NEW

OVER 2 HOURS OF VIDEO TUTORIALS

The Python Book

The ultimate guide to coding with Python

Over 200 essential tips

Learn to use Python • Program games • Get creative with Pi

Welcome to

The Python Book

Python is an incredibly versatile, expansive language which, due to its similarity to everyday language, is surprisingly easy to learn even for inexperienced programmers. It has seen a huge increase in popularity since the release and rise of the Raspberry Pi, for which Python is the officially recognised programming language. In this new edition of The Python Book, you'll find plenty of creative projects to help you get to grips with the combination of your Raspberry Pi and Python's powerful functionality, plus lots of tutorials that focus on Python's effectiveness away from the Raspberry Pi. You'll learn all about how to code with Python from a standing start, with our comprehensive masterclass, then go on to complete tutorials that will consolidate your skills and help you to become fluent in the language. You'll learn how to make Python work for you with tutorials on coding with Django, Flask, Pygame and even more useful third-party frameworks. Get ready to become a true Python expert with the wealth of information contained within these pages.

The Python Book

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

Editor in Chief
Jon White

Production Editor
Jasmin Snook

Senior Art Editor
Greg Whitaker

Assistant Designer
Steve Dacombe

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
Gordon & Gotch Australia Pty Ltd, 26 Rodborough Road, Frenchs Forest, NSW, 2086 Australia
Tel +61 2 9972 8800 www.gordongotch.com.au

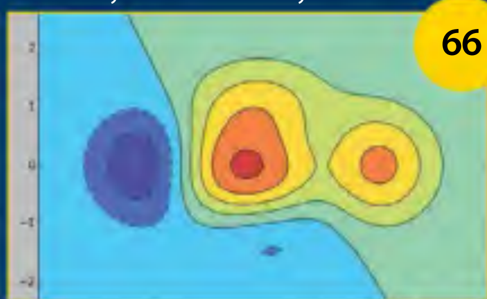
ISBN 978 1785 463 396

Part of the LinuxUser & Developer bookazine series

IMAGINE PUBLISHING

The Python Book

Contents



120

Web development

Use Python with Pi

26

50 Python tips

160

"You'll be surprised by the diversity of what you can make"

Get started with Python

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

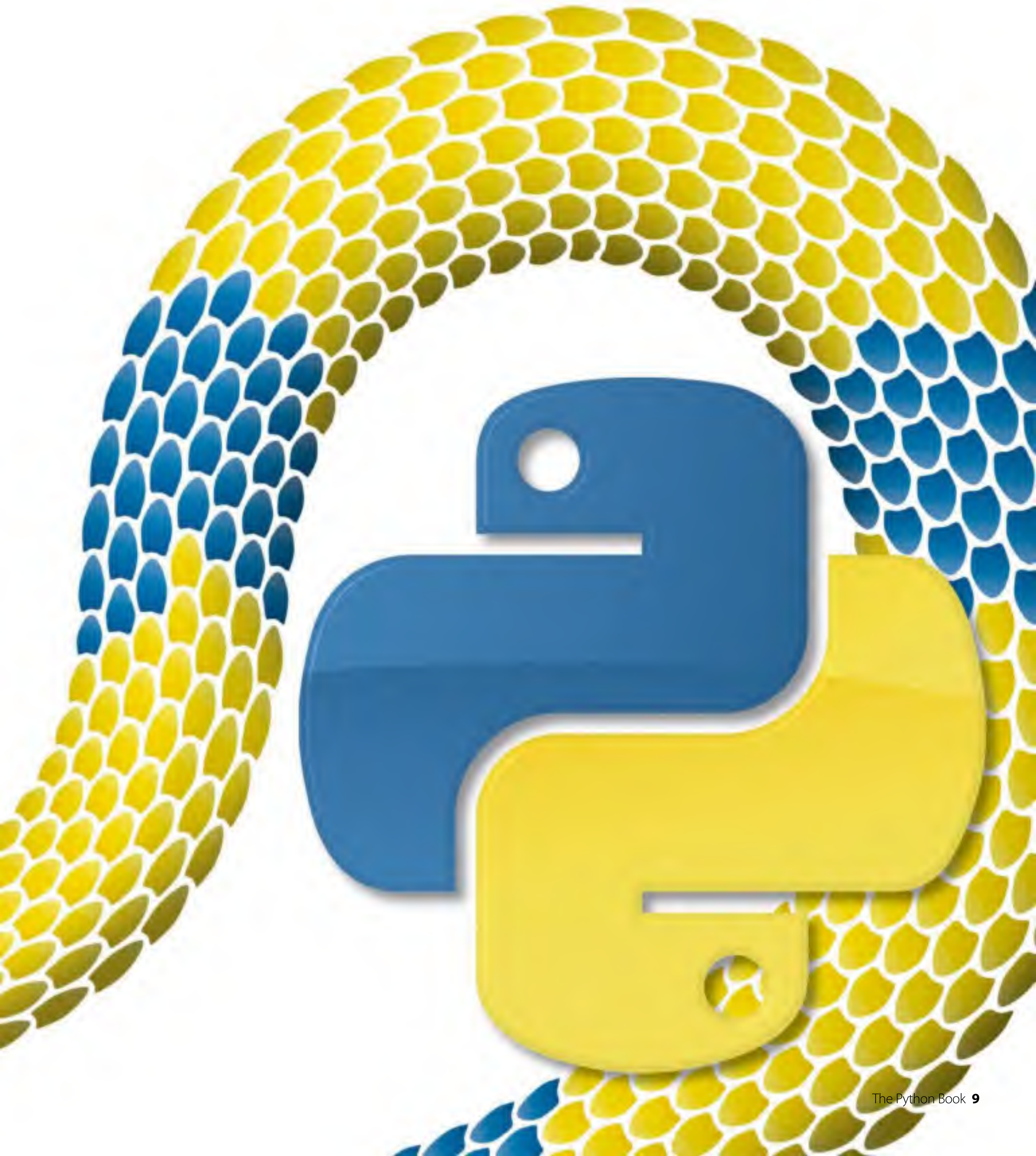
Python is a great programming language for both beginners and experts. It is designed with code readability in mind, making it an excellent choice for beginners who are still getting used to various programming concepts.

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 guide 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 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 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 expected, 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
```

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.

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

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 the element's position 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.

> "A variable is a name in source code that is associated with an area in memory that you can use to store data"

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.

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

The same principal is true of Boolean values

We create a tuple in the following way

And a list in this way

You could also create the same list in the following way

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

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

```
#!/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"

hello_int = 21

hello_bool = True

hello_tuple = (21, 32)

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.

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 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.

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

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.

> "Any text in a Python file that follows a # character will be ignored"

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

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

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

```
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.

print(hello_dict["first_name"] + " " + hello_dict["last_name"] + " has " +
    hello_dict["eye_colour"] + " eyes.")

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

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 structures include:

- 'For' statements, which allow you to iterate over items in collections, or to repeat a piece of code 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. The output from the above program is as follows:

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 that you 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 happen to open a file in more than one editor.

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

```
[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
```

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.

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

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

```
#!/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

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

    except:
        print("You must enter an integer")

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

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


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

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")`

```
# 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
```

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 used in programming to break processes down in"

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

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

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


test_string = "This is a test string"

modify_string(test_string)
print(test_string)

test_string = modify_string_return(test_string)
print(test_string)

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

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

The test string won't be changed in this code

However, we can call the function like this

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

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

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

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

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

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

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

cont = False

var = 0
if cont:
    var = 1234

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

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.

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.

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

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 enough 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.

Happy programming!

50 ESSENTIAL PYTHON COMMANDS

Python is known as a very dense language, with lots of modules capable of doing almost anything. Here, we will look at the core essentials that everyone needs to know

Python has a massive environment of extra modules that can provide functionality in hundreds of different disciplines. However, every programming language has a core set of functionality that everyone should know in order to get useful work done. Python is no different in this regard. Here, we will look at 50 commands that we consider to be essential to programming in Python. Others may pick a slightly different set, but this list contains the best of the best.

We will cover all of the basic commands, from importing extra modules at the beginning of a program to returning values to the calling environment at the end. We will also be looking at some commands that are useful in learning about the current session within Python, like the current list of variables that have been defined and how memory is being used.

Because the Python environment involves using a lot of extra modules, we will also look at a few commands that are strictly outside of Python. We will see how to install external modules and how to manage multiple environments for different development projects. Since this is going to be a list of commands, there is the assumption that you already know the basics of how to use loops and conditional structures. This piece is designed to help you remember commands that you know you've seen before, and hopefully introduce you to a few that you may not have seen yet.

Although we've done our best to pack everything you could ever need into 50 tips, Python is such an expansive language that some commands will have been left out. Make some time to learn about the ones that we didn't cover here, once you've mastered these.

01 Importing modules

The strength of Python is its ability to be extended through modules. The first step in many programs is to import those modules that you need. The simplest import statement is to just call 'import modulename'. In this case, those functions and objects provided are not in the general namespace. You need to call them using the complete name (modulename.methodname). You can shorten the 'modulename' part with the command 'import modulename as mn'. You can skip this issue completely with the command 'from modulename import *' to import everything from the given module. Then you can call those provided capabilities directly. If you only need a few of the provided items, you can import them selectively by replacing the '*' with the method or object names.

02 Reloading modules

When a module is first imported, any initialisation functions are run at that time. This may involve creating data objects, or initiating connections. But, this is only done the first time within a given session. Importing the same module again won't re-execute any of the initialisation code. If you want to have this code re-run, you need to use the reload command. The format is 'reload(modulename)'. Something to keep in mind is that the dictionary from the previous import isn't dumped, but only written over. This means that any definitions that have changed between the import and the reload are updated correctly. But if you delete a definition, the old one will stick around and still be accessible. There may be other side effects, so always use with caution.

03 Installing new modules

While most of the commands we are looking at are Python commands that are to be executed within a Python session, there are a few essential commands that need to be executed outside of Python. The first of these is pip. Installing a module involves downloading the source code, and compiling any included external code. Luckily, there is a repository of hundreds of Python modules available at **http://pypi.python.org**. Instead of doing everything manually, you can install a new module by using the command 'pip install modulename'. This command will also do a dependency check and install any missing modules before installing the one you requested. You may need administrator rights if you want this new module installed in the global library for your computer. On a Linux machine, you would simply run the pip command with sudo. Otherwise, you can install it to your personal library directory by adding the command line option '—user'.

"Every programming language out there has a core set of functionality that everyone should know in order to get useful work done. Python is no different"

04 Executing a script

Importing a module does run the code within the module file, but does it through the module maintenance code within the Python engine. This maintenance code also deals with running initialising code. If you only wish to take a Python script and execute the raw code within the current session, you can use the 'execfile("filename.py")' command, where the main option is a string containing the Python file to load and execute. By default, any definitions are loaded into the locals and globals of the current session. You can optionally include two extra parameters the execfile command. These two options are both dictionaries, one for a different set of locals and a different set of globals. If you only hand in one dictionary, it is assumed to be a globals dictionary. The return value of this command is None.

05 An enhanced shell

The default interactive shell is provided through the command 'python', but is rather limited. An enhanced shell is provided by the command 'ipython'. It provides a lot of extra functionality to the code developer. A thorough history system is available, giving you access to not only commands from the current session, but also from previous sessions. There are also magic commands that provide enhanced ways of interacting with the current Python session. For more complex interactions, you can create and use macros. You can also easily peek into the memory of the Python session and decompile Python code. You can even create profiles that allow you to handle initialisation steps that you may need to do every time you use iPython.

06 Evaluating code

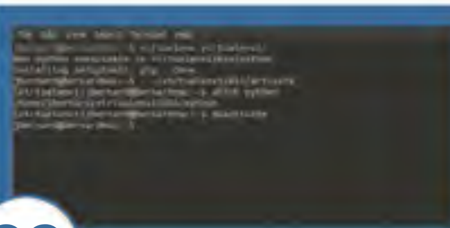
Sometimes, you may have chunks of code that are put together programmatically. If these pieces of code are put together as a string, you can execute the result with the command 'eval("code_string")'. Any syntax errors within the code string are reported as exceptions. By default, this code is executed within the current session, using the current globals and locals dictionaries. The 'eval' command can also take two other optional parameters, where you can provide a different set of dictionaries for the globals and locals. If there is only one additional parameter, then it is assumed to be a globals dictionary. You can optionally hand in a code object that is created with the compile command instead of the code string. The return value of this command is None.

07 Asserting values

At some point, we all need to debug some piece of code we are trying to write. One of the tools useful in this is the concept of an assertion. The assert command takes a Python expression and checks to see if it is true. If so, then execution continues as normal. If it is not true, then an AssertionError is raised. This way, you can check to make sure that invariants within your code stay invariant. By doing so, you can check assumptions made within your code. You can optionally include a second parameter to the assert command. This second parameter is Python expression that is executed if the assertion fails. Usually, this is some type of detailed error message that gets printed out. Or, you may want to include cleanup code that tries to recover from the failed assertion.

08 Mapping functions

A common task that is done in modern programs is to map a given computation to an entire list of elements. Python provides the command 'map()' to do just this. Map returns a list of the results of the function applied to each element of an iterable object. Map can actually take more than one function and more than one iterable object. If it is given more than one function, then a list of tuples is returned, with each element of the tuple containing the results from each function. If there is more than one iterable handed in, then map assumes that the functions take more than one input parameter, so it will take them from the given iterables. This has the implicit assumption that the iterables are all of the same size, and that they are all necessary as parameters for the given function.



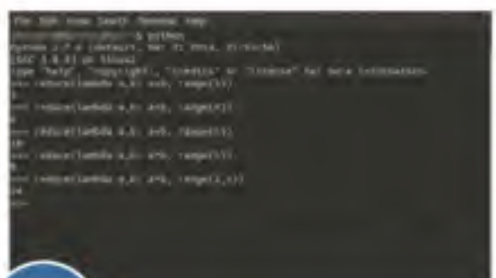
09 Virtualenvs

Because of the potential complexity of the Python environment, it is sometimes best to set up a clean environment within which to install only the modules you need for a given project. In this case, you can use the virtualenv command to initialise such an environment. If you create a directory named 'ENV', you can create a new environment with the command 'virtualenv ENV'. This will create the subdirectories bin, lib and include, and populate them with an initial environment. You can then start using this new environment by sourcing the script 'ENV/bin/activate', which will change several environment variables, such as the PATH. When you are done, you can source the script 'ENV/bin/deactivate' to reset your shell's environment back to its previous condition. In this way, you can have environments that only have the modules you need for a given set of tasks.

"While not strictly commands, everyone needs to know how to deal with loops. The two main types of loops are a fixed number of iterations loop (for) and a conditional loop (while)"

10 Loops

While not strictly commands, everyone needs to know how to deal with loops. The two main types of loops are a fixed number of iterations loop (for) and a conditional loop (while). In a for loop, you iterate over some sequence of values, pulling them off the list one at a time and putting them in a temporary variable. You continue until either you have processed every element or you have hit a break command. In a while loop, you continue going through the loop as long as some test expression evaluates to True. While loops can also be exited early by using the break command, you can also skip pieces of code within either loop by using a continue command to selectively stop this current iteration and move on to the next one.

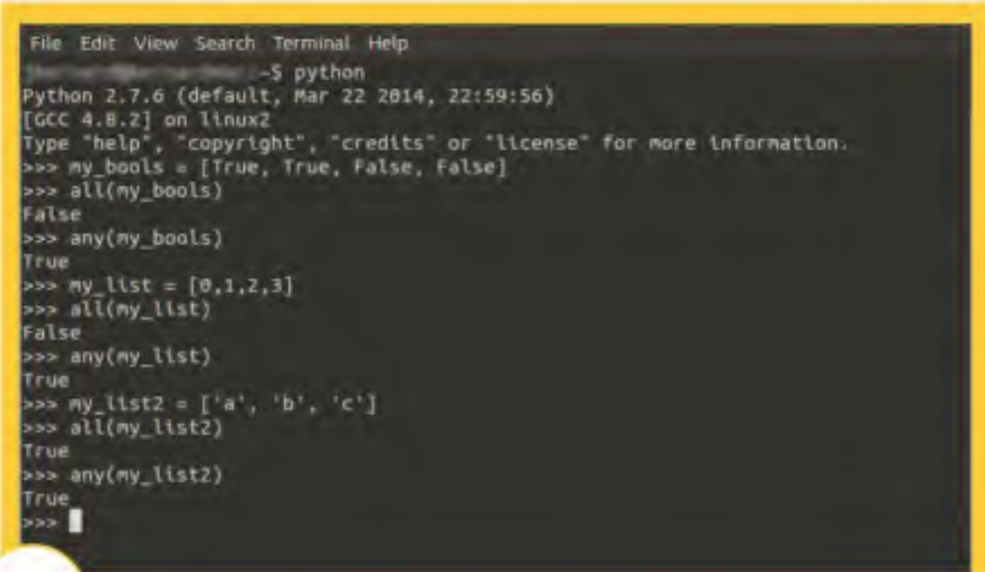


12 Reductions

In many calculations, one of the computations you need to do is a reduction operation. This is where you take some list of values and reduce it down to a single value. In Python, you can use the command 'reduce(function, iterable)' to apply the reduction function to each pair of elements in the list. For example, if you apply the summation reduction operation to the list of the first five integers, you would get the result ((((1+2)+3)+4)+5). You can optionally add a third parameter to act as an initialisation term. It is loaded before any elements from the iterable, and is returned as a default if the iterable is actually empty. You can use a lambda function as the function parameter to reduce to keep your code as tight as possible. In this case, remember that it should only take two input parameters.

11 Filtering

Where the command map returns a result for every element in an iterable, filter only returns a result if the function returns a True value. This means that you can create a new list of elements where only the elements that satisfy some condition are used. As an example, if your function checked that the values were numbers between 0 and 10, then it would create a new list with no negative numbers and no numbers above 10. This could be accomplished with a for loop, but this method is much cleaner. If the function provided to filter is 'None', then it is assumed to be the identity function. This means that only those elements that evaluate to True are returned as part of the new list. There are iterable versions of filter available in the itertools module.



13 How true is a list?

In some cases, you may have collected a number of elements within a list that can be evaluated to True or False. For example, maybe you ran a number of possibilities through your computation and have created a list of which ones passed. You can use the command 'any(list)' to check to see whether any of the elements within your list are true. If you need to check whether all of the elements are True, you can use the command 'all(list)'. Both of these commands return a True if the relevant condition is satisfied, and a False if not. They do behave differently if the iterable object is empty, however. The command 'all' returns a True if the iterable is empty, whereas the command 'any' returns a False when given any empty iterable.

14 Enumerating

Sometimes, we need to label the elements that reside within an iterable object with their indices so that they can be processed at some later point. You could do this by explicitly looping through each of the elements and building an enumerated list. The enumerate command does this in one line. It takes an iterable object and creates a list of tuples as the result. Each tuple has the 0-based index of the element, along with the element itself. You can optionally start the indexing from some other value by including an optional second parameter. As an example, you could enumerate a list of names with the command 'list(enumerate(names, start=1))'. In this example, we decided to start the indexing at 1 instead of 0.

15 Casting

Variables in Python don't have any type information, and so can be used to store any type of object. The actual data, however, is of one type or another. Many operators, like addition, assume that the input values are of the same type. Very often, the operator you are using is smart enough to make the type of conversion that is needed. If you have the need to explicitly convert your data from one type to another, there are a class of functions that can be used to do this conversion process. The ones you are most likely to use is 'abs', 'bin', 'bool', 'chr', 'complex', 'float', 'hex', 'int', 'long', 'oct', and 'str'. For the number-based conversion functions, there is an order of precedence where some types are a subset of others. For example, integers are "lower" than floats. When converting up, no changes in the ultimate value should happen. When converting down, usually some amount of information is lost. For example, when converting from float to integer, Python truncates the number towards zero.

16 What is this?

Everything in Python is an object. You can check to see what class this object is an instance of with the command 'isinstance(object, class)'. This command returns a Boolean value.

17 Is it a subclass?

The command 'issubclass(class1, class2)' checks to see if class1 is a subclass of class2. If class1 and class2 are the same, this is returned as True.

18 Global objects

You can get a dictionary of the global symbol table for the current module with the command 'globals()'.

19 Local objects

You can access an updated dictionary of the current local symbol table by using the command 'locals()'.

20 Variables

The command 'vars(dict)' returns writeable elements for an object. If you use 'vars()', it behaves like 'locals()'.

21 Making a global

A list of names can be interpreted as globals for the entire code block with the command 'global names'.

22 Nonlocals

In Python 3.X, you can access names from the nearest enclosing scope with the command 'nonlocal names' and bind it to the local scope.

23 Raising an exception

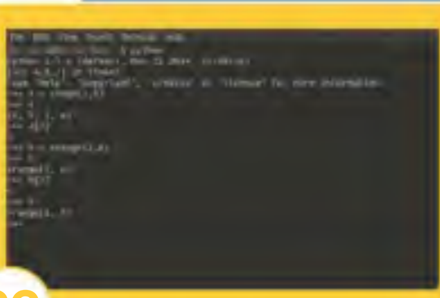
When you identify an error condition, you can use the 'raise' command to throw up an exception. You can include an exception type and a value.

24 Dealing with an exception

Exceptions can be caught in a try-except construction. If the code in the try block raises an exception, the code in the except block gets run.

25 Static methods

You can create a statis method, similar to that in Java or C++, with the command 'staticmethod(function_name)'.



26 Ranges

You may need a list of numbers, maybe in a 'for' loop. The command 'range()' can create an iterable list of integers. With one parameter, it goes from 0 to the given number. You can provide an optional start number, as well as a step size. Negative numbers count down.

27 Xranges

One problem with ranges is that all of the elements need to be calculated up front and stored in memory. The command 'xrange()' takes the same parameters and provides the same result, but only calculates the next element as it is needed.

28 Iterators

Iteration is a very Pythonic way of doing things. For objects which are not intrinsically iterable, you can use the command 'iter(object_name)' to essentially wrap your object and provide an iterable interface for use with other functions and operators.

29 Sorted lists

You can use the command 'sorted(list1)' to sort the elements of a list. You can give it a custom comparison function, and for more complex elements you can include a key function that pulls out a ranking property from each element for comparison.

30 Summing items

Above, we saw the general reduction function reduce. A specific type of reduction operation, summation, is common enough to warrant the inclusion of a special case, the command 'sum(iterable_object)'. You can include a second parameter here that will provide a starting value.

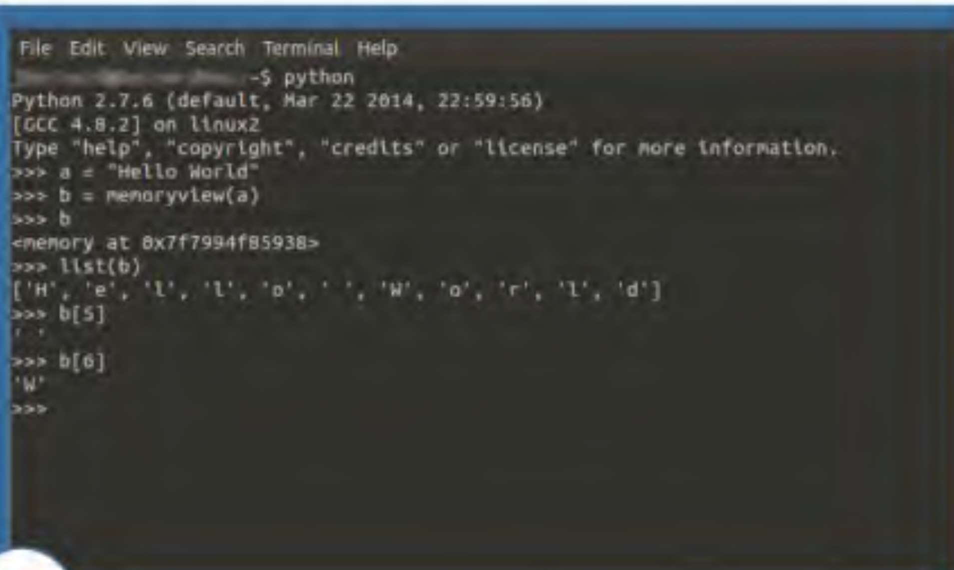
31 With modules

The 'with' command provides the ability to wrap a code block with methods defined by a context manager. This can help clean up code and make it easier to read what a given piece of code is supposed to be doing months later. A classic example of using 'with' is when dealing with files. You could use something like 'with open("myfile.txt", "r") as f:'. This will open the file and prepare it for reading. You can then read the file in the code block with 'data=f.read()'. The best part of doing this is that the file will automatically be closed when the code block is exited, regardless of the reason. So, even if the code block throws an exception, you don't need to worry about closing the file as part of your exception handler. If you have a more complicated 'with' example, you can create a context manager class to help out.

32 Printing

The most direct way of getting output to the user is with the print command. This will send text out to the console window. If you are using version 2.X of Python, there are a couple of ways you can use the print command. The most common way had been simply call it as 'print "Some text"'. You can also use print with the same syntax that you would use for any other function. So, the above example would look like 'print("Some text")'. This is the only form available in version 3.X. If you use the function syntax, you can add extra parameters that give you finer control over this output. For example, you can give the parameter 'file=myfile.txt' and get the output from the print command being dumped into the given text file. It also will accept any object that has some string representation available.

> "A classic example of using 'with' is when dealing with files. The best part of doing this is that the file will automatically be closed when the code block is exited, regardless of the reason"



33 Memoryview

Sometimes, you need to access the raw data of some object, usually as a buffer of bytes. You can copy this data and put it into a bytearray, for example. But this means that you will be using extra memory, and this might not be an option for large objects. The command 'memoryview(object_name)' wraps the object handed in to the command and provides an interface to the raw bytes. It gives access to these bytes an element at a time. In many cases, elements are the size of one byte. But, depending on the object details, you could end up with elements that are larger than that. You can find out the size of an element in bytes with the property 'itemsize'. Once you have your memory view created, you can access the individual elements as you would get elements from a list (mem_view[1], for example).

34 Files

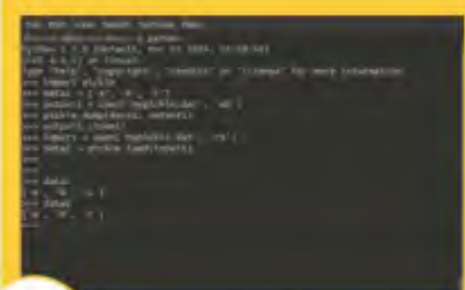
When dealing with files, you need to create a file object to interact with it. The file command takes a string with the file name and location and creates a file object instance. You can then call the file object methods like 'open', 'read' and 'close', to get data out of the file. If you are doing file processing, you can also use the 'readline' method. When opening a file, there is an explicit 'open()' command to simplify the process. It takes a string with the file name, and an optional parameter that is a string which defines the mode. The default is to open the file as read-only ('r'). You can also open it for writing ('w') and appending ('a'). After opening the file, a file object is returned so that you can further interact with it. You can then read it, write to it, and finally close it.

35 Yielding

In many cases, a function may need to yield the context of execution to some other function. This is the case with generators. The preferred method for a generator is that it will only calculate the next value when it is requested through the method 'next()'. The command 'yield' saves the current state of the generator function, and return execution control to the calling function. In this way, the saved state of the generator is reloaded and the generator picks up where it left off in order to calculate the next requested value. In this way, you only need to have enough memory available to store the bare minimum to calculate the next needed value, rather than having to store all of the possible values in memory all at once.

36 Weak references

You sometimes need to have a reference to an object, but still be able to destroy it if needed. A weak reference is one which can be ignored by the garbage collector. If the only references left to n object are weak references, then the garbage collector is allowed to destroy that object and reclaim the space for other uses. This is useful in cases where you have caches or mappings of large datasets that don't necessarily have to stay in memory. If an object that is weakly referenced ends up being destroyed and you try to access it, it will appear as a None. You can test for this condition and then reload the data if you decide that this is a necessary step.



37 Pickling data

There are a few different ways of serialising memory when you need to checkpoint results to disk. One of these is called pickling. Pickle is actually a complete module, not just a single command. To store data on to the hard drive, you can use the dump method to write the data out. When you want to reload the same data at some other point in the future, you can use the load method to read the data in and unpickle it. One issue with pickle is its speed, or lack of it. There is a second module, cPickle, that provides the same basic functionality. But, since it is written in C, it can be as much as 1000 times faster. One thing to be aware of is that pickle does not store any class information for an object, but only its instance information. This means that when you unpickle the object, it may have different methods and attributes if the class definition has changed in the interim.

