Sometimes you misuse widgets and put a box one where a flow widget is expected, or whatever. *Urwid* is not very friendly in this case, and all you get is a cryptic **ValueError** exception:

```
... Few other calls here ...
            File "/path/to/urwid/widget.py", line 1004, in
render
  (maxcol,) = size
ValueError: too many values to unpack
```

It originates from the way widgets are rendered. You don't need to dig into details of this backtrace, just remember that if you see it, you've probably missed a decoration widget.

Hello, Urwid world!

It's time to write some code. Like many other (if not all) UI frameworks, *Urwid* is built around the main loop, represented by the MainLoop class. This loop dispatches events such as key presses or mouse clicks to the widget hierarchy rooted at the topmost box widget, passed as the first argument to the MainLoop constructor (and available later as a 'widget' attribute on the main loop object). In this way, a simplest *Urwid* program might look like this:

```
from urwid import MainLoop, SolidFill
mainloop = MainLoop(SolidFill('#'))
mainloop.run()
```

This will fill the screen with hashmarks. The **run()** method is where the main loop starts. To terminate it, raise the **ExitMainLoop** exception:

```
def callback(key):
        raise ExitMainLoop()
mainloop = MainLoop(SolidFill('#'),
          unhandled_input=callback)
```

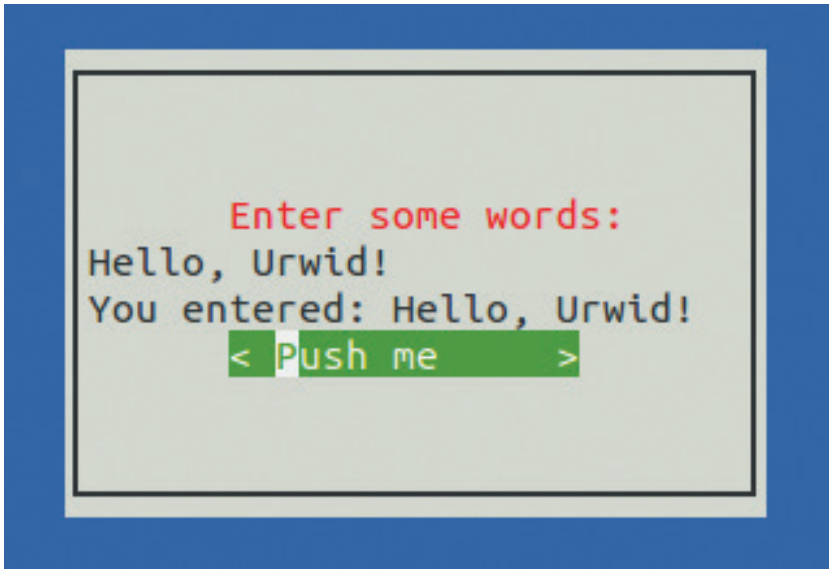
unhandled_input callback is executed for any event that is not handled by the topmost widget (or its descendants). Since **SolidFill()** doesn't respond to keypresses, any key will stop the program. You can check this yourself – just make sure you have installed *Urwid* with your package manager (it's called **python-urwid** or similar).

Add some colour

Black and white text is boring. *Urwid* can paint colours, but it needs a palette first:

```
single_color = [('basic', 'yellow', 'dark blue')]
mainloop = MainLoop(AttrMap(SolidFill('#'),
    'basic'), palette=single_color)
```

Here, the palette contains a single colour: yellow text on a blue background. You can define a palette with as many colours as you want, but keep in mind that not all colours (and attributes) are supported by all terminals. If you don't target a specific environment, it is better to stick to "safe" colours, as defined in the "Display Attributes" section of the *Urwid* manual.

