

CODE FOR ALL

Programming isn't just the domain of übergeeks – in the Year of Code, everyone can get involved. Follow our guides and create programs for the desktop, mobile phone and command line.

It wasn't long ago that programmers were viewed by the general public as strange wizards who spent hours alone every day, tapping incomprehensible gobbledygook into black boxes on the screen. But in recent years, the perception of programming as a hobby (and profession) has changed enormously. Today it's cool to hack the Raspberry Pi. Today it's trendy to write software for iOS and Android.

Today you can tell someone at the pub that you enjoy programming, and not be looked at like you've just started speaking Tagalog.

We're all hackers on the Linux Voice team, so we support every effort to inspire people to code. It's not all boring, complicated and alien like some people claim; coding can be fun, stimulating and useful for developing future skills.

With this in mind, we wanted to make this issue's cover feature all about coding. But not just as a generic introduction to a language or platform – no, we wanted to show you how to do useful things. So over the next nine pages we have three projects explaining how to make real-world desktop, mobile phone and command line applications. As you'll see, coding is for everyone, and you can code for any platform.

Desktop apps with Python

First step: building the required skills.

To kick off, we're going to use Python, because it's an excellent all-round language which combines highly readable code with oodles of advanced features. Python code tends to be self-explanatory, so it's a great way to dip your feet into programming. We'll start here with a quick overview of the language; if you're already familiar with Python, you may want to quickly skim over this and then turn over the page, where we start using it in our application.

A Python primer

Python is installed by default in most major distributions, and it's an interpreted language, so you don't need to compile your code before running it. In the following examples, save the source code in plain text format as **test.py**. Then run it like so:

python test.py

Let's start with a very simple program:

```
name = "Linux Voice"
print name + " is the best Linux mag"
```

This demonstrates two aspects of the language: variables and output. In the first line, we create a new variable (like a storage space) called **name** – in Python, you don't need to explicitly state the type of a variable when you create it. In the second line, we print the contents of the variable to the screen, along with another string of characters. Dead easy, right?

Let's look at numeric variables:

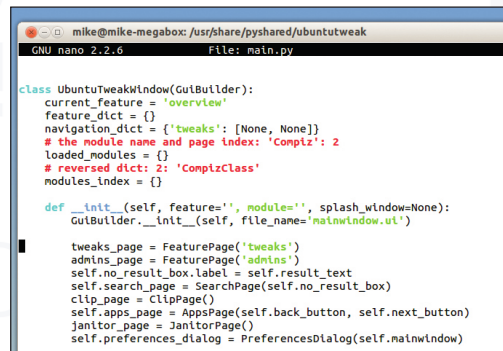
```
x = 1
while x < 10:
    print "x is", x
    x = x + 1
print "Finished!"
```

Here we declare the **x** variable to contain 1 at the start, and then begin a loop. While the contents of **x** are less than 10, we print the contents, and add one to **x**. It's important to note the indentation here: in Python, you use tabs to say which lines belong to a chunk of code. Here we say that the **print** and **x = x + 1** lines belong in the **while** loop, because they're indented. If you removed the tab from the **x = x + 1** line, the contents of **x** would never be incremented in the loop, so the loop would go on forever.

When the loop has finished, the program continues, so no indentation is required. (If you have loops inside loops, you will have multiple levels of indentation.) Next, let's move on to input and comparisons:

```
x = input("Enter a number: ")
if x == 1:
    print "You entered 1"
else:
    print "That wasn't 1"
```

Here we have our **x** variable again, and we call Python's in-built **input** routine, which displays the



```
mike@mike-megabox: /usr/share/pyshared/ubuntuTweak
GNU nano 2.2.6 File: main.py

class UbuntuTweakWindow(GtkBuilder):
    current_feature = 'overview'
    feature_dict = {}
    navigation_dict = {'tweaks': [None, None]}
    # the module name and page index: 'Comptz': 2
    loaded_modules = {}
    # reversed dict: 2: 'ComptzClass'
    modules_index = {}

    def __init__(self, feature='', module='', splash_window=None):
        GtkBuilder.__init__(self, file_name='mainwindow.ui')

        tweaks_page = FeaturePage('tweaks')
        admins_page = FeaturePage('admins')
        self.no_result_box.label = self.result_text
        self.search_page = SearchPage(self.no_result_box)
        clip_page = ClipPage()
        self.apps_page = AppsPage(self.back_button, self.next_button)
        janitor_page = JanitorPage()
        self.preferences_dialog = PreferencesDialog(self.mainwindow)
```

Many good text editors include syntax highlighting, to make it easier to read code – here's Nano, for example.

specified text in quotes. Then we ask Python: if the contents of **x** are 1 after the input, print one thing, and if the contents are not 1, (shown in the code as the **else** statement) print something else. Why the **if x == 1** though – why the double equals signs? Well, it's just to make a very clear distinction from **x = 1** (a single equals sign), which stores a number or line of text within a variable.