38 Shelving data

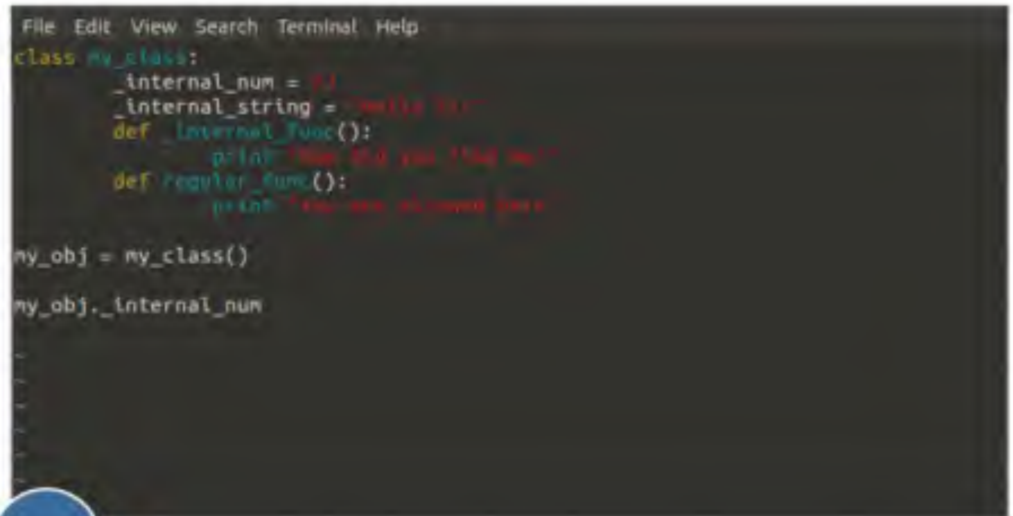
While pickling allows you save data and reload it, sometimes you need more structured object permanence in your Python session. With the shelve module, you can create an object store where essentially anything that can be pickled can be stored there. The backend of the storage on the drive can be handled by one of several systems, such as dbm or gdbm. Once you have opened a shelf, you can read and write to it using key value pairs. When you are done, you need to be sure to explicitly close the shelf so that it is synchronised with the file storage. Because of the way the data may be stored in the backing database, it is best to not open the relevant files outside of the shelve module in Python. You can also open the shelf with writeback set to True. If so, you can explicitly call the sync method to write out cached changes.

39 Threads

You can do multiple threads of execution within Python. The 'thread()' command can create a new thread of execution for you. It follows the same techniques as those for POSIX threads. When you first create a thread, you need to hand in a function name, along with whatever parameters said function needs. One thing to keep in mind is that these threads behave just like POSIX threads. This means that almost everything is the responsibility of the programmer. You need to handle mutex locks (with the methods 'acquire' and 'release'), as well as create the original mutexes with the method 'allocate_lock'. When you are done, you need to 'exit' the thread to ensure that it is properly cleaned up and no resources get left behind. You also have fine-grained control over the threads, being able to set things like the stack size for new threads.

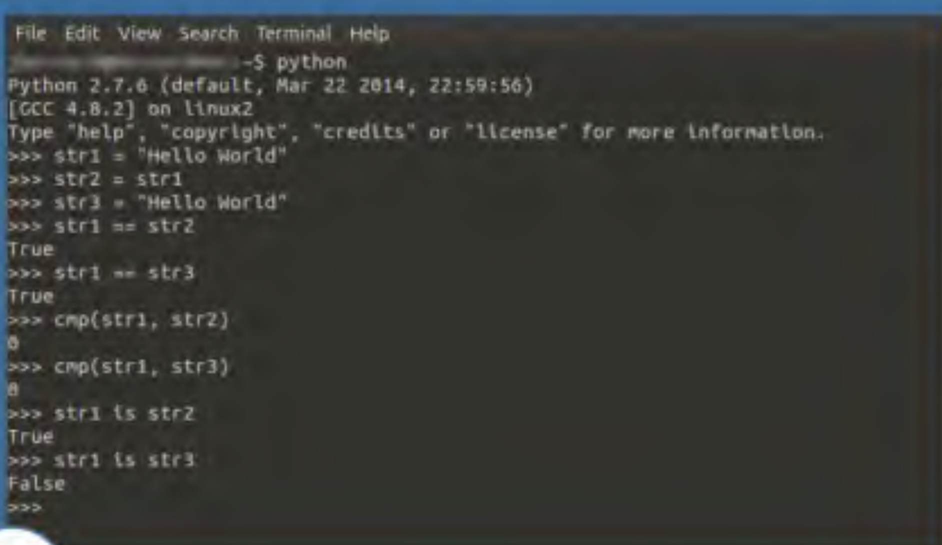
40 Inputting data

Sometimes, you need to collect input from an end user. The command 'input()' can take a prompt string to display to the user, and then wait for the user to type a response. Once the user is done typing and hits the enter key, the text is returned to your program. If the readline module was loaded before calling input, then you will have enhanced line editing and history functionality. This command passes the text through eval first, and so may cause uncaught errors. If you have any doubts, you can use the command 'raw_input()' to skip this problem. This command simply returns the unchanged string inputted by the user. Again, you can use the readline module to get enhanced line editing.



41 Internal variables

For people coming from other programming languages, there is a concept of having certain variables or methods be only available internally within an object. In Python, there is no such concept. All elements of an object are accessible. There is a style rule, however, that can mimic this type of behaviour. Any names that start with an underscore are expected to be treated as if they were internal names and to be kept as private to the object. They are not hidden, however, and there is no explicit protection for these variables or methods. It is up to the programmer to honour the intention from the author the class and not alter any of these internal names. You are free to make these types of changes if it becomes necessary, though.



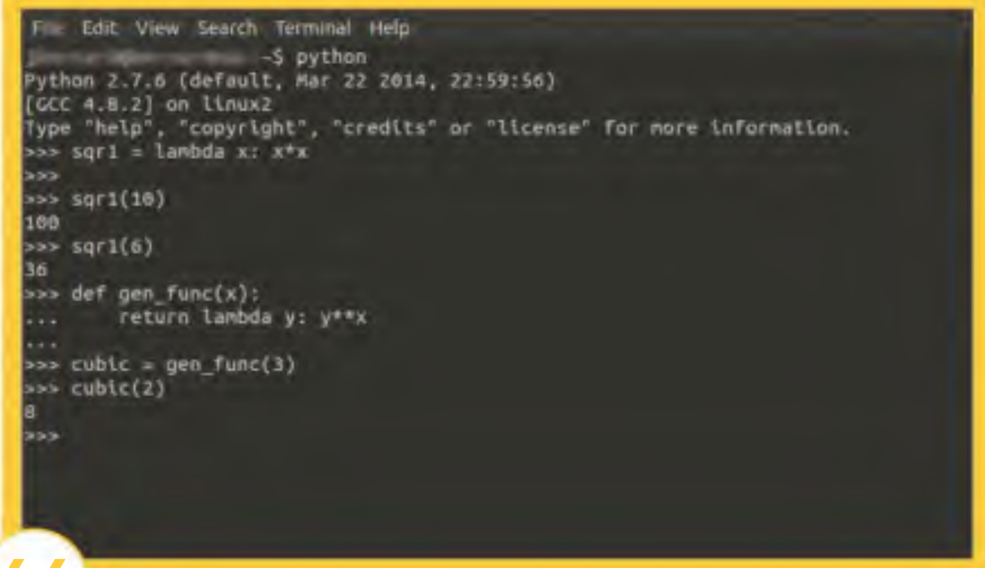
42 Comparing objects

There are several ways to compare objects within Python, with several caveats. The first is that you can test two things between objects: equality and identity. If you are testing identity, you are testing to see if two names actually refer to the same instance object. This can be done with the command 'cmp(obj1, obj2)'. You can also test this condition by using the 'is' keyword. For example, 'obj1 is obj2'. If you are testing for equality, you are testing to see whether the values in the objects referred to by the two names are equal. This test is handled by the operator '==', as in 'obj1 == obj2'. Testing for equality can become complex for more complicated objects.

43 Slices

While not truly a command, slices are too important a concept not to mention in this list of essential commands. Indexing elements in data structures, like lists, is one of the most common things done in Python. You can select a single element by giving a single index value. More interestingly, you can select a range of elements by giving a start index and an end index, separated by a colon. This gets returned as a new list that you can save in a new variable name. You can even change the step size, allowing you to skip some number of elements. So, you could grab every odd element from the list 'a' with the slice 'a[1::2]'. This starts at index 1, continues until the end, and steps through the index values 2 at a time. Slices can be given negative index values. If you do, then they start from the end of the list and count backwards.

"Python is an interpreted language, which means that the source code that you write needs to be compiled into a byte code format. This byte code then gets fed into the actual Python engine"



44 Lambda expressions

Since objects, and the names that point to them, are truly different things, you can have objects that have no references to them. One example of this is the lambda expression. With this, you can create an anonymous function. This allows you use functional programming techniques within Python. The format is the keyword 'lambda', followed by a parameter list, then a colon and the function code. For example, you could build your own function to square a number with 'lambda x: x*x'. You can then have a function that can programmatically create new functions and return them to the calling code. With this capability, you can create function generators to have self-modifying programs. The only limitation is that they are limited to a single expression, so you can't generate very complex functions.

45 Compiling code objects

Python is an interpreted language, which means that the source code that you write needs to be compiled into a byte code format. This byte code then gets fed into the actual Python engine to step through the instructions. Within your program, you may have the need to take control over the process of converting code to byte code and running the results. Maybe you wish to build your own REPL. The command 'compile()' takes a string object that contains a collection of Python code, and returns an object that represents a byte code translation of this code. This new object can then be handed in to either 'eval()' or 'exec()' to be actually run. You can use the parameter 'mode=' to tell compile what kind of code is being compiled. The 'single' mode is a single statement, 'eval' is a single expression and 'exec' is a whole code block.

46 __init__ method

When you create a new class, you can include a private initialisation method that gets called when a new instance of the class is created. This method is useful when the new object instance needs some data loaded in the new object.

47 __del__ method

When an instance object is about to be destroyed, the __del__ method is called. This gives you the chance to do any kind of cleanup that may be required. This might be closing files, or disconnecting network connections. After this code is completed, the object is finally destroyed and resources are freed.

48 Exiting your program

There are two pseudo-commands available to exit from the Python interpreter: 'exit()' and quit()'. They both take an optional parameter which sets the exit code for the process. If you want to exit from a script, you are better off using the exit function from the sys module ('sys.exit(exit_code)'.

49 Return values

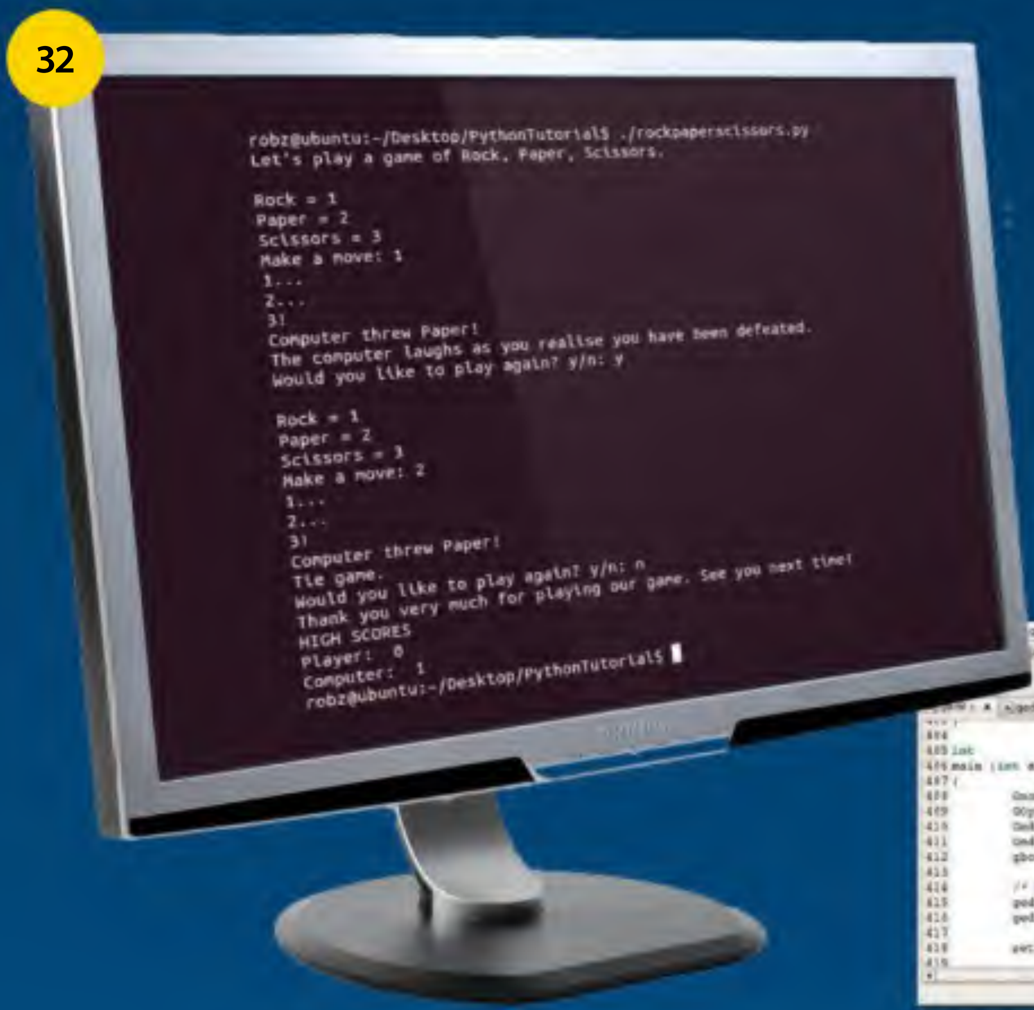
Functions may need to return some value to the calling function. Because essentially no name has a type, this includes functions. So functions can use the 'return' command to return any object to the caller.

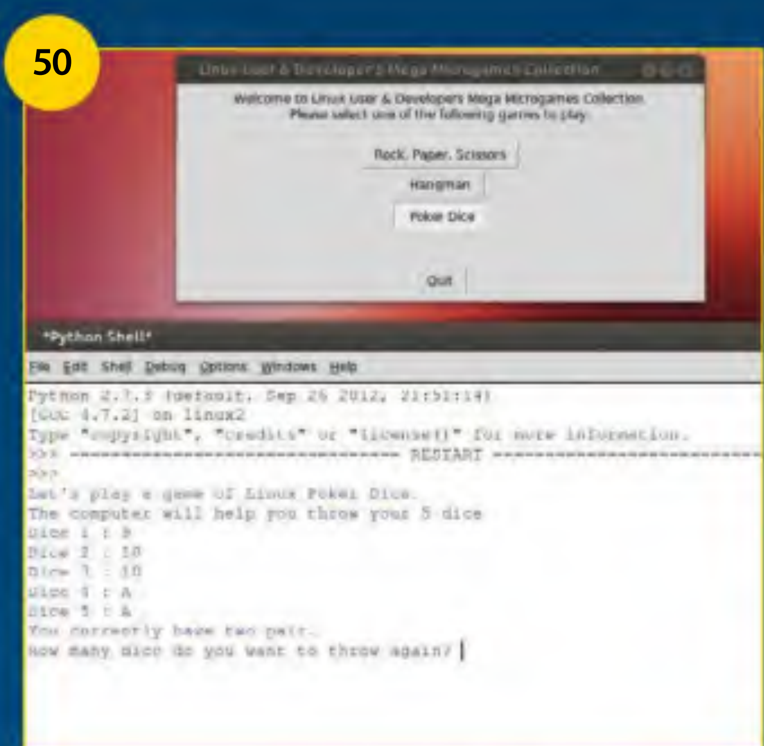
50 String concatenation

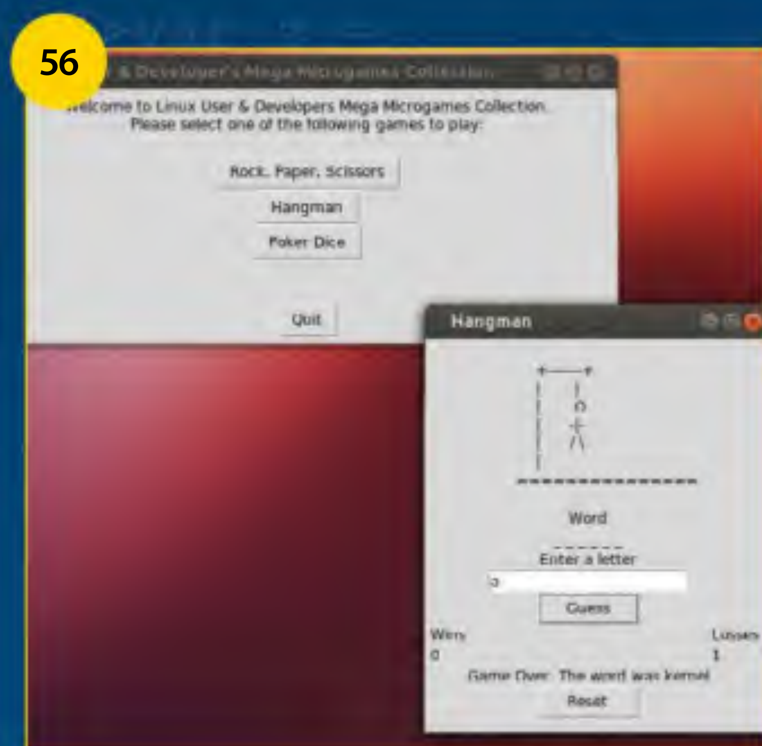
We will finish with what most lists start with – string concatenation. The easiest way to build up strings is to use the '+' operator. If you want to include other items, like numbers, you can use the 'str()' casting function to convert it to a string object.

Python Essentials

"Python is one of the most popular programming languages"




50
Welcome to Linux User & Developers Mega Microgames Collection
Please select one of the following games to play
Rock, Paper, Scissors
Hangman
Poker Dice
Quit
Python Shell
File Edit Shell Debug Options Windows Help
RESTART
Let's play a game of Linux Poker Dice.
The computer will help you throw your 5 dice
You currently have two pair.
How many dice do you want to throw again?


56
& Developer's Mega Microgames Collection
Welcome to Linux User & Developers Mega Microgames Collection.
Please select one of the following games to play:
Rock, Paper, Scissors
Hangman
Poker Dice
Quit
Hangman
Word
Enter a letter
Guess
Wins
0
Losses
1
Game Over. The word was kernel
Reset

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

01 Running Python scripts

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

```
$ python mypyprog.py
```

02 Running Python programs from Python interpreter

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.

03 Dynamic typing

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

04 Python statements

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.

05 == and = operators

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.

06 Concatenating strings

You can use '+' to concatenate strings.

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

07 The __init__ method

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
```

08 Modules

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

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
x,y = my_function.minmax(25, 6.3)
```

09 Module defined names

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']
```

10 Module internal documentation

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 as much precision as the system records.

11 Passing arguments to a Python script

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
```

12 Loading modules or commands at startup

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 .

13 Converting a string to date object

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

Example:

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

14 Converting a list to a string for display

You can convert a list to string in either of the following ways.

1st method:

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

2nd method:

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

15 Tab completion in Python interpreter

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.

16 Python documentation tool

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.

17 Python documentation server

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>
```

18 Python development software

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

19 Executing functions at the time of Python interpreter termination

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
```

20 Converting from integer to binary, hexadecimal and octal

Use 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’
```

21 Converting any charset to UTF-8

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

```
data.decode(“input_charset_here”).
encode(‘utf-8’)
```

22 Removing duplicates from lists

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()
```

23 Do-while loops

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
```

24 Detecting system platform

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’
```

25 Disabling and enabling garbage collection

Sometimes you may want to enable or disable the garbage collector 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>
```

26 Using C-based modules for better performance

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

Example:

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

27 Calculating maximum, minimum and sum out of any list or iterable

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).

28 Representing fractional numbers

Fraction instance can be created using the following constructor:

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

29 Performing math operations

These 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.
```

30 Working with arrays

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.
```

31 Sorting items

The following functions 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.

```
bisect.insort_left(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 left of any existing entries.

32 Using regular expression-based search

You can use the function 're.search()' with a regexp-based expression. Check out the example 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
...
```

33 Working with bzip2 (.bz2) compression format

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'
```

34 Using SQLite database with Python

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

35 Working with zip files

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). '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.

36 Using UNIX-style wildcards to search for filenames

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']
```

37 Performing basic file operations (copy, delete and rename)

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.

38 Executing UNIX commands from Python

This is not available in Python 3 – instead you need to use the module 'subprocess'.

Example:

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

39 Reading environment variables

You can use the module 'os' to gather 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'
```

40 Sending email

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()
```

41 Accessing FTP server

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

```
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 `ftp://archive.download.redhat.com/pub/redhat/linux/`

42 Launching a webpage with the default web browser

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
```

43 Creating secure hashes

The 'hashlib' module supports a plethora of secure hash algorithms.

Example (create hex digest of the given text):

```
>>> import hashlib
# sha1 Digest
>>> hashlib.sha1("MI6 Classified
Information 007").hexdigest()
'e224b1543f229cc0cb935a1eb9593
18ba1b20c85'
# sha224 Digest
>>> hashlib.sha224("MI6 Classified
Information 007").hexdigest()
'3d01e2f741000b0224084482f905e9b7b97
7a59b480990ea8355e2c0'
# sha256 Digest
>>> hashlib.sha256("MI6 Classified
Information 007").hexdigest()
'2fdde5733f5d47b672fcb39725991c89
b2550707cbf4c6403e fdb33b1c19825e'
# sha384 Digest
>>> hashlib.sha384("MI6 Classified
Information 007").hexdigest()
'5c4914160f03dfbd19e14d3ec1e74bd8b99
dc192edc138aaf7682800982488daaf540be
9e0e50fc3d3a65c8b6353572d'
# sha512 Digest
>>> hashlib.sha512("MI6 Classified
Information 007").hexdigest()
'a704ac3dbef6e8234578482a31d5ad29d25
2c822d1f4973f49b850222edcc0a29bb89077
8aea807a0a48ee4ff8bb18566140667fbaf7
3a1dc1ff192febc713d2'
# MD5 Digest
>>> hashlib.md5("MI6 Classified
Information 007").hexdigest()
'8e2f1c52ac146f1a999a670c826f7126'
```

44 Seeding random numbers

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, current system time is used; current system time is also used to initialise the generator when the module is first imported.

45 Working with CSV (comma-separated values) files

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)
```

46 Installing third-party modules using setup tools

'setuptools' is a Python package which lets you download, build, install, upgrade and uninstall packages very easily.

To use 'setuptools' you will need to install from your distribution's package manager. After installation you can use the command 'easy_install' to perform Python package management tasks.

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
```

47 Logging to system log

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

48 Generating PDF documents

'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
```

49 Using Twitter API

You can connect to Twitter 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(username='kunaldeo',password='xxxxxx')
>>> 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',..']
```

50 Doing Yahoo! news search

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
................
```

Code a game of rock, paper, scissors

Learn how to do some basic Python coding by following our breakdown of a simple rock, paper, scissors game

This tutorial will guide you through making a rock, paper, scissors game in Python. The code applies the lessons from the masterclass – and expands on what was included there – and doesn't require any extra Python modules to run, like Pygame.

Rock, paper, scissors is the perfect game to show off a little more about what exactly Python can do. Human input, comparisons, random selections and a whole host of loops are used in making a working version of the game. It's also easy enough to adapt and expand as you see fit, adding rules and results, and even making a rudimentary AI if you wish.

For this particular tutorial, we also recommend using IDLE. IDLE is a great Python IDE that is easily obtainable in most Linux distributions and is available by default on Raspbian for Raspberry Pi. It helps you by highlighting any problems there might be with your code and allows you to easily run it to make sure it's working properly.

Resources

Python 2: www.python.org/download

IDLE: www.python.org/idle

01 This section imports the extra Python functions we'll need for the code – they're still parts of the standard Python libraries, just not part of the default environment

02 The initial rules of the game are created here. The three variables we're using and their relationship is defined. We also provide a variable so we can keep score of the games

03 We begin the game code by defining the start of each round. The end of each play session comes back through here, whether we want to play again or not

04 The game is actually contained all in here, asking for the player input, getting the computer input and passing these on to get the results. At the end of that, it then asks if you'd like to play again

05 Player input is done here. We give the player information on how to play this particular version of the game and then allow their choice to be used in the next step. We also have something in place in case they enter an invalid option

06 There are a few things going on when we show the results. First, we're putting in a delay to add some tension, appending a variable to some printed text, and then comparing what the player and computer did. Through an if statement, we choose what outcome to print, and how to update the scores

07 We now ask for text input on whether or not someone wants to play again. Depending on their response, we go back to the start, or end the game and display the results

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

# Linux User & Developer presents: Rock, Paper, Scissors: The Video Game

import random
import time

rock = 1
paper = 2
scissors = 3

names = { rock: "Rock", paper: "Paper", scissors: "Scissors" }
rules = { rock: scissors, paper: rock, scissors: paper }


player_score = 0
computer_score = 0

def start():
    print "Let's play a game of Rock, Paper, Scissors."
    while game():
        pass
    scores()

def game():
    player = move()
    computer = random.randint(1, 3)
    result(player, computer)
    return play_again()

def move():
    while True:
        print
        player = raw_input("Rock = 1\nPaper = 2\nScissors = 3\nMake a move: ")
        try:
            player = int(player)
            if player in (1,2,3):
                return player
        except ValueError:
            pass
        print "Oops! I didn't understand that. Please enter 1, 2 or 3."

def result(player, computer):
    print "1..."
    time.sleep(1)
    print "2..."
    time.sleep(1)
    print "3!"
    time.sleep(0.5)
    print "Computer threw {0}!".format(names[computer])
    global player_score, computer_score
    if player == computer:
        print "Tie game."
    else:
        if rules[player] == computer:
            print "Your victory has been assured."
            player_score += 1
        else:
            print "The computer laughs as you realise you have been defeated."
            computer_score += 1

def play_again():
    answer = raw_input("Would you like to play again? y/n: ")
    if answer in ("y", "Y", "yes", "Yes", "Of course!"):
        return answer
    else:
        print "Thank you very much for playing our game. See you next time!"

def scores():
    global player_score, computer_score
    print "HIGH SCORES"
    print "Player: ", player_score
    print "Computer: ", computer_score

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

The breakdown

01 We need to start with the path to the Python interpreter here. This allows us to run the program inside a terminal or otherwise outside of a Python-specific IDE like IDLE. Note that we're also using Python 2 rather than Python 3 for this particular script, which needs to be specified in the code to make sure it calls upon the correct version from the system.

02 We're importing two extra modules on top of the standard Python code so we can use some extra functions throughout the code. We'll use the random module to determine what move the computer will throw, and the time module to pause the running of the code at key points. The time module can also be used to utilise dates and times, either to display them or otherwise.

03 We're setting each move to a specific number so that once a selection is made by the player during the game, it will be equated to that specific variable. This makes the code slightly easier later on, as we won't need to parse any text for this particular function. If you so wish, you can add additional moves, and this will start here.

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

# Linux User & Developer presents: Rock, Paper, Scissors: The Video Game

import random
import time

rock = 1
paper = 2
scissors = 3

names = { rock: "Rock", paper: "Paper", scissors: "Scissors" }
rules = { rock: scissors, paper: rock, scissors: paper }

player_score = 0
computer_score = 0
```

04 Here we specify the rules for the game, and the text representations of each move for the rest of the code. When called upon, our script will print the names of any of the three moves, mainly to tell the player how the computer moved. These names are only equated to these variables when they are needed – this way, the number assigned to each of them is maintained while it's needed.

05 Similar to the way the text names of the variables are defined and used only when needed, the rules are done in such a way that when comparing the results, our variables are momentarily modified. Further down in the code we'll explain properly what's happening, but basically after determining whether or not there's a tie, we'll see if the computer's move would have lost to the player move. If the computer move equals the losing throw to the player's move, you win.

06 Very simply, this creates a variable that can be used throughout the code to keep track of scores. We need to start it at zero now so that it exists, otherwise if we defined it in a function, it would only exist inside that function. The code adds a point to the computer or player depending on the outcome of the round, although we have no scoring for tied games in this particular version.

Python modules

There are other modules you can import with basic Python. Some of the major ones are shown to the right. There are also many more that are included as standard with Python.

string	Perform common string operations
datetime and **calendar**	Other modules related to time
math	Advanced mathematical functions
json	JSON encoder and decoder
pydoc	Documentation generator and online help system

07 Here we define the actual beginning of the code, with the function we've called 'start'. It's quite simple, printing our greeting to the player and then starting a while loop that will allow us to keep playing the game as many times as we wish. The pass statement allows the while loop to stop once we've finished, and could be used to perform a number of other tasks if so wished. If we do stop playing the game, the score function is then called upon – we'll go over what that does when we get to it.

08 We've kept the game function fairly simple so we can break down each step a bit more easily in the code. This is called upon from the start function, and first of all determines the player move by calling upon the move function below. Once that's sorted, it sets the computer move. It uses the random module's randint function to get an integer between one and three (1, 3). It then passes the player and computer move, stored as integers, onto the result function which we use to find the outcome.

07

```
def start():
    print "Let's play a game of Rock, Paper, Scissors."
    while game():
        pass
    scores()
```

08

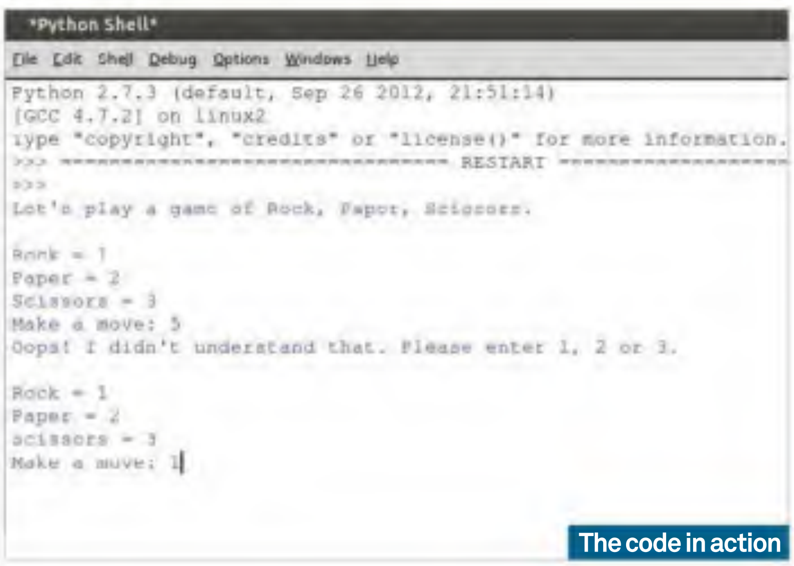
```
def game():
    player = move()
    computer = random.randint(1, 3)
    result(player, computer)
    return play_again()
```

09

```
def move():
    while True:
        print
        player = raw_input("Rock = 1\nPaper = 2\nScissors = 3\nMake a move: ")
```

10

```
        try:
            player = int(player)
            if player in (1,2,3):
                return player
        except ValueError:
            pass
        print "Oops! I didn't understand that. Please enter 1, 2 or 3."
```



The code in action

09 We start the move function off by putting it into a while loop. The whole point of move is to obtain an integer between one and three from the player, so the while loop allows us to account for the player making an unsupported entry. Next, we are setting the player variable to be created from the player's input with raw_input. We've also printed instruction text to go along with it. The '\n' we've used in the text adds a line break; this way, the instructions appear as a list.

10 The try statement is used to clean up code and handle errors or other exceptions. We parse what the player entered by turning it into an integer using int(). We use the if statement to check if it is either 1, 2, or 3 – if it is, move returns this value back up to the game function. If it throws up a ValueError, we use except to do nothing. It prints an error message and the while loop starts again. This will happen until an acceptable move is made.

11 The result function only takes the variables player and computer for this task, which is why we set that in result(player, computer). We're starting off by having a countdown to the result. The printed numbers are self-explanatory, but we've also thrown in sleep from the time module we imported. Sleep pauses the execution of the code by the number of seconds in the brackets. We've put a one-second pause between counts, then half a second after that to show the results.

12 To print out what the computer threw, we're using string.format(). The {0} in the printed text is where we're inserting the move, which we have previously defined as numbers. Using names[computer], we're telling the code to look up what the text version of the move is called from the names we set earlier on, and then to insert that where {0} is.

13 Here we're simply calling the scores we set earlier. Using the global function allows for the variable to be changed and used outside of the variable, especially after we've appended a number to one of their scores.

