Ubuntu Application Development

Ubuntu Developer Doc Team

Ubuntu Application Development An electronic copy of this book can be downloaded for free. We permit and even encourage you to distribute a copy of this book to colleagues, friends, family, and anyone else who might be interested.

http://developer.ubuntu.com

Revision number: 103 Revision date: 2011-02-04 22:59:25 +0000

Contents

Prologue

Welcome

Welcome to the *Ubuntu Application Development Manual*, an introductory guide to help you write software on and for Ubuntu.

If you've written web pages or software for other platforms, and would like to write software for Ubuntu, this manual is for you.

The goal of this book is to cover the basic tools available to you as an Ubuntu software developer and to guide you through the steps required to create an application, test it, and deliver it to your users. This book is for developers who want to "learn by doing". It is designed to be simple for developers who are just getting started with python, but complete enough for experienced developers migrating from writing applications on other platforms.

The Ubuntu Application Development Manual is not intended to be comprehensive. It is more like a quick-start guide that will get you doing the things you need to do to write applications for Ubuntu quickly and easily, without getting bogged down in technical details.

Please bear in mind that this manual is still very much a work in progress and always will be. It is written specifically for Ubuntu 11.04, and although we have aimed to not limit our instructions to this version, it is unavoidable that some things will change with each new release of Ubuntu. As updates are made, the latest version will be available at https://developer.ubuntu.com.

Ubuntu philosophy

The term "Ubuntu" is a traditional African concept that originated from the Bantu languages of southern Africa. It can be described as a way of connecting with others—living in a global community where your actions affect all of humanity. Ubuntu is more than just an operating system: it is a community of people that come together voluntarily to collaborate on an international software project that aims to deliver the best possible user experience.

The Ubuntu promise

- Ubuntu will always be free of charge, along with its regular enterprise releases and security updates.
- Ubuntu comes with full commercial support from Canonical and hundreds of companies from across the world.
- Ubuntu provides the best translations and accessibility features that the free software community has to offer.

- Ubuntu core applications are all free and open source. We want you to use free and open source software, improve it, and pass it on.

Contact details

Many people have contributed their time freely to this project. If you notice any errors or think we have left something out, feel free to contact us. We do everything we can to make sure that this manual is up to date, informative, and professional. Our contact details are as follows:

The Ubuntu Developer's Manual Team

Website: http://developer.ubuntu.com

Email: ubuntu-developer-manual@lists.launchpad.net

IRC: #quicky on irc.freenode.net

Conventions used in this book

The following typographic conventions are used in this book:

- Application names, button names, menu items, and other GUI elements are set in **boldfaced type**.
- Menu sequences are sometimes set as **System ▸ Preferences ▸ Appearance**, which means, "Choose the **System** menu, then choose the **Preferences** submenu, and then select the **Appearance** menu item."
- `Monospaced type` is used for code samples and terminal commands.

Introduction to Ubuntu Application Development

Introduction

In this chapter we will review the tools that you will learn to use to develop applications. Development for Linux desktops such as Ubuntu is very easy. While there are many IDEs to choose from, they are not required. It is more common to select a favorite tool for each job, and let them work together in a federated way to create your applications. The tools you will use in this manual are:

- The Terminal command line
- The Python Language
- PyGtk Widget Toolkit
- Glade GUI editor
- Quickly, a program which helps tie them altogether

With those tools we will learn to create a project using Quickly, and to familiarize ourselves with that project.

The Terminal

Terminal is the program that gives you access to the command line. If you've never used a command line while programming, but are used to choosing menu items, clicking buttons, and filling in dialogs to issue commands and configure your program, using a command line may seem a bit unfamiliar at first. However, you will soon learn that the command line is easier for most programming tasks., Don't worry, this manual will be very specific about what commands to type and how.

Gedit Editor

Gedit is the text editor that comes with Ubuntu. Gedit is called simply "Text Editor" on the Ubuntu menu.

It is a very capable text editor. It features Python syntax highlighting right out of the box. Gedit is also highly extensible and customizable. There are many useful and cool plug-ins that can make you more productive. Gedit also allows you to customize it's behavior using Python, so you can make it work however you want.

Figure 1: Terminal in action

Python Language

Python is the language of choice for application programming on Ubuntu. It's a highly fun and productive language, with a great library, and integration into to GNOME windowing system. I won't sing the praises of Python because I could never do better than Eric Raymond:

Figure 2: screenshot of Gedit in action

To say I was astonished would have been positively wallowing in understatement. It's remarkable enough when implementations of simple techniques work exactly as expected the first time; but my first metaclass hack in a new language, six days from a cold standing start? Even if we stipulate that I am a fairly talented hacker, this is an amazing testament to Python's clarity and elegance of design. Eric Raymond, Why Python? (http://www.linuxjournal.com/article/3882)

You don't need to do anything to get Python installed on your computer. It gets installed for you when you install Ubuntu. For now, to test Python, simply type **`python`** into the command. The next chapter will provide an overview of the Python language. You will get a prompt where you can start to write Python code.

PyGtk Widget Toolkit

Gtk originally stood for the Gimp Tool Kit, because it was created to enable programming the Gimp image editor. However, it soon became the standard toolkit for the Gnome Desktop, upon which Ubuntu is based. The toolkit includes everything you need, such as buttons, windows, dialogs, menus, widgets of all sorts, to create a richly interactive application. When you use Gtk with Python, it's call PyGtk.

Glade GUI Editor

Glade is a tool that allows you to lay out a Window and its widgets visually.

Figure 3: screenshot of Glade in action

Quickly

Quickly is a tool that makes programming easy and fun by bringing opinionated choices about how to write different kinds of programs to developers. Included is a Ubuntu application template for making applications that integrate smoothly into the Ubuntu software infrastructure.

You use Quickly by issuing a set of simple commands on the Terminal command line. Let's get started by creating an application with Quickly and see what it does for us before we even start programming.

Get Quickly

It's free and easy to get everything you need to start programming installed, you just need to install Quickly and everything will be set up for you. To install Quickly, open a Terminal window, and type:

```
$ sudo apt-get install quickly
```

(Don't type in the `$` . That's just a convention to show that you are typing on a command line.) You'll need to supply your user password, of course. Quickly will be downloaded and installed. There's a lot to it, so it may take a while to get everything downloaded and installed.

Start A Project

You start a new Quickly project using Quickly's "create command". In this manual, you will be starting and enhancing a Feed Reader for identi.ca, a free service similar to Twitter. The application looks roughly like this:

For the demonstration purposes, we can name this application "Feed Reader". I know, not to original, but it does the job. Get started by typing:

```
$ quickly create ubuntu-application feed-reader
```

Let's look briefly at each part of this command. quickly = tells the terminal to use the program "quickly". Note that capitalization is important. "Quickly" with a capital "Q" is the name of the project, but "quickly" with a lower case "q" is the name of the program. Terminal won't know what "Quickly" with a capital "Q" is.

create = the Quickly command to use. Quickly has a set of commands you can use. Besides create, there is edit, design, share, release, license, add, and even a few others. This manual will explain commands as they are used.

ubuntu-application = the name of the template to pass to the create command. There are other templates, such as a template for creating a command line application, and one for creating a game. But this manual only covers using the ubuntu-application template.

feed-reader = the name of the project you are creating. The way naming

Figure 4: screenshot of FeedReader

works, is you always name your project with lower case words, and if you want to have more than one word for the name, separate the words with a dash, not a space. Quickly will take care of converting that input correctly in titles, library names, modules, etc…

First Look

After running the create command, Quickly will output some information to the Terminal, and then will finish by actually running your application. Note that the window is named correctly. Also, there are menus and an About Box and even a preference dialog, that all work.

After you quit the application, you need to move the terminal into the directory for your project. Quickly created a directory called ”feed-reader” for this purpose. Use the cd command to ”change directory” into this newly created directory:

```
$ cd feed-reader
```

Run the Project

Once you are in the project directory, you can run your application again by issuing the run command:

```
$ quickly run
```

Use `$ quickly` run after you make changes to your GUI or to your code to test them out.

Figure 5: screenshot of first run application, with About Box showing

Explore the Code

To take a look at the code that Quickly created for you, use the ”edit” command.

```
$ quickly edit
```

This will open all of your code in Gedit. Lots of code was created for you, this is called the ”boiler plate” code. You can edit any of the files to make your application work the way you want to. In practice you will mostly be editing the main bin file, and other files that you may add, such as dialogs for example. You an switch to the feed-reader file, and take a look at the generated class ”FeedReaderWindow”.

In gedit, the bin file is always named the same as the project, and is stored in the bin directory. This is because it is a special file that Ubuntu runs when you ask Ubuntu to run the application. All the other files are considered part of your projects library or data, so those files are in different directories.

Look at the GUI

If you want to start modifying the GUI, you can open Glade. You use the ”design” command to do this. Don’t try to open Glade and then open your project from in there, it won’t work. So, just use this:

```
$ quickly design
```

And Glade will open with your project loaded.

Figure 6: screenshot of glade

Creating an application with Quickly

Introduction to Quickly

Quickly helps you create software or command line applications and python games quickly, it can also be used to create other things. You can select from a set of templates and use some simple Quickly commands to create, edit code and GUI, and publish your software for others to use.

Quickly's templates are easy to write. So if you are a fan of language foo, you can create a foo-project template. Or if you want to help people making plugins for your killer app, you can make a killer-app-plugin template. You can even create a template for managing corporate documents, creating your awesome LaTeX helpers, the sky is the limit!

Quickly is a command line based application which makes programming easy and fun by bringing opinionated choices about how to write different kinds of programs to developers. There are several commands that you will need to know about which we will explain in this chapter.

You will need to open **Applications ▸ Accessories ▸ Terminal**

Quickly create

Quickly create

after installing Quickly, you can type in the Terminal window $ quickly getstarted to get some hints on how-to-start.

Glade

Glade

This chapter will describe how to layout and use widgets to make a functional user interface for Ubuntu. We'll do this by laying out a user interface for a feed reader application using Glade. The application will look like this:

Figure 7: picture of a feedreader application

Boxes

The Gtk library positions widgets in relation to each other and their parent containers. This works much like a grid bag layout manager in Java's awt library, HTML, or XAML Windows Presentation Foundation. The essential container elements are the VBox and the HBox. The VBox orients child widgets vertically, and the HBox orients child elements horizontally. By combining HBoxes and VBoxes, you can create virtually any layout desire.

To follow along, create a Quickly application:

```
$ quickly create ubuntu-application feed-reader
```

To run Glade for your Quickly application, you must use the command `$ quickly design`. If you try to run Glade directly Glade will not work because it requires certain configuration options to work correctly with Quickly.

So, cd into your project directory and type:

```
$ quickly design
```

Your Quickly application will open. You can use the tabs to select between the different windows and dialogs in your application. If it's not already selected, choose "FeedReaderWindow" from the project menu.

Figure 8: picture of FeedReaderWindow open in Glade

Notice that there are some default widgets in the Window. You should start by deleting those. Click on the big Ubuntu logo. When it is selected, press the delete key on your keyboard to delete it. Do the same for labels on the form.