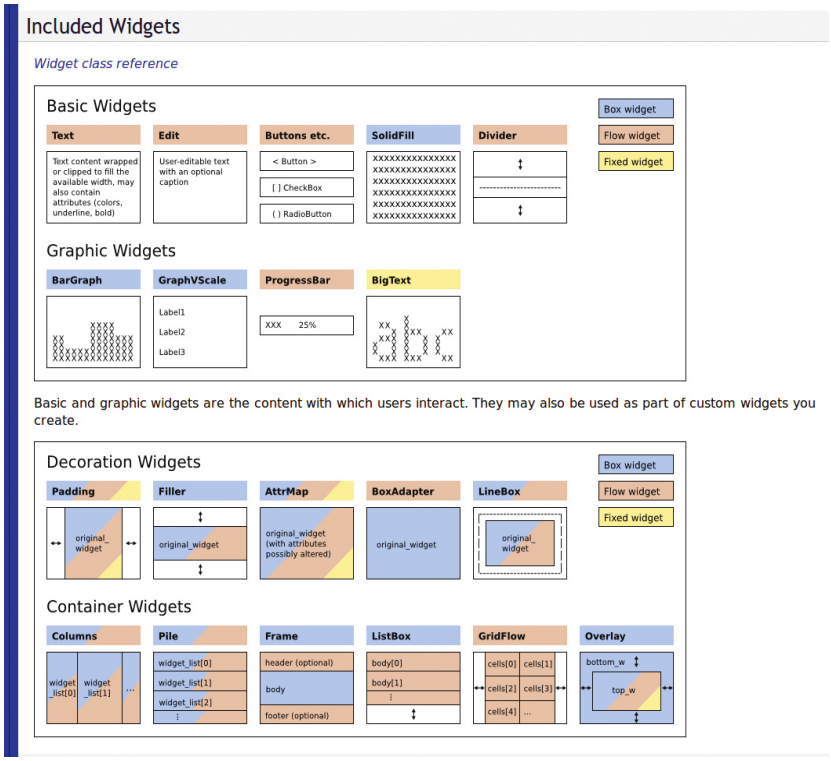


Our first *Urwid* program: basic, but fully functional.

The **palette = keyword** argument installs the palette for your application, but the **AttrMap** decoration widget is where the colour is actually applied. **'basic'** serves as an identifier, and can be anything you want.

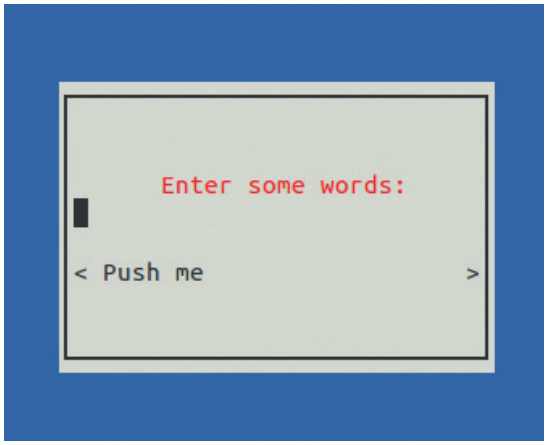
Let's open windows

Programs usually interface with users via some dialog windows. In text mode, they look like framed rectangular areas, so let's create one. To make things more interesting, we'll also include a few basic widgets. A blue background can be created with **SolidFill(' ')** the usual way (let's creatively call this widget 'background'). To create a framed area, we can use the **LineBox()** decoration widget (don't forget to import widgets from the **urwid** package as they appear in the text):



The *Urwid* manual has a neat refresher for widget types and more.

By default, **Pile** stretches widgets to the whole parent's width.




```
window = LineBox(interior)
```

By default, **LineBox** draws a single line around the supplied widget; however, you can configure every aspect of the frame using Unicode box drawing characters (**http://unicode-table.com/en/#box-drawing**). Forget about the 'interior' widget for now – we'll get to it shortly. But for now, how do we put the dialog over the background? *Urwid* provides the **Overlay()** widget for that:

```
topw = Overlay(window, background,
    'center', 30, 'middle', 10)
main_loop = MainLoop(topw,
            palette=some_palette)
main_loop.run()
```

This lays out a 30x10 window centred on the background and starts the main loop. Note that we've used **Overlay** as the topmost widget. Should we need to change the view, the **main_loop.widget** is to be set to something different.

Now, back to the 'interior'. We want some labels (**Text**), an input (**Edit**), and a push button (**Button**) stacked vertically one over another. The way to do it in *Urwid* is to use a Pile container:

```
caption = Text(('caption', 'Enter some words:'),
    align='center')
input = Edit(multiline=False)
# Will be set from the code
scratchpad = Text('')
button = Button('Push me')
button_wrap = Padding(AttrMap(button,
            'button.normal', 'button.focus'),
            align='center', width=15)
interior = Filler(Pile([caption, input,
            scratchpad, button_wrap])
```

Here, we see two new ways to apply attributes (colours). The **Text** widget can accept a markup (a tuple or a list of tuples), and **AttrMap** can assign different attributes to focused and unfocused widgets. As we create widgets, we store them in variables for further reference.

