NEW

# Python

## The Complete Manual

The essential handbook for Python users

Master Python today!

# 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 bookazine, 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 quadcopter drone and sending text messages from Raspberry Pi. Let's get coding!

# Python

## The Complete Manual

Imagine Publishing Ltd
Richmond House
33 Richmond Hill
Bournemouth
Dorset BH2 6EZ
☎ +44 (0) 1202 586200
Website: www.imagine-publishing.co.uk
Twitter: @Books_Imagine
Facebook: www.facebook.com/ImagineBookazines

Publishing Director
Aaron Asadi

Head of Design
Ross Andrews

Production Editor
Alex Hoskins

Senior Art Editor
Greg Whitaker

Designer
Perry Wardell-Wicks

Photographer
James Sheppard

**Printed by**
William Gibbons, 26 Planetary Road, Willenhall, West Midlands, WV13 3XT

Distributed in the UK, Eire & the Rest of the World by
Marketforce, 5 Churchill Place, Canary Wharf, London, E14 5HU
Tel 0203 787 9060 www.marketforce.co.uk

Distributed in Australia by
Network Services (a division of Bauer Media Group), Level 21 Civic Tower, 66-68 Goulburn Street, Sydney, New South Wales 2000, Australia, Tel +61 2 8667 5288



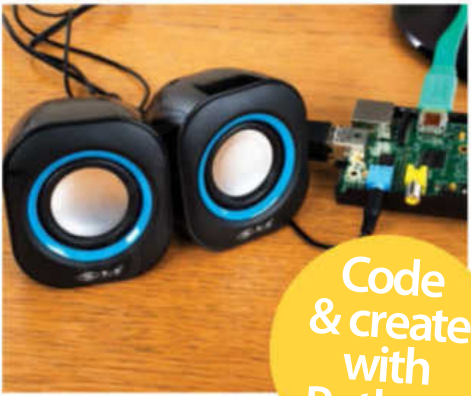
ISBN 978 1785 462 689

Part of the
LinuxUser & Developer
bookazine series

IMAGINE PUBLISHING

# Contents

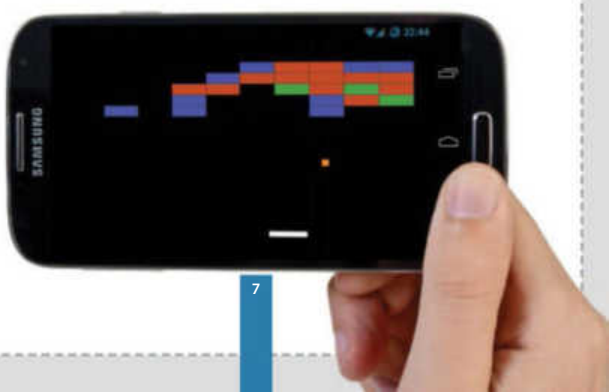
What you can find inside the bookazine



Code & create with Python!

# Get started with Python





# 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.

## 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

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

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

```
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"

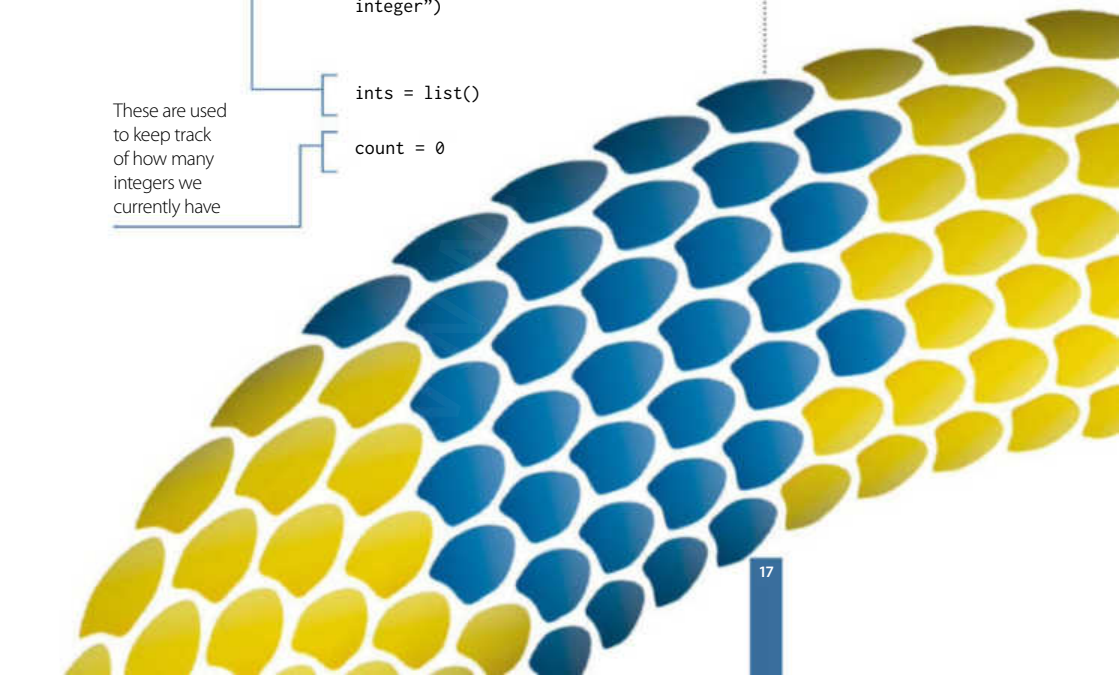
## Full code listing

```
#!/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

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")

ints = list()

count = 0
```

The number of integers we want in the list

A list to store the integers

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



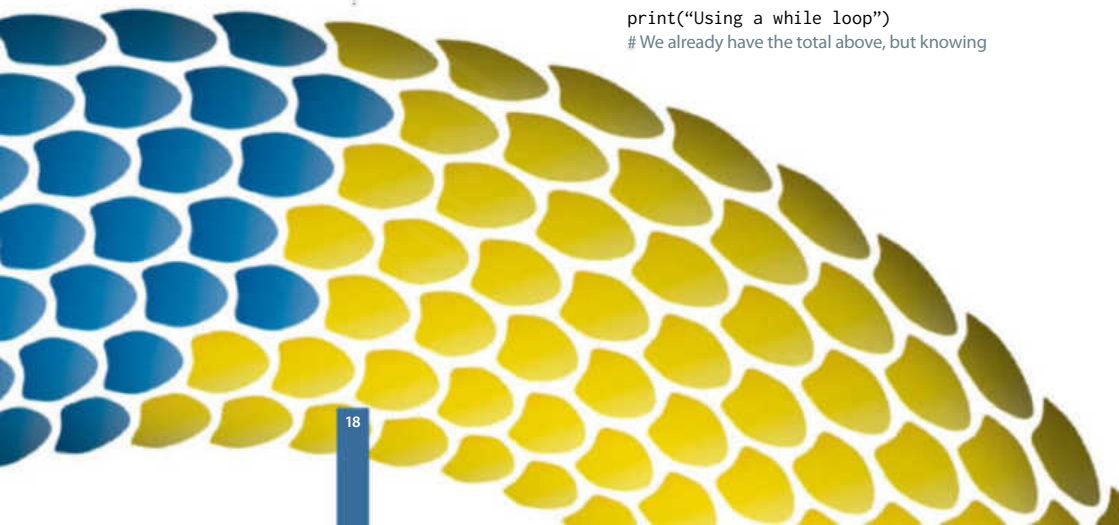
## Get started with Python

```
# 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)
        isint = True
    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
```

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

```
test_string = "This is a test string"
```

The test string won't be changed in this code

```
modify_string(test_string)
print(test_string)
```

```
test_string = modify_string_
return(test_string)
print(test_string)
```

However, we can call the function like this

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

```
[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.

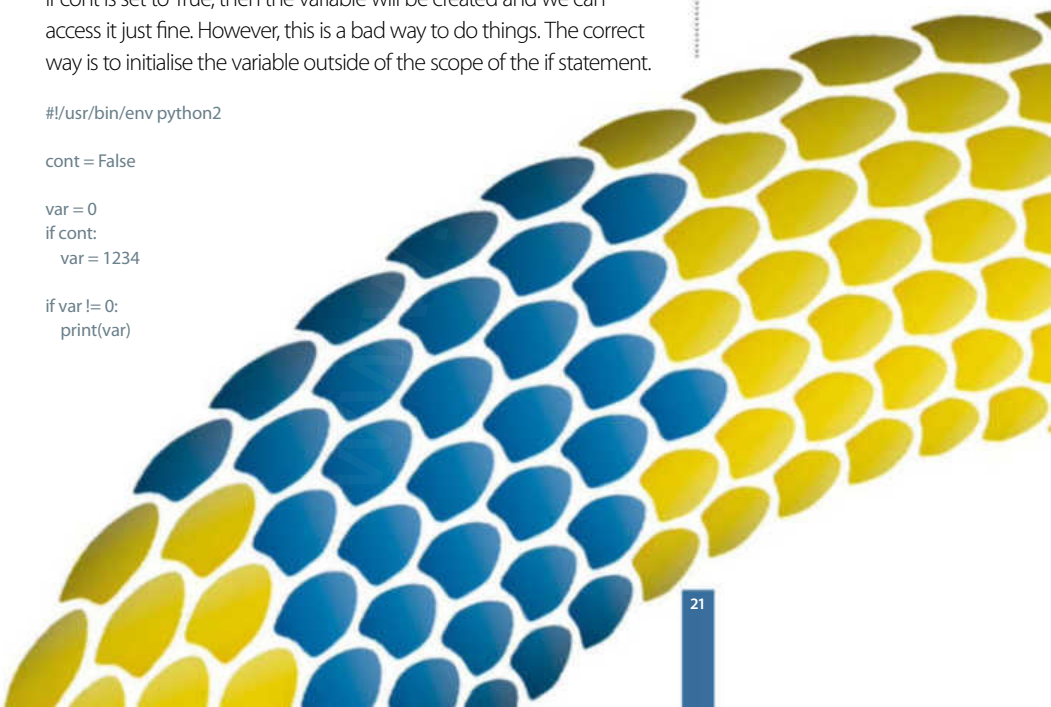
```
#!/usr/bin/env python2
cont = False
if cont:
  var = 1234
print(var)
```

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)
```



## 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:

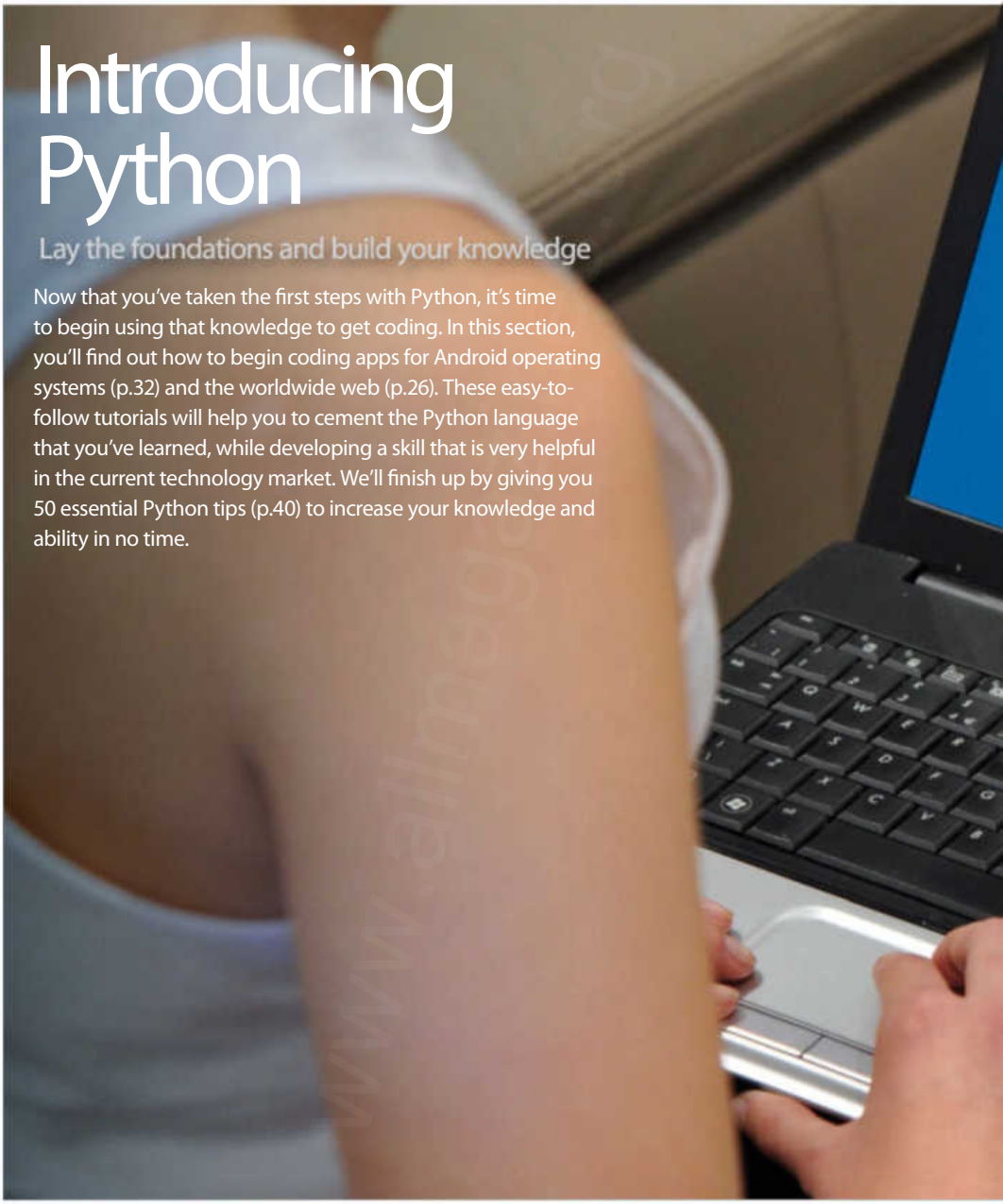
| | |
|---|---|
| < | strictly less than |
| <= | less than or equal |
| > | strictly greater than |
| >= | greater than or equal |
| == | equal |
| != | not equal |

## 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

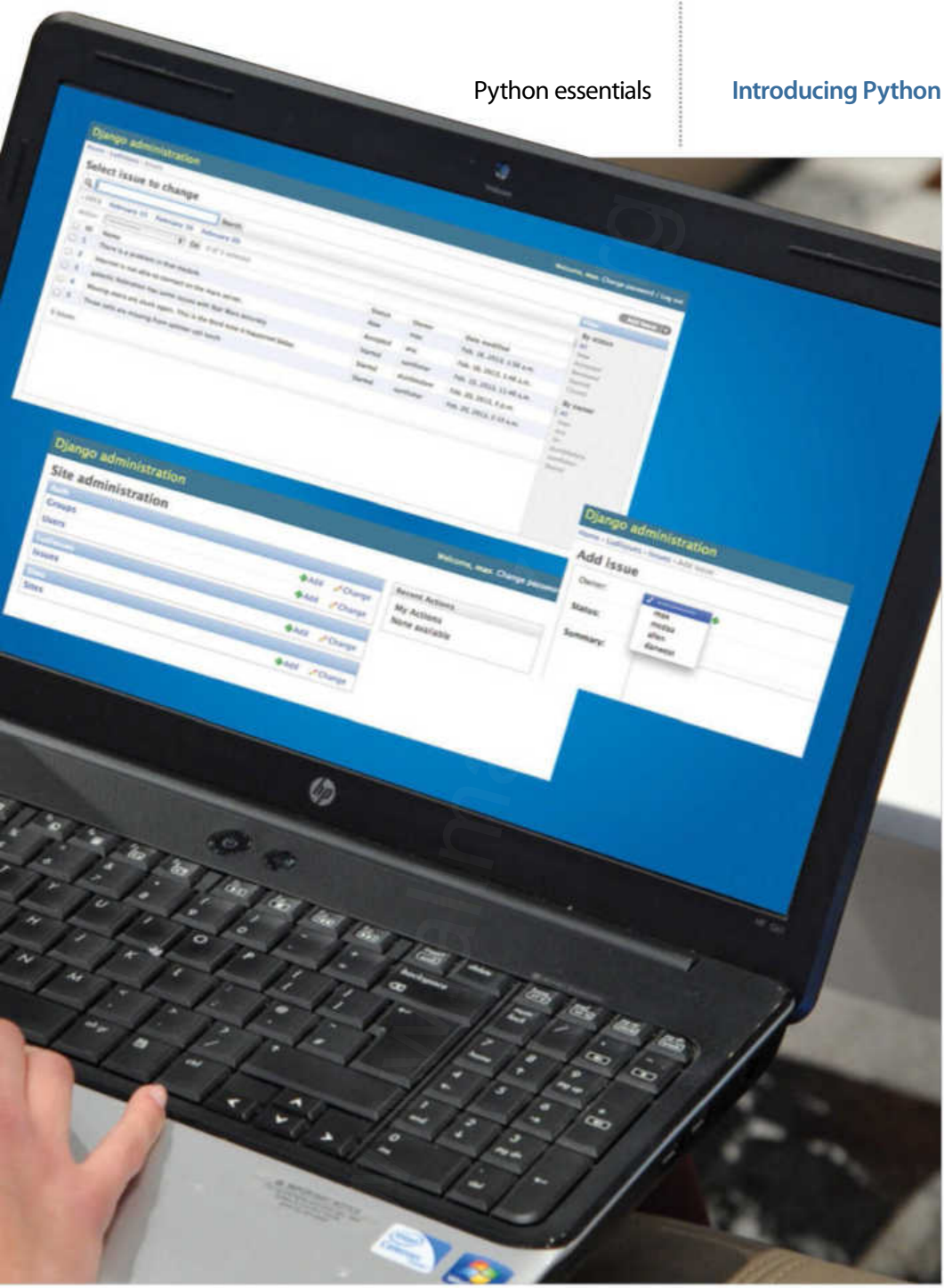
This article should have introduced you to the basics of programming in Python. Hopefully you are getting used to the syntax, indentation and general look and feel of a Python program. The next step is to learn how to come up with a problem that you want to solve, and break it down into small steps that 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.ForeignKey(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','name','status','owner','modified_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.

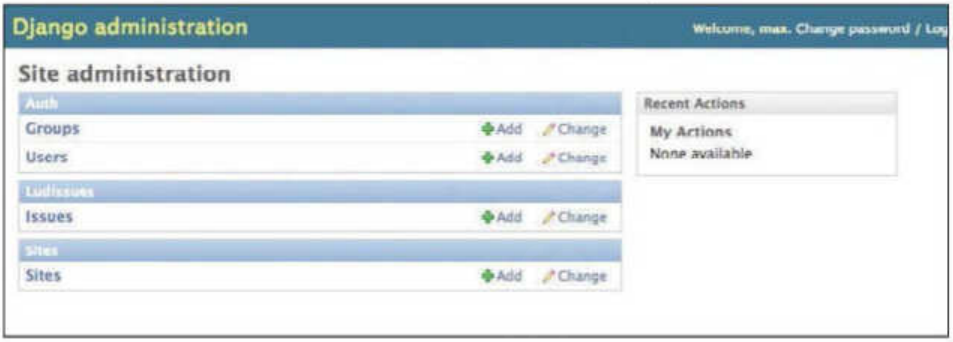

After logging in, you will notice that all the apps installed in your project are available here. We are only interested in the Auth and LudIssues app.

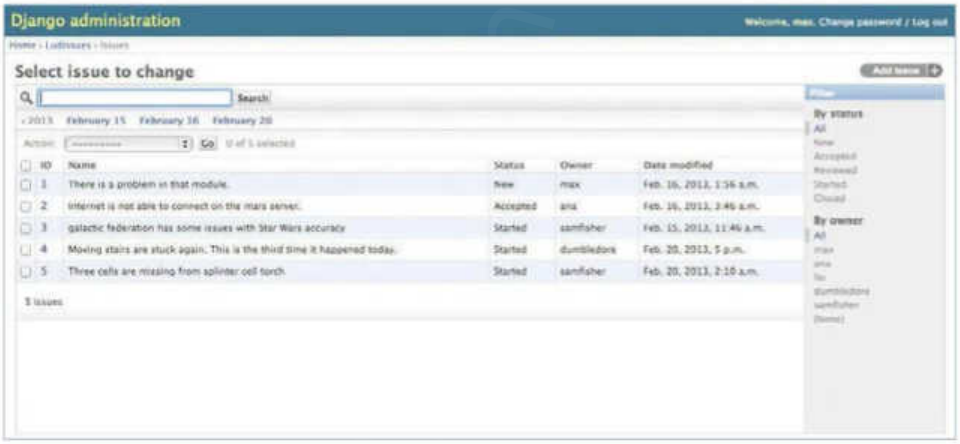
You can click the +Add to add a record. Click the Add button next to Users and add a few users to the site.

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 %}
```

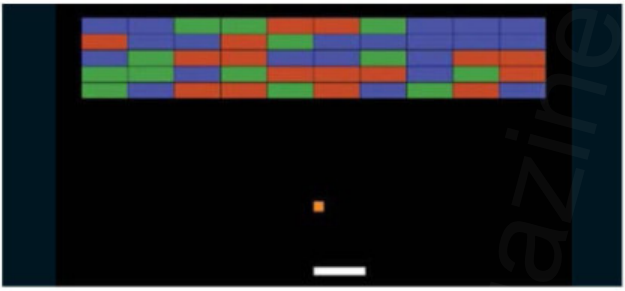
Here we are inheriting the base.html file that we created earlier. {% for issue in object_list %} runs on the object sent by the urls.py. Then we are iterating on the object_list for issue.id and issue.name.
Now we will create issue_detail.html. This template is responsible for displaying the detail 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()
```

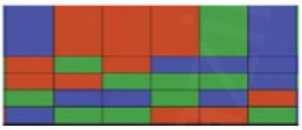
## Fig 02

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

ObjectProperty,
StringProperty)
```

```
    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.

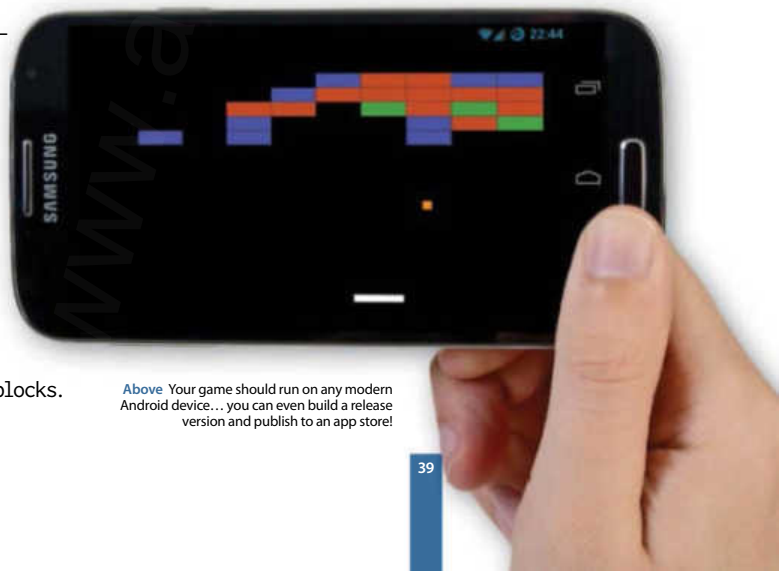
Assuming everything goes fine (it should!), your Android APK will be in a newly created 'bin' directory with the name 'KivyBreakout-0.1-debug.apk'. You can send it to your phone any way you like (eg email), though you may need to enable application installation from unknown sources in your Settings before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

## Fig 10

```
                self.parent.do_
layout()
            self.parent.destroy_
blocks(self)

class Game(FloatLayout):
    def destroy_blocks(self,
ball):
                for i, block in
enumerate(self.blocks):
                    if ball.
collide_widget(block):
                        y_overlap
= (
                            ball.
top - block.y if ball.
velocity[1] > 0
                            else
block.top - ball.y) / block.
size_hint_y
                        x_overlap
= (
                            ball.
right - block.x if ball.
velocity[0] > 0
                            else
block.right - ball.x) /
block.size_hint_x
                        if x_
overlap < y_overlap:

ball.velocity[0]    *=
-1

else:

ball.velocity[1]    *=
-1

                        self.
remove_widget(block)
                        self.blocks.
pop(i)
                        if len(self.
blocks) == 0:
                            self.
win()
                        return  #
Only remove at most 1 block
per frame
```

## Fig 11

```
title = Kivy Breakout    #
Displayed in your app drawer
package.name = breakout    #
Just a unique identifying
string,
                            #
along with the package.
domain
fullscreen = 0  # This will
mean the navbar is not
covered
log_level = 2  # Not vital,
but this will print a lot
more debug
                # information
and may be useful if
something
                # goes wrong
```




**Above** Your game should run on any modern Android device… you can even build a release version and publish to an app store!

### 2 Sign up as a Google Play Developer

Visit https://play.google.com/apps/publish/signup, and follow the instructions. You'll need to pay a one-off $25 charge, but then you can upload as many apps as you like.

### 3 Upload your app to the store

Click 'Add new application' to submit your app the store, including uploading your APK and adding description text. When everything is ready, simply click Publish, and it should take just a few hours for your app to go live!

# 50 Python tips

Python is a programming language that lets you work more quickly and integrate your systems more effectively. Today, Python is one of the most popular programming languages in the open source space. Look around and you will find it running everywhere, from various configuration tools to XML parsing. Here is the collection of 50 gems to make your Python experience worthwhile…

## Basics

### Running Python scripts

**01** On most of the UNIX systems, you can run Python scripts from the command line.

```
$ python mypyprog.py
```

### Running Python programs from Python interpreter

**02** The Python interactive interpreter makes it easy to try your first steps in programming and using all Python commands. You just issue each command at the command prompt (>>>), one by one, and the answer is immediate.

**Python interpreter can be started by issuing the command:**

```
$ python
kunal@ubuntu:~$ python
Python 2.6.2 (release26-maint, Apr 19 2009, 01:56:41)
[GCC 4.3.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> <type commands here>
```

**In this article, all the code starting at the >>> symbol is meant to be given at the Python prompt.**

It is also important to remember that Python takes tabs very seriously – so if you are receiving any error that mentions tabs, correct the tab spacing.

### Dynamic typing

**03** In Java, C++, and other statically typed languages, you must specify the data type of the function return value and each function argument. On the other hand, Python is a dynamically typed language. In Python you will never have to explicitly specify the data type of anything you use. Based on what value you assign, Python will automatically keep track of the data type internally.

### Python statements

**04** Python uses carriage returns to separate statements, and a colon and indentation to separate code blocks. Most of the compiled programming languages, such as C and C++, use semicolons to separate statements and curly brackets to separate code blocks.

### == and = operators

**05** Python uses '==' for comparison and '=' for assignment. Python does not support inline assignment, so there's no chance of accidentally assigning the value when you actually want to compare it.

### Concatenating strings

**06** You can use '+' to concatenate strings.

```
>>> print 'kun'+'al'
kunal
```

### The __init__ method

**07** The __init__ method is run as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object. The __init__ method is analogous to a constructor in C++, C# or Java.

**Example:**

```
class Person:
  def __init__(self, name):
    self.name = name
  def sayHi(self):
        print 'Hello, my name is', self.name
p = Person('Kunal')
p.sayHi()
```

**Output:**

```
[~/src/python $:] python initmethod.py
Hello, my name is Kunal
```

## Modules

**08** To keep your programs manageable as they grow in size you may want to make them into several files. Python allows you to put multiple function definitions into a file and use them as a module that can be imported. These files must have a .py extension however.

**Example:**

```
# file my_function.py
def minmax(a,b):
if a <= b:
min, max = a, b
else:
min, max = b, a
return min, max
Module Usage
import my_function
```

## Module defined names

**09** **Example:**
The built-in function 'dir()' can be used to find out which names a module defines. It returns a sorted list of strings.

```
>>> import time
>>> dir(time)
    ['__doc__', '__file__',
    '__name__', '__package__',
'accept2dyear', 'altzone',
'asctime', 'clock', 'ctime',
'daylight', 'gmtime', 'localtime',
'mktime', 'sleep', 'strftime',
'strptime', 'struct_time',
'time', 'timezone', 'tzname',
'tzset']file']
```

## Module internal documentation

**10** You can see the internal documentation (if available) of a module name by looking at .__doc__.

**Example:**

```
>>> import time
>>> print time.clock.__doc__
clock() -> floating
point number
```

This example returns the CPU time or real time since the start of the process or since the first call to clock(). This has just as much precision as the system records do.

## Passing arguments to a Python script

**11** Python lets you access whatever you have passed to a script while calling it. The 'command line' content is stored in the sys.argv list.

```
import sys
print sys.argv
```

## Loading modules or commands at startup

**12** You can load predefined modules or commands at the startup of any Python script by using the environment variable $PYTHONSTARTUP. You can set environment variable $PYTHONSTARTUP to a file which contains the instructions load necessary modules or commands .

## Converting a string to date object

**13** You can use the function 'DateTime' to convert a string to a date object.

**Example:**

```
from DateTime import DateTime
dateobj = DateTime(string)
```

## Converting a string to date object

**14** You can convert a list to string in the following ways.

**1st method:**

```
>>> mylist = ['spam', 'ham',
'eggs']
>>> print ', '.join(mylist)
spam, ham, eggs
```

**2nd method:**

```
>>> print '\n'.join(mylist)
spam
ham
eggs
```

## Tab completion in Python interpreter

**15** You can achieve auto completion inside Python interpreter by adding these lines to your .pythonrc file (or your file for Python to read on startup):

```
import rlcompleter, readline
readline.parse_and_bind('tab:
complete')
```

This will make Python complete partially typed function, method and variable names when you press the Tab key.

## Python documentation tool

**16** You can pop up a graphical interface for searching the Python documentation using the command:

```
$ pydoc -g
```

You will need python-tk package for this to work.

"Today, Python is certainly one of the most popular programming languages to be found in the open source space"

## Accessing the Python documentation server

**17** You can start an HTTP server on the given port on the local machine. This will give you a nice-looking access to all Python documentation, including third-party module documentation.

