Welcome to

Python

The Complete Manual

Python is a versatile language and its rise in popularity is certainly no surprise. Its similarity to everyday language has made it a perfect companion for the Raspberry Pi, which is often a first step into practical programming. But don't be fooled by its beginner-friendly credentials – Python has plenty of more advanced functions. In this new edition, you will learn how to program in Python, discover amazing projects to improve your understanding, and find ways to use Python to enhance your experience of computing. You'll also create fun projects including programming a *Space Invaders* clone and teaching your Raspberry Pi to multi-task. Let's get coding!

Contents

What you can find inside the bookazine

Code & create with Python!

Get started with Python

8 **Masterclass**
Discover the basics of Python

Introducing Python

26 **Make web apps**
Master this starter project

32 **Build an app for Android**
Take your apps on the move

40 **50 Python tips**
A selection of handy tips

Work with Python

50 **Replace your shell**
Say goodbye to Bash

58 **Scientific computing**
Discover NumPy's power

64 **Python for system admins**
How to tweak your settings

72 **Scrape Wikipedia**
Start using Beautiful Soup

Create with Python

80 **Tic-tac-toe with Kivy**
Program a simple game

86 **Make a Pong clone**
Enhance your game skills

88 **Program a Space Invaders clone**
Have fun with Pivaders

94 **Space Invaders clone 2**
Continue making Pivaders

Use Python with Pi

100 **Using Python on Pi**
Optimise your code

106 **Use Python in Minecraft**
Produce fantastic creations

110 **Handle multiple task**
Learn to multi-task with your Raspberry Pi

114 **Create a Pi-powered virtual reality setup**
Use Python-VRZero

120 **Find and check your phone**
Discover and log Bluetooth devices

Get started with Python

Always wanted to have a go at programming? No more excuses, because Python is the perfect way to get started!

Python is a great programming language for both beginners and experts. It is designed with code readability in mind, making it an excellent choice for beginners who are still getting used to various programming concepts.

The language is popular and has plenty of libraries available, allowing programmers to get a lot done with relatively little code.

You can make all kinds of applications in Python: you could use the Pygame framework to write simple 2D games, you could use the GTK libraries to create a windowed application, or you could try something a little more ambitious like an app such as creating one using Python's Bluetooth and Input libraries to capture the input from a USB keyboard and relay the input events to an Android phone.

For this tutorial we're going to be using Python 2.x since that is the version that is most likely to be installed on your Linux distribution.

In the following tutorials, you'll learn how to create popular games using Python programming. We'll also show you how to add sound and AI to these games.

Hello World

Let's get stuck in, and what better way than with the programmer's best friend, the 'Hello World' application! Start by opening a terminal. Its current working directory will be your home directory. It's probably a good idea to make a directory for the files that we'll be creating in this tutorial, rather than having them loose in your home directory. You can create a directory called Python using the command mkdir Python. You'll then want to change into that directory using the command cd Python.

The next step is to create an empty file using the command 'touch' followed by the filename. Our expert used the command touch hello_world.py. The final and most important part of setting up the file is making it executable. This allows us to run code inside the hello_world.py file. We do this with the command chmod +x hello_world.py. Now that we have our file set up, we can go ahead and open it up in nano, or alternatively any text editor of your choice. Gedit is a great editor with syntax highlighting support that should be available on any distribution. You'll be able to install it using your package manager if you don't have it already.

```
[liam@liam-laptop ~]$ mkdir Python
[liam@liam-laptop ~]$ cd Python/
[liam@liam-laptop Python]$ touch hello_world.py
[liam@liam-laptop Python]$ chmod +x hello_world.py
[liam@liam-laptop Python]$ nano hello_world.py
```

Our Hello World program is very simple, it only needs two lines. The first line begins with a 'shebang' (the symbol #! – also known as a hashbang) followed by the path to the Python interpreter. The program loader uses this line to work out what the rest of the lines need to be interpreted with. If you're running this in an IDE like IDLE, you don't necessarily need to do this.

The code that is actually read by the Python interpreter is only a single line. We're passing the value Hello World to the print function by placing it in brackets immediately after we've called the print function. Hello World is enclosed in quotation marks to indicate that it is a literal value and should not be interpreted as source code. As we would expect, the print function in Python prints any value that gets passed to it from the console.

You can save the changes you've just made to the file in nano using the key combination Ctrl+O, followed by Enter. Use Ctrl+X to exit nano.

```
#!/usr/bin/env python2
print("Hello World")
```

You can run the Hello World program by prefixing its filename with ./ – in this case you'd type: ./hello_world.py.

```
[liam@liam-laptop Python]$ ./hello_world.py
Hello World
```

Variables and data types

A variable is a name in source code that is associated with an area in memory that you can use to store data, which is then called upon throughout the code. The data can be one of many types, including:

Integer	Stores whole numbers
Float	Stores decimal numbers
Boolean	Can have a value of True or False
String	Stores a collection of characters. "Hello World" is a string

"A variable is associated with an area in memory that you can use to store data"

Tip

If you were using a graphical editor such as gedit, then you would only have to do the last step of making the file executable. You should only have to mark the file as executable once. You can freely edit the file once it is executable.

Get started with Python

Tip

At this point, it's worth explaining that any text in a Python file that follows a # character will be ignored by the interpreter. This is so you can write comments in your code.

As well as these main data types, there are sequence types (technically, a string is a sequence type but is so commonly used we've classed it as a main data type):

List	Contains a collection of data in a specific order
Tuple	Contains a collection immutable data in a specific order

A tuple would be used for something like a co-ordinate, containing an x and y value stored as a single variable, whereas a list is typically used to store larger collections. The data stored in a tuple is immutable because you aren't able to change values of individual elements in a tuple. However, you can do so in a list.

It will also be useful to know about Python's dictionary type. A dictionary is a mapped data type. It stores data in key-value pairs. This means that you access values stored in the dictionary using that value's corresponding key, which is different to how you would do it with a list. In a list, you would access an element of the list using that element's index (a number representing where the element is placed in the list).

Let's work on a program we can use to demonstrate how to use variables and different data types. It's worth noting at this point that you don't always have to specify data types in Python. Feel free to create this file in any editor you like. Everything will work just fine as long as you remember to make the file executable. We're going to call ours variables.py.

Interpreted vs compiled languages

An interpreted language such as Python is one where the source code is converted to machine code and then executed each time the program runs. This is different from a compiled language such as C, where the source code is only converted to machine code once – the resulting machine code is then executed each time the program runs.

Full code listing

```
#!/usr/bin/env python2

# We create a variable by writing the name of the variable we want followed# by an equals sign, which is followed by the value we want to store in the# variable. For example, the following line creates a variable called# hello_str, containing the string Hello World.
hello_str = “Hello World”
```

The following line creates an integer variable called hello_int with the # value of 21. Notice how it doesn't need to go in quotation marks

```
hello_int = 21
```

The same principal is true of Boolean values

```
hello_bool = True
```

We create a tuple in the following way

```
hello_tuple = (21, 32)
```

And a list in this way

```
hello_list = [“Hello,”, “this”, “is”,
“a”, “list”]

# This list now contains 5 strings. Notice that there are no spaces# between these strings so if you were to join them up so make a sentence # you'd have to add a space between each element.
```

You could also create the same list in the following way

```
hello_list = list()
hello_list.append(“Hello,”)
hello_list.append(“this”)
hello_list.append(“is”)
hello_list.append(“a”)
hello_list.append(“list”)

# The first line creates an empty list and the following lines use the append# function of the list type to add elements to the list. This way of using a# list isn't really very useful when working with strings you know of in # advance, but it can be useful when working with dynamic data such as user# input. This list will overwrite the first list without any warning
```

Get started with Python

as we# are using the same variable name as the previous list.

We might as well create a dictionary while we're at it. Notice how we've aligned the colons below to make the code tidy

```
hello_dict = { “first_name” : “Liam”,
               “last_name”  :
“Fraser”,
               “eye_colour” : “Blue” }
```

Let's access some elements inside our collections# We'll start by changing the value of the last string in our hello_list and# add an exclamation mark to the end. The "list" string is the 5th element # in the list. However, indexes in Python are zero-based, which means the # first element has an index of 0.