Now your Window has a VBox ready for filling with widgets. In a Quickly application, "vbox1" is the name of the primary container. The window we are designing is laid out in a more or less top to bottom format, in four rows of controls, so we need four empty rows in vbox1. The first row is already taken by the menu bar, and the bottom row by the status bar, so we need to add two rows. You can do this by selecting vbox1 in the inspector. Then in the editor, select the General tab and change "Number of Items" fro 4 to 6. A new row will appear.

Adding Widgets

We'll add the widget from top to bottom. Start by adding the tool bar to the top. In the Toolbox window, click on the Tool Bar icon. Then, click in the first empty spot in VBox. Since the tool bar is empty, the spot will immediately

Figure 9: picture showing Glade set up with vbox1 set to have 5 sections

collapse down to have no height. Click on the pencil in Glade's tool bar to bring up the Tool Bar Editor.

Figure 10: picture showing toolbar editor with collapsed tool bar behind

Switch to the Hierarchy tab of the Tool Bar Editor. Then click the Add button to a new widget to the tool bar. To make a refresh button, set the Stock Id to refresh

"Stock" widgets in Gtk are incredibly useful. When you set a widget to a

stock widget, Gtk will apply the whatever theme desktop theme the user has chosen to the widget, including the right icon if required. It will also supply the right word, so that the widget will show up translated into whatever language the user is using on their desktop with no translating.

So to create the Refresh button, choose gtk-refresh as the stock id, and you'll get the right icon and translations, and everything for free.

Figure 11: picture of Tool Bar Editor with the stock icon set

To make it a bit easier to program later, rename the button from `toolbutton1` to `refresh_button`. You could easily add more widgets to the tool bar, but for now, we'll only need the refresh button.

Next step is to add label at the top to display you last dent. Select Label from Controls and Display and add that to the next open area. If you want, you can quickly change the label. Click on the label to select it, then select the General tab in the editor. Then set whatever you want in the "Label" field.

Next, we'll add the text entry and button fields at the bottom. These are two widgets, laid out horizontally, so we'll use an HBox with two items. Click on HBox in the containers section of the palette, and then click in the empty space at the bottom of the Vbox you just created.

To add the entry field, click on TextEntry under the Control and Display section of the toolbox, the click in the left hand pane of the HBox you just added.

To add the button, click on "Button" (also in the Control and Display section of the palette), and put a button in the right hand box.

Figure 12: insert screen shot of changing label text

Notice that the widgets don't look like what we're designing. For instance, the button is huge and just says "button," and the spacing above the entry looks odd. We'll fix the sizing problems next, but first, it's easy to change the label on the button. Simply select the button, then on the general tab, change the Label property to "Dent".

Figure 13: picture of Glade with the HBox added

Figure 14: picture of Glade with the entry and button added

The middle section of window will be taken up by all the dents, and we'll use a VBox for as a container. However, we want the VBox to be scrollable, so we'll add a scrollable region. Under Containers in the toolbox, click on Scrolled Window, and then add it to the VBox.

Next, we want to add a VBox to the scrolledwindow1. However, it won't work if we try to add it directly, because VBoxes and HBoxes don't sup-

Figure 15: picture of Glade with button text set

port scrolling. So, add a View Port (also under Containers in the toolbox) to scrolledwindow1, and then add a VBox to the Viewport. You don't much have to worry about the number of items in the VBox, it will grow as you add items from code.

Since we will be accessing the ScrolledWindow, The Label.The TextEndtry, and the Button from code, it is a good idea to give them names that will make them easier to remember. For example, after adding the scrolled window, in the editor, under General, change the name to something like "dent_vbox". Do the same for the other widgets, such as "dent_entry", "dent_button", "dent_label". <picture of Glade showing renamed vbox>

Packing

Make sure to save your file in Glade so the changes that you made take effect before running. Then switch back to your terminal and use:

```
$ quickly run
```

to test the application.

As you maximize or resize the application, you'll notice: 1. The button at the bottom is big, and stretches to fill the space no matter how big or small we make the window. 2. The labels for the previous dent does the same thing. 3. The space between the controls and the outside of the window aren't right. 4. The button just says "button."

We'll fix the first three problems using the "Packing" tab of the editor. We'll get to #4 a little later.

Expand and Fill

"Fill" is a property that tells a widget that all of it's children should fill up it's space. Expand tells a widget to expand to fill all the space allotted to it. You can use these two properties to control the size of widgets in relation to each other.

To get started fixing the buttons, choose the HBox in the inspector that contains the buttons. Choose the Packing tab of the editor. Notice that Expand and Fill are both set to "Yes". Click the Fill button to set it to "No". Notice that the buttons no longer fill up the whole Height of the Hbox. That's because you just told it not to tell it's children to fill up the space alloted to it.

Now click the "Expand" button to set that one to "No". Notice that the Hbox is not only as high as it's child widgets. That's because you just told it not to expand to fill the space alloted to it.

Now we need to apply the same logic to the widgets in that HBox.. Click on the button to select it, and then set expand and fill to "No".

The entry for the dents to be posted, should expand to fill all the space alloted to it. Therefore, there is no need to change the packing for dent_entry.

Change the Expand and Fill for the dent_label to ”No” as well. If you run the application again, you’ll see that except for the missing dents, which won’t be added until the next chapter, the window is working properly.

The spacing of the controls could be nicer. We’ll use the padding and border properties to fix this. Padding tells a widget how much space to put between itself and it’s sibling widgets (and also the side of it’s parent), while border tells a widget how much space to surround itself with.

These two properties are additive, so that if you set a border property, it tells the widget to include that space in addition to any space that is already taken up by padding.

To create the right spacing between all the other widgets, select each and set their padding to 5. Padding is in the Packing tab of the editor.

Menus

Quickly provide a set of default menus for FeedReader. If you want to adjust those menus, you can do that by selecting the menu in the inspector, and then clicking the ”Edit” button on Glade’s toolbar. This will bring up the editor dialog window. You can switch to the Hierarchy tab to add, remove, and move menu items.

Responding to Signals

Your code will need to respond to three of the widgets that you’ve added, but buttons, and also the user should be able to just use the enter key to send their dent.

We’ll write just enough code to prove to ourselves that we are able to respond to signals from these widgets. To start writing code, we need to open it in an editor. You can use the edit command for this”

```
$ quickly edit &
```

This will open all of your code in Gedit. The file called ”feed-reader” is your main file, so choose that for editing. We’ll use the auto-signals feature to wire up the signals. Add the following methods to the FeedReaderWindow class:

```
1         def refresh_button_clicked_event(self, widget, data=None):
2             print "REFRESH"
3
4         def dent_button_clicked_event(self, widget, data=None):
5             print "DENT"
6
7         def dent_entry_activate_event(self, widget, data=None):
8             print "DENT"
```

Now you can run the project using the run command, and when you click the buttons, notice that the terminal prints the correct message.

```
$ quickly run
```

How did this work? Your Quickly project used the auto-signals feature to connect the button to the event. To use auto-sginals, simple follow this pattern when you create a signal handler:

```
9 def widgetname_eventname_event(self, widget, data=None):
```

Sometimes a signal handler will require a different signature, but (self, widget, data=None) is the most common. We'll start to make the feed reader really work in the next chapter.

Creating Widgets on the Fly

Introduction

Sometimes using Glade to layout controls is not always suitable. Typically, this is because you don't know the exact contents of the Window when you are writing the program. This chapter describes how to create Windows and populate them with Widgets in code so that you can create dynamic user interfaces at run time.

General approach

The general strategy for creating a widget at run time is to:

1. Instantiate the widget and assign it to a variable.
2. Use set and get functions to configure the widget as desired.
3. Show the widget.
4. Pack the widget into a container, such as a VBox or HBox.

Create a window on the fly

You may often want to create a window on the fly. A Window is just a kind of widget, so creating the window is not much different than creating the widgets that will populate it.

We'll start with an empty program that has a class that doesn't do anything but import the necessary modules, and then print something.

```
#!/usr/bin/env python
try:
  import pygtk
  pygtk.require("2.0")
  import gtk
except:
  print "couldn't␣load␣dependencies"

class RuntimeWidgets:
  def __init__(self):
    print "let's␣create␣a␣window"
  if __name__ == "__main__":
    p = RuntimeWidgets()
    gtk.main()
```

We'll do all the work in the __init__ function, but you could imagine that this could be any kind of program or function.

To start, we'll create a window on the fly, and show it. Since we only imported the gtk module, and not any of the specific classes within it, we'll have to access them with the "gtk." notation. So to instantiate the window and assign it to a variable, use:

```
test.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
```

`gtk.WINDOW_TOPLEVEL` is a constant that tells the new window to assume that it has no parent and that it should be tracked in the GNOME application bar. Application windows and dialogs should all be `gtk.WINDOW_TOPLEVEL`. The other kind of window is `gtk.WINDOW_POPUP`. These windows have no frames, have parents, and are not tracked by the application bar. These sorts of windows are for tooltips and things of that nature.

After creating the window, we want it to be visible. To do that, simply call the show() function.

```
test.window.show()
```

All widgets are invisible when they are initially created, so you must always call show() when you want users to be able to see them. You can add containers without showing them, but they won't be visible until you call show().

The whole program now looks like this:

```
#!/usr/bin/env python
try:
  import pygtk
  pygtk.require("2.0")
  import gtk
except:
  print "couldn't␣load␣dependencies"

class RuntimeWidgets:
  def __init__(self):
    # Create a window on the fly
    test_window = gtk.Window(gtk.WINDOW_TOPLEVEL)
    test_window.show()

if __name__ == "__main__":
  p = RuntimeWidgets()
  gtk.main()
```

If you run it, you'll see that you have created a window.

Let's connect the window's destroy signal to `gtk.main_quit()` so that the program exits when we quit the window. As described in chapter 4, you only do this for the "main" application window. If you are creating a dialog or other window, you probably don't want the application to quit when you

Figure 16: python-created window, based on the code above

close the window, so in those cases, don't connect `gtk,main_quit()` to the destroy event.

```
1 test_window.connect("destroy", gtk.main_quit)
```

Let's also configure the window a little bit. we'll start by setting the title of the window. When configuring widgets, you always access the different properties and settings through methods that start with "get_*something*" or "set_*something*". So if we want to set the title, we'll use `set_title`:

```
1 test_window.set_title("run␣time␣widget␣creation␣demo")
```

We can also tell the window how large we would like it to be by setting the default size. The first argument is the width, and the second is the height:

```
1 test_window.set_default_size(200,75)
```

So now the __init__ function creates a window, connects to an event on the window, and configures the title and default size:

```
1 test_window = gtk.Window(gtk.WINDOW_TOPLEVEL)
2 test_window.connect("destroy", gtk.main_quit)
3 test_window.set_title("run␣time␣widget␣creation␣demo")
4 test_window.set_default_size(200,75)
5 test_window.show()
```

If you run the program, you can see that the window is configured as desired:

Figure 17: configured window as per above code

So now that we have the window set up, we'll populate it with widgets. We'll create a window that doesn't do much, but will be useful for demonstrating this code. The window will have a progress bar and two buttons. One button will cause the progress to increment, and the other will clear the progress.