```
$ pydoc -p <portNumber>
```

## Python development software

**18** There are plenty of tools to help with Python development. Here are a few important ones:

**IDLE:** The Python built-in IDE, with autocompletion, function signature popup help, and file editing.

**IPython:** Another enhanced Python shell with tab-completion and other features.

**Eric3:** A GUI Python IDE with autocompletion, class browser, built-in shell and debugger.

**WingIDE:** Commercial Python IDE with free licence available to open-source developers everywhere.

# Built-in modules

## Executing at Python interpreter termination

**19** You can use 'atexit' module to execute functions at the time of Python interpreter termination.

**Example:**

```
def sum():
        print(4+5)
def message():
        print("Executing  Now")
import atexit
atexit.register(sum)
atexit.register(message)
```

**Output:**

```
Executing Now
9
```

## Converting from integer to binary and more

**20** Python provides easy-to-use functions – bin(), hex() and oct() – to convert from integer to binary, decimal and octal format respectively.

**Example:**

```
>>> bin(24)
'0b11000'
>>> hex(24)
'0x18'
>>> oct(24)
'030'
```

## Converting any charset to UTF-8

**21** You can use the following function to convert any charset to UTF-8.

```
data.decode("input_charset_
here").encode('utf-8')
```

## Removing duplicates from lists

**22** If you want to remove duplicates from a list, just put every element into a dict as a key (for example with 'none' as value) and then check dict.keys().

```
from operator import setitem
def distinct(l):
    d = {}
    map(setitem, (d,)*len(l),
l, [])
    return d.keys()
```

## Do-while loops

**23** Since Python has no do-while or do-until loop constructs (yet), you can use the following method to achieve similar results:

```
while True:
    do_something()
    if condition():
        break
```

## Detecting system platform

**24** To execute platform-specific functions, it is very useful to detect the platform on which the Python interpreter is running. You can use 'sys.platform' to find out the current platform.

**Example:**

**On Ubuntu Linux**

```
>>> import sys
>>> sys.platform
'linux2'
```

**On Mac OS X Snow Leopard**

```
>>> import sys
>>> sys.platform
'darwin'
```

## Disabling and enabling garbage collection

**25** Sometimes you may want to enable or disable the garbage collector function at runtime. You can use the 'gc' module to enable or disable the garbage collection.

**Example:**

```
>>> import gc
>>> gc.enable
<built-in function enable>
>>> gc.disable
<built-in function
disable>
```

## Using C-based modules for better performance

**26** Many Python modules ship with counterpart C modules. Using these C modules will give a significant performance boost in your complex applications.

**Example:**

```
cPickle instead of
Pickle, cStringIO instead
of StringIO .
```

## Calculating maximum, minimum and sum

**27** You can use the following built-in functions.
**max:** Returns the largest element in the list.
**min:** Returns the smallest element in the list.
**sum:** This function returns the sum of all elements in the list. It accepts an optional second argument: the value to start with when summing (defaults to 0).

## Representing fractional numbers

**28** Fraction instance can be created in Python using the following constructor:

```
Fraction([numerator
[,denominator]])
```

## Performing math operations

**29** The 'math' module provides a plethora of mathematical functions. These functions work on integer and float numbers, except complex numbers. For complex numbers, a separate module is used, called 'cmath'.
**For example:**

```
math.acos(x): Return arc
cosine of x.
math.cos(x): Returns cosine
of x.
math.factorial(x) : Returns x
factorial.
```

## Working with arrays

**30** The 'array' module provides an efficient way to use arrays in your programs. The 'array' module defines the following type:

```
array(typecode [,
initializer])
```

Once you have created an array object, say myarray, you can apply a bunch of methods to it. Here are a few important ones:

```
myarray.count(x): Returns the
number of occurrences of x
in a.
myarray.extend(x): Appends x
at the end of the array.
myarray.reverse(): Reverse the
order of the array.
```

## Sorting items

**31** The 'bisect' module makes it very easy to keep lists in any possible order. You can use the following functions to order lists.

```
bisect.insort(list, item [,
low [, high]])
```

Inserts item into list in sorted order. If item is already in the list, the new entry is inserted to the right of any existing entries there.

```
bisect.insort_left(list, item
[, low [, high]])
```

Inserts item into list in sorted order. If item is already within the list, the new entry is inserted to the left of any existing entries.

## Using regular expression-based search

**32** The 're' module makes it very easy to use regxp-based searches. You can use the function 're.search()' with a regexp-based expression. Check out the example included below.
**Example:**

```
>>> import re
>>> s = "Kunal is a bad boy"
>>> if re.search("K", s):
print "Match!" # char literal
...
Match!
>>> if re.search("[@A-Z]", s):
print "Match!"  # char class
... # match either at-sign or
capital letter
Match!
>>> if re.search("\d", s):
print "Match!" # digits class
...
```

## Working with bzip2 (.bz2) compression format

**33** You can use the module 'bz2' to read and write data using the bzip2 compression algorithm.

```
bz2.compress() :  For bz2
compression
bz2.decompress() : For bz2
decompression
```

**Example:**

```
# File: bz2-example.py
import bz2
MESSAGE = "Kunal is a bad
boy"
compressed_message = bz2.
compress(MESSAGE)
decompressed_message = bz2.
decompress(compressed_message)
print "original:",
repr(MESSAGE)
print "compressed message:",
repr(compressed_ message)
print "decompressed message:",
repr(decompressed_message)
```

**Output:**

```
[~/src/python $:] python bz2-
example.py
original: 'Kunal is a bad
boy'
compressed message:
'BZh91AY&SY\xc4\x0fG\x98\ x00\
x00\x02\x15\x80@\x00\x00\x084%\
x8a  \x00"\x00\x0c\x84\r\x03C\
xa2\xb0\xd6s\xa5\xb3\x19\x00\xf8\
xbb\x92)\xc2\x84\x86 z<\xc0'
decompressed message: 'Kunal
is a bad boy'
```

"There are tools to help develop with Python"

## Using SQLite database with Python

**34** SQLite is fast becoming a very popular embedded database because of the zero configuration that is needed, and its superior levels of performance. You can use the module 'sqlite3' in order to work with these SQLite databases.

**Example:**

```
>>> import sqlite3
>>> connection = sqlite.
connect('test.db')
>>> curs = connection.
cursor()
>>> curs.execute('''create
table item
... (id integer primary key,
itemno text unique,
... scancode text, descr text,
price real)''')
<sqlite3.Cursor object at
0x1004a2b30>
```

## Working with zip files

**35** You can use the module 'zipfile' to work with zip files.

`zipfile.ZipFile(filename [, mode [, compression [,allowZip64]]])`

Open a zip file, where the file can be either a path to a file (a string) or a file-like object.

`zipfile.close()¶`

Close the archive file. You must call 'close()' before exiting your program or essential records will not be written.

`zipfile.extract(member[, path[, pwd]])`

Extract a member from the archive to the current working directory; 'member' must be its full name (or a zipinfo object). Its file information is extracted as accurately as possible. 'path' specifies a different directory to extract to. 'member' can be a filename or a zipinfo object. 'pwd' is the password used for encrypted files.

## Using wildcards to search for filenames

**36** You can use the module 'glob' to find all the pathnames matching a pattern according to the rules used by the UNIX shell. *, ?, and character ranges expressed with [ ] will be matched.

**Example:**

```
>>> import glob
>>> glob.glob('./[0-9].*')
['./1.gif', './2.txt']
>>> glob.glob('*.gif')
['1.gif', 'card.gif']
>>> glob.glob('?.gif')
['1.gif']
```

## Performing basic file operations

**37** You can use the module 'shutil' to perform basic file operation at a high level. This module works with your regular files and so will not work with special files like named pipes, block devices, and so on.

`shutil.copy(src,dst)`

Copies the file src to the file or directory dst.

`shutil.copymode(src,dst)`

Copies the file permissions from src to dst.

`shutil.move(src,dst)`

Moves a file or directory to dst.

`shutil.copytree(src, dst, symlinks [,ignore]])`

Recursively copy an entire directory at src.

`shutil.rmtree(path [, ignore_errors [, onerror]])`

Deletes an entire directory.

## Executing UNIX commands from Python

**38** You can use module commands to execute UNIX commands. This is not available in Python 3 – instead, in this, you will need to use the module 'subprocess'.

**Example:**

```
>>> import commands
>>> commands.
getoutput('ls')
'bz2-example.py\ntest.py'
```

## Reading environment variables

**39** You can use the module 'os' to gather up some operating-system-specific information:

**Example:**

```
>>> import os
>>> os.path <module 'posixpath'
from '/usr/lib/python2.6/
posixpath.pyc'>>>>  os.environ
{'LANG': 'en_IN', 'TERM': 'xterm-
color', 'SHELL':
'/bin/bash', 'LESSCLOSE':
'/usr/bin/lesspipe %s %s',
'XDG_SESSION_COOKIE':
'925c4644597c791c704656354adf56d6-
1257673132.347986-1177792325',
'SHLVL': '1', 'SSH_TTY': '/dev/
pts/2', 'PWD': '/ home/kunal',
'LESSOPEN': '| /usr/bin
lesspipe
.....}
>>> os.name
'posix'
>>> os.linesep
'\n'
```

"Look around and you will find Python everywhere, from various configuration tools to XML parsing"

## Sending email

**40** You can use the module 'smtplib' to send email using an SMTP (Simple Mail Transfer Protocol) client interface.

```
smtplib.SMTP([host [, port]])
```

**Example (send an email using Google Mail SMTP server):**

```
import smtplib
# Use your own to and from email address
fromaddr = 'kunaldeo@gmail.com'
toaddrs  = 'toemail@gmail.com'
msg = 'I am a Python geek. Here is the  proof.!'
# Credentials
# Use your own Google Mail credentials  while running the program
username = 'kunaldeo@gmail.com'
password = 'xxxxxxxx'
# The actual mail send
server = smtplib.SMTP('smtp.gmail.com:587')
# Google Mail uses secure connection for  SMTP connections
server.starttls()
server.login(username,password)
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

## Accessing FTP server

**41** 'ftplib' is a fully fledged client FTP module for Python. To establish an FTP connection, you can use the following function:

```
smtplib.SMTP([host [, port]])
```

**Example (send an email using Google Mail SMTP server):**

```
ftplib.FTP([host [, user [, passwd [, acct  [, timeout]]]]])
```

**Example:**

```
host = "ftp.redhat.com"
username = "anonymous"
password = "kunaldeo@gmail.com"
import ftplib
import urllib2
ftp_serv = ftplib.FTP(host,username,password)
# Download the file
u = urllib2.urlopen ("ftp://ftp.redhat.com/ pub/redhat/linux/README")
# Print the file contents
print (u.read())
```

**Output:**

```
[~/src/python $:] python ftpclient.py
```

Older versions of Red Hat Linux have been moved to the following location: ftp://archive.download.redhat.com/pub/redhat/linux/

## Launching a webpage with the web browser

**42** The 'webbrowser' module provides a convenient way to launch webpages using the default web browser.

**Example (launch google.co.uk with system's default web browser):**

```
>>> import webbrowser
>>> webbrowser.open('http://google.co.uk')
True
```

## Creating secure hashes

**43** The 'hashlib' module supports a plethora of secure hash algorithms including SHA1, SHA224, SHA256, SHA384, SHA512 and MD5.

**Example (create hex digest of the given text):**

```
>>> import hashlib
# sha1 Digest
>>> hashlib.sha1("MI6 Classified Information  007").hexdigest()
'e224b1543f229cc0cb935a1eb959318ba1b20c85'
# sha224 Digest
>>> hashlib.sha224("MI6 Classified
Information 007").hexdigest()
'3d01e2f741000b0224084482f905e9b7b977a59b480990ea8355e2c0'
# sha256 Digest
>>> hashlib.sha256("MI6 Classified Information 007").hexdigest()
'2fdde5733f5d47b672fcb39725991c89b2550707cbf4c6403e fdb33b1c19825e'
# sha384 Digest
>>> hashlib.sha384("MI6 Classified Information 007").hexdigest()
'5c4914160f03dfbd19e14d3ec1e74bd8b99dc192edc138aaf7682800982488daaf540be9e0e50fc3d3a65c8b6353572d'
# sha512 Digest
>>> hashlib.sha512("MI6 Classified Information 007").hexdigest()
'a704ac3dbef6e8234578482a31d5ad29d252c822d1f4973f49b850222edcc0a29bb890778aea807a0a48ee4ff8bb18566140667fbaf73a1dc1ff192febc713d2'
# MD5 Digest
>>> hashlib.md5("MI6 Classified Information 007").hexdigest()
'8e2f1c52ac146f1a999a670c826f7126'
```

## Seeding random numbers

**44** You can use the module 'random' to generate a wide variety of random numbers. The most used one is 'random.seed([x])'. It initialises the basic random number generator. If x is omitted or None, the current system time is used; the current system time is also used to initialise the generator when the module is first imported.

"Programming in Python lets you work more quickly and integrate your systems much more effectively"

## Working with CSV files

**45** CSV files are very popular for data exchange over the web. Using the module 'csv', you can read and write CSV files.

**Example:**

```
import csv
# write stocks data as comma- separated values
writer = csv.writer(open('stocks.csv', 'wb', buffering=0))
writer.writerows([
('GOOG', 'Google, Inc.', 505.24, 0.47, 0.09),
('YHOO', 'Yahoo! Inc.', 27.38, 0.33, 1.22),
('CNET', 'CNET Networks, Inc.', 8.62, -0.13, -1.49)
])
# read stocks data, print status messages
stocks = csv.reader(open('stocks.csv', 'rb'))
status_labels = {-1: 'down', 0: 'unchanged', 1: 'up'}
for ticker, name, price, change, pct in stocks:
    status = status_labels[cmp(float(change), 0.0)]
    print '%s is %s (%s%%)' % (name, status, pct)
```

## Installing third-party modules using setuptools

**46** 'setuptools' is a Python package which lets you download, build, install, upgrade and uninstall packages very easily. To use the 'setuptools' package you will need to install these from your distribution's package manager.

After installation you can use the command 'easy_install' to perform any Python package management tasks that are necessary at that point.

**Example (installing simplejson using setuptools):**

```
kunal@ubuntu:~$ sudo easy_ install simplejson
Searching for simplejson
Reading http://pypi.python.org/simple/simplejson/
Reading http://undefined.org/python/#simplejson
Best match: simplejson 2.0.9
Downloading http://pypi.python.org/packages/source/s/simplejson/simplejson-2.0.9.tar.gz#md5=af5e67a39ca3408563411d357e6d5e47
Processing simplejson-2.0.9.tar.gz
Running simplejson-2.0.9/setup.py -q bdist_egg --dist-dir /tmp/easy_install-FiyfNL/simplejson-2.0.9/egg-dist-tmp-3YwsGV
Adding simplejson 2.0.9 to easy-install.pth file
Installed /usr/local/lib/python2.6/dist-packages/simplejson-2.0.9-py2.6-linux-i686.egg
Processing dependencies for simplejson
Finished processing dependencies for simplejson
```

> "You can use the module 'random' to generate a wide variety of random numbers with the basic generator"

## Logging to system log

**47** You can use the module 'syslog' to write to system log. 'syslog' acts as an interface to UNIX syslog library routines.

**Example:**

```
import syslog
syslog.syslog('mygeekapp: started logging')
for a in ['a', 'b', 'c']:
 b = 'mygeekapp: I found letter '+a
 syslog.syslog(b)
syslog.syslog('mygeekapp: the script goes to sleep now, bye,bye!')
```

**Output:**

```
$ python mylog.py
$ tail -f /var/log/messages
Nov  8 17:14:34 ubuntu -- MARK --
Nov  8 17:22:34 ubuntu python: mygeekapp: started logging
Nov  8 17:22:34 ubuntu python: mygeekapp: I found letter a
Nov  8 17:22:34 ubuntu python: mygeekapp: I found letter b
Nov  8 17:22:34 ubuntu python: mygeekapp: I found letter c
Nov  8 17:22:34 ubuntu python: mygeekapp: the script goes  to sleep now, bye,bye!
```

# Third-party modules

## Generating PDF documents

**48** 'ReportLab' is a very popular module for PDF generation from Python.

**Perform the following steps to install ReportLab**

```
$ wget http://www.reportlab.org/ftp/
```

```
ReportLab_2_3.tar.gz
  $ tar xvfz ReportLab_2_3.
tar.gz
  $ cd ReportLab_2_3
  $ sudo python setup.py
install
```

**For a successful installation, you should see a similar message:**

```
  ############SUMMARY
INFO############
  ##########################
#########
  #Attempting install of _rl_
accel, sgmlop &  pyHnj
  #extensions from '/home/
kunal/python/ ReportLab_2_3/
src/rl_addons/rl_accel'
  ##########################
#########
  #Attempting install of
_renderPM
  #extensions from '/home/
kunal/python/ ReportLab_2_3/
src/rl_addons/renderPM'
  # installing with freetype
version 21
  ##########################
#########
```

**Example:**

```
  >>> from reportlab.pdfgen.
canvas import Canvas
  # Select the canvas of
letter page size
  >>> from reportlab.lib.
pagesizes import letter
  >>> pdf = Canvas("bond.pdf",
pagesize = letter)
  # import units
  >>> from reportlab.lib.units
import cm, mm,  inch, pica
  >>> pdf.setFont("Courier",
60)
  >>> pdf.setFillColorRGB(1,
0, 0)
  >>> pdf.
drawCentredString(letter[0]
/ 2, inch *  6, "MI6
CLASSIFIED")
  >>> pdf.setFont("Courier",
40)
  >>> pdf.
drawCentredString(letter[0] /
2, inch *  5, "For 007's Eyes
Only")
  # Close the drawing for
current page
  >>> pdf.showPage()
  # Save the pdf page
  >>> pdf.save()
```

**Output:**

```
  @image:pdf.png
  @title: PDF Output
```

## Using Twitter API

**49** You can connect to Twitter easily using the 'Python-Twitter' module.

**Perform the following steps to install Python-Twitter:**

```
  $ wget  http://python-
twitter.googlecode.com/files/
python-twitter-0.6.tar.gz
  $ tar xvfz python-twitter*
  $ cd python-twitter*
  $ sudo python setup.py
install
```

**Example (fetching followers list):**

```
  >>> import twitter
# Use you own twitter account
here
  >>> mytwi = twitter.Api(us
ername='kunaldeo',password='x
xxxxx')
  >>> friends = mytwi.
GetFriends()
  >>> print [u.name for u in
friends]
  [u'Matt Legend Gemmell',
u'jono wells', u'The MDN
Big Blog', u'Manish Mandal',
u'iH8sn0w', u'IndianVideoGamer.
com', u'FakeAaron Hillegass',
u'ChaosCode', u'nileshp', u'Frank
Jennings',..']
```

## Doing Yahoo! news search

**50** You can use the Yahoo! search SDK to access Yahoo! search APIs from Python.

**Perform the following steps to install it:**

```
  $wget http://developer.
yahoo.com/download/files/
yws- 2.12.zip
  $ unzip yws*
  $ cd yws*/Python/
pYsearch*/
  $ sudo python setup.py
install
```

**Example:**

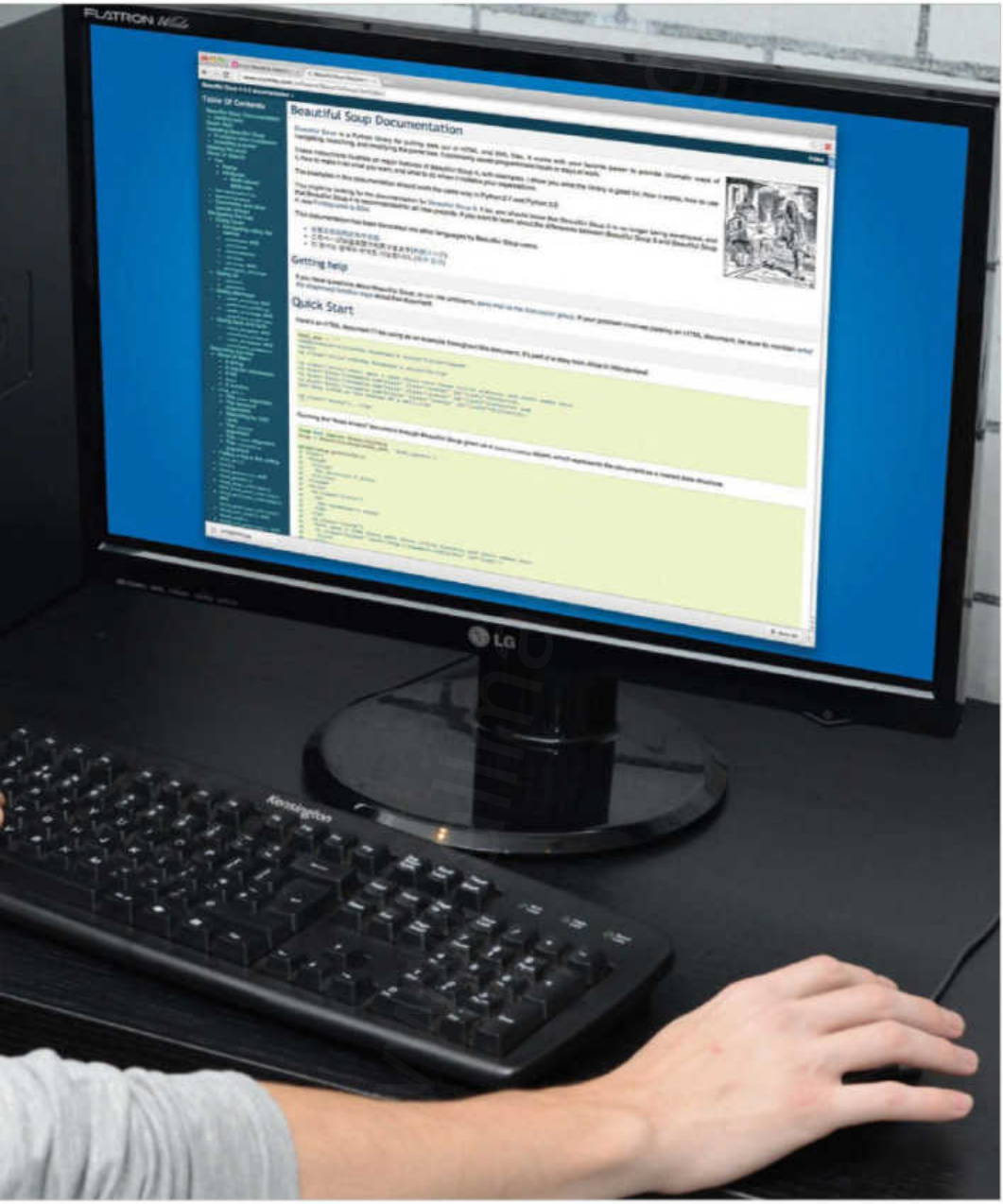
```
  # Importing news search
API
  >>> from yahoo.search.
news import NewsSearch
  >>> srch =
NewsSearch('YahooDemo',
query='London')
  # Fetch Results
  >>> info = srch.parse_
results()
  >>> info.total_results_
available
  41640
  >>> info.total_results_
returned
  10
  >>> for result in info.
results:
  ... print "'%s', from
%s" % (result['Title'],
result['NewsSource'])
...
  'Afghan Handover to
Be Planned at London
Conference,  Brown Says',
from Bloomberg
.....
```

"There are plenty of services such as IPython and IDLE available to users to help them with Python development"

# Work with Python

## Put the powerful programming language to work

With a more solid understanding of Python, you can really begin to make it work for you. It is a highly functional and versatile language, and in this section, we'll show you how to use this versatility in your own projects. First, we'll show you how to ditch the primary shell and replace it using Python (p.50), then look at how NumPy can help with scientific computing (p.58). We'll also look at how Python can help with system administration (p.64), and how you can use it with Beautiful Soup to read Wikipedia offline (p.72). Get ready to use Python to its full potential.


Beautiful Soup Documentation
Getting help
Quick Start
LG
Kensington

What you'll need…

**Python**
www.python.org/doc

# Replace your shell with Python

Python is a great programming language, but did you know it can even replace your primary shell?

We all use shell on a daily basis. For most of us, shell is the gateway into our Linux system. For years and even today, Bash has been the default shell for Linux. But it is getting a bit long in the tooth.

No need to be offended: we still believe Bash is the best shell out there when compared to some other UNIX shells such as Korn Shell (KSH), C Shell (CSH) or even TCSH.

This tutorial is not about Bash being incapable, but it is about how to breathe completely new life into the shell to do old things conveniently and new things which were previously not possible, even by a long shot. So, without further delay, let's jump in.

While the Python programming language may require you to write longer commands to accomplish a task (due to the way Python's modules are organised), this is not something to be particularly concerned about. You can easily write aliases to the equivalent of the Bash command that you intend to replace. Most of the time there will be more than one way to do a thing, but you will need to decide which way works best for you.

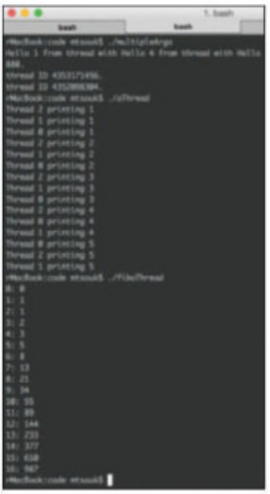
Python provides support for executing system commands directly (via the os or subprocess module), but where possible we will focus on Python-native implementations here, as this allows us to develop portable code.

## SECTION 1: Completing basic shell tasks in Python

### 1. File management

The Python module shutil provides support for file and directory operations. It provides support for file attributes, directory copying, archiving etc. Let's look at some of its important functions.

`shutil module`



Above You may never need to use Bash again, with some dedicated Python modules at hand

`copy (src,dst)`: Copy the src file to the destination directory. In this mode permissions bits are copied but metadata is not copied.
`copy2 (src,dst)`: Same as copy() but also copies the metadata.
`copytree(src, dst[, symlinks=False[, ignore=None]])`: This is similar to 'cp -r', it allows you to copy an entire directory.
`ignore_patterns (*patterns)`: ignore_patterns is an interesting function that can be used as a callable for copytree(), it allows you to ignore files and directories specified by the glob-style patterns.
`rmtree(path[, ignore_errors[, onerror]])`: rmtree() is used to delete an entire directory.
`move(src,dst)`: Similar to mv command it allows you to recessively move a file or directory to a new location.

Example:

```
from shutil import copytree, ignore_patterns
copytree(source, destination, ignore=ignore_patterns('*.pyc', 'tmp*'))
```

`make_archive(base_name, format[, root_dir[, base_dir[, verbose[, dry_run[, owner[, group[, logger]]]]]]]`: Think of this as a replacement for tar, zip, bzip etc. make_archive() creates an archive file in the given format such as zip, bztar, tar , gztar. Archive support can be extended via Python modules.

Example

```
from shutil import make_archive
import os
archive_name = os.path.expanduser(os.path.join('~', 'ludarchive'))
root_dir = os.path.expanduser(os.path.join('~', '.ssh'))
make_archive(archive_name, 'gztar', root_dir)
```

```
'/Users/kunal/ludarchive.tar.gz'
```

## 2. Interfacing operating system & subprocesses

Python provides two modules to interface with the OS and to manage processes, called os and subprocess. These modules let you interact with the core operating system shell, and work with the environment, processes, users and file descriptors. The subprocess module was introduced to support better management of subprocesses (paalready

## Replace your shell with Python

in Python and is aimed to replace os.system, os.spawn*, os.popen, popen2.* and commands.* modules.

os module
**environ**: environment represents the OS environment variables in a string object.

Example:

```
import os
os.environ
```

```
{'VERSIONER_PYTHON_PREFER_32_BIT': 'no', 'LC_CTYPE': 'UTF-
8', 'TERM_PROGRAM_VERSION': '297', 'LOGNAME': 'kunaldeo',
'USER': 'kunaldeo', 'PATH': '/System/Library/Frameworks/
Python.framework/Versions/2.7/bin:/Users/kunaldeo/narwhal/
bin:/opt/local/sbin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/
sbin:/usr/local/bin:/usr/X11/bin:/opt/local/bin:/Applications/
MOTODEV_Studio_For_Android_2.0.0_x86/android_sdk/tools:/
Applications/MOTODEV_Studio_For_Android_2.0.0_x86/android_sdk/
platform-tools:/Volumes/CyanogenModWorkspace/bin', 'HOME':
'/Users/kunaldeo', 'PS1': '\\[\\e[0;32m\\]\\u\\[\\e[m\\] \\
[\\e[1;34m\\]\\w\\[\\e[m\\] \\[\\e[1;32m\\]\\$\\[\\e[m\\] \\
[\\e[1;37m\\]', 'NARWHAL_ENGINE': 'jsc', 'DISPLAY': '/tmp/launch-
s2LUfa/org.x:0', 'TERM_PROGRAM': 'Apple_Terminal', 'TERM':
'xterm-color', 'Apple_PubSub_Socket_Render': '/tmp/launch-
kDul5P/Render', 'VERSIONER_PYTHON_VERSION': '2.7', 'SHLVL': '1',
'SECURITYSESSIONID': '186a5', 'ANDROID_SDK': '/Applications/
MOTODEV_Studio_For_Android_2.0.0_x86/android_sdk','_': '/System/
Library/Frameworks/Python.framework/Versions/2.7/bin/python',
'TERM_SESSION_ID': 'ACFE2492-BB5C-418E-8D4F-84E9CF63B506',
'SSH_AUTH_SOCK': '/tmp/launch-dj6Mk4/Listeners', 'SHELL': '/bin/
bash', 'TMPDIR': '/var/folders/6s/pgknm8b118737mb8psz8x4z80000
gn/T/', 'LSCOLORS': 'ExFxCxDxBxegedabagacad', 'CLICOLOR': '1',
'__CF_USER_TEXT_ENCODING': '0x1F5:0:0', 'PWD': '/Users/kunaldeo',
'COMMAND_MODE': 'unix2003'}
```

You can also find out the value for an environment value:

```
os.environ['HOME']
'/Users/kunaldeo'
```

`putenv(varname,value)` : Adds or sets an environment variable with the given variable name and value.
`getuid()` : Return the current process's user id.
`getlogin()` : Returns the username of currently logged in user
`getpid(pid)` : Returns the process group id of given pid. When used without any parameters it simply returns the current process id.
`getcwd()` : Return the path of the current working directory.
`chdir(path)` : Change the current working directory to the given path.
`listdir(path)` : Similar to ls, returns a list with the content of directories and file available on the given path.