Notice that there will now be two exclamation marks present when we print the element

```
print(hello_list[4])
hello_list[4] += “!”
# The above line is the same as
hello_list[4] = hello_list[4] + “!”
print(hello_list[4])
```

Remember that tuples are immutable, although we can access the elements of them like so

```
print(str(hello_tuple[0]))
# We can't change the value of those elements like we just did with the list
# Notice the use of the str function above to explicitly convert the integer
# value inside the tuple to a string before printing it.
```

Let's create a sentence using the data in our hello_dict

```
print(hello_dict[“first_name”] + “ ” + hello_
dict[“last_name”] + “ has ” +
    hello_dict[“eye_colour”] + “ eyes.”)
```

A much tidier way of doing this would be to use Python's string formatter

```
print(“{0} {1} has {2} eyes.”.format(hello_
dict[“first_name”],
                hello_dict[“last_name”],
                hello_dict[“eye_colour”]))
```

Indentation in detail

As previously mentioned, the level of indentation dictates which statement a block of code belongs to. Indentation is mandatory in Python, whereas in other languages, sets of braces are used to organise code blocks. For this reason, it is essential to use a consistent indentation style. Four spaces are typically used to represent a single level of indentation in Python. You can use tabs, but tabs are not well defined, especially if you open a file in more than one editor.

Control structures

In programming, a control structure is any kind of statement that can change the path that the code execution takes. For example, a control structure that decided to end the program if a number was less than 5 would look something like this:

```
#!/usr/bin/env python2
import sys # Used for the sys.exit function
int_condition = 5
if int_condition < 6:
    sys.exit("int_condition must be >= 6")
else:
    print("int_condition was >= 6 - continuing")
```

The path that the code takes will depend on the value of the integer int_condition. The code in the 'if' block will only be executed if the condition is true. The import statement is used to load the Python system library; the latter provides the exit function, allowing you to exit the program, printing an error message. Notice that indentation (in this case four spaces per indent) is used to indicate which statement a block of code belongs to. 'If' statements are probably the most commonly used control structures. Other control

"The path the code takes will depend on the value of the integer int_condition" structures include: the following items, which you should be aware of when using Python:

- **For statements, which allow you to iterate over items in collections, or to repeat a piece of code again a certain number of times;**
- **While statements, a loop that continues while the condition is true.**

We're going to write a program that accepts user input from the user to demonstrate how control structures work. We're calling it construct.py. The 'for' loop is using a local copy of the current value, which means any changes inside the loop won't make any changes affecting the list. On the other hand however, the 'while' loop is directly accessing elements in the list, so you could change the list there should you want to do so. We will talk about variable scope in some more detail later on in the article. The output from the above program is as follows:

```
[liam@liam-laptop Python]$ ./
construct.py
How many integers? acd
You must enter an integer

[liam@liam-laptop Python]$ ./
construct.py
How many integers? 3
Please enter integer 1: t
You must enter an integer
Please enter integer 1: 5
Please enter integer 2: 2
Please enter integer 3: 6
Using a for loop
5
2
6
Using a while loop
5
2
6
```

"The 'for' loop uses a local copy, so changes in the loop won't affect the list"

Full code listing

```
#!/usr/bin/env python2

# We're going to write a program that will ask the user to input an arbitrary
# number of integers, store them in a collection, and then demonstrate how the
# collection would be used with various control structures.

import sys # Used for the sys.exit function
```

The number of integers we want in the list

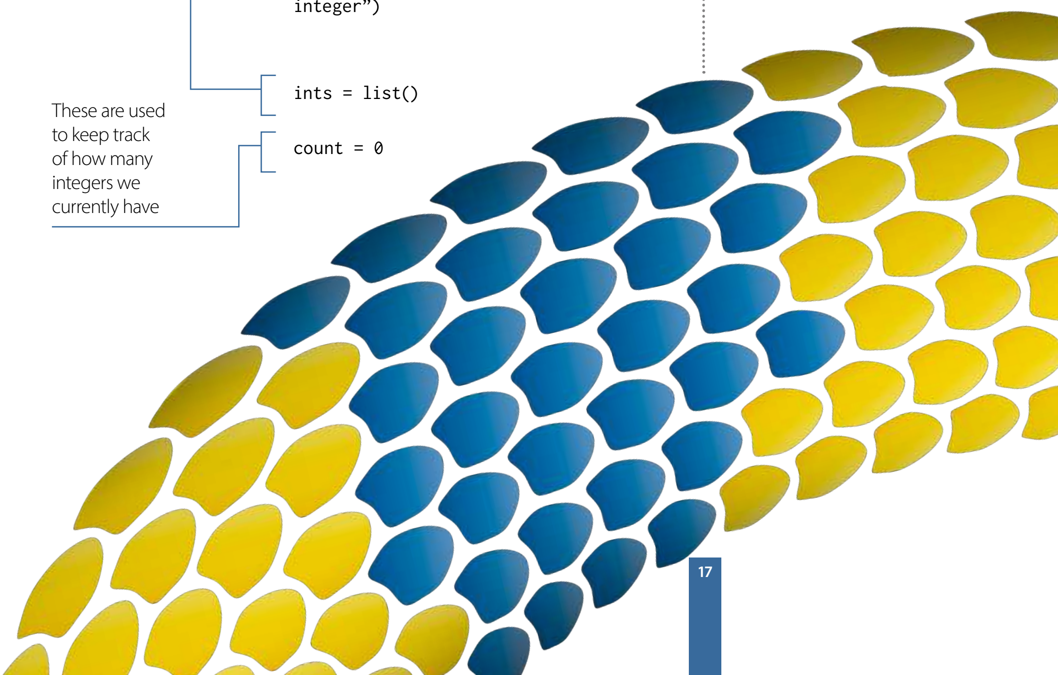
```
target_int = raw_input("How many integers? ")

# By now, the variable target_int contains a string representation of
# whatever the user typed. We need to try and convert that to an integer but
# be ready to # deal with the error if it's not. Otherwise the program will
# crash.
try:
    target_int = int(target_int)
except ValueError:
    sys.exit("You must enter an integer")
```

A list to store the integers

```
ints = list()
```

These are used to keep track of how many integers we currently have

```
count = 0
```



Get started with Python

```
# Keep asking for an integer until we have the required number
while count < target_int:
    new_int = raw_input("Please enter integer {0}: ".format(count + 1))
    isint = False
    try:
        new_int = int(new_int)
    except:
        print("You must enter an integer")

    # Only carry on if we have an integer. If not, we'll loop again
    # Notice below I use ==, which is different from =. The single equals is an
    # assignment operator whereas the double equals is a comparison operator.

    if isint == True:
        # Add the integer to the collection
        ints.append(new_int)
        # Increment the count by 1
        count += 1
```

If the above succeeds then isint will be set to true: isint =True

By now, the user has given up or we have a list filled with integers. We can loop through these in a couple of ways. The first is with a for loop

```
print("Using a for loop")
for value in ints:
    print(str(value))

# Or with a while loop:
print("Using a while loop")
# We already have the total above, but knowing
```

```
the len function is very
# useful.
total = len(ints)
count = 0
while count < total:
    print(str(ints[count]))
    count += 1
```

More about a Python list

A Python list is similar to an array in other languages. A list (or tuple) in Python can contain data of multiple types, which is not usually the case with arrays in other languages. For this reason, we recommend that you only store data of the same type in a list. This should almost always be the case anyway due to the nature of the way data in a list would be processed.

Functions and variable scope

Functions are used in programming to break processes down into smaller chunks. This often makes code much easier to read. Functions can also be reusable if designed in a certain way. Functions can have variables passed to them. Variables in Python are always passed by value, which means that a copy of the variable is passed to the function that is only valid in the scope of the function. Any changes made to the original variable inside the function will be discarded. However, functions can also return values, so this isn't an issue. Functions are defined with the keyword def, followed by the name of the function. Any variables that can be passed through are put in brackets following the function's name. Multiple variables are separated by commas. The names given to the variables in these brackets are the ones that they will have in the scope of the function, regardless of what the variable that's passed to the function is called.