```
def result(player, computer):
    print "1..."
    time.sleep(1)
    print "2..."
    time.sleep(1)
    print "3!"
    time.sleep(0.5)
    print "Computer threw {0}!".format(names[computer])
    global player_score, computer_score
    if player == computer:
        print "Tie game."
    else:
        if rules[player] == computer:
            print "Your victory has been assured."
            player_score += 1
        else:
            print "The computer laughs as you realise you have been defeated."
            computer_score += 1
```

14 The way we're checking the result is basically through a process of elimination. Our first check is to see if the move the player and computer used were the same, which is the simplest part. We put it in an if statement so that if it's true, this particular section of the code ends here. It then prints our tie message and goes back to the game function for the next step.

15 If it's not a tie, we need to keep checking, as it could still be a win or a loss. Within the else, we start another if statement. Here, we use the rules list from earlier to see if the losing move to the player's move is the same as the computer's. If that's the case, we print the message saying so, and add one to the player_score variable from before.

16 If we get to this point, the player has lost. We print the losing message, give the computer a point and it immediately ends the result function, returning to the game function.

Python Shell

File Edit Shell Debug Options Windows Help

```
Python 2.7.3 (default, Sep 26 2012, 21:51:14)
[GCC 4.7.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
Let's play a game of Rock, Paper, Scissors.

Rock - 1
Paper - 2
Scissors - 3
Make a move: 5
Oops! I didn't understand that. Please enter 1, 2 or 3.

Rock - 1
Paper - 2
Scissors - 3
Make a move: 1
1...
2...
3!
Computer threw Rock!
Tie game.
```

The code in action

17 The next section of game calls upon a play_again function. Like the move function, we have human input, asking the player if they would like to play again via a text message with raw_input, with the simple 'y/n' suggestion in an attempt to elicit an expected response.

18 Giving users an option of y/n like we have should expect a response in kind. The if statement checks to see if any of our defined positive responses have been entered. As Python doesn't differentiate between upper or lower case, we've made sure that it accepts both y and Y. If this is the case, it returns a positive response to game, which will start it again.

19 If we don't get an expected response, we will assume the player does not want to play again. We'll print a goodbye message, and that will end this function. This will also cause the game function to move onto the next section and not restart.

```
17 def play_again():
       answer = raw_input("Would you like to play again? y/n: ")
18     if answer in ("y", "Y", "yes", "Yes", "Of course!"):
           return answer
19     else:
           print "Thank you very much for playing our game. See you next time!"

20 def scores():
       global player_score, computer_score
       print "HIGH SCORES"
       print "Player: ", player_score
       print "Computer: ", computer_score

21 if __name__ == '__main__':
       start()
```

```
Python Shell
File Edit Shell Debug Options Windows Help
Python 2.7.3 (default, Sep 26 2012, 21:51:14)
[GCC 4.7.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
Let's play a game of Rock, Paper, Scissors.

Rock = 1
Paper = 2
Scissors = 3
Make a move: 5
Oops! I didn't understand that. Please enter 1, 2 or 3.

Rock = 1
Paper = 2
Scissors = 3
Make a move: 1
1...
2...
3!
Computer threw Rock!
Tie game.
Would you like to play again? y/n: n
Thank you very much for playing our game. See you next time!
HIGH SCORES
Player:  0
Computer:  0
>>>
```

The code in action

ELIF

IF also has the ELIF (else if) operator, which can be used in place of the second IF statement we employed. It's usually used to keep code clean, but performs the same function.

20 Going back to the start function, after game finishes we move onto the results. This section calls the scores, which are integers, and then prints them individually after the names of the players. This is the end of the script, as far as the player is concerned. Currently, the code won't permanently save the scores, but you can have Python write it to a file to keep if you wish.

21 The final part allows for the script to be used in two ways. Firstly, we can execute it in the command line and it will work fine. Secondly, we can import this into another Python script, perhaps if you wanted to add it as a game to a collection. This way, it won't execute the code when being imported.

Program a game of Hangman

Learn how to do some more Python coding by following our breakdown of a simple Hangman game

This section imports the extra Python functions we'll need for the code – they're still parts of the standard Python libraries, just not part of the default environment

We're again providing variables so we can keep score of the games played, and they're updated each round

Our very basic graphics involve ASCII art of the game's stages, printed out after every turn

Code listing

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

from random import *

player_score = 0
computer_score = 0

def hangedman(hangman):
    graphic = [
    """
        +-------+
        |
        |
        |
        |
        |
     ==============
    """,
    """
        +-------+
        |       |
        |       O
        |
        |
        |
    ===============
    """,
```

```
    """,
    """
        +-------+
        |       |
        |       O
        |      -|-
        |      / \
        |
    ===============
    """]

    print graphic[hangman]
    return
```

Resources

Python 2: www.python.org/download

IDLE: www.python.org/idle

One of the best ways to get to know Python is by building lots of simple projects so you can understand a bit more about the programming language. This time round, we're looking at Hangman, a multi-round game relying on if and while loops and dealing with strings of text in multiple ways. We'll be using some of the techniques we implemented last time as well, so we can build upon them.

Hangman still doesn't require the Pygame set of modules, but it's a little more advanced than rock-paper-scissors. We're playing around with a lot more variables this time. However, we're still looking at comparisons, random selections and human input, along with splitting up a string, editing a list and even displaying rudimentary graphics.

You should continue to use IDLE for these tutorials. As we've mentioned before, its built-in debugging tools are simple yet effective and it can be used on any Linux system, as well as the Raspberry Pi.

The actual game starts here, with a while loop to let you continually play the game until you decide otherwise, then ending the program

The game rules are decided here, as well as the setup for the word and keeping track of tries and incorrect answers

Each round of the game is played here, asking for an input, then telling you if you were correct or not. It prints out the graphic and changes any variables that need to be updated, especially incorrect and correct guesses

After each round, the code checks if you've won or lost yet – the win condition being that you guessed the word, or losing if you've made six guesses

The human input for the game takes the letter and turns it into something the code can use. It's verified in the previous block of code and then referred back to if you've entered an unsupported or already used character

The same class as last time, which allows you to select whether or not you wish to play again

Upon quitting the game, scores are given for the duration of the play session. We also end the script with the if __name__ code like before

Code highlighting

IDLE automatically highlights the code to make reading your work that bit easier. It also allows you to change these colours and highlighting in IDLE's Preferences, in case you're colour blind or are just used to a different colour scheme in general.

Code listing continued

```
def start():
    print "Let's play a game of Linux Hangman."
    while game():
        pass
    scores()

def game():
    dictionary = ["gnu","kernel","linux","mageia","penguin","ubuntu"]
    word = choice(dictionary)
    word_length = len(word)
    clue = word_length * ["_"]
    tries = 6
    letters_tried = ""
    guesses = 0
    letters_right = 0
    letters_wrong = 0
    global computer_score, player_score

    while (letters_wrong != tries) and ("".join(clue) != word):
        letter=guess_letter()
        if len(letter)==1 and letter.isalpha():
            if letters_tried.find(letter) != -1:
                print "You've already picked", letter
            else:
                letters_tried = letters_tried + letter
                first_index=word.find(letter)
                if  first_index == -1:
                    letters_wrong +=1
                    print "Sorry,",letter,"isn't what we're looking for."
                else:
                    print"Congratulations,",letter,"is correct."
                    for i in range(word_length):
                        if letter == word[i]:
                            clue[i] = letter
        else:
            print "Choose another."

        hangedman(letters_wrong)
        print " ".join(clue)
        print "Guesses: ", letters_tried

        if letters_wrong == tries:
            print "Game Over."
            print "The word was",word
            computer_score += 1
            break
        if "".join(clue) == word:
            print "You Win!"
            print "The word was",word
            player_score += 1
            break
    return play_again()

def guess_letter():
    print
    letter = raw_input("Take a guess at our mystery word:")
    letter.strip()
    letter.lower()
    print
    return letter

def play_again():
    answer = raw_input("Would you like to play again? y/n: ")
    if answer in ("y", "Y", "yes", "Yes", "Of course!"):
        return answer
    else:
        print "Thank you very much for playing our game. See you next time!"

def scores():
    global player_score, computer_score
    print "HIGH SCORES"
    print "Player: ", player_score
    print "Computer: ", computer_score

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

I see ASCII

Here's a close-up of the seven stages we've used for Hangman's graphics. You can change them yourself, but you need to make sure the quote marks are all in the correct place so that the art is considered a text string to be printed out.

```
“””
     +-------+
     |
     |
     |
     |
     |
  ==============
“””,
“””
     +-------+
     |       |
     |       O
     |
     |
     |
===============
“””,
“””
     +-------+
     |       |
     |       O
     |       |
     |
     |
===============
“””,
“””
     +-------+
     |       O
     |      -|
     |
     |
     |
===============
“””,
“””
     +-------+
     |       |
     |       O
     |      -|-
     |
     |
===============
“””,
“””
     +-------+
     |       |
     |       O
     |      -|-
     |      /
     |
===============
“””,
“””
     +-------+
     |       |
     |       O
     |      -|-
     |      / \
     |
===============
“””]
```

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

02 from random import *

03 player_score = 0
   computer_score = 0

04 def hangedman(hangman):
       graphic = [
       “””
            +-------+
            |
            |
            |
            |
            |
         ==============
       “””,
       “””

05 def start():
       print “Let’s play a game of Linux Hangman.”
       while game():
           pass
       scores()
```

The rules

Although we've moved some of the rules to the 'game' function this month, you can always put them back here and call upon them using the global variable, as we would do with the scores. For the words, you could also create a separate file and import them like the random module.

01 We begin by using this line to enter the path to the Python interpreter. This allows us to run the program inside a terminal or otherwise outside of a Python-specific IDE like IDLE. Note that we're also using Python 2 for this particular script, as it is installed by default on most Linux systems and will therefore ensure compatibility.

02 We're importing the 'random' module slightly differently this time, importing the actual names of the functions from random rather than just the module itself. This allows us to use the functions without having syntax like random.function. The asterisk imports all the functions from random, although you can switch that for specific names of any of random's functions. We'll be using the random function to select a word for the player to guess.

03 Very simply, this creates a variable that can be used throughout the code to keep track of scores. We need to start it at zero now so that it exists; otherwise if we defined it in a function, it would only exist inside that function. The code adds a point to the computer or player depending on the outcome of the round.

04 Our simple graphics consist of a series of ASCII hanging man stages. We're storing these in a function as a list of separate string objects so we can call upon them by passing on the number of incorrect guesses to it. There are seven graphics in all, like in the pen-and-paper version. We also include the print command with the function, so when it's called it will completely handle the selection and display of the hanging man, with the first one being printed after the first letter is guessed.

05 Here we define the actual beginning of the code, with the function we've called 'start'. It's quite simple, printing our greeting to the player and then starting a while loop that will allow us to keep playing the game as many times as we wish. The pass statement allows the while loop to stop once we've finished, and could be used to perform a number

```
def game():
    dictionary = ["gnu","kernel","linux","mageia","penguin","ubuntu"]
    word = choice(dictionary)
    word_length = len(word)
    clue = word_length * ["_"]
    tries = 6
    letters_tried = ""
    guesses = 0
    letters_right = 0
    letters_wrong = 0
    global computer_score, player_score

    while (letters_wrong != tries) and ("".join(clue) != word):
        letter=guess_letter()
        if len(letter)==1 and letter.isalpha():
            if letters_tried.find(letter) != -1:
                print "You've already picked", letter
```

of other tasks if so wished. If we do stop playing the game, the score function is then called upon –we'll go over what that does when we get to it.

06 We have put a majority of the game code in the 'game' function this time around, as there's not as much that needs to be split up. You can split it up further if you wish, using the style of code from last issue, if it would make the code cleaner for you or help you understand the building blocks a bit more.

07 The first four lines quickly set up the word for the player to guess. We've got a small selection of words in a list here. However, these can be imported via HTML or expanded upon. Choice is used to select a random element from the list, which comes from the random module we imported. Finally, we ascertain how long the string is of the word to guess, and then create the clue variable with a number of underscores of that length. This is used to display the word as you build it up from guesses.

08 We start to set up the rules and the individual variables to keep track of during the game. There can only be six incorrect guesses before the hanging man is fully drawn, or in our case displayed, so we set the tries variable to six. We'll keep track of the letters through letters_tried to make sure that not only will the player know, but also the code for when it's checking against letters already played. Finally, we create empty variables for the number of guesses made, letters correct and letters incorrect, to make the code slightly easier. We also import the global scores here.

09 We're starting a while loop to perform the player selection and check the status of the game. This loop continues until the player wins or loses. It starts by checking if all the tries have been used up by seeing if letters_wrong is not equal to tries. As each try will only add one point to wrong, it will never go above six. It then concatenates 'clue' and sees if it's the same as the word the computer selected. If both these statements are true, it goes on to the next turn.

10 We call upon the function we're using to input a letter and give it the variable 'letter'. We check what it returns by first of all making sure it's only a single letter, with len(letter), then by using isalpha to see if it's one of the 26 letters of the alphabet. If these conditions are satisfied, we start a new if statement to make sure it's a new guess, and tell the player if it's already been chosen so they can start again. If all this is acceptable, we move on to the next section of the code to see if it's a correct guess or not.

Indentations

While IDLE will keep track of the indents in the code, if you're using a text editor to write some Python, you'll have to make sure you're using them correctly. Python is very sensitive to whether or not indents are used correctly, and it does aid in readability as well.

Continuation

This code is still part of the game function we started on the previous page, so make sure your indentations are in alignment if you're not using an IDE. If you plan to split this code up, we'd suggest starting with the word selection and results.

```
        11 else:
                letters_tried = letters_tried + letter
                first_index=word.find(letter)
                if  first_index == -1:
                    letters_wrong +=1
                    print "Sorry,",letter,"isn't what we're looking for."
                else:
                    print"Congratulations,",letter,"is correct." 12
                 13 for i in range(word_length):
                        if letter == word[i]:
                            clue[i] = letter
        else:
            print "Choose another." 14

        hangedman(letters_wrong)
        print " ".join(clue)
        print "Guesses: ", letters_tried

     15 if letters_wrong == tries:
            print "Game Over."
            print "The word was",word
            computer_score += 1
            break
        if "".join(clue) == word:
            print "You Win!"
            print "The word was",word
            player_score += 1
            break
    return play_again() 16
```

11 If it's a new letter that we find acceptable, the first thing we do is add it to the list of letters tried. This is done simply by adding the strings together. We then use the find command to search the word string for the letter entered, which will then return a number of the placement of the letter in the string. If it doesn't find the letter, it returns a -1 value, which we use in the next if statement to see if the first_index variable is -1. If so, it adds one to the number of letters_wrong and then prints a message to let the player know that it was an incorrect guess.

12 If we've got this far and the letter is not incorrect, than we can only assume it is correct. Through this simple process of elimination, we first print out a message to let the player know that they've been successful and then make a record of it.

13 We're going to start a small loop here so we can update the clue with the correct letter we've added. We use the range function to tell the code how many times we wish to iterate over the clue by using the word_length variable. We then check to see which letter in the word has been guessed correctly and change that specific part of the clue to be that letter so it can be printed out for the player to see, and for us to check whether or not the game is over.

14 We end the original if statement by telling the player to choose again if they did not enter a supported input. Before we go on to the next round of choices, we print out the hanging man graphic as it stands, by calling the graphic in the list that corresponds to the number of incorrect guesses that have been made. We then print how the clue currently looks, with a space in between each character, and then print the number of guesses that have been made.

15 Here we check to see if the game is over again, first of all comparing the letters_wrong to the number of tries. If that's true, we print a message that the game has ended and reveal the mystery of the hidden word. We increase the computer's score and break the loop. The next loop checks to see if the full clue concatenated is the same as the original word – if that's the case, we print the win message, the full word and add one point to the player score before breaking the loop again. This can also be done with ifs and elifs to avoid using breaks.

```
17  def guess_letter():
        print
        letter = raw_input("Take a guess at our mystery word:")
        letter.strip()
        letter.lower()
        print
        return letter

18  def play_again():
        answer = raw_input("Would you like to play again? y/n: ")
19      if answer in ("y", "Y", "yes", "Yes", "Of course!"):
            return answer
20      else:
            print "Thank you very much for playing our game. See you next time!"

21  def scores():
        global player_score, computer_score
        print "HIGH SCORES"
        print "Player: ", player_score
        print "Computer: ", computer_score

22  if __name__ == '__main__':
        start()
```

16 We end the entire game function loop by calling upon return again, which we will then pass all the way up to the start function once it's finished.

17 The human input function first of all prints out a raw_input message. Once the player enters the letter, the function parses it to be used with the rest of the code. Firstly, strip is used to remove any white space from the input given, as we've not given it any extra parameters. We then convert it into lower-case letters, as Python will not be able to correctly compare an upper-case character with a lower-case alternative. We then print the selection for the record and return it up to the game function.

18 The last part of the game function is to ask the player if they wish to try again. The play_again function takes a human input with a simple message and then analyses the input so it knows what to send back.

19 Giving users an option of y/n like we have should expect a response in kind. The if statement checks to see if any of our defined positive responses have been entered. As Python doesn't differentiate between upper or lower case, we've made sure it accepts both y and Y. If this is the case, it returns a positive response to game, which will start it again.

20 If we don't get an expected response, we will assume the player does not want to play again. We'll print a goodbye message and that will end this function. This will also cause the start function to move onto the next section and not restart.

21 Going all the way back to the start function, after game finishes we move onto the results. This section is quite simple – it calls the scores, which are integers, and then prints them individually after the names of the players. This is the end of the script, as far as the player is concerned. Currently, the code will not permanently save the scores, but you can have Python write it to a file to keep if you wish.

22 The final part of the code allows for the script to be used in two ways. Firstly, we can execute it in the command line and it will work fine. Secondly, we can import this into another Python script, perhaps if you wanted to add it as a game to a collection. This way, it will not execute the code when being imported.

Homework

Now that you've finished with the code, why not make your own changes? Increase the word count; create different, selectable word categories; or even let people guess the full word. You have all the tools to do this in the current code and last month's tutorial.

Play poker dice using Python

Put on your poker face and get ready to gamble as you hone your programming skill with a bit of poker dice

So you've learnt how to program tic-tac-toe and guessed your way to victory at hangman. Now it's time to head to Las Vegas and play our cards right. Or in this case, virtual dice, and more like Reno as we continue with our Python game tutorials and introduce you to some poker dice.

We're again using some of the lessons we've already learnt, including random number generation, list creation and modification, human input, rule setting, scoring and more. But we'll also be adding some new skills in this tutorial. Namely, we'll be creating and appending lists with random numbers, and using functions multiple times in one block of code to cut down on bloat.

Again, we recommend using IDLE, and we're using Python 2 to ensure compatibility with a wider variety of distros, including the Raspberry Pi. So, we hope luck is a lady for you and that the odds are ever in your favour – just keep those fingers crossed that you don't roll a snake eyes (we are coding in Python, after all)!

Resources

Python 2: www.python.org/download

IDLE: www.python.org/idle

The Start
Here we're doing some minor setups so we can get our code to run with some extra modules not included with the basics

The Rules
We're setting names for each dice roll so they can be properly identified to the player – much more interesting than numbers

The Score
Again we've got some basic variables set up so we can keep score of the games if we want to

The Script
The game is handled here, passing the player onto the next function to actually play, and handling the end of the session as well

The Game
We access the full game loop via here, and the function that allows us to play again if we're so inclined

The Throw
The initial hand is dealt, so to speak, at the start of the throws function. This function handles all the decision making in the game, while passing off the dice rolls to another function

The Hand
We've also got a special function so we can inform the player exactly what style of hand they have

The Decision
There are two rounds in this version of poker dice, and you can select how many dice you wish to re-roll in this small while loop that makes sure you're also using a correct number

Code listing

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

import random
from itertools import groupby

nine = 1
ten = 2
jack = 3
queen = 4
king = 5
ace = 6

names = { nine: "9", ten: "10", jack: "J", queen: "Q", king: "K", ace: "A" }

player_score = 0
computer_score = 0

def start():
    print "Let's play a game of Linux Poker Dice."
    while game():
        pass
    scores()

def game():
    print "The computer will help you throw your 5 dice"
    throws()
    return play_again()

def throws():
    roll_number = 5
    dice = roll(roll_number)
    dice.sort()
    for i in range(len(dice)):
        print "Dice",i + 1,":",names[dice[i]]

    result = hand(dice)
    print "You currently have", result

    while True:
        rerolls = input("How many dice do you want to throw again? ")
        try:
            if rerolls in (1,2,3,4,5):
                break
        except ValueError:
            pass
        print "Oops! I didn't understand that. Please enter 1, 2, 3, 4 or 5."
```

The Re-roll
We're doing the second set of rolls and starting the end of the game here by calling on the same function as before, but we're also aware that choosing no re-rolls means the end of the game

The Dice
Here we're finding out which dice the player wants to re-roll, and also making sure that they enter a valid number. Just so they know they're doing something, we print something after every turn

Second Hand
We change and display the new dice hand to end the game. Again, we make sure to tell the player what the actual hand they have is

The Rolls
The function we reuse to roll our virtual six dice using a simple while loop. This allows us to keep the codebase smaller

The Analysis
There are eight possible types of hands in poker dice, and we can use a bit of logic to work out all but one of them without checking against all 7,776 outcomes – in fact, we only specifically have to check for two

The Question
Our simple 'play again' function that parses player input so we can restart or end the script

The End
Scores are displayed at the end of the script, and the very final part allows us to import this into other Python scripts as a module

EXTRA FUNCTIONS
Splitting up actions into functions makes it easier to not only perform them multiple times, but reduce the amount of code. On larger projects, this can aid with speed.

Code listing continued

```
    if rerolls == 0:
        print "You finish with", result
    else:
        roll_number = rerolls
        dice_rerolls = roll(roll_number)
        dice_changes = range(rerolls)
        print "Enter the number of a dice to reroll: "
        iterations = 0
        while iterations < rerolls:
            iterations = iterations + 1
            while True:
                selection = input("")
                try:
                    if selection in (1,2,3,4,5):
                        break
                except ValueError:
                    pass
                print "Oops! I didn't understand that. Please enter 1, 2, 3, 4 or 5."
            dice_changes[iterations-1] = selection-1
            print "You have changed dice", selection

        iterations = 0
        while iterations < rerolls:
            iterations = iterations + 1
            replacement = dice_rerolls[iterations-1]
            dice[dice_changes[iterations-1]] = replacement

        dice.sort()
        for i in range(len(dice)):
            print "Dice",i + 1,":",names[dice[i]]

        result = hand(dice)
        print "You finish with", result

def roll(roll_number):
    numbers = range(1,7)
    dice = range(roll_number)
    iterations = 0
    while iterations < roll_number:
        iterations = iterations + 1
        dice[iterations-1] = random.choice(numbers)
    return dice

def hand(dice):
    dice_hand = [len(list(group)) for key, group in groupby(dice)]
    dice_hand.sort(reverse=True)
    straight1 = [1,2,3,4,5]
    straight2 = [2,3,4,5,6]

    if dice == straight1 or dice == straight2:
        return "a straight!"
    elif dice_hand[0] == 5:
        return "five of a kind!"
    elif dice_hand[0] == 4:
        return "four of a kind!"
    elif dice_hand[0] == 3:
        if dice_hand[1] == 2:
            return "a full house!"
        else:
            return "three of a kind."
    elif dice_hand[0] == 2:
        if dice_hand[1] == 2:
            return "two pair."
        else:
            return "one pair."
    else:
        return "a high card."

def play_again():
    answer = raw_input("Would you like to play again? y/n: ")
    if answer in ("y", "Y", "yes", "Yes", "Of course!"):
        return answer
    else:
        print "Thank you very much for playing our game. See you next time!"

def scores():
    global player_score, computer_score
    print "HIGH SCORES"
    print "Player: ", player_score
    print "Computer: ", computer_score

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

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

import random
from itertools import groupby

nine = 1
ten = 2
jack = 3
queen = 4
king = 5
ace = 6

names = { nine: "9", ten: "10", jack: "J", queen: "Q", king: "K", ace: "A" }

player_score = 0
computer_score = 0

def start():
    print "Let's play a game of Linux Poker Dice."
    while game():
        pass
    scores()

def game():
    print "The computer will help you throw your 5 dice"
    throws()
    return play_again()
```

RECYCLING

There are a few variables that have duplicates throughout the code – while we've been careful to make sure they work where we want them to, it's not the best code conduct. The names of the variables don't specifically matter – it's just best to label them in a way you understand for bug fixing and others to read.

01 Begin

As before, we use this line to enter the path to the Python interpreter. This allows us to run the program inside a terminal or otherwise outside of a Python-specific IDE like IDLE. Note that we're also using Python 2 for this script.

02 Importing

As well as importing the random module for our dice throws, we need to get the groupby function so we can order the dice in a way that is more readable and also easier for analysis when telling the player what hand they have.

03 Cards

While we're using random numbers for the dice rolls, unless we assign the correct cards to each number, the player won't know what they've rolled and what constitutes a better hand. We set each card to a number and then equate what these should be printed out as.

04 Scores

As usual, we have the empty scores for the player and computer so we can update these as we go. While it's not specifically used in this version of the code, it's easy enough to expand on it and add your own simple computer roll, or limited AI for both rolls.

05 Start

We're starting the interactive part of the code with the 'start' function. It prints a greeting to the player, then starts a while loop that'll allow us to replay the game as many times as we wish. The pass statement allows the while loop to stop once we've finished. If we do stop playing the game, the score function is then called upon.

06 Game

Like our Rock, Paper, Scissors code, def game pawns the rest of the game onto other functions, with its main function allowing us to keep repeating the game by passing the player through to the play_again function.

07 Throws

For our first throw, we want to have five random dice. We've set a variable here to pass on to our throwing function, allowing us to reuse it later with a different number that the player chooses. We get five random numbers in a list returned from the function, and we order it using sort to make it a bit more readable for the player and also later on for the hand function.

08 Dice display

We print out each dice, numbering them so the player knows which dice is which, and also giving it the name we set at the start of the script. We're doing this with a loop that repeats itself the number of times as the dice list is long using the range(len(dice)) argument. The i is increased each turn, and it prints out that specific number of the dice list.

09 Current hand

We want to find the type of hand the player has multiple times during the game, so set a specific function to find out. We pass the series of dice we have on to this function, and print.

10 Throw again

Before we can throw the dice for the second round, we need to know which dice the

```
07  def throws():
        roll_number = 5
        dice = roll(roll_number)
        dice.sort()
08      for i in range(len(dice)):
            print "Dice",i + 1,":",names[dice[i]]

09      result = hand(dice)
        print "You currently have", result

10      while True:
            rerolls = input("How many dice do you want to throw again? ")
            try:
                if rerolls in (1,2,3,4,5):
                    break
            except ValueError:
                pass
            print "Oops! I didn't understand that. Please enter 1, 2, 3, 4 or 5."
11      if rerolls == 0:
            print "You finish with", result
12      else:
            roll_number = rerolls
            dice_rerolls = roll(roll_number)
            dice_changes = range(rerolls)
            print "Enter the number of a dice to reroll: "
            iterations = 0
            while iterations < rerolls:
                iterations = iterations + 1
                while True:
                    selection = input("")
                    try:
                        if selection in (1,2,3,4,5):
                            break
                    except ValueError:
                        pass
13                  print "Oops! I didn't understand that. Please enter 1, 2, 3, 4 or 5."
                dice_changes[iterations-1] = selection-1
                print "You have changed dice", selection
```

INDENTATIONS

Watch the indentations again as we split the else function. The following page's code is on the same level as roll roll_number, dice_rerolls and dice_changes in the code.

WHITE SPACE

The big if function at the end of throws doesn't have many line breaks between sections – you can add these as much as you want to break up the code into smaller chunks visually, aiding debugging.

player wants to roll again. We start this by asking them how many re-rolls they want to do, which allows us to create a custom while loop to ask the user which dice to change that iterates the correct number of times.

We also have to make sure it's a number within the scope of the game, which is why we check using the try function, and print out a message which tells the user if and how they are wrong.

11 Stick

One of the things we've been trying to do in these tutorials is point out how logic can cut down on a lot of coding by simply doing process of eliminations or following flow charts. If the user wants to re-roll zero times, then that means they're happy with their hand, and it must be the end of the game. We print a message to indicate this and display their hand again.

12 The re-rolls

Here's where we start the second roll and the end of the game, using a long else to the if statement we just started. We first of all make sure to set our variables – updating roll_number to pass onto the roll function with the re-roll number the user set, and creating the list that's the exact length of the new set of rolls we wish to use thanks to range(rerolls).

13 Parse

We ask the player to enter the numbers of the dice they wish to re-roll. By setting an iterations variable, we can have the while loop last the same number of times as we want re-rolls by comparing it to the reroll variable itself. We check each input to make sure it's a number that can be used, and add the valid choices to the dice_changes list. We use iterations-1 here as Python lists begin at 0 rather than 1. We also print out a short message so the player knows the selection was successful.