Figure 18: progress bar clear

The cancel button will return the progress bar to this state.

Figure 19: progress bar pulsed

Every time you click the pulse button, it will cause the progress bar to pulse.

Since a Window is a widget that can only contain one child, we'll start by adding a VBox to the Window. We'll call it the "mainbox". We don't want the children of the mainbox to expand and we want it to have two slots, so we'll use "False" for the first argument, and "2" for the second when we instantiate it.

```
1 mainbox = gtk.VBox(False, 2)
```

Of course we want mainbox and it's children to be visible, so we'll call its `show()` function. If we didn't do this, we could add it to the window, but neither mainbox, or any of it's children would ever be visible until `show()` was called.

After showing it, we'll go ahead and add it to the window:

```
1 mainbox = gtk.VBox(False, 2)
2 mainbox.show()
3 test_window.add(mainbox)
```

In the top slot of mainbox, we'll want an HBox with two slots along the top, one for the progress bar and one for the cancel button. We'll use the same idea to create that HBox, and we'll call it "topbox".

```
1 topbox = gtk.HBox(False, 2)
2 topbox.show()
```

However, we dont want to just "add" it to mainbox, as we want to control where it goes and how it displays. To do this, we'll "pack" it into the mainbox. Since we want topbox to be along the top, we'll use the "pack_start" function. This will put it in from the start of the mainbox. Since the mainbox is a VBox, the start is the top. While we're at it, we'll tell mainbox that we don't want topbox to expand vertically. We do this by passing in False for the second argument:

```
1 mainbox.pack_start(topbox, False)
```

Let's go ahead and create the progress bar and then add it to the topbox:

```
1 progressbar = gtk.ProgressBar()
2 progressbar.show()
3 topbox.pack_start(progressbar, True)
```

If we add the code to the __init__() function and then run it, we can see that the progress bar displays:

Figure 20: window with progress bar

We'll add the cancel button next. When we create the button, we'll tell the button that we want it to be a stock cancel button. As discussed in Chapter 5, this will ensure that the button is themed with the correct icon, and the text is set in the language currently set for the desktop. We tell a button to be a stock button by using an optional parameter "stock" and assigning it to the desired constant:

```
1 cancel_button = gtk.Button(stock=gtk.STOCK_CANCEL)
```

Now that we've got a stock cancel button, we can add it to topbox. We don't want the button to expand horizontally, so when we pack topbox, we'll use False as the second argument. We'll also connect the clicked signal to a member function called "cancel".

```
cancel_button.show()
# passing in false tells it not to expand horizontally
topbox.pack_end(cancel_button, False)
```

Now we need to tell the cancel button to clear the progress bar. The way to clear the progress bar is to call it's `set_fraction()` function, and pass in zero, telling it that it has no progress. So you want to connect the button's clicked signal with `progress_bar.set_fraction(0)` in some way. But how?

The typical way to do this is to create a function and pass in the function to the button's connect function, like this:

```
cancel_button.connect("clicked", self.cancel)
```

However, in order to call the progress bar's `set_fraction` function, we would have to keep a reference to that function somewhere. Fortunately, there is an easier way to do this, since we only want to call a single line of code. We'll use Python's lambda function facility. This allows us to pass a function straight into a function that expects a function name.

Lambda functions in Python are quite similar to anonymous methods in .NET and JavaScript.

The syntax for doing this looks like this:

```
cancel_button.connect("clicked", lambda x: progressbar.set_fraction(0))
```

This is quite useful, as we don't have to store and track a reference to a dynamically created widget. You could imagine how useful this would be if you were dynamically creating and destroying numerous widgets over the course of a programs lifetime.

Now we want to add the pulse button. There is no stock pulse button, so we'll create a normal button. We'll pass in a string when we create it, and the button will use that string for its caption:

```
pulse_button = gtk.Button("_Pulse")
pulse_button.show()
```

The underscore before the P tells the button to add a keyboard shortcut for the button, Alt + P will click the button.

We'll pack the button into mainbox, and tell mainbox not to expand the button vertically. Finally, we'll use another lambda function to connect the pulse button's clicked signal to the progress bar's pulse functionL:

```
mainbox.pack_start(pulse_button, False)
pulse_button.connect("clicked", lambda x: progressbar.pulse())
```

The end result

So now your finished application's code should look like this:

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

try:
  import pygtk
  pygtk.require("2.0")
  import gtk
except:
  print "couldn't␣load␣dependencies"

class RuntimeWidgets:
  def __init__(self):
    # create a window on the fly
    test_window = gtk.Window(gtk.WINDOW_TOPLEVEL)
    test_window.connect("destroy", gtk.main_quit)
    test_window.set_title("run␣time␣widget␣creation␣demo")
    test_window.set_default_size(200,75)
    test_window.show()

    # create a VBox to hold the main contents
    # passing in False tells the VBox not to expand controls added to it vertically
    mainbox = gtk.VBox(False, 2)
    mainbox.show()
    test_window.add(mainbox)

    # create a VBox to hold the progress bar and cancel button
    topbox = gtk.HBox(False, 2)
    topbox.show()
    # passing in False tells it not to expand vertically
    mainbox.pack_start(topbox, False)

    # create a progress bar and pack it into the VBox
    # use self to make it part of the parent object so it can be used in other functions
    progressbar = gtk.ProgressBar()
    progressbar.show()
    # passing in True tells it to expand horizontally
    topbox.pack_start(progressbar, True)

    # create a stock cancel button and pack it into the vbox
    cancel_button = gtk.Button(stock=gtk.STOCK_CANCEL)
    cancel_button.show()
    # passing in False tells it not to expand horizontally
```

```
42      topbox.pack_end(cancel_button, False)
43
44      # connect the cancel button to the cancel function
45      cancel_button.connect("connect", lambda x: progressbar.set_fraction(0))
46
47      # create a custom button
48      pulse_button = gtk.Button("_Pulse")
49      pulse_button.show()
50      # passing in False tells it not to expand vertically
51      mainbox.pack_start(pulse_button, False)
52      # connect it to the pulse function
53      pulse_button.connect("clicked", lambda x: progressbar.pulse())
54
55  if __name__ == "__main__":
56    p = RuntimeWidgets()
57    gtk.main()
```

When we click on the pulse button, it makes the progress bar move:

Figure 21: progress bar moving, finished application

And when we click the cancel button, it clears the progress bar.

So we created a simple window and added interactive widgets at runtime. We only set a few select properties of a few controls. Refer to the pyGtk reference at “link goes here” for details on all the widgets and properties you can set on them.

Figure 22: clear progress bar, finished application

In the next chapter we'll extend these concepts and learn to roll custom widgets.

Multimedia

Introduction

This chapter will introduce the basics of adding multimedia to your applications. We'll start by creating a simple image viewer application, and then change that application to support overlays. We'll finish by adding the ability to play movies to the application.

To prepare, create an application using `$ quickly create`, and delete the boiler plate widgets as per usual.

Figure 23: screenshot of application, ready to start

Displaying an Image

For the first iteration, we will display and image from the user's Pictures directory, assuming the user has added a Picture there called `portrait.jpg`. The simplest way to present an image is to use a `gtk.Image` widget. A `gtk.Image` widget is a container for a `gtk.gdk.pixbuf`. The `gtk.gdk.pixbuf` contains the actual image data. Add a `gtk.Image` to an application using Glade. Since the `gtk.Image` has no data to start, the "broken image" icon will display by default.

Using `glib.get_user_special_dir()` to find a path to Pictures directory ensures that the application works well even after it's been translated or if the Pictures directory is moved for some reason.

First, import the glib library:

Figure 24: Image with broken image

```
1 import glib
```

Then add the following code to the `finish_initializing` function to create a path To access standard directories, use `glib.get_user_special_dir()` like this:

```
1         picture_dir = glib.get_user_special_dir(glib.USER_DIRECTORY_PICTURES)
2         path = os.path.join(picture_dir, "portrait.png").
```

Then set the `gtk.Image` to display that picture:

```
1         self.ui.image1.set_from_file(path)
```

But as you can see, the image isn't displayed very well. So we should change the size of the picture before displaying it. But since the image is just a widget for image data in a `gtk.gdk.PixBuf`, we don't change the size of the `gtk.Image`, we have to load the picture as a `gtk.gdk.PixBuf`, and then resize it, and then add it into the image. So the following code will load the picture data, scale it, and set the `gtk.Image` to use that data:

```
1         #load the pixbuf
2         picture_dir = glib.get_user_special_dir(glib.USER_DIRECTORY_PICTURES)
3         path = os.path.join(picture_dir, "portrait.jpg")
4         pixbuf = gtk.gdk.pixbuf_new_from_file(path)
5
6         #calculate new sizes for the pixbuf
7         new_width = 600
8         new_height = 600
9         if pixbuf.get_height() > pixbuf.get_width():
10            new_width = int((float(new_height)/pixbuf.get_height()) * pixbuf.get_width())
```

Figure 25: picture of running application

```
11          elif pixbuf.get_width() > pixbuf.get_height():
12              new_height = int((float(new_width)/pixbuf.get_width()) * pixbuf.get_height())
13
14          #scale the pixbuf and use the scaled image
15          scaled_buf = pixbuf.scale_simple(new_width,new_height,gtk.gdk.INTERP_BILINEAR)
16          self.ui.image1.set_from_pixbuf(scaled_buf)
```

Now when you run the app, you can see the image is scaled properly.

Figure 26: Screenshot of running application

GooCanvas

Rather than just displaying the picture, let's say we want to display a little Ubuntu badge on the image, and a title. For composing with images and graphics, we use a `goocanvas.Canvas` object with some related classes. GooCanvas is a widget that allows mashing up all kinds of graphics, like pictures, shapes, text, paths, etc. For just displaying a picture, it's a bit more complex than using a `gtk.Image`, but it's a lot easier for mashing up graphics.

Add a `goocanvas.Canvas`

Delete the `gtk.Image` from Glade, and then back in the code, start out by importing `goocanvas`:

```
import goocanvas
```

Also, go ahead and delete the pixbuf scaling code, and the code that adds the pixbuf to the `gtk.Image`. So delete this code:

```
        #scale the pixbuf and use the scaled image
        scaled_buf = pixbuf.scale_simple(new_width,new_height,gtk.gdk.INTERP_BILINEAR)
        self.ui.image1.set_from_pixbuf(scaled_buf)
```

The goocanvas library isn't integrated into Glade, so you need to create the widget and add it on the fly.

```
        #create and add the goocanvas
        canvas = goocanvas.Canvas()
        canvas.set_size_request(500,400)
        canvas.show()
        self.ui.vbox1.pack_start(canvas)
```

Now when you run the application, you can see the blank `goocanvas.Canvas`:

Add an Image to the `goocanvas.Canvas`

We already wrote the code to load the image from the file, as well as to calculate height and width to make it fit. So it's a matter of creating a `goocanvas.Image` object, and associating that object with the `goocanvas.Canvas`. Note that when you create the `goocanvas.Image`, you tell it that it is associated with the `goocanvas.Canvas`, rather than adding it to the canvas.