Funky functions

Next, we'll dip our toes into functions. These are self-contained chunks of code that you can use to build bigger programs. You can re-use them with different parameters, to keep your code small and easy to understand. For instance:

```
def multiplier(a, b):
    print a, "multiplied by", b, "is: "
    print a * b
multiplier(6, 7)
multiplier(10, 20)
multiplier(211, 2352)
```

Here, we **def**ine a function called **multiplier**, which receives two numbers from the calling program, and stores them in the variables **a** and **b**. This function then prints out the result of multiplication (using the ***** operator). Note the tab indentation again here, and also note that this function is defined at the start of the program, but it isn't executed immediately – Python starts execution in the non-indented part.

So we can call our **multiplier** function with different numbers, as shown in the three lines at the bottom of the code. This is a trivial example

(you could just do the multiplications in the main code), but it shows how you can build up programs from self-contained units.

So, that's the basics of Python covered – now turn the page and let's do something useful.

"Python is installed by default in most major Linux distributions."

A kiosk-like web browser

Write your own locked-down, ultra secure web browser.

With our Python skills freshly prepped, let's make a real application. Here we're going to create a simple, and locked-down web browser than can only visit certain web pages and not escape onto the big, bad web. Why would you do this? Well, let's say you're setting up a web terminal for a school, shop or museum, and you want to restrict access to certain places. You could use a normal web browser, load it up with kiosk-like extensions and filtering proxies and hope that it's secure, but clever users may still be able to break out of the restrictions and cause trouble.

With our browser, we have just the bare essentials. And even if setting up a web kiosk isn't high on your list of things to do, it's well worth following this tutorial to see how a simple web browser is implemented.

The code

Here it is – a whole web browser, contained within 35 lines of Python. You probably don't want to type this out by hand, so grab it from www.linuxvoice.com/code/microbrowser.py. This browser is set up to view the Debian website at www.debian.org and nothing else, but you can of course change that.

```
import gtk, webkit
```

```
def goback(button):
```

```
    view.go_back()
```

```
def navrequest(thisview, frame, networkRequest):
```

```
    address = networkRequest.get_uri()
    if not "debian.org" in address:
        md = gtk.MessageDialog(win, gtk.
            DIALOG_DESTROY_WITH_PARENT, gtk.MESSAGE_INFO, gtk.BUTTONS_CLOSE, "Not allowed to leave the site!")
        md.run()
        md.destroy()
        view.open("http://www.debian.org")

view = webkit.WebView()
view.connect("navigation-requested", navrequest)

sw = gtk.ScrolledWindow()
sw.add(view)

button = gtk.Button("Back")
button.connect("clicked", goback)

vbox = gtk.VBox()
vbox.pack_start(button, False, False, 0)
vbox.add(sw)

win = gtk.Window(gtk.WINDOW_TOPLEVEL)
win.set_size_request(800, 600)
win.connect("destroy", gtk.main_quit)
win.set_title("Linux Voice browser")
win.add(vbox)
win.show_all()

view.open("http://www.debian.org")
gtk.main()
```

Now, we're not going to write the entire HTML, CSS and JavaScript rendering engine ourselves – that would take months of hard effort and fill many issues of the magazine. No, we'll leave that to WebKit, an extremely capable rendering engine used in several notable browsers such as Chrome/Chromium and Safari. (Well, Chrome now uses the Blink rendering engine, but this is based on WebKit.)

There's a great Python module to interface with WebKit, which we're using here: you'll find it in the **python-webkit** package in Debian and Ubuntu-based distros. You'll also need **python-gtk2** for the interface. So let's step through the code:

```
import gtk, webkit
```

This is simple enough – it just tells Python that we want to use the GTK module to provide the GUI widgets, and the WebKit module for the rendering engine. Then we have two functions:

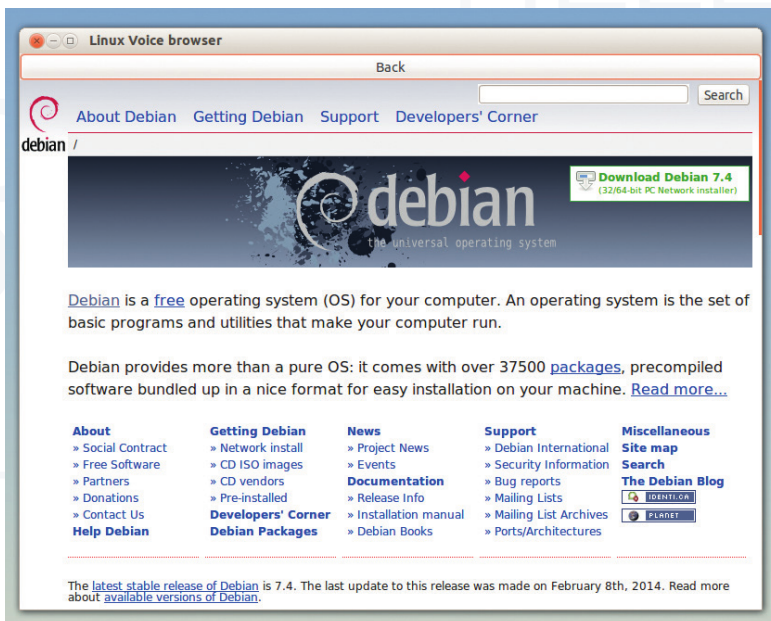
```
def goback(button):
```

```
...
```

```
def navrequest(thisview, frame, networkRequest):
```

```
...
```

We'll come back to these in a moment, because first



Here it is: our funky mega skillo web browser, which we can restrict to wherever we want, written in just 35 lines of Python. How cool is that?

we need to set up some code to use them. So execution of the program begins here:

```
view = webkit.WebView()
view.connect("navigation-requested", navrequest)
```

This is the heart of the program. We create a new object called **view** (an object is like a variable, but it can store and do a lot more), which is a WebKit WebView. Put simply, this is a plain web browser view – but with no buttons, no surrounding window, or anything like that. It's just a canvas to render web pages inside.