```
14  iterations = 0
    while iterations < rerolls:
        iterations = iterations + 1
        replacement = dice_rerolls[iterations-1]
        dice[dice_changes[iterations-1]] = replacement

15  dice.sort()
    for i in range(len(dice)):
        print "Dice",i + 1,":",names[dice[i]]

    result = hand(dice)
    print "You finish with", result

16 def roll(roll_number):
    numbers = range(1,7)
17  dice = range(roll_number)
    iterations = 0
18  while iterations < roll_number:
        iterations = iterations + 1
        dice[iterations-1] = random.choice(numbers)
    return dice
```

HIGHER OR LOWER

Which hand is best? What are the odds of getting certain hands in the game? Some of the answers are surprising, as the poker hands they're based on trump the differing odds the dice produce. We've ranked hands from highest to lowest.

Five of a Kind	6/7776
Four of a Kind	150/7776
Full House	300/7776
Straight	240/7776
Three of a Kind	1200/7776
Two Pairs	1800/7776
One Pair	3600/7776
High Card	480/7776

14 New dice
We're resetting and reusing the iterations variable to perform a similar while loop to update the rolls we've done to the original dice variable. The main part of this while loop is using the iterations-1 variable to find the number from dice_changes list, and using that to change that specific integer in the dice list with the number from the replacement list. So if the first item on the dice_changes list is two, then the second item on the dices list is changed to the number we want to replace it with.

15 Sorting
We're ending the throw function in basically the same way we ended the first throw. First of all, we re-sort the dice list so that all the numbers are in ascending order. Then we print out the final cards that the dice correspond to, before again passing it onto the hand function so that we can fully determine the hand that the player has. We print out this result and that ends the function, sending the whole thing back to the game function to ask if you want to play again.

16 Dice rolling
The roll function is used twice in the code for both times that we roll the dice. Being able to use the same code multiple times means we can cut down on bloat in the rest of the script, allowing it to run a little faster, as we've explained. It also means in this case that we can use it again if you want to change the game to three rounds, or modify it for real poker.

17 Number of rolls
We begin the whole thing by bringing over the roll_number variable into the function – this is because while in the original roll it will always be five, the second roll could between one and the full five dice. We create a list with the number of entries we need for each roll, and again set an iterations variable for the upcoming while loop.

18 Remember
Much like the while loops in the rest of the code so far, we're keeping it going until iterations is the same as roll_number. Each entry in the dice list is replaced with a random number using the random.choice function and keeping it in the range of the numbers variable, which is one to six for each side of the dice. After this is done, we return the dice variable to the throw function that makes up the majority of the game.

19 Hand analysis
While not technically a hand of cards, the poker terminology still applies. We start in this function by setting up a few things. The first part uses the groupby function we imported – this is used in this case to count the numbers that make up the dice variable. If there are three twos, a four and a five, it will return [3, 1, 1]. We're using this to ascertain what kind of hand the player has. As the output of this groupby won't be in any specific order, we use the sort function again to sort it; however, this time we use the reverse=TRUE argument to make the analysis easier again.

20 Straights
Straights and high cards are odd ones out in poker dice, as they do not rely on being able to count any repetitions in the cards. There are, however, only two hands that create a straight in poker dice, so we have created two lists here that contain them. We can then check first to see if the dice make these hands, and then if all other checks fail, it has to be a high card.

21 Your hand
While seemingly lengthy, this a fairly simple if statement. As we stated before, we check to see if it's one of the two straight hands. As there are no flushes or royal straight flushes in poker dice, we don't have to worry about those. We then check to see if the first item in the list is five, which can only result in five of a kind; similarly, if the first item is four then the hand must be four of a kind. If the first number is three, then it can be either a full house or three of a kind,

```
19 def hand(dice):
       dice_hand = [len(list(group)) for key, group in groupby(dice)]
       dice_hand.sort(reverse=True)
20     straight1 = [1,2,3,4,5]
       straight2 = [2,3,4,5,6]

21     if dice == straight1 or dice == straight2:
           return "a straight!"
       elif dice_hand[0] == 5:
           return "five of a kind!"
       elif dice_hand[0] == 4:
           return "four of a kind!"
       elif dice_hand[0] == 3:
           if dice_hand[1] == 2:
               return "a full house!"
           else:
               return "three of a kind."
       elif dice_hand[0] == 2:
           if dice_hand[1] == 2:
               return "two pair."
           else:
               return "one pair."
       else:
           return "a high card."

22 def play_again():
       answer = raw_input("Would you like to play again? y/n: ")
       if answer in ("y", "Y", "yes", "Yes", "Of course!"):
           return answer
       else:
           print "Thank you very much for playing our game. See you next time!"

23 def scores():
       global player_score, computer_score
       print "HIGH SCORES"
       print "Player: ", player_score
       print "Computer: ", computer_score

24 if __name__ == '__main__':
       start()
```

TEXT EDITORS

Instead of the IDE we've suggested, you should also try coding in a text editor. Some of them are a little more lightweight and format code similar to the way the IDE does, separating functions and strings by colours etc. Some of the ones we'd recommend are the classic gedit, a popular text editor from GNOME desktops; Geany, which has a few IDE-esque features written into it; TEA, a multifunctioning text editor and project manager; and Jedit, a text editor that lives in the command line for minimum resource usage. These can also be used with multiple programming languages, so you can get used to them with Python, then make the switch.

HOMEWORK

There is currently no scoring in place for this version of the game. Try adding a computer player, or create a rule set that requires a certain hand or higher. You could even make it two-player.

so we nest an if statement. Again, we do this for pairs, where that could be one or two pairs. If all else fails then, by a process of elimination, it can only be a high card. We give each outcome a text string to send back to the throw function so that it can be printed.

22 Play again

As before, we ask the player for raw input with the text offering another game. Instead of parsing it, we assume the player will choose a specified yes response based on the text, and if none of these versions is received, we print out the message thanking them for playing the game. This ends the game function.

23 Final scores

Going all the way back to the start function, after the game finishes we move onto the results. This section is quite simple – it calls the scores, which are integers, and then prints them individually after the names of the players. This is the end of the script, as far as the player is concerned. Currently, the code will not permanently save the scores, but you can have Python write it to a file to keep if you wish.

24 Modules

The final part of the code allows for the script to be used in two ways. Firstly, we can execute it in the command line and it will work just fine. Secondly, we can import this into another Python script, perhaps if you wanted to add it as a game to a collection. This last piece of code will prevent our script being executed when imported by another module – it will only do so when being run directly.

Create a graphical interface for Python games

Bring everything together with a Python GUI and take the next step in programming your own software

Resources

Python 2: www.python.org/download

IDLE: www.python.org/idle

The three basic games we have made in Python so far have all run in the command line or via IDLE, a Python IDE. While this allowed us to show off different ways to use Python code, we haven't actually shown you how to present it yet. In this tutorial, we will take all three games and put them all into one neatly unified graphical interface.

To this end, we'll be making use of the small line of code we added at the bottom of each previous tutorial so we can import them as modules into our main graphical script. We'll also modify the existing code to add some graphical elements. To do all this we'll be using Tkinter, a default module available in Python that allows you to create windows and frames with fairly simple code.

All you need for this tutorial is an up-to-date copy of Python, from your distro's repository or the website, and the IDLE development environment. This will also work great on Raspberry Pi distros, such as Raspbian.

The start
Here we're doing some minor setup, including getting a new module that helps us create a simple graphical interface

The imports
We're importing the three games we created in past issues so we can call upon or use them

The window
Create a graphical window and give it a name so we can add some functions to it

The frame
Define the dimensions of the window and give a rough guide to placement of the objects within

The welcome
Print a message in the window and place it in a specific orientation. This works a little differently to print

The button
The focus of this month's tutorial is making Rock-Paper-Scissors work in a graphical interface, so we're calling a new function we're creating

The interface
Creating and formatting buttons to start the other two tutorial games in the command line or shell

The exit
Here we create a button that quits the window and ends the script. We've also placed it specifically at the bottom of the window

The loop
The mainloop allows the main window to continue to work and be updated without exiting the program unless specified

Main Interface Code Listing

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

#Linux User & Developer presents: Mega Microgrames Collection

from Tkinter import *

import rockpaperscissors
import hangman
import pokerdice

root = Tk()
root.title ("Linux User & Developer's Mega Microgames Collection")

mainframe = Frame(root, height = 200, width = 500)
mainframe.pack_propagate(0)
mainframe.pack(padx = 5, pady = 5)

intro = Label(mainframe, text = """Welcome to Linux User & Developers Mega
Microgames Collection.
Please select one of the following games to play:
""")
intro.pack(side = TOP)

rps_button = Button(mainframe, text = "Rock, Paper, Scissors", command =
rockpaperscissors.gui)
rps_button.pack()

hm_button = Button(mainframe, text = "Hangman", command = hangman.start)
hm_button.pack()

pd_button = Button(mainframe, text = "Poker Dice", command = pokerdice.start)
pd_button.pack()

exit_button = Button(mainframe, text = "Quit", command = root.destroy)
exit_button.pack(side = BOTTOM)

root.mainloop()
```

New imports
Import new modules that allow us to create the GUI part of Rock, Paper, Scissors, as well as removing the modules we no longer need

New interface
Our new main function allows us to call the majority of the game script when the rps_button is pressed. This contains the game components and the graphical components

New start
We've changed the start function so that it no longer goes to the score function after it's finished. We've also removed the score function, as we track that differently so it can be displayed properly

New game
We've changed the game function so that it now takes the input from our graphical interface. We use a new variable to do this that works with the GUI, otherwise it works roughly the same as before

New results
The result function remains largely unchanged, only now it sends the outcome message to a variable we use for the interface, and generally uses the new GUI's variables

New window
We create the game window with a slightly different method due to already having a 'mainloop' root window. We're also giving it a name so you can identify it properly

New variables
Our new variables are set up so they can interact with both the game code and the interface code properly. We've also made sure to have a default selection for the player so that the code runs properly

New frame
Determine the size and layout of the window for the game using a slightly different method than before. We've also allowed for elements to be anchored in certain positions around the window

New choice
Here we place radio buttons in a specific configuration in the window, giving the user the choice of three moves. This is then passed along to the variable and used by the game code

New move
Here we allow for the computer's move to be displayed under the 'Computer' label

New button
Pressing the Play button we've put here runs the game script, prints out the scores and finally a message based on the outcome

New ending
We've changed this so that the main script begins with gui now rather than the start function

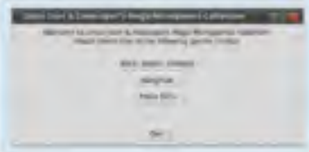
Modified RPS Code Listing

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

# Linux User & Developer presents: Rock, Paper, Scissors: The Video Game: The Module

from Tkinter import *
from ttk import *
import random

def gui():

    rock = 1
    paper = 2
    scissors = 3

    names = { rock: "Rock", paper: "Paper", scissors: "Scissors" }
    rules = { rock: scissors, paper: rock, scissors: paper }

    def start():
        while game():
            pass

    def game():
        player = player_choice.get()
        computer = random.randint(1, 3)
        computer_choice.set(names[computer])
        result(player, computer)

    def result(player, computer):
        new_score = 0
        if player == computer:
            result_set.set("Tie game.")
        else:
            if rules[player] == computer:
                result_set.set("Your victory has been assured.")
                new_score = player_score.get()
                new_score += 1
                player_score.set(new_score)
            else:
                result_set.set("The computer laughs as you realise you have been defeated.")
                new_score = computer_score.get()
                new_score += 1
                computer_score.set(new_score)

    rps_window = Toplevel()
    rps_window.title ("Rock, Paper, Scissors")

    player_choice = IntVar()
    computer_choice = StringVar()
    result_set = StringVar()
    player_choice.set(1)
    player_score = IntVar()
    computer_score = IntVar()

    rps_frame = Frame(rps_window, padding = '3 3 12 12', width = 300)
    rps_frame.grid(column=0, row = 0, sticky=(N,W,E,S))
    rps_frame.columnconfigure(0, weight=1)
    rps_frame.rowconfigure(0,weight=1)

    Label(rps_frame, text='Player').grid(column=1, row = 1, sticky = W)
    Radiobutton(rps_frame, text ='Rock', variable = player_choice, value = 1).grid(column=1,
    row=2, sticky=W)
    Radiobutton(rps_frame, text ='Paper', variable = player_choice, value = 2).grid(column=1,
    row=3, sticky=W)
    Radiobutton(rps_frame, text ='Scissors', variable = player_choice, value =
    3).grid(column=1, row=4, sticky=W)

    Label(rps_frame, text='Computer').grid(column=3, row = 1, sticky = W)
    Label(rps_frame, textvariable = computer_choice).grid(column=3, row=3, sticky = W)

    Button(rps_frame, text="Play", command = start).grid(column = 2, row = 2)

    Label(rps_frame, text = "Score").grid(column = 1, row = 5, sticky = W)
    Label(rps_frame, textvariable = player_score).grid(column = 1, row = 6, sticky = W)

    Label(rps_frame, text = "Score").grid(column = 3, row = 5, sticky = W)
    Label(rps_frame, textvariable = computer_score).grid(column = 3, row = 6, sticky = W)

    Label(rps_frame, textvariable = result_set).grid(column = 2, row = 7)

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

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

#Linux User & Developer presents: Mega Micrograms Collection

02 from Tkinter import *

import rockpaperscissors
03 import hangman
import pokerdice

root = Tk()
04 root.title ("Linux User & Developer's Mega Microgames Collection")

mainframe = Frame(root, height = 200, width = 500)
05 mainframe.pack_propagate(0)
mainframe.pack(padx = 5, pady = 5)

intro = Label(mainframe, text = """Welcome to Linux User & Developers Mega Microgames Collection.
Please select one of the following games to play:
06 """)
intro.pack(side = TOP)

07 rps_button = Button(mainframe, text = "Rock, Paper, Scissors", command = rockpaperscissors.gui)
rps_button.pack()

08 hm_button = Button(mainframe, text = "Hangman", command = hangman.start)
hm_button.pack()

pd_button = Button(mainframe, text = "Poker Dice", command = pokerdice.start)
pd_button.pack()

exit_button = Button(mainframe, text = "Quit", command = root.destroy)
09 exit_button.pack(side = BOTTOM)

root.mainloop()
```

MAIN WINDOW

The main interface window that this code creates is fairly basic, but contains the functions we require. The window exit button will do the same job as the Quit button, and the Hangman and Poker Dice buttons run the old scripts in the Python shell.



01 First line

We use this line to enter the path to the Python interpreter. This lets us run the program inside a terminal or otherwise outside of a Python-specific IDE like IDLE. Note that we're also using Python 2 for this particular script.

02 Import graphics

Tkinter is the graphical interface we're using and while it's a standard Python function, you'll need to import the module so you can use it. We've used the 'from [module] import *' method so that we can use the functions from it without having to add Tkinter at the beginning.

03 Import games

We're importing the modules for the three games. We added the line at the bottom of each script so we can do this. To make sure to differentiate the functions in each game, we will have to specify [module].[function] so there are no errors in the code.

04 Root window

Using the Tk() function creates the window we're going to be placing everything into. We've decided to call it root for now; however, you can call it anything you like, as long as you're consistent with it. We've also named it using the title command from Tkinter and a string of text.

05 Main frame

The first line has us set the variable mainframe as a Frame in the interface. We've attached it to root, the main window, and given it a minimum height and width in pixels. We use pack_propogate to create the window, and then make sure it's the size that we've defined. We've then used pack to pad the borders, allowing the contents of the window to not touch the sides of it.

06 Introductions

We create the intro variable as a label that lives in the main frame. We give it text to introduce the interface, using the triple quote marks to have it go across multiple lines and format better. We then use pack to display it, and tell Tkinter to put it at the top of the interface.

07 Rock, Paper, Scissors

We create a button for the Rock, Paper, Scissors game using the Button function. We attach to it the main frame, give it a label using

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

10 # Linux User & Developer presents: Rock, Paper, Scissors: The Video Game: The Module

   from Tkinter import *
   from ttk import *
   import random

11 def gui():

       rock = 1
       paper = 2
12     scissors = 3

       names = { rock: "Rock", paper: "Paper", scissors: "Scissors" }
       rules = { rock: scissors, paper: rock, scissors: paper }

       def start():
13         while game():
               pass

       def game():
           player = player_choice.get()
14         computer = random.randint(1, 3)
           computer_choice.set(names[computer])
           result(player, computer)
```

PYTHON SHELL

Our other code will run in the shell or via a command line in the same way as before when the buttons are pressed.

text that appears on the button, and then have it run a command. In this case, we use the modified rockpapershotgun.py code that has a gui function, hence rockpapershotgun.py. We then use pack to place it in the window

08 Other games

For the other two games, the code is mostly the same; however, we call upon the start function in both of them. In the final interface, this will cause the games to run in the shell or command line as they've been running before.

09 Break the loop

The exit button works similarly to the other buttons we've created, but instead it uses the command root.destroy. This ends the loop that we've created with root.mainloop(), which allows the interface code to continue looping, allowing us to continually use it. We place the exit button at the bottom of the window with 'side = BOTTOM'.

10 Game code

Nothing much has changed in the start of this code, other than a few import changes. The code for running it in the command line is still there, and with a few modifications the code will run independently of the main interface. We've removed the time module, as we no longer need it, and imported not only the Tkinter module, but the ttk module. The ttk module allows us to arrange the GUI in a grid, which will be slightly easier to use and understand.

11 Game interface

One of the biggest changes we're making to this script is having it all contained in one function, 'def gui'. The interface code needs to be put into a function, otherwise it will be run during import. While we've chosen to put the entirety of the code in a function, you can also try just having the graphical interface code in one. All our variables are kept in here so that they still work properly.

12 Game variables

The variables are staying the same so that we can do the same comparisons we made in the original code. We've put them into the function itself so that they don't affect the other imported code into the main interface – and so that when calling just this function, we don't need to use global to bring them in.

13 Start function

We've removed the part that calls the score function from the start function, as we have the interface handle the scoring now. It still calls upon the game function, though, putting it into a loop so it can be used continuously. This function is called by the interface to begin the game by setting a computer move and then comparing it to the player's choice.

14 Game function

The game function has had a few modifications to make sure it works with the interface. First of all, the player variable is retried using get() on the special variable we've created to contain the player choice. We do a similar thing for the computer, using 'set' to change the variable in our interface-friendly computer_choice value. We still use the name variable to set the text that goes into computer_choice. This then passes the player and computer variables along in the same way we did before.

```
15 def result(player, computer):
       new_score = 0
16     if player == computer:
           result_set.set("Tie game.")
       else:
           if rules[player] == computer:
17             result_set.set("Your victory has been assured.")
               new_score = player_score.get()
               new_score += 1
               player_score.set(new_score)
           else:
18             result_set.set("The computer laughs as you realise you have been defeated.")
               new_score = computer_score.get()
               new_score += 1
               computer_score.set(new_score)

19 rps_window = Toplevel()
   rps_window.title ("Rock, Paper, Scissors")

   player_choice = IntVar()
   computer_choice = StringVar()
20 result_set = StringVar()
   player_choice.set(1)
   player_score = IntVar()
   computer_score = IntVar()
```

GAME WINDOW

In its default state, the game window will have rock selected and no message will be displayed. Once the player makes a move, the message will be displayed at the bottom and the computer's move will be printed. There's no quit button on this menu, but clicking the window exit will bring you back to the main interface.

15 Result function

The result function still takes the same two variables as before, which we set in the game function. While technically we can use the variables set up for the interface, these are not pure integers and can cause an error if not handled correctly. With that in mind, we've created an empty new_score variable that we can use to effectively clean the interface value before adding it back into it.

16 Tie

The logic for determining the result is the same as before. We first do the easy check – whether or not the numeric value for the player and computer variable is the same. What changes this time is that, instead of printing the text, we send the "Tie game" message to our result variable using the set function from Tkinter.

17 Win

The if statement continues by seeing if the player has won. Like before, we use the rules we set to make the comparison for the code to make. We set the result_set like we did in the tie game, with a different message to the user. Finally, we set the new_score variable to be the current player score, using the get function to obtain it, plus one to the score, and then use set again to put it back into the player_score variable. We can't use += with the player_score variable, as it is not a standard variable.

18 Lose

This part of the overall if statement works in the same way as before, by assuming that if it isn't a tie or a win, it's a loss. Like the new version of the win code, it then uses set to change the message that will be displayed to the player, and calls upon and changes the computer score by putting it through the new_score variable.

19 New window

As the original window is part of the mainloop, we cannot have the window be created using Tk() like in the main interface code. As this window is coming off it, though, we instead create it using Toplevel(). This allows the window to run separately and on top of the main window. We've also given it a name, which will not change the main window's name in the process.

20 Interface variables

Here is the reason we had to call and change the variables in a different manner. For Tkinter, we need to let the interface know whether or not a variable is an integer or a text value. IntVar and StringVar allow for these respectively. We've also set the player_choice variable to be one, which we have already set as the choice for rock. This means there will at least be a default choice when the game is started, and it won't cause an error.

21 Game frame

We've created the frame for our interface items slightly differently. Instead of using the pack command in the main interface, we're using grid to make sure they're orientated in such a way that makes sense for the user. Padding does just that, setting up values to make sure the items in the frame don't touch the edge of the window. Using the .grid command, we then create this frame. The row and column variables allow for rows and columns to be included in the structure of

```
    rps_frame = Frame(rps_window, padding = '3 3 12 12', width = 300)
    rps_frame.grid(column=0, row = 0, sticky=(N,W,E,S))
21  rps_frame.columnconfigure(0, weight=1)
    rps_frame.rowconfigure(0,weight=1)

    Label(rps_frame, text='Player').grid(column=1, row = 1, sticky = W)
    Radiobutton(rps_frame, text ='Rock', variable = player_choice, value = 1).grid(column=1, row=2,
    sticky=W)
22  Radiobutton(rps_frame, text ='Paper', variable = player_choice, value = 2).grid(column=1, row=3,
    sticky=W)
    Radiobutton(rps_frame, text ='Scissors', variable = player_choice, value = 3).grid(column=1,
    row=4, sticky=W)

23  Label(rps_frame, text='Computer').grid(column=3, row = 1, sticky = W)
    Label(rps_frame, textvariable = computer_choice).grid(column=3, row=3, sticky = W)

24  Button(rps_frame, text="Play", command = start).grid(column = 2, row = 2)

    Label(rps_frame, text = "Score").grid(column = 1, row = 5, sticky = W)
    Label(rps_frame, textvariable = player_score).grid(column = 1, row = 6, sticky = W)

25  Label(rps_frame, text = "Score").grid(column = 3, row = 5, sticky = W)
    Label(rps_frame, textvariable = computer_score).grid(column = 3, row = 6, sticky = W)

    Label(rps_frame, textvariable = result_set).grid(column = 2, row = 7)

23 if __name__ == '__main__':
       gui()
```

the window, and the sticky allows us to justify items with specific directions – in this case top, left, right and bottom justification. Finally, we then make sure each column and row is treated equally by giving them the same weighting, and starting from zero.

22 Player's choice

We create a label for the player's move and assign it to a grid location, on the first row, on the first column. We also justify it to the left using 'sticky = W'. We then add the radio buttons for the player's move, each on the same column but the following row down. We give each choice a name, then assign it to the player_choice variable. We then make each choice have a numerical value that corresponds to the moves we've determined in the first set of rules.

23 Computer's move

We display the computer move here. First of all, we label what this is and then create a second label to display the actual move. We do this by adding the textvariable option to Label, and using the computer_choice variable we updated earlier in the game function. This merely prints the text from the names list and justifies this to the left.

24 Press Play

The running of the code all hinges on the Play button. It's very simple: we put it in the row between the Player and Computer move as part of our three-column system; and it runs the start function using the command option. Due to the loop of the interface, we can keep pressing this without needing to be asked to play again. Simply exiting the window will go back to the main interface window as well, meaning we do not need a specific quit button.

25 Running score

We have two sets of scores to display – one for the player and the other for the computer. We label these the same way we've done with labelling the Player and Computer move, having them on a lower row but still in the relevant columns. Below that, we use the textvariable option again to get the numerical score we assigned to the separate score variable. Finally, we create another label to display the message for the game's outcome

26 End game

The final part of the code allows for the script to be used by the main window, and also allows for it to run on its own when used in the command line or shell. You'll need to perform some modifications to make it run on its own, such as making it the mainloop and not a Toplevel window. However, it will run just fine from both without the need to be launched from the main interface.

Bring graphics to simple Python games

Complete your trio of games with a graphical interface for the hangman and poker dice code

Resources

Python 2: www.python.org/download

IDLE: www.python.org/idle

We have now created a simple selector for the trio of Python games we made previously. This interface was able to launch a GUI for our rock, paper, scissors game, and run the other two in the terminal. Now, we're going to convert the hangman and poker dice codes to work in a similar way to rock, paper, scissors.

The trick with hangman comes in allowing for a different type of input, text, and the ability to have multiple rounds of the game. Tkinter allows for text entry, and we rely a lot less on 'while' loops to play the game in its entirety. Poker Dice needs to keep the dice analysis code, and the option to change specific dice using checkboxes.

We'll be modifying a large amount of the original code to fit in with the new graphical scheme. This mainly involves cutting specific parts and having the Tkinter-specific code handle these itself. The code listings on these pages include the modified code – we'll discuss the graphical part on the following pages.