```
        #load the pixbuf
        picture_dir = glib.get_user_special_dir(glib.USER_DIRECTORY_PICTURES)
        path = os.path.join(picture_dir, "portrait.jpg")
        pixbuf = gtk.gdk.pixbuf_new_from_file(path)

        #calculate new sizes for the pixbuf
        new_width = 500
```

Figure 27: insert picture of app with blank goocanvas

```
8         new_height = 400
9         old_width = pixbuf.get_width()
10        old_height = pixbuf.get_height()
11        if old_height > old_width:
12            new_width = int((float(new_height)/old_height) * old_width)
13        elif old_width > old_height:
14            new_height = int((float(new_width)/old_width) * old_height)
15
16        #add the image
17        root_item = canvas.get_root_item()
18        image = goocanvas.Image(parent=root_item, pixbuf=pixbuf)
```

When you run this code, you can see that the image is added:

But of course, the size is not set. For a `goocanvas.Items` that you add to a `goocanvas`, you don't the height and the width, you scale the image. So a small calculation is required to turn the `new_width` and `new_height` variables into the proper scales, before calling the scale function:

```
1         w_scale = float(new_width)/old_width
2         h_scale = float(new_height)/old_height
3         image.scale(w_scale, h_scale)
```

Now, the image is scaled to the proper size.

Add a Badge to the Image

Next we'll place a Ubuntu badge on the image, using the file `background.png`. In a Quickly project, media files like images and sounds should always go into the data/media directory so that when users install your programs, the files

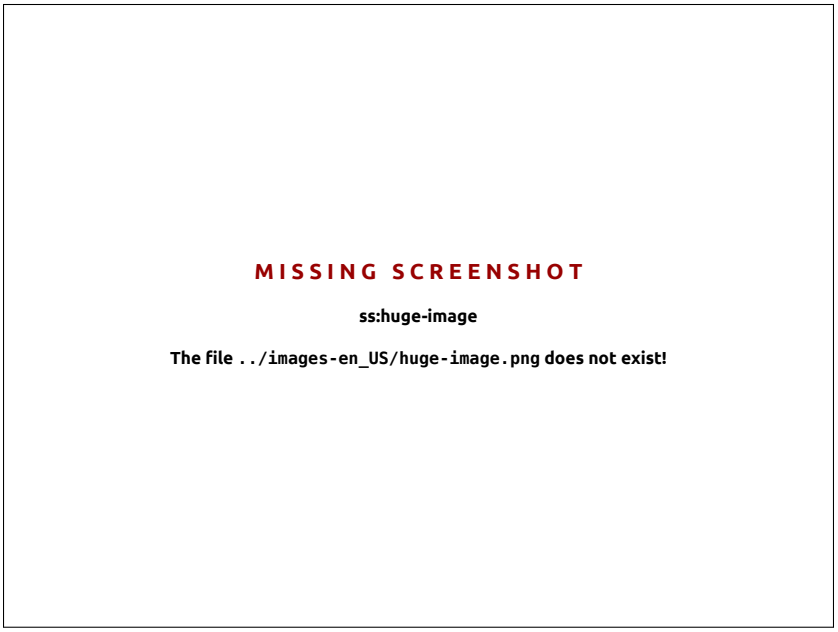

Figure 28: insert picture of huge image

Figure 29: insert picture of nicely scaled image

will go to the correct place. There is a helper function called `get_media_file` built into Quickly projects to get a URI for any media file in the media directory. You should always use this function to get a path to media files, as this function will work even when your program is installed and the files are put into different places on the user's computer. `get_media_file` returns a URI, but a pixbuf expects a normal path. It's easy to fix this stripping out the beginning of the URI. Since it was created for you, can could also change the way `get_media_player` works, or create a new function, but this works too:

```
1         logo_file = helpers.get_media_file("background.png")
```

```
        logo_file = logo_file.replace("file:///","")
        logo_pb = gtk.gdk.pixbuf_new_from_file(logo_file)
```

Then add the logo to `goocanvas.Canvas` in the same way you added the main image, except set the `x` and `y` for the position of the logo when creating it.

```
        logo = goocanvas.Image(parent=root_item, pixbuf=logo_pb,x=2200, y=1550)
        logo.scale(.2,.2)
```

Figure 30: picture of image with logo

You may notice that `x` and `y` for the logo are on a different scale than for the main image, the canvas is 500 by 400 pixels, but the logo is visible even though it is set at 2200×1550. This is because `goocanvas.Image` is a `goocanvas.Item`, and `goocanvas.Items` maintain their own coordinate systems even when you do things like scale and rotate them. There are two functions to help you convert between different scales. To convert a point on a canvas to a point in an item's scale use the canvas's `convert_to_item_space` function. So to find out what the point 100×100 on the canvas is to in a logo's space:

```
item_x, item_y = canvas.convert_to_item_space(logo, 100,100)
```

You can translate back using `canvas.convet_from_item_space`. So to find the position 50×50 for the logo on the canvas:

```
canvas_x, canvas_y = canvas.convert_from_item_space(logo, 50, 50).
```

Add Text to an Image

To add an title to the picture, use the `goocanvas.Text` class. It works in a similar manner to the `goocanvas.Image` class:

```
        title = goocanvas.Text(parent=root_item,text="Ilsabe", x=120, y=10)
        title.set_property("font","Ubuntu")
        title.scale(2,2)
```

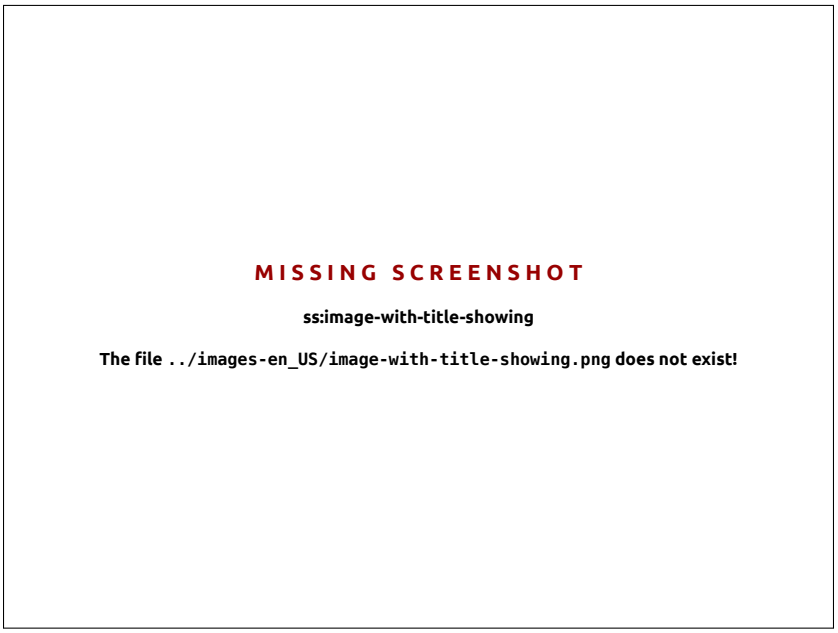

Figure 31: picture of image with title showing

The goocanvas library affords you the flexibility to be very creative and easily do lots of fun things in your applications. There are different kinds of Items and many of interesting visual things you can do with them. There are items like shapes and paths. You can change things like their scale, rotation, and opacity. You can even animate them!

The whole block of code for adding the image, logo and text looks like this:

```
        #create and add the goocanvas
        canvas = goocanvas.Canvas()
        canvas.set_size_request(500,400)
        canvas.show()
        self.ui.vbox1.pack_start(canvas)

        #load the pixbuf
        picture_dir = glib.get_user_special_dir(glib.USER_DIRECTORY_PICTURES)
        path = os.path.join(picture_dir, "portrait.jpg")
        pixbuf = gtk.gdk.pixbuf_new_from_file(path)

        #calculate new sizes for the pixbuf
        new_width = 500
        new_height = 400
        old_width = pixbuf.get_width()
        old_height = pixbuf.get_height()
        if old_height > old_width:
```

```
            new_width = int((float(new_height)/old_height) * old_width)
        elif old_width > old_height:
            new_height = int((float(new_width)/old_width) * old_height)

        #add the image
        root_item = canvas.get_root_item()
        image = goocanvas.Image(parent=root_item, pixbuf=pixbuf)

        #scale the image
        w_scale = float(new_width)/old_width
        h_scale = float(new_height)/old_height
        image.scale(w_scale, h_scale)

        #read in and add the logo file
        logo_file = helpers.get_media_file("background.png")
        logo_file = logo_file.replace("file:///","")
        logo_pb = gtk.gdk.pixbuf_new_from_file(logo_file)
        logo = goocanvas.Image(parent=root_item, pixbuf=logo_pb, y=1550,x=2200)
        logo.scale(.2,.2)

        #add a title
        title = goocanvas.Text(parent=root_item,text="Ilsabe", x=120, y=10)
        title.set_property("font","Ubuntu")
        title.scale(2,2)
```

Using the Web Cam

You can easily add Web Cam functionality to your application using `quickly.widgets.WebCamBox`.

Add a `WebCamBox`

Start by importing the `WebCamBox`:

```
from quickly.widgets.web_cam_box import WebCamBox
```

Then create and add a `WebCamBox` in the `finish_initializing` function:

```
        self.cam = WebCamBox()
        self.cam.show()
        self.cam.set_size_request(400,300)
        self.ui.vbox1.pack_start(self.cam)
```

If you run the application now, the web cam doesn't appear.

This is because the camera has not been started. It's important to give the application a chance to realize all its widgets before you try to run the `WebCamBox` because it might not be ready to start painting, and the camera

Figure 32: picture of blank window

won't start. You can add a button to start it playing, but you can also use `gobject.timeout_add` to tell it to start after a certain period of time, for instance, to start it playing after half a second (500 milliseconds), start by importing the `gobject` library:

```
1 import gobject
```

Then create a function to start the camera. It's important for the function to return `False`, otherwise, `timeout_add` will call it over and over again.

```
1     def play_web_cam(self):
2         self.cam.play()
3         return False
```

And then add this to your code after creating the `WebCamBox` in `finish_initializing` to tell it to start:

```
1         gobject.timeout_add(500, self.play_web_can)
```

This calls the `WebCamBox`'s `play()` function after 500 milliseconds. As you may have guessed, you can stop the camera using the `WebCamBox`'s `stop()` function.

Taking a Picture

The `WebCamBox` has a built in function called `take_picture`. This takes a picture from the camera, names it with the current time stamp, and drop's it in the `Pictures` directory.

So maybe a fun application would start up, take a picture, and then quit.

First, create a function to take the picture and then stop the camera:

Figure 33: picture of webcam working

```
    def take_picture_from_cam(self):
        self.cam.take_picture()
        self.cam.stop()
        self.destroy()
        return False
```

Then add a line to call that function:

```
        gobject.timeout_add(3000, self.take_picture_from_cam)
```

Here's all the code to make the web cam start, and take a picture, and then stop:

```
        #create a web cambox
        self.cam = WebCamBox()
        self.cam.show()
        self.cam.set_size_request(400,300)
        self.ui.vbox1.pack_start(self.cam)

        #run functions after a period of delay
        gobject.timeout_add(500, self.play_web_cam)
        gobject.timeout_add(3000, self.take_picture_from_cam)

    def play_web_cam(self):
        """play_web_cab - starts the web cam.