Example:

```
os.listdir("/home/homer")
```

```
['.gnome2', '.pulse', '.gconf', '.gconfd', '.beagle',
'.gnome2_private', '.gksu.lock', 'Public', '.ICEauthority',
'.bash_history', '.compiz', '.gvfs', '.update-notifier',
'.cache', 'Desktop', 'Videos', '.profile', '.config', '.esd_
auth', '.viminfo', '.sudo_as_admin_successful', 'mbox',
'.xsession-errors', '.bashrc', 'Music', '.dbus', '.local',
'.gstreamer-0.10', 'Documents', '.gtk-bookmarks', 'Downloads',
'Pictures', '.pulse-cookie', '.nautilus', 'examples.desktop',
'Templates', '.bash_logout']
```

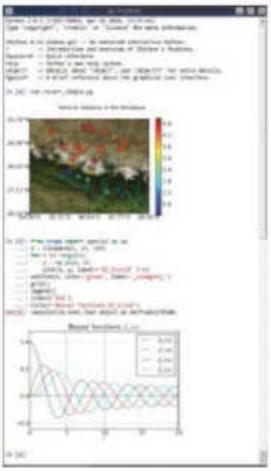
`mkdir(path[, mode])` : Creates a directory with the given path with the numeric code mode. The default mode is 0777.
`makedirs(path[, mode])` : Creates given path (inclusive of all its directories) recursively. The default mode is 0777. :

Example:

```
import os
path = "/home/kunal/greatdir"
os.makedirs( path, 0755 );
```

`rename (old,new)` : The file or directory "old" is renamed to "new" If "new" is a directory, an error will be raised. On Unix and Linux, if "new" exists and is a file, it will be replaced silently if the user has permission to do so.

`renames (old,new)` : Similar to rename but also creates any directories



Above A screenshot of the IPython Gt console with GUI capabilities

recessively if necessary.

`rmdir(path)` : Remove directory from the path mentioned. If the path already has files you will need to use shutil.rmdtree()

`subprocess:`

`call(*popenargs, **kwargs)` : Runs the command with arguments. On process completion it returns the returncode attribute.

Example:

```
import subprocess
print subprocess.call(["ls","-l"])
```

```
total 3684688
drwx------+   5 kunaldeo  staff     170 Aug 19 01:37 Desktop
drwx------+  10 kunaldeo  staff     340 Jul 26 08:30
                                     Documents
drwx------+  50 kunaldeo  staff    1700 Aug 19 12:50
                                     Downloads
drwx------@ 127 kunaldeo  staff    4318 Aug 19 01:43 Dropbox
drwx------@  42 kunaldeo  staff    1428 Aug 12 15:17 Library
drwx------@   3 kunaldeo  staff     102 Jul  3 23:23 Movies
drwx------+   4 kunaldeo  staff     136 Jul  6 08:32 Music
drwx------+   5 kunaldeo  staff     170 Aug 12 11:26 Pictures
drwxr-xr-x+   5 kunaldeo  staff     170 Jul  3 23:23 Public
-rwxr-xr-x    1 kunaldeo  staff  1886555648 Aug 16 21:02
                                     androidsdk.tar
drwxr-xr-x    5 kunaldeo  staff     170 Aug 16 21:05 sdk
drwxr-xr-x   19 kunaldeo  staff     646 Aug 19 01:47 src
-rw-r--r--    1 root      staff     367 Aug 16 20:36
                                     umbrella0.log
```

**Above** IPython previously offered a notebook feature, enabling users to create HTML documents where images, code and mathematical formulae were correctly formatted. This has since been split off into a separate (but tightly integrated) service called Jupyter

`STD_INPUT_HANDLE`: The standard input device. Initially, this is the console input buffer.
`STD_OUTPUT_HANDLE`: The standard output device. Initially, this is the active console screen buffer.
`STD_ERROR_HANDLE`: The standard error device. Initially, this is the active console screen buffer.

## SECTION 2: IPython: a ready-made Python system shell replacement

In section 1 we have introduced you to the Python modules which allow you to do system shell-related tasks very easily using vanilla Python. Using the same features, you can build a fully featured shell and remove a lot of Python boilerplate code along the way. However, if you are kind of person who wants everything ready-made, you are in luck. IPython provides a powerful and interactive Python shell which you can use as your primary shell. IPython supports Python 2.6 to 2.7 and 3.1 to 3.2 . It supports two type of Python shells: Terminal based and Qt based.

Just to reiterate, IPython is purely implemented in Python and provides a 100% Python-compliant shell interface, so everything that you have learnt in section 1 so far can be run inside IPython without any problems.

IPython is already available in most Linux distributions. Search your distro's repositories to look for it. In case you are not able to find it, you can also install it using easy_install or PyPI.

IPython provides a lot of interesting features which makes it a great shell replacement. . .

**Tab completion:** Tab completion provides an excellent way to explore any Python object that you are working with. It also helps you to avoid making typos.

**Example** :

```
In [3]: import o  {hit tab}
objc      opcode      operator      optparse      os      os2emxpath

In [3]: import os

In [4]: os.p    {hit tab}
os.pardir      os.pathconf_names      os.popen       os.popen4
os.path        os.pathsep             os.popen2      os.putenv
os.pathconf    os.pipe                os.popen3
```

**`Built In Object Explorer`**: You can add '?' after any Python object to view its details such as Type, Base Class, String Form, Namespace, File and Docstring.

## Replace your shell with Python

Example:

```
In [28]: os.path?
Type:        module
Base Class: <type 'module'>
String Form:<module 'posixpath' from '/System/Library/
Frameworks/Python.framework/Versions/2.7/lib/python2.7/
posixpath.pyc'>
Namespace:   Interactive
File:        /System/Library/Frameworks/Python.framework/
Versions/2.7/lib/python2.7/posixpath.py
Docstring:
Common operations on POSIX pathnames.
```

Instead of importing this module directly, import os and refer to this module as os.path. The 'os.path' name is an alias for this module on POSIX systems; on other systems (eg Mac, Windows), os.path provides the same operations in a manner specific to that platform, and is an alias to another module (eg macpath, ntpath).

Some of this can actually be useful on non-POSIX systems too, eg for manipulation of the pathname component of URLs. You can also use double question marks (??) to view the source code for the relevant object.

**Magic functions**: IPython comes with a set of predefined 'magic functions' that you can call with a command-line-style syntax. IPython 'magic' commands are conventionally prefaced by %, but if the flag %automagic is set to on, then you can call magic commands without the %. To view a list of available magic functions, use 'magic function %lsmagic'. They include functions that work with code such as %run, %edit, %macro, %recall etc; functions that affect shell such as %colors, %xmode, %autoindent etc; and others such as %reset, %timeit, %paste etc. Most cool features of IPython are powered using magic functions.

Example:

```
In [45]: %lsmagic
Available magic functions:
%alias  %autocall  %autoindent  %automagic  %bookmark  %cd
%colors  %cpaste  %debug  %dhist  %dirs  %doctest_mode  %ed
%edit  %env  %gui  %hist  %history  %install_default_config
%install_profiles  %load_ext  %loadpy  %logoff  %logon
%logstart  %logstate  %logstop  %lsmagic  %macro  %magic
```

```
%page  %paste  %pastebin  %pdb  %pdef  %pdoc  %pfile
%pinfo  %pinfo2  %popd  %pprint  %precision  %profile  %prun
%psearch  %psource  %pushd  %pwd  %pycat  %pylab  %quickref
%recall  %rehashx  %reload_ext  %rep  %rerun  %reset
%reset_selective  %run  %save  %sc  %sx  %tb  %time  %timeit
%unalias  %unload_ext  %who  %who_ls  %whos  %xdel  %xmode
```

Automagic is OFF, % prefix IS needed for magic functions. To view help on any Magic Function, call '%somemagic?' to read its docstring.

Python script execution and runtime code editing: You can use %run to run any Python script. You can also control-run the Python script with pdb debugger using -d, or pdn profiler using -p. You can also edit a Python script using the %edit command which opens the given Python script in the editor defined by the $EDITOR environment variable.

`Shell command support`: To just run a shell command, prefix the command with ! .

Example :

```
In [5]: !ps
  PID TTY          TIME    CMD
 4508 ttys000     0:00.07 -bash
84275 ttys001     0:00.03 -bash
17958 ttys002     0:00.18 -bash

In [8]: !clang prog.c -o prog
prog.c:2:1: warning: type specifier missing, defaults to
'int' [-Wimplicit-int]
main()
^~~~
1 warning generated.
```

`Qt console` : IPython comes with a Qt-based console. This provides features only available in a GUI, like inline figures, multiline editing with syntax highlighting, and graphical calltips. Start the Qt console with:

```
$ ipython qtconsole
```

If you get errors about missing modules, ensure that you have installed dependent packages – PyQt, pygments, pyexpect and ZeroMQ.

## Conclusion

As you can see, it's easy to tailor Python for all your shell environment needs. Python modules like os, subprocess and shutil are available at your disposal to do just about everything you need using Python. IPython turns this whole experience into an even more complete package. You get to do everything a standard Python shell does and with much more convenient features. IPython's magic functions really do provide a magical Python shell experience. So next time you open a Bash session, think again: why settle for gold when platinum is a step away?

## What you'll need…

**NumPy**
www.numpy.org

**SciPy**
www.scipy.org

**Matplotlib**
www.matplotlib.org

# Scientific computing with NumPy

Make some powerful calculations with NumPy, SciPy and Matplotlib

NumPy is the primary Python package for performing scientific computing. It has a powerful N-dimensional array object, tools for integrating C/C++ and Fortran code, linear algebra, Fourier transform, and random number capabilities, among other things. NumPy also supports broadcasting, which is a clever way for universal functions to deal in a meaningful way with inputs that do not have exactly the same form.

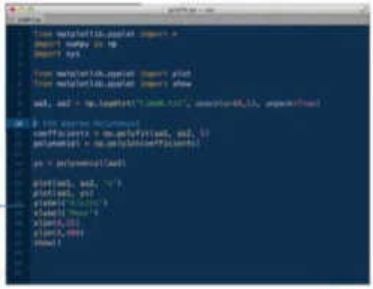
Apart from its capabilities, the other advantage of NumPy is that it can be integrated into Python programs. In other words, you may get your data from a database, the output of another program, an external file or an HTML page and then process it using NumPy.

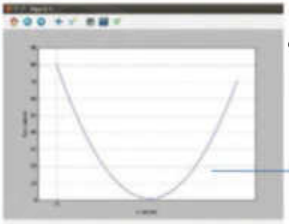
This article will show you how to install NumPy, make calculations, plot data, read and write external files, and it will introduce you to some Matplotlib and SciPy packages that work well with NumPy.

NumPy also works with Pygame, a Python package for creating games, though explaining its use is unfortunately beyond of the scope of this article.

It is considered good practice to try the various NumPy commands inside the Python shell before putting them into Python programs. The examples in this article use either Python shell or iPython.

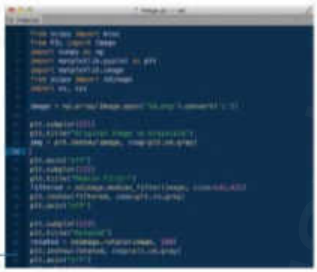
A simple Python program for Polynomial Fitting!

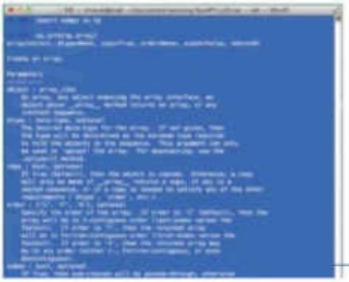




Matplotlib generated output

A Python script that uses SciPy to process an image





Finding help is easy

## Installing NumPy

**01** Most Linux distributions have a ready-to-install package you can use. After installation, you can find out the NumPy version you are using by executing the following:

```
$ python
Python 2.7.3 (default, Mar 13
2014, 11:03:55)
[GCC 4.7.2] on linux2
Type “help”, “copyright”,
“credits” or “license” for
more information.
>>> numpy.version.version
Traceback (most recent call
last):
  File “<stdin>”, line 1, in
<module>
NameError: name ‘numpy’ isnot
defined
>>> import numpy
>>> numpy.version.version
‘1.6.2’
>>>
```

Not only have you found the NumPy version but you also know that NumPy is properly installed.

## About NumPy

**02** Despite its simplistic name, NumPy is a powerful Python package that is mainly for working with arrays and matrices. There are many ways to create an array but the simplest one is to make use of the array() function:

```
>>> oneD = array([1,2,3,4])
```

The aforementioned command creates a one-dimensional array. If you want to create a two-dimensional array, you can use the array() function as follows:

```
>>> twoD = array([ [1,2,3],
...                 [3,3,3],
...                 [-1,-0.5,4],
...                 [0,1,0]] )
```

You can also create arrays with some more dimensions.

## Making simple calculations using NumPy

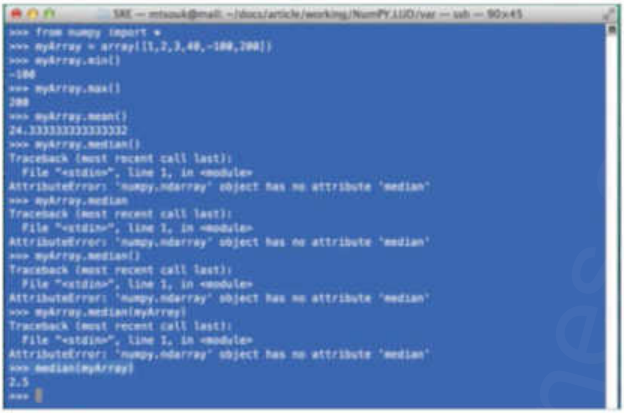
**03** Given an array named myArray, you can find the minimum and maximum values in it by executing the following commands:

```
>>> myArray.min()
>>> myArray.max()
```

Should you wish to find the mean value of all array elements, you can run the next command:

```
>>> myArray.mean()
```

Similarly, you can find the median of the




array by running the following Python command:

```
>>> median(myArray)
```

The median value of a set is an element that divides the data set into two subsets (left and right subsets) with the same number of elements. If the data set has an odd number of elements, then the median is part of the data set. On the other side, if the data set has an even number of elements, then the median is the mean value of the two centre elements of the sorted data set.

## Using arrays with NumPy

**04** NumPy not only embraces the indexing methods used in typical Python for strings and lists but also extends them. If you want to select a given element from an array, you can use the following notation:

```
>>> twoD[1,2]
```

You can also select a part of an array (a slice) using the following notation:

```
>>> twoD[:1,1:3]
```

Finally, you can convert an array into a Python list using the tolist() function.

## Reading files

**05** Imagine that you have just extracted information from an Apache log file using AWK and you now want to go and process the text file using NumPy.

The following AWK code finds out the total number of requests per hour:

```
$ cat access.log | cut -d[ -f2 | cut -d] -f1 | awk -F: '{print $2}' | sort -n | uniq -c | awk '{print $2, $1}' > timeN.txt
```

The format of the text file (timeN.txt) with the data is the following:

```
00 191
01 225
02 121
03 104
```

Reading the timeN.txt file and assigning it to a new array variable can be done as follows:

```
aa = np.loadtxt("timeN.txt")
```

## Writing to files

**06** Writing variables to a file is largely similar to reading

> "When you apply a function to an array, the function is automatically applied to all of the array elements"

a file. If you have an array variable named aa1, you can easily save its contents into a file called aa1.txt by using the following command:

```
In [17]: np.savetxt("aa1.txt", aa1)
```

As you can easily imagine, you can read the contents of aa1.txt later by using the loadtxt() function.

## Common functions

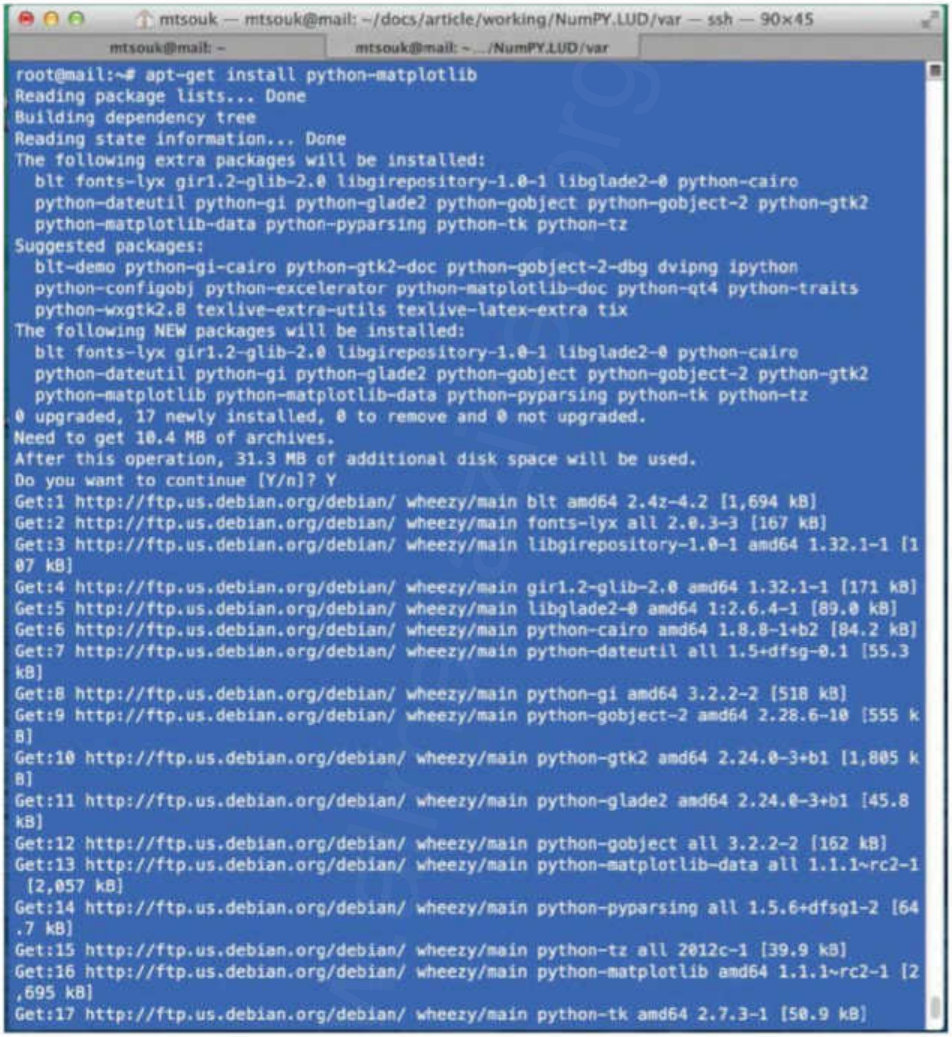
**07** NumPy supports many numerical and statistical functions. When you apply a function to an array, the function is then automatically applied to all of the array elements.

When working with matrices, you can find the inverse of a matrix AA by typing "AA.I". You can also find its eigenvalues by typing "np.linalg.eigvals(AA)" and its eigenvector by typing "np.linalg.eig(BB)".

## Working with matrices

**08** A special subtype of a two-dimensional NumPy array is a matrix. A matrix is like an array except that matrix multiplication replaces element-by-element multiplication. Matrices are generated using the matrix (or mat) function as follows:

```
In [2]: AA = np.mat('0 1 1; 1 1 1; 1 1 1')
```




You can add matrices named AA and BB by typing AA + BB. Similarly, you can multiply them by typing AA * BB.

## Plotting with Matplotlib

**09** The first move you should make is to install Matplotlib. As you can see, Matplotlib has many dependencies that you should also install. The first thing you will learn is how to plot a polynomial function. The necessary commands for plotting the 3x^2-x+1 polynomial are the following:

```
import numpy as np
import matplotlib.pyplot
```

"Try the various NumPy commands inside the Python shell"

```
as plt
myPoly = np.poly1d(np.
array([3, -1, 1]).
astype(float))
x = np.linspace(-5, 5, 100)
y = myPoly(x)
plt.xlabel('x values')
plt.ylabel('f(x) values')
xticks = np.arange(-5, 5, 10)
yticks = np.arange(0, 100,
10)
plt.xticks(xticks)
plt.yticks(yticks)
plt.grid(True)
plt.plot(x,y)
```

The variable that holds the polynomial is myPoly. The range of values that will be plotted for x is defined using "x = np.linspace(-5, 5, 100)". The other important variable is y, which calculates and holds the values of f(x) for each x value.

It is important that you start ipython using the "ipython --pylab=qt" parameters in order to see the output on your screen. If you are interested in plotting polynomial functions, you should experiment more, as NumPy can also calculate the derivatives of a function and plot multiple functions in the same output.

## About SciPy

**10** SciPy is built on top of NumPy and is more advanced than NumPy. It supports numerical integration, optimisations, signal processing, image and audio processing, and statistics.
The example below uses just one small part of the scipy.stats package about statistics.

> "For plotting polynomial functions, experiment more"

```
In [36]: from scipy.stats
import poisson, lognorm
In [37]: mySh = 10;
In [38]: myMu = 10;
In [39]: ln =
lognorm(mySh)
In [40]: p = poisson(myMu)
In [41]: ln.rvs((10,))
Out[41]:
array([  9.29393114e-
02,   1.15957068e+01,
9.78411983e+01,
          8.26370734e-
07,   5.64451441e-03,
4.61744055e-09,
          4.98471222e-
06,   1.45947948e+02,
9.25502852e-06,
          5.87353720e-02])
In [42]: p.rvs((10,))
Out[42]: array([12, 11,  9,
9,  9, 10,  9,  4, 13,  8])
In [43]: ln.pdf(3)
Out[43]:
0.013218067177522842
```

The example uses two statistics distributions and may be difficult to understand, but it is presented in order to give you a better taste of SciPy commands.

## Using SciPy for image processing

**11** Now we will show you how to process and transform a PNG image using SciPy. The most important part of the code is the following line:

```
image = np.array(Image.
open('SA.png').convert('L'))
```

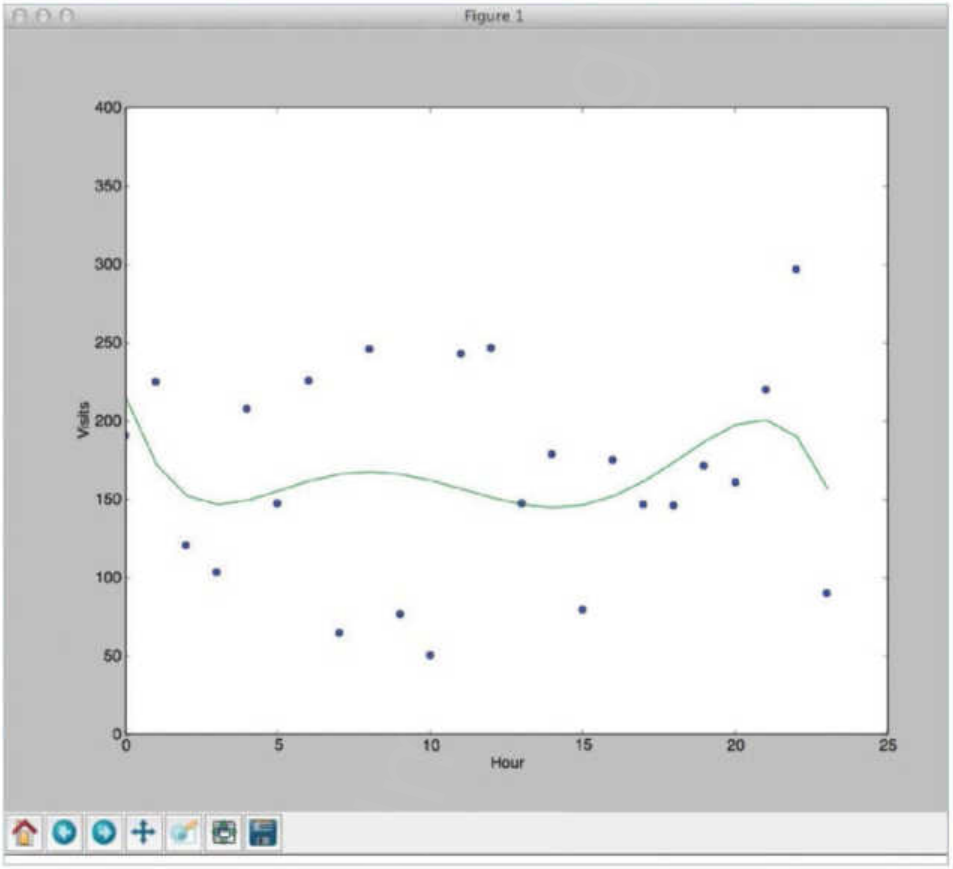
This line allows you to read a usual PNG file and convert it into a NumPy array for additional processing. The program will also separate the output into four parts and displays a different image for each of these four parts.

## Other useful functions

**12** It is very useful to be able to find out the data type of the elements in an array; it can be done using the dtype() function. Similarly, the ndim() function returns the number of dimensions of an array.

When reading data from external files, you can save their data columns into separate variables using the following method:

```
In [10]: aa1,aa2 =
```




**Above** Fitting to Polynomials

```
np.loadtxt("timeN.txt",
usecols=(0,1), unpack=True)
```

The aforementioned command saves column 1 into variable aa1 and column 2 into variable aa2. The "unpack=True" allows the data to be assigned to two different variables. Please note that the numbering of columns starts with 0.

## Fitting to polynomials

**13** The NumPy polyfit() function tries to fit a set of data points to a polynomial. The data was found from the timeN.txt file, created earlier.

The Python script uses a fifth degree polynomial, but if you want to use a different degree instead then you only have to change the following line:

```
coefficients = np.polyfit(aa1,
aa2, 5)
```

## Array broadcasting in NumPy

**14** To close, we will talk more about array broadcasting because it is a very useful characteristic. First, you should know that array broadcasting has a rule: in order for two arrays to be considered for array broadcasting, "the size of the trailing axes for both arrays in an operation must either be the same size or one of them must be one."

Put simply, array broadcasting allows NumPy to "change" the dimensions of an array by filling it with data in order to be able to do calculations with another array. Nevertheless, you cannot stretch both dimensions of an array to do your job.

# Python for system administrators

Learn how Python can help by daring to replace the usual shell scripting…

## What you'll need…

**Python-devel**
Python development libraries, required for compiling third-party Python module

**setuptools**
setuptools allows you to download, build, install, upgrade, and uninstall Python packages with ease

**Note**

This is written for the Python 2.X series, as it is still the most popular and default Python distribution across all the platforms (including all Linux distros, BSDs and Mac OS X).

System administration is an important part of our computing environment. It does not matter whether you are managing systems at your work our home. Linux, being a UNIX-based operating system, already has everything a system administrator needs, such as the world-class shells (not just one but many, including Bash, csh, zsh etc), handy tools, and many other features which make the Linux system an administrator's dream. So why do we need Python when Linux already has everything built-in? Being a dynamic scripting language, Python is very easy to read and learn. That's just not us saying that, but many Linux distributions actually use Python in core administrative parts. For example, Red Hat (and Fedora) system setup tool Anaconda is written in Python (read this line again, got the snake connection?). Also, tools like GNU Mailman, CompizConfig Settings Manager (CCSM) and hundreds of tiny GUI and non-GUI configuration tools are written using Python. Python does not limit you on the choice of user interface to follow – you can build command-line, GUI and web apps using Python. This way, it has got covered almost all the possible interfaces. Here we will look into executing sysadmin-related tasks using Python.

## Parsing configuration files

Configuration files provide a way for applications to store various settings. In order to write a script that allows you to modify settings of a particular application, you should be able to parse the configuration file of the application. In this section we learn how to parse INI-style configuration files. Although old, the INI file format is very popular with much modern open source software, such as PHP and MySQL.

Excerpt for php.ini configuration file:

```
[PHP]
engine = On
```

```
zend.ze1_compatibility_mode = Off
short_open_tag = On
asp_tags = Off
precision     =  14
y2k_compliance = On
output_buffering = 4096
;output_handler =
zlib.output_compression = Off

[MySQL]
; Allow or prevent persistent links.
mysql.allow_persistent = On
mysql.max_persistent = 20
mysql.max_links = -1
mysql.default_port = 3306
mysql.default_socket =
mysql.default_host = localhost
mysql.connect_timeout = 60
mysql.trace_mode = Off
```

Python provides a built-in module called ConfigParser (known as configparser in Python 3.0). You can use this module to parse and create configuration files.

**@code:** writeconfig.py
**@description:** The following demonstrates adding MySQL section to the php.ini file.
**@warning:** Do not use this script with the actual php.ini file, as it's not designed to handle all aspects of a complete php.ini file.

```
import ConfigParser
config = ConfigParser.RawConfigParser()

config.add_section('MySQL')
config.set('MySQL','mysql.trace_mode','Off')
config.set('MySQL','mysql.connect_
timeout','60')
config.set('MySQL','mysql.default_
host','localhost')
config.set('MySQL','mysql.default_
port','3306')
config.set('MySQL','mysql.allow_persistent',
'On' )
config.set('MySQL','mysql.max_
persistent','20')

with open('php.ini', 'ap') as configfile:
    config.write(configfile)
```

```
Output:php.ini
[MySQL]
mysql.max_persistent = 20
mysql.allow_persistent = On
mysql.default_port = 3306
mysql.default_host = localhost
mysql.trace_mode = Off
mysql.connect_timeout = 60
```

```
@code: parseconfig.py
@description: Parsing and updating the
config  file
import ConfigParser
config = ConfigParser.ConfigParser()
config.read('php.ini')
# Print config values
print config.get('MySQL','mysql.default_
host')
print config.get('MySQL','mysql.default_
port')
config.remove_option('MySQL','mysql.trace_
mode')
with open('php.ini', 'wb') as configfile:
    config.write(configfile)
```

## Parsing JSON data

JSON (also known as JavaScript Object Notation) is a lightweight modern data-interchange format. JSON is an open standard under ECMA-262. It is a text format

and is completely language-independent. JSON is also used as the configuration file format for modern applications such as Mozilla Firefox and Google Chrome. JSON is also very popular with modern web services such as Facebook, Twitter, Amazon EC2 etc. In this section we will use the Python module 'simplejson' to access Yahoo Search (using the Yahoo Web Services API), which outputs JSON data.