1 Imported
Here we're doing a minor setup, including getting the Tkinter module that helps us create a simple graphical interface

2 Words
We're keeping our variables that determine the word to guess here so it can be easily accessed anywhere in the code

3 Function
Like last time, we're putting the majority of our original code into a new function, gui

4 Analysis
We select the word and analyse it before continuing on with the rest of the code

5 Graphics
The hangedman function is largely unchanged, albeit with new code to display our ASCII graphics on the interface

6 Guesses
We check the number of mistakes made, and call the guess_letter function to check the letter entered

Hangman Code Listing

01
```
from Tkinter import *
from ttk import *
from random import *
```
02
```
word = 0
word_length = 0
clue = 0
```
03
```
def gui():
    global word, word_length, clue
    dictionary = ["gnu","kernel","linux","magei
a","penguin","ubuntu"]
```
04
```
    word = choice(dictionary)
    word_length = len(word)
    clue = word_length * ["_"]
    tries = 6
```
05
```
    def hangedman(hangman):
        graphic = [
        """
            +--------+
            |        |
            |        O
            |       -|-
            |       / \
            |
        ===============
        """]
        graphic_set = graphic[hangman]
        hm_graphic.set(graphic_set)
```
06
```
    def game():
        letters_wrong = incorrect_guesses.get()
        letter=guess_letter()
        first_index=word.find(letter)
        if   first_index == -1:
            letters_wrong +=1
            incorrect_guesses.set(letters_
            wrong)
        else:
            for i in range(word_length):
                if letter == word[i]:
                    clue[i] = letter
        hangedman(letters_wrong)
        clue_set = " ".join(clue)
        word_output.set(clue_set)
        if letters_wrong == tries:
            result_text = "Game Over. The word
was " + word
            result_set.set(result_text)
            new_score = computer_score.get()
            new_score += 1
            computer_score.set(new_score)
        if "".join(clue) == word:
            result_text = "You Win! The word
was " + word
            result_set.set(result_text)
            new_score = player_score.get()
            new_score += 1
            player_score.set(new_score)

    def guess_letter():
        letter = letter_guess.get()
        letter.strip()
        letter.lower()
        return letter

    def reset_game():
        global word, word_length, clue
        incorrect_guesses.set(0)
        hangedman(0)
        result_set.set("")
        letter_guess.set("")
        word = choice(dictionary)
        word_length = len(word)
        clue = word_length * ["_"]
        new_clue = " ".join(clue)
        word_output.set(new_clue)

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

1 More imports
We've added the new imported modules we need to make Tkinter work and keep the rest the same

2 Dice list
The list that holds the dice is kept outside the main function so that it can be accessed everywhere

3 Rolls
Same goes for the roll function. It doesn't specifically need to be inside the gui function anyway

4 Decisions
The checkboxes in the graphical code we're going to create later will give us numbers we can analyse for the code. We retrieve these numbers and check them to find out which dice the user wishes to re-roll

5 Hands
Finally, our hand analysis function is the last part of the original code that is kept outside the gui function. Both this and the above function pass the necessary details back up the chain to then be added into the new graphical elements of the new interface

6 No dice
If no dice have been selected to re-roll, the hand output is changed to show a final message

7 Re-roll
This part is almost the same as before – a new set of dice are rolled and then inserted into the list of dice like before, then re-sorted to make the hand analysis easier

8 More functions
The new gui function is the main change to the Poker Dice code, and as before includes the Tkinter elements and other parts of the original code

9 Game start
A simple function that we can use to activate the re-rolls of the dice

10 New hand
The new dice are named, analysed, and everything is then set for the gui to display the final outcome

11 Reset
Like with the hangman code, we have a function to reset all the variables, allowing you to start the game again

Poker Dice Code Listing

01
```
from Tkinter import *
from ttk import *
import random
from itertools import groupby
```
02
```
dice = 0
```
03
```
def roll(roll_number):
        numbers = range(1,7)
        dice = range(roll_number)
        iterations = 0
        while iterations < roll_number:
            iterations = iterations + 1
            dice[iterations-1] = random.
choice(numbers)
        return dice
```
04
```
def hand(dice):
    dice_hand = [len(list(group)) for key,
group in groupby(dice)]
    dice_hand.sort(reverse=True)
    straight1 = [1,2,3,4,5]
    straight2 = [2,3,4,5,6]
    if dice == straight1 or dice ==
straight2:
        return "a straight!"
    elif dice_hand[0] == 5:
        return "five of a kind!"
    elif dice_hand[0] == 4:
        return "four of a kind!"
    elif dice_hand[0] == 3:
        if dice_hand[1] == 2:
            return "a full house!"
        else:
            return "three of a kind."
    elif dice_hand[0] == 2:
        if dice_hand[1] == 2:
            return "two pair."
        else:
            return "one pair."
    else:
        return "a high card."
```
05
```
def gui():
    global dice
    dice = roll(5)
    dice.sort()
    nine = 1
    ten = 2
    jack = 3
    queen = 4
    king = 5
    ace = 6
    names = { nine: "9", ten: "10", jack:
"J", queen: "Q", king: "K", ace: "A" }
    result = "You have " + hand(dice)
```
06
```
    def game():
        throws()

    def throws():
```
07
```
        global dice
        dice1_check = dice1.get()
        dice2_check = dice2.get()
        dice3_check = dice3.get()
        dice4_check = dice4.get()
        dice5_check = dice5.get()
        dice_rerolls = [dice1_check,
dice2_check, dice3_check, dice4_check,
dice5_check]
```
08
```
        for i in range(len(dice_rerolls)):
            if 0 in dice_rerolls:
                dice_rerolls.remove(0)
        if len(dice_rerolls) == 0:
            result = "You finish with " +
hand(dice)
            hand_output.set(result)
```
09
```
        else:
            roll_number = len(dice_rerolls)
            number_rerolls = roll(roll_num-
ber)
            dice_changes = range(len(dice_
rerolls))
            iterations = 0
            while iterations < roll_number:
                iterations = iterations + 1
                dice_changes[iterations-1]
= number_rerolls[iterations-1]
            iterations = 0
            while iterations < roll_number:
                iterations = iterations + 1
                replacement = number_
rerolls[iterations-1]
                dice[dice_
changes[iterations-1]] = replacement
            dice.sort()
            new_dice_list = [0,0,0,0,0]
            for i in range(len(dice)):
                new_dice_list[i] =
names[dice[i]]
            final_dice = " ".join(new_dice_
list)
```
10
```
            dice_output.set(final_dice)
            final_result = "You finish with
" + hand(dice)
            hand_output.set(final_result)
```
11
```
    def reset_game():
        global dice
        dice = roll(5)
        dice.sort()
        for i in range(len(dice)):
            empty_dice[i] =  names[dice[i]]
        first_dice = " ".join(empty_dice)
        dice_output.set(first_dice)
        result = "You have " + hand(dice)
        hand_output.set(result)

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

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

01 from Tkinter import *
   from ttk import *
   from random import *
   word = 0
02 word_length = 0
   clue = 0

03 def gui():
       global word, word_length, clue
       dictionary = [“gnu”,”kernel”,”linux”,”mageia”,”penguin”,”ubuntu”]
       word = choice(dictionary)
04     word_length = len(word)
       clue = word_length * [“_”]
       tries = 6

       def hangedman(hangman):
           graphic = [
           “””
               +-------+
               |       |
05             |       O
               |      -|-
               |      / \
               |
           ===============
           “””]
           graphic_set = graphic[hangman]
           hm_graphic.set(graphic_set)

       def game():
06         letters_wrong = incorrect_guesses.get()
           letter=guess_letter()
           first_index=word.find(letter)
           if  first_index == -1:
               letters_wrong +=1
07             incorrect_guesses.set(letters_wrong)
           else:
               for i in range(word_length):
                   if letter == word[i]:
                       clue[i] = letter
           hangedman(letters_wrong)
08         clue_set = “ “.join(clue)
           word_output.set(clue_set)
           if letters_wrong == tries:
               result_text = “Game Over. The word was “ + word
               result_set.set(result_text)
               new_score = computer_score.get()
               new_score += 1
               computer_score.set(new_score)
09         if “”.join(clue) == word:
               result_text = “You Win! The word was “ + word
               result_set.set(result_text)
               new_score = player_score.get()
               new_score += 1
               player_score.set(new_score)
```

YOU LOSE

When you’ve run out of guesses, the game stops. From here, you can also reset the game to play again if you wish.

01 First lines

As usual, we start off each program with the code that lets us run it in the command line, followed by importing the necessary modules: **random**, to determine the word to use; **Tkinter**, for the majority of the graphical code; and **ttk**, for the grid code we’ll be using to align the different elements.

02 Global variables

We have kept these three variables outside of the **gui** function so they can be accessed at all points in the code. Python 2 does not allow you to call upon global variables when you’re in a nested function, whereas in Python 3 this could have gone into the gui function.

03 Graphical function

We’re putting all the working code into the gui function so it can be activated from the main interface. This means we can import the Hangman code into the interface without the game window popping up, and only run it when we activate the gui function from here.

04 Random word

We bring in the three variables with **global** so we can modify them throughout the code, and then set the word. As before, a random item from the list of words is selected with choice, the length is ascertained, and the clue to display is set.

05 The hanged man

The main difference this time for the Hangman graphics is that instead of printing these out, we’re going to display them in the interface. When the function is called and the graphic selected, it’s placed in the variable we’ve set up in the interface code that we’re using to display the result.

06 Games begin

All the analysis of the letter we’ve entered is done in this function. To that end, we start by obtaining the incorrect guesses so far from the variable we’ve set up so the interface can access it if we want it to. The letter from the entry field in the interface is then obtained and cleaned up so it can be used with the rest of the code.

07 Check the letter

This section of the code is again largely unchanged – the letter is taken and compared to the word with **find** to see if it matches with one of the letters. The if statement then adds one to the incorrect guess variable, or updates the clue variable to add the letter in the right spot.

08 Update interface

These three lines set the graphic for this round, join the current clue together as a string, and then set it on the variable for the interface to read.

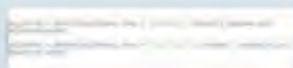
09 Update scores

Exactly as before, we check to see if the player has won or lost yet. In the event of either, a message is displayed to signify this, and the wins and losses score is updated using **set**.

```
    def guess_letter():
        letter = letter_guess.get()
10      letter.strip()
        letter.lower()
        return letter

    def reset_game():
        global word, word_length, clue
        incorrect_guesses.set(0)
        hangedman(0)
        result_set.set("")
        letter_guess.set("")
        word = choice(dictionary)
        word_length = len(word)
        clue = word_length * ["_"]
        new_clue = " ".join(clue)
        word_output.set(new_clue)

11  hm_window = Toplevel()
    hm_window.title ("Hangman")
    incorrect_guesses = IntVar()
    incorrect_guesses.set(0)
    player_score = IntVar()
12  computer_score = IntVar()
    result_set = StringVar()
    letter_guess = StringVar()
    word_output = StringVar()
    hm_graphic = StringVar()

    hm_frame = Frame(hm_window, padding = '3 3 12 12', width = 300)
13  hm_frame.grid(column=0, row = 0, sticky=(N,W,E,S))
    hm_frame.columnconfigure(0, weight=1)
    hm_frame.rowconfigure(0,weight=1)

    Label(hm_frame, textvariable = hm_graphic).grid(column=2, row = 1)
14  Label(hm_frame, text='Word').grid(column=2, row = 2)
    Label(hm_frame, textvariable = word_output).grid(column=2, row = 3)

    Label(hm_frame, text='Enter a letter').grid(column=2, row = 4)
15  hm_entry = Entry(hm_frame, exportselection = 0, textvariable = letter_guess).grid(column = 2, row = 5)
    hm_entry_button = Button(hm_frame, text = "Guess", command = game).grid(column = 2, row = 6)

    Label(hm_frame, text = "Wins").grid(column = 1, row = 7, sticky = W)
    Label(hm_frame, textvariable = player_score).grid(column = 1, row = 8, sticky = W)
    Label(hm_frame, text = "Losses").grid(column = 3, row = 7, sticky = W)
16  Label(hm_frame, textvariable = computer_score).grid(column = 3, row = 8, sticky = W)
    Label(hm_frame, textvariable = result_set).grid(column = 2, row = 9)
    replay_button = Button(hm_frame, text = "Reset", command = reset_game).grid(column = 2, row = 10)

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

ORIGINAL INTERFACE

You'll also need the interface code from last issue, which already works with the modified Rock, Paper, Scissors code. The way it was left off means it won't work with the new code, so you'll have to change the command in each button from **[game].start** to **[game].gui**.



THE HANGMAN GUI

Press the updated Hangman button to launch a new window. Here we have the initial graphic, word clue and entry for the player to interact with. The scores are set to zero, and no result message is displayed as no games have been played yet.

10 Sanitise input

The **guess_letter** function purely gets the letter from the player input variable, strips it of any formatting, makes it lower case, and then returns it back to the game function. This is so the letter can be used properly.

11 New window

We use the **Toplevel** command from Tkinter like last month to separate the loops of the main interface and game window. We then use **title** to call it Hangman.

12 Interface variables

Tkinter only works with specific variables – we've created all the ones we need or can use here. IntVars take integers, while StringVars take strings. We've used **get** and **set** throughout the rest of the code with these to get and set values.

> "The frame is set up as before"

13 Framed window

The frame is set up the same way as last time. We pad the frame from the edge of the window, set a grid, give it sticky points at compass points, and allow for setting objects with specific row and column points.

14 Clue to Hangman

These labels are fairly straightforward – we're either giving them fixed text, or telling them to use a specific **textvariable** so they can be updated as we play the game.

15 Text entry

Entry here sets a text box we will add the letters to. The **exportselection** option makes it so selecting the letter won't immediately copy it to the clipboard, and the **textvariable** selection is where the code stores the letter added. The button activates the **game** function, analysing the letter the player entered.

16 Results and reset

The rest of the code is similar to what we've done already: labels to display fixed text and the scores/result text that change. The button that activates the **reset** function is also put at the bottom here. The final two lines allow us to import the module into the interface code.

17 Start over
The usual array of command-line compatibility and module importing here. The **groupby** function is specifically imported here for dice analysis.

18 Outside dice
For Poker Dice, there's only one variable to show at any one time, the dice. Again, due to the nested functions, and because we're using Python 2, we need to call it with **global** from here to make sure the game can be reset properly.

19 Dice rolls
The **roll** function has been removed from the **gui** function so as not to create any code errors with some of its variables. It can be easily called within the nested functions. It hasn't changed at all from the original code.

20 Hand of dice
Like **roll**, nothing has changed for the **hand** function. It's simply now placed outside the **gui** function for the exact same reasons. It also means that you can easily import this function into another script if you wish.

21 GUI start
As we've mentioned last month and in the Hangman code, we put all the GUI code into a function so that we can call on it when we want to. In this case, pressing the Poker Dice button on the main interface activates **pokerdice.gui**, which is this function.

22 First roll
As the window opens, we immediately make the first roll. This is then sorted, each number is attributed to a card, and then the result is created to be displayed in the main window. This is similar to how it worked before, but instead it's now entered into the StringVars for the interface towards the end of the script

23 Start game
When we activate the button that starts **game**, it immediately sends us to the rest of the code. This would also work if you had the button go to the **throws** function instead; however, you can add other functions to this part if you wish.

24 Dice selection
The first thing we do is find out what checkboxes have been ticked by the player. We then put these in a list so we can change out the correct dice numbers. We've also brought in **dice** so we can check against that what the current dice rolls are.

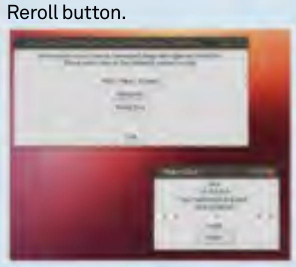
```
#!/usr/bin/env python2

from Tkinter import *
from ttk import *
import random
from itertools import groupby
dice = 0

def roll(roll_number):
        numbers = range(1,7)
        dice = range(roll_number)
        iterations = 0
        while iterations < roll_number:
            iterations = iterations + 1
            dice[iterations-1] = random.choice(numbers)
        return dice

def hand(dice):
    dice_hand = [len(list(group)) for key, group in groupby(dice)]
    dice_hand.sort(reverse=True)
    straight1 = [1,2,3,4,5]
    straight2 = [2,3,4,5,6]
    if dice == straight1 or dice == straight2:
        return “a straight!”
    elif dice_hand[0] == 5:
        return “five of a kind!”
    elif dice_hand[0] == 4:
        return “four of a kind!”
    elif dice_hand[0] == 3:
        if dice_hand[1] == 2:
            return “a full house!”
        else:
            return “three of a kind.”
    elif dice_hand[0] == 2:
        if dice_hand[1] == 2:
            return “two pair.”
        else:
            return “one pair.”
    else:
        return “a high card.”

def gui():
    global dice
    dice = roll(5)
    dice.sort()
    nine = 1
    ten = 2
    jack = 3
    queen = 4
    king = 5
    ace = 6
    names = { nine: “9”, ten: “10”, jack: “J”, queen: “Q”, king: “K”,
ace: “A” }
    result = “You have “ + hand(dice)

    def game():
        throws()

    def throws():
        global dice
        dice1_check = dice1.get()
        dice2_check = dice2.get()
        dice3_check = dice3.get()
        dice4_check = dice4.get()
        dice5_check = dice5.get()
        dice_rerolls = [dice1_check, dice2_check, dice3_check, dice4_
check, dice5_check]
```

EXTRA GAME FUNCTIONS

We mentioned that the **game** function doesn't necessarily need to be used right now. You can either clean up the code and remove it, or add extra functions, such as being able to choose a random new selection of dice, or making it two-player. Experiment with what you want to do!

THE POKER DICE GUI

Two things are being printed out on the initial window. The first set of dice, ordered in the way we did last time, and the current hand. The checkboxes activate a specific number that is used when re-rolling dice with the Reroll button.



```
      for i in range(len(dice_rerolls)):
25        if 0 in dice_rerolls:
              dice_rerolls.remove(0)
      if len(dice_rerolls) == 0:
26        result = “You finish with “ + hand(dice)
          hand_output.set(result)
      else:
          roll_number = len(dice_rerolls)
          number_rerolls = roll(roll_number)
          dice_changes = range(len(dice_rerolls))
27        iterations = 0
          while iterations < roll_number:
              iterations = iterations + 1
              dice_changes[iterations-1] = number_rerolls[iterations-1]
          iterations = 0
          while iterations < roll_number:
              iterations = iterations + 1
              replacement = number_rerolls[iterations-1]
              dice[dice_changes[iterations-1]] = replacement
          dice.sort()
          new_dice_list = [0,0,0,0,0]
          for i in range(len(dice)):
              new_dice_list[i] =  names[dice[i]]
          final_dice = “ “.join(new_dice_list)
          dice_output.set(final_dice)
28        final_result = “You finish with “ + hand(dice)
          hand_output.set(final_result)

  def reset_game():
      global dice
      dice = roll(5)
      dice.sort()
      for i in range(len(dice)):
          empty_dice[i] =  names[dice[i]]
      first_dice = “ “.join(empty_dice)
      dice_output.set(first_dice)
      result = “You have “ + hand(dice)
      hand_output.set(result)

  pd_window = Toplevel()
  pd_window.title (“Poker Dice”)
  dice_output = StringVar()
  empty_dice = [0,0,0,0,0]
  for i in range(len(dice)):
      empty_dice[i] =  names[dice[i]]
  first_dice = “ “.join(empty_dice)
  dice_output.set(first_dice)
29 hand_output = StringVar()
  hand_output.set(result)
  dice1 = IntVar()
  dice2 = IntVar()
  dice3 = IntVar()
  dice4 = IntVar()
  dice5 = IntVar()
  result_set = StringVar()
  player_score = IntVar()
  computer_score = IntVar()

  pd_frame = Frame(pd_window, padding = ‘3 3 12 12’, width = 300)
  pd_frame.grid(column=0, row = 0, sticky=(N,W,E,S))
  pd_frame.columnconfigure(0, weight=1)
  pd_frame.rowconfigure(0,weight=1)
  Label(pd_frame, text='Dice').grid(column=3, row = 1)
  Label(pd_frame, textvariable = dice_output).grid(column=3, row = 2)
  Label(pd_frame, textvariable = hand_output).grid(column=3, row = 3)

  Label(pd_frame, text='Dice to Reroll?').grid(column=3, row = 4)
  reroll1 = Checkbutton(pd_frame, text = “1”, variable = dice1, onvalue = 1, offvalue
= 0).grid(column=1, row = 5)
  reroll2 = Checkbutton(pd_frame, text = “2”, variable = dice2, onvalue = 2, offvalue
= 0).grid(column=2, row = 5)
  reroll3 = Checkbutton(pd_frame, text = “3”, variable = dice3, onvalue = 3, offvalue
30 = 0).grid(column=3, row = 5)
  reroll4 = Checkbutton(pd_frame, text = “4”, variable = dice4, onvalue = 4, offvalue
= 0).grid(column=4, row = 5)
  reroll5 = Checkbutton(pd_frame, text = “5”, variable = dice5, onvalue = 5, offvalue
= 0).grid(column=5, row = 5)
  pd_reroll_button = Button(pd_frame, text = “Reroll”, command = game).grid(column =
3, row = 6)
  replay_button = Button(pd_frame, text = “Reset”, command = reset_game).grid(column
= 3, row = 7)

if __name__ == ‘__main__’:
    gui()
```

ONE WINDOW

The way we've made these Tkinter interfaces is to have the games launch in a separate window. You can have them all running in one window, though, by replacing the labels and buttons of the original interface by putting them as different functions or classes. Make sure to add a quit button to the games that lets you go back to the main page.

“The check buttons are new”

25 Dice to re-roll

If a checkbox isn't selected, we have it set to give a zero value. We want to remove these from the list so that the correct dice are changed, so we use the **for** loop to check each part of the list, and then use the **remove** function when the element does equal zero.

26 Early finish

If no dice have been selected to re-roll, the list will contain all 0s, which will then be removed. The length of this list will then also be zero, meaning we can use that to end the game if the player hits Reroll without selecting any dice.

27 New dice

This **else** function works roughly the same as before. We start by getting the necessary information for how many dice to roll, and a list to put the re-rolls. We then roll as many new dice as we need with the first **while** loop

28 Game over

We use the same kind of **while** loop to replace the new numbers into the original list, much like last time. Then the dice are re-sorted, analysed, joined as a string and then set into the interface's variable. The final hand message is also create and set.

29 Graphical variables

As we're rolling the dice as soon as we launch the game, but the interface code doesn't start until the end, you can see that after creating the necessary variables, we also then set them. Of note, the dice have to be made into a string separately with the **for** loop before adding to the variable.

30 Check buttons

The main new addition to this code is the check buttons with **Checkbutton**. You can set an on and off value, with default off being 0. We've made it so that the check buttons return the same number as the dice they're changing, which we explained how we used earlier in the code. The **variable** option sets whatever the outcome is to the specific Tkinter variable.

Embed Python in C

Here, we will learn how to use Python code within your usual C program to get the best of both worlds

Back in issue 155, we looked at how to call C functions from within a Python program to get more speed. But, there are times, within a C program, when you may want to execute some piece of Python code. Maybe you want to be able to run user code within your program, for example. This means you can enable users to use plug-ins to extend your program's functionality. The way we can do this is by embedding Python within the C program.

We will look at how to embed, how to run your Python code, and how to interact with the Python interpreter you've set up. This is functionality built into Python itself, so you don't need to install anything extra on your Raspberry Pi, aside from the development package for Python and GCC. You will need to install them with the command:

```
sudo apt-get install python-dev gcc
```

You should now have all of the tools you need to compile your code.
The first step is to start the interpreter. To access the functions you need, you will have to add the following line to the head of your C source code file:

```
#include <Python.h>
```

You can now start to embed Python. The first function you need is void Py_Initialize(). The only other functions that can be called, before you initialise the interpreter, are Py_SetProgramName(), Py_SetPythonHome(), PyEval_InitThreads(), PyEval_ReleaseLock() and PyEval_AcquireLock(). Once this function finishes, you can start to interact with this interpreter. This starts up the interpreter, and loads the core modules __builtin__, __main__ and sys. But what about other modules? You can set the search path, where the interpreter will look to find modules, by using the function void Py_SetPythonHome(char *home). If you need the information, you can find the current module path with the function char* Py_GetPythonHome().

It does not set sys.argv, however. You need to use the function void PySys_SetArgvEx(int argc, char **argv, int updatepath). This way, you can access any command line arguments that your Python code needs. You can check to see whether the interpreter is properly initialised by using the function int Py_IsInitialized(). It returns an integer for either true (nonzero) or false (zero). The simplest way to use your new interpreter is to use the function int PyRun_SimpleString(const char *command). This function takes a string that contains some arbitrary bit of code. If you have multiple lines of code that you want to run, you can use newline characters to separate lines. For example, you can print out the sine of an angle with:

```
PyRun_SimpleString("import math\na = math.sin(45)\nprint('The sine of 45 is ' + a)");
```

This function is a simplified version of int PyRun_SimpleStringFlags(const char *command, PyCompilerFlags *flags). This not only takes the command string, but also takes a struct of compiler flags for the Python compiler. You will need to check the development documentation online to see the details for these compiler flags. Let's say that you have a more complicated bit of code to execute. There are equivalent functions to work with Python script files. The simplified version is int PyRun_SimpleFile(FILE *fp, const char *filename). You actually hand in two references to your script. The first is a file handle that you get from the C function fopen() to open your script file, and the seconde is the name of the script you just opened. You will need to open your script file with the read permission. You also now need to worry about whether your program will have the correct file permissions on the file system to open this script. Proper coding means you should check this call to fopen() to verify that it completed and gave you a valid file handle. This simplified version doesn't use any compiler flags, and closes the file handle after the function returns. The full version of the function is int PyRun_SimpleFileExFlags(FILE *fp, const char *filename, int closeit, PyCompilerFlags *flags). If closeit is true, then the file handle is closed. If the script is something you will want to run several times, set closeit to false so the file handle remains open. You can set any flags for the Python interpreter in the flags struct, similar to the PyRun_SimpleStringFlags() function call. If this simple way of running code isn't powerful enough, there are ways of interacting with the interpreter in a more direct fashion. The first step is learning how to send data back and forth between the Python interpreter and the main body of your C program. The basic workflow is to convert your C variables to their Python equivalents, then call the Python functions you wish to use, and convert the Python results back into their equivalents within C. Python is an object-oriented language, so the core of communicating with the interpreter happens with the Py_Object construct. This provides the base for all the other types of objects you can use to communicate with Python. For example, create a Python string object with:

```
PyObject *pName;
pName = PyString_FromString("print('Hello World')");
```

You can then use this Python object when using Python functions. . You can also get access to Python functions from your C code. You store a reference to the function in a PyObject, just as with data objects. The first step is to get the dictionary of the function names for the module in question with:

```
my_module = PyImport_AddModule("__main__");
my_dict = PyModule_GetDict(my_module);
```

Once you have the dictionary, you can get a reference to specific function with:

```
my_func = PyDict_GetItemString(my_dict, func_name);
```

After doing this you will find that where func_name is a string containing the function you want access to, you can then run the function with a command like:

Extend your Python

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

With the Python interpreter, you aren't just limited to what is already available; you can extend the available functionality by defining your own Python objects, with their own methods and data, within your C code. These newly-created objects can then be called from within the Python interpreter. They are defined as static objects, with code like:

```
static PyObject* my_func(PyObject *self,
PyObject *args) {
        ...
  }
```

The new PyObject contains the executable code for the methods to be used for your new object. You also need to create a method definition to be able to register the details with the Python interpreter. You do this by creating a PyMethodDef array:

```
static PyMethodDef my_methods[] = {
  {"my_method", my_func, METH_VARARGS,
        "This is my method"},
    {NULL, NULL, 0, NULL}
          };
```

Once you have finished these two parts, you are almost ready to start using your new module code. You need to initialise it with the function:

```
Py_InitModule("my_module", my_methods);
```

Now, you can import this new module in your Python code, just like any other module installed on your system. Within your Python code, you can write::

```
import my_module
my_module.my_method()
```

One thing to be aware of is that this level of control also gives you a huge level of responsibility. For example, you will need to start worrying about things like reference counts for objects. The garbage collector for the interpreter needs to know when an object is okay to delete. You need to increment the reference count each time something points to your newly created object. Every time a reference is removed, you need to decrement the counter.

You can also create multiple sub-interpreters within a single program. You can create a new sub-interpreter with the function Py_NewInterpreter(). This way, you can have multiple Python threads running concurrently, and mostly independently. When you are done, you can shut them down with the function Py_EndInterpreter(). There is no limit to what you can do with all of this power.

```
PyObject_CallObject(my_func, NULL);
```

With this access, you should be able to do just about anything you wish in Python.

Up to now, we have been specifically looking at code interacting with the Python interpreter. But there are occasions when you want to allow the end user to have access to the interpreter. In these cases, you probably want to give your user access to a full Python console. You can do just such a thing with the function call Py_Main(argc, argv), where you hand in the argc and argv that you have from the C side of your program. This is fine for a console-based program, but for a GUI program, you need to create some kind of terminal window to allow the user to interact with the Python interpreter. This console will automatically continue to run until the user explicitly quits from Python.

The last thing you need to do is to clean up after the interpreter. You can do this with the function void Py_Finalize(). The major issue with this function is that it destroys objects in a random order. If they depend on other objects, they may not be able to get cleaned up correctly. If you then try and re-initialise the interpreter again, it may fail due to an unclean finalisation step.

Now that you have your program written, you need to compile it. You need to include flags to tell the compiler where to find everything. Luckily, you can get these from Python itself. The flags needed for compiling are available with the command python-config --cflags. You also need to know where to find the libraries to link in, which are available with python-config --ldflags. After completing all of this you will have access to Python anywhere, even within another program.