Let's see this in action. The output from the program opposite is as follows:

> "Functions are defined with the keyword def, then the name of the function"

Get started with Python

```
#!/usr/bin/env python2 # Below is a function
called modify_string, which accepts a variable
# that will be called original in the scope of the
function. Anything # indented with 4 spaces
under the function definition is in the
# scope.
def modify_string(original):
    original += " that has been
modified."
  # At the moment, only the local copy of this
string has been modified

def modify_string_return(original):
    original += " that has been
modified."
  # However, we can return our local copy to the
caller. The function# ends as soon as the return
statement is used, regardless of where it # is in
the function.
    return original

test_string = "This is a test string"

modify_string(test_string)
print(test_string)

test_string = modify_string_
return(test_string)
print(test_string)

# The function's return value is stored in the
variable test string, # overwriting the original and
therefore changing the value that is # printed.
```

We are now outside of the scope of the modify_string function, as we have reduced the level of indentation

The test string won't be changed in this code

However, we can call the function like this

```
[liam@liam-laptop Python]$ ./functions_and_
scope.py
This is a test string
This is a test string that has been modified.
```

Scope is an important thing to get the hang of, otherwise it can get you into some bad habits. Let's write a quick program to demonstrate this. It's going to have a Boolean variable called cont, which will decide if a number will be assigned to a variable in an if statement. However, the variable hasn't been defined anywhere apart from in the scope of the if statement. We'll finish off by trying to print the variable.

```
#!/usr/bin/env python2
cont = False
if cont:
    var = 1234
print(var)
```

In the section of code above, Python will convert the integer to a string before printing it. However, it's always a good idea to explicitly convert things to strings – especially when it comes to concatenating strings together. If you try to use the + operator on a string and an integer, there will be an error because it's not explicitly clear what needs to happen. The + operator would usually add two integers together. Having said that, Python's string formatter that we demonstrated earlier is a cleaner way of doing that. Can you see the problem? Var has only been defined in the scope of the if statement. This means that we get a very nasty error when we try to access var.

```
[liam@liam-laptop Python]$ ./scope.py
Traceback (most recent call last):
  File "./scope.py", line 8, in <module>
    print var
NameError: name 'var' is not defined
```

If cont is set to True, then the variable will be created and we can access it just fine. However, this is a bad way to do things. The correct way is to initialise the variable outside of the scope of the if statement.

```
#!/usr/bin/env python2

cont = False

var = 0
if cont:
    var = 1234

if var != 0:
    print(var)
```

Get started with Python

Tip

You can define defaults for variables if you want to be able to call the function without passing any variables through at all. You do this by putting an equals sign after the variable name. For example, you can do:
def modify_string (original=" Default String")

The variable var is defined in a wider scope than the if statement, and can still be accessed by the if statement. Any changes made to var inside the if statement are changing the variable defined in the larger scope. This example doesn't really do anything useful apart from illustrate the potential problem, but the worst-case scenario has gone from the program crashing to printing a zero. Even that doesn't happen because we've added an extra construct to test the value of var before printing it.

> "Google, or any other search engine, is very helpful if you are stuck with anything, or have an error message you can't work out how to fix"

Comparison operators

The common comparison operators available in Python include:

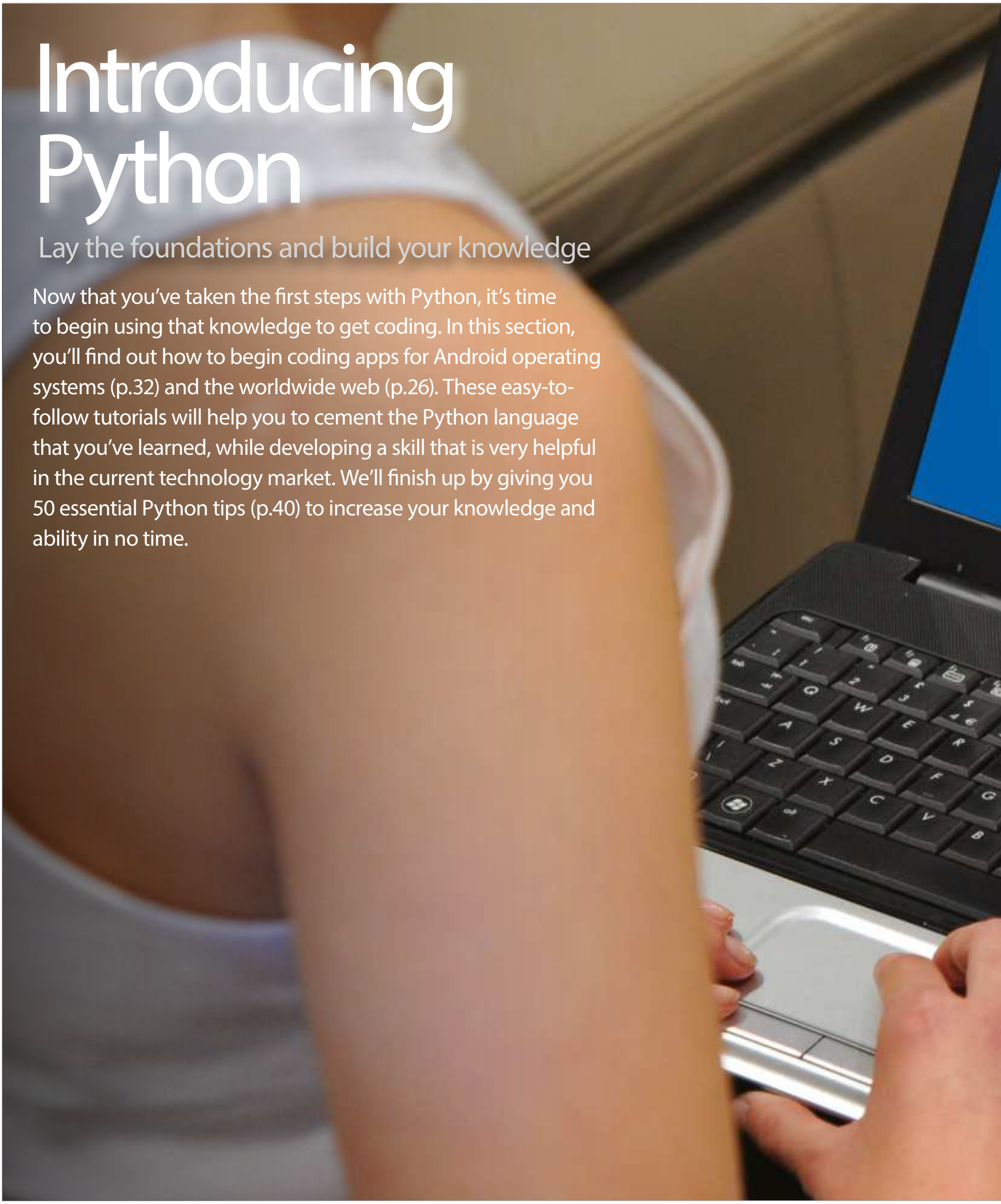
<	strictly less than
<=	less than or equal
>	strictly greater than
>=	greater than or equal
==	equal
!=	not equal

Coding style

It's worth taking a little time to talk about coding style. It's simple to write tidy code. The key is consistency. For example, you should always name your variables in the same manner. It doesn't matter if you want to use camelCase or use underscores as we have. One crucial thing is to use self-documenting identifiers for variables. You shouldn't have to guess what a variable does. The other thing that goes with this is to always comment your code. This will help anyone else who reads your code, and yourself in the future. It's also useful to put a brief summary at the top of a code file describing what the application does, or a part of the application if it's made up of multiple files.