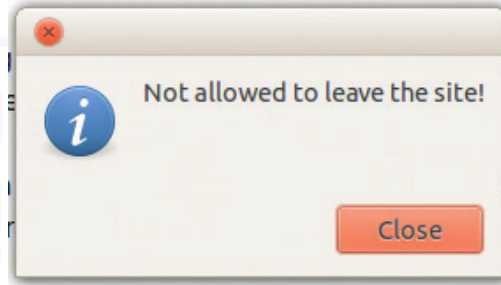
The second line of this snippet is crucially important, and demonstrates a callback function. We tell our web page view that if a navigation event occurs (ie the user clicks on a link), we should call the **navrequest** function as included at the start of the code. This function retrieves the address of the clicked link (**networkRequest.get_uri**) and checks to see if **debian.org** is contained in the address. If not, we show a dialog box and go back to the Debian home page. (Yes, this doesn't make it 100% impossible for people to escape **www.debian.org**, but you could narrow down the allowed links even further with regular expressions – that's beyond the scope of this guide though!)

So, we have a web browsing pane which checks links as they're clicked. Next are these lines:

```
sw = gtk.ScrolledWindow()
sw.add(view)
button = gtk.Button("Back")
button.connect("clicked", goback)
```

As mentioned, the WebKit view isn't attached to anything yet – it just exists in memory somewhere and that's it. Here we attach it to a GTK scrolling window so that users have scroll bars to move around in the page. We create a new **ScrolledWindow** and add the **view** object to it.

Then we create a new GTK button with the label **Back**, and also attach it to a callback function we wrote earlier: **goback**. So whenever the user clicks this button, the function **goback** is called. In that function, we tell the view to step back a page (**view.go_back()**) and return to the main code.



And here's what happens if users try to stray beyond the limits, as we defined in the **navrequest()** function.

Now we have to pack the button and WebKit view into a single space, for which we use a vertical box widget in GTK:

```
vbox = gtk.VBox()
vbox.pack_start(button, False, False, 0)
vbox.add(sw)
```

Then the following six lines, beginning with **win**, create a new application window, set its size, say what to do when the close button is clicked and set a title. They also add the vertical box widget to the application window and make sure that all of the widgets inside it are visible. And the last two lines are:

```
view.open("http://www.debian.org")
gtk.main()
```

Here we tell the WebKit view to open a specific page, and run GTK's main event loop (where it watches for button clicks and window operations – we leave it alone from here).

And that's it! We've crammed a lot in here, so if you have any questions, check out the websites for Python GTK (www.pygtk.org) and Python WebKit (<https://code.google.com/p/pywebkitgtk/>). If you need further help, post on our forums at <http://forums.linuxvoice.com> and we'll do our best to answer.

“There's a great Python module to interface with WebKit, which we're using here.”

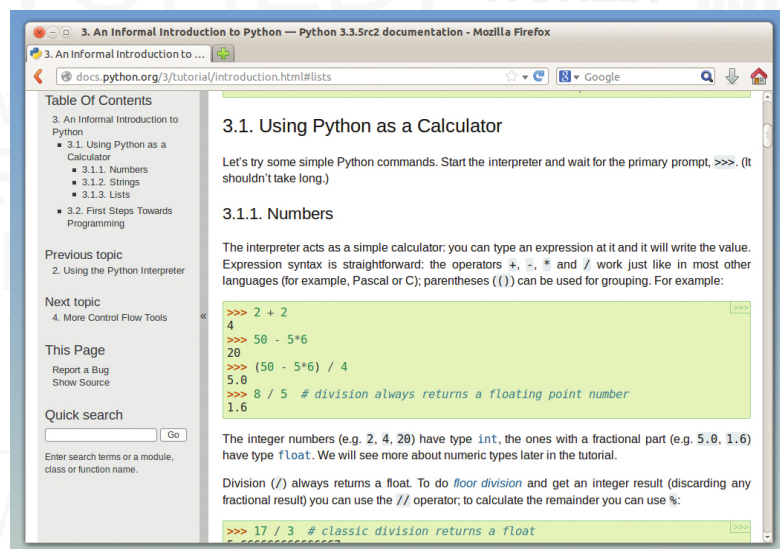
Want to delve into more advanced Python topics? See <http://docs.python.org> for a wealth of tutorials.

Making Python apps directly executable

It's easy to run our browser from the command line by typing **python microbrowser.py**, but what if you want to make it directly executable – so you can run it by simply double-clicking on it? The trick is to add this to the top of the file:

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

Now make the file executable (eg **chmod +x microbrowser.py**) and run it (**./microbrowser.py**). This extra first line tells the operating system which interpreter should be used with the following code, so you don't need to specify Python manually at the command line. In your kiosk setup, you can easily trim down a desktop or window manager to the bare essentials, and add a single launcher pointing at **microbrowser.py** somewhere on your system.