Code listing

```
# A simple way to run Python code
#include <Python.h>

int main(int argc, char *argv[]) {
  Py_SetProgramName(argv[0]);
  # Initialize the Python
interpreter
  Py_Initialize();
  # Run your Python code
  PyRun_SimpleString("from time
import time,ctime\n"
                     "print 'Today
is',ctime(time())\n");
  # Don't forget to clean up
  Py_Finalize();
  return 0;
}
-----------------------------------
# You can create an interactive
Python console
#include <Python.h>
int main(int argc, char *argv[]) {
  Py_Initialize();
  Py_Main(argc, argv);
  Py_Finalize();
}
-----------------------------------
# You can even run a script file
#include <Python.h>

int main(int argc, char *argv[]) {
  FILE *fp;
  Py_Initialize();
  fp = fopen("my_script.py", "r");
  PyRun_SimpleFile(fp, "my_script.
py");
  Py_Finalize();
  fclose(fp);
}
```

WHY PYTHON

It's the official language of the Raspberry Pi. Read the docs at python.org/doc

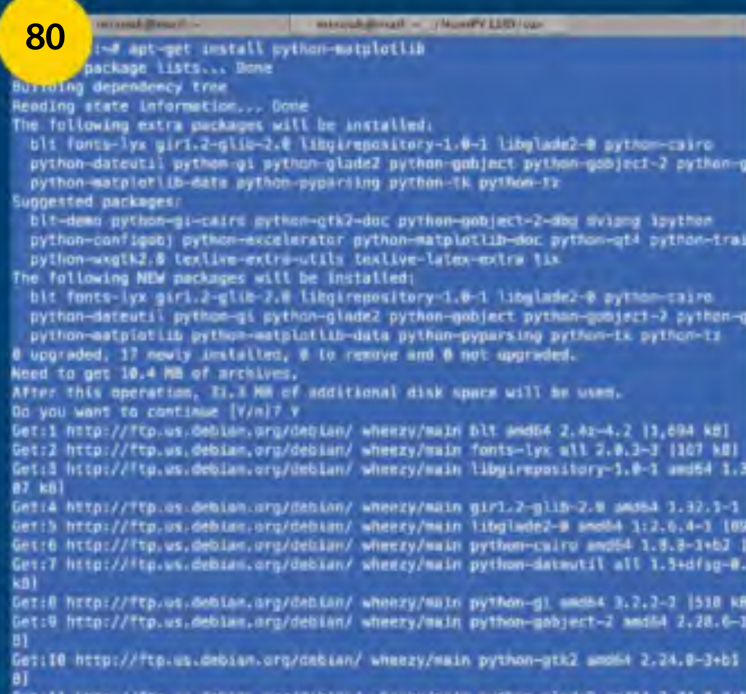
Work with Python

"Python is the world's most popular easy-to-use open source language"

74

Add-on Information
LUD Reddit Viewer
Type: Media sources
Author: LUD
Version: 0.0.1
Rating:
Summary: LUD Ent Example
Description
LUD Example Addon to demonstrate XBMC Extension System.
AddonScriptorDE.
Configure
Uninstall
Disable
Change log

80



apt-get install python-matplotlib
Reading state information... Done
The following extra packages will be installed:
Suggested packages:
The following NEW packages will be installed:
Need to get 10.4 MB of archives.
Do you want to continue [Y/n]? Y

PYTHON FOR **PROFESSIONALS**

Python is relied upon by web developers, engineers and academic researchers across the world. Here's how to put your Python skills to professional use

System administration

Get the most out of Python in handling all of the day-to-day upkeep that keeps your system healthy

System administration tasks are some of the most annoying things that you need to deal with when you have to maintain your own system. Because of this, system administrators have constantly been trying to find ways to automate these types of tasks to maximise their time. They started with basic shell scripts, and then moved on to various scripting languages. For a long time, Perl had been the language of choice for developing these types of maintenance tools. However, Python is now growing in popularity as the language to use. It has reached the point where most Linux distributions have a Python interpreter included in order to run system scripts, so you shouldn't have any excuse for not writing your own scripts.

Because you will be doing a lot system level work, you will have most need of a couple of key Python modules. The first module is 'os'. This module provides the bulk of the interfaces to interacting with the underlying system. The usual first step is to look at the environment your script is running in to see what information might exist there to help guide your script. The following code gives you a mapping object where you can interact with the environment variables active right now:

```
import os
os.environ
```

You can get a list of the available environment variables with the function "os.environs.keys()", and then access individual variables with "os.environs[key]". These environment variables are used when you spawn a subprocess, as well. So you will want to change values, like the PATH or the current working directory, in order for you to run these subprocesses correctly. While there is a "putenv" function that edits these values for you, it unfortunately does not exist on all systems. So the better way to do this is to edit the values directly within the environs mapping.

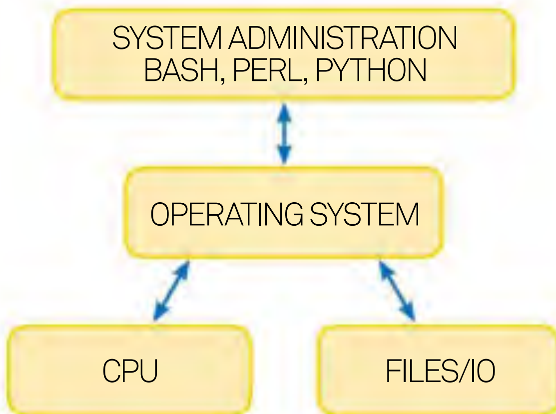
Another category of tasks you may want to automate is when working with files. For example, you can get the current working directory with code like

```
cwd = os.getcwd()
```

You can then get a list of the files in this directory with

```
os.listdir(cwd)
```

You can move around the file system with the function "os.chdir(new_path)". Once you've found the file you are interested in, you can open it with "os.open()" and open it for reading, writing and/or appending. You can then read or write to it with the functions "os.read()" and "os.write()". Once done, you can close the file with "os.close()".



Left Python scripts enable you to instruct and interact with your operating system

Running subprocesses from Python

The underlying philosophy of Unix is to build small, specialised programs that do one job extremely well. You then chain these together to build more complex behaviours. There is no reason why you shouldn't use the same philosophy within your Python scripts. There are several utility programs available to use with very little work on your part. The older way of handling this was through using functions like "popen()" and "spawnl()" from the os module, but a better way of running other programs is by using the subprocess module instead. You can then launch a program, like ls, by using:

```
import subprocess
subprocess.run(['ls', '-l'])
```

This provides you with a long file listing for the current directory. The function "run()" was introduced in Python 3.5 and is the suggested way of handling this. If you have an older version, or if you require more control than that, then you can employ the underlying "popen()" function that we mentioned earlier instead. If you want to get the output, you can use the following:

```
cmd_output = subprocess.run(['ls', '-l'],
stdout=subprocess.PIPE)
```

The variable "cmd_output" is a CompletedProcess object that contains the return code and a string holding the stdout output.

Scheduling with cron

Once you have your script all written up, you may want to schedule them to run automatically without your intervention. On Unix systems, you can have cron run your script on whatever schedule is necessary. The utility "crontab -l" lists the current contents of your cron file, and "crontab -e" lets you edit the scheduled jobs that you want cron to run.

Web development

Python has several frameworks available for all of your various web development tasks. We will look at some of the more popular ones

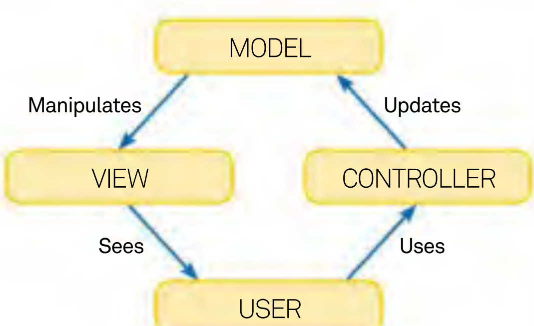
With the content and the bulk of the computing hosted on a server, a web application can better guarantee a consistent experience for the end user. The popular Django framework provides a very complete environment of plugins and works on the DRY principle (Don't Repeat Yourself). Because of this, you should be able to build your web application very quickly. Since Django is built on Python, you should be able to install it with "sudo pip install Django". Most distributions should have a package for Django, too. Depending on what you want to do with your app, you may need to install a database like MySQL or postgresql to store your application data.

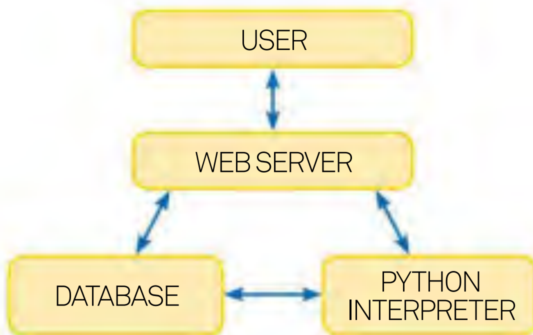
There are Django utilities available to automatically generate a starting point for your new project's code:

```
django-admin startproject newsite
```

This command creates a file named "manage.py" and a subdirectory named "newsite". The file "manage.py" contains several utility functions you can use to administer your new application. The newly created subdirectory contains the files "__init__.py", "settings.py", "urls.py" and "wsgi.py". These files and the subdirectory they reside in comprise a Python package that gets loaded when your web site is started up. The core configuration for your site can be found in the file "settings.py". The URL declarations, basically a table of contents for your site, are stored in the file "urls.py". The file "wsgi.py" contains an entry point for WSGI-compatible web servers.

Once your application is done, it should be hosted on a properly configured and hardened web server. But, this is inconvenient if you are in the process of developing your web application. To help you out, Django has a web server built into the framework. You can start it up by changing directory to the "newsite" project directory and running the following command:






```
python manage.py runserver
```

This will start up a server listening to port 8000 on your local machine. Because this built in server is designed to be used for development, it automatically reloads your Python code for each request. This means that you don't need to restart the server to see your code changes.

All of these steps get you to a working project. You are now ready to start developing your applications. Within the "newsite" subdirectory, you can type:

```
python manage.py startapp newapp
```

This will create a new subdirectory named "newapp", with the files "models.py", "tests.py" and "views.py", among others. The simplest possible view consists of the code:

```
from django.http import HttpResponse
def index(request):
 return HttpResponse("Hello world")
```

This isn't enough to make it available, however. You will also need to create a URLconf for the view. If the file "urls.py" doesn't exist yet, create it and then add the code:

```
from django.conf.urls import url
from . Import views
 urlpatterns = [ url(r'^$', views.index,
   name='index'), ]
```

The last step is to get the URL registered within your project. You can do this with the code

```
from django.conf.urls import include, url
from django.contrib import admin
urlpatterns = [ url(r'^newapp/',
include('newapp.urls')),
```

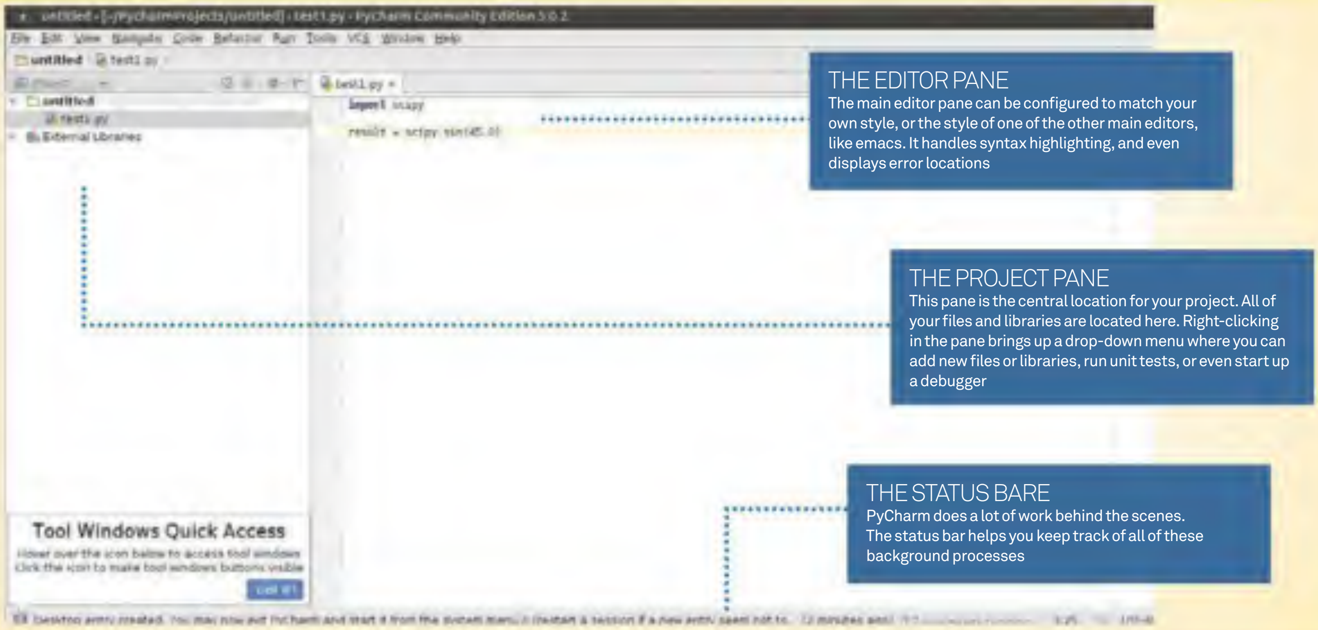
Left Python interpreters work with your databases to power a web server

Bottom The Model-View-Controller architecture is often used for UIs

Virtual environments

When you start developing your own applications, you may begin a descent into dependency hell. Several Python packages depend on other Python packages. This is its strength, but also its weakness. Luckily, you have virtualenv available to help tame this jungle. You can create new virtual environments for each of your projects. Thankfully with this, you can be sure to capture all of the dependencies for your own package.

Using the PyCharm IDE




```
url(r'^admin', admin.site.urls), ]
```

This needs to be put in the "urls.py" file for the main project. You can now pull up your newly created application with the URL **http://localhost:8000/newapp/**.

The last part of many applications is the database side. The actual connection details to the database, like the username and password, are contained in the file "settings.py". This connection information is used for all of the applications that exist within the same project. You can create the core database tables for your site with this command:

```
python manage.py migrate
```

For your own applications, you can define the data model you need within the file "models.py". Once the data model is created, you can add your application to the INSTALLED_APPS section of the "settings.py" so that django knows to include it in any database activity. You initialize it with:

```
python manage.py makemigrations newapp
```

Once these migrations have been created, you need to apply them to the database by using the command:

```
python manage.py migrate
```

Any time you make changes to your model, you will need to run the makemigrations and migrate steps again.

Once you have your application finished, you can make the move to the final hosting server. Don't forget to check the available code within the Django framework before putting too much work into developing your own code.

Terminal development environments

When you are in the middle of developing your application, you may need to have several different terminal windows open in order to have a code editor open, a monitor on the server, and potentially testing output. If you are doing this on your own machine, this isn't an issue. But, if you are working remotely, you should look into using tmux. It can provide a much more robust terminal environment for you.

Other Python frameworks

While Django is one of the most popular frameworks around for doing web development, it is by no means the only one around. There are several others available that may prove to be a better fit for particular problem domains. For example, if you are looking for a really self-contained framework, you could look at web2py. Everything you need to be able to have a complete system, from databases to web servers to a ticketing system, are included as part of the framework. It is so self-contained that it can even run from a USB drive

If you need even less of a framework, there are several mini-frameworks that are available. For example, CherryPy is a purely Pythonic multi-threaded web server that you can embed within your own application. This is actually the server included with TurboGears and web2py. A really popular microframework is a project called flask. It includes integrated unit testing support, jinja2 templating and RESTful request dispatching.

One of the oldest frameworks around is zope, now up to version 3. This latest version was renamed BlueBream. Zope is fairly low-level, however. You may be more interested in looking at some of the other frameworks that are built on top of what is provided by zope. For example, pyramid is a very fast, easy to use framework that focuses on the most essential functions required by most web applications. To this end, it provides templating, the serving of static content, mapping of URLs to code, among other functions. It handles this while providing tools for application security.

If you are looking for some ideas, there are several open source projects that have been built using these frameworks, from blogs, to forums to ticketing systems. These projects can provide some best-practices when you go to construct your own application.

Computational science

Python is fast becoming the go-to language for computational science

Python has become one of the key languages used in science. There is a huge number of packages available to handle almost any task that you may have and, importantly, Python knows what it isn't good at. To deal with this, Python has been designed to easily incorporate code from C or FORTRAN. This way, you can offload any heavy computations to more efficient code.

The core package of most of the scientific code available is numpy. One of the problems in Python is that the object oriented nature of the language is the source of its inefficiencies. With no strict types, Python always needs to check parameters on every operation. Numpy provides a new datatype, the array, which helps solve some of these issues. Arrays can only hold one type of object, and because Python knows this it can use some optimisations to speed things up to almost what you can get from writing your code directly in C or FORTRAN. The classic example of the difference is the for loop. Lets say you wanted to scale a vector by some value, something like a*b. In regular Python, this would look like

```
for elem in b:
c.append(a * elem)
In numpy, this would look like:
  a*b
```

Left The numpy package makes it simple to visualise your data

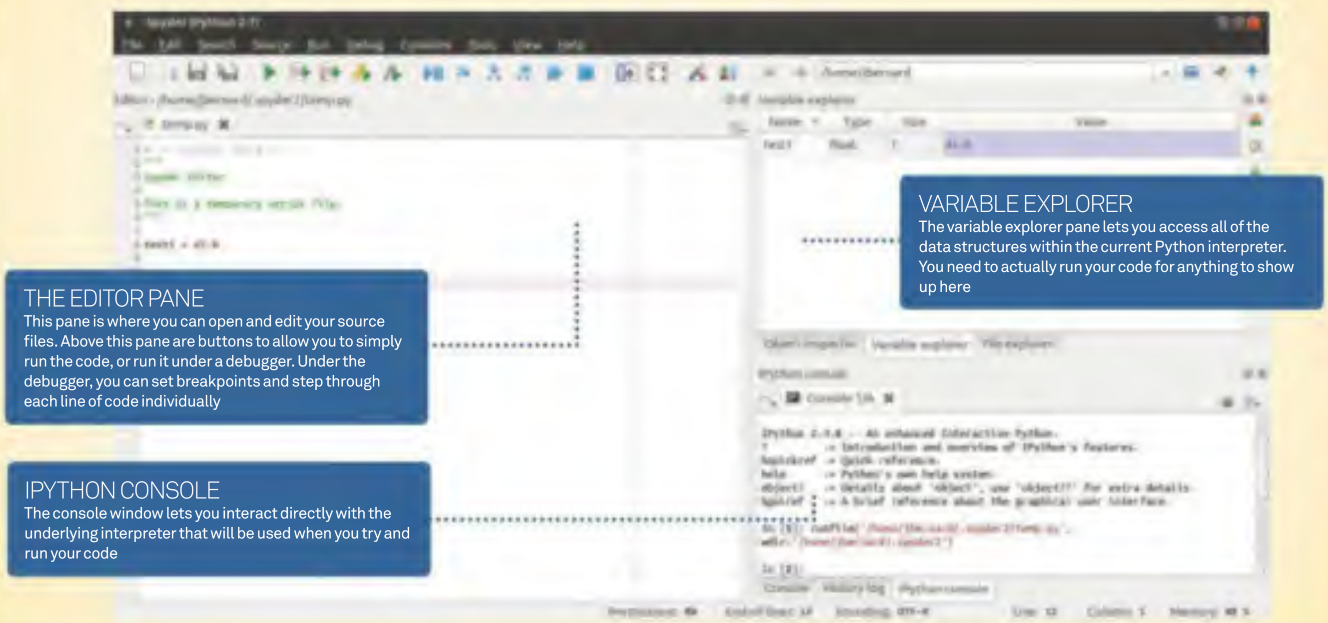
So, not only is it faster, it is also written in a shorter, clearer form. Along with the new datatype, numpy provides overloaded forms of all of the operators that are of most use, like multiplication or division. It also provides optimised versions of several functions, like the trig functions, to take advantage of this new datatype.

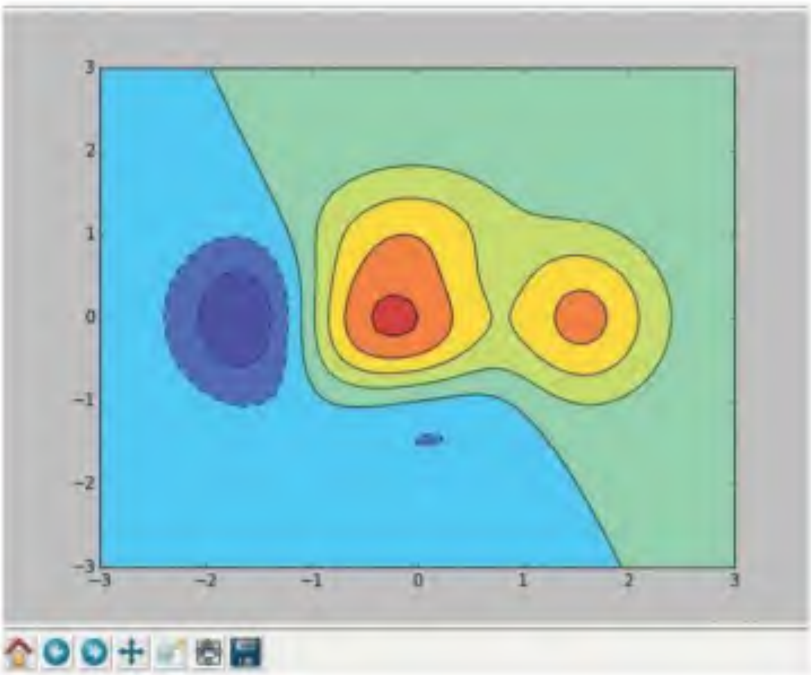
The largest package available, that is built on top of numpy, is scipy. Scipy provides sub-sections in several areas of science. Each of these sub-sections need to be imported individually after importing the main scipy package. For example, if you are doing work with differential equations, you can use the "integrate" section

Parallel Python

One of the really powerful parts of Ipython (or jupyter) is that it is built with a client/server model. This means that it is relatively easy to setup multiple machines to act as a server pool. You can then farm out multiple tasks to these other machines to get even more work done. While this doesn't run any particular function in parallel, it does let you run longer functions in the background while you work on something else.

Spyder, the IDE for scientists





Above The ability to generate complex plots is essential

to solve them with code that looks like

```
import scipy
import scipy.integrate
result = scipy.integrate.quad(lambda x:
sin(x), 0, 4.5)
```

Differential equations crop up in almost every scientific field. You can do statistical analysis with the "stats" section. If you want to do some signal processing, you can use the "signal" section and the "fftpack" section. This package is definitely the first stop for anyone wanting to do any scientific processing.

Once you have collected your data, you usually need to graph it, in order to get a visual impression of patterns within it. The primary package you can use for this is matplotlib. If you have ever used the graphics package in R before, the core design of matplotlib has borrowed quite a few ideas. There are two categories of functions for graphing, low-level and high-level. High-level functions try to take care of as many of the menial tasks, like creating a plot window, drawing axes, selecting a coordinate system, as possible. The low-level functions give you control over almost every part of a plot, from drawing individual pixels to controlling every aspect of the plot window. It also borrowed the idea of drawing graphs into a memory based window. This means that it can draw graphs while running on a cluster.

If you need to do symbolic math, you may be more used to using something like Mathematica or Maple. Luckily, you have sympy that can be used to do many of the same things. You can use Python to do symbolic calculus, or to solve algebraic equations. The one weird part of sympy is that you need to use the "symbols()" function to tell sympy what variables are valid to be considered in your equations. You can then start doing manipulations using these registered variables.

You may have large amounts of data that you need to work with and analyze. If so, you can use the pandas package to help deal with that. Pandas has support for several different file formats, like CSV files, Excel spreadsheets or HDF5. You can merge and join datasets, or do slicing or subsetting. In order to get the best performance out of the code, the heaviest lifting is done by Cython code that incorporates functions written in C. Quite a few ideas on how to manipulate your data was borrowed from how things are done in R.

You now have no reason not to start using Python for your scientific work. You should be able to use it for almost any problem that comes up!

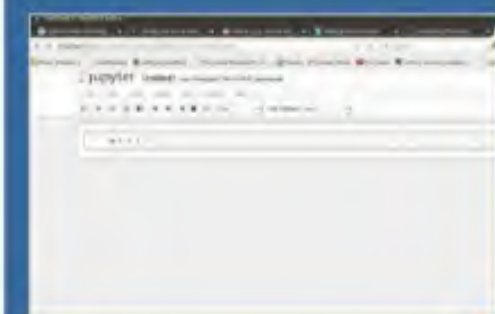
The need for speed

Sometimes you need as much speed as your are capable of pushing on your hardware. In these cases, you always have the option of using Cython. This lets you take C code from some other project, which has probably already been optimised, and use it within your own Python program. In scientific programming, you are likely to have access to code that has been worked on for decades and is highly specialised. There is no need to redo the development effort that has gone into it.

Interactive science with jupyter

For a lot of scientific problems, you need to play with your data in an interactive way. The original way you would do this was to use the Ipython web notebook. This project has since been renamed Jupyter. For those who have used a program like Mathematica or Maple, the interface should seem very familiar. Jupyter starts a server process, by default on port 8888, and then will open a web browser where you can open a worksheet. Like most other programs of this type, the entries run in chronological order, not in the order that they happen on the worksheet. This can be a bit confusing at first, but it means that if you go to edit an earlier entry, all of the following entries need to be re-executed manually in order to propagate that change through the rest of the computations.

Jupyter has support for pretty printing math within the produced web page. You can also mix documentation blocks and code blocks within the same page. This means that you can use it to produce very powerful educational material, where students can read about some technique, and then actually run it and see it in action. By default, Jupyter will also embed matplotlib plots within the same worksheet as a results section, so you can see a graph of some data along with the code that generated it. This is huge in the growing need for reproducible science. You can always go back and see how any analysis was done and be able to reproduce any result at all.



Above Jupyter Notebook is a web application that is used for creating and sharing documents that contain live code and equations

Robotics and electronics

Robotics is the most direct interface between your code and the real world around you

Robotics is the most direct way that your code can interact with the world around you. It can read actual sensor information and move real actuators and get real work done.

The first thing your robot needs is the ability to sense the world around it. The one sense that we as humans feel is most useful is sight. With web cameras being so cheap and easy to connect to hardware, vision is easy to give to your robot. The real problem is how to interpret this data. Luckily, you can use the OpenCV project to do just that. It is a vision package that can provide simple image gathering and processing, to extremely complex functions like face recognition and extraction of 3D objects. You can identify and track objects moving through your field of view. You can also use OpenCV to give you robot some reasoning capabilities, too. OpenCV includes a set of functions for machine learning, where you can do statistical classification or data clustering, and use it to feed decision trees or even neural networks.

Another important sense that you may want to use is sound. The jasper project is one that is developing a complete voice control system. This project would give you the structure you need to give your robot the ability to listen for and respond to your verbal commands. The project has gotten to the point where you can give it a command and the voice recognition software can translate this into text. You then need to build a mapping of what pieces of text correspond to what commands to execute.

There are lots of other sensors you could have, but this begins to leave the realm of store-bought hardware. Most other sensors, like temperature, pressure, orientation or location, need specialised hardware that needs to be interfaced to the computer brain for your robot. This

Arduino

In contrast to the Raspberry Pi, which runs a full OS from its SD card, the Arduino boards are microcontrollers rather than complete computers. Instead of running an OS, the Arduino platform executes code that is interpreted by its firmware. It is mainly used to interface with hardware such as motors and servos, sensors, and devices such as LEDs, and is incredibly capable in this regard. Arduinos are widely used in robotics projects and can be a powerful complement to the Pi.

Raspberry Pi

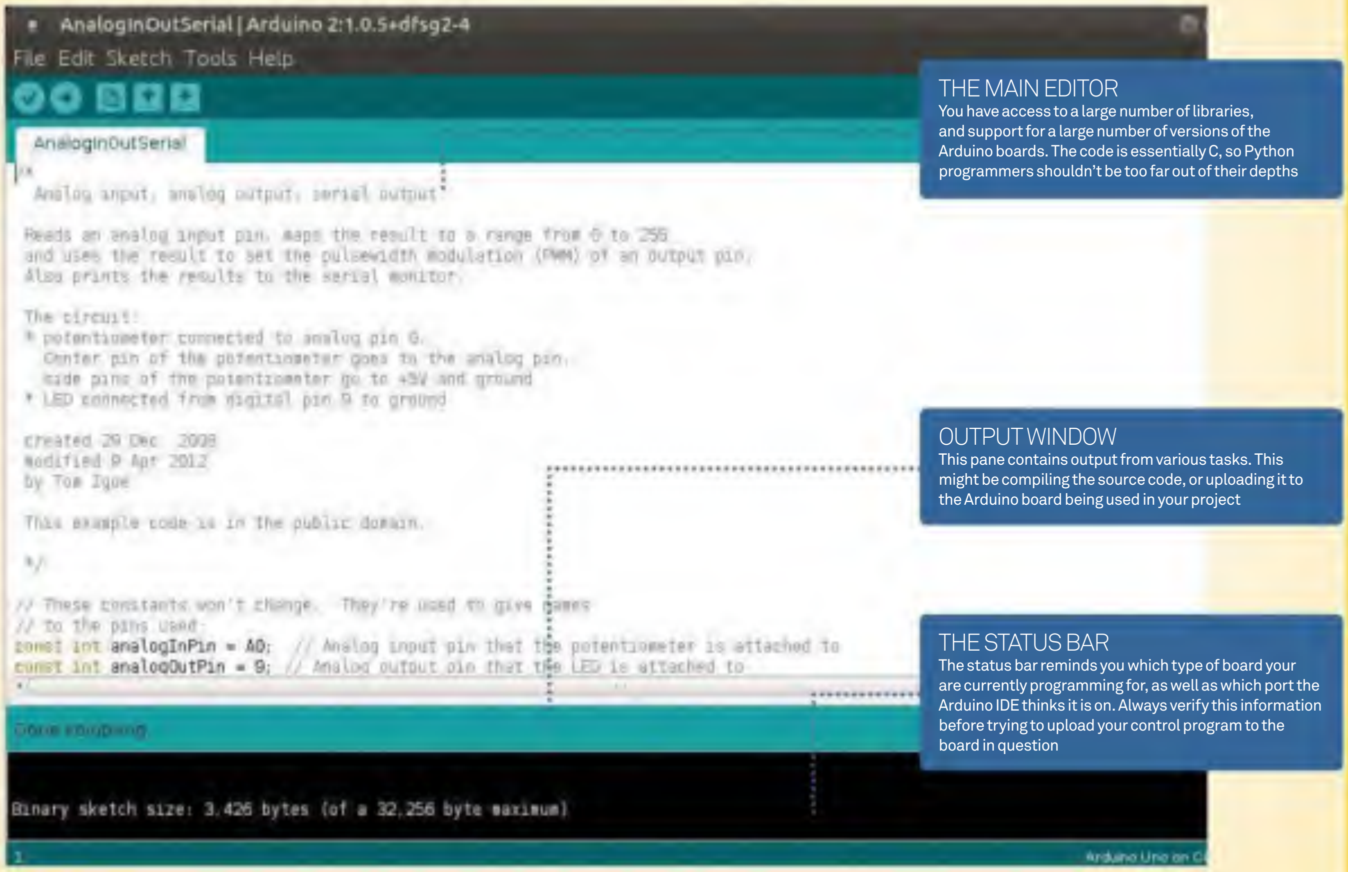
While we haven't discussed what kind of computer to use for your robotics project, you should consider the famous Raspberry Pi. This tiny computer should be small enough to fit into almost any robot structure that you might be building. Since it is already running Linux and Python, you should be able to simply copy your code development work to the Pi. It also includes its own IO bus so that you can have it read it's own sensors.