        """

```

```
        self.cam.play()
        return False

    def take_picture_from_cam(self):
        """take_picture_from_cam - taks a picture, and quits the program.

        """

        self.cam.take_picture()
        self.cam.stop()
        self.destroy()
        return False
```

The `take_picture` function returns a path to the picture it took, in case a program wants to do some processing with that picture. This tweak will simply print the path of the picture to the terminal:

```
        path_to_pic = self.cam.take_picture()
        print path_to_pic
```

Using GStreamer

A `MediaPlayerBox` is a wrapper around a GStreamer library's `camerabin` object. GStreamer gives you access to significant power and flexibility in editing and modifying the output of the webcam. A `MediaPlayerBox` as a `camerabin` property, so simply use `cam.camerabin` to get a reference to the gstreamer `camerabin`.

Playing Media

You can easily add the ability to play video or sound files to your application using `quickly.widgets.MediaPlayerBox`. A `MediaPlayerBox` works just was well for sound or video.

Add a `MediaPlayerBox`

Adding a `MediaPlayerBox` is a lot like adding a `WebCamBox`. Start by importing `MediaPlayerBox`:

```
from quickly.widgets.media_player_box import MediaPlayerBox
```

Then in the finish initializing function, create and pack it into the window:

```
        self.player = MediaPlayerBox(True)
        self.player.show()
        self.player.set_size_request(600,450)
        self.ui.vbox1.pack_start(self.player)
```

When creating a `MediaPlayerBox`, by default, it has no controls. However, you can pass in `True` when creating `MediaPlayerBox` to tell it to show controls by default. If you run the application, you can see the controls, but of course, they don't work because there is no file for the application to play.

Figure 34: screenshot of application with controls visible

Rather than a reference to file, `MediaPlayerBox` uses a URI to know what to play. So, in order to use he movie `minecraft.avi` in the `Video directory`, create a path, and then turn it into a well formed URI before setting the `MediaPlayerBox`'s property:

```
        #create the URI
        video_dir = glib.get_user_special_dir(glib.USER_DIRECTORY_VIDEOS)
        path = os.path.join(video_dir, "minecraft.avi")
        uri = "file://" + path

        #Set the URI for the player
        self.player.uri = uri
```

Now when you run the application you can use the play button to play and pause the video, and you can use the scroller to control the position.

Controlling the `MediaPlayerBox`

A `MediaPlayerBox` is designed to be easy to control from your program. So you can play, pause, and stop media playing using functions:

```
        self.player.play()
```

Figure 35: screenshot of application with mediaplayer showing

```
        self.player.pause()
        self.player.stop()
```

You can also get the current position in the media file in seconds:

```
current_second = self.player.position
```

You can change the position, too. To jump to the first minute of the video:

```
        self.player.position = 60
```

You can toggle whether the controls show:

```
        self.player.controls_visible = True
```

Knowing When a File is Done

It's often useful to know when a media file is done playing, for example to load another file, or to loop the file. To set up looping, or auto-replaying of a file, start by creating a `replay` function.

```
    def replay(self, widget, data=None):
        self.player.stop()
        self.player.play()
        self.player.position = 0
```

Then in `finish_initializing`, after you create the `MediaPlayerBox`, connect the `replay` function to the end of file signal:

```
        self.player.connect("end-of-file",self.replay)
```

Now the video will replay when it reaches the end. Here's all the code for looping the `minecraft.avi` video:

```
        #create a MediaPlayerBox
        self.player = MediaPlayerBox(True)
        self.player.show()
        self.player.set_size_request(600,450)
        self.ui.vbox1.pack_start(self.player)

        #create the URI
        video_dir = glib.get_user_special_dir(glib.USER_DIRECTORY_VIDEOS)
        path = os.path.join(video_dir, "minecraft.avi")
        uri = "file://" + path

        #Set the URI for the player
        self.player.uri = uri
        self.player.connect("end-of-file",self.replay)

    def replay(self, widget, data=None):
        self.player.stop()
        self.player.play()
        self.player.position = 0
```

Using GStreamer

Just like the `WebCamBox` wraps a `camerabin`, `MediaPlayerBox` wraps a Gstreamer `playbin`. If you want to access it, just use the `player.playbin` property.

Informing the user with Indicators

In this chapter we'll learn about Ayatana Indicators. We'll learn how to create an Indicator in order to be able to,

- Provide the user a persistent view over the state of the application.
- Provide the user an easily accessible group of functions to control the application.

About Indicators

Indicators are the tiny little icons sitting on the top-right of your screen. They provide the user quick access to common controls of an application and also represent the state of the application in the panel. An Indicator is made up of two things,

- A set of icons to represent various states.
- A GTK Menu to hold widgets for user interaction.

Status and Notification

State is represented with the help of icons. For example, the messaging indicator changes its icon to a green message box whenever the user has unread messages. This icon only goes back to normal when the user takes some action regarding the unread messages like reading or discarding them. This way the user will be notified of events even if he is away from the computer and misses the notification pop-up. Thus, indicators serve as a persistent notification system.

Consistency

All Indicators behave in the same fashion, application developers can in no way modify certain behaviour like alter the position of the indicators or have different menus for right and left click. This helps the indicators to be consistent and eliminates guess work on behalf of the user.

Interaction

Each indicator has a GTK Menu attached to it which groups commonly used functions of the application together. This way users can control the application without bringing the main window up. For example, a download manager can expose it's pause/resume functionality to it's indicator providing the users a quick and easy way to control downloads without bringing up the main application window.

Examples

Let's create a simple, bare bone appindicator in the python interpreter.

```
import appindicator
import gtk

indicator = appindicator("my-app-name","distributor-logo", appindicator.CATEGORY_APPLICATION_STATUS)
indicator.set_status(appindicator.STATUS_ACTIVE)
menu = gtk.Menu()
indicator.set_menu(menu)
gtk.main()
```

That should get us a simple indicator up and running in the panel with the Ubuntu ('distributor-logo') icon.

Let's add some functions to the indicator and examine the code in detail. We'll create an indicator with following two functions,

- Show or hide main application window
- Quit the application

In order to accomplish this, we'll need to create two menu items and connect the signals to respective functions just like we did in previous chapters.

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

import appindicator
import gtk

class Application():
    def __init__(self):
        '''
            Initializes the Application by creating a new GTK Window and Indicator
        '''
        self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
        self.window.show()
        self.initialize_indicator()

    def initialize_indicator(self):
        '''
            Initializes the Indicator
        '''
        # Create an indicator instance;
        # Arguments: Application name, default icon name and type of indicator.
        self.indicator = appindicator.Indicator("Application␣Name"
                                                ,"info"
                                                ,appindicator.CATEGORY_APPLICATION_STATUS)
```

```
24
25          # Set Icons for various states
26          self.indicator.set_attention_icon("error")
27
28          # Set Indicator status
29          self.indicator.set_status(appindicator.STATUS_ACTIVE)
30
31          # Create a GTK menu
32          self.menu = gtk.Menu()
33
34          # Create Items for the Menu and connect signals
35          self.quit_button = gtk.MenuItem("Quit")
36          self.quit_button.connect('activate',gtk.main_quit)
37
38          # Add an item to the menu
39          self.show_hide_button = gtk.MenuItem("Show/Hide")
40          self.show_hide_button.connect('activate',self.show_hide_func)
41
42          # Add the items to the menu; You can force the position of an Item by using Insert instead of append or add
43          self.menu.add(self.quit_button)
44          self.menu.insert(self.show_hide_button,0)
45
46          #Show all gtk items and tell the indicator to use the menu we just created
47          self.menu.show_all()
48          self.indicator.set_menu(self.menu)
49
50      def show_hide_func(self,widget,data=None):
51          '''
52              Handles showing and hiding of the main application window.
53          '''
54          if self.window.get_visible():
55              self.window.hide()
56          else:
57              self.window.show()
58
59  if __name__ == "__main__":
60      app = Application()
61      gtk.main()
```

The above code creates a gtk.Window object which acts as our example application. It then creates an indicator and connects the indicator items to the Window object. Adding indicators to your applications cannot get simpler than that, or can it?

Indicators with Quickly

All right, so I was lying. It does get a tad simpler thanks to quickly. You can add an indicator to your quickly application using the quickly add command just like dialogs. Just issue 'quickly add indicator' on the command line and boilerplate indicator code will be added to your application. Next time you run the application, an indicator should show up in the panel with some basic items. You can then edit the indicators.py file to customize the indicator.

Messaging Menu

As you install more applications which export some functionality via indicators, the panel gets messier and cluttered. To prevent this from happening the Unity panel groups indicators with common purposes into menus. For example, the messaging menu contains applications related to communication like chat and email clients or notifiers. If your application falls into this category, you should add it to the messaging menu instead of the panel directly.

API

All messaging menu items consist of an item and zero or more sub items. The API calls the main item server and it's sub items indicators. The server is represented by a .desktop file. We'll create a messaging menu entry for a fake application called "My Mail Client".

We'll need an example .desktop file to represent our application. Create a text file with the following contents and save it as "my-mail-client.desktop" anywhere you like. This desktop file will launch the evolution mail client when opened.

.desktop files act as application launchers. They contain information about the applications like launch command, icon, application category. All .desktop files are located at '/usr/share/applications' or ' /.local/share/applications/' former being system wide and later being user specific.

```
#!/usr/bin/env xdg-open
[Desktop Entry]
Name=My Mail Client
Exec=evolution
Icon=evolution
Type=Application
Categories=GNOME;Network;
```

Indicator code

If you are not using quickly for development, you'll have to create a .desktop file for you application and save it to /usr/share/applications at the time of install. If you are using quickly, this will be taken care of automatically.

```
>>> import indicate
>>> server = indicate.indicate_server_ref_default()
>>> server.set_type('message.mail')
>>> server.set_desktop_file('/home/USER/desktop/my-mail-client.desktop')
>>> server.show()

>>> indicator = indicate.Indicator()
```

```
>>> indicator.set_property('subtype','mail')
>>> indicator.set_property('name','New␣Emails')
>>> indicator.set_property('count','10')
>>> indicator.set_property('draw-attention','true')
>>> indicator.show()
```

We can add multiple indicators for the server which will act as sub items. Also, the server and the indicators can be connected to gtk events just like any other gtk widget. This way one can connect clicks on the items to show/hide the main application just like we did in our appindicator section.

Bazaar Revision Control

What is Bazaar?

Bazaar is the Ubuntu and Launchpad revision control system. It is usually called “bzr”.

http://bazaar.canonical.com/

That is, bzr is similar to git, svn, and cvs (other revision control systems). Unlike those systems, bzr leverages Launchpad capabilities and has native support for “distributed development” (covered below).