Summary

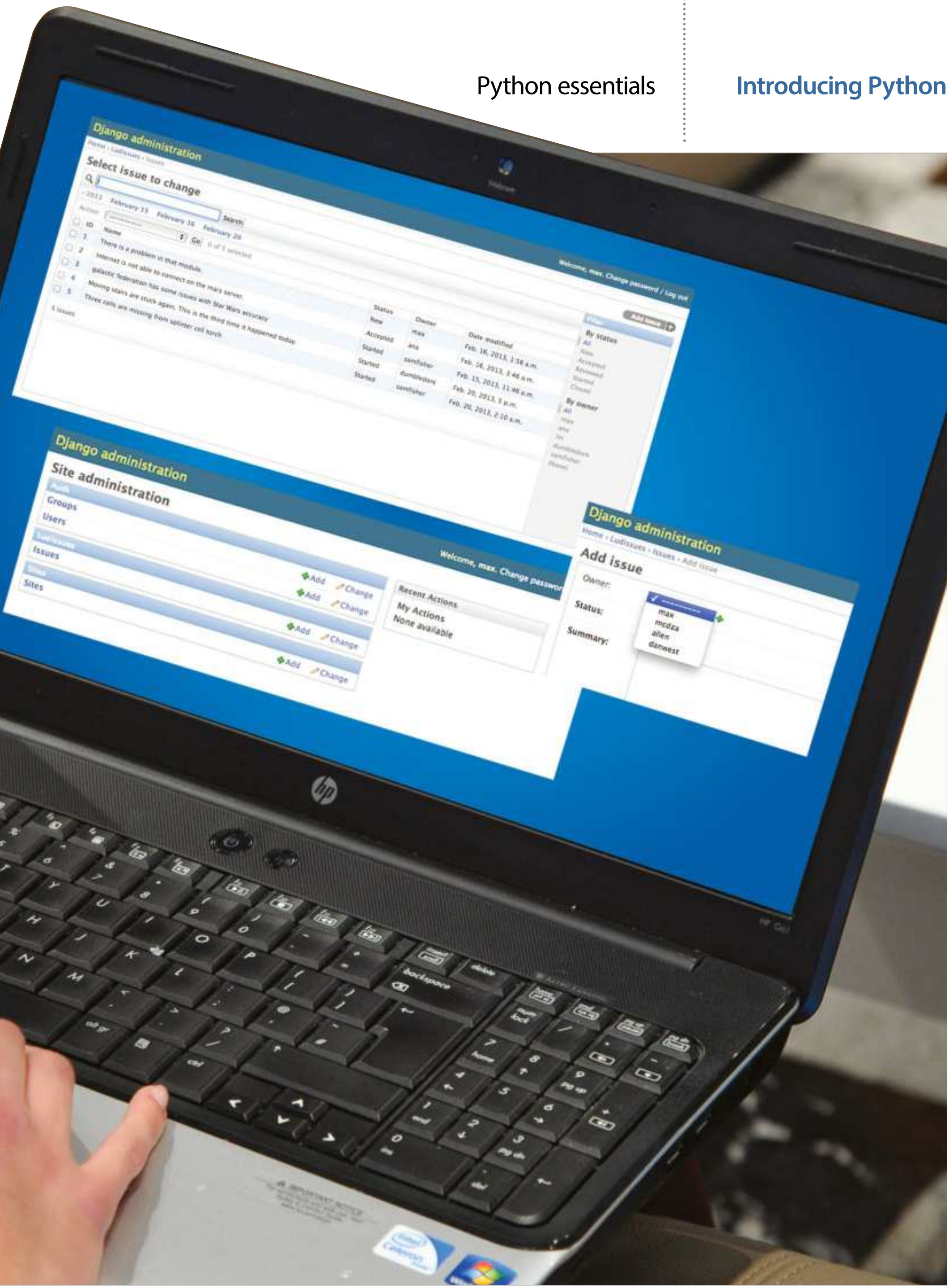
This article should have introduced you to the basics of programming in Python. Hopefully you are getting used to the syntax, indentation and general look and feel of a Python program. The next step is to learn how to come up with a problem that you want to solve, and break it down into small steps that you can implement in a programming language. Google, or any other search engine, is very helpful. If you are stuck with anything, or have an error message you can't work out how to fix, stick it into Google and you should be a lot closer to solving your problem. For example, if we Google 'play mp3 file with python', the first link takes us to a Stack Overflow thread with a bunch of useful replies. Don't be afraid to get stuck in – the real fun of programming is solving problems one manageable chunk at a time.



Introducing Python

Lay the foundations and build your knowledge

Now that you've taken the first steps with Python, it's time to begin using that knowledge to get coding. In this section, you'll find out how to begin coding apps for Android operating systems (p.32) and the worldwide web (p.26). These easy-to-follow tutorials will help you to cement the Python language that you've learned, while developing a skill that is very helpful in the current technology market. We'll finish up by giving you 50 essential Python tips (p.40) to increase your knowledge and ability in no time.


Django administration
Select issue to change
Django administration
Site administration
Groups
Users
Issues
Sites
Recent Actions
My Actions
None available
Django administration
Add issue
Owner:
Status:
Summary:

What you'll need…

Python 2.7:
https://www.python.org/download/releases/2.7/

Django version 1.4:
https://www.djangoproject.com/

Make web apps with Python

Python provides quick and easy way to build applications, including web apps. Find out how to use it to build a feature-complete web app

Python is known for its simplicity and capabilities. At this point it is so advanced that there is nothing you cannot do with Python, and conquering the web is one of the possibilities. When you are using Python for web development you get access to a huge catalogue of modules and community support – make the most of them.

Web development in Python can be done in many different ways, right from using the plain old CGI modules to utilising fully groomed web frameworks. Using the latter is the most popular method of building web applications with Python, since it allows you to build applications without worrying about all that low-level implementation stuff. There are many web frameworks available for Python, such as Django, TurboGears and Web2Py. For this tutorial we will be using our current preferred option, Django.

The Django Project magazine issue tracker

01 The django-admin.py file is used to create new Django projects. Let's create one for our issue tracker project here…

In Django, a project represents the site and its settings. An application, on the other hand, represents a specific feature of the site, like blogging or tagging. The benefit of this approach is that your Django application becomes portable and can be integrated with other Django sites with very little effort.

```
$ django-admin.py startproject
ludIssueTracker
```

A project directory will be created. This will also act as the root of your development web server that comes with Django. Under the project directory you will find the following items…

manage.py: Python script to work with your project.

ludIssueTracker: A python package (a directory with __init__.py file) for

your project. This package is the one containing your project's settings and configuration data.
ludIssueTracker/settings.py: This file contains all the configuration options for the project.
ludIssueTracker/urls.py: This file contains various URL mappings.
wsgi.py: An entry-point for WSGI-compatible web servers to serve your project. Only useful when you are deploying your project. For this tutorial we won't be needing it.

Configuring the Django project

02 Before we start working on the application, let's configure the Django project as per our requirements.
Edit ludIssueTracker/settings.py as follows (only parts requiring modification are shown):
Database Settings: We will be using SQLite3 as our database system here.
NOTE: Red text indicates new code or
updated code.

```
'default': {
        'ENGINE':
'django.db.backends.
sqlite3',
        'NAME': 'ludsite.
db3,
```

Path settings

Django requires an absolute path for directory settings. But we want to be able to pass in the relative directory references. In order to do that we will add a helper Python function. Insert the following code at the top of the settings.py file:

```
import os
def getabspath(*x):
 return os.path.join(os.
path.abspath(os.path.
dirname(__file__)), *x)
```

Now update the path options:

```
@code
TEMPLATE_DIRS = (
 getabspath('templates')
)
MEDIA_ROOT =
getabspath('media')
MEDIA_URL = '/media/'
```

Now we will need to enable the admin interface for our Django site. This is a neat feature of Django which allows automatic creation of an admin interface of the site based on the data model. The admin interface can be used to add and manage content for a Django site.
Uncomment the following line:

```
INSTALLED_APPS = (


    'django.contrib.auth',
    'django.contrib.
contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.
staticfiles',
    'django.contrib.admin',
    # 'django.contrib.
admindocs',
)
```

Creating ludissues app

03 In this step we will create the primary app for our site, called ludissues. To do that, we will use the manage.py script:

```
$ python manage.py startapp
ludissues
```