ROS – Robot Operating System

While you could simply write some code that runs on a standard computer and a standard Linux distribution, this is usually not optimal when trying to handle all of the data processing that a robot needs when dealing with events in realtime. When you reach this point, you may need to look at a dedicated operating system – the Robot Operating System (ROS). ROS is designed to provide the same type of interface between running code the computer hardware it is running on, with the lowest possible overhead. One of the really powerful features of ROS is that it is designed to facilitate communication between different processes running on the computer, or potentially over multiple computers connected over some type of network. Instead of each process being a silo that is protected from all other processes, ROS is more of a graph of processes with messages being passed between them all.

Because ROS is a complete operating system, rather than a library, it is wrong to think that you can use it in your Python code. It is better to think that you can write Python code that can be used in ROS. The fundamental design is to be as agnostic as possible. This means that interfaces to your code should be clean and not particularly care where they running or who is talking to them. Then, it can be used within the graph of processes running within ROS. There are standard libraries available that allow you to do coordinate transformations, useful for figuring out where sensors or limbs are in space. There is a library available for creating preemptible tasks for data processing, and another for creating and managing the types of messages that can be handed around the various processes. For extremely time-sensitive tasks, there is a plugin library that allows you to write a C++ plugin that can be loaded within ROS packages.

For low-level work, check out Arduinos



means it is time to get your soldering iron out. As for reading the data in, this is most often done over a basic serial connection. You can then use the pySerial module to connect to the serial port and read data off the connection. You can use:

```
import serial
```

to load the module and start communicating with your sensor. The problem is that this is a very low-level way to communicate. You, as the programmer, are responsible for all of the details. This includes communication speed, byte size, flow control; basically everything. So this will definitely be an area of your code where you should plan on spending some debugging time.

Now that you have all of this data coming in, what will you do with it? You need to be able to move actuators out in the world and have real effects. This could be motors for wheels or tracks, levers to shift objects, or potentially complete limbs, like arms or legs. While you could try and drive these types of electronic devices directly from the output ports of your computer, there usually isn't enough current available to provide the necessary power. So, you will need to have some off-board brains capable of handling the supplying of power to these devices. One of the most popular candidates for this task is the Arduino.

Luckily, the Arduino is designed to connect to the serial port of your computer, so you can simply use pySerial to talk to it. You can send commands to code that you have written and uploaded to the Arduino to handle the actual manipulations of the various actuators. The Arduino can talk back, however. This means that you can read feedback data to see what effect your movements have had. Did you end up turning your wheels as far as you wanted to? This means that you could also use the Arduino as an interface between your sensors and the computer, thus simplifying your Python code even more. There are loads of add-on modules available, too, that might be able to provide the sensing capabilities that you require straight out of the box. There are also several models of Arduino, so you may be able to find a specialised model that best fits your needs.

Now that you have all of this data coming in and the ability to act out in the real world, the last step is giving your robot some brains. This is where the state of the art unfortunately does not live up to the fantasy of R2-D2 or C-3P0. Most of your actual innovative coding work will likely take place in this section of the robot. The general term for this is artificial intelligence. There are several projects currently underway that you could use as a starting point to giving your robot some real reasoning capability, like SimpleAI or PyBrain.

Bypassing the GIL

For robotics work, you may need to run some code truly in parallel, on multiple CPUs. Python currently has the GIL, which means that there is a fundamental bottleneck built into the interpreter. One way around this is to actually run multiple Python interpreters, one for each thread of execution. The other option is to move from Cpython to either Jython or IronPython, as neither has a GIL.

Make extensions for XBMC with Python

Python is the world's most popular easy-to-use open source language. Learn how to use it to build your own features for XBMC, the world's favourite FOSS media centre

Resources

XBMC: www.xbmc.org/download
Python 2.7x
Python IDE (optional)
Code on FileSilo

XBMC is perhaps the most important thing that has ever happened in the open source media centre space. It started its life on the original Xbox videogames console and since then it has become the de facto software for multimedia aficionados. It also has been forked into many other successful media centre applications such as Boxee and Plex. XBMC has ultimately grown into a very powerful open source application with a solid community behind it. It supports almost all major platforms, including different hardware architectures. It is available for Linux, Windows, Mac OS X, Android, iOS and Raspberry Pi.

In these pages we will learn to build extensions for XBMC. Extensions are a way of adding features to XBMC without having to learn the core of XBMC or alter that core in any way. One additional advantage is that XBMC uses Python as its scripting language, and this can be also used to build the extensions. This really helps new developers get involved in the project since Python is easy to learn compared to languages like C/C++ (from which the core of XBMC is made).

XBMC supports various types of extensions (or Add-ons): Plugins, Programs and Skins. Plugins add features to XBMC. Depending on the type of feature, a plug-in will appear in the relevant media section of XBMC. For example, a YouTube plug-in would appear in the Videos section. Scripts/Programs are like mini-applications for XBMC. They appear in the Programs section. Skins are important since XBMC is a completely customisable application – you can change the look and feel of just about every facet of the package.

Depending upon which category your extension fits, you will have to create the extension directory accordingly. For example…

Plug-ins:

plugin.audio.ludaudi: An audio plug-in
plugin.video.ludvidi: A video plug-in
script.xxx.xxx: A program

In this tutorial we will build an XBMC plug-in called LUD Entertainer. This plug-in will provide a nice way to watch videos from Reddit from within XBMC. Our plug-in will show various content such as trailers and documentaries from Reddit. We'll also allow our users to add their own Subreddit. Each video can then be categorised as Hot, New, Top, Controversial etc. With this plug-in we will demonstrate how easy it is hook into XBMC's built-in method to achieve a very high-quality user experience.

Due to space limitations, we aren't able to print the full code here. We recommend downloading the complete code from FileSilo.

01 Preparing the directory structure

As we have mentioned previously, each XBMC extension type follows a certain directory naming convention. In this case we are building a video plug-in, so the plug-in directory name would be **plugin.video.ludlent**. But that's just the root directory name – we will need several other folders and files as well.

The following describes the directory structure of LUD Linux Entertainer:

plugin.video.ludent – Root Plugin directory
|-- addon.xml
|-- changelog.txt
|-- default.py
|-- icon.png
|-- LICENSE.txt
|-- README
`-- resources
  |-- lib
  `-- settings.xml

02 Creating addon.xml

An **addon.xml** file needs to be created in the root of the extension directory. The addon.xml file contains the primary metadata from a XBMC extension. It contains overview, credits, version information and dependencies information about the extension.

The root element of addon.xml is the **<addon>** element. It is defined as:

```
<addon id="plugin.video.
ludent" name="LUD HSW Viewer"
version="0.0.1" provider-
name="LUDK">
rest of the content is placed here
</addon>
```

Here, **id** is the identifier for the plug-in, so it should be unique among all the XBMC extensions, and id is also used for the directory name; **version** tells XBMC the extension version number, which helps in its ability to deliver automatic updates – XBMC follows the Major.Minor.Patch versioning convention; **name** is the English title of the plug-in.

Note: Steps 3 to 5 cover entries that need to be added within the **addon.xml** file.

03 Adding dependency information

Dependency inside an extension is managed using the **<requires>** element.

```
<requires>
<import addon="xbmc.python"
version="2.1.0"/>
<import addon="plugin.video.
youtube" version="3.0.0"/>
<import addon="plugin.video.vimeo"
version="2.3.0"/>
<import addon="plugin.video.
dailymotion_com" version="1.0.0"/>
</requires>
```

In the above code we have added a dependency to a library called **xbmc.python** version 2.1. Currently it is added as a mandatory dependency. To make the dependency optional you will need to add **optional="true"**; eg `<import addon="kunal.special" version="0.1.0" optional="true" />`

In the above example we have added core dependency xbmc.python to 2.1.0 because it's the version shipped with XBMC version Frodo 12.0 and 12.1 . If you were to add xbmc.python to 2.0 then it would only work in XBMC Eden 11.0 and not in the latest version.

For the current version of XBMC 12.1, the following versions of core XBMC components are shipped:

xbmc.python 2.1.0
xbmc.gui 4.0.0
xbmc.json 6.0.0
xbmc.metadata 2.1.0
xbmc.addon 12.0.0

In addition to xbmc.python we are also adding some third-party plug-ins as dependencies, such as plugin.video.youtube. These plug-ins will be installed automatically when we install plugin.video.ludent.

04 Setting up the provider and entry point

Our extension is supposed to provide the video content for XBMC. In order to convey that, we have to set up the following element:

```
<extension point="xbmc.python.
pluginsource" library="default.
py">
<provides>video</provides>
</extension>
```

Here, the **library** attribute sets up the plug-in entry point. In this example default.py will be executed when the user activates the plug-in. The **<provides>** elements sets up the media type it provides. This also gets reflected in the placement of the plug-in. Since ours is a video plug-in, it will show up in the Videos section of XBMC.

05 Setting up plug-in metadata

Metadata about the plug-in is provided in **<extension point="xbmc.addon.metadata">**. The following are the important elements...

<platform>: Most of the time, XBMC extensions are cross-platform compatible. However, if you depend on the native platform library that is only available on certain platforms then you will need to set the supported platforms here. Accepted values for the platform are: all, linux, osx, osx32, osx64, ios (Apple iOS) , windx (Windows DirectX), wingl (Windows OpenGL) and android.

<summary lang="en">: This gives a brief description of the plug-in. Our example sets the language attribute as English, but you can use other languages too.
<description>: A detailed description of the plug-in.
<website>: Webpage where the plug-in is hosted.
<source>: Source code repository URL. If you are hosting your plug-in on GitHub, you can mention the repository URL here.
<forum>: Discussion forum URL for your plug-in.
<email>: Author email. You can directly type email or use a bot-friendly email address like max at domain dot com.

06 Setting changelog, icon, fanart and licence

We need a few additional files in the plug-in directory...

changelog.txt: You should list the changes made to your plug-in between releases. The changelog is visible from the XBMC UI.

An example changelog:
0.0.1
- Initial Release
0.0.2
- Fixed Video Buffering Issue

icon.png: This will represent the plug-in in the XBMC UI. It needs to be a non-transparent PNG file of size 256x256.

fanart.jpg (optional): The fanart.jpg is rendered in the background if a user selects the plug-in in XBMC. The art needs to be rendered in HDTV formats, so its size can range from 1280x720 (720p) up to the maximum 1920x1080 (1080p).

License.txt: This file contains the licence of the distributed plug-in. The XBMC project recommends the use of the Creative Commons Attribution-ShareAlike 3.0 licence for skins, and GPL 2.0 for add-ons. However, most of the copyleft licences can be used.

Note: For the purpose of packaging, extensions/add-ons/themes/plug-ins are the same.

07 Providing settings for the plug-in

Settings can be provided by the file **resources/settings.xml**. These are great for user-configurable options.

Partial: resources/settings.xml

```
<settings>
<category label="30109">
<setting id="filter" type="bool"
label="30101" default="false"/>
<setting type="sep" />
<setting id="showAll" type="bool"
label="30106" default="false"/>
<setting id="showUnwatched"
type="bool" label="30107"
default="true"/>
<setting id="showUnfinished"
type="bool" label="30108"
default="false"/>
<setting type="sep" />
<setting id="forceViewMode"
type="bool" label="30102"
default="true"/>
<setting id="viewMode" type="number"
label="30103" default="504"/>
</category>
<category label="30110">
<setting id="cat_hot" type="bool"
label="30002" default="true"/>
<setting id="cat_new" type="bool"
label="30003" default="true"/>
</category>
</settings>
```

Here, **label** defines the language id string which will then be used to display the label. **id** defines the name which will be used for programmatic access. **type** defines the data type you want to collect; it also affects the UI which will be displayed for the element. **default** defines the default value for the setting. You should always use a default value wherever possible to provide a better user experience.

The following are a few important settings types that you can use…

text: Used for basic string inputs.
ipaddress: Used to collect internet addresses.
number: Allows you enter a number. XBMC will also provide an on-screen numeric keyboard for the input.
slider: This provides an elegant way to collect integer, float and percentage values. You can get the slider setting in the following format:

```
<setting label="21223" type="slider"
id="sideinput" default="10"
range="1,1,10" option="int" />
```

In the above example we are creating a slider with min range 1, max range 10 and step as 1. In the **option** field we are stating the data type we are interested in – we can also set option to "float" or "percent".

bool: Provides bool selection in the form of on or off.
file: Provides a way to input file paths. XBMC will provide a file browser to make the selection of file. If you are looking to make selection for a specific type of file you can use audio, video, image or executable instead of file.

folder: Provides a way to browse for a folder…

Example:

```
<setting label="12001" type="folder"
id="folder" source="auto"
option="writeable"/>
```

Here, **source** sets the start location for the folder, while **option** sets the write parameter for the application.

sep & lsep: sep is used to draw a horizontal line in the setting dialog; lsep is used for drawing a horizontal line with text. They do not collect any input but are there for building better user interface elements…

```
<setting label="21212" type="lsep"
/>
```

08 Language support

Language support is provided in the form of the **strings.xml** file located in **resources/languages/[language name]**. This approach is very similar to many large software projects, including Android, where static strings are never used.

resource/language/english/string.xml example:

```
<?xml version="1.0" encoding="utf-8"
standalone="yes"?>
<strings>
<string id="30001">Add subreddit</
string>
<string id="30002">Hot</string>
<string id="30003">New</string>
<string id="30004">Top</string>
<string id="30005">Controversial</
string>
<string id="30006">Hour</string>
<string id="30007">Day</string>
<string id="30008">Week</string>
<string id="30009">Month</string>
<string id="30010">Year</string>
</strings>
```

As you may have seen in the settings.xml example, all the labels are referring to string ids. You can have many other languages as well. Depending upon the language XBMC is running in, the correct language file will be loaded automatically.

Post XBMC Frodo (12.1), strings.xml will be deprecated. Post Frodo, XBMC will be moved to a GNU gettext-based translation system; gettext uses PO files. You can use a tool called xbmc-xml2po to convert strings.xml into equivalent PO files.

09 Building default.py

Since our plug-in is small, it will all be contained inside **default.py**. If you are developing a more complex add-on then you can create supporting files in the same directory. If your library depends upon third-party libraries, you have two ways to go about it. You can either place the third-party libraries into the **resources/lib** folder; or bundle the library itself into a plug-in, then add that plug-in as the dependency in the **addon.xml** file.

Our plug-in works with **reddit.tv**. This is the website from Reddit which contains trending videos shared by its readers. Videos posted on Reddit are actually sourced from YouTube, Vimeo and Dailymotion.

We will be starting off default.py using the following imports:

```
import urllib
import urllib2
…
import xbmcplugin
```

```
import xbmcgui
import xbmcaddon
```

Apart from xbmcplugin, xbmcgui and xbmcaddon, the rest are all standard Python libraries which are available on PyPI (Python Package Index) via pip. You will not need to install any library yourself since the Python runtime for XBMC has all the components built in.

urllib and **urllib2** help in HTTP communication. **socket** is used for network I/O; **re** is used for regular expression matching; **sqlite3** is the Python module for accessing an SQLite embedded database; **xbmcplugin**, **xbmcgui** and **xbmcaddon** contain the XBMC-specific routine.

10 Initialising

During the initialisation process, we will be reading various settings from **settings.xml**. Settings can be read in the following way:

```
addon = xbmcaddon.Addon()
filterRating = int(addon.
getSetting("filterRating"))
filterVoteThreshold = int(addon.getS
etting("filterVoteThreshold"))
```

In order to read settings of type bool you will need to do something like:

```
filter = addon.getSetting("filter")
== "true"
```

We are also setting the main URL, plug-in handle and the user agent for it:

```
pluginhandle = int(sys.argv[1])
urlMain = "http://www.reddit.com"
userAgent = "Mozilla/5.0 (Windows NT
6.2; WOW64; rv:22.0) Gecko/20100101
Firefox/22.0"
opener = urllib2.build_opener()
opener.addheaders = [('User-Agent',
userAgent)]
```

11 Reading localised strings

As mentioned, XBMC uses **strings.xml** to serve up the text. In order to read those strings, you will need to use **getLocalizedString**.

```
translation = addon.
getLocalizedString
translation(30002)
```

In this example, **translation(30002)** will return the string "Hot" when it is running in an English environment.

idFile	idPath	strFilename	playCount	lastPlayed	dateAdded
1	1	plugin://plugin.		2013-08-06 23:47	
2	2	plugin://plugin.	1	2013-08-07 22:42	
3	2	plugin://plugin.	1	2013-08-08 00:09	
4	2	plugin://plugin.	1	2013-08-08 00:55	
5	2	plugin://plugin.	1	2013-08-08 00:58	

12 Building helper functions

In this step we will look at some of the important helper functions.

getDbPath(): This returns the location of the SQLite database file for videos. XBMC stores library and playback information in SQLite DB files. There are separate databases for videos and music, located inside the **.xbmc/userdata/Database** folder. We are concerned with the videos DB. It is prefixed with 'MyVideos'…

```
def getDbPath():
    path = xbmc.
translatePath("special://userdata/
Database")
    files = os.listdir(path)
    latest = ""
    for file in files:
        if file[:8] == 'MyVideos'
and file[-3:] == '.db':
            if file > latest:
                latest = file
    return os.path.join(path,
latest)
```

getPlayCount(url): Once we have the database location, we can get the play count using a simple SQL query. The MyVideo database contains a table called **files**, which keeps a record of all the video files played in XBMC by filename. In this case it will be URL.

```
dbPath = getDbPath()
conn = sqlite3.connect(dbPath)
c = conn.cursor()

def getPlayCount(url):
    c.execute('SELECT playCount FROM
files WHERE strFilename=?', [url])
    result = c.fetchone()
    if result:
        result = result[0]
        if result:
            return int(result)
        return 0
    return -1
```

The above table is an example of a files table.

addSubreddit(): Our plug-in allows users to add their own Subreddit. This function takes the Subreddit input from the user, then saves it in the **subreddits** file inside the addon data folder.

The following sets the subreddits file location:

```
subredditsFile = xbmc.
translatePath("special://profile/
addon_data/"+addonID+"/subreddits")
this translates into .xbmc/userdata/
addon_data/plugin.video.ludent/
subreddits

def addSubreddit():
    keyboard = xbmc.Keyboard('',
translation(30001))
    keyboard.doModal()
    if keyboard.isConfirmed() and
keyboard.getText():
        subreddit = keyboard.
getText()
        fh = open(subredditsFile,
'a')
        fh.write(subreddit+'\n')
        fh.close()
```

This function also demonstrates how to take a text input from the user. Here we are calling the Keyboard function with a text title. Once it detects the keyboard, it writes the input in the subreddits file with a newline character.

getYoutubeUrl(id): When we locate a YouTube URL to play, we pass it on to the YouTube plug-in (plugin.video.youtube) to handle the playback. To do so, we need to call it in a certain format…

```
def getYoutubeUrl(id):
    url = "plugin://plugin.
video.youtube/?path=/root/
video&action=play_video&videoid=" +
id
    return url
```

Similarly for Vimeo:

```
def getVimeoUrl(id):
    url = "plugin://plugin.video.
vimeo/?path=/root/video&action=play_
video&videoid=" + id
    return url
```

And for Dailymotion:

```
def getDailyMotionUrl(id):
    url = "plugin://plugin.video.
dailymotion_com/?url=" + id +
"&mode=playVideo"
    return url
```

Once we have the video URL resolved into the respective plug-in, playing it is very simple:

```
def playVideo(url):
    listitem = xbmcgui.
ListItem(path=url)
    xbmcplugin.
setResolvedUrl(pluginhandle, True,
listitem)
```

13 Populating plug-in content listing

xbmcplugin contains various routines for handling the content listing inside the plug-ins UI. The first step is to create directory entries which can be selected from the XBMC UI. For this we will use a function called **xbmcplugin.addDirectoryItem**.

For our convenience we will be abstracting addDirectoryItem to suit it to our purpose, so that we can set name, URL, mode, icon image and type easily.

```
def addDir(name, url, mode,
iconimage, type=""):
    u = sys.argv[0]+"?url="+urllib.
quote_plus(url)+"&mode="+str(mode)+"
&type="+str(type)
    ok = True
    liz = xbmcgui.ListItem(name,
iconImage="DefaultFolder.png",
thumbnailImage=iconimage)
    liz.setInfo(type="Video",
infoLabels={"Title": name})
    ok = xbmcplugin.
addDirectoryItem(handle=int(sys.
argv[1]), url=u, listitem=liz,
isFolder=True)
    return ok
```

On the same lines, we can build a function to place links as well...

```
def addLink(name, url, mode,
iconimage, description, date):
    u = sys.argv[0]+"?url="+urllib.
quote_plus(url)+"&mode="+str(mode)
    ok = True
    liz = xbmcgui.ListItem(name,
iconImage="DefaultVideo.png",
thumbnailImage=iconimage)
    liz.setInfo(type="Video",
infoLabels={"Title": name, "Plot":
description, "Aired": date})
    liz.setProperty('IsPlayable',
'true')
    ok = xbmcplugin.
addDirectoryItem(handle=int(sys.
argv[1]), url=u, listitem=liz)
    return ok
```

Based on the abstractions we have just created, we can create the base functions which will populate the content. But before we do that, let's first understand how Reddit works. Most of the Reddit content filters are provided through something called Subreddits. This allows you to view discussions related to a particular topic. In our plug-in we are interested in showing videos; we also want to show trailers, documentaries etc. We access these using Subreddits. For example, for trailers it would be **reddit.com/r/trailers**. For domains we can use **/domain**; for example, to get all the YouTube videos posted on Reddit, we will call **reddit.com/domain/youtube.com**. Now you may ask what is the guarantee that this Subreddit will only list videos? The answer is that it may not. For that reason we scrape the site ourselves to find videos. More on this in the next step.

The first base function we'll define is **index()**. This is called when the user starts the plug-in.

```
def index():
    defaultEntries = ["videos",
"trailers", "documentaries",
"music"]
    entries = defaultEntries[:]
    if os.path.
exists(subredditsFile):
        fh = open(subredditsFile,
'r')
        content = fh.read()
        fh.close()
        spl = content.split('\n')
        for i in range(0, len(spl),
1):
            if spl[i]:
                subreddit = spl[i]
                entries.
append(subreddit)
    entries.sort()
    for entry in entries:
        if entry in defaultEntries:
            addDir(entry.title(),
"r/"+entry, 'listSorting', "")
        else:
            addDirR(entry.title(),
"r/"+entry, 'listSorting', "")
    addDir("[ Vimeo.com ]",
"domain/vimeo.com", 'listSorting',
"")
    addDir("[ Youtu.be ]", "domain/
youtu.be", 'listSorting', "")
    addDir("[ Youtube.com
]", "domain/youtube.com",
'listSorting', "")
    addDir("[ Dailymotion.com
]", "domain/dailymotion.com",
'listSorting', "")
    addDir("[B]-
"+translation(30001)+" -[/B]", "",
'addSubreddit', "")
    xbmcplugin.
endOfDirectory(pluginhandle)
```

Here, the penultimate entry makes a call to addSubreddit. **listSorting** takes care of sorting out the data based on criteria such as Hot, New etc. It also calls in Reddit's JSON function, which returns nice easy-to-parse JSON data.

We have created a settings entry for all the sorting criteria. Based on what is set, we go ahead and build out the sorted list.

```
def listSorting(subreddit):
    if cat_hot:
        addDir(translation(30002),
urlMain+"/"+subreddit+"/hot/.
json?limit=100", 'listVideos', "")
    if cat_new:
        addDir(translation(30003),
urlMain+"/"+subreddit+"/new/.
json?limit=100", 'listVideos', "")
    if cat_top_d:

addDir(translation(30004)+":
"+translation(30007),
urlMain+"/"+subreddit+"/
top/.json?limit=100&t=day",
'listVideos', "")
    xbmcplugin.
endOfDirectory(pluginhandle)
```

```
def listVideos(url):
    currentUrl = url
    xbmcplugin.setContent(pluginhandle, "episodes")
    content = opener.open(url).read()
    spl = content.split('"content"')
    for i in range(1, len(spl), 1):
        entry = spl[i]
        try:
            match = re.compile('"title": "(.+?)"', re.DOTALL).findall(entry)
            title = match[0].replace("&amp;", "&")
            match = re.compile('"description": "(.+?)"', re.DOTALL).
findall(entry)
            description = match[0]
            match = re.compile('"created_utc": (.+?),', re.DOTALL).findall(entry)
            downs = int(match[0].replace("}", ""))
            rating = int(ups*100/(ups+downs))
            if filter and (ups+downs) > filterVoteThreshold and rating <
filterRating:
                continue
            title = title+" ("+str(rating)+"%)"
            match = re.compile('"num_comments": (.+?),', re.DOTALL).
findall(entry)
            comments = match[0]
            description = dateTime+"  |  "+str(ups+downs)+" votes:
"+str(rating)+"% Up  |  "+comments+" comments\n"+description
            match = re.compile('"thumbnail_url": "(.+?)"', re.DOTALL).
findall(entry)
            thumb = match[0]
            matchYoutube = re.compile('"url": "http://www.youtube.com/
watch\\?v=(.+?)"', re.DOTALL).findall(entry)
            matchVimeo = re.compile('"url": "http://vimeo.com/(.+?)"',
re.DOTALL).findall(entry)
            url = ""
            if matchYoutube:
                url = getYoutubeUrl(matchYoutube[0])
            elif matchVimeo:
                url = getVimeoUrl(matchVimeo[0].replace("#", ""))
            if url:
                addLink(title, url, 'playVideo', thumb, description, date)
        except:
            pass
    match = re.compile('"after": "(.+?)"', re.DOTALL).findall(entry)
    xbmcplugin.endOfDirectory(pluginhandle)
    if forceViewMode:
        xbmc.executebuiltin('Container.SetViewMode('+viewMode+')')
```

14 Populating the episode view (listing videos)

At this point we have the URL in hand, which returns JSON data; now we need to extract the data out of it which will make sense to us.

By looking at the JSON data, you can see there's a lot of interesting information present here. For example, **url** is set to youtube.com/watch?v=n4rTztvVx8E; **title** is set to 'The Counselor – Official Trailer'. There also many other bits of data that we will use, such as ups, downs, num_comments, thumbnail_url and so on. In order to filter out the data that we need, we will use regular expressions.

There is one more thing to note: since we are not presenting directories any more but are ready to place content, we have to set the **xbmcplugin.setContent** to episodes mode.

In the code listed to the left here, we are opening the URL, then – based on regular expression matches – we are discovering the location title, description, date, ups, downs and rating. We are also locating video thumbnails and then passing them on to XBMC.

Later in the code, we also try to match the URL to a video provider. With our plug-in we are supporting YouTube, Vimeo and Dailymotion. If this is detected successfully, we call the helper functions to locate the XBMC plug-in based playback URL. During this whole parsing process, if any exception is raised, the whole loop is ignored and the next JSON item is parsed.

15 Installing & running the add-on

You can install the add-on using one of the following two methods:

• You can copy the plug-in directory to .xbmc/addons.

• You can install the plug-in from the zip file. To do so, compress the add-on folder into a zip file using the command:

```
$ zip -r plugin.video.ludent.zip
plugin.video.ludent
```

To install the plug-in from the zip file, open XBMC, go to System then Add-ons, then click 'Install from zip file'. The benefit of installing from a zip file is that XBMC will automatically try to install all the dependent plug-ins as well.

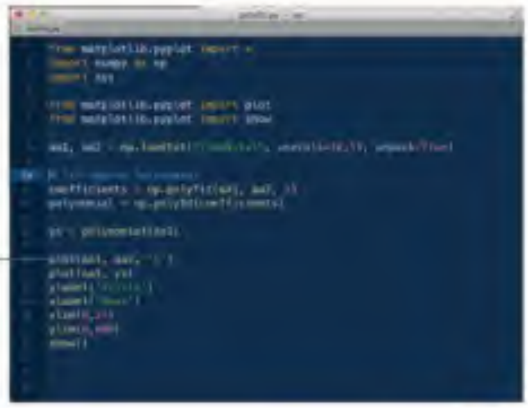
Once you have the plug-in installed, you can run it by going to the Videos Add-ons section of XBMC, selecting Get More… and then clicking on LUD Reddit Viewer.

You can access the settings dialog of the plug-in by right-clicking the LUD Reddit Viewer, then selecting 'Add-on settings'.