Mobile Linux

Don't rely on the app store for software – create your own.

Over the last few years, Linux has taken over the mobile computing marketplace. Android is hugely popular, and there's also Amazon's Fire OS, Firefox OS, Sailfish OS, Tizen, Bada and soon there'll be Ubuntu Touch as well. Developing for mobile platforms isn't like desktop Linux, where the same code is just repackaged for different distros. This allows developers to make their apps fit the look and feel of the OS, but it also means that you have to spend a long time developing for the different devices.

Fortunately, it doesn't have to be this way. It is possible to maintain a single codebase for use across all the Linux-based mobile platforms (and a few non-Linux ones like iOS and Windows Phone as well). This way is Apache Cordova.

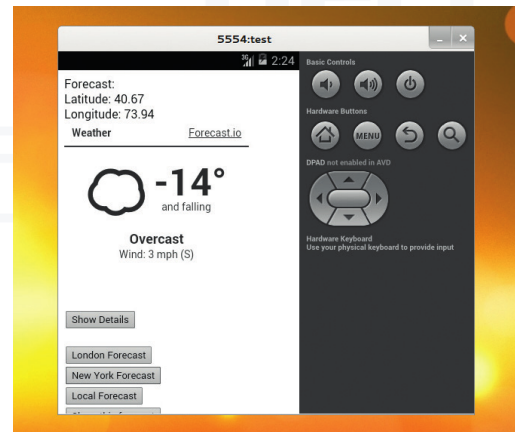
Apache Cordova is a framework that enables you to develop in HTML and JavaScript, then package it up for each different environment.

While it may once have been suitable only for displaying web pages, and perhaps a little interaction, JavaScript has grown in to a powerful programming environment. If you don't believe us, have a look at these examples and see for yourself:

- The Unreal Engine 3 has been ported to JavaScript and you can explore Epic Citadel in your browser: www.unrealengine.com/html5.
- Tearable cloth: <http://codepen.io/suffick/pen/KrAwX>.
- Freerider II: www.freeriderhd.com/t/1016-layers.

With the canvas HTML 5 element, you can make 2D JavaScript graphics as complex as you like, and with WebGL, you can even harness the power of the device's GPU to create accelerated 3D graphics. WebRTC can be used to set up a communication channel between two browsers, and Webaudio helps you add sound to your creations. We should point out that not all of these features are as yet possible with Cordova on all devices (even new ones), but it's only a matter of time.

The performance of JavaScript has long been considered a problem for web app development, but in recent years, this has improved dramatically. Now, well-written JavaScript should perform at about half



the speed of natively compiled code. This means that it's still not quite there for some high-performance applications, but it should be fine for most cases.

The people at Mozilla know more than most about the power of HTML and JavaScript, and they believe in it so much that they built an entire phone operating system built around it.

Another great advantage of the HTML and JavaScript approach is that it makes it really easy to get started. You can create simple pages in point-and-click HTML editors, then progress onwards as you learn more about programming and the environment.

First build

To get started, you'll need the appropriate SDK for the platform (or platforms) you're developing for, and the appropriate version of Cordova (a full list can be found at cordova.apache.org/#download). In this tutorial we're going to use Android, as it's the most popular mobile Linux, but you shouldn't have any problem transferring the work to a different platform such as Ubuntu Phone or Firefox OS. Sailfish OS should be able to run Android apps, but we haven't been able to test this particular app.

You'll need the SDK for every environment you're developing for. For Android, you can get it from <http://developer.android.com/sdk>. You'll get a ZIP file that you can extract. You need to add the **sdk/** **platform-tools** and **sdk/tools** directories from this ZIP to your path. In the author's environment, this was with the following command, though you'll have to change it depending on where you unzip the SDK:

```
export PATH=$PATH:/home/ben/Downloads/adt-bundle-linux-x86-20131030/sdk/platform-tools:/home/ben/Downloads/adt-bundle-linux-x86-20131030/sdk/tools
```

PhoneGap

Cordova is closely related to Adobe PhoneGap. In fact, they're so closely related that they're often mistaken for one another. Officially, PhoneGap is an implementation of Cordova, but it doesn't add much to the core release in the same way a distribution of Linux adds a lot of software around the kernel. For now we prefer to build on a framework released by the Apache Software Foundation than one developed by Adobe.

Cordova plugins

To access phone features, you'll need to use plugins. The standard ones that come as part of Cordova are: Accelerometer, Camera, Capture, Compass, Connection, Contact, Device, Events, File, Geolocation, Globalization, InAppBrowser, Media, Notifications, Splashscreen and Storage. There's also a good selection of plugins available at <http://plugreg.com>, should the standard ones not do everything you need.

You'll have to re-run this every time you restart your computer unless you add it to the **.bashrc** file in your home directory.

Cordova is a **node.js** application, and you'll need both **node.js** and **npm** for it to run. On Ubuntu and derivatives, this can be done with the following code. If you're using a different distro, check the available packages:

```
sudo apt-add-repository ppa:chris-lea/node.js
```

```
sudo apt-get update
```

```
sudo apt-get install npm nodejs and
```

```
npm config set registry http://registry.npmjs.org/
```

```
sudo npm install -g cordova
```

Cordova works with a specific directory structure. To create a new project directory and appropriate subdirectories run **cordova create myProject**, where **myProject** is the name of the new project. Now you can move into the new directory with **cd myProject**.