We will need to enable this app in the config file as well:

```
INSTALLED_APPS = (
    ............
    'django.contrib.admin',
    'ludissues',
)
```

Creating the data model

04 This is the part where we define the data model for our app. Please see the inline comments to understand what is happening here.
From django.db import models:

```
# We are importing the
user authentication module so
that we use the built
# in authentication model
in this app
from django.contrib.auth.
models import User
# We would also create an
admin interface for our app
from django.contrib import
admin

# A Tuple to hold the
multi choice char fields.
# First represents the
field name the second one
repersents the display name
ISSUE_STATUS_CHOICES = (
    ('new', 'New'),
    ('accepted','Accepted'),
    ('reviewed','Reviewed'),
    ('started','Started'),
    ('closed','Closed'),
)
```

> "When you are using Python for web development you get access to a huge catalogue of modules and support"

```
class Issue(models.Model):
     # owner will be a
foreign key to the User
model which is already built-
in Django
     owner = models.ForeignKe
y(User,null=True,blank=True)
     # multichoice with
defaulting to "new"
     status = models.
CharField(max_
length=25,choices=ISSUE_
STATUS_CHOICES,default='new')
     summary = models.
TextField()
     # date time field which
will be set to the date time
when the record is created
     opened_on = models.
DateTimeField('date opened',
auto_now_add=True)
     modified_on = models.
DateTimeField('date modified',
auto_now=True)

     def name(self):
          return self.summary.
split('\n',1)[0]

# Admin front end for the
app. We are also configuring
some of the
# built in attributes for
the admin interface on
# how to display the list,
how it will be sorted
# what are the search
fields etc.
class IssueAdmin(admin.
ModelAdmin):
     date_hierarchy =
'opened_on'
     list_filter =
('status','owner')
     list_display = ('id','n
ame','status','owner','modifi
ed_on')
     search_fields =
['description','status']

# register our site with
the Django admin interface
admin.site.
register(Issue,IssueAdmin)
```

To have the created data model reflected in the database, run the following command:

```
$ python manage.py syncdb
```

You'll be also asked to create a superuser for it:

```
You just installed Django's auth
system, which means you don't
have any superusers defined.
Would you like to create one
now? (yes/no): yes
```

Enabling the admin site

05 The admin site is already enabled, but we need to enable it in the urls.py file – this contains the regex-based URL mapping from model to view. Update the urls.py file as follows:

```
from django.conf.urls import
patterns, include, url
from django.contrib import
admin
admin.autodiscover()

urlpatterns = patterns('',
    url(r'^admin/',
include(admin.site.urls)),
)
```

Starting the Django web server

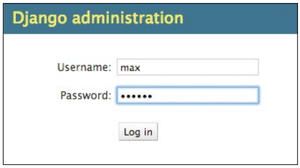
06 Django includes a built-in web server which is very handy to debug and test Django applications. Let's start it to see how our admin interface works…

To start the web server:

```
$ python manage.py
runserver
```

If you do not have any errors in your code, the server should be available on port 8000. To launch the admin interface, navigate your browser to http://localhost:8000/admin.

You will be asked to log in here. Enter the username and password that you created while you were syncing the database.



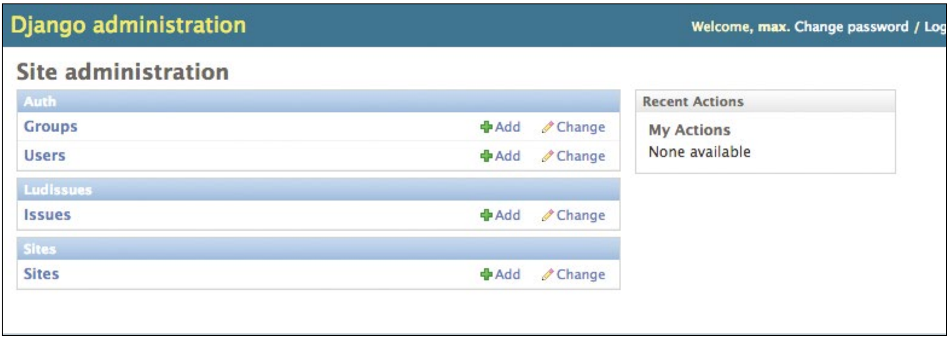
After logging in, you will notice that all the apps installed in your project are available here. We are only interested in the Auth and LudIssues app.

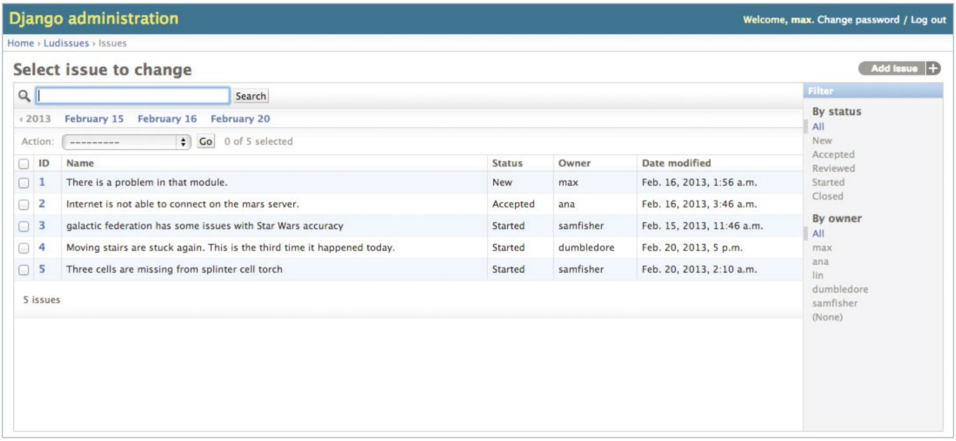
You can click the +Add to add a record. Click the Add button next to Users and add a few users to the site.

Once you have the users inside the system, you can now add a few issues to the system.

Click the Add button next to Issues. Here you will notice that you can enter Owner, Status and Summary for the issue. But what about the opened_on and modified_on field that we

"It's great that the owner field is automatically populated with details of the users inside the site"





defined while modelling the app? They are not here because they are not supposed to be entered by the user. opened_on will automatically set to the date time it is created and modified_on will automatically set to the date time on which an issue is modified.

Another cool thing is that the owner field is automatically populated with all the users inside the site.

We have defined our list view to show ID, name, status, owner and 'modified on' in the model. You can get to this view by navigating to http://localhost:8000/admin/ludissues/issue/.

Creating the public user interface for ludissues

07 At this point, the admin interface is working. But we need a way to display the data that we have added using the admin interface. But there is no public interface. Let's create it now.

We will have to begin by editing the main urls.py (ludIssueTracker/urls.py).

```
urlpatterns = patterns('',

(r'^',include('ludissues.
urls')),
    (r'^admin/',
include(admin.site.urls)),
)
```

This ensures that all the requests will be processed by ludissues.urls first.

Creating ludissues.url

08 Create a urls.py file in the app directory (ludissues/urls.py) with the following content:

```
from django.conf.urls
import patterns, include, url
# use ludissues model
from models import
ludissues

# dictionary with all the
```

```
objects in ludissues
info = {
    'queryset':ludissues.
objects.all(),
}

# To save us writing lots of
python code
# we are using the list_
detail generic view

#list detail is the name of
view we are using
urlpatterns =
patterns('django.views.generic.
list_detail',
#issue-list and issue-detail
are the template names
#which will be looked in the
default template
#directories
    url(r'^$','object_
list',info,name='issue-list'),
    url(r'^(?P<object_
id>\d+)/$','object_
detail',info,name='issue-detail'),

)
```

To display an issue list and details, we are using a Django feature called generic views. In this case we are using views called list and details. This allow us to create an issue list view and issue detail view. These views are then applied using the issue_list.html and issue_detail.html template. In the following steps we will create the template files.