If you try to run this code now, you'll see it fails with the **ValueError** we've already discussed. This is because the **Pile** widget's type is determined by its children, and **Text**, **Edit** and **Button** are flow widgets. **LineBox** works the same way, so finally 'window' is a flow widget in our program. However, the way we use **Overlay** implies that the top widget is a box one (since we allocated both the width and height for it ourselves), and this is the problem. We need to wrap 'interior' into something to make it boxed. The natural choice is **Filler**: we'll let flowed interior widget decide how many rows it needs, and **Filler** will take the rest. By default, **Filler** centres its contents vertically, and this is also what we want:

```
interior = Filler(Pile([...]))
```

Now the program runs; however, the button is wider than needed. That's because **Pile** makes all children equal width, so the button needs some padding:

```
button_wrap = Padding(AttrMap(...),
    align='center', width=15)
```

By default, **Padding** makes contents left-aligned, so we explicitly tell it we need them centred. Width can be an integer (the exact number of columns for the contents), 'pack' (try to find optimal width, which may not work out), or ('relative', percentage) if you want the contents to scale with the container.

Now, the interface looks as needed, however, it still does nothing. Let's change the scratchpad's contents when the button is clicked (either with the Enter key or with the mouse):

```
from urwid import connect_signal
def button_clicked(button, user_data):
        input, scratchpad = user_data
        scratchpad.set_text('You entered: %s' %\
                input.edit_text)
connect_signal(button, 'click', button_clicked,
    [input, scratchpad])
```

We pass references to **input** and **scratchpad** in **user_data**; in real-world code they will likely be some object's attributes. If you no longer want the button to work, you can disconnect the signal with the **disconnect_signal()** function. For **Button**, you can achieve the same results with the **on_press=** and **user_data=** constructor arguments, however the approach we just saw works for any event and widget (for example, Edit emits a 'change' signal when the text is changed).

Our simple program is now fully functional, except that there's no way to exit from it. We can reuse the **unhandled_input** trick, but this time, let's exit only if the user presses the F10 key:

```
def unhandled_input(key):
    if key == 'f10':
```

Walking through the lists

ListBox doesn't dictate how the contents (including focused widgets) are stored: it simply manages them using the **ListWalker** interface. The latter is quite simple, and there are some stock *Urwid* classes that already implement it (like the **SimpleFocusListWalker** we saw), but you can always create your own. This is reasonable when **ListBox** contents are unsuitable to store in a Python list as a whole: they are large, take a long time to receive or whatever else. *ListWalker* solves the problem by providing the way to get (or set) the current (focused) item, and to retrieve siblings for any position in the list. This is enough to display the currently visible part of the contents. For more details, look at the **fib.py** and **edit.py** examples that ship with **Urwid**.

```
            raise ExitMainLoop()
```

If you want to, you can also add another button to close the application.

A secret weapon

As we've already learned, *Urwid* is missing many advanced widgets. However, it includes one very powerful one: **ListBox**. You might imagine a box with a few lines of text and a highlighting bar, but *Urwid*'s **ListBox** is different (although it can look and behave this way as well). It's a scrollable list (or even tree) of arbitrary widgets that's generated dynamically, and it can serve various purposes, including creating menus, sequence editors and almost anything else (except coffee makers, you know).