With bzr. you can:

- Enable a directory for revision control: **`bzr init`**
- Add the files you want: **`bzr add`**
- Commit changes with a log message: **`bzr commit`**
- See the bzr commit log: **`bzr log`**
- See uncommitted changes in a file: **`bzr diff <file>`**
- Push changes to a central “trunk”: **`bzr push <trunk>`**
- Get a trunk and put it in a new “branch” directory named for trunk: **`bzr branch <trunk>`**
- Carry trunk changes into local: **`bzr merge`** and **`bzr pull`**.
- Roll back changes to previous states: **`bzr revert`**
- Display bzr help: **`bzr help`**

The central branch for a project on Launchpad is called the “trunk”.

A directory under bzr revision control is called a “branch”.
More on **`bzr pull`** and **`bzr merge`** below.

And a lot more.

You can use bzr in any directory on your system without any connection to Launchpad.

Launchpad lets you centralize revision control for your project, allowing people to work on common files stored on Launchpad. See Chapter : Launchpad and Your Project

Let’s start with an example of using bzr in a local system to track changes to some files. Then, we will dive into using bzr with others to develop a common “trunk” branch.

Using Bzr Locally

Here, we create a bzr branch, add some files, commit them with the log message, then change one and use bzr diff and revert to undo the change.

1. Create a directory named ’bzrdemo’ and move into it:

   ```
   $ mkdir bzrdemo && cd bzrdemo
   ```

2. Add some files:

   ```
   $ touch file1 file2
   ```

3. Initialize the directory as a bzr branch:

```
$ bzr init
```

4. Add all files in the directory to the branch:

```
$ bzr add
```

- You can add single files by naming them: **bzr add thisfile thatfile**
- You can add all files in a subdirectory by naming the subdirectory: **bzr add <dir>**

5. Commit all file changes. You are prompted to enter a commit message that goes into the bzr log:

```
$ bzr commit
```

You can commit changes to specific files by naming them: **bzr commit file1 file2**

6. Display the bzr log to see your commit message:

```
$ bzr log | less
```

It is useful to pipe the log through `less` so the latest log entries do not scroll out of site.

7. Modify file1 and display status, which indicates the files that have changed and have not been committed:

```
$ bzr status
```

8. Display a diff of file1 that shows the changes that are not yet committed:

```
$ bzr diff file1
```

You can display the bzr diff for all files with **bzr diff** with no file argument.

9. Let's say you don't want to keep the changes to file1 and want to revert file1 to the last commit state:

```
$ bzr revert file1
```

You can revert all files to their last commit state with **bzr revert** with no file argument.

With the commands just covered you can do most of what is needed to work with an isolated branch.

When developing Ubuntu Applications, there is a trunk branch on Launchpad from which people create local branches. These local branches are modified and pushed to trunk. But what happens if trunk has changed in the meantime? This is covered next.

Bzr Distributed Development

While some revision control systems lock down a central resource when one person wants to modify it, bzr takes another path. bzr allow people to simultaneously modify the same resource. In bzr terms, this is called "distributed development."

bzr allows multiple people to simultaneously modify the "same" files.

Distributed development creates a situation where the trunk branch can be changed by someone else after you made a local branch from it and before you try to push you changes to it. Naturally, you are not able to push your branch to trunk if trunk has changed without taking steps. This section covers how to handle this. First covered is the simple case where trunk gets ahead of your local branch. Then, when the two branches have diverged.

When Trunk Gets Ahead of Local

Let's say you pushed some changes to trunk a while ago. Now you want to make some more changes. You still have your local branch. But, someone may have made changes to trunk—trunk may be "ahead" of your local branch.

This is shown in Figure 36.

Before making any changes to your locale branch, bring your local branch up-to-date with the trunk branch with the **`bzr pull`** command.

After that, use **`bzr status`** to see new log messages pulled in from trunk, if any.

You cannot use **`bzr pull`** when your branch has uncommitted changes. Use **`bzr status`** to show whether a branch has uncommitted changes.

Figure 36: Trunk Branch is Ahead of Local Branch

When Trunk and Local 'Diverge'

If trunk is ahead of your local branch *and* you have local committed changes, then the two branches are "diverged". In this case you cannot push your changes to trunk until you first use **`bzr merge`**.

This is shown in Figure 37.

The best approach is to get a fresh trunk and merge your local branch *into* trunk *from* the trunk directory.

As an example, let's say the project is named "blackbird". Because you branched from blackbird's trunk branch, your local branch directory is named "blackbird". After trying to **`bzr push`**, you found the two branches have diverged. Here is how you can resolve this situation.

You could merge trunk changes into your local branch, but that is not a good practice because the log message of the branch being pulled in are "collapsed". Because you do not want to collapse log messages already pushed to trunk, it is a "best-practice" to allow your local log messages to be collapsed. You can do this by getting a fresh branch of trunk and *from inside trunk* executing **`bzr merge <path-to-local-branch>`**.

- Move to your local branch's parent directory, and rename your local "blackbird" directory to "blackbird-local":

```
$ cd .. && mv blackbird blackbird-local
```

Figure 37: Trunk Branch is Diverged from Local Branch

- Get a fresh branch of "blackbird" trunk:

```
$ bzr branch lp:blackbird
```

- Move into the fresh trunk branch directory, "blackbird":

```
$ cd blackbird
```

- Merge your local branch into the trunk branch:

```
$ bzr merge ../blackbird-local
```

In many cases, bzr merges diverged files automatically. If so, you can simply **`bzr commit`**.

Sometimes, bzr needs your help to merge diverged files. If so, bzr reports the files that need your attention. In this common case, you need to edit the files manually. bzr inserts labels into the text that show the parts of the files that need your attention. You must edit the files until they are correct.

Use **`bzr conflicts`** to list files that need your attention.

Then, execute **`bzr resolve`** to notify bzr you have the files in their correct state.

Lastly, commit the files with **`bzr commit`**

Now, this trunk branch is:

- The up-to-date trunk branch
- With your local changes merged in

You can push your merged trunk to Launchpad with **`bzr push`**.

Launchpad and Your Project

Launchpad is a web site that hosts open source development projects.

Its first project was Ubuntu: http://launchpad.net/ubuntu

Launchpad now hosts thousands of projects. Each project benefits from a dedicated bug tracking system, a translation web portal, bzr revision control, "teams" for permissions, and Personal Package Archives (PPAs), which automatically build source packages into binary (installable) packages and publish them.

Here, we quickly cover the points necessary to use Launchpad for quick success for opportunistic Ubuntu application development.

Naturally, before you can do anything in Launchpad, you need to create a user account.

http://launchpad.net

Open a project on Launchpad by adding the project name to the URL: http://launchpad.net/unity

Launchpad has an extensive help system that covers everything you need to know: http://help.launchpad.net/FrontPage

Create your user account from the launchpad home page: http://launchpad.net/

Getting Started

Let's suppose that you have an idea for a new Ubuntu application. This means a new Debian source package, developed, built, translated, published. You have started with Quickly locally. You also need bug tracking, a development team with source file revision control, and life cycle management tools.

This is where Launchpad *projects* comes in.

You can experiment with Launchpad, including creating projects and teams, with http://staging.launchpad.net. Note that work on staging is automatically deleted periodically, so it is strictly for temporary experimentation.

Creating a Launchpad Project

Now that your user account is created, you can create a Launchpad project here: https://launchpad.net/projects/+new

There are a few fields you need to fill in that are self-explanatory.

Your new project home page is: https://launchpad.net/<project>.

For example, let's say I created a project named "blackbird". Its home page would be: https://launchpad.net/blackbird

You can also create a new project from the launchpad home page: https://launchpad.net

If I created blackbird project on staging, its URL would be: https://staging.launchpad.net/blackbird

Creating a Launchpad Team

Although you may be ready to push your local Quickly project branch to the Launchpad project, you know other people who want to help out with development. You probably want to limit who can push changes to just these people. You need a Launchpad *team*.

Create your new team here: https://launchpad.net/people/+newteam

Your new team's home page is: https://launchpad.net/ <team>

For example, let's say I created a team named "blackbird-team". Its home page would be: https://launchpad.net/~blackbird-team.

Teams always start with a tilde ("~") in Launchpad URLs.

Note, *teams* also host Launchpad PPAs.

Enabling Project Code Hosting

Now that your team is in place, you can enable your project to host bzr branches.

Enable code hosting with the project's **Code** tab and then **Configure code hosting**.

Use the displayed options to:

- Create a new, empty branch
- Set the branch name to "trunk"
- Set the branch owner to your team

Projects host branches. Teams maintain branches. So you can see branches for a project or for a team, and the URLs reflect this.

- *Project's* code is here: https://code.launchpad.net/<project>
- *Team's* code is here: https://code.launchpad.net/~<team>
- *Team's project* code is here: https://code.launchpad.net/~<team>/<project>

But, we haven't yet actually *pushed* any branches, so let's do that.

Pushing to Trunk

Now that code hosting is enabled, you can push your branch to the project's trunk as follows:

```
bzr push lp:<project>
```

Note that because you configured the *trunk* capability when setting up code hosting, you can use the "lp:project" style of URL to push. For example, if the project is named "blackbird", you could push to trunk with

```
bzr push lp:blackbird
```

Note: It may be necessary on the very first push to use the "–use-existing" argument:

```
bzr push --use-existing lp:blackbird
```

Now that you have pushed an actual branch to trunk, you can view information about it, such as commit log messages, from its Launchpad page: https://code.launchpad.net/~<team>/<project>/trunk

For the imaginary blackbird project and blackbird-team, the URL would be: https://code.launchpad.net/~blackbird-team/blackbird/trunk

From any such Launchpad branch page, you can display all files in the branch and see what has changed with each commit (bzr log messages and color-coded diff visualizations).

Tip: bzr "merge proposals" let people propose commits with a review process and other helpful work flow.

Now that you and your team can push directly to trunk, what next?

To push *non-trunk* branches to you project, use a *complete* bzr-style URL that specifies the team, the project and the branch: **`lp:~<team>/<project>/branch`**. For example, I might want to push an experimental branch to blackbird as follows: **`bzr push lp:~blackbird-team/blackbird/experimental`**

Next Steps

Creating Bug Tracking

Before you release and distribute your project, you are likely to go through a period of development in which bugs are found and need to be tracked to resolution (to "fix-committed" status).

Launchpad provides project bug tracking for this.

See the **Bugs** tab of the project home page.

First, you need to configure bug tracking for your project by clicking the **Bugs** tab of your project home page and using **Configure bug tracking**.

Simply provide the few bits of required information, and bug tracking is enabled for the project and available on the project's **Bugs** tab.

You can also open the **Bugs** page with a URL: http://bugs.launchpad.net/<project>

Bug tracking allows you and others to create bugs, assign them, change their status, add tags to them, and target bugs for any "milestones" you create for your project.

So let's take a look at milestones.

Managing Development to Milestones

The "finish line" is the release, but to get there, you may want to go through a few phases. For example, you might want an alpha phase (proof of concept) and one or more beta phases (to test and find and fix bugs).