Setting up template and media directories

09 In this step we will create the template and media directories. We have already mentioned the template directory as

```
TEMPLATE_DIRS = (
    getabspath('templates')
)
```

Which translates to ludIssueTracker/ludIssueTracker/templates/. Since we will be accessing the templates from the ludissues app, the complete directory path would be ludIssueTracker/ludIssueTracker/templates/ludissues. Create these folders in your project folder.

Also, create the directory ludIssueTracker/ludIssueTracker/media/ for holding the CSS file. Copy the style.css file from the resources directory of the code folder. To serve files from this folder, make it available publicly. Open settings.py and add these lines in ludIssueTracker/ludIssueTracker/urls.py:

```
from django.conf.urls import
patterns, include, url
from django.conf import
settings
# Uncomment the next two
lines to enable the admin:
from django.contrib import
admin
admin.autodiscover()

urlpatterns = patterns('',
    (r'^',include('ludissues.
urls')),
    (r'^admin/', include(admin.
site.urls)),
    (r'^media/
(?P<path>.*)$','django.views.
static.serve',
     {'document_root':settings.
MEDIA_ROOT})
)
```

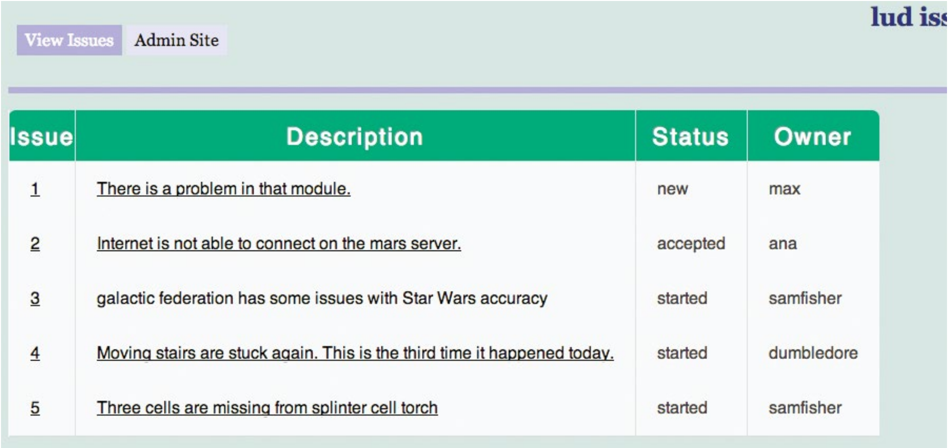
Creating the template files

10 Templates will be loaded from the ludIssueTracker/ludIssueTracker/templates directory. In Django, we start with the ludIssueTracker/ludIssueTracker/templates/base.html template. Think of it as the master template which can be inherited by slave ones.

ludIssueTracker/ludIssueTracker/templates/base.html

```
<!DOCTYPE html PUBLIC "-//
W3C//DTD XHTML Strict//EN"
" HYPERLINK "http://www.
w3.org/TR/xhtml1/DTD/xhtml1-
strict.dtd" http://www.w3.org/TR/
xhtml1/DTD/xhtml1-strict.dtd">
<html>
        <head>
        <title>{% block title
%}{% endblock %}LUD Issues</
title>
        <link rel="stylesheet"
href="{{ MEDIA_URL }}style.css"
type="text/css" media="screen"
/>
</head>
<body>
        <div id="hd">
                <h1>LUD
Issue Tracker</span></h1>
        </div>
        <div id="mn">
                <ul>

<li><a href="{% url issue-list
%}" class="sel">View Issues</
a></li>

<li><a href="/admin/">Admin
Site</a></li>
                </ul>
        </div>
        <div id="bd">
                {% block
content %}{% endblock %}
        </div>
</body>
</html>
```

"To display an issue list and details here, we are using a Django feature called generic views"



{{ variablename }} represents a Django variable.
(% block title %} represents blocks. Contents of a block are evaluated by Django and are displayed. These blocks can be replaced by the child templates.
Now we need to create the issue_list.html template. This template is responsible for displaying all the issues available in the system.
ludIssueTracker/ludIssueTracker/templates/ludissues/issue_list.html

```
{% extends ‘base.html’ %}
{% block title %}View Issues - {% endblock %}
{% block content %}
<table cellspacing=”0” class=”column-options”>
        <tr>
        <th>Issue</th>
        <th>Description</th>
        <th>Status</th>
        <th>Owner</th>
        </tr>
        {% for issue in object_list %}
        <tr>
        <td><a href=”{% url issue-detail issue.id %}”>{{ issue.id }}</a></td>
        <td><a href=”{% url issue-detail issue.id %}”>{{ issue.name }}</a></td>
        <td>{{ issue.status }}</td>
        <td>{{ issue.owner}}</td>
        </tr>
        {% endfor %}
</table>
{% endblock %}
```

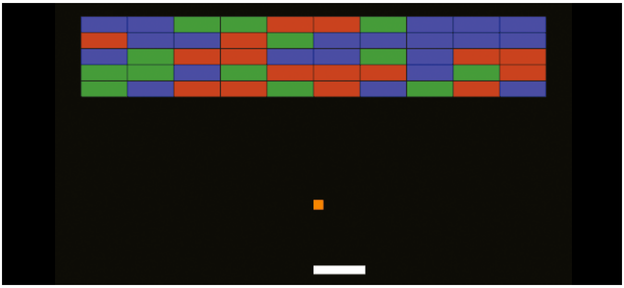
Here we are inheriting the base.html file that we created earlier. {% for issue in object_list %} runs on the object sent by the urls.py. Then we are iterating on the object_list for issue.id and issue.name.
Now we will create issue_detail.html. This template is responsible for displaying the detail view of a case.
ludIssueTracker/ludIssueTracker/templates/ludissues/issue_detail.html

```
{% extends ‘base.html’ %}
{% block title %}Issue #{{ object.id }} - {% endblock %}
{% block content %}
<h2>Issue #{{ object.id }} <span>{{ object.status }}</span></h2>
<div class=”issue”>
        <h2>Information</h2>
    <div class=”date”>
        <p class=”cr”>Opened {{ object.opened_on }} ago</p>
        <p class=”up”>Last modified {{ object.modified_on }} ago</p>
    </div>
        <div class=”clear”> </div>
        <div class=”block w49 right”>
        <p class=”ass title”>Owner</p>
        <p class=”ass”>{{ object.owner }}</p>
        </div>
        <div class=”clear”> </div>
    <div class=”block”>
        <p class=”des title”>Summary</p>
        <p class=”des”>{{ object.summary }}</p>
        </div>
</div>
{% endblock %}
```

And that’s everything! The issue tracker app is now complete and ready to use. You can now point your browser at localhost:8000 to start using the app.

Build an app for Android with Python

Master Kivy, the excellent cross-platform application framework to make your first Android app…



The great thing about Kivy is there are loads of directions we could take it in to do some pretty fancy things. But, we're going to make a beeline for one of Kivy's coolest features - the ability it affords you to easily run your programs on Android.

We'll approach this by first showing how to make a new app, this time a dynamic Breakout-style game. We'll then be able to compile this straight to an Android APK that you can use just like any other.

Of course, once you have mastered the basic techniques you aren't limited to using any particular kind of app, as even on Android you can make use of all your favourite Python libraries to make any sort of program you like.

Once you've mastered Kivy, your imagination is the only limit. If you're pretty new to Kivy, don't worry, we won't assume that you have any pre-existing knowledge. As long as you have mastered some of the Python in this book so far, and have a fairly good understanding of the language, you shouldn't have any problems following along with this.

Before anything else, let's throw together a basic Kivy app (Fig. 01). We've pre-imported the widget types we'll be using, which this time are just three: the basic Widget with no special behaviour, the ModalView with a pop-up behaviour as used last time, and the FloatLayout as we will explain later. Kivy has many other pre-built widgets for creating GUIs, but this time we're going to focus on drawing the whole GUI from scratch using Kivy's graphics instructions. These comprise either vertex instructions to create shapes (including rectangles, lines, meshes, and so on) or contextual graphics changes (such as translation, rotation, scaling, etc), and are able to be drawn anywhere on your screen and on any widget type.