To use this section, you should have the following:

**1. Python module: simplejson.**

Note: You can install Python modules using the command 'easy_install <module name>'. This command assumes that you have a working internet connection.

**2. Yahoo App ID:**

The Yahoo App ID can be created from https://developer.apps.yahoo.com/dashboard/createKey.html. The Yahoo App ID will be generated on the next page. See the screenshot below for details.

simplejson is very easy to use. In the following example we will use the capability of mapping JSON data structures directly to Python data types. This gives us direct access to the JSON data without developing any XML parsing code.

**JSON PYTHON DATA MAPPING**

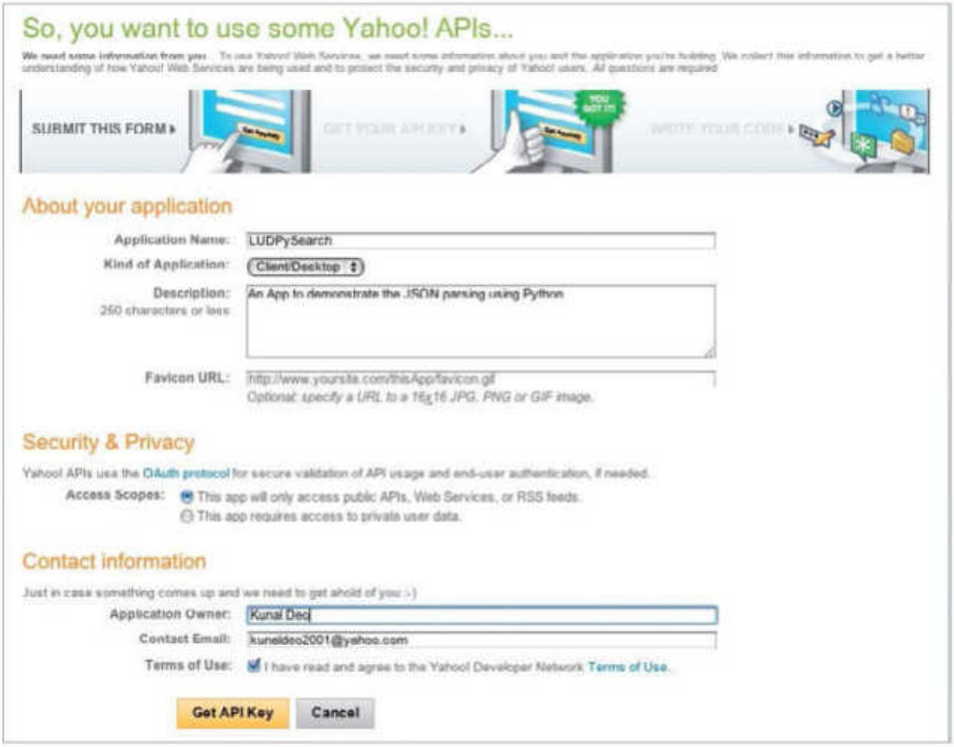
| JSON | Python |
|---|---|
| object | dict |
| array | list |
| string | unicode |
| number (int) | int, long |
| number (real) | float |
| TRUE | TRUE |
| FALSE | FALSE |
| null | None |

For this section we will use the simplejson.load function, which allows us to deserialise a JSON object into a Python object.

```
@code: LUDSearch.py
import simplejson, urllib
APP_ID = 'xxxxxxxx' # Change this to
your APP ID
SEARCH_BASE = 'http://search.yahooapis.
com/WebSearchService/V1/webSearch'

class YahooSearchError(Exception):
    pass

def search(query, results=20, start=1,
**kwargs):
    kwargs.update({
        'appid': APP_ID,
        'query': query,
        'results': results,
        'start': start,
        'output': 'json'
    })
    url = SEARCH_BASE + '?' + urllib.
urlencode(kwargs)
    result = simplejson.load(urllib.
urlopen(url))
    if 'Error' in result:
        # An error occurred; raise an
exception
        raise YahooSearchError,
result['Error']
    return result['ResultSet']
```

Let's use the code listed above from the Python shell to see how it works. Change to the directory where you have saved the LUDYSearch.py and open a Python shell.

```
@code: Python Shell Output. Lines
```




Above Generating the Yahoo App ID

```
starting with '>>>' indicate input
>>> execfile("LUDYSearch.py")
>>> results = search('Linux User and
Developer')
>>> results['totalResultsAvailable']
123000000
>>> results['totalResultsReturned']
20
>>> items = results['Result']
>>> for Result in items:
...       print Result['Title'],Result['Url']
...
```

Linux User http://www.linuxuser.co.uk/
Linux User and Developer - Wikipedia, the free encyclopedia http://en.wikipedia.org/wiki/Linux_User_and_Developer
Linux User & Developer | Linux User http://www.linuxuser.co.uk/tag/linux-user-developer/

## Gathering system information

An important job for a system administrator is gathering system information. Here we will use the SIGAR (System Information Gatherer And Reporter) API to demonstrate how we can gather system information using Python. SIGAR is a very complete API and can provide lots of information, including:

1. System memory, swap, CPU, load average, uptime, logins.

2. Per-process memory, CPU, credential info, state, arguments, environment, open files.
3. File system detection and metrics.
4. Network interface detection, configuration info and metrics.
5. TCP and UDP connection tables.
6. Network route table.

## Installing SIGAR

The first step is to build and install SIGAR. SIGAR is hosted at GitHub, so make sure that you have Git installed in your system. Then perform the following steps to install SIGAR and its Python bindings:

```
$ git clone git://github.com/hyperic/sigar.git sigar.git
$ cd sigar.git/bindings/python
$ sudo python setup.py install
```

At the end you should see a output similar to the following :

Writing /usr/local/lib/python2.6/dist-packages/pysigar-0.1.egg-info

SIGAR is a very easy-to-use library and can be used to get information on almost every aspect of a system. The next example shows you how to do this. The following code shows the memory and the file system information.

```
@code: PySysInfo.py
import os
import sigar
sg = sigar.open()
mem = sg.mem()
swap = sg.swap()
fslist = sg.file_system_list()
print "==========Memory Information============="
print "\tTotal\tUsed\tFree"
print "Mem:\t",\
      (mem.total() / 1024), \
      (mem.used() / 1024), \
      (mem.free() / 1024)
print "Swap:\t", \
      (swap.total() / 1024), \
      (swap.used() / 1024), \
      (swap.free() / 1024)
print "RAM:\t", mem.ram(), "MB"
print "==========File System Information============="
def format_size(size):
      return sigar.format_size(size * 1024)
print 'Filesystem\tSize\tUsed\tAvail\tUse%\tMounted on\tType\n'
for fs in fslist:
      dir_name = fs.dir_name()
      usage = sg.file_system_usage(dir_name)
      total = usage.total()
      used = total - usage.free()
      avail = usage.avail()
      pct = usage.use_percent() * 100
      if pct == 0.0:
            pct = '-'
      print fs.dev_name(), format_size(total), format_size(used), format_size(avail),\
            pct, dir_name, fs.sys_type_name(), '/', fs.type_name()
@Output
==========Memory Information==============
        Total     Used      Free
Mem:    8388608 6061884 2326724
Swap:   131072  16048   115024
RAM:    8192 MB
==========File System Information============
Filesystem      Size      Used      Avail
Use%    Mounted on      Type
```

```
/dev/disk0s2 300G 175G 124G 59.0 / hfs /
local
devfs 191K 191K   0   - /dev devfs /
none
```

## Accessing Secure Shell (SSH) services

SSH (Secure Shell) is a modern replacement for an old remote shell system called Telnet. It allows data to be exchanged using a secure channel between two networked devices. System administrators frequently use SSH to administrate networked systems. In addition to providing remote shell, SSH is also used for secure file transfer (using SSH File Transfer Protocol, or SFTP) and remote X server forwarding (allows you to use SSH clients as X server). In this section we will learn how to use the SSH protocol from Python using a Python module called paramiko, which implements the SSH2 protocol for Python. paramiko can be installed using the following steps:

```
$ git clone https://github.com/robey/
paramiko.git
$ cd paramiko
$ sudo python setup.py install
```

To the core of paramiko is the SSHClient class. This class wraps L{Transport}, L{Channel}, and L{SFTPClient} to handle most of the aspects of SSH. You can use SSHClient as:

```
client = SSHClient()
client.load_system_host_keys()
client.connect('some.host.com')
stdin, stdout, stderr = client.exec_
command('dir')
```

The following example demonstrates a full SSH client written using the paramiko module.

```
@code: PySSHClient.py
```

```
import base64, getpass, os, socket, sys,
socket, traceback
import paramiko
import interactive
# setup logging
paramiko.util.log_to_file('demo_simple.
log')
# get hostname
username = ''
if len(sys.argv) > 1:
    hostname = sys.argv[1]
    if hostname.find('@') >= 0:
        username, hostname = hostname.
split('@')
else:
    hostname = raw_input('Hostname: ')
if len(hostname) == 0:
    print '*** Hostname required.'
    sys.exit(1)
port = 22
if hostname.find(':') >= 0:
    hostname, portstr = hostname.
split(':')
    port = int(portstr)
# get username
if username == '':
    default_username = getpass.getuser()
    username = raw_input('Username [%s]:
' % default_username)
    if len(username) == 0:
        username = default_username
password = getpass.getpass('Password for
%s@%s: ' % (username, hostname))
# now, connect and use paramiko Client
to negotiate SSH2 across the connection
try:
    client = paramiko.SSHClient()
    client.load_system_host_keys()
    client.set_missing_host_key_
policy(paramiko.WarningPolicy)
```

```
    print '*** Connecting...'
    client.connect(hostname, port,
username, password)
    chan = client.invoke_shell()
    print repr(client.get_transport())
    print '*** SSH Server Connected!
***'
    print
    interactive.interactive_shell(chan)
    chan.close()
    client.close()
except Exception, e:
    print '*** Caught exception: %s:
%s' % (e.__class__, e)
    traceback.print_exc()
    try:
        client.close()
    except:
        pass
    sys.exit(1)
```

To run this code you will also need a custom Python class interactive.py which implements the interactive shell for the SSH session. Look for this file on FileSilo and copy it into the same folder where you have created PySSHClient.py .

```
@code_Output
kunal@ubuntu-vm-kdeo:~/src/paramiko/
demos$ python demo_simple.py
Hostname: 192.168.1.2
Username [kunal]: luduser
Password for luduser@192.168.1.2:
*** Connecting...
<paramiko.Transport at 0xb76201acL
(cipher aes128-ctr, 128 bits) (active; 1
open channel(s))>
*** SSH Server Connected! ***
Last login: Thu Jan 13 02:01:06 2011
from 192.168.1.9
[~ $:]
```

If the host key for the SSH server is not added to your $HOME/ssh/known_hosts file, the client will throw the following error:

```
*** Caught exception: <type 'exceptions.
TypeError'>: unbound method missing_
host_key() must be called with
WarningPolicy instance as first
argument (got SSHClient instance
instead)
```

This means that the client cannot verify the authenticity of the server you are connected to. To add the host key to known_hosts, you can use the ssh command. It is important to remember that this is not the ideal way to add the host key; instead you should use ssh-keygen. But for simplicity's sake we are using the ssh client.

```
kunal@ubuntu-vm-kdeo:~/.ssh$ ssh
luduser@192.168.1.2
The authenticity of host '192.168.1.2
(192.168.1.2)' can't be established.
RSA key fingerprint is be:01:76:6a:b9:bb:6
9:64:e3:dc:37:00:a4:36:33:d1.
Are you sure you want to continue
connecting (yes/no)? yes
Warning: Permanently added '192.168.1.2'
(RSA) to the list of known hosts.
```

So now you've seen just how easy it can be to carry out the complex sysadmin tasks using Python's versatile language.

As is the case with all Python coding, the code that is presented here can fairly easily be adopted into your GUI application (using software such as PyGTK or PyQt) or a web application (using a framework such as Django or Grok).

## Writing a user interface using Python

Administrators are comfortable with running raw scripts by hand, but end-users are not. So if you are writing a script that is supposed to be used by common users, it is a good idea to create a user-friendly interface on top of the script. This way end-users can run the scripts just like any other application. To demonstrate this, we will create a simple GRUB configuration tool which allows users to select default boot entry and the timeout. We will be creating a TUI (text user interface) application and will use the Python module 'snack' to facilitate this (not to be confused with the Python audio library, tksnack).

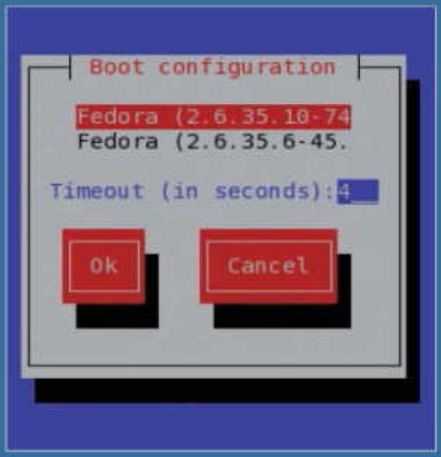
This app consists of two files…

**grub.py**: GRUB Config File (grub.conf) Parser (available on FileSilo). It implements two main functions, readBootDB() and writeBootFile(), which are responsible for reading and writing the GRUB configuration file.
**grub_tui.py**: Text user interface file for manipulating the GRUB configuration file using the functions available in grub.py.

```
@code:grub_tui.py
import sys
from snack import *

from grub import (readBootDB, writeBootFile)

def main(entry_value='1',kernels=[]):
    try:
        (default_value, entry_value,
kernels)=readBootDB()
    except:
        print >> sys.stderr, ("Error reading /boot/
grub/grub.conf.")
sys.exit(10)

    screen=SnackScreen()

    while True:
        g=GridForm(screen, ("Boot configuration"),1,5)
        if len(kernels)>0 :
            li=Listbox(height=len(kernels), width=20,
returnExit=1)
            for i, x in enumerate(kernels):
                li.append(x,i)
            g.add(li, 0, 0)
            li.setCurrent(default_value)

        bb = ButtonBar(screen, ((("Ok"), "ok"),
(("Cancel"), "cancel")))

        e=Entry(3, str(entry_value))
        l=Label(("Timeout (in seconds):"))
        gg=Grid(2,1)
        gg.setField(l,0,0)
        gg.setField(e,1,0)

        g.add(Label(''),0,1)
        g.add(gg,0,2)
        g.add(Label(''),0,3)
        g.add(bb,0,4,growx=1)
        result = g.runOnce()
        if bb.buttonPressed(result) == 'cancel':
            screen.finish()
            sys.exit(0)
        else:
            entry_value = e.value()
            try :
                c = int(entry_value)
                break
            except ValueError:
                continue

    writeBootFile(c, li.current())
    screen.finish()

if __name__ == '__main__':
    main()
```

Start the tool using the sudo command (as it reads the grub.conf file)

```
$ sudo grub_tui.py
```



## What you'll need…

**Beautiful Soup**
www.crummy.com/software/BeautifulSoup/

**HTML5Lib**
https://github.com/html5lib/html5lib-python

**Python 2.6+ & WikiParser.zip Six**
https://pypi.python.org/pypi/six/

# Scrape Wikipedia with Beautiful Soup

Use the Beautiful Soup Python library to parse Wikipedia's HTML and store it for offline reading

In this tutorial we'll use the popular Python library Beautiful Soup to scrape Wikipedia for links to articles and then save those pages for offline reading. This is ideal for when travelling or in a location with a poor internet connection.

The plan is simple: using Beautiful Soup with the HTML5Lib Parser, we're going to load a Wikipedia page, remove all of the GUI and unrelated content, search the content for links to other Wikipedia articles and then, after a tiny bit of modification, write them to a file.

Even though it's now the de facto knowledge base of the world, Wikipedia isn't great when it comes to DOM consistency – that is, IDs and classes are sometimes quite loose in their usage. Because of this, we will also cover how to handle all of the excess bits and bobs of the Wikipedia GUI that we don't need, as well as the various erroneous links that won't be of much use to us. You can find the CSS stylings sheet and a Python script pertaining to this tutorial at **http://bit.ly/19MibBv**.

## Infinite Links

Wikipedia has a lot of links and when you start following links to links to links, the number of pages you have to parse can grow exponentially, depending on the subject matter. By passing through the levels value, we put a cap on the amount of pages we can grab— although the number of files stored can still vary greatly. Use it wisely.

## Install Beautiful Soup & HTML5Lib

**01** Before we can start writing code, we need to install the libraries we'll be using for the program (Beautiful Soup, HTML5Lib, Six). The installation process is fairly standard: grab the libraries from their respective links, then unzip them. In the terminal, enter the unzipped directory and run python setup.py install for each library. They will now be ready for use.

> "Wikipedia isn't great when it comes to DOM consistency"

## Full code listing

**1 Import libraries**
These are the libraries we are going to be using for this program

```
01 import os, sys, urllib2, argparse, datetime, atexit
   from bs4 import BeautifulSoup
```

```
02 addresses = []
   deepestAddresses = []

   maxLevel = 1
   storeFolder = "Wikistore " + str(datetime.datetime.now().strftime("%Y-%m-%d %H:%M"))

   undesirables = [ {"element" : "table", "attr" : {'class' : 'infobox'} }, {"element" :
   "table", "attr" : {'class' : 'vertical-navbox'}}, {"element" : "span", "attr" : {'class'
   : 'mw-editsection'}}, {"element" : "div", "attr" : {'class' : 'thumb'}}, {"element"
   : "sup", "attr" : {'class' : 'reference'}}, {"element" : "div", "attr" : {'class' :
   'reflist'}}, {"element" : "table", "attr" : {'class' : 'nowraplinks'}}, {"element" :
   "table", "attr" : {'class' : 'ambox-Refimprove'}}, {"element" : "img", "attr" : None},
   {"element" : "script", "attr" : None}, {"element" : "table", "attr" : {'class' :
   'mbox-small'}} , {"element" : "span", "attr" : {"id" : "coordinates"}}, {"element" :
   "table", "attr" : {"class" : "ambox-Orphan"}}, {"element" : "div", "attr" : {"class" :
   "mainarticle"}}, {"element" : None, "attr" : {"id" : "References"}} ]
```

**2 Set up variables**
These are some variables we'll use to keep track of the script's progress

```
03 def init():
       parser = argparse.ArgumentParser(description='Handle the starting page and number
   of levels we\'re going to scrape')
       parser.add_argument('-URL', dest='link', action='store', help='The Wikipedia page
   from which we will start scraping')
       parser.add_argument('-levels', dest="levels", action='store', help='How many levels
   deep should the scraping go')
       args = parser.parse_args()

       if(args.levels != None):
           global maxLevel8
           maxLevel = int(args.levels)

       if(args.link == None):
           print("You need to pass a link with the -URL flag")
           sys.exit(0)
       else:
           if not os.path.exists(storeFolder):
               os.makedirs(storeFolder)

           grabPage(args.link, 0, args.link.split("/wiki/")[1].strip().replace("_", " "))

       atexit.register(cleanUp)

   def isValidLink(link):

       if "/wiki/" in link and ":" not in link and "http://" not in link and "wikibooks"
   not in link and "#" not in link and "wikiquote" not in link and "wiktionary" not in
   link and "wikiversity" not in link and "wikivoyage" not in link and "wikisource" not
   in link and "wikinews" not in link and "wikiversity" not in link and "wikidata" not
   in link:
           return True
       else:
           return False
```

**3 Initialisation**
This is the initialising function that we will use to handle the input coming from the user

```
04 def grabPage(URL, level, name):

       opener = urllib2.build_opener()
       opener.addheaders = [('User-agent', 'Mozilla/5.0')]
       req = opener.open(URL)
```

## Wiki-Everything

Wikipedia has so many different services that interlink with each other; however, we don't want to grab those pages, so we've got quite a lengthy conditional statement to stop that. It's pretty good at making sure we only get links from Wikipedia.

### Creating some useful variables

**02** These variables will keep track of the links we've accessed while the script has been running: addresses is a list containing every link we've accessed; deepestAddresses are the links of the pages that were the furthest down the link tree from our starting point; storeFolder is where we will save the HTML files we create and maxLevel is the maximum depth that we can follow the links to from our starting page.

### Handling the user's input

**03** In the first few lines of this function, we're just creating a helper statement. Afterwards, we're parsing any arguments passed into the program on its execution and looking for a -URL flag and a -levels flag. The -levels flag is optional as we already have a preset depth that we'll follow the links to, but we need a link to start from so if the -URL flag is missing, we'll prompt the user and exit. If we have a link, then we quickly check whether or not we have a directory to store files in – which we'll create if we don't – and then we'll fire off the function to get that page. Finally, we register a handler for when the script tries to exit. We'll get to that bit later.

### Retrieving the page from the URL

**04** Here we're using URLLib2 to request the page the the user has asked for and then, once we've received that page, we're going to pass the content through to Beautiful Soup with the soup variable. This gives us access to the methods we're going to call as we parse the document.

### Trimming the fat

**05** Wikipedia has a lot of nodes that we don't want to parse. The content variable allows us to straight away ignore most of Wikipedia's GUI, but there are still lots of elements that we don't want to parse. We remedy this by iterating through the list 'undesirables' that we created earlier on in the document. For each different div/section/node that we don't want, we call Beautiful Soup's find_all() method and use the extract() method to remove that node from the document. At the end of the undesirables loop, most of the content we don't want any more will be gone. We also look for the 'also' element in the Wiki page. Generally, everything after this div is of no use to us. By calling the find_all_next() method on the also node, we can get a list of every other element we can remove from that point on.

"The HTML page uses built-in browser styles when rendering the page"

"Wikipedia has so many different services that interlink with each other; we don't want to grab those pages"

**4 Get the page**
Here we grab the page we want to store and remove the bits of the document we don't need

**5 Check links**
Then we iterate through all of the <a> tags and check if there's a valid link to another page we can grab, and tweak them for our own use

```
page = req.read()

req.close()

soup = BeautifulSoup(page, "html5lib", from_encoding="UTF-8")

content = soup.find(id="mw-content-text")

if hasattr(content, 'find_all'):

    global undesirables

    for notWanted in undesirables:

        removal = content.find_all(notWanted['element'], notWanted['attr'])
        if len(removal) > 0:
            for el in removal:
                el.extract()

    also = content.find(id="See_also")

    if(also != None):
        also.extract()
        tail = also.find_all_next()
        if(len(tail) > 0):
            for element in tail:
                element.extract()

    for link in content.find_all('a'):

        href = link["href"]

        if isValidLink(href):

            if level < maxLevel:

                stored = False;
                for addr in addresses:
                    if addr == link.get("href"):
                        stored = True

                if(stored == False):
                    title = link.get('href').replace("/wiki/", "")
                    addresses.append(str(title + ".html"))
                    grabPage("http://en.wikipedia.org" + link.get('href'), level +
1, title)
                    print title

            link["href"] = link["href"].replace("/wiki/", "") + ".html"

    fileName = str(name)

    if level == maxLevel:
        deepestAddresses.append(fileName.replace('/', '_') + ".html")
```

## Styling

Currently, the HTML page will use the built-in browser styles when rendering the page. If you like, you can include the style sheet included in the tutorial resources to make it look a little nicer. To use it, you can minify the script and include it inside a <style> tag in the head string on line 102, or you can rewrite the head string to something like:

```
head = "<head><meta
charset=\"UTF-8\" /><title>" +
fileName + "</title><style>" +
str(open("/PATH/TO/STYLES", 'r').
read()) + "</style></head>"
```


**Above** Find the documentation for Beautiful Soup at http://bit.ly/O2H8iD

## Grabbing the links

**06** By calling content.find_all('a') we get a list of every <a> in the document. We can iterate through this and check whether or not there is a valid Wikipedia link in the <a>'s href. If the link is a valid link, we quickly check how far down the link tree we are from the original page. If we've reached the maximum depth we can go, we'll store this page and call it quits, otherwise we'll start looking for links that we can grab within it. For every page we request, we append its URL to the addresses list; to make sure we don't call the same page twice for each link we find, we check if we've already stored it. If we have, then we'll skip over the rest of the loop, but if we've not then we'll add it to the list of URLs that we've requested and fire off a request. Once that check is done, We then do a quick string replace on that link so that it points to the local directory, not to the subfolder /wiki/ that it's looking for.

## Writing to file

**07** Now we create a file to store the newly parsed document in for later reading. We change any '/' in the filename to '_' so the script doesn't try and write to a random folder. We also do a quick check to see how many links we've followed since the first page. If it's the max level, we'll add it to the deepestAddresses list. We'll use this a little bit later.

## Tying up loose ends

**08** After our script has iterated through every link on every page to the maximum level of depth that it can, it will try to exit. On line 34 of the code (on the disc and online) in the init function, we registered the function cleanUp to execute on the program trying to exit; cleanUp's job is to go through the documents that we've downloaded and check that every link we've left in the pages does in fact link to a file that we have available. If it can't match the link in the href to a file in the addresses list, it will remove it. Once we're done, we will have a fully portable chunk of Wikipedia we can take with us.

**6 Copy to file**
After that, We take the content we've parsed and put it into a brand new HTML file

```
06          doctype = "<!DOCTYPE html>"

            head = "<head><meta charset=\"UTF-8\" /><title>" + fileName + "</title></
head>"

            f = open(storeFolder + "/" + fileName.replace('/', '_') + ".html", 'w')
            f.write(doctype + "<html lang=\"en\">" + head + "<body><h1>" + fileName + "</
h1>" + str(content) + "</body></html>")
            f.close()
```

**7 Clean up**
Once every page has been parsed and stored, we'll go on through and try to remove any dead links

```
07  def cleanUp():

        print("\nRemoving links to pages that have not been saved\n")

        for deepPage in deepestAddresses:

            rF = open(storeFolder + "/" + deepPage, 'r')

            deepSoup = BeautifulSoup(rF.read(),  "html5lib", from_encoding="UTF-8")

            for deepLinks in deepSoup.find_all('a'):
                link = deepLinks.get("href")

                pageStored = False

                for addr in addresses:
                    if addr == link:
                        pageStored = True

                if pageStored == False:

                    if link is not None:

                        if '#' not in link:
                            del deepLinks['href']
                        elif '#' in link and len(link.split('#')) > 1 or ':' in link:
                            del deepLinks['href']

            wF = open(storeFolder + "/" + deepPage, 'w')
            wF.write(str(deepSoup))
            wF.close()

        print("Complete")
```

**8 Initialise**
This is how we will initialise our script

```
08  if __name__ == "__main__":
        init()
```

# Create with Python

## Use Python to get creative and program games

What could be more satisfying than playing a game that you have programmed yourself? In this section we're going to show you how to do just that. We'll get started with a simple game of tic-tac-toe, made with the help of Kivy (p.80), before stepping things up a notch and cloning the classic favourite, Pong (p.86). Then, it's time to have a go at making a Space Invaders-inspired game complete with retro graphics (p.88). Finally, you'll learn how to make a stripped-back 'choose-your-own-adventure' game (p.98).


"Making a playable game is not
as difficult as you may think"

## What you'll need…

**Python**
www.python.org/doc

# Build tic-tac-toe with Kivy

Ease into the workings of Kivy by creating the pen-and-paper classic in just over 100 lines of Python...

Kivy is a highly cross-platform graphical framework for Python, designed for the creation of innovative user interfaces like multitouch apps. Its applications can run not only on the traditional desktop platforms of Linux, OS X and Windows, but also Android and iOS, plus devices like the Raspberry Pi.

That means you can develop cross-platform apps using Python libraries such as Requests, SQLAlchemy or even NumPy. You can even access native mobile APIs straight from Python using some of Kivy's sister projects. Another great feature is the Cython-optimised OpenGL graphics pipeline, allowing advanced GPU effects even though the basic Python API is very simple.

Kivy is a set of Python/Cython modules that can easily be installed via pip, but you'll need a few dependencies. It uses Pygame as a rendering backend (though its API is not exposed), Cython for compilation of the speedy graphics compiler internals, and GStreamer for multimedia. These should all be available through your distro's repositories, or via pip where applicable.

With these dependencies satisfied, you should be able install Kivy with the normal pip incantation. The current version is 1.8.0, and the same codebase supports both python2 and python3. The code in this tutorial is also version-agnostic, running in python2.7 and python3.3.

```
pip install kivy
```

If you have any problems with pip, you can use easy_install via easy_install kivy.

There are also packages or repositories available for several popular distros. You can find more information on Kivy's website. A kivy application is started by instantiating and running an 'App' class. This is what initialises our pp's window, interfaces with the OS, and provides an

entry point for the creation of our GUI. We can start by making the simplest Kivy app possible:

```
from kivy.app import App

class TicTacToeApp(App):
  pass

if __name__ == "__main__":
  TicTacToeApp().run()
```

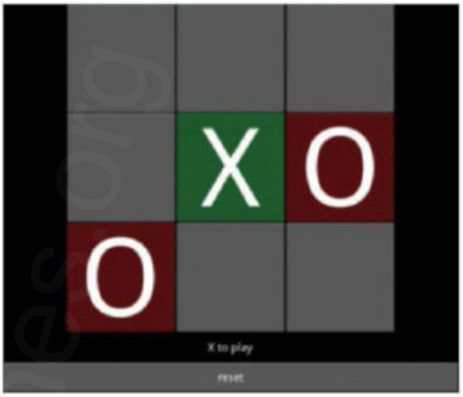
You can already run this, your app will start up and you'll get a plain black window. Exciting!