Milestones help coordinate development activities.

You can create milestones for each "series" and then add those milestones from a drop-down list to individual bugs. For example, you can target a set of bugs to a "beta-2" trunk milestone. You can also view just those bugs that have been targeted to any milestone.

"Series" enable differentiation of key branches. For example, the *trunk* series is the current development focus. You may use other series for other Ubuntu releases, for example 10.10 (Maverick).

To create a milestone:

1. Navigate to your project's code page: https://code.launchpad.net/<project>
2. Find the trunk branch **Series: trunk**, and click on the trunk link there. This takes you to a page like this: https://launchpad.net/<project>/trunk
3. Use **Create milestone** to create the milestone name. Fill in any other fields you find useful.

Now, you can target a bug to a milestone from the bug web page (open the bug) with **Target to milestone**.

Milestones are displayed in various places. If you click one, you can see bugs that are targeted to it.

What good are bugs if your application is not built and distributed? For that, you need a PPA.

Creating your Team PPA

You can use Quickly to build your project locally (for testing). But, use Launchpad builds for distribution (at least during development—other distribution option are available for wide-spread distribution.)

That is what PPAs are for: predictable, uniform source package builds into binaries (for target CPU architectures) and hosting them for distribution.

PPAs distribute both *source* packages and their built *binary* packages.

To create a PPA for your team::

1. Navigate to your team's code page: https://code.launchpad.net/~<team>
2. Use **Create a new PPA**

You can also use the **dput** command to push a source package to a PPA.

A few key points about PPAs:

- PPAs host and build packages for multiple different Ubuntu series
- PPAs build packages for multiple different CPU architectures
- You can display details about packages with the **View package details** link. From the details page, you can download the package (source and binary) and see the package build log.
- You can install from and push to the PPA using the instructions at the top of the page

We have covered some (but by no means all) Launchpad features. Did you know that you can help develop Launchpad?

Develop Launchpad

Launchpad is open source. That means you can help out, by finding and fixing bugs, and even by adding new features.

See https://dev.launchpad.net/

Persistent Data

Introducing gstreamer

In this chapter you will learn about the persistent data: desktopcouch, reading and writing text files and writing binary files.

Placeholder file.

Desktopcouch

Figure 38: Indicators in action

What is desktopcouch

Desktopcouch is shared storage for all your applications, synchronized to all your computers, without any work by developers or users.

Desktopcouch is built to synchronize your data between different machines. So if your application stores its settings and its data in desktopcouch, then your users will find that using your app on their laptop means that it's also set up and ready to use on their netbook or their desktop computer, without them having to do any work to have that happen, and (importantly) without you having to do any work to enable that. Desktopcouch takes care of all the details of synchronization for you, so you don't have to even think about it; all your users need to do is use Ubuntu One, and support for synchronizing data to Ubuntu One is built right in to desktopcouch out of the box.

Storing your own data in desktopcouch is only half the win, though. Because it's a standard central database for everyone's applications to store data in, you can make use of other applications' data as well! Do you want your apps to use the user's addressbook? That data is in desktopcouch for you to make use of. Their bookmarks? In desktopcouch. If your app works with a particular sort of information — let us say, to do lists — then other apps that

also work on tasks can all work on the same to do lists! You can concentrate on making your chosen app have a great UI without having to worry about writing "importers" from other apps.

Using desktopcouch also opens up your application to be able to share data with the web. If an app stores its data in desktopcouch, then it will also be possible to write a companion web application which works with your users' data, by using that data from Ubuntu One's web service. So you can build a great desktop application and a great web application and have them work on the same data!

Working with desktopcouch

Desktopcouch is a database, but it's a little different from any SQL databases that you may have experience with.

It is CouchDB.

FIXME FIXME introductory verbiage

To demonstrate all this, let's build a simple application to store a list of DVDs I own. We'll skip over actually creating the application itself, and just talk about the desktopcouch parts.

FIXME insert application branch here

So, we have an application which can store and show your list of DVDs, but it doesn't know how to actually store things yet! That's what we'll build here. Our "back end", the bit which does the storage, looks like this:

```
class DesktopcouchBackend(object):
    def __init__(self):
        pass

    def add_movie(self, name, actors, director):
        pass

    def get_movie(self, movie_id):
        pass

    def get_rendering_widget(self):
        """Get something to render the data from this backend.

           This should return a widget which knows how to render the data
           stored in this backend. If you need signals on it, set them in here.
        """
        pass

    def get_dvds_by_director(self, director):
        """Return a list of DVD names with the director as passed."""
```

```
        pass
```

Databases

Records are stored in databases. You can create a database for your application, or you can use a database that already exists. For our DVD application, we'll want our own database. Creating a database is done like this:

```
from desktopcouch.records.server import CouchDatabase
db = CouchDatabase("dvds", create=True)
```

Adding `create=True` means that if the database doesn't exist, it's created (it still works if the database existed already). So you should pass `create=True` most of the time, unless you're working with someone else's database and want to give up if that database isn't there.

So we can update our DVD backend like so:

```
DB_NAME = "dvds"


class DesktopcouchBackend(object):
    def __init__(self):
        self.db = CouchDatabase(DB_NAME, create=True)
```

Records

Your data is stored in records; a record defines a thing. So we'll have one record per DVD, and that record will store the DVD's name, the actors in it, and the name of the director.

Every record has a record type. Record types define what sort of record you're dealing with; you get to think them up. So if you're storing details of your DVDs and CDs in desktopcouch, a DVD would be one record type and a CD would be another (because they refer to different things). A record type should be a URL, and that URL should point at a page that describes the record type. For example, the record type for "contacts", used by Evolution and Akonadi and Ubuntu One, is http://www.freedesktop.org/wiki/Specifications/desktopcouch/contact – at that URL there's a description of the contents of a contact record.

To create a record, use `desktopcouch.records.records.Record`:

```
my_record = Record({"name": "Stuart Langridge",
    "project": "Desktop Couch", "hair_colour": "red" },
    record_type='http://example.com/testrecord')
```

A created Record object isn't automatically stored in the database. To save your newly created Record into the database, use the database's `put_record` method:

```
db.put_record(my_record)
```

Our desktopcouch backend needs to create and save a record for a particular DVD in `add_movie`:

```
RECORD_TYPE = "http://developer.one.ubuntu.com/data/sprint/dvd-collection/dvd"
class DesktopcouchBackend(object):

    ...

    def add_movie(self, name, actors, director):
        """Save the data in desktop couch."""
        r = Record({
            "name": name,
            "actors": actors,
            "director": director
        }, record_type=RECORD_TYPE)
        self.db.put_record(r)
        return r
```

Retrieving records back out of desktopcouch again is done, at first, by ID. Every record in desktopcouch has an ID, once it's been saved to the database. Calling `db.put_record` actually returns the ID of the record you've just put, and given an ID, you can get the record for it:

```
r = Record({"title": "Ubuntu Developer Manual"}, record_type="http://example.com/books")
record_id = db.put_record(r)
...
record = db.get_record(record_id)
print record["title"]
```

So our desktopcouch backend can very easily get a record for a given DVD, because the main program already knows the ID it's looking for:

```
class DesktopcouchBackend(object):

    ...

    def get_movie(self, movie_id):
        return self.db.get_record(movie_id)
```

Records can be edited just like Python dictionaries:

```
record = db.get_record(record_id)
print record["title"]
record["title"] = "New Title"
db.put_record(record)
```

Batch storage

Storing lots of data into desktopcouch all at once is best done as a batch. Use `db.put_records_batch()`:

```
records = []
for i in range(100):
    r = Record({"integer": i}, record_type="http://example.com/records/integer")
    records.append(r)
db.put_records_batch(records)
```

This is considerably faster than putting each record individually.

Querying your data

Being able to store records and get them back again by ID is all well and good, and you can go a long way just with that. Sometimes, though, you want to slice and dice your data in a different way, and that's when you use views. A view, in desktopcouch, is a pre-defined way to slice up the data differently than just "by record".

FIXME FIXME write more about views

Tools

Resources

record types - DONE basic storage and retrieval of records - DONE views changes feed batch uploads - DONE attachments working with the web - authentication, tokens, oauth tools - futon, slipcover, ubutnuone-couchdb-query, desktopcouch-pair, couchgrid references to other resources - freedesktop, mailing list, couchdb docs

Distributing Your Work

Distribution

In this chapter, we will explore how to share and distribute your program with other people. From just a few friends to the entire Ubuntu community sharing your work not only enhances the community but lets users and other developers test, use and even help you make your work better. By the end of this chapter, you should be able to publish packages into Personal Package Archives (PPAs) and submit applications to the Ubuntu Software Center.

Overview

When users install your application, they'll be installing a file called a "binary package," a file that ends with the .deb extension. This is transparent to your users but knowing how to generate packages and publish them is important for testing and releasing your software.

Users will either install your application from a PPA or the Ubuntu Software Center. A PPA is a channel for delivering application updates to users that you have full control over. It's most useful for pre-release versions or to quickly and easily share new versions. But to make your application available to a wider audience, you should submit your application to the Ubuntu Software Center, which we'll cover that process at the end of this chapter.

Getting Set Up

One thing you probably should do if you're going to be publishing software is have a project web page. See https://help.launchpad.net/Projects/Registering for more information on registering a project on Launchpad.

Another good thing to do before actually sharing your application is to make sure that certain bits of information about your application are filled out. This includes project contact information, a description of what your application does and any other helpful information.

Run the following command from your Quickly project directory:

```
$ gedit setup.py
```

This file contains the code to package your software. Most of this is not intended for direct editing. But near the bottom you will see a code block that looks like the following:

```
1 DistUtilsExtra.auto.setup(
2     name='test',
3     version='0.1',
```

```
    #license='GPL-3',
    #author='Your Name',
    #author_email='email@ubuntu.com',
    #description='UI for managing ...',
    #long_description='Here a longer description',
    #url='https://launchpad.net/test',
    cmdclass={'install': InstallAndUpdateDataDirectory}
    )
```

Change these fields to something more suitable for your project. Make sure the line isn't commented out (*i.e.*, make sure to remove the '#' character at the beginning of the lines you change). You should end up with something like the following:

```
DistUtilsExtra.auto.setup(
    name='test',
    version='0.1',
    license='GPL-3',
    author='Oliver Twist',
    author_email='twist@example.com',
    description='Example feed reader',
    long_description='This test program lets you read all your favorite RSS feeds',
    url='https://launchpad.net/test',
    cmdclass={'install': InstallAndUpdateDataDirectory}
    )
```

Testing Locally

It is important to make sure that your application installs correctly before you publish your work. You will need to generate a binary package file and install it manually for testing.

Starting in your Quickly project directory, run the following command:

```
$ quickly package
```

This will generate a test_0.1_all.deb in the directory above your current one. Install it with the following command:

```
$ sudo dpkg -i ../test_0.1_all.deb
```

Your application is installed! You can run it by going to the Applications menu or running **test** on the command line.