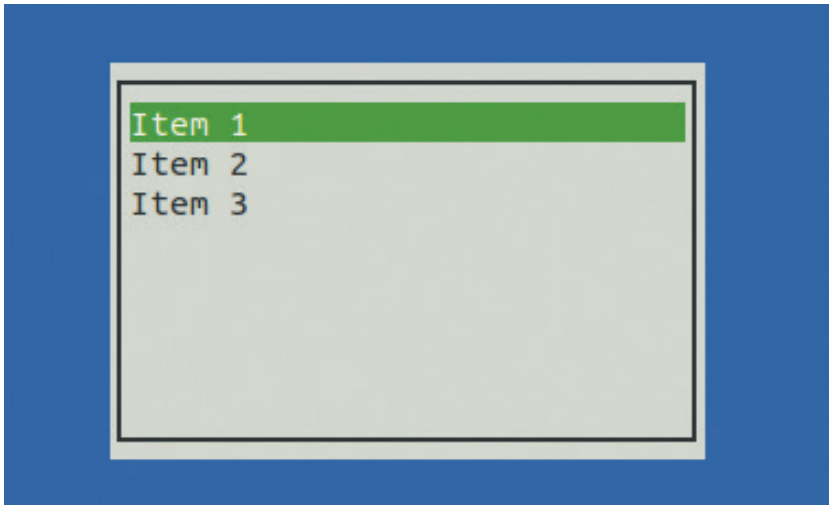
ListBox is a bit like **Pile** in that it takes a list of widgets and stacks them vertically. However, there are many discrepancies, and they are quite important. First, passing **ListBox** a list of widgets is the most simple, limited and somewhat discouraged way to set its contents. Second, **ListBox** is always a box widget that contains flow widgets; in other words, it decides what part of the contents will be shown at given time. To make this decision, **ListBox** manages focus: if, for instance, you press the Down key, the focus will be shifted to the next child, and its contents will be scrolled accordingly.

While **ListBox** is a real Swiss Army knife, we'll use it to create a simple menu. Let's start with the **MenuItem** class. A simple menu item is just a text label that's highlighted when it has focus and responds in some way to activation (like pressing the Enter key). This means the Text widget is a perfect base class for it. We need to register a signal (let's call it 'activate'), intercept the Enter key and make the widget selectable (that's a basic property of all widgets in *Urwid*; only selectable widgets receive focus from the **ListBox** container).

```
from urwid import register_signal, emit_signal
class MenuItem(Text):
        def __init__(self, caption):
            Text.__init__(self, caption)
            register_signal(self.__class__, ['activate'])
        def keypress(self, size, key):
            if key == 'enter':
                    emit_signal(self, 'activate')
            else:
                    return key
        def selectable(self):
            return True
```

Signals are registered per-class with **register_signal()** and emitted with **emit_signal()** later. The **keypress()** method is defined in the base **Widget** class and overridden by all widgets that want to respond to the keyboard (its size is the current widget's size). If the widget successfully handled the key it returns **none**, or **key** otherwise. There is a similar **mouse_event()** method, but we won't discuss it here.

Next, we need to pack **MenuItem** objects into **ListBox**. To make current focus visible, we'll use an **AttrMap** the same way we did it for the button earlier:



ListBox is a natural choice for, er, a list box widget.

```
def exit_app():
        raise ExitMainLoop()
contents = []
for caption in ['Item 1', 'Item 2', 'Item 3']:
        item = MenuItem(caption)
            connect_signal(item, 'activate', exit_app)
            contents.append(AttrMap(item,
                    'item.normal', 'item.focus'))
interior = ListBox(SimpleFocusListWalker(contents))
```

This assumes that the overall program layout is the same as in the previous example; however, since **ListBox** is box widget, there is no need to wrap 'interior' with **Filler**. We connect the 'activate' signal to the **exit_app()** function that simply terminates the program.

The **SimpleFocusListWorker** class is a basic adapter to make **ListBox** work on top of a static widget list. It derives from **ListWalker**, and you can use its other subclasses here, including the ones you create yourself, as well. The primary reason to do this is to make the contents of **ListBox** dynamic, for example, read lines from a file only when the user scrolls down to them. This is where **ListBox** comes to its full powers.

Where to go next?

That's basically all for the introduction. There are some concepts, like text layout or canvas cache, that we haven't discussed, and there are others we've touched only briefly. However what you've learned today will hopefully help you to master more advanced concepts quickly. Should you need to create a sophisticated *Urwid* UI, bundled examples and existing applications (**http://excess.org/urwid/wiki/ApplicationList**) are great resources for *Urwid* programming ideas and techniques. Just don't forget to post your *Urwid* toolbox to some code hosting site for community's benefit, too!

Dr Valentine Sinitsyn has committer rights in KDE but prefers to spend his time mastering virtualisation and doing clever things with Python.