Before we can do any of this we'll need a class for each kind of game object, which we're going to pre-populate with some of the properties that we'll need later to control them. Remember from last time, Kivy properties are special attributes declared at class level, which (among other things) can be modified via kv language and dispatch events when they are modified. The Game class will be one big widget containing the entire game. We've specifically made it a subclass of FloatLayout because this special layout is able to position and size its children in proportion to its own position and size – so no matter where we run it or how much we resize the window, it will place all the game objects appropriately.

Next we can use Kivy's graphics instructions to draw various shapes on our widgets. We'll just demonstrate simple rectangles to show their locations, though there are many more advanced options you might like to investigate. In a Python file we can apply any instruction by declaring it on the canvas of any widget, an example of which is shown in Fig. 03.

This would draw a red rectangle with the same position and size as the player at its moment of instantiation – but this presents a problem, unfortunately, because the drawing is static. When we later go on to move the player widget, the red rectangle will stay in the same place, while the widget will be invisible when it is in its real position.

We could fix this by keeping references to our canvas instructions and repeatedly updating their properties to track the player, but there's actually an easier way to do all of this - we can use the Kivy language we introduced last time. It has a special syntax for drawing on the widget canvas, which we can use here to draw each of our widget shapes:

```
<Player>:
  canvas:
    Color:
      rgba: 1, 1, 1, 1
    Rectangle:
      pos: self.pos
      size: self.size

<Ball>:
  canvas:
    Color:
      rgb: 1, 0.55, 0
    Rectangle:
      pos: self.pos
      size: self.size

<Block>:
  canvas:
    Color:
      rgb: self.colour
      # A property we
predefined above
    Rectangle:
      pos: self.pos
      size: self.size
    Color:
      rgb: 0.1, 0.1, 0.1
    Line:
      rectangle:
        [self.x, self.y,
        self.width, self.
height]
```

The canvas declaration is special, underneath it we can write any canvas instructions we like. Don't get confused, canvas is not a widget and nor are graphics instructions like Line. This is just a special syntax that is unique to the canvas. Instructions all have different properties that can be set, like the pos and size of the rectangle, and you can check the Kivy documentation online for all the different possibilities. The biggest advantage is that although we still declare simple canvas instructions, kv language is able to detect what Kivy properties we have referred to and automatically track them, so when they are updated (the widget moves or is resized) the canvas instructions move to follow this!

Fig 01

```
from kivy.app import App
from kivy.uix.widget import
Widget
from kivy.uix.floatlayout
import FloatLayout
from kivy.uix.modalview
import ModalView

__version__ = '0.1' #
Used later during Android
compilation

class BreakoutApp(App):
    pass

BreakoutApp().run()
```

Fig 02

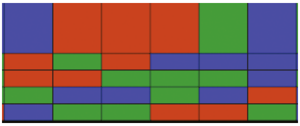
```
from kivy.properties
import (ListProperty,
NumericProperty,

ObjectProperty,
StringProperty)
```

Build an app for Android with Python

```
class Game(FloatLayout): # Will contain everything
    blocks = ListProperty([])
    player = ObjectProperty() # The game's Player instance
    ball = ObjectProperty() # The game's Ball instance

class Player(Widget): # A moving paddle
    position = NumericProperty(0.5)
    direction = StringProperty('none')

class Ball(Widget): # A bouncing ball
    # pos_hints are for proportional positioning, see below
    pos_hint_x = NumericProperty(0.5)
    pos_hint_y = NumericProperty(0.3)
    proper_size = NumericProperty(0.)
    velocity = ListProperty([0.1, 0.5])

class Block(Widget): # Each coloured block to destroy
    colour = ListProperty([1, 0, 0])
```

Fig 03

```
from kivy.graphics.context_instructions import Color
from kivy.graphics.vertex_instructions import Rectangle

class Player(Widget):
    def __init__(self, **kwargs):
        super(Player, self).__init__(**kwargs)
        with self.canvas:
            Color(1, 0, 0, 1) # r, g, b, a -> red
            Rectangle(pos=self.pos, size=self.size)
            # or without the with syntax, self.canvas.add(...)
```



Above Running the app shows our coloured blocks on the screen… but they all overlap! We can fix that easily

You probably noticed we had one of the Block's 'Color' instructions refer to its colour property. This means that we can change the property any time to update the colour of the block, or in this case to give each block a random colour (Fig. 04).

Now that each of our widgets has a graphical representation, let's now tell our Game where to place them, so that we can start up the app and actually see something there.

```
class Game(FloatLayout):
    def setup_blocks(self):
        for y_jump in range(5):
            for x_jump in range(10):
                block = Block(pos_hint={
                    'x': 0.05 + 0.09*x_jump,
                    'y': 0.05 + 0.09*y_jump})
                self.blocks.append(block)
                self.add_widget(block)
class BreakoutApp(App):
    def build(self):
        g = Game()
        g.setup_blocks()
        return g
```

Here we create the widgets we want then use add_widget to add them to the graphics tree. Our root widget on the screen is an instance of Game and every block is added to that to be displayed.

The only new thing in there is that every Block has been given a pos_hint. All widgets have this special property, and it is used by FloatLayouts like our Game to set their position proportionately to the layout.

The dictionary is able to handle various parameters, but in this case 'x'and 'y' give x and y Block position as a relative fraction of the parent width and height.

You can run the app now, and this time it will add 50 blocks to the Game before displaying it on the screen. Each should have one of the three possible random colours and be positioned in a grid, but you'll now notice their sizes haven't been manually set so they all overlap. We can fix this by setting their size_hint properties – and let's also take this opportunity to do the same for the other widgets as well (Fig. 05).

This takes care of keeping all our game widgets positioned and sized in proportion to the Game containing them. Notice that the Player and Ball use references to the properties we set earlier, so we'll be able to move them by just setting these properties and letting kv language automatically update their positions.

The Ball also uses an extra property to remain square rather than rectangular, just because the alternative would likely look a little bit odd.

We've now almost finished the basic graphics of our app! All that remains is to add a Ball and a Player widget to the Game.

```
<Game>:
    ball: the_ball
    player: the_player
    Ball:
        id: the_ball
    Player:
        id: the_player
```

You can run the game again now, and should be able to see all the graphics working properly. Nothing moves yet, but thanks to the FloatLayout everything should remain in proportion if you resize the game/window.

Now we just have to add the game mechanics. For a game like this you usually want to run some update function many times per second, updating the widget positions and carrying out game logic – in this case collisions with the ball (Fig. 06).

The Clock can schedule any function at any time, either once or repeatedly. A function scheduled at interval automatically receives the time since its last call (dt here), which we've passed through to the ball and player via the references we created in kv language. It's good practice to scale the update (eg ball distance moved) by this dt, so things remain stable even if something interrupts the clock and updates don't meet the regular 1/60s you want.

At this point we have also added the first steps toward handling keyboard input, by binding to the kivy Window to call a method of the Player every time a key is pressed. We can then finish off the Player class by adding this key handler along with touch/mouse input.

```
class Player(Widget):
    def on_touch_down(self, touch):
        self.direction = (
            'right' if touch.x > self.parent.center_x else 'left')

    def on_touch_up(self, touch):
        self.direction = 'none'

    def on_key_down(self, keypress, scancode, *args):
        if scancode == 275:
            self.direction = 'right'
        elif scancode == 276:
            self.direction = 'left'
        else:
            self.direction = 'none'

    def on_key_up(self, *args):
        self.direction = 'none'

    def update(self, dt):
        dir_dict = {'right': 1, 'left': -1, 'none': 0}
        self.position += (0.5 * dt * dir_dict[self.direction])
```

These on_touch_ functions are Kivy's general method for interacting with touch or mouse input, they are automatically called when the input is detected and you can do anything you like in response to the touches you receive. In this case we set the Player's direction property in response to either keyboard and touch/mouse input, and use this direction to move the Player when its update method is called. We can also add the right behaviour for the ball (Fig. 07).