Publishing Development Releases

To share an in-progress version of your application, you can publish into a PPA. Let's say you have a ppa called "testing."

```
$ quickly share --ppa testing
```

Now you can go to https://launchpad.net/people/+me/+archive/testing and see the package building. Wait 30 minutes then follow the instructions on the page to add the PPA to your system.

Publishing Stable Releases

To publish a new stable version of your application, again you'll publish into a PPA. It's recommended you use a separate PPA from your pre-release versions. For this example you will use a PPA called "stable" to release version 1.0:

```
$ quickly release --ppa stable 1.0
```

If you don't specify a version, a default version of year.month.release will be used. For example, your first release during July 2011 will be versioned 11.07. Your next release that month will be 11.07.1.

Now you can go to https://launchpad.net/people/+me/+archive/stable and see the package building. Wait 30 minutes then follow the instructions on the page to add the PPA to your system.

Submitting to the Ubuntu Software Center

When you're ready to go the final step and put your software in the Ubuntu Software Center, you will follow the steps outlined on the Ubuntu Wiki: https://wiki.ubuntu.com/AppReviews

The process includes:

1. Preparing your application for assessment
2. Submitting your application to the Application Review Board via Launchpad
3. Incorporating feedback from the Board

When the application is approved by the Board it will be imported to the Ubuntu Software Center.

Glossary

Canonical Canonical, the financial backer of Ubuntu, provides support for the core Ubuntu system. It has over 310 paid staff members worldwide who ensure that the foundation of the operating system is stable, as well as checking all the work submitted by volunteer contributors. To learn more about Canonical, go to http://www.canonical.com. 7

GUI The GUI (which stands for Graphical User Interface) is a type of user interface that allows humans to interact with the computer using graphics and images rather than just text. 8

IDE The IDE (which stands for Integrated Development Environment) is a graphical interface particularly suited the programming languages. It usually includes context highlighting of code to enable easier reading of code. It also makes commonly used functions available via menus. 9

Credits

This manual wouldn't have been possible without the efforts and contributions from the following people:

Team Leads

Belinda Lopez—Team Lead
Kevin Godby—Lead TEXnician
Rick Spencer—Project Lead
Ryan MacNish—Author/Coordinator
Kyle Nitsche—Author/Translation Coordinator
Chris Woollard—Technical
Thorsten Wilms—Design
Luke Jennings—Quickshot developer
Neil Tallim—Quickshot developer
Simon Vermeersh—Quickshot developer

Authors

Didier Roche
Owais Lane
Mike Terry
Stuart Landgridge
Rick Spencer

Editors

Chris Woollard
Belinda Lopez

Designers

K. Vishnoo Charan Reddy
Wolter Hellmund
Benjamin Humphrey
David Nel
Thorsten Wilms

Developers

Adnane Belmadiaf
Kevin Godby
Luke Jennings
Neil Tallim
Simon Vermeersh

Translation Editors

Xuacu Saturio
Shazedur Rahim Joardar
Daniel Schury
John Xygonakis
Chris Woollard
Fran Diéguez
Hannie Dumoleyn
Shrinivasan

Special thanks

Chris_Ilias
Bo
underpass
jehurd
cl58
kjhass
djstsys
mozilla_help_view_project
Joey-Elijah Alexithymia
Jono Bacon
Manualbot
Chris Johnson
Elan Kugelmass
Elizabeth Krumbach
Josh Leverette
Walter Méndez
Martin Owens
Tim Penhey
Andy Piper
Alan Pope
Matthew Paul Thomas

The Ubuntu Documentation Team

The Ubuntu Community Learning Project

Index

A The Python Language Crash Course

Introducing the python language

Python is a language and a standard library. In this chapter you will receive a very brief introduction to the python language, as well as a few words on some of the differences between Python programming and programming in some other languages that you may be used to. The purpose of this chapter is to help you get started with the later chapters. You will not become a Python guru by reading this chapter. There are several great introductions to the Python language, including the stellar, and totally free, "Dive into Python", by Mark Pilgrim. This is available for free at http://diveintopython.org/.

The Python language reference is available at http://docs.python.org/ref/ref.html.

Python also has a rich and highly functional library. Later chapters will delve into some key parts of this library. The full Python library documentation can be found at http://docs.python.org/lib/.

Case sensitivity

Python is a case sensitive language. So `Print` is not the same as **`print`**, and `x` is not the same as `X`.

White space and indentation

Python clauses do not have closing statements, characters, or keywords. For example, in Visual Basic every `If Then` is closed by an `End`, and in C based languages, every `{` is closed by an `}`. Python does this quite differently, by using indentation level. Indentation level controls nesting and scope in Python. Every line at a particular indentation level has the same level of scope. Observe the following loop:

```
>>> for x in range(1,10):
...   print x
...
1
2
3
4
4
5
6
7
8
```

```
9
```

Note that line two is indented one space compared to line one. This indentation is what tells Python that line two is nested inside the for loop. Because the indentation level controls nesting and scope, Python does not need to close such statements.

Here's some code for a slightly more complex example program:

Lines 1, 2 and 5 have the same scope. Line 3 is inside the for loop, and line 6 is inside the conditional.

This may seem very unnatural and constrictive at first, but don't worry, you ill quickly adapt to it. Many programmers find that they naturally indent their code anyway, so it doesn't change how they work much.

Declaring variables in python

It's easy to declare a variable in python.

```
>>> x = 1
>>> print x
1
```

Notice that when the variable x was created, no type specifier was included.

Note that a variable has a "type" after its been created. So a variable of a type cannot be used as a variable of another type without being converted. Because x was set to an integer above, this doesn't work:

```
>>> x = 1
>>> print x + " is a string"
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

You can only concatenate two strings together, and Python won't turn your integer into a string.

This also wont work:

```
>>> x = "string"
>>> x += 1
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

You can only increment an integer or float.

However, you can convert an integer to a string and do this:

```
>>> x = 1
>>> print str(x) + " is a string"
1 is a string
```

You can also convert a string to an integer:

```
>>> x = "1"
>>> x = int(x)
>>> x += 1
>>> x
2
```

Numeric types

There are four different numeric types that are native to python, but typically, you'll only care about the difference between an integer and a float. In a nutshell, a float has a decimal point, and an integer doesn't.

Integer example:

```
>>> x = 5
>>> type(x)
<type 'int'>
>>> x = int(5)
>>> type(x)
<type 'int'>
>>> x = int(5.01)
>>> x
5
>>> type(x)
<type 'int'>
```

Float example:

```
>>> x = 5.0
>>> type(x)
<type 'float'>
>>> x = float(5)
>>> x
5.0
>>> type(x)
<type 'float'>
```

Numeric operations

Python uses the common operators for mathematical calculations.

Addition:

```
>>> x = 5
>>> y = 5
>>> x + y
10
```

Subtraction:

```
>>> x = 10
>>> y = 5
>>> x - 5
5
```

Multiplication:

```
>>> x = 5
>>> y = 5
>>> x * y
25
```

Division:

```
>>> x = 25
>>> y = 5
>>> x / y
5
>>> x = 26
>>> y = 5
>>> x / y
5
```

Mixing numeric types in calculations

One application of Python's dynamic typing is that you can mix different numeric types in mathematical calculations without casting or converting to the same type. The data type of the resulting calculation will be of the least restrictive type used in the calculation. See the following examples:

```
>>> x = 5
```

```
>>> y= 5.0
>>> x * y
25.0
>>> x = 25
>>> y = 5.01
>>> x / y
4.9900199600798407
```

Strings

Python strings are very functional. They have many useful built in functions for things like manipulating case, finding substrings, etc. Refer to the Python reference for the whole list of functions.

When declaring strings, you can use either double quotes or single quotes. You can concatenate strings using the "+" operator or the "+=" operator. You can convert objects to strings using the "string()" function.

```
>>> x = 'string'
>>> x
'string'
>>> x = "string"
>>> x
'string'
>>> x = "x"
>>> x
'x'
>>> x += " is a string"
>>> x
'x is a string'
>>> x = "x"
>>> y = " is a string"
>>> x + y
'x is a string'
>>> y + x
' is a stringx'
>>> x = 1.01
>>> str(1.01) + " is a string"
'1.01 is a string'
```

None

In Python, an uninitialized object has the value of "None". This is similar to "null" in C#, C/C++ and Java and similar to "Nothing" in Visual Basic. You can also set a variable to "None" after it has been initialized. You can treat "None" as an object and test if a variable has the identity of "None". This is a good way to avoid possible errors while working with a variable that may be set to "None".

```
>>> x
```

```
>>> print x
None
>>> x = 1
>>> print x
1
>>> x = None
>>> print x
None
>>> x is None
True
>>> x / 10
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'NoneType' and 'int'
```

Lists and tuples

Lists and tuples are used where you would have used collection objects in visual basic, or arraylists in .NET, or arrays in many other languages and libraries. The key difference between a list and a tuple is that a tuple is immutable (you can't change them after they are created), where as a list is designed to be changed at run time.

Unlike in some languages, Python lists and tuples can combine any object types. So they can be a mix of integers, strings, other lists and tuples, etc.

Creating lists and tuples

To create a tuple, use "()":

```
>>> wintermonths = ("December","January","February")
>>> wintermonths
('December', 'January', 'February')
```

To create a list, use "[]":

```
>>> grades = ["A","A-","B","C"]
>>> grades
['A', 'A-', 'B', 'C']
```

To create a list with multiple object types, use the same syntax:

```
>>> grades = ["A-","B+",2.5]
>>> grades
['A-', 'B+', 2.5]
```

Accessing items in lists and tuples

To access an item in a tuple or a list, use a "[]". One cool thing about Python is that you can use a negative number to index from the end. Starting from

the beginning, lists and tuples are zero based. Starting from the end, they are -1 based.

To access the first element use:

```
>>> wintermonths
('December', 'January', 'February')
>>> wintermonths[0]
'December'
```

To acces the last element use:

```
>>> grades
['A', 'A-', 'B', 'C']
>>> grades[-1]
'C'
>>> grades[-2]
'B'
```

You can test if a list or tuples contains an item using "in":

```
>>> grades = ["A","A-","B","C"]
>>> grades
['A', 'A-', 'B', 'C']
>>> "A" in grades
True
>>> "F" in grades
False
```

You can find the first position of an item in a list using the member function "index()":

```
>>> grades
['A', 'A-', 'B', 'C']
>>> i = grades.index("A")
>>> i
0
>>> i = grades.index("B")
>>> i
2
```

Note that Python will throw an error if the item is not in the list:

```
>>> grades
['A', 'A-', 'B', 'C']
>>> i = grades.index("F")
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: list.index(x): x not in list
```

It's a good idea to test for it first:

```
>>> grades
['A', 'A-', 'B', 'C']
>>> i = -1
```

```
>>> if "F" in grades:
...   i = index("F")
...
>>> i
-1
```

Note that tuples do not have the "index" function.

Modifying lists and tuples

Pretty much all you can do with a tuple is create it, access items in it and loop through it. The one exception is that you can turn it into a list using the "list" function:

```
>>> wintermonths
('December', 'January', 'February')
>>> wintermonths = list(wintermonths)
>>> wintermonths
['December', 'January', 'February']
```