Projects start off without any platforms. You can add as many as you like provided you have the SDKs installed, and Cordova will manage the builds for you. We'll just add Android with:

```
cordova platform add android
```

And you can compile the example code (that each project is created with) using:

```
cordova build android
```

In order to test your app, you either need an Android device, or to use an emulator. To create a new emulated device, open the version of Eclipse that came bundled with the Android SDK. Go to File > New > Other > Android > Android Project From Existing Code and select the **platforms/android** folder from your project's directory. Then do to Window > Android Virtual Device Manager and create a new virtual device. Once this is set up, you can run the project from the command line with:

```
cordova emulate android
```

Some of these work far better with a physical device than an emulator, so if you have an Android phone or tablet, it'll be easier to follow the rest of this tutorial on that. The method for doing this varies depending on the version of Android you have. Visit <http://developer.android.com/tools/device.html> for more details. Once you've set up your phone, and connected it to your computer via the USB cable, you can load and run the app with:

```
cordova run android
```

In its basic state, this enables you to package up

HTML and JavaScript for phones. In essence, it allows you to create off-line websites. This certainly has its uses, and many apps are nothing more than this.

Harness phone-specific features

However, to be a true phone app, it should have access to more of the phone's features, such as the GPS, accelerometer or filesystem. These aren't available through normal JavaScript, but Cordova allows plugins that expose certain features to its JavaScript API. Take a look at the boxout above for the standard plugins.

As an example, we're going to create a simple weather forecast app. It'll get the current location, then display a weather forecast for the area. We'll also add the ability to share it using social media, as apparently all good apps do this.

To start with, we need to get the HTML and JavaScript to grab a forecast based on latitude and longitude. Since everything is in the web technologies, it's far easier to test it out using web development tools than the Android-specific ones. Once everything's working properly in a browser, you can then transfer it to Cordova and check it in Android.

We'll start with a really simple HTML doc that just grabs a forecast for London. Change the **www/index.html** file so that it contains:

```
<!DOCTYPE html>
<html>
<head>
<title>Weather Forecast</title>
</head>
<body>
Forecast:
<br>
<iframe id="forecast_embed" type="text/html" frameborder="0"
height="245px" width="245px" src="http://forecast.io/embed/#l
at=51.5072&lon=0.1275&units=uk"> </iframe>
</body>
</html>
```

Save this as **index.html** in the **www** folder of your app. You could run this using Cordova on your phone or an emulator, but it's easier at this stage to open it in your normal web browser.

51.5072, 0.1275 are the coordinates we'll use (this is in London). This grabs an iframe with the current forecast from

forecast.io. In order grab the forecast for the current location, all you need to do is create an iframe with the right latitude and longitude.

Writing iframes in JavaScript is easy, since you can manipulate the HTML inside an element. All you need to do is create a **<div>** **</div>** that you can put the iframe inside. Now we'll add the ability to switch the location between London and New York. First change the code between the **<body>** **</body>** tags to:

“Another advantage of HTML and JavaScript is that it makes it really easy to get started.”

Forecast:

```

<div id="location"></div>
<div id="forecast">Select Location</div>
<br>
<button onclick="getForecast(51.5072, 0.1275)">London
Forecast</button>
<br>
<button onclick="getForecast(40.6700, 73.9400)">New York
Forecast</button>
<br>
<div id="getlocalforecast"></div>
<div id="forecastDetails"></div>

```

This has two divs with different IDs that we'll use now, and some more that we'll use in a bit. We'll use JavaScript to update them to what we need them to be. The two buttons call the **getForecast(latitude, longitude)** function that we'll now define.

Add the following just before the **</head>** tag:

```

<script type="text/javascript">
function getForecast(latitude, longitude) {
    var element = document.getElementById('location');
    element.innerHTML = 'Latitude: ' + latitude + '<br />' +
        'Longitude: ' + longitude + '<br />';
    window.iframeurl='http://forecast.io/embed/#lat=' + latitude +
        '&lon=' + longitude + '&units=uk';
    showSimple();
}

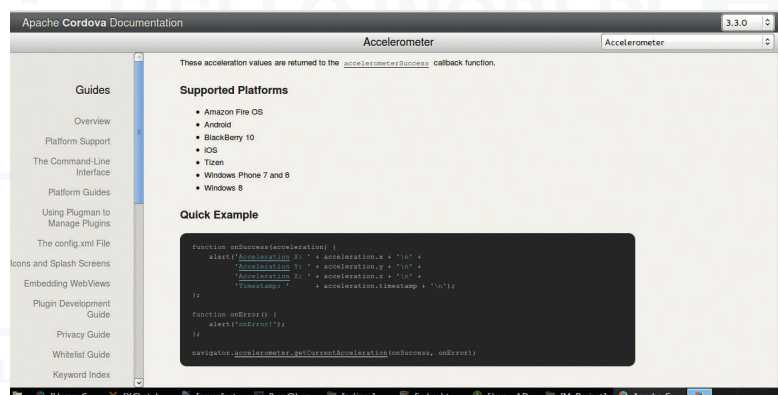
function showSimple() {
    var weather = document.getElementById('forecast');
    weather.innerHTML = '<iframe id="forecast_embed"
type="text/html" frameborder="0" +
        'height="245px" width="245px" src="' + window.iframeurl +
">"> </iframe>';
}
</script>

```

This splits the execution up into two stages. The first sets the contents of **<div id="location"></div>**, and the variable **window.iframeurl**. By defining this variable as attached to **window**, it makes it available to all our functions, rather than just local to the current function. The same effect could have been achieved by using a global variable.