We can build our own GUI out of Kivy widgets. Each is a simple graphics element with some specific behaviour of its own ranging from standard GUI functionality (eg the Button, Label or TextInput), to those that impose positioning on their child widgets (eg the BoxLayout, FloatLayout or GridLayout), to those abstracting a more involved task like interacting with hardware (eg the FileChooser, Camera or VideoPlayer). Most importantly, Kivy's widgets are designed to be easily combined - rather than including a widget for every need imaginable, widgets are kept simple but are easy to join to invent new interfaces. We'll see some of that in this tutorial.

Since 'Hello World!' is basically compulsory in any programming tutorial, let's get it over with by using a simple 'Label' widget to display the text:

```
from kivy.uix.label import Label
```

We'll display the 'Label' by returning it as our app's root widget. Every app has a single root widget, the top level of its widget tree, and it will automatically be sized to fill the window. We'll see later how to construct a full GUI by adding more widgets for this one, but for now it's enough to set the root widget by adding a new method to the 'App':




**Above** The game with final additions, making the grid square and extending the interface

```
def build(self):
  return Label(text='Hello World!',
  font_size=100,
  color=0, 1, 0, 1))  # (r, g, b, a)
```

The 'build' method is called when the 'App' is run, and whatever widget is returned automatically becomes the root widget of that App'. In our case that's a Label, and we've set several properties - the 'text', 'font_size' and 'color'. All widgets have different properties controlling aspects of their behaviour, which can be dynamically updated to alter their appearance later, though here we set them just once upon instantiation.

Note that these properties are not just Python attributes but instead Kivy properties. These are accessed like normal attributes but provide extra functionality by hooking into Kivy's event system. We'll see examples of creating properties shortly, and you should do the same if you want to use your variables with Kivy's event or binding functionality.

That's all you need to show some simple text, so run the program again to check that this does work. You can experiment with the parameters if it's unclear what any of them are doing.

## Our own widget: tic-tac-toe

Since Kivy doesn't have a tic-tac-toe widget, we'll have to make our own! It's natural to create a new widget class to contain this behaviour:

```
from kivy.uix.gridlayout import GridLayout
class TicTacToeGrid(GridLayout):
  pass
```

Now this obviously doesn't do anything yet, except that it inherits all the behaviour of the Kivy GridLayout widget - that is, we'll need to tell it how many columns to have, but then it will automatically arrange any child widgets to fit nicely with as many rows as necessary. Tic-tac-toe requires three columns and nine children.

Here we introduce the Kivy language (kv), a special domain-specific language for making rules describing Kivy widget trees. It's very simple but removes a lot of necessary boilerplate for manipulating the GUI with Python code - as a loose analogy you might think of it as the HTML/CSS to Python's JavaScript. Python gives us the dynamic power to do anything, but all that power gets in the way if we just want to declare the basic structure of our GUI. Note that you never need kv language, you can always do the same thing in Python alone, but the rest of the example may show why Kivy programmers usually like to use kv.

Kivy comes with all the tools needed to use kv language; the simplest way is to write it in a file with a name based on our App class. That is, we should place the following in a file named 'tictactoe.kv':

```
<TicTacToeGrid>:
  cols: 3
```

This is the basic syntax of kv language; for each widget type we may write a rule defining its behaviour, including setting its properties and adding child widgets. This example demonstrates the former, creating a rule for the 'TicTacToeGrid' widget by declaring that every 'TicTacToeGrid' instantiated should have its 'cols' property set to 3.

We'll use some more kv language features later, but for now let's go back to Python to create the buttons that will be the entries in our tic-tac-toe grid.

```
from kivy.uix.button import Button
from kivy.properties import ListProperty

class GridEntry(Button):
  coords = ListProperty([0, 0])
```

This inherits from Kivy's 'Button' widget, which interacts with mouse or touch input, dispatching events when interactions toggle it. We can hook into these events to call our own functions when a user presses the button, and can set the button's 'text' property to display the 'X' or 'O'. We also created a new Kivy property for our widget, 'coords' – we'll show how this is useful later on. It's almost identical to making a normal Python attribute by writing 'self.coords = [0, 0]' in 'GridEntry.__init__'.

As with the 'TicTacToeGrid', we'll style our new class with kv language, but this time we get to see a more interesting feature.

```
<GridEntry>:
  font_size: self.height
```

As before, this syntax defines a rule for how a 'GridEntry' widget should be constructed, this time setting the 'font_size' property that controls the size of the text in the button's label. The extra magic is that kv language automatically detects that we've referenced the Button's own height and will create a binding to update this relationship – when a 'GridEntry' widget's height changes, its 'font_size' will change so the text fits perfectly. We could have

made these bindings straight from Python (another usage of the 'bind' method used later on), but that's rarely as convenient as referencing the property we want to bind to.

Let's now populate our 'TicTacToeGrid' with 'GridEntry' widgets.

```
class TicTacToeGrid(GridLayout):
  def __init__(self, *args, **kwargs):
  super(TicTacToeGrid, self).__init__(*args,
**kwargs)
  for row in range(3):
    for column in range(3):
    grid_entry = GridEntry(
      coords=(row, column))
    grid_entry.bind(on_release=self.button_
pressed)
    self.add_widget(grid_entry)

  def button_pressed(self, instance):
  print('{} button clicked!'.format(instance.
coords))
```

This introduces a few new concepts: When we instantiated our 'GridEntry' widgets, we were able to set their 'coords' property by simply passing it in as a kwarg. This is a minor feature that is automatically handled by Kivy properties.

We used the 'bind' method to call the grid's 'button_pressed' method whenever the `GridEntry` widget dispatches an 'on_release' event. This is automatically handled by its 'Button' superclass, and will occur whenever a user presses, then releases a 'GridEntry' button. We could also bind to 'on_press', which is dispatched when the button is first clicked, or to any Kivy property of the button, dispatched dynamically when the property is modified.

We added each 'GridEntry' widget to our 'Grid' via the 'add_widget' method. That means each one is a child widget of the 'TicTacToeGrid', and so it will display them and knows it should automatically arrange them into a grid with the number of columns we set earlier.

Now all we have to do is replace our root widget (returned from 'App.build') with a 'TicTacToeGrid' and we can see what our app looks like.

```
def build(self):
  return TicTacToeGrid()
```

With this complete you can run your main Python file again and enjoy your new program. All being well, the single Label is replaced by a grid of nine buttons, each of which you can click (it will automatically change colour) and release (you'll see the printed output information from our binding).

We could customise the appearance by modifying other properties of the Button, but for now we'll leave them as they are.

## Has anyone won yet?

We'll want to keep track of the state of the board to check if anyone has won, which we can do with a couple more Kivy properties:

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

class TicTacToeGrid(GridLayout):
  status = ListProperty([0, 0, 0, 0, 0, 0,
                         0, 0, 0])
  current_player = NumericProperty(1)
```

This adds an internal status list representing who has played where, and a number to represent the current player (1 for 'O', -1 for 'X').

By placing these numbers in our status list, we'll know if somebody wins because the sum of a row, column or diagonal will be +–3. Now we can update our graphical grid when a move is played.

```
def button_pressed(self, button):
    player = {1: 'O', -1: 'X'}
    colours = {1: (1, 0, 0, 1), -1: (0, 1, 0,
              1)}  # (r, g, b, a)

    row, column = button.coords

    status_index = 3*row + column
    already_played = self.status[status_index]

    if not already_played:
      self.status[status_index] = self.
                  current_player
      button.text = {1: 'O', -1: 'X'}[self.
                  current_player]
      button.background_color = colours[self.
                  current_player]
      self.current_player *= -1
```

You can run your app again to see exactly what this did, and you'll find that clicking each button now places an 'O' or 'X' as well as a coloured background depending on whose turn it is to play. Not only that, but you can only play one move in each button thanks to our status array that keeps track of the existing moves.

This is enough to play the game but there's one vital element missing... a big pop-up telling you when you've won! Before we can do that, we need to add some code to check if the game is over.

Kivy properties have another useful feature here, whenever they change they automatically call an 'on_propertyname' method if it exists and dispatch a corresponding event in Kivy's event system. That makes it very easy to write code that will run when a property changes, both in Python and kv language. In our case we can use it to check the status list every time it is updated, doing something special if a player has filled a column, row or diagonal.

```
def on_status(self, instance, new_value):
  status = new_value

  sums = [sum(status[0:3]),   # rows
    sum(status[3:6]),
    sum(status[6:9]),
    sum(status[0::3]),   # columns
    sum(status[1::3]),
    sum(status[2::3]),
    sum(status[::4]),   # diagonals
    sum(status[2:-2:2])]

  if 3 in sums:
  print('Os win!')
  elif -3 in sums:
  print('Xs win!')
  elif 0 not in self.status:   # Grid full
  print('Draw!')
```

This covers the basic detection of a won or drawn board, but it only prints the result to stdout. At this stage we probably want to reset the board so that the players can try again, along with displaying a graphical indicator of the result.

```
def reset(self, *args):
  self.status = [0 for _ in range(9)]

  for child in self.children:
    child.text = ''
    child.background_color = (1, 1, 1, 1)

  self.current_player = 1
```

Finally, we can modify the `on_status` method to both reset the board and display the winner in a 'ModalView' widget.

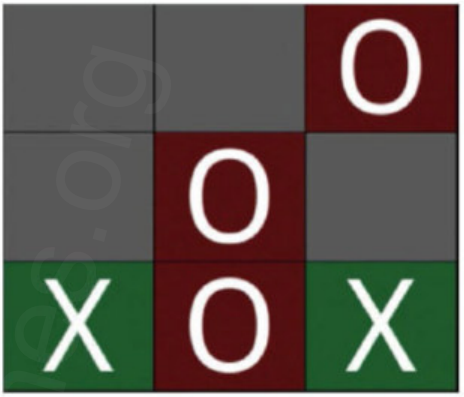
```
from kivy.uix.modalview import ModalView
```

This is a pop-up widget that draws itself on top of everything else rather than as part of the normal widget tree. It also automatically closes when the user clicks or taps outside it.

```
winner = None
if -3 in sums:
  winner = 'Xs win!'
elif 3 in sums:
  winner = 'Os win!'
elif 0 not in self.status:
  winner = 'Draw...nobody wins!'

if winner:
  popup = ModalView(size_hint=0.75, 0.5))
  victory_label = Label(text=winner,
  font_size=50)
  popup.add_widget(victory_label)
  popup.bind(on_dismiss=self.reset)
  popup.open()
```

This mostly uses the same ideas we already covered, adding the 'Label' widget to the 'ModalView' then letting the 'ModalView' take care of drawing itself and its children on top of everything else. We also use another binding; this time to 'on_dismiss', which is an event dispatched by the 'ModalView' when it is closed. Finally, we made use of the 'size_hint' property common to all widgets, which in this case is used to set the 'ModalView' size proportional to the window – while a 'ModalView' is open you can resize the window to see it dynamically resize, always maintaining these proportions. This is another trick made possible by a binding with the 'size_hint' Kivy property, this time managed internally by Kivy.

That's it, a finished program! We can now not only play tic-tac-toe, but our program automatically tells us when somebody has won, and resets the board so we can play again. Simply run your program and enjoy hours of fun!




**Above** A tic-tac-toe grid now accepting input, adding in an O or X alternately, each go

## Time to experiment

This has been a quick tour through some of Kivy's features, but hopefully it demonstrates how to think about building a Kivy application. Our programs are built from individual Kivy widgets, interacting by having Python code run when their properties change (eg our 'on_status' method) or when they dispatch events (eg 'Button' 'on_release'). We also briefly saw kv language and experienced how it can automatically create bindings between properties.

You can find a copy of the full program on FileSilo, reference this to check you've followed everything correctly. We've also added an extra widget, the 'Interface', with a structure coded entirely in kv language that demonstrates how to add child widgets. Test it by uncommenting the 'return Interface()' line in 'TicTacToeGrid.build'. It doesn't do anything fundamentally different to what we already covered, but it does make extensive use of kv language's binding ability to automatically update a label showing the current player, and to resize the TicTacToeGrid so it is always square to fit within its parent. You can play with the settings to see how it fits together, or try swapping out the different widget types to see how other widgets behave.

## What you'll need…

**Latest Raspbian Image**
www.raspberrypi.org/downloads

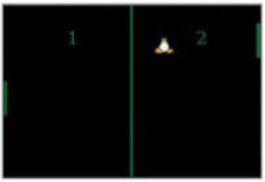
**Pillow**
https://github.com/python-imaging/Pillow

**SimpleGUITk**
https://github.com/dholm/simpleguitk/

# Make a Pong clone with Python

We update the retro classic Pong for the Linux generation with a new library called SimpleGUITk

Below 'Tux for Two' is a great little Pong clone using the beloved Linux mascot, Tux, in the centre of the action




The Raspberry Pi is a fantastic way to start learning how to code. One area that can be very rewarding for amateur coders is game programming, allowing for a more interactive result and a greater sense of accomplishment. Game programming can also teach improvisation and advanced mathematics skills for code. We'll be using the fantastic SimpleGUITk module in Python, a very straightforward way of creating graphical interfaces based on Tkinter.

## Python module preparation

**01** Head to the websites we've listed in 'What you'll need' and download a zip of the source files from the GitHub pages. Update your Raspbian packages and then install the following:

```
$ sudo apt-get install python-dev python-setuptools tk8.5-dev tcl8.5-dev
```

## Install the modules

**02** Open the terminal and use cd to move to the extracted Pillow folder. Once there, type:

```
$ sudo python setup.py install
```

Once that's complete, move to the **simpleguitk** folder and use the same command to install that as well.

## Write your code

**03** Launch IDLE 2, rather than IDLE 3, and open a new window. Use the code listing to create our game 'Tux for Two'. Be careful to follow along with the code to make sure you know what you're doing. This way, you can make your own changes to the game rules if you wish.

## Set up the game

**04** There's nothing too groundbreaking to start the code: Tux's and the paddles' initial positions are set, along with the initial speed and direction of Tux. These are also used when a point is won and the playing field is reset. The direction and speed is set to random for each spawn.

## The SimpleGUI code

**05** The important parts in the **draw** function are the **draw_line**, **draw_image** and **draw_text** functions. These are specifically from SimpleGUI, and allow you to easily put these objects on the screen with a position, size and colour. You need to tie them to an object, though – in this case, **canvas**. This tells the software that we want to put these items on the screen for people to see.

## SimpleGUI setup code

**06** The last parts are purely for the interface. We tell the code what to do when a key is depressed and then released, and give it a frame to work in. The frame is then told what functions handle the graphics, key functions etc. Finally, we give it **frame.start()** so it starts.

## Full code listing

```
import simpleguitk as simplegui
import random

w, h = 600, 400
tux_r = 20
pad_w= 8
pad_h = 80

def tux_spawn(right):
    global tux_pos, tux_vel
    tux_pos = [0,0]
    tux_vel = [0,0]
    tux_pos[0] = w/2
    tux_pos[1] = h/2
    if right:
        tux_vel[0] = random.randrange(2, 4)
    else:
        tux_vel[0] = -random.randrange(2, 4)
    tux_vel[1] = -random.randrange(1, 3)

def start():
    global paddle1_pos, paddle2_pos, paddle1_vel, paddle2_vel
    global score1, score2
    tux_spawn(random.choice([True, False]))
    score1, score2 = 0,0
    paddle1_vel, paddle2_vel = 0,0
    paddle1_pos, paddle2_pos = h/2, h/2

def draw(canvas):
    global score1, score2, paddle1_pos, paddle2_pos, tux_pos, tux_vel
    if paddle1_pos > (h - (pad_h/2)):
        paddle1_pos = (h - (pad_h/2))
    elif paddle1_pos < (pad_h/2):
        paddle1_pos = (pad_h/2)
    else:
        paddle1_pos += paddle1_vel
    if paddle2_pos > (h - (pad_h/2)):
        paddle2_pos = (h - (pad_h/2))
    elif paddle2_pos < (pad_h/2):
        paddle2_pos = (pad_h/2)
    else:
        paddle2_pos += paddle2_vel
    canvas.draw_line([w / 2, 0],[w / 2, h], 4, "Green")
    canvas.draw_line([(pad_w/2), paddle1_pos + (pad_h/2)], [(pad_w/2), paddle1_pos - (pad_h/2)], pad_w, "Green")
    canvas.draw_line([w - (pad_w/2), paddle2_pos + (pad_h/2)], [w - (pad_w/2), paddle2_pos - (pad_h/2)], pad_w, "Green")
    tux_pos[0] += tux_vel[0]
    tux_pos[1] += tux_vel[1]
    if tux_pos[1] <= tux_r or tux_pos[1] >= h - tux_r:
        tux_vel[1] = -tux_vel[1]*1.1
    if tux_pos[0] <= pad_w + tux_r:
        if (paddle1_pos+(pad_h/2)) >= tux_pos[1] >= (paddle1_pos-(pad_h/2)):
            tux_vel[0] = -tux_vel[0]*1.1
            tux_vel[1] *= 1.1
        else:
            score2 += 1
            tux_spawn(True)
    elif tux_pos[0] >= w - pad_w - tux_r:
        if (paddle2_pos+(pad_h/2)) >= tux_pos[1] >= (paddle2_pos-(pad_h/2)):
            tux_vel[0] = -tux_vel[0]
            tux_vel[1] *= 1.1
        else:
            score1 += 1
            tux_spawn(False)
    canvas.draw_image(tux, (265 / 2, 314 / 2), (265, 314), tux_pos, (45, 45))
    canvas.draw_text(str(score1), [150, 100], 30, "Green")
    canvas.draw_text(str(score2), [450, 100], 30, "Green")

def keydown(key):
    global paddle1_vel, paddle2_vel
    acc = 3
    if key == simplegui.KEY_MAP["w"]:
        paddle1_vel -= acc
    elif key == simplegui.KEY_MAP["s"]:
        paddle1_vel += acc
    elif key==simplegui.KEY_MAP["down"]:
        paddle2_vel += acc
    elif key==simplegui.KEY_MAP["up"]:
        paddle2_vel -= acc

def keyup(key):
    global paddle1_vel, paddle2_vel
    acc = 0
    if key == simplegui.KEY_MAP["w"]:
        paddle1_vel = acc
    elif key == simplegui.KEY_MAP["s"]:
        paddle1_vel = acc
    elif key==simplegui.KEY_MAP["down"]:
        paddle2_vel = acc
    elif key==simplegui.KEY_MAP["up"]:
        paddle2_vel = acc

frame = simplegui.create_frame("Tux for Two", w, h)
frame.set_draw_handler(draw)
frame.set_keydown_handler(keydown)
frame.set_keyup_handler(keyup)
tux = simplegui.load_image('http://upload.wikimedia.org/wikipedia/commons/a/af/Tux.png')

start()
frame.start()
```

## What you'll need…

**Raspbian**
www.raspberrypi.org/downloads

**Python**
www.python.org/doc

**Pygame**
www.pygame.org/docs

# Program a Space Invaders clone

Write your own RasPi shooter in 300 lines of Python

When you're learning to program in a new language or trying to master a new module, experimenting with a familiar and relatively simply project is a very useful exercise to help expand your understanding of the tools you're using. Our Space Invaders clone is one such example that lends itself perfectly to Python and the Pygame module – it's a simple game with almost universally understood rules and logic.

We've tried to use many features of Pygame, which is designed to make the creation of games and interactive applications easier. We've extensively used the Sprite class, which saves dozens of lines of extra code in making collision detection simple and updating the screen and its many actors a single-line command.

Have fun with the project and make sure you tweak and change things to make it your own!

### Did you know…

Space Invaders was one of the biggest arcade hits in the world. It's a great first game since everyone knows how to play!


**Right** Pivaders is a Space Invaders clone we've made especially for the Pi

## Full code listing

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

import pygame, random

BLACK = (0, 0, 0)
BLUE = (0, 0, 255)
WHITE = (255, 255, 255)
RED = (255, 0, 0)
ALIEN_SIZE = (30, 40)
ALIEN_SPACER = 20
BARRIER_ROW = 10
BARRIER_COLUMN = 4
BULLET_SIZE  = (5, 10)
MISSILE_SIZE = (5, 5)
BLOCK_SIZE = (10, 10)
RES = (800, 600)

class Player(pygame.sprite.Sprite):
  def __init__(self):
    pygame.sprite.Sprite.__init__(self)
    self.size = (60, 55)
    self.rect = self.image.get_rect()
    self.rect.x = (RES[0] / 2) - (self.size
[0] / 2)
    self.rect.y = 520
    self.travel = 7
    self.speed = 350
    self.time = pygame.time.get_ticks()

  def update(self):
    self.rect.x += GameState.vector * self.
travel
    if self.rect.x < 0:
      self.rect.x = 0
    elif self.rect.x > RES[0] - self.size[0]:
      self.rect.x = RES[0] - self.size[0]

class Alien(pygame.sprite.Sprite):
  def __init__(self):
    pygame.sprite.Sprite.__init__(self)
    self.size = (ALIEN_SIZE)
    self.rect = self.image.get_rect()
    self.has_moved = [0, 0]
    self.vector = [1, 1]
    self.travel = [(ALIEN_SIZE[0] - 7),
ALIEN_SPACER]
    self.speed = 700
    self.time = pygame.time.get_ticks()

  def update(self):
    if GameState.alien_time - self.time >
self.speed:
      if self.has_moved[0] < 12:
        self.rect.x += self.vector[0] * self.
travel[0]
        self.has_moved[0] +=1
      else:
        if not self.has_moved[1]:
          self.rect.y += self.vector[1] *
self.travel[1]
        self.vector[0] *= -1
        self.has_moved = [0, 0]
        self.speed -= 20
        if self.speed <= 100:
          self.speed = 100
      self.time = GameState.alien_time

class Ammo(pygame.sprite.Sprite):
  def __init__(self, color, (width, height)):
    pygame.sprite.Sprite.__init__(self)
    self.image = pygame.Surface([width,
height])
    self.image.fill(color)
    self.rect = self.image.get_rect()
    self.speed = 0
    self.vector = 0

  def update(self):
    self.rect.y += self.vector * self.speed
    if self.rect.y < 0 or self.rect.y > RES[1]:
      self.kill()

class Block(pygame.sprite.Sprite):
  def __init__(self, color, (width, height)):
    pygame.sprite.Sprite.__init__(self)
    self.image = pygame.Surface([width,
height])
    self.image.fill(color)
    self.rect = self.image.get_rect()
class GameState:
  pass

class Game(object):
  def __init__(self):
    pygame.init()
    pygame.font.init()
    self.clock = pygame.time.Clock()
    self.game_font = pygame.font.Font(
    'data/Orbitracer.ttf', 28)
    self.intro_font = pygame.font.Font(
    'data/Orbitracer.ttf', 72)
    self.screen = pygame.display.set_
mode([RES[0], RES[1]])
    self.time = pygame.time.get_ticks()
    self.refresh_rate = 20
    self.rounds_won = 0
    self.level_up = 50
    self.score = 0
    self.lives = 2
    self.player_group = pygame.sprite.Group()
    self.alien_group = pygame.sprite.Group()
    self.bullet_group = pygame.sprite.Group()
    self.missile_group = pygame.sprite.Group()
    self.barrier_group = pygame.sprite.Group()
```



Continued on page 91

## Setting up dependencies

**01** If you're looking to get a better understanding of programming games with Python and Pygame, we strongly recommend you copy the Pivaders code in this tutorial into your own program. It's great practice and gives you a chance to tweak elements of the game to suit you, be it a different ship image, changing the difficulty or the ways the alien waves behave. If you just want to play the game, that's easily achieved too, though. Either way, the game's only dependency is Pygame, which (if it isn't already) can be installed from the terminal by typing:

```
sudo apt-get install python-pygame
```

## Installation

**02** For Pivaders we've used Git, a brilliant form of version control used to safely store the game files and retain historical versions of your code. Git should already be installed on your Pi; if not, you can acquire it by typing:

```
sudo apt-get install git
```

As well as acting as caretaker for your code, Git enables you to clone copies of other people's projects so you can work on them, or just use them. To clone Pivaders, go to your home folder in the terminal (`cd ~`), make a directory for the project (mkdir pivaders), enter the directory (cd pivaders) and type:

```
git pull https://github.com/russb78/pivaders.git
```

## Testing Pivaders

**03** With Pygame installed and the project cloned to your machine (you can also find the .zip on this issue's cover DVD – simply unpack it and copy it to your home directory to use it), you can take it for a quick test drive to make sure everything's set up properly. All you need to do is type python pivaders.py from within the pivaders directory in the terminal to get started. You can start the game with the space bar, shoot with the same button and simply use the left and right arrows on your keyboard to move your ship left and right.

## Creating your own clone

**04** Once you've racked up a good high score (anything over 2,000 points is respectable) and got to know our simple implementation, you'll get more from following along with and exploring the code and our brief explanations of what's going on. For those who want to make their own project, create a new project folder and use either IDLE or Leafpad (or perhaps install Geany) to create and save a .py file of your own.

## Global variables & tuples

**05** Once we've imported the modules we need for the project, there's quite a long list of variables in block capitals. The capitals denote that these variables are constants (or global variables). These are important numbers that never change – they represent things referred to regularly in the code, like colours, block sizes and resolution. You'll also notice that colours and sizes hold multiple numbers in braces – these are tuples. You could use square brackets (to make them lists), but we use tuples here since they're immutable, which means you can't reassign individual items within them. Perfect for constants, which aren't designed to change anyway.

## Classes – part 1

**06** A class is essentially a blueprint for an object you'd like to make. In the case of our player, it contains all the required info, from which you can make multiple copies (we create a player instance in the make_player() method halfway through the project). The great thing about the classes in Pivaders is that they inherit lots of capabilities and shortcuts from Pygame's Sprite class, as denoted by the pygame.sprite.Sprite found within the braces of the first line of the class. You can read the docs to learn more about the Sprite class via **www.pygame.org/docs/ref/sprite.html**.

"We've tried to use many features of Pygame, which is designed to make the creation of games and interactive applications easier"

Continued from page 89

```
    self.all_sprite_list = pygame.sprite.Group()
    self.intro_screen = pygame.image.load('data/start_screen.jpg').convert()
    self.background = pygame.image.load('data/Space-Background.jpg').convert()
    pygame.display.set_caption('Pivaders - ESC to exit')
    pygame.mouse.set_visible(False)
    Player.image = pygame.image.load('data/ship.png').convert()
    Player.image.set_colorkey(BLACK)
    Alien.image = pygame.image.load('data/Spaceship16.png').convert()
    Alien.image.set_colorkey(WHITE)
    GameState.end_game = False
    GameState.start_screen = True
    GameState.vector = 0
    GameState.shoot_bullet = False

  def control(self):
    for event in pygame.event.get():
      if event.type == pygame.QUIT:
        GameState.start_screen = False
        GameState.end_game = True
      if event.type == pygame.KEYDOWN \
      and event.key == pygame.K_ESCAPE:
        if GameState.start_screen:
          GameState.start_screen = False
          GameState.end_game = True
          self.kill_all()
        else:
          GameState.start_screen = True
    self.keys = pygame.key.get_pressed()
    if self.keys[pygame.K_LEFT]:
      GameState.vector = -1
    elif self.keys[pygame.K_RIGHT]:
      GameState.vector = 1
    else:
      GameState.vector = 0
    if self.keys[pygame.K_SPACE]:
      if GameState.start_screen:
        GameState.start_screen = False
        self.lives = 2
        self.score = 0
        self.make_player()
        self.make_defenses()
        self.alien_wave(0)
      else:
        GameState.shoot_bullet = True

  def splash_screen(self):
    while GameState.start_screen:
      self.kill_all()
      self.screen.blit(self.intro_screen, [0, 0])
      self.screen.blit(self.intro_font.render(
      "PIVADERS", 1, WHITE), (265, 120))
      self.screen.blit(self.game_font.render(
      "PRESS SPACE TO PLAY", 1, WHITE), (274, 191))
      pygame.display.flip()
      self.control()

  def make_player(self):
    self.player = Player()
    self.player_group.add(self.player)
    self.all_sprite_list.add(self.player)

  def refresh_screen(self):
    self.all_sprite_list.draw(self.screen)
    self.refresh_scores()
    pygame.display.flip()
    self.screen.blit(self.background, [0, 0])
    self.clock.tick(self.refresh_rate)

  def refresh_scores(self):
    self.screen.blit(self.game_font.render(
    "SCORE " + str(self.score), 1, WHITE), (10, 8))
    self.screen.blit(self.game_font.render(
    "LIVES " + str(self.lives + 1), 1, RED), (355, 575))

  def alien_wave(self, speed):
    for column in range(BARRIER_COLUMN):
      for row in range(BARRIER_ROW):
        alien = Alien()
        alien.rect.y = 65 + (column * (
        ALIEN_SIZE[1] + ALIEN_SPACER))
        alien.rect.x = ALIEN_SPACER + (
        row * (ALIEN_SIZE[0] + ALIEN_SPACER))
        self.alien_group.add(alien)
        self.all_sprite_list.add(alien)
        alien.speed -= speed

  def make_bullet(self):
    if GameState.game_time - self.player.time > self.player.speed:
      bullet = Ammo(BLUE, BULLET_SIZE)
      bullet.vector = -1
      bullet.speed = 26
      bullet.rect.x = self.player.rect.x + 28
      bullet.rect.y = self.player.rect.y
      self.bullet_group.add(bullet)
      self.all_sprite_list.add(bullet)
      self.player.time = GameState.game_time
    GameState.shoot_bullet = False

  def make_missile(self):
    if len(self.alien_group):
      shoot = random.random()
      if shoot <= 0.05:
        shooter = random.choice([
        alien for alien in self.alien_group])
        missile = Ammo(RED, MISSILE_SIZE)
```



Continued on page 93

## Classes – part 2

**07** In Pivader's classes, besides creating the required attributes for the object, you'll also notice all the classes have an update() method apart from the Block class (a method is a function within a class). The update() method is called in every loop through the main game and simply asks the iteration of the class we've created to move. In the case of a bullet from the Ammo class, we're asking it to move down the screen. If it goes off either end, we destroy it.


## Ammo

**08** What's most interesting about classes, though, is that you can use one class to create lots of different things. You could, for example, have a pet class. From that class you could create a cat (that meows) and a dog (that barks). They're different in many ways, but they're both furry and have four legs, so can be created from the same parent class. We've done exactly that with our Ammo class, using it to create both the player bullets and the alien missiles. They're different colours and they shoot in opposite directions, but they're fundamentally one and the same.