This makes the ball bounce off every wall by forcing its velocity to point back into the Game, as well as bouncing from the player paddle – but with an extra kick just to let the ball speed change. It doesn't yet handle any interaction with the blocks or any win/lose conditions, but it does try to call Game.lose() if the ball hits the bottom of the player's screen, so let's now add in some game end code to handle all of this (Fig. 08). And then add the code in Fig. 09 to your 'breakout.kv 'file.

This should fully handle the loss or win, opening a pop-up with an appropriate message and providing a button to try again. Finally, we have to handle destroying blocks when the ball hits them (Fig. 10).

This fully covers these last conditions, checking collision via Kivy's built-in collide_widget method that compares their bounding boxes (pos and size). The bounce direction will depend on how far the ball has penetrated, as this will tell us how it first collided with the Block.

So there we have it, you can run the code to play your simple Breakout game. Obviously it's very simple right now, but hopefully you can see lots of different ways to add whatever extra behaviour you like – you could add different types of blocks and power-ups, a lives system, more sophisticated paddle/ball interaction, or even build a full game interface with a menu and settings screen as well.

We're just going to finish showing one cool thing that you can already do – compile your game for Android! Generally speaking you can take any Kivy app and turn it straight into an Android APK that will run on any of your Android devices. You can even access the normal Android API to access hardware or OS features such as vibration, sensors or native notifications.

We'll build for Android using the Buildozer tool, and a Kivy sister project wrapping other build tools to create packages on different systems. This takes care of downloading and running the Android build tools (SDK, NDK, etc) and Kivy's Python-for-Android tools that create the APK.

Fig 04

```
import random

    class Block(Widget):
        def __init__(self,
**kwargs):
            super(Block,
self).__init__(**kwargs)
            self.colour =
random.choice([
                (0.78, 0.28,
0), )0.28, 0.63, 0.28), )0.25,
0.28, 0.78)])
```

Fig 05

```
<Block>:
    size_hint: 0.09, 0.05
    # ... canvas part

<Player>:
    size_hint: 0.1, 0.025
        pos_hint: {'x': self.
position, 'y': 0.1}
    # ... canvas part

<Ball>:
     pos_hint: {'x': self.pos_
hint_x, 'y': self.pos_hint_y}
    size_hint: None, None
    proper_size:
            min(0.03*self.parent.
height, 0.03*self.parent.width)
        size: self.proper_size,
self.proper_size
    # ... canvas part
```

Fig 06

```
        from kivy.clock import
Clock
          from kivy.core.window
import Window
        from kivy.utils import
platform

    class Game(FloatLayout):
        def update(self, dt):
                    self.ball.
update(dt)  # Not defined yet
                  self.player.
update(dt)  # Not defined yet

              def start(self,
*args):
              Clock.schedule_
interval(self.update, 1./60.)

        def stop(self):
                        Clock.
unschedule(self.update)

        def reset(self):
            for block in
self.blocks:
                 self.remove_
widget(block)
            self.blocks = []
                  self.setup_
blocks()
            self.ball.velocity
= [random.random(), 0.5]
                  self.player.
position = 0.5

    class BreakoutApp(App):
        def build(self):
```

```
        g = Game()
        if platform() != 'android':
            Window.bind(on_key_down=g.player.on_key_down)
            Window.bind(on_key_up=g.player.on_key_up)
        g.reset()
        Clock.schedule_once(g.start, 0)
        return g
```

Fig 07

```
class Ball(Widget)
    def update(self, dt):
        self.pos_hint_x += self.velocity[0] * dt
        self.pos_hint_y += self.velocity[1] * dt
        if self.right > self.parent.right:    # Bounce from right
            self.velocity[0] = -1 * abs(self.velocity[0])
        if self.x < self.parent.x:    # Bounce from left
            self.velocity[0] = abs(self.velocity[0])
        if self.top > self.parent.top:    # Bounce from top
            self.velocity[1] = -1 * abs(self.velocity[1])
        if self.y < self.parent.y:    # Lose at bottom
            self.parent.lose()    # Not implemented yet
        self.bounce_from_player(self.parent.player)

    def bounce_from_player(self, player):
        if self.collide_widget(player):
            self.velocity[1] = abs(self.velocity[1])
            self.velocity[0] += (
                0.1
                * ((self.center_x - player.center_x) /
                player.width))
```

Fig 08

```
class GameEndPopup(ModalView):
    message = StringProperty()
    game = ObjectProperty()

class Game(Widget):
    def lose(self):
        self.stop()
        GameEndPopup(
            message='[color=#ff0000]You lose![/color]',
            game=self).open()

    def win(self):  # Not called yet, but we'll need it later
        self.stop()
        GameEndPopup(
            message='[color=#00ff00]You win![/color]',
            game=self).open()
```

Fig 09

```
<GameEndPopup>:
    size_hint: 0.8, 0.8
    auto_dismiss: False  # Don't close if player clicks outside
    BoxLayout:
        orientation: 'vertical'
        Label:
            text: root.message
            font_size: 60
            markup: True
            halign: 'center'
        Button:
            size_hint_y: None
            height: sp(80)
            text: 'Play again?'
            font_size: 60
            on_release: root.game.start(); root.dismiss()
```

Here you will be needing some basic dependencies, which can be installed with ease just by using your distro's normal repositories. The main ones to use are OpenJDK7, zlib, an up-to-date Cython, and Git. If you are using a 64-bit distro you will also be in need of 32-bit compatibility libraries for zlib, libstdc++, as well as libgcc. You can then go on and download and install Buildozer:

Putting your APK on the Play Store

Find out how to digitally sign a release APK and upload it to an app store of your choice

1 Build and sign a release APK

Begin by creating a personal digital key, then using it to sign a special release version of the APK. Run these commands, and follow the instructions.

```
## Create your personal
digital key           ##
You can choose your own
## keystore name, alias,
and passwords.
$ keytool -genkey -v
-keystore test- ↵release-
key.keystore \
   -alias test-alias
-keyalg RSA ↵

-keysize 2048 -validity
10000
## Compile your app in
release mode
$ buildozer android
release ↵
## Sign the APK with your
new key
$ jarsigner -verbose
-sigalg ↵
SHA1withRSA -digestalg
SHA1 \
   -keystore ./test-
release-key.keystore \
   ./bin/KivyBreakout-0.1-
release- ↵
unsigned.apk test-alias
## Align the APK zip file
$ ~/.buildozer/android/
platform/android- ↵sdk-21/
tools/zipalign -v 4 \
   ./bin/KivyBreakout-0.1-
release- ↵
unsigned.apk \
   ./bin/KivyBreakout-0.1-
release.apk
```

> "Check through the whole file just to see what's available, but most of the default settings will be fine"

```
git clone git://github.com/
kivy/buildozer
cd buildozer
sudo python2.7 setup.py
install
```

When you're done with that part you can then go on and navigate to your Kivy app, and you'll have to name the main code file 'main.py', this is the access point that the Android APK will expect. Then:

```
buildozer init
```

This creates a 'buildozer.spec' file, a settings file containing all the information that Buildozer needs to create your APK, from the name and version to the specific Android build options. We suggest that you check through the whole file just to see what's available but most of the default settings will be fine, the only thing we suggest changing is (Fig. 11).

There are various other options you will often want to set, but none are really all that vital right now, so you're able to immediately tell Buildozer to build your APK and get going!

```
buildozer android debug
```

This will take some time, so be patient and it will work out fine. When you first run it, it will download both the Android SDK and NDK, which are large (at least hundreds of megabytes) but vital to the build. It will also take time to build these and to compile the Python components of your APK. A lot of this only needs to be done once, as future builds will take a couple of minutes if you change the buildozer.spec, or just a few seconds if you've only changed your code.

The APK produced is a debug APK, and you can install and use it. There are extra steps if you want to digitally sign it so that it can be posted on the Play store. This isn't hard, and Buildozer can do some of the work, but check the documentation online for full details.

Assuming everything goes fine (it should!), your Android APK will be in a newly created 'bin' directory with the name 'KivyBreakout-0.1-debug.apk'. You can send it to your phone any way you like (eg email), though you may need to enable application installation from unknown sources in your Settings before you can install it.