The second function sets the **<div id="forecast"></div>** to be an iframe with the appropriate location. The reason we've split this up into two functions will

The documentation at <http://cordova.apache.org/docs/en/3.3.0/> is a great place to find help. It has guides for all the standard plugins including comprehensive code samples.



Signing apps

Cordova can build a final version of your app using the **--release** option to the build command. However, this won't install on any phone until it's been signed. You can create a key for signing it yourself, so this isn't a restriction on distributing your software. There are details of how to do it here: <http://developer.android.com/tools/publishing/app-signing.html>.

You can distribute your app without an app store if you want. Just send the **.apk** file to people and (as long as they have sideloading enabled) they can install it themselves. Of course, you can put your app on the main Google Play store if. You'll find details about how to do this here:

<http://developer.android.com/distribute/googleplay/publish/preparing.html>.

Google Play isn't the only Android app store though. If you open source your app, you may wish to add it to the F-Droid store. Take a look at www.f-droid.org for details.

become clear later. Again, you can test this out in a web browser and it should work fine.

Not just a website!

Now let's add the phone-specific stuff. Cordova uses plugins to add access to different features, so in order for our app to be able to access the location, we need to use the Geolocation plugin. This is done by running the following command in the root directory of the web app:

```
cordova plugin add org.apache.cordova.geolocation
```

This will add it to every platform you have registered as long as the plugin works on that platform.

In order to access the Cordova features, you need the **cordova.js** script, so add the following line just below **</title>**:

```
<script type="text/javascript" charset="utf-8" src="cordova.js"></script>
```

With this in place, you can add the following functions inside your main **<script>** tag:

```

document.addEventListener("deviceready", onDeviceReady,
false);
function onDeviceReady() {
    localforecast = document.getElementById('getlocalforecast');
    localforecast.innerHTML = '<button onclick="getLocal()">Local
Forecast</button>';
}

function getLocal() {
    navigator.geolocation.getCurrentPosition(onSuccess,
onError);
}

function onSuccess(position) {
    getForecast(position.coords.latitude, position.coords.
longitude);
}

function onError(error) {
    alert('code: ' + error.code + '\n' +
        'message: ' + error.message + '\n');
}

```

The first line listens for the **deviceready** event. This tells it to run the function **onDeviceReady** once the

app is running properly. We've added this to stop people trying to get a local forecast too soon.

The function `getLocal` can just call `navigator.geolocation.getCurrentPosition()`. We've passed it two parameters: the first is the name of the function to call if it succeeds in getting the location; the second is the function to call if there's an error.

`onSuccess` passes the returned values on to `getForecast()`, while `onError()` displays the error message as a JavaScript alert.

With all this entered and saved, it's ready for its first proper test. To compile and run it, enter the following terminal commands in the app's root directory:

```
cordova build android
```

```
cordova run android
```

If you've got your phone attached, this will send it across and open it on your device, otherwise it'll start the emulator.

You've just created a phone app! It's quite limited, but accesses one of the phone's features. Since all good mobile applications have social features, we'll add this facility now. We won't make specific links to social media, but use the phone's features to share the forecast with other applications. The user can then pick how they want to share the forecast.

Engage Twitbook

As you may have guessed, this feature comes from another plugin, but this time it's one that's not part of the main Cordova release. You can add plugins straight from Git, so in a terminal in the app's root directory, enter:

```
cordova plugin add https://github.com/leecrossley/cordova-plugin-social-message.git
```

As with the previous plugin, this exposes more JavaScript functions that we can access. In this case, it's `socialmessage.send()`. Using this, you can interact with the other apps on the phone. Add the following function inside the `<script></script>` tags:

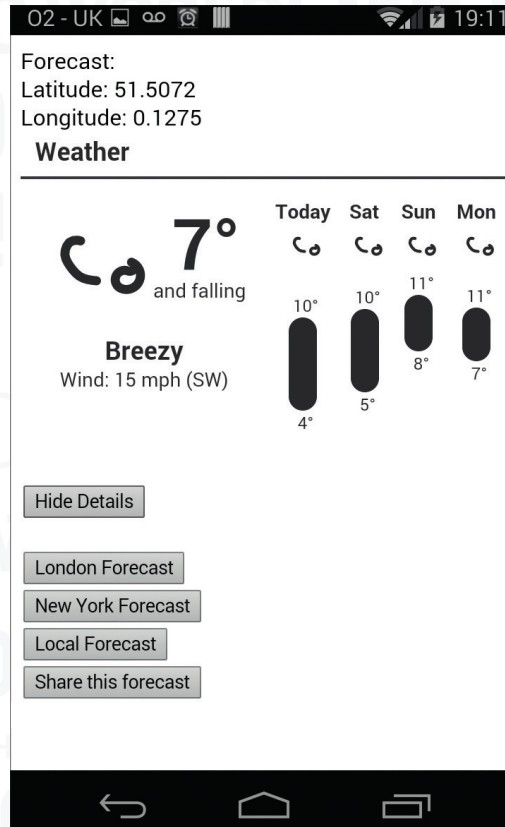
```
function share() {
  var message = {
    subject: "Weather Forecast",
    text: "Check out my local forecast",
    url: window.iframeurl
  }
  window.socialmessage.send(message);
}
```