## The game

**09** Our final class is called Game. This is where all the main functionality of the game itself comes in, but remember, so far this is still just a list of ingredients – nothing can actually happen until a 'Game' object is created (right at the bottom of the code). The Game class is where the central mass of the game resides, so we initialise Pygame, set the imagery for our protagonist and extraterrestrial antagonist and create some GameState attributes that we use to control key aspects of external classes, like changing the player's vector (direction).

## The main loop

**10** There are a lot of methods (class functions) in the Game class, and each is designed to control a particular aspect of either setting up the game or the gameplay itself. The logic that dictates what happens within any one round of the game is contained in the main_loop() method right at the bottom of the pivaders.py script and is the key to unlocking exactly what variables and functions you need for your game.

## Main loop key logic – part 1

**11** Firstly the game checks that the end_game attribute is false – if it's true, the entire loop in main_loop() is skipped and we go straight to pygame.quit(), exiting the game. This flag is set to true only if the player closes the game window or presses the Esc key when on the start_screen. Assuming end_game and start_screen are false, the main loop can start proper, with the control() method, which checks to see if the location of the player needs to change. Next we attempt to make an enemy missile and we use the random module to limit the number of missiles that can be created. Next we call the update() method for each and every actor on the screen using a simple for loop. This makes sure everyone's up to date and moved before we check collisions in calc_collisions().

## Main loop key logic – part 2

**12** Once collisions have been calculated, we need to see if the game is still meant to continue. We do so with is_dead() and defenses_breached() – if either of these methods returns true, we know we need to return to the start screen. On the other hand, we also need to check to see if we've killed all the aliens, from within win_round(). Assuming we're not dead, but the aliens are, we know we can call the next_round() method, which creates a fresh batch of aliens and increases their speed around the screen. Finally, we refresh the screen so everything that's been moved, shot or killed can be updated or removed from the screen. Remember, the main loop happens 20 times a second – so the fact we don't call for the screen to update right at the end of the loop is of no consequence.

Continued from page 91

```
        missile.vector = 1
        missile.rect.x = shooter.rect.x + 15
        missile.rect.y = shooter.rect.y + 40
        missile.speed = 10
        self.missile_group.add(missile)
        self.all_sprite_list.add(missile)

    def make_barrier(self, columns, rows, spacer):
        for column in range(columns):
            for row in range(rows):
                barrier = Block(WHITE, (BLOCK_SIZE))
                barrier.rect.x = 55 + (200 * spacer) ↵
+ (row * 10)
                barrier.rect.y = 450 + (column * 10)
                self.barrier_group.add(barrier)
                self.all_sprite_list.add(barrier)

    def make_defenses(self):
        for spacing, spacing in ↵
enumerate(xrange(4)):
            self.make_barrier(3, 9, spacing)

    def kill_all(self):
        for items in [self.bullet_group, self. ↵
player_group,
        self.alien_group, self.missile_group, ↵
self.barrier_group]:
            for i in items:
                i.kill()

    def is_dead(self):
        if self.lives < 0:
            self.screen.blit(self.game_font.render(
            "The war is lost! You scored: " + str(
            self.score), 1, RED), (250, 15))
            self.rounds_won = 0
            self.refresh_screen()
            pygame.time.delay(3000)
            return True

    def win_round(self):
        if len(self.alien_group) < 1:
            self.rounds_won += 1
            self.screen.blit(self.game_font.render(
            "You won round " + str(self.rounds_won) +
            "  but the battle rages on", 1, RED), ↵
(200, 15))
            self.refresh_screen()
            pygame.time.delay(3000)
            return True

    def defenses_breached(self):
        for alien in self.alien_group:
            if alien.rect.y > 410:
                self.screen.blit(self.game_font.render(
                "The aliens have breached Earth ↵
defenses!",
                1, RED), (180, 15))
                self.refresh_screen()
                pygame.time.delay(3000)
                return True

    def calc_collisions(self):
        pygame.sprite.groupcollide(
        self.missile_group, self.barrier_group, ↵
True, True)
        pygame.sprite.groupcollide(
        self.bullet_group, self.barrier_group, ↵
True, True)
        if pygame.sprite.groupcollide(
        self.bullet_group, self.alien_group, ↵
True, True):
            self.score += 10
        if pygame.sprite.groupcollide(
        self.player_group, self.missile_group, ↵
False, True):
            self.lives -= 1

    def next_round(self):
        for actor in [self.missile_group,
        self.barrier_group, self.bullet_group]:
            for i in actor:
                i.kill()
        self.alien_wave(self.level_up)
        self.make_defenses()
        self.level_up += 50

    def main_loop(self):
        while not GameState.end_game:
            while not GameState.start_screen:
                GameState.game_time = pygame.time. ↵
get_ticks()
                GameState.alien_time = pygame.time. ↵
get_ticks()
                self.control()
                self.make_missile()
                for actor in [self.player_group, ↵
self.bullet_group,
                self.alien_group, self.missile_group]:
                    for i in actor:
                        i.update()
                if GameState.shoot_bullet:
                    self.make_bullet()
                self.calc_collisions()
                if self.is_dead() or self.defenses_ ↵
breached():
                    GameState.start_screen = True
                if self.win_round():
                    self.next_round()
                self.refresh_screen()
            self.splash_screen()
        pygame.quit()

if __name__ == '__main__':
    pv = Game()
    pv.main_loop()
```

# Pivaders Pt 2: graphics & sound

This time we'll expand our Space Invaders clone to include immersive animation and sound

## What you'll need…

**Raspbian**
www.raspberrypi.org/downloads

**Python**
www.python.org/doc

**Pygame**
www.pygame.org/docs

**Art assets**
opengameart.org

We had great fun creating our basic Space Invaders clone, Pivaders, in the previous guide. Pygame's ability to group, manage and detect collisions thanks to the Sprite class really made a great difference to our project, not just in terms of code length, but in simplicity too. If you missed the first part of the project, you can find the v0.1 code listing on GitHub via **git.io/cBVTBg**, while you can find version v0.2 of the code, including all the images, music and sound effects we've used at **git.io/8QsK-w**.

To help keep our project code manageable and straightforward (as your projects grow keeping your code easy to follow becomes increasingly harder) we integrated a few animation methods into our Game class and opted to use a sprite sheet. Not only does it make it very easy to draw to the screen, but it also keeps the asset count under control and keeps performance levels up, which is especially important for the Raspberry Pi. We hope you have fun using our techniques to add animation and sound to your projects!

### Did you know…

Space Invaders is one of the most cloned games in the world! It makes a great first project for game programmers.

## Setting up dependencies

**01** You'll get much more from the exercise if you download the code (**git.io/8QsK-w**) and use it for reference as you create your own animations and sound effects. Regardless of whether you just want to simply preview and play or walk-through the code to get a better understanding of basic game creation, you're still going to need to satisfy some basic dependencies. The two key requirements here are Pygame and Git, both of which are installed by default on up-to-date Raspbian installations. That's easy!

## Downloading pivaders

**02** Git is a superb version control solution that helps programmers safely store their code and associated files. Not only does it help you retain a full history of changes, it means you can 'clone' entire projects to use and work on from places like **github.com**. To clone the version of the project we created for this tutorial, go to your home folder from the command line (cd ~) and type:

```
git pull  https://github.com/russb78/pivaders.git
```

This creates a folder called pivaders.

## Navigating the project

**03** Within pivaders sits a licence, readme and a second pivaders folder. This contains the main game file, pivaders.py, which launches the application. Within the data folder you'll find subfolders for both graphics and sound assets, as well as the font we've used for the title screen and scores. To take pivaders for a test-drive, simply enter the pivaders subdirectory (cd pivaders/pivaders) and type:

```
python pivaders.py
```

Use the arrow keys to steer left and right and the space bar to shoot. You can quit with the Escape key.

## Code listing (starting from line 87)

```
class Game(object):
    def __init__(self):
        pygame.init()
        pygame.font.init()
        self.clock = pygame.time.Clock()
        self.game_font = pygame.font.Font(
        'data/Orbitracer.ttf', 28)
        self.intro_font = pygame.font.Font(
        'data/Orbitracer.ttf', 72)
        self.screen = pygame.display.set_mode([RES[0], RES[1]])
        self.time = pygame.time.get_ticks()
        self.refresh_rate = 20; self.rounds_won = 0
        self.level_up = 50; self.score = 0
        self.lives = 2
        self.player_group = pygame.sprite.Group()
        self.alien_group = pygame.sprite.Group()
        self.bullet_group = pygame.sprite.Group()
        self.missile_group = pygame.sprite.Group()
        self.barrier_group = pygame.sprite.Group()
        self.all_sprite_list = pygame.sprite.Group()
        self.intro_screen = pygame.image.load(
        'data/graphics/start_screen.jpg').convert()
        self.background = pygame.image.load(
        'data/graphics/Space-Background.jpg').convert()
        pygame.display.set_caption('Pivaders - ESC to exit')
        pygame.mouse.set_visible(False)
        Alien.image = pygame.image.load(
        'data/graphics/Spaceship16.png').convert()
        Alien.image.set_colorkey(WHITE)
        self.ani_pos = 5 # 11 images of ship
        self.ship_sheet = pygame.image.load(
        'data/graphics/ship_sheet_final.png').convert_alpha()
        Player.image = self.ship_sheet.subsurface(
                    self.ani_pos*64, 0, 64, 61)
        self.animate_right = False
        self.animate_left = False
        self.explosion_sheet = pygame.image.load(
        'data/graphics/explosion_new1.png').convert_alpha()
        self.explosion_image = self.explosion_sheet.subsurface(
0, 0, 79, 96)
        self.alien_explosion_sheet = pygame.image.load(
        'data/graphics/alien_explosion.png')
        self.alien_explode_graphics = self.alien_explosion_sheet.↵
subsurface(0, 0, 94, 96)
        self.explode = False
        self.explode_pos = 0; self.alien_explode = False
        self.alien_explode_pos = 0
        pygame.mixer.music.load('data/sound/10_Arpanauts.ogg')
        pygame.mixer.music.play(-1)
        pygame.mixer.music.set_volume(0.7)
        self.bullet_fx = pygame.mixer.Sound(
        'data/sound/medetix__pc-bitcrushed-lazer-beam.ogg')
        self.explosion_fx = pygame.mixer.Sound(
        'data/sound/timgormly__8-bit-explosion.ogg')
        self.explosion_fx.set_volume(0.5)
        self.explodey_alien = []
```

Continued on page 96

## Animation & sound

**04** Compared with the game from last month's tutorial, you'll see it's now a much more dynamic project. The ship now leans into the turns as you change direction and corrects itself when stationary. When you shoot an alien ship, it explodes with several frames of animation and should you take fire, a smaller explosion occurs on your ship. Music, lasers and explosion sound effects also accompany the animations as they happen.

## Finding images to animate

**05** Before we can program anything, it's wise to have assets set up correctly. We've opted to use sprite sheets; these can be found online or created with GIMP with a little practice. They're a mosaic made up of individual 'frames' of equally sized and spaced images representing each frame. We found ours at **opengameart.org**.

## Tweaking assets

**06** While many of the assets you'll find online can be used as is, you may want to import them into an image-editing application like GIMP to configure them to suit your needs. We started with the central ship sprite and centred it into a new window. We set the size and width of the frame and then copy-pasted the other frames either side of it. We ended up with 11 frames of exactly the same size and width in a single document. Pixel-perfect precision on size and width is key, so we can just multiply it to find the next frame.

## Loading the sprite sheet

**07** Since we're inheriting from the Sprite class to create our Player class, we can easily alter how the player looks on screen by changing Player.image. First, we need to load our ship sprite sheet with pygame.image.load(). Since we made our sheet with a transparent background, we can append .convert_alpha() to the end of the line so the ship frames render correctly (without any background). We then use subsurface to set the initial Player.image to the middle ship sprite on the sheet. This is set by self.ani_pos, which has an initial value of 5. Changing this value will alter the ship image drawn to the screen: '0' would draw it leaning fully left, '11' fully to the right.

## Animation flags

**08** Slightly further down the list in the initialising code for the Game class, we also set two flags for our player animation: self.animate_left and self.animate_right. As you'll see in the Control method of our Game class, we use these to 'flag' when we want animations to happen with True and False. It also allows us to 'automatically' animate the player sprite back to its natural resting state (otherwise the ship will continue to look as if it's flying left when it has stopped).

## The animation method

**09** We use flags again in the code for the player: animate_player(). Here we use nested if statements to control the animation and physically set the player image. It states that if the animate_right flag is True and if the current animation position is different to what we want, we incrementally increase the ani_pos variable and set the player's image. The Else statement then animates the ship sprite back to its resting state and the same logic is then applied in the opposite direction.

```
        GameState.end_game = False
        GameState.start_screen = True
        GameState.vector = 0
        GameState.shoot_bullet = False

    def control(self):
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                GameState.start_screen = False
                GameState.end_game = True
            if event.type == pygame.KEYDOWN \
            and event.key == pygame.K_ESCAPE:
                if GameState.start_screen:
                    GameState.start_screen = False
                    GameState.end_game = True
                    self.kill_all()
                else:
                    GameState.start_screen = True
        self.keys = pygame.key.get_pressed()
        if self.keys[pygame.K_LEFT]:
            GameState.vector = -1
            self.animate_left = True
            self.animate_right = False
        elif self.keys[pygame.K_RIGHT]:
            GameState.vector = 1
            self.animate_right = True
            self.animate_left = False
        else:
            GameState.vector = 0
            self.animate_right = False
            self.animate_left = False

        if self.keys[pygame.K_SPACE]:
            if GameState.start_screen:
                GameState.start_screen = False
                self.lives = 2
                self.score = 0
                self.make_player()
                self.make_defenses()
                self.alien_wave(0)
            else:
                GameState.shoot_bullet = True
                self.bullet_fx.play()

    def animate_player(self):
        if self.animate_right:
            if self.ani_pos < 10:
                Player.image = self.ship_sheet.subsurface(
                    self.ani_pos*64, 0, 64, 61)
                self.ani_pos += 1
        else:
            if self.ani_pos > 5:
                self.ani_pos -= 1
                Player.image = self.ship_sheet.subsurface(
                    self.ani_pos*64, 0, 64, 61)

        if self.animate_left:
            if self.ani_pos > 0:
                self.ani_pos -= 1
```

```
                Player.image = self.ship_sheet.subsurface(
                    self.ani_pos*64, 0, 64, 61)
        else:
            if self.ani_pos < 5:
                Player.image = self.ship_sheet.subsurface(
                    self.ani_pos*64, 0, 64, 61)
                self.ani_pos += 1

    def player_explosion(self):
        if self.explode:
            if self.explode_pos < 8:
                self.explosion_image = self.explosion_sheet.
subsurface(0, self.explode_pos*96, 79, 96)
                self.explode_pos += 1
                self.screen.blit(self.explosion_image, [self.player.
rect.x -10, self.player.rect.y - 30])
            else:
                self.explode = False
                self.explode_pos = 0

    def alien_explosion(self):
        if self.alien_explode:
            if self.alien_explode_pos < 9:
                self.alien_explode_graphics = self.alien_
explosion_sheet.subsurface(0, self.alien_explode_pos*96, 94,
96)
                self.alien_explode_pos += 1
                self.screen.blit(self.alien_explode_graphics,
[int(self. explodey_alien[0]) - 50 , int(self.explodey_alien[1]) -
60])
            else:
                self.alien_explode = False
                self.alien_explode_pos = 0
                self.explodey_alien = []

    def splash_screen(self):
        while GameState.start_screen:
            self.kill_all()
            self.screen.blit(self.intro_screen, [0, 0])
            self.screen.blit(self.intro_font.render(
            "PIVADERS", 1, WHITE), (265, 120))
            self.screen.blit(self.game_font.render(
            "PRESS SPACE TO PLAY", 1, WHITE), (274, 191))
            pygame.display.flip()
            self.control()
            self.clock.tick(self.refresh_rate / 2)

    def make_player(self):
        self.player = Player()
```

Find the rest of the code at github.com/russb78/pivaders

"Sprite sheets make it easy to draw to the screen, but it also keeps the asset count down and performance levels up"

## Animating explosions

**10** The player_explosion() and alien_explosion() methods that come after the player animation block in the Game class are similar but simpler executions of the same thing. As we only need to run through the same predefined set of frames (this time vertically), we only need to see if the self.explode and self.alien_explode flags are True before we increment the variables that change the image.

## Adding music

**11** Pygame makes it easy to add a musical score to a project. Just obtain a suitable piece of music in your preferred format (we found ours via freemusicarchive.org) and load it using the Mixer Pygame class. As it's already been initialised via pygame.init(), we can go ahead and load the music. The music.play(-1) requests that the music should start with the app and continue to loop until it quits. If we replaced -1 with 5, the music would loop five times before ending. Learn more about the Mixer class via **www.pygame.org/docs/ref/mixer.html.**

## Using sound effects

**12** Loading and using sounds is similar to how we do so for images in Pygame. First we load the sound effect using a simple assignment. For the laser beam, the initialisation looks like this:

```
self.bullet_fx = pygame.mixer.Sound('location/of/file')
```

Then we simply trigger the sound effect at the appropriate time. In the case of the laser, we want it to play whenever we press the space bar to shoot, so we place it in the Game class's Control method, straight after we raise the shoot_bullet flag. You can get different sounds from **www.freesound.org**.

## What you'll need…

**Python**
www.python.org/doc

**Pygame**
www.pygame.org/docs

**IDLE Python IDE**

**Game assets**

**Code from FileSilo (optional)**

# Make a visual novel game

Bridge the gap between books and videogames by creating an interactive novel with Python

Most people look for a compelling story in modern videogames, and those that don't have one are appearing less and less. A great way to tell a pure story is through the genre of visual novels, and you can make one fairly simply in Python. These interactive novels are an extremely popular form of entertainment in Japan, and usually work by having the player click through a story and make decisions as they go along in order to experience different plot points and endings.

In Python, this is a relatively simple project to create, but with the addition of the Pygame module we can make it easier still, and even more expandable for the future. Pygame adds better support for positioning the images and text, creating display windows and using mouse and keyboard inputs, thereby simplifying the coding process.

We'll be coding this in standard Python 2, so make sure to run it in IDLE 2 and not IDLE 3 while you are writing, testing and coding.

## Get Pygame dependencies

**01** The best way to install Pygame for your system is to compile it. To do this you need to first install the right dependencies. Open up the terminal and install the following packages, which in Ubuntu looks like:

```
$ sudo apt-get install
mercurial python-dev
python-numpy libav-tools
libsdl-image1.2-dev libsdl-
mixer1.2-dev libsdl-ttf2.0-dev
libsmpeg-dev libsdl1.2-dev
libportmidi-dev libswscale-dev
libavformat-dev libavcodec-dev
```

## Get the Pygame code

**02** Next we need to download the code for Pygame direct from the source. Still in the terminal, you can do this by typing in:

```
$ hg clone https://bitbucket.
org/pygame/pygame
```

Which will download it to the folder 'pygame'. Move to that using CD pygame in the terminal so we can continue building it.

## Build the Pygame module

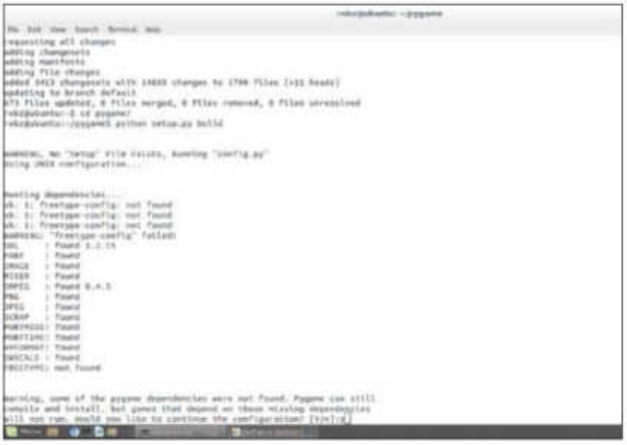
**03** To install it, we need to do it in two steps. First we need to prepare the code to install using the terminal with:

```
$ python setup.py build
```

Once that's finished you can then actually install it with:

```
$ sudo python setup.py install
```

This won't take too long.



## Install in other ways

**04** If the above doesn't work (or is a bit daunting) you can check the website for binary and executable files that will work on other operating systems and Linux distros. Head to http://pygame.org/download.shtml to get the files you need for your specific system, including Windows and OS X. The rest of the tutorial will work in any OS.

## Get the visual novel files

**05** We've uploaded the code to FileSilo, and here we're going to walk you through what we've done to make it work. Download the files for the visual novel and unzip them. The two files we care about for the moment are the visualnovel.py and script.py python files – this is where all the important code is.

## Understand the script file

**06** For the moment the script file is small and literally just holds the script for the game. It's made up of events for the visual novel to move between, line by line, by splitting it up into scenes. This includes the location of each line, the character, the actual line itself and information on how the game flows. These are matrices with the information in, and are completely customisable.

"Pygame adds better support for positioning the images and text"

## How the script relates

**07** In our game, the code pulls in elements from the script file as it goes. We'll explain how that works later, but this also allows us to implement decisions later on to change which direction the game might take you in.

```
import pygame, time, script
pygame.init()
```

## Starting the main game

**08** We don't need many modules for the current state of the visual novel. Here we've imported the new Pygame module, our script as a module and the time module for aesthetic reasons – we're going to have the code pause in bits rather than just instantly change scenes to the next line. We also initialise Pygame with a simple pygame.init()

## Add variables and assets

**09** We add a mixture of information we need to run the novel. We define the size of the display screen to use (1000 pixels wide and 563 high), along with some RGB colours for the code to use. We're also telling Pygame what font to use and how large for certain sections and also loading images for the game.

## Start the game

**10** Pygame works by constantly updating the display with new information. The menu function adds elements to the display (which we've titled screen), like filling it with colour, adding shapes and using blit to add images or in this case text. With a buffer of changes to the screen, update it with the flip() function.

```
screen = pygame.display.set_mode(size)

def menu_screen():
    screen.fill(grey)
    screen.blit(title_font.render('VN SCHOOL', 1, black), (330, 100))
    start_button = pygame.draw.rect(screen, (black), pygame.Rect(410, 310, 180, 55))
    screen.blit(menu_font.render('Start game', 1, white), (425, 310))
    pygame.display.flip()
    while True:
        pygame.event.get()
        click_pos = pygame.mouse.get_pos()
        if (pygame.mouse.get_pressed())[0] == 1 and 410 < click_pos[0] < 590 and 310
            start_game()
```

## See the mouse

**11** As we've created the button as a rectangle and now an image on the menu, we need to recognise when the mouse is hovering over it to know when the button is clicked. First we have to use event.get() to see the mouse in general, then we look for the position with get_pos(). After that, we wait for it to click, see where it clicked (using the co-ordinates of the rectangle) and make a decision after that.

## Start the story

**12** Our start_game function is called when the mouse clicks the right position and we prepare the game, getting the characters, locations and progression through the game script. The rest of this function uses this info to pull in data from the script to make the game flow properly.

## First screen

**13** The first screen is handled differently, and acts to get every element up on the interface – it makes the code take a little less time to process as we begin. The getattr allows us to use the string/integer associated with our place in the story and call upon the relevant scene function from the script file.

We then use an if statement with an iterative function to successively add screen elements to give the illusion that it's building up the first screen. We finish it by advancing the progression value.

"Our next if statement and iteration checks what is different on the next line"

```
elif turn > 0 and click_state[0] == 1:
    line_start = 0
    for i in range (4):
        if line_start == 0 and line[0] != '0':
            print line[0]
            screen.blit(location[line[0]], [0, 0])
            time.sleep(1)
        elif line_start == 1 and line[1] != '0':
            screen.blit(character[line[1]], [377, 113])
            time.sleep(1)
        elif line_start == 2:
            pygame.draw.rect(screen, (grey), pygame.Rect(130, 423, 740, 120))
        elif line_start == 3:
            screen.blit(game_font.render(line[2], 1, white), (135, 430))
            turn += 1
            if  line[3] != '0':
                game_script = line[3]
                time.sleep(0.5)
            game_state = line[4]
            clicked = 0

        line_start += 1
        pygame.display.flip()
```

```
def game():
    global game_running
    while game_running == True:
        menu_screen()

game()
```

## Add variables and assets

**14** Similarly to the way that our original startup code works, our next if statement and iteration checks to see what is different on the next line, and if it moves to a different scene function. In addition, it will also change anything that is different without filling up the buffer more than needed. Where we've made no change is labelled with a 0 in the scripts.

## The starting function

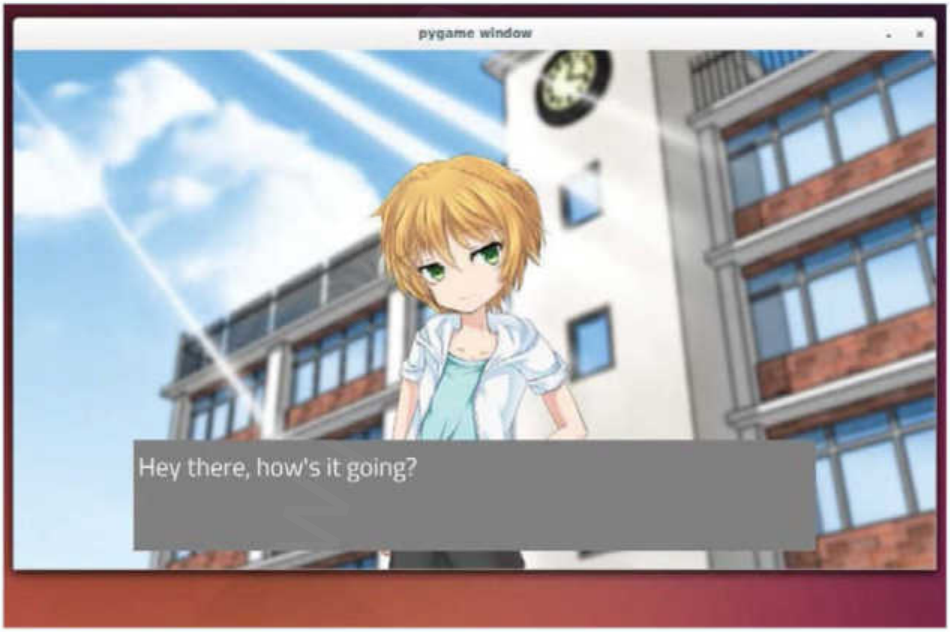
**15** We finish our code bit with a simple function that starts off the entire game. This is just to encapsulate the entire code and allows us to add different ways of turning it off in the future. IDLE when running the file will load everything up and then run the game() function at the end – this is similar to how you can add a __main__ function at the end which will start the code in the command line.

## Expand your code

**16** The code written is very expandable, allowing you to add decisions that are logged to take you to different scenes (or routes in visual novel terminology) and make your game feel more interactive. This would not require much more code than the if statements, and it would also be a good way for you to look into adding graphical buttons to click and use the collide function.

## Move the assets

**17** Currently the code has the script-specific assets in the main visualnovel file. These can be moved to the script, allowing you to make the visualnovel file much more modular so that can you have multiple scripts with different assets to load at startup.



# Use Python with Pi

## Amazing creations with Python code and Raspberry Pi

From the tutorials up to this point, you'll have a firm grounding in Python. Now we're going to add the Raspberry Pi computer. You'll discover exciting projects such as sending SMS texts from your Raspberry Pi to a mobile phone (p.102), a voice synthesiser (p.114), and this quadcopter (right, p.116). You'll also learn how to code a Twitter bot (p.122), and control an LED with Python (p.124).

## What you'll need…

**A Raspberry Pi with all necessary peripherals**

**SD card with latest Debian image for Pi**
www.raspberrypi.org/downloads

# Using Python on Raspberry Pi

Program in Python with the Raspberry Pi, and lay the foundations for all your future projects

This tutorial follows on from the one last issue: 'Setting up the Raspberry Pi', where we showed you how to prepare your SD card for use with the Raspberry Pi. The beauty of using an SD card image is that the operating system is ready to go and a development environment is already configured for us.

We'll use a lightweight integrated development environment (IDE) called Geany for our Python development. Geany provides a friendlier interface compared to text-based editors such as nano to make it easier to get into the swing of things. This tutorial will cover topics such as:

- Basic arithmetic
- Comparison operators, for example 'equal to' and 'not equal to'
- Control structures, for example loops and if statements

By the end, we'll have an advanced version of our 'hello world' application. Let's dive straight in…


## Staying organised

**01** We don't want to have messy folders on our new Pi, so let's go to the file manager and organise ourselves. Open the file manager by clicking the icon next to the menu icon on the bottom left of the screen. Create a new folder by right-clicking and selecting New>Folder, then type a name and click OK. We created a folder called Python, and inside that created a folder called Hello World v2.

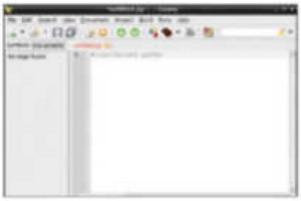
It's good practice to describe what the program's purpose is at the top of the file. This will help you out when working on larger projects with multiple files

It's important to think about data types. We convert the number to decimal to make sure that we don't lose any decimal numbers during arithmetic

The stopping condition for a while loop has to be satisfied at some point in the code; otherwise the loop will never end!

The print function can only accept string data types, so we need to convert any variables with a number data type to a string before we can print them to the screen

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

# An advanced Hello World program that will demonstrate the basics of
# programming in Python via a series of examples. Created by Liam Fraser
# for Linux User and Developer - 22/04/2012.

# Import the sys libary for the sys.exit function
import sys
# Import everything from the Decimal libary
from decimal import *

# Get the users first name and output a welcome message
firstName = raw_input("Please enter your first name: ")
print("Welcome " + firstName + "\n")

# Ask the user for a number and square it, double it and halve it
number = raw_input("Please enter a number: ")
number = Decimal(number)
numberHalved = number / 2
numberDoubled = number * 2
numberSquared = number * number

# Print out the values we just worked out, converting each float value
# to a string
print("The result of halving that number: " + str(numberHalved))
print("The result of doubling that number: " + str(numberDoubled))
print("The result of squaring that number: " + str(numberSquared))
print("")

# The stopping condition for a while loop
yesOrNo = False

# A while loop that will run untill a user enters either "yes" or "no"
while yesOrNo == False:
    result = raw_input("Do you want to continue? (yes/no) ")

    if result == "yes" or result == "no":
        yesOrNo = True
    else:
        print("Error, please type yes or no" + "\n")

# Deal with the result
if result == "yes":
    print("\nContinuing")
else:
    print("\nExiting")
    sys.exit()

# Create the count which will be a stopping condition for a while loop
count = 1

# Use a while loop to add 5 to the number and output the value each time
print ("Incrementing the number by 5\n")

while count <= 5:
    number += 1
    print("number + " + str(count) + " = " + str(number))

    # Increment the count
    count += 1

# Finish off by printing that we are exiting
print("\nExiting")
```

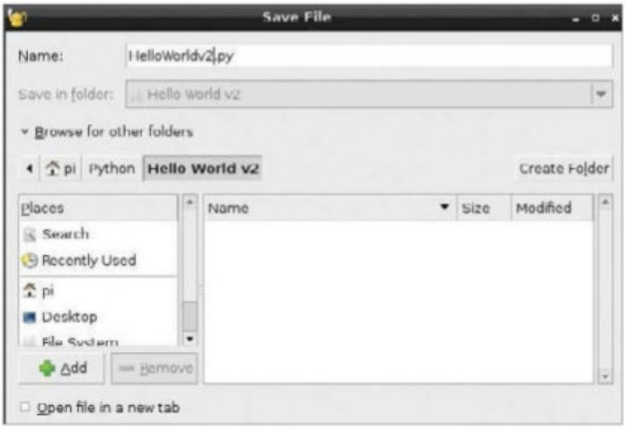


## Starting Geany

**02** Start Geany by going to the LXDE menu and going to Programs. From here, select Geany. Once you're in the Geany interface, create a new Python file from a template by selecting 'New (with template)>main.py'. Delete everything in this template apart from the first line: #!/usr/bin/env python. This line is important because it means you can run the code from the command line and the Bash shell will know to open it with the Python interpreter.

## Saving your work

**03** It's always a good idea to keep saving your work with Ctrl+S as you program, because it would be a shame to lose anything you've been working on. To save your file for the first time, either press Ctrl+S or go to the File menu and select Save. Give the file a sensible name and save it in the tidy folder structure you created before. It's a good habit to be well organised when programming, because it makes things much easier when your projects become bigger and more complicated.




## Setting it up

**04** Having detailed comments in your code is important because it allows you to note down things you find confusing and document complex procedures. If another programmer has to work with your code in the future, they'll be extremely grateful. Start by adding a comment with a description of what the program will do and your name. All comment lines start with a hash (#) and are not interpreted as code by the Python interpreter. We import the sys library so we can use the sys.exit function to close the program later on. We also import everything from the decimal library because we want to make use of the decimal type.

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

# An advanced Hello World
# programming in Python v
# for Linux User and Deve

# Import the sys libary f
import sys
# Import everything from
from decimal import *
```

"It's a good habit to be well organised when programming"

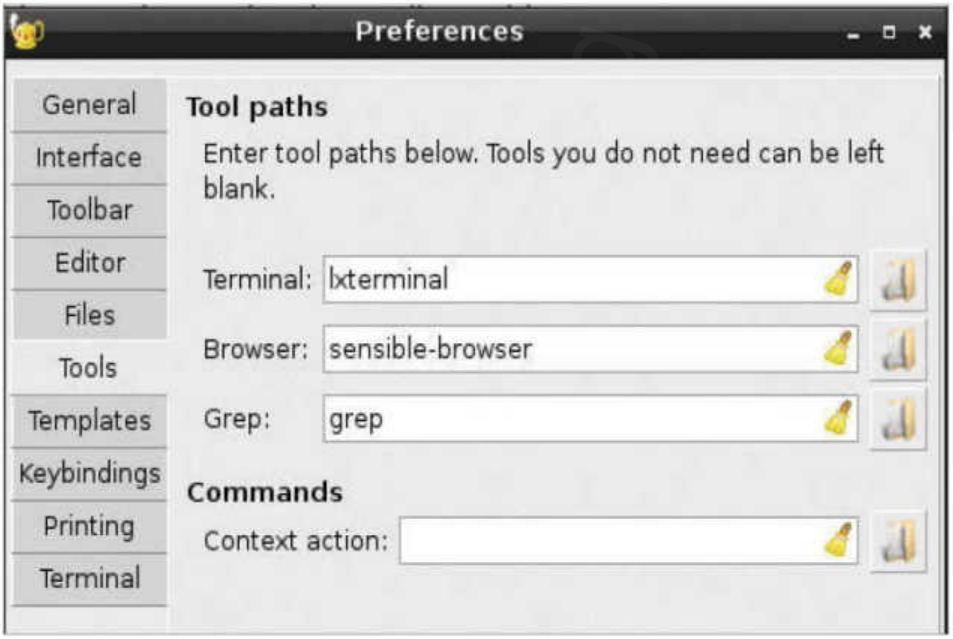
## Variables

**05** A variable is data that is stored in memory and can be accessed via a name. Our program is going to start by asking for your first name, store that in a variable and then print out a welcome message. We're going to add a comment that explains this and create a variable called firstName. Notice how we've capitalised the first letter of the second word to make it easier to read.

We want the firstName variable to hold the value returned by a function called raw_input, that will ask the user for input. The question is passed into the print function within brackets, and because this is a string it is enclosed within quotation marks. A string type is basically a collection of characters. Note the extra space we've added after the colon because the user types their input straight after this question.

## Printing a message

**06** Now that we have a value in firstName, we need to output a welcome message to the screen. We print to the screen in Python using the print function. The print function is followed by a pair of brackets which enclose the values to print. When using the addition operator with strings, they are joined together. Note how firstName doesn't need to be enclosed by quotation marks because it is the name of a variable. If it was enclosed in quotation marks, the text firstName would be output. We finish off by adding a '\n' character (new line character) to our output to leave one blank line before we start our next example.





## Fixing a small issue

**07** The Debian image that we're currently using has a small misconfiguration issue in Geany. You'll know if you have this problem by trying to run your program with either the F5 key or going to the Build menu and selecting Execute. If the issue is present then nothing will happen and you'll see a message saying 'Could not find terminal: xterm'. Not to worry, it's easy to fix. Go to the Edit menu and then select Preferences. Go to the Tools tab and change the value for Terminal from xterm to lxterminal.

## Testing our program

**08** Now we've done that part, why not test it? It's worth noting that you have to save before running the program, or anything you've done since you last saved won't be interpreted by Python. Run the program by pressing the F5 key. Input your name by typing it and then pressing the Enter key. Once you have done this, you'll see a welcome message. If the program exits with the code 0 then everything was run successfully. Press Enter to close the terminal.

## Working with numbers

**09** We're going to ask the user for a number by basically repeating the first couple of lines we did. Once the user gives us a number, we'll halve, square and double it. The raw_input function returns the value that the user input as a string. A string is a text-based value so we can't perform arithmetic on it. The integer type in Python can only store whole numbers whereas the decimal type can store numbers with decimals. We're going to do something called a type cast, which basically converts a value with one type to another type. We're going to convert our number string to a decimal value because it's likely that decimals will be involved if we are halving numbers. If the number was of an integer type, any decimal values would simply be cut off the end, without any rounding. This is called truncation.

## Performing arithmetic

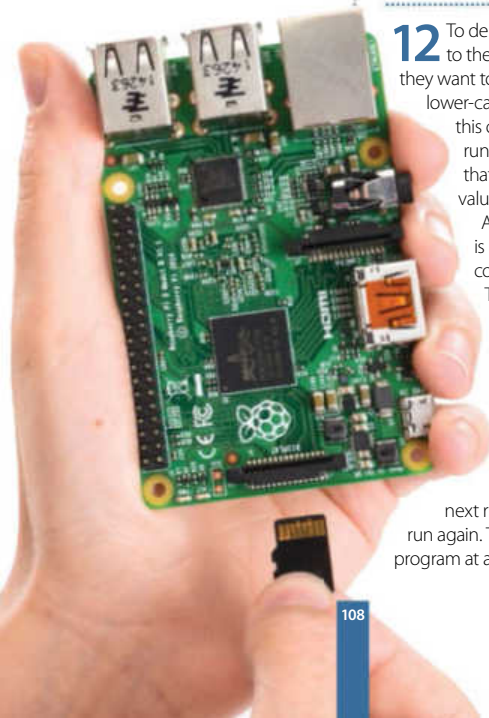
**10** The main arithmetic operators in Python are + - / *, the latter two being divide and multiply respectively. We've created three new variables called numberHalved, numberDoubled and numberSquared. Notice that we don't need to specify that they should be decimal because Python gives a type to its variables from the type of their initial value. The number variable is a decimal type, so all values returned from performing arithmetic on that number will also be of a decimal type.

```
23  # Print out the values we just worked out, converting each float value
24  # to a string
25  print("The result of halving that number: " + str(numberHalved))
26  print("The result of doubling that number: " + str(numberDoubled))
27  print("The result of squaring that number: " + str(numberSquared))
28  print("")
```

## Printing our numbers

**11** Now that we have performed our arithmetic, we need to print the results using the print function. The print function only accepts string values passed to it. This means that we need to convert each decimal value to a string using the str() function before they can be printed. We're using a print statement with nothing between the quotation marks to print one blank line. This works because the print function always adds a new line at the end of its output unless told otherwise, so printing an empty string just prints a new line.

**Below** The Raspberry Pi takes the 'Pi' part of its name from its compatibility with the Python programming language



## Input validation with While loops and If statements

**12** To demonstrate a while loop and if statements, we will output a question to the user that requires a yes or no answer. We're going to ask them if they want to continue – and for this we require either a lower-case 'yes', or a lower-case 'no'. A while loop is a loop that runs until a condition is met. In this case, we will create a variable called yesOrNo and the while loop will run while yesOrNo is false. The yesOrNo variable will be a Boolean type that can be either True or False. The variable will be initialised with a value of False, or the while loop will not run.

A while loop has the format 'while [condition]:' – where any code that is part of the while loop needs to be indented in the lines below the colon. Any code that is not indented will not be part of the while loop. This is the same for an if statement. The condition is checked with the comparison operator '=='. A single '=' is an assignment operator whereas a double equals is a comparison operator. Another common comparison operator is '!=' – which means 'not equal to'.

We create a variable called 'result', which holds the result of the question, do you want to continue? We then check this result is valid with an if statement. Notice the 'or' operator which allows two conditions to be tested. If the user inputs a correct value then we set yesOrNo to True, which stops the while loop on the next run. Otherwise, we output an error message and the while loop will run again. The user can use the Ctrl+C command at the terminal to exit the program at any time.

```
42    # Deal with the result
43  if result == "yes":
44      print("\nContinuing")
45  else:
46      print("\nExiting")
47      sys.exit()
```

## Continue or exit?

**13** Next we will deal with the result that was stored during the while loop with if statements. If the user typed 'yes' then we will print 'Continuing'. Otherwise, we will print 'Exiting' and then call the sys.exit function. You don't have to do anything else for the program to continue because it will simply carry on if the sys.exit function wasn't called. This code also shows that the newline character \n can be used anywhere in a string, not just in separate quotation marks like above.

## Loops with numbers

**14** We'll be using a while loop that uses a number and a <= (less than or equal to) operator as its stopping condition. The while loop will be used to increment the number by 1, printing the change on each loop until the stopping condition is met. The count variable allows us to know exactly how many times we have been through the while loop.

## Incrementing numbers with a loop

**15** The while loop will run until the count is 6, meaning that it will run for a total of 5 times because the count begins at 1. On each run, the while loop increments the number variable and then prints what is being added to the original number, followed by the result. Finally, the count is incremented.

"The print function always adds a new line at the end of its output"

```
52    # Use a while loop to add 5 to the number and output the value each time
53    print ("Incrementing the number by 5\n")
54
55  while count <= 5:
56      number += 1
57      print("number + " + str(count) + " = " + str(number))
58
59      # Increment the count
60      count += 1
```

## Finishing off

**16** The final step is to print that the program is exiting. This is the last line and we don't have to do anything else because Python simply finishes when there are no more lines to interpret.

```
Please enter your first name: Liam
Welcome Liam

Please enter a number: 12.12
The result of halving that number: 6.06
The result of doubling that number: 24.24
The result of squaring that number: 146.8944

Do you want to continue? (yes/no) yes

Continuing
Incrementing the number by 5

number + 1 = 13.12
number + 2 = 14.12
number + 3 = 15.12
number + 4 = 16.12
number + 5 = 17.12

Exiting
```

## Admire your work

**17** Now that we've finished coding, save any changes you have made and run your program with the F5 key.

## What you'll need…

**Raspberry Pi**

**Twilio account**

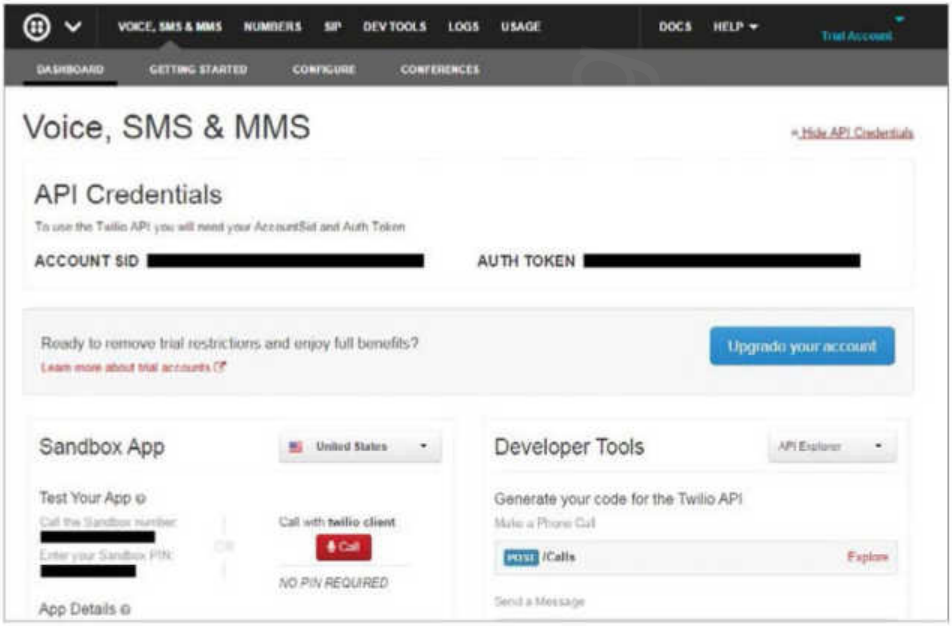
# Send an SMS from your Pi

Create a program combining Twilio and simple Python code and send an SMS from your Pi

Text messaging, or SMS (Short Message Service), has become a staple of everyday communication. For many of us, not a day will go by without sending a text message. What began life as a 40 pence message service is now offered by most tariff providers as an unlimited service.

Twilio, a cloud communications company, enables you to send SMS messages for free from your Raspberry Pi to a mobile phone using just six lines of code. So, you no longer need to be chained to your mobile while you work, and can focus on one screen rather than two!


**Left** With this method, you could get your Pi to drop you a text when it finishes running a script




**Above** You will be able to find your AccountSid and your Auth Token on the Twilio dashboard

## Set up your Twilio account

**01** The first step of this project is to register for a Twilio account and Twilio number. This is free and will enable you to send an SMS to a registered, verified phone. Once signed up, you will receive a verification code via SMS to the registered phone. When prompted, enter this onto the Twilio site to authenticate your account and phone. Go to twilio.com/try-twilio and create your account.

## Register and verify mobile numbers

**02** Your Twilio account is a trial account (unless you pay the upgrade fee), which means you can only send and receive communications from a validated phone number. Enter the phone number of the mobile that you want to verify, ensuring that you select the correct country code. Twilio will text you a verification code. Enter this code into the website form and press submit.

## The dashboard

**03** Once registered and logged in, visit the dashboard page, which will display your AccountSid and your Auth Token. These are both required to use the Twilio REST. Keep these secure and private, but be sure to make a note of them as you will need them for your Python program later.

## REST

REST stands for Representational State Transfer. (It is sometimes spelt "ReST".) It relies on a stateless, client-server, cacheable communications protocol – and in virtually all cases, the HTTP protocol is used. REST is an architecture style for designing networked applications.

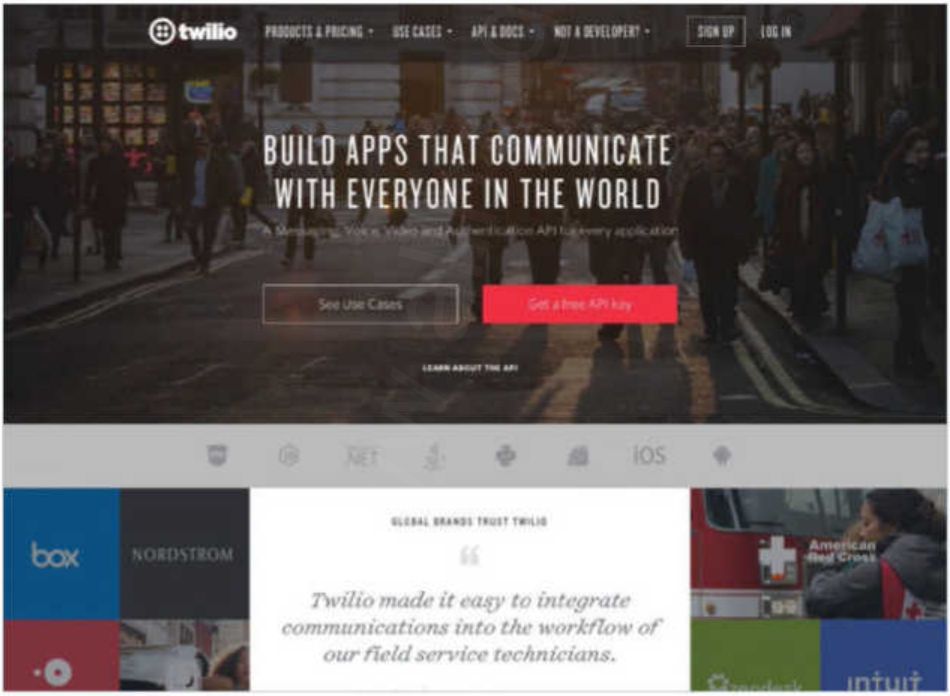
## Install the software

**04** Boot up your Raspberry Pi and connect it to the Internet. Before you install the Twilio software, it is worth updating and upgrading your Pi. In the LX Terminal, type sudo apt-get update, then sudo apt-get upgrade. Once complete, type sudo easy_install twilio or sudo pip install twilio to install the software. (If you need to install pip, type sudo apt-get install python-pip python-dev, press Enter, then type sudo pip install -U pip.)

## Twilio authentication

**05** Now you are ready to create the SMS program that will send the text message to your mobile phone. Open your Python editor and import the Twilio REST libraries (line one, below). Next, add your AccountSid and Auth Token, replacing the X with yours, as you will find on your dashboard:

```
from twilio.rest import TwilioRestClient
account_sid = "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
                    # Enter Yours
auth_token = "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
                    # Enter Yours
client = TwilioRestClient(account_sid, auth_token)
```




**Below** Twilio, whose website is pictured, has been used by large corporations like Coca Cola, Uber and Nordstrom

"Twilio provides a wide range of API codes and reference documents to create other communication programs"

## Create your message

**06** You will probably want to be able to change your text messages rather than send the same one. Create a new variable in your program called message. This will prompt you to enter the phrase that you want to send to the mobile phone. When the program runs, this is the message that will be sent:

```
message = raw_input("Please enter your message")
```

## Add your numbers

**07** To send the message, you need to add the code line below and your two phone numbers. The first number is your mobile phone number, which is registered and validated with Twilio (Step 2). The second number is your Twilio account number, which can be retrieved from your dashboard page under 'Call the Sandbox number'. Change the Sandbox number to your country location and remember to add the international country code.

```
message = client.messages.create(to="+44YOURMOBNUMBER",
from_="+44YOURTWILIONUMBER", body=message)
```

## Send the message

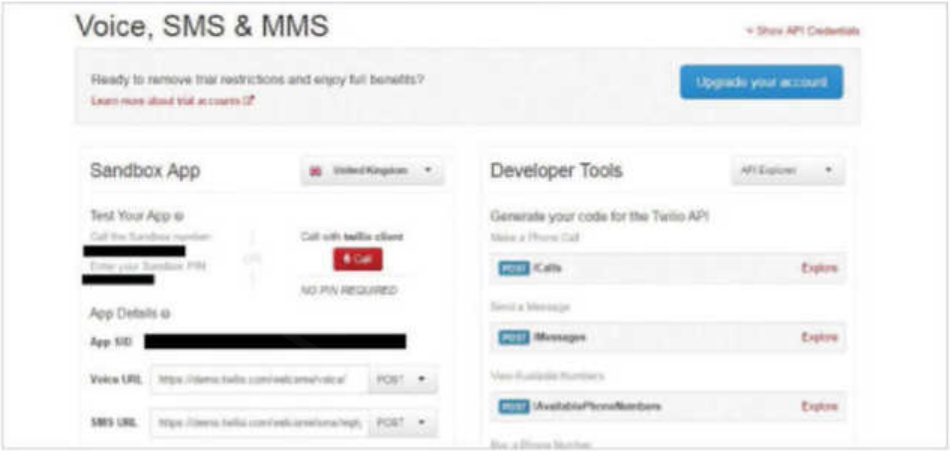
**08** Now send your message. The code below is not required, but useful to indicate your message has been sent. Add the lines and save your program. Ensure your Raspberry Pi is connected to the Internet and that your mobile is on, then run your program. You have just texted from your Raspberry Pi!

```
print message.sid
print "Your message is being sent"
print "Check your phone!"
```

## Other API and codes

**09** Twilio provides a wide range of API codes and reference documents to create other communication programs beyond sending SMS, such as making phone calls, recording your calls, and retrieving data including caller IDs and call duration.

The API also complements a wide range of other programming languages, including Ruby, PHP, Java and Node.js (twilio.com/api).



## What you'll need…

**Portable USB speakers**
**python-espeak module**
**eSpeak**
**Raspbian (latest image)**

### Did you know…

Using eSpeak you can control the way the words are spoken to add emphasis or make the voice sound different

# Voice synthesizer

Add the power of speech to your Raspberry Pi projects with the versatile eSpeak Python library

We've shown how the Raspberry Pi can be used to power all kinds of projects, but as a tiny computer it can also be the centre of an Internet of Things in your house too. For these reasons and more, using the Raspberry Pi for text-to-voice commands could be just what you're looking for. Due to the Debian base of Raspbian, the powerful eSpeak library is easily available for anyone looking to make use of it. There's also a module that allows you to use eSpeak in Python, going beyond the standard command-line prompts so you can perform automation tasks.

## Everything you'll need

**01** We'll install everything we plan to use in this tutorial at once. This includes the eSpeak library and the Python modules we need to show it off. Open the terminal and install with:

```
$ sudo apt-get install espeak python-espeak python-tk
```

## Pi's first words

**02** The eSpeak library is pretty simple to use – to get it to just say something, type in the terminal:

```
$ espeak "[message]"
```

This will use the library's defaults to read whatever is written in the message, with decent clarity. Though this simple command is fun, there's much more you can do…

## Say some more

**03** You can change the way eSpeak will read text with a number of different options, such as gender, read speed and even the way it pronounces syllables. For example, writing the command like so:

```
$ espeak -ven+f3 -k5 -s150 "[message]"
```

…will turn the voice female, emphasise capital letters and make the reading slower.

## Taking command with Python

**04** The most basic way to use eSpeak in Python is to use **subprocess**. Import it, then use:

```
subprocess.call(["espeak", "[options 1]", "[option 2]",..."[option n]", "[your message here]"])
```

## The native tongue

**05** The Python eSpeak module is quite simple to use to just convert some text to speech. Try this sample code:

```
from espeak import espeak
espeak.synth("[message]")
```

You can then incorporate this into Python, like you would any other module, for automation.

## A voice synthesiser

**06** Using the code listing, we're creating a simple interface with Tkinter with some predetermined voice buttons and a custom entry method. We're showing how the eSpeak module can be manipulated to change its output. This can be used for reading tweets or automated messages. Have fun!

## Full code listing

Import the necessary eSspeak and GUI modules, as well as the module to find out the time

Define the different functions that the interface will use, including a simple fixed message, telling the time, and a custom message

Create the basic window with Tkinter for your interface, as well as creating the variable for text entry

The text entry appends to the variable we created, and each button calls a specific function that we defined above in the code

```
from espeak import espeak
from Tkinter import *
from datetime import datetime

def hello_world():
    espeak.synth("Hello World")

def time_now():
    t = datetime.now().strftime("%k %M")
    espeak.synth("The time is %s"%t)

def read_text():
    text_to_read = input_text.get()
    espeak.synth(text_to_read)

root = Tk()
root.title("Voice box")
input_text = StringVar()
box = Frame(root, height = 200, width = 500)
box.pack_propagate(0)
box.pack(padx = 5, pady = 5)

Label(box, text="Enter text").pack()
entry_text = Entry(box, exportselection = 0, textvariable = input_text)
entry_text.pack()
entry_ready = Button(box, text = "Read this", command = read_text)
entry_ready.pack()

hello_button = Button(box, text = "Hello World", command = hello_world)
hello_button.pack()
time_button = Button(box, text = "What's the time?", command = time_now)
time_button.pack()

root.mainloop()
```

"There's even a module that allows you to use eSpeak in Python, so you can perform automated tasks"

## What you'll need…

**Raspbian**
www.raspberrypi.org/downloads

**Python**
www.python.org/doc

## Did you know…

Andy Baker, the writer of this article, keeps a blog at **blog.pistuffing.co.uk** that gives great insight into his project.

# Program a quadcopter

How do you push the limits of the Pi? You give it wings. Andy Baker shows us how it's done…

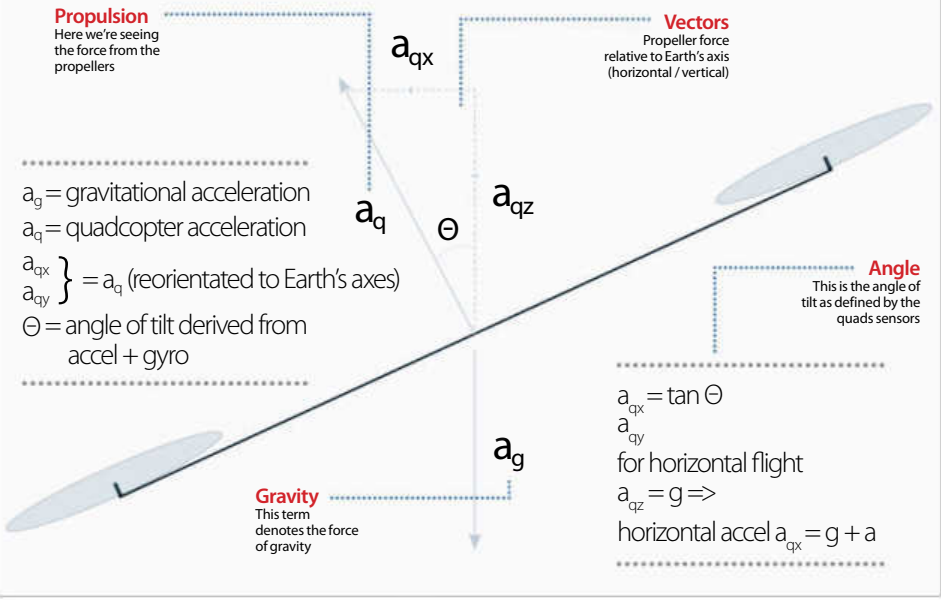
The Raspberry Pi is a fantastic project board. Since we love a challenge, we set out looking for something that would really push the limits of our hacking and coding skills. We chose a quadcopter.

This article uses the Python code available online as a guide through what's needed to build a quadcopter, metaphorically bolting it together so that by the end, you don't just understand the code but also the interaction with the real-world to enable you to build your own quadcopter with confidence. You can find Andy's code on his blog.

**Right** Put your Raspberry Pi in the sky with our expert coding guide!

## Sensor viewpoint diagram

How sensors in the quadcopter's point of view are converted to the Earth (horizontal/vertical) viewpoint to provide horizontal motion




## Interpreter

**01** The command interpreter converts a series of commands either from a radio control or programmed into the code itself. The commands combine the direction and speed compared to the horizon that the user want the quadcopter to follow. The code converts these commands into a series of targets for vertical speed, horizontal speed and yaw speed – any command from a pair of joysticks can be broken down into a set of these targets.

## Inputs

**02** The inputs to the quadcopter come from a series of electronic sensors providing information about its movement in the air. The main two are an accelerometer which measures acceleration force (including gravity) in the three axes of the quadwcopter, and a gyroscope which measures the angular speed with which the quadcopter is pitching (nose/tail up and down), rolling (left/right side up and down), and yawing (spinning clockwise and anticlockwise around the central axis of the quadcopter itself).

## Axes

**03** The accelerometer is relative to the orientation of quadcopter axes, but the command targets are relative to the Earth's axes – the horizon and gravity. To convert the sensor output between the quadcopter axes and the Earth axes requires the use of a bit of mathematics, specifically trigonometry. You also need knowledge of the tilt angles in pitch and roll axes of the quadcopter with respect to the Earth. It's pretty complicated stuff, but the diagrams on these pages should help you understand the logic.

**Above** Kits are available as ready-to-fly (RTF) if you just want the joy of flight, but where's the challenge in that? We started with an almost-ready-to-fly (ARF) kit – the DJI Flame Wheel F450 – all the hardware, but none of the control electronics or software. Many enthusiasts have created DIY quadcopters using Arduino microcontrollers, so we knew a DIY build was possible, but very few, if any, have successfully used the Raspberry Pi

**Did you know…**

You can buy Ready To Fly (RTF) quadcopter kits from a lot of online retails. Prices start from £150 / $200, but rise quickly!