You'll also need a button in the body of the HTML to access it. However, you can't share the forecast until it's received, so the button should only appear once there's a forecast. The easiest way to do this is by adding the lines:

```
var weatherDetails = document.getElementById('forecastDetails');
weatherDetails.innerHTML = '<button onclick="share()"> +
'Share this forecast</button>';
```

to the end of the `getForecast()` function.

We're almost done with our app now. The last little feature we'll add is the ability to show a simple or



The app in action.

Unfortunately, that's the best weather we've had in months.

detailed forecast. Fortunately, **Forecast.io** does most of the hard work on this. The only thing we have to do is change the size of the iframe.

You'll need to adjust the `showSimple()` function and add `showDetails()` as per the following:

```
function showDetails() {
  var weather = document.getElementById('forecast');
  weather.innerHTML = '<iframe id="forecast_embed"
type="text/html" +
'frameborder="0" height="245px" width="500px" src="' +
window.iframeurl +
"> </iframe><br><button onclick="showSimple()">Hide
Details</button>;
}

function showSimple() {
  var weather = document.
getElementById('forecast');
  weather.innerHTML =
'<iframe id="forecast_embed"
type="text/html" +
'frameborder="0" height="245px" width="245px" src="' +
window.iframeurl +
"> </iframe><br><button onclick="showDetails()">Show
Details</button>;
}
```

“Cordova uses plugins to access different features – we need the Geolocation plugin.”

Of course, it still looks a bit plain, and you could add many more options, but this isn't a tutorial on creating the perfect weather forecasting app, it's a tutorial on getting started with mobile Linux development. It's up to you to decide what to do with it now.

Programming the command line

Automate everything, then sit back and relax as your computer takes care of itself.

So far we've talked about programming in terms of making new software. However, programming can also be a way of linking together existing software to automate tasks. In this way, you don't create anything that you didn't have access to before, but you make it much easier to use. Let's take a really quick example. Suppose you're a writer, and you save all your work in ODT files. These files are scattered about your home directory (because most writers aren't organised enough to keep their files in one place), and you want to do a full backup of all your writing.

There are many ways you could do this. One of the easiest is to create a simple program that searches for all the files and copies them to a remote computer.

Bash is the shell that most Linuxes use, and while many users know it only as a command line environment, it's also a programming language in its own right. We can use it to link up a series of Linux commands to execute based on the information that other commands provide. In this example, we'll use the command:

```
find /home/ben -name "*.odt"
```

To find all the required files. Not surprisingly, the **find** is command for finding things, and is far more powerful than this command shows. Using other options, you can find files based on the time they were created, the time they were last modified, and a whole host of other things. See the man page (type **man find** in a terminal) for more details.

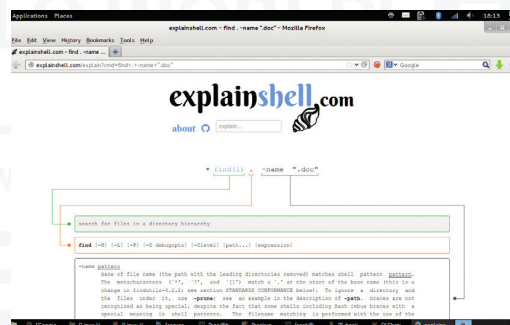
We'll then copy all the files into a backup folder (which could be on an external drive). The bash code to do all this is:

```
#!/bin/bash
```

```
find /home/ben -name "*.odt" | while read f;  
do
```

```
  cp -f "$f" /home/ben/backup  
done
```

You'll need to change **/home/ben** to the location of your home directory.



Explain Shell (www.explainshell.com) is a tool for linking bash commands to their help text.

Lets take a look at this in detail. The first line is called a shebang, and it tells the computer that this is a Bash script and that it should be executed with the command **/bin/bash**.

The second line does two things. First, it executes the **find** command, which we explained above; then it pipes the output of the command into a **while** loop. Piping (which is done using the character **|**) is an essential feature of Bash programming, and it can also be done on the command line. It just tells the system to take the output of one command and feed it into the next. As another example, if you're using a terminal and you're in a directory with loads and loads of files, it sometimes doesn't work very well if you just run **ls** to list them (the filenames can go off the top of the screen). Instead, you can pipe the output into a text viewer such as **less** with:

```
ls | less
```

This allows you to scroll up and down through the list of files. You can also do it for other commands that produce a lot of output.

Digression over

Back to our backup script though. In this case, the program outputs the result of the **find** command into **while read f**. This slightly cryptic statement starts a loop for every line in the output and tells it to store the line in the variable **f**. In other words, everything between **do** and **done** is executed once for every line in the output of the **find** command, that is:

```
cp -f "$f" /ben/backup
```

The **\$f** tells Bash to insert the line output from the **find** command here. It's in quote marks because otherwise filenames with spaces in them will cause problems.