## Angles

**04** Both the accelerometer and gyro can provide this angle information, but both have flaws. The accelerometer output can be used to calculate the angle by using the Euler algorithm. However, the accelerometer output is plagued by noise from the motors/propellers, meaning a single reading can be hugely inaccurate; on the plus side, the average reading remains accurate over time.

In contrast, the gyro output does not suffer from the noise, but since it is the angular speed being measured, it needs to be integrated over time to find the absolute angle of the quadcopter in comparison to the horizon. Rounding errors in the integration lead to ever increasing errors over time, ultimately curtailing the maximum length of a flight.

## Filter

**05** Although independently they are both flawed, they can be merged mathematically such that each compensates for the flaws in the other, resulting in a noise-free, long-term accurate reading. There are many versions of these mathematical noise/drift filters. The best common one is by Kalman; the one we've chosen is slightly less accurate, but easier to understand and therefore to code: the complementary filter.

Now with an accurate angle in hand, it's possible to convert accelerometer sensor data to inputs relative to the Earth's axes and work out how fast the quadcopter is moving up, down, left, right and forwards and backwards compared to the targets that have been set.

## PIDs

**06** So we now have a target for what we want the quadcopter to do, and an input for what it's doing, and some motors to close the gap between the two; all we need now is a way to join these together. A direct mathematical algorithm is nigh on impossible – accurate weight of the quadcopter, power per rotation of each blade, weight imbalance etc would need to be incorporated into the equation. And yet none of these factors is stable: during flights (and crashes!), blades get damaged, batteries move in the frame, grass/mud/moisture changes the weight of the 'copter, humidity and altitude would need to be accounted for. Hopefully it's clear this approach simply won't fly.

Instead, an estimation method is used with feedback from the sensors to fine-tune that estimate. Because the estimation/feedback code loop spins at over 100 times a second, this approach can react to 'errors' very quickly indeed, and yet it knows nothing about all the factors which it is compensating for – that's all handled blindly by the feedback; this is the PID algorithm. It takes the target, subtracts the feedback input, resulting in the error. The error is then processed via a Proportional, Integral and a Differential algorithm to produce the output.

## Blender

**07** The outputs are applied to each ESC in turn: the vertical speed output is applied equally to all blades; the pitch rate output is split 50/50 subtracting from the front blades and adding to the back, producing the pitch. Roll is handled similarly. Yaw too is handled in a similar way, but applied to diagonal blades which spin in the same direction.

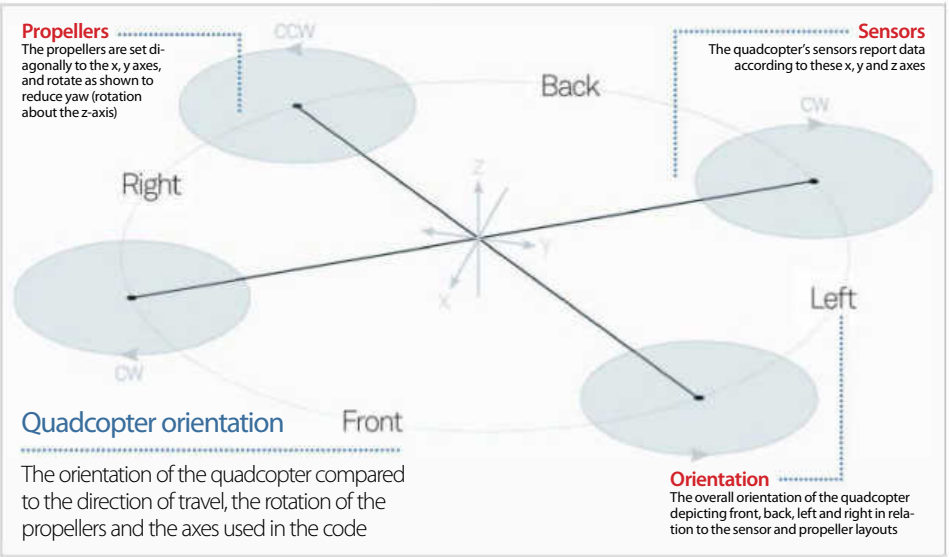
These ESC-specific outputs are then converted to a PWM signal to feed to the hardware ESCs with the updated propeller/motor speeds.

## Understanding quadcopters...

Although this article focuses on software, a very basic background in the hardware from the kit is necessary to provide context.

A quadcopter has four propellers (hence the name) pointing upwards to the sky, each attached to its own brushless DC motor at one of the four corners of (usually) a square frame. Two motors spin clockwise, two anticlockwise, to minimise angular momentum of the quadcopter in flight.

Each motor is driven independently by an electronic speed controller (ESC). The motors themselves have three sets of coils (phases), and the ESCs convert a pulse-width-modulation (PWM) control signal from software/hardware to the three phase high-current output to drive the motors at a speed determined by the control signal. The power for the ESCs and everything else on the system comes from a Lithium Polymer battery (LiPo) rated at 12V, 3300mA with peak surge current of 100A – herein lies the power!




### Quadcopter orientation

The orientation of the quadcopter compared to the direction of travel, the rotation of the propellers and the axes used in the code

## Code and reality

**08** In this code, there are nine PIDs in total. In the horizontal plane, for both the X and Y axes, the horizontal speed PID converts the user-defined desired speed to required horizontal acceleration/ angle of tilt; the angles PID then converts this desired tilt angle to desired tilt rate which the rotation speed PID converts to changes in motors speeds fed to the front/back or left/right motors for pitch/roll respectively.

In the vertical direction, a single PID converts the desired rate of ascent/descent to the acceleration output applied to each plate equally.

Finally, prevention of yaw (like a spinning top) uses two PIDs – one to set the desired angle of yaw, set to 0, and one to set the yaw rotation speed. The output of these is fed to the diagonally opposing motors which spin in the same direction. The most critical of the nine are pitch/roll/yaw stability. These ensure that whatever other requirements enforced by other PIDs and external factors, the quadcopter is stable in achieving those other targets; without this stability, the rest of the PIDs cannot work. Pitch is controlled by relative speed differences between the front and back propellers; roll by left and right differences, and yaw by clockwise/anticlockwise differences from the corresponding PIDs' outputs. The net outputs of all three PIDs are then applied to the appropriate combination of motors' PWM channels to set the individual pulse widths.

With stability assured, some level of take-off, hover and landing can be achieved using the vertical speed PID. Placing the quadcopter on a horizontal surface, set the target to 0.5 m/s and off she zooms into the air, while the stability PID ensures that the horizontal attitude on take-off is maintained throughout the short flight, hover and landing.

Up to this stage, the PIDs are independent. But what about for horizontal movement target, and suppression of drifting in the wind?

This is where things get event more complicated. Taking the drift suppression first, a quadcopter in a headwind will drift backwards due to the force applied by the wind. To compensate for this, it must tilt nose down at some angle so that some of the propellers' thrust is applied horizontally to counteract the wind. In doing so, some of the power keeping the 'copter hovering at a fixed height is now battling the wind; unless the overall power is increased, the 'copter will start descending.

Horizontal movement is more complex still. The target is to move forwards at say 1 metre per second. Initially the requirement is similar to the headwind compensation – nose down plus increased power will apply a forward force leading to forward acceleration. But once that horizontal speed is attained, the quadcopter needs to level off to stop the acceleration, but at the same time, friction in the air will slow the movement. So there's a dynamic tilting fore/aft to maintain this stable forward velocity.

Both wind-drift suppression and controlled horizontal movement use something called nested PIDs; the X and Y axes horizontal speed PIDs' outputs are used as the pitch and roll angle PIDs targets; their output feeds the pitch and roll rate PIDs to ensure stability while meeting those angular targets. The sensor feedback ensures that as the desired horizontal speed is approached, the horizontal speed PID errors shrink, reducing the targets for the angular pitch PID, thus bringing the quadcopters nose back up to horizontal again.

Hopefully it now becomes clearer why accurate angle tracking is critical: in the nose-down, headwind example, the input to the vertical speed PID from the sensors is reduced by the cosine of the measured angle of 'copter tilt with respect to the horizon.

Similarly, X and Y axis speed PID sensor inputs need compensating by pitch and roll angles when comparing target speeds against accelerometer readings. Don't forget to check the diagrams in the article for clear, graphical representations of many of the themes we're covering.

"Taking the drift suppression first, a quadcopter in a headwind will drift backwards due to the force applied by the wind"

## Experimentation and tuning

**09** While the code accurately reflects everything we've described here, there's one critical set of steps which can only be found through live testing; these are the PID gains. For each PID running, there is an independent Proportional, Integral and Differential gain that can only be found with estimation/experimentation. The results for every quadcopter will be different. Luckily there is a relatively safe way to proceed.

First, find the PWM take-off speed: this is done by sitting your quadcopter on the ground and slowly increasing the PWM value until she starts looking light-footed – for your expert, this was about the 1590us pulse width (or 1000us + 590us, as shown in the code).

Next, sorting out the stability PIDs – assuming your quadcopter is square and its balance is roughly central, then the result of pitch tuning also applies to yaw tuning. For pitch tuning, disable two diagonally opposed motors and rest these on a surface – the quadcopter sits horizontal in between. Power up the dangling motors' PWM to just under take-off speed (1550us pulse width in our expert's case). Does the quad rock manically, wobble in some pretence of control, self-right when nudged, or do nothing? Tweak the P gain accordingly. Once P gain is good, add a touch of I gain – this will ensure return to 0 as well as stability. D gain is optional, but adds firmness and crisp response. Tapping a D-gain stable quad is like knocking on a table – it doesn't move.

Vertical speed PID can be guesstimated. 1590us is taking off; desired take-off speed is 0.5m/s so a P gain of 100 is okay. No I or D gain needed. With that a real take-off, hover and landing are safe, which is good as these are the only way to tune the directional PIDs. Just be cautious here – excessive gains lead to quadcopters slamming into walls or performing somersaults in mid-air before powering themselves into the ground. Best executed outside in a large open field/garden/park where the ground is soft after overnight rain!

There isn't a shortcut to this, so just accept there will be crashes and damage and enjoy the carnage as best you can!

Assuming all the above has gone to plan, then you have a quadcopter that takes off, hovers and lands even in breezy conditions. Next step is to add a remote control, but we'll save that for another day!

## What you'll need…

**Internet connectivity**

**Latest version of Raspbian**
www.raspberrypi.org/downloads

# Code your own Twitter bot

Create your very own Twitter bot that can retweet chunks of wisdom from others

Twitter is a useful way of sharing information with the world and it's our favourite method of giving our views quickly and conveniently. Many millions of people use the microblogging platform from their computers, mobile devices and possibly even have it on their televisions.

You don't need to keep pressing that retweet button, though. With a sprinkling of Python, you can have your Raspberry Pi do it for you. Here's how to create your own Twitter bot…

## Installing the required software

**01** Log into the Raspbian system with the username Pi and the password raspberry. Get the latest package lists using the command sudo apt-get update. Then install the Python Package installer using sudo apt-get install python-pip. Once you've done that, run sudo pip install twython to install the Twitter library we'll be using.

## Registering your 'app' with Twitter

**02** We need to authenticate with Twitter using OAuth. Before this, you need to go to **https://dev.twitter.com/apps** and sign in with the account you'd like your Pi to tweet from. Click the 'Create a new application' button. We called our application 'LUD Pi Bot' and set the website to **www.linuxuser.co.uk**.

## Full code listing

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

# A Twitter Bot for the Raspberry Pi that retweets any
content from

import sys
import time
from datetime import datetime
from twython import Twython

class bot:
    def __init__(self, c_key, c_secret, a_token, a_token_
secret):
        # Create a Twython API instance
        self.api = Twython(c_key, c_secret, a_token,
a_token_secret)

        # Make sure we are authenticated correctly
        try:
            self.api.verify_credentials()
        except:
            sys.exit("Authentication Failed")

        self.last_ran = datetime.now()

    @staticmethod
```

## Creating an access token

**03** Go to the Settings tab and change the Access type from 'Read only' to 'Read and Write'. Then click the 'Update this Twitter application's settings' button. Next we create an access token. Click the 'Create my access token' button. If you refresh the details page, you should have a consumer key, a consumer secret and access token, plus an access token secret. This is everything we need to authenticate with Twitter.

## Authenticating with Twitter

**04** We're going to create our bot as a class, where we authenticate with Twitter in the constructor. We take the tokens from the previous steps as parameters and use them to create an instance of the Twython API. We also have a variable, last_ran, which is set to the current time. This is used to check if there are new tweets later on.

## Retweeting a user

**05** The first thing we need to do is get a list of the user's latest tweets. We then loop through each tweet and get its creation time as a string, which is then converted to a datetime object. We then check that the tweet's time is newer than the time the function was last called – and if so, retweet the tweet.

## The main section

**06** The main section is straightforward. We create an instance of the bot class using our tokens, and then go into an infinite loop. In this loop, we check for any new retweets from the users we are monitoring (we could run the retweet task with different users), then update the time everything was last run, and sleep for five minutes.

```
    def timestr_to_datetime(timestr):
        # Convert a string like Sat Nov 09 09:29:55 +0000
        # 2013 to a datetime object. Get rid of the timezone
        # and make the year the current one
        timestr = "{0} {1}".format(timestr[:19], datetime.
now().year)

        # We now have Sat Nov 09 09:29:55 2013
        return datetime.strptime(timestr, '%a %b %d %H:%M:
%S %Y')

    def retweet_task(self, screen_name):
        # Retweets any tweets we've not seen
        # from a user
        print "Checking for new tweets from
@{0}".format(screen_name)

        # Get a list of the users latest tweets
        timeline = self.api.get_user_timeline
(screen_name = screen_name)

        # Loop through each tweet and check if it was
        # posted since we were last called
        for t in timeline:
            tweet_time = bot.timestr_to_datetime
(t['created_at'])
            if tweet_time > self.last_ran:
                print "Retweeting {0}".format(t['id'])
                self.api.retweet(id = t['id'])

if __name__ == "__main__":
    # The consumer keys can be found on your application's
    # Details page located at https://dev.twitter.com/
    # apps(under "OAuth settings")
    c_key=""
    c_secret=""

    # The access tokens can be found on your applications's
    # Details page located at https://dev.twitter.com/apps
    # (located under "Your access token")
    a_token=""
    a_token_secret=""

    # Create an instance of the bot class
    twitter = bot(c_key, c_secret, a_token, a_token_secret)

    # Retweet anything new by @LinuxUserMag every 5 minutes
    while True:
        # Update the time after each retweet_task so we're
        # only retweeting new stuff
        twitter.retweet_task("LinuxUserMag")
        twitter.last_ran = datetime.now()
        time.sleep(5 * 60)
```

## What you'll need…

**Breadboard:**
www.proto-pic.co.uk/half-size-breadboard

**3mm LED light:**
www.ultraleds.co.uk/led-product-catalogue/basic-leds-3-5-8-10mm.html

**Wires:**
www.picomake.com/product/breadboard-wires

**270-ohm resistor:**
http://goo.gl/ox4FTp5ntj0091

Fig 1: All the items you will need to get going adjusting an LED using PWM. The wires should have a male and female ends

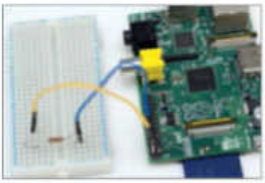


Fig 2: Place the female end onto the Pi, noting pin number 1 being identified by the small 'P1'. The blue wire is ground

Fig 3: Once everything is connected up, plug in your USB power cable

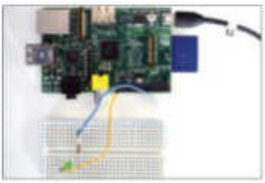


Fig 4: Switch the power on. The LED will light up. If it's dim, use a lower-rated resistor

# Control an LED using GPIO

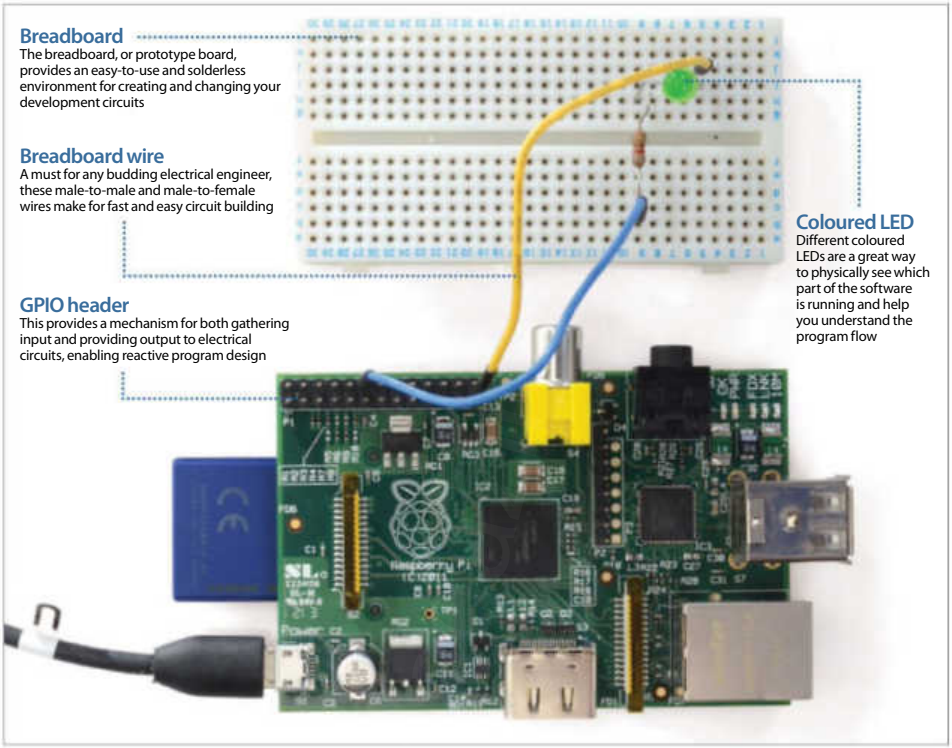
An introduction into using an external output, such as an LED, on the Pi

After you have fired up your Pi, maybe installed XBMC and had a play around with streaming, you might be ready for your next challenge. One route to go down is to find interesting uses for one of the many desktop OSs available for the little computer, perhaps using it as a web server, an NAS or retro arcade console. This is all great fun, but an often-overlooked feature of the Pi is its hardware pinouts. If you've never done any electronics before, then the Pi is a great place to start. Or maybe you have used a programmable microcontroller such as Arduino in the past; the Pi, with its increased CPU and RAM over the Arduino, opens up many more possibilities for fun projects.

The Raspberry Pi features a single PWM (pulse width modulation) output pin, along with a series of GPIO (General Purpose Input/Output) pins. These enable electronic hardware such as buzzers, lights and switches to be controlled via the Pi. For people who are used to either just 'using' a computer, or only programming software that only acts on the machine itself, controlling a real physical item such as a light can be a revelation.

This tutorial will assume no prior knowledge of electronics or programming, and will take you through the steps needed to control an LED using the Raspberry Pi, from setting it up to coding a simple application.

"We'll take you through the steps needed to control an LED using the Pi, from setting it up to coding"




## The Pi's pins

**01** Before we dive into writing code, let's take a look at the layout of the pins on the Pi. If you have your Pi in a case, take it out and place it in front of you with the USB ports on the right. Over the next few steps we'll look at some of the issues you'll encounter when using the GPIO port.

## Pi revision 1 or 2?

**02** Depending on when you purchased your Pi, you may have a 'revision 1' or 'revision 2' model. The GPIO layout is slightly different for each, although they do have the same functionality. Here we have a revision 1; revision 2s became available towards the end of 2012.

## Pin numbers

**03** If you take a look at the top left of the board you will see a small white label, 'P1'. This is pin 1 and above it is pin 2. To the right of pin 1 is pin 3, and above 3 is 4. This pattern continues until you get to pin 26 at the end. As you'll see in the next step, some pins have important uses.

## Pin uses

**04** Pin 1 is 3V3, or 3.3 volts. This is the main pin we will be using in this guide to provide power to our LED. Pins 2 and 4 are 5V. Pin 6 is the other pin we will use here, which is ground. Other ground pins are 9, 14, 20 and 25. You should always ensure your project is properly grounded.

## GPIO pins

**05** The other pins on the board are GPIO (General Purpose Input/Output). These are used for other tasks that you need to do as your projects become more complex and challenging. Be aware that the USB power supply doesn't offer much scope for powering large items.

## Basic LED lighting

**06** Okay, so let's get down to business and start making something. Firstly, get your breadboard, two wires, a 270Ω resistor and an LED. Note the slightly bent leg on one side of the LED; this is important for later on. Make sure your Pi is unplugged from the mains supply.

## Wiring the board

**07** Plug one wire into the number 1 pin, and the other end into the breadboard. Note that it doesn't matter where on the breadboard you plug it in, but make sure there are enough empty slots around it to add the LED and resistor to. Now get another wire ready.

## Add another wire

**08** Place the female end of the wire into pin number 6 (ground) and the other end into the breadboard, making sure to leave room for the resistor, depending on how large it is. Next, get your resistor ready. You can use a little higher or lower than 270 ohms, but not using a resistor at all will likely blow the LED.

## Add the resistor

**09** Next we need to add our resistor. Place one end next to the ground wire on the breadboard, and the other one slot below the 3V3 wire connection. This will sit next to the LED when we add it in a second. Note that there is no correct or incorrect way to add a resistor.

## Add the LED

**10** Grab your LED and place the 'bent' leg end next to the 3V3 wire in the breadboard. Place the other leg next to the resistor leg opposite the ground wire. This now completes the circuit and we are ready to test out our little task.

## Power it up

**11** Now get your micro-USB socket and either plug the mains end into the wall, or plug it into a computer or laptop port (and powered on!). You should see the LED light up. If not, then check your connections on the breadboard or Pi, or try a different LED.

## Set up programming environment

**12** Now, we need to be able to do a little bit more than just turn a light on – we want to be able to control it via code. Set up a new Raspbian installation (a guide to this is found on page 28). You don't need a GUI for this – it can all be done via the terminal if you so wish. Before starting, it's best to check everything is up to date with:

```
sudo apt-get dist-upgrade
```

## Open up terminal

**13** Assuming we want to use the GUI, rather than SSH into the Pi, open up a new terminal window by double-clicking on the LXTerminal icon. We need root access to control the LEDs, so either enter su now, or remember to prefix any commands with sudo.

```
su
```

followed by password or add

```
sudo
```

to the start of each command.

## Download GPIO library

**14** There is a handy GPIO Python library that makes manipulating the GPIO pins a breeze. From within your terminal window, use wget to download the tarball file, then extract it using tar. This will give us all the files in a new directory.

```
wget https://pypi.python.org/packages
source/R/RPi.GPIO/RPi.GPIO-0.5.2a.tar.gz
tar zxf Rpi.GPIO-0.5.2a.tar.gz
cd Rpi.GPIO-0.5.2a
```

## Install the library

**15** Now we need to install the library. This is simply a case of using Python's install method; so we need the dev version of Python. Make sure you are in the directory of the library before running this command.

```
sudo apt-get install python-dev
sudo python setup.py install
```

## Import the library in a script

**16** Create a new Python script. Next import the main GPIO library and we'll put it in a try-except block. Save the file using Ctrl+X and choosing 'yes'.

```
cd /
cd Desktop
sudo nano gpio.py
try:
        import RPi.GPIO as GPIO
except RuntimeError:
        print("Error importing GPIO lib")
```

## Did you know…

The official Python docs are a great resource for beginners and professionals alike. **http://python.org/doc**.

## Test the script

**17** Now to make sure that the script imported okay, we just need to run the python command and then tell it the name of the script that we just created. If all goes well, you shouldn't see any error messages. Don't worry if you do, though. Just go back through the previous steps to check everything is as it should be.

```
sudo python gpio.py
```

## Set GPIO mode

**18** Reload the script in nano again. We will then set the GPIO mode to BOARD. This method is the safest for a beginner to adopt and will work whichever revision of the Pi you are using. It's best to pick a GPIO convention and stick to it because this will save confusion later on.

```
sudo nano gpio.py
GPIO.setmode(GPIO.BOARD)
```

## Set pin mode

**19** A pin has to be defined as either an input or an output before it can work. This is simplified in the GPIO library by calling the GPIO.setup method. You then pass in the pin number, and then GPIO.OUT or GPIO.IN. As we want to use an LED, it's an output. You'll be using these conventions frequently, so learn them as best you can so they soak in!

```
GPIO.setup(12, GPIO.OUT)
```

## Using PWM

**20** The next step is to tell the pin to output and then set a way of escaping our program. Here we call the GPIO class again and then the PWM method, passing in the pin number; the second value is the frequency in hertz – in this case, 0.5.

```
p = GPIO.PWM(12, 0.5)
p.start(1)
input('Press return to stop:')
p.stop()
GPIO.cleanup()
```

## Adjust PWM

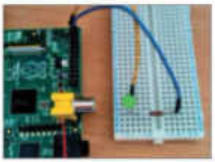
**21** To add a timer to the LED so it fades out, we first need to import the time library and then set the 12 pin to have 50Hz frequency to start off with.

```
import time
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BOARD)
GPIO.setup(12, GPIO.OUT)
p = GPIO.PWM(12, 50)  # channel=12 frequency=50Hz
p.start(0)
```

## Add the fade

**22** Then we add in another try-except block, this time checking what power the LED is at – and once it reaches a certain level, we reverse the process. To run this code, simply save it from nano and then sudo python gpio.py.

```
while 1:
        for dc in range(0, 101, 5):
                p.ChangeDutyCycle(dc)
                time.sleep(0.1)
        for dc in range(100, -1, -5):
                p.ChangeDutyCycle(dc)
                time.sleep(0.1)
        except KeyboardInterrupt:
                pass
p.stop()
GPIO.cleanup()
```



"We'll set the GPIO mode to BOARD. This will work whichever revision of the Pi you are using"

# HOW TO USE

**EVERYTHING YOU NEED TO KNOW** ABOUT ACCESSING YOUR NEW DIGITAL DEPOSITORY

## To access FileSilo, please visit filesilo.co.uk/bks-893

01 Follow the on-screen instructions to create an account with our secure FileSilo system, log in and unlock the bookazine by answering a simple question about it. One word answers only! You can now access the content for free at any time.

02 Once you have logged in, you are free to explore the wealth of content tutorials and online guides to downloadable resources. And the more bookazines you purchase, the more your instantly accessible collection of digital content will build up.

03 You can access FileSilo on any desktop, tablet or smartphone device using any popular browse. However, we recommend that you use a desktop to download content, as you may not be able to download files to your phone or tablet.

04 If you have any problems with accessing the content on FileSilo, or with the registration process, take a look at the FAQs online or email filesilohelp@imagine-publishing.co.uk

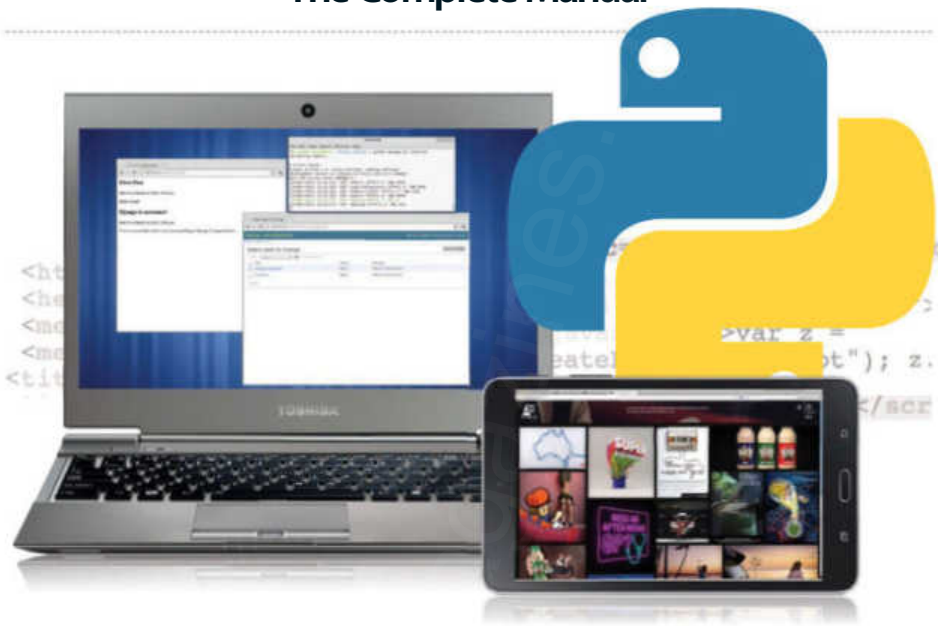

# NEED HELP WITH THE TUTORIALS?

Having trouble with any of the techniques in this bookazine's tutorials? Don't know how to make the best use of your free resources? Want to have your work critiqued by those in the know? Then why not visit the Linux User & Developer and Imagine Bookazines Facebook pages for all your questions, concerns and qualms. There is a friendly community of fellow Linux enthusiasts waiting to help you out, as well as regular posts and updates from the team behind Linux User & Developer magazine. Like us today and start chatting!

facebook.com/ImagineBookazines
facebook.com/LinuxUserUK

# Python

## The Complete Manual



### ✔ Learn to use Python

Master the essentials and code simple projects as you learn how to work with one of the most versatile languages around

### ✔ Program games

Use what you've learnt to create playable games, and see just how powerful Python can be

### ✔ Essential tips

Discover everything you need to know about writing clean code, getting the most from Python's capabilities and much more

### ✔ Amazing projects

Get creative and complete projects including programming a drone and sending texts from Pi

### ✔ Master building apps

Make your own web and Android apps with step-by-step tutorials

### ✔ Create with Raspberry Pi

Unlock the real potential of your Raspberry Pi computer using Python, its officially recognised coding language

### ✔ Put Python to work

Use Python for functional projects such as scientific computing and make reading websites offline easier and more enjoyable

### ✔ Free online resources

Download all of the tutorial files you need to complete the steps in the book, plus watch videos and more with your free FileSilo resources

www.imaginebookshop.co.uk