This is a really simple example, but it shows how you can build up scripts in Bash. The two main ways of combining commands are piping output, and running loops over multiple lines. With these two techniques, you can combine all the command line tools in Linux into your own powerful scripts.

Before running it, you have to make the backup directory with:

```
mkdir ~/backup
```

If you save this program as **backup.sh**, you can run it from the command line with:

```
bash backup.sh
```

As long as you are in the same directory that you saved the file. Alternatively, if you make it executable with the command:

```
chmod a+x backup.sh
```

you can run it with:

```
./backup.sh
```

Sometimes, it's not enough just to send the output of one command straight into another. Sometimes

you need to make a decision based on the output that's being processed. For example, what if you didn't want to copy all files straight into the backup directory? What if you wanted to sort them and put different files in different places?

In the next example, we'll find all LibreOffice Writer and Calc files (ODT and ODS respectively) and all MS Office Word and Excel files (DOC/DOCX and XLS/XLSX respectively), and split them into word processor and spreadsheet folders.

This can be done with the following:

```
#!/bin/bash
find /home/ben \( -name "*.odt" -o -name "*.ods" -o -name
 "*.doc" -o -name "*.docx" -o -name "*.xls" -name "*.xlsx" \) |
while read f;
do
    if [[ $f == *.odt ]] || [[ $f == *.doc ]] || [[ $f == *.docx ]]
    then
        cp -f "$f" /home/ben/backup/wordprocessor
    fi
    if [[ $f == *.ods ]] || [[ $f == *.xls ]] || [[ $f == *.xlsx ]]
    then
        cp -f "$f" /home/ben/backup/spreadsheet
    fi
done
```

The Bash **if** command allows you to execute a code block only if a particular condition is true. It's both hugely powerful and quite complex. Used like this (with **[[string1 == string2]]**) it matches filenames, and you can use asterisks in the same way you can at the command line, so ***.doc** matches any file that ends with **.doc**. The **||** is used to group multiple conditions together so that the code block is run if any one of them is true.

Running automatically

Writing scripts like this can really simplify general tasks like backing up data, but wouldn't it be great if you could automate running them as well?

Almost all versions of Linux (and, for that matter, Unix in general) come with a tool called **crontab**. The name doesn't give much away, but it's for scheduling tasks to run at certain times (it's named after Chronos, the Greek god of time). There are only really two options that you need to know: **-e** and **-l**. The first is used to edit scheduled commands, and the second is used to list them.

So, to set up our script to back up commands, run:

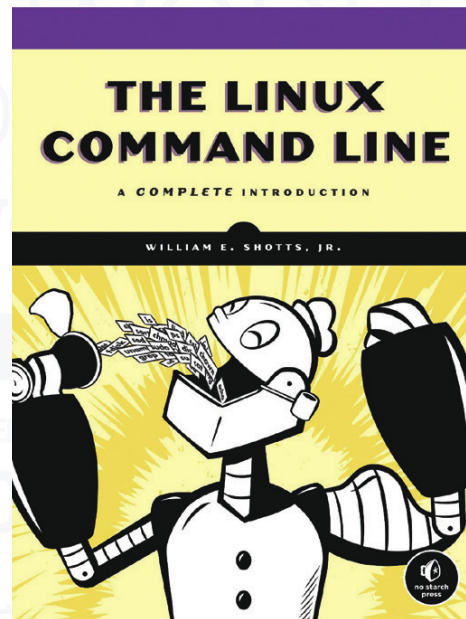
```
crontab -e
```

This will start a text editor (usually either Vim or Nano). If blank lines at the bottom of the file are displayed as **~**, then it's probably opened in Vim. This is a powerful editor, but it can be quite confusing if you haven't used it before. If you want to switch to something easier, exit Vim by pressing Escape, **:**, **Q** and **!**. They you can tell the system to use Nano instead by running:

```
export EDITOR=nano
crontab -e
```

Books

Bash scripting is incredibly, but can also get quite technical. If you're interested in taking it further, there are loads of good books on the subject. *The Linux Command Line* and *The Advanced Bash Scripting* guide are both excellent choices. The latter is a bit more technical than the former. What's more, the both have e-book versions that are free as in zero cost and free as in speech. You can get them from www.linuxcommand.org/tlcl.php and www.tldp.org/LDP/abs/html/index.html respectively.



If you prefer your books in paper form, see *The Linux Command Line* website for purchasing options.

You should now see 'GNU nano' in the top-left corner. Depending on your distribution, you may find that you already have some scheduled tasks, you may have a blank file, or you may have some lines that start with a **#** (these are comments).

To schedule tasks, you need to add a line to this file that tells it what to run and when. Schedules are broken up into five parts, each of which can be a number or an asterisk. For example:

```
0 2 *** /home/ben/backup.sh
```

The five scheduling segments represent the minute of the hour, the hour of the day, the day of the month, the month of the year, and the day of the week. An asterisk means 'every'. The above line will run the backup script at 2.00am every day. If you were more cautious, you could run it every hour with:

```
0 *** /home/ben/backup.sh
```

Or you could run it twice a day with:

```
0 0,12 *** /home/ben/backup.sh
```

“Sometimes it's not enough just to send the output of one command into another.”