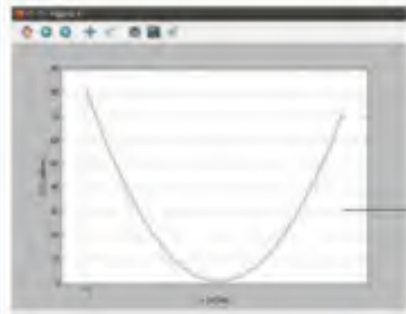
So, you have seen how robust and powerful XBMC's extension system is. In this example, we were able to leverage the full power of Python (including those magical regular expression matches) from within XBMC. XBMC itself also offers a robust UI framework, which provides a very professional look for our add-on.

As powerful as it may seem, we have only built a video plug-in. XBMC's extension system also provides a framework for building fully fledged programs (called Programs). We will cover this in a later issue.

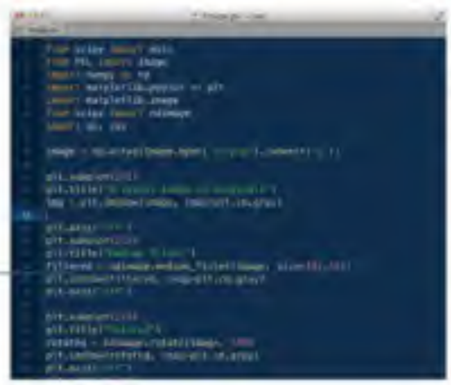
A simple Python program for Polynomial Fitting!



Matplotlib generated output



A Python script that uses SciPy to process an image



Finding help is easy

Scientific computing with NumPy

Powerful calculations with NumPy, SciPy and Matplotlib

Resources

NumPy:
www.numpy.org

SciPy:
www.scipy.org

Matplotlib:
www.matplotlib.org

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 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 are using either Python shell or iPython.

01 Installing NumPy

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' is not 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.

02 About NumPy

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 is by using 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 more dimensions.

03 Making simple calculations using NumPy

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

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

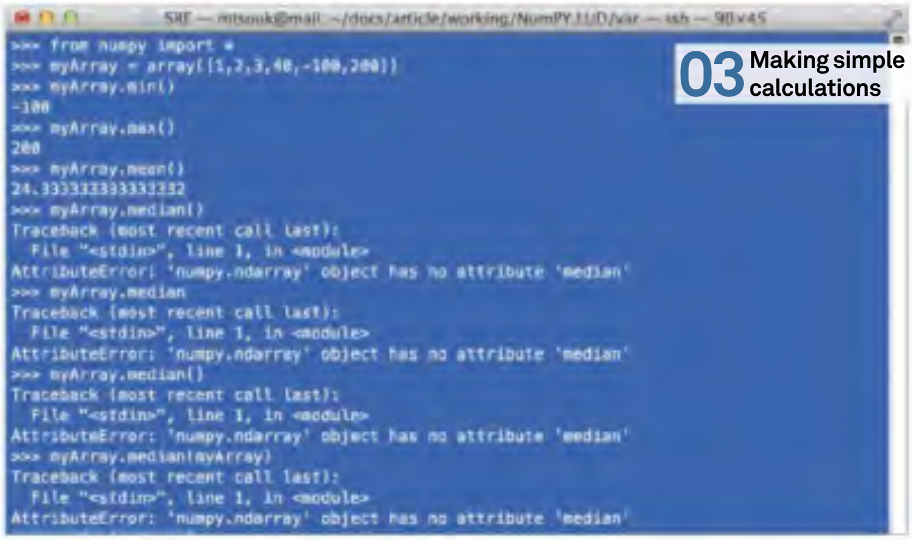
Should you wish to find the mean value of all array elements, run the next command:

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

Similarly, you can find the median of the array by running the following 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.



03 Making simple calculations

04 Using arrays with NumPy

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.

05 Reading files

Imagine that you have just extracted information from an Apache log file using AWK and you want to 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")
```

06 Writing to files

Writing variables to a file is largely similar to reading 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.

07 Common functions

NumPy supports many numerical and statistical functions. When you apply a function to an array, the function is automatically applied to all 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)".

08 Working with matrices

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

09 Plotting with Matplotlib

> **SciPy is built on top of NumPy and is more advanced**

Fig 01

```
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
```

09 Plotting with Matplotlib

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 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.

10 About SciPy

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 in **Fig. 01** (to the left) uses a small part of the scipy.stats package that is about statistics.

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

11 Using SciPy for image processing

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.

12 Other useful functions

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 way:

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

13 Fitting to polynomials

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 in this article.

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:

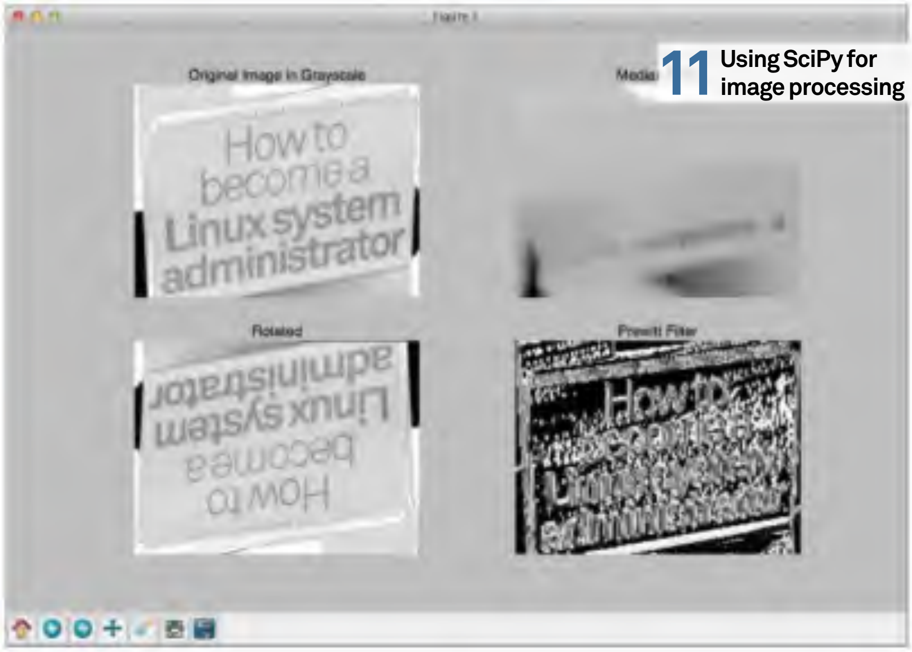
```
coefficients = np.polyfit(aa1, aa2, 5)
```

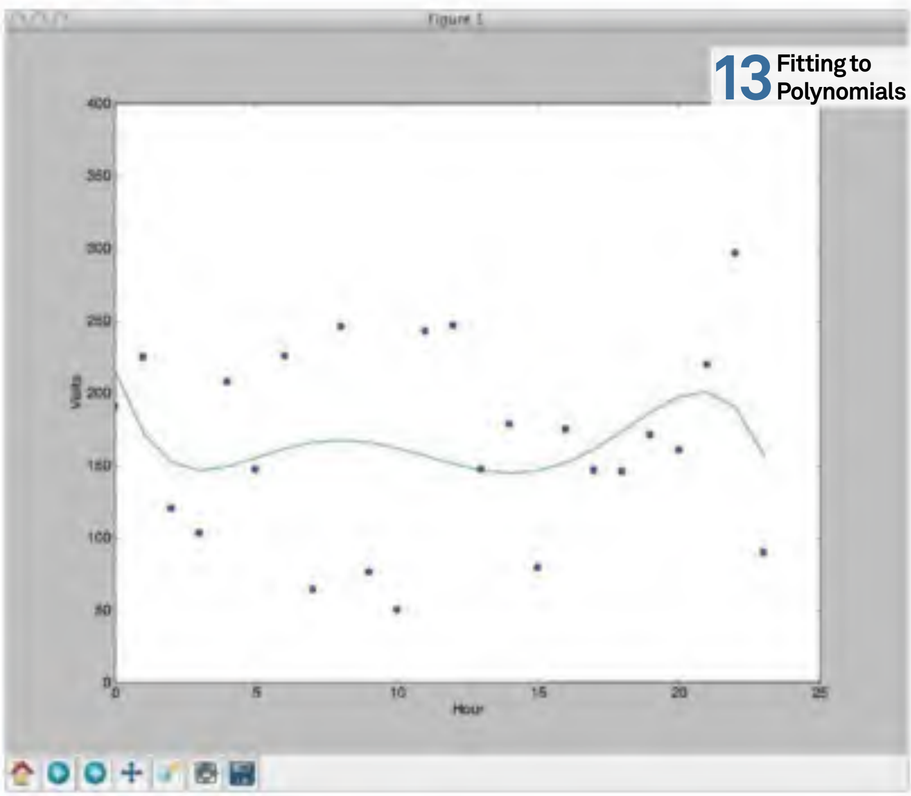
14 Array broadcasting in NumPy

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.

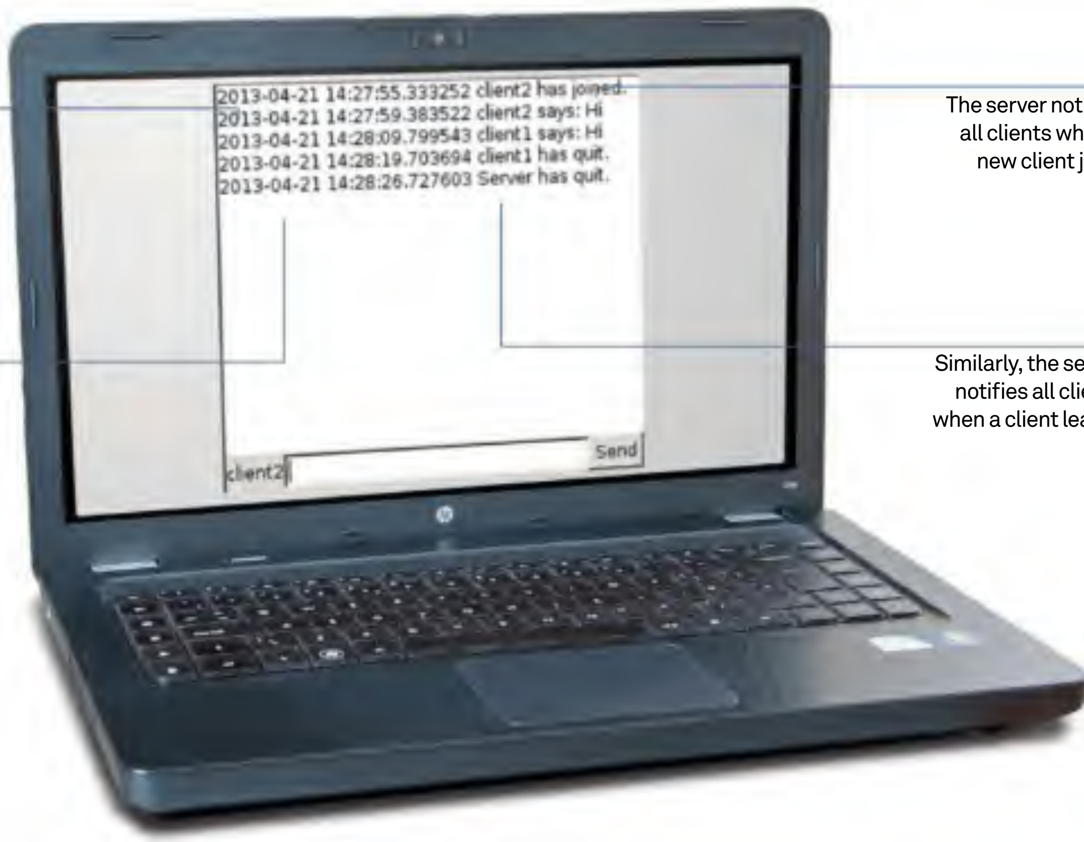
"Process and transform a PNG image using SciPy"





Each message has a time stamp prefixed to it

A client can detect when the server exits without crashing or hanging



The server notifies all clients when a new client joins

Similarly, the server notifies all clients when a client leaves

Instant messaging with Python

How to program both the client, complete with a GUI, and server of a simple instant messenger in Python

Resources

A computer – running your favourite Linux distribution

Internet connection – to access documentation

Python 2.x, PyGTK and GObject – packages installed

Here we'll be implementing an instant messenger in Python with a client-server architecture. This means each client connects to the server, which relays any message that one client sends to all other clients. The server will also notify the other clients when someone joins or leaves the server. The instant messenger can work anywhere a TCP socket can: on the same computer with the loopback interface, across various computers on a LAN, or even over the internet if you were to configure your router correctly. However, our messages aren't encrypted, so we wouldn't recommend that.

Writing an instant messenger is an interesting technical problem that covers a bunch of areas that you may not have come across while programming before:

- We'll be employing sockets, which are used to transmit data across networks.
- We'll also be using threading, which allows a program to do multiple things at once.
- We'll cover the basics of writing a simple graphical user interface with GTK, as well as how to interact with that from a different thread.
- Finally, we'll be touching on the use of regular expressions to easily analyse and extract data from strings.

Before getting started, you'll need to have a Python2.x interpreter installed, as well as the PyGTK bindings and the Python2 GObject bindings. The chances are that if you have a system with a fair amount of software on it, you will already have these packages, so it may be easier to wait and see if you're missing any libraries when you attempt to import them. All of the above packages are commonly used, so you should be able to install them using your distro's package manager.

01 The server

The server will do the following jobs:

- Listen for new clients
- Notify all clients when a new client joins
- Notify all clients when a client leaves
- Receive and deliver messages to all clients

We're going to write the server side of the instant messenger first, as the client requires it. There will be two code files, so it's a good idea to make a folder to keep them inside. You can create an empty file with the command `touch [filename]`, and mark that file as executable using `chmod +x [filename]`. This file is now ready to edit in your favourite editor.

```
[liam@liam-laptop Python]$ mkdir
Python-IM
[liam@liam-laptop Python]$ cd
Python-IM/
[liam@liam-laptop Python-IM]$ touch
IM-Server.py
[liam@liam-laptop Python-IM]$ chmod
+x IM-Server.py
```

02 Starting off

As usual, we need to start off with the line that tells the program loader what it needs to interpret the rest of the file with. In your advisor's case, that line is:

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

On your system, it may need to be changed to `#!/usr/bin/env/ python2.6` or `#!/usr/bin/env python2.7`

After that, we've written a short comment about what the application does, and imported the required libraries. We've already mentioned what the **threading** and **socket** libraries are for. The **re** library is used for searching strings with regular expressions. The **signal** library is used for dealing with signals that will kill the program, such as SIGINT. SIGINT is sent when Ctrl+C is pressed. We handle these signals so that the program can tell the clients that it's exiting rather than dying unexpectedly. The **sys** library is used to exit the program. Finally, the **time** library is used to put a sensible limit on how frequently the body of while loops execute.

```
#!/usr/bin/env python2
# The server side of an instant
messaging application. Written as
part of a Linux User & Developer
tutorial by Liam Fraser in 2013.
import threading
import socket
import re
import signal
import sys
import time
```

03 The Server class

The **Server** class is the main class of our instant messenger server. The **initialiser** of this class accepts a port number to start listening for clients on. It then creates a socket, binds the socket to the specified port on all interfaces, and then starts to listen on that port. You can optionally include an IP address in the tuple that contains the port. Passing in a blank string like we have done causes it to listen on all interfaces. The value of 1 passed to the **listen** function specifies the maximum number of queued connections we can accept. This shouldn't be a problem as we're not expecting a bunch of clients to connect at exactly the same time.

Now that we have a socket, we'll create an empty array that will be later used to store a collection of client sockets that we can echo messages to. The final part is to tell the signal library to run the **self.signal_handler** function, which we have yet to write, when a SIGINT or SIGTERM is sent to the application so that we can tidy up nicely.

```
class Server():
    def __init__(self, port):
# Create a socket and bind it to a
port
        self.listener = socket.
socket(socket.AF_INET, socket.SOCK_
STREAM)
        self.listener.bind((‘’,
port))
        self.listener.listen(1)
        print “Listening on port
{0}”.format(port)
# Used to store all of the client
sockets we have, for echoing
to them
        self.client_sockets = []
# Run the function self.signal_
handler when Ctrl+C is pressed
        signal.signal(signal.SIGINT,
self.signal_handler)
        signal.signal(signal.
SIGTERM, self.signal_handler)
```

04 The server's main loop

The server's main loop essentially accepts new connections from clients, adds that client's socket to the collection of sockets and then starts an instance of the **ClientListener** class, which we have yet to write, in a new thread. Sometimes, defining interfaces you are going to call before you've written them is good, because it can give an overview of how the program will work without worrying about the details.

Note that we're printing information as we go along, to make debugging easier should we need to do it. Sleeping at the end of the loop is useful to make sure the **while** loop can't run quickly enough to hang the machine. However, this is unlikely to happen as the line that accepts new connections is blocking, which means that the program waits for a connection before moving on from that line. For this reason, we need to enclose the line in a **try** block, so that we can catch the socket error and exit when we can no longer accept connections. This will usually be when we've closed the socket during the process of quitting the program.

```
    def run(self):
        while True:
# Listen for clients, and create a
ClientThread for each new client
            print “Listening for
more clients”

            try:
                (client_socket,
client_address) = self.listener.
accept()
            except socket.error:
                sys.exit(“Could not
```

Useful documentation

Threading: docs.python.org/2/library/threading.html

Sockets: docs.python.org/2/library/socket.html

Regular expressions: docs.python.org/2/library/re.html

The signal handler: docs.python.org/2/library/signal.html

PyGTK: www.pygtk.org/pygtk2reference

GObject: www.pygtk.org/pygtk2reference/gobject-functions.html

```
accept any more connections")

            self.client_sockets.
append(client_socket)

            print "Starting client
thread for {0}".format(client_
address)
            client_thread =
ClientListener(self, client_socket,
client_address)
            client_thread.start()

            time.sleep(0.1)
```

05 The echo function

We need a function that can be called from a client's thread to echo a message to each client. This function is pretty simple. The most important part is that sending data to sockets is in a **try** block, which means that we can handle the exception if the operation fails, rather than having the program crash.

```
    def echo(self, data):
# Send a message to each socket in
self.client_socket
        print "echoing: {0}".
format(data)
        for socket in self.client_
sockets:
# Try and echo to all clients
            try:
                socket.sendall(data)
            except socket.error:
                print "Unable to send
message"
```

06 Finishing the Server class

The remainder of the **Server** class is taken up with a couple of simple functions; one to remove a socket from the collection of sockets, which doesn't need an explanation, and the **signal_handler** function that we talked about in the **initialiser** of the class. This function stops listening for new connections, and unbinds the socket from the port it was listening on. Finally, we send a message to each client to let them know that we are exiting. The signal will continue to close the program as expected once the signal_handler function has ended.

```
    def remove_socket(self, socket):
# Remove the specified socket from the
client_sockets list
        self.client_sockets.
remove(socket)
    def signal_handler(self, signal,
frame):
# Run when Ctrl+C is pressed
        print "Tidying up"
# Stop listening for new connections
        self.listener.close()
# Let each client know we are quitting
        self.echo("QUIT")
```

07 The client thread

The class that is used to deal with each client inherits the **Thread** class. This means that the class can be created, then started with `client_thread.start()`. At this point, the code in the **run** function of the class will be run in the background and the main loop of the **Server** class will continue to accept new connections.

We have to start by initialising the Thread base class, using the **super** keyword. You may have noticed that when we created a new instance of the **ClientListener** class in the server's main loop, we passed through the server's **self** variable. We do this because it's better for each instance of the ClientListener class to have its own reference to the server, rather than using the global one that we'll create later to actually start the application.

```
class ClientListener(threading.
Thread):
    def __init__(self, server,
socket, address):
# Initialise the Thread base class
        super(ClientListener,
self).__init__()
# Store the values that have been
passed to the constructor
        self.server = server
        self.address = address
        self.socket = socket
        self.listening = True
        self.username = "No
Username"
```

08 The client thread's loop

The loop that runs in the client thread is pretty similar to the one in the server. It keeps listening for data while **self.listening** is true, and passes any data it gets to a **handle_msg** function that we will write shortly. The value passed to the **socket.recv** function is the size of the buffer to use while receiving data.

```
    def run(self):
# The thread's loop to receive and
process messages
        while self.listening:
            data = ""
            try:
                data = self.socket.
recv(1024)
            except socket.error:
                "Unable to recieve
data"
            self.handle_msg(data)
            time.sleep(0.1)
# The while loop has ended
        print "Ending client thread
for {0}".format(self.address)
```

09 Tidying up

We need to have a function to tidy up the thread. We'll call this either when the client sends us a blank string (indicating that it's stopped listening on the socket) or sends us the string "QUIT". When this happens, we'll echo to every client that the user has quit.

```
    def quit(self):
# Tidy up and end the thread
        self.listening = False
        self.socket.close()
        self.server.remove_
socket(self.socket)
        self.server.echo("{0} has
quit.\n".format(self.username))
```

10 Handling messages

There are three possible messages our clients can send:

- QUIT
- USERNAME user
- Arbitrary string to be echoed to all clients

The client will also send a bunch of empty messages if the socket has been closed, so we will end their thread if that happens. The code should be pretty self-explanatory apart from the regular expression part. If someone sends the USERNAME message, then the server tells every client that a new user has joined. This is tested with a regular expression. **^** indicates the start of the string, **$** indicates the end, and the brackets containing **.*** extract whatever comes after "USERNAME ".

> **“We need to tell GObject that we’ll be using threading”**

```
    def handle_msg(self, data):
# Print and then process the message
we’ve just recieved
        print "{0} sent: {1}".
format(self.address, data)
# Use regular expressions to test for
a message like "USERNAME liam"
        username_result =
re.search('^USERNAME (.*)$', data)
        if username_result:
            self.username =
username_result.group(1)
            self.server.echo("{0}
has joined.\n".format(self.
username))
        elif data == "QUIT":
# If the client has sent quit then
close this thread
            self.quit()
        elif data == "":
# The socket at the other end is
probably closed
            self.quit()
        else:
# It’s a normal message so echo it to
everyone
            self.server.echo(data)
```

11 Starting the server

The code that actually starts the **Server** class is as follows. Note that you are probably best picking a high-numbered port as you need to be root to open ports <1024.

```
if __name__ == "__main__":
    # Start a server on port 59091
    server = Server(59091)
    server.run()
```

12 The client

Create a new file for the client as we did for the server and open it in your favourite editor. The client requires the same imports as the server, as well as the gtk, gobject and datetime libraries. One important thing we need to do is to tell GObject that we’ll be using threading, so we can call functions from other threads and have the main window, which is running in the main GTK thread, update.

```
#!/usr/bin/env python2
# The client side of an instant
messaging application. Written as
part of a Linux User & Developer
tutorial by Liam Fraser in 2013.

import threading
import gtk
import gobject
import socket
import re
import time
import datetime

# Tell gobject to expect calls from
multiple threads
gobject.threads_init()
```

13 The client graphical user interface

The user interface of the client isn’t the main focus of the tutorial, and won’t be explained in as much detail as the rest of the code. However, the code should be fairly straightforward to read and we have provided links to documentation that will help.

Our **MainWindow** class inherits the **gtk Window** class, so we need to start by initialising that using the **super** keyword. Then we create the controls that will go on the window, connect any events they have to functions, and finally lay out the controls how we want. The **destroy** event is raised when the program is closed, and the other events should be obvious.

GTK uses a packing layout, in which you use Vboxes and Hboxes to lay out the controls. V and H stand for vertical and horizontal. These controls essentially let you split a window up almost like a table, and will automatically decide the size of the controls depending on the size of the application.

GTK doesn’t come with a control to enter basic information, such as the server’s IP address, port and your chosen username, so we’ve made a function called **ask_for_info**, which creates a message box, adds a text box to it and then retrieves the results. We’ve done this because it’s simpler and uses less code than creating a new window to accept the information.

```
class MainWindow(gtk.Window):
    def __init__(self):
# Initialise base gtk window class
        super(MainWindow, self).__
init__()
# Create controls
        self.set_title("IM Client")
        vbox = gtk.VBox()
        hbox = gtk.HBox()
        self.username_label = gtk.
Label()
        self.text_entry = gtk.
Entry()
        send_button = gtk.
Button("Send")
        self.text_buffer = gtk.
TextBuffer()
        text_view = gtk.
TextView(self.text_buffer)
# Connect events
        self.connect("destroy",
self.graceful_quit)
        send_button.
connect("clicked", self.send_
message)
# Activate event when user presses
Enter
        self.text_entry.
connect("activate", self.send_
message)
# Do layout
        vbox.pack_start(text_view)
        hbox.pack_start(self.
username_label, expand = False)
```

»

```
        hbox.pack_start(self.text_
entry)
        hbox.pack_end(send_button,
expand = False)
        vbox.pack_end(hbox, expand
= False)
# Show ourselves
        self.add(vbox)
        self.show_all()
# Go through the configuration
process
        self.configure()
    def ask_for_info(self,
question):
# Shows a message box with a text
entry and returns the response
        dialog = gtk.
MessageDialog(parent = self, type =
gtk.MESSAGE_QUESTION,

flags = gtk.DIALOG_MODAL |

gtk.DIALOG_DESTROY_WITH_PARENT,

buttons = gtk.BUTTONS_OK_CANCEL,

message_format = question)
        entry = gtk.Entry()
        entry.show()
        dialog.vbox.pack_end(entry)
        response = dialog.run()
        response_text = entry.
get_text()
        dialog.destroy()
        if response == gtk.RESPONSE_
OK:
            return response_text
        else:
            return None
```

14 Configuring the client

This code is run after we've added the controls to the main window, and asks the user for input. Currently, the application will exit if the user enters an incorrect server address or port; but this isn't a production system, so that's fine.

```
    def configure(self):
# Performs the steps to connect to
the server
# Show a dialog box asking for server
address followed by a port
        server = self.ask_for_
info("server_address:port")
# Regex that crudely matches an IP
address and a port number
        regex = re.search('^(\d+\.\
d+\.\d+\.\d+):(\d+)$', server)
        address = regex.group(1).
strip()
        port = regex.group(2).
strip()
# Ask for a username
        self.username = self.ask_
for_info("username")
        self.username_label.set_
text(self.username)
# Attempt to connect to the server
and then start listening
        self.network =
Networking(self, self.username,
address, int(port))
        self.network.listen()
```

15 The remainder of MainWindow

The rest of the MainWindow class has plenty of comments to explain itself, as follows. One thing to note is that when a client sends a message, it doesn't display it in the text view straight away. The server is going to echo the message to each client, so the client simply displays its own message when the server echoes it back. This means that you can tell if the server is not receiving your messages when you don't see a message that you send.

```
    def add_text(self, new_text):
# Add text to the text view
        text_with_timestamp = "{0}
{1}".format(datetime.datetime.now(),

new_text)
# Get the position of the end of
the text buffer, so we know where to
insert new text
        end_itr = self.text_buffer.
get_end_iter()
# Add new text at the end of the buffer
        self.text_buffer.insert(end_
itr, text_with_timestamp)
    def send_message(self, widget):
# Clear the text entry and send the
message to the server
# We don't need to display it as it
will be echoed back to each client,
including us.
        new_text = self.text_entry.
get_text()
        self.text_entry.set_text("")
        message = "{0} says: {1}\n".
format(self.username, new_text)
        self.network.send(message)
    def graceful_quit(self, widget):
# When the application is closed,
tell GTK to quit, then tell the
server we are quitting and tidy up
the network
        gtk.main_quit()
        self.network.send("QUIT")
        self.network.tidy_up()
```

> **"The server is going to echo the message to each client"**

16 The client's Networking class

Much of the client's **Networking** class is similar to that of the server's. One difference is that the class doesn't inherit the **Thread** class – we just start one of its functions as a thread.

```
class Networking():
    def __init__(self, window,
username, server, port):
# Set up the networking class
        self.window = window
        self.socket = socket.
socket(socket.AF_INET, socket.SOCK_
STREAM)
        self.socket.connect((server,
port))
        self.listening = True
 # Tell the server that a new user
has joined
        self.send("USERNAME {0}".
format(username))

    def listener(self):
# A function run as a thread that
listens for new messages
        while self.listening:
            data = ""
            try:
                data = self.socket.
recv(1024)
            except socket.error:
                "Unable to recieve
data"
            self.handle_msg(data)
# Don't need the while loop to be
ridiculously fast
            time.sleep(0.1)
```

17 Running a function as a thread

The listener function above will be run as a thread. This is trivial to do. Enabling the **daemon** option on the thread means that it will die if the main thread unexpectedly ends.

```
    def listen(self):
# Start the listening thread
        self.listen_thread =
threading.Thread(target=self.
listener)
# Stop the child thread from keeping
the application open
        self.listen_thread.daemon =
True
        self.listen_thread.start()
```

18 Finishing the Networking class

Again, most of this code is similar to the code in the server's Networking class. One difference is that we want to add some things to the text view of our window. We do this by using the **idle_add** function of GObject. This allows us to call a function that will update the window running in the main thread when it is not busy.

```
    def send(self, message):
# Send a message to the server
        print "Sending: {0}".
format(message)
        try:
            self.socket.
sendall(message)
        except socket.error:
            print "Unable to send
message"

    def tidy_up(self):
# We'll be tidying up if either we are
quitting or the server is quitting
        self.listening = False
        self.socket.close()
# We won't see this if it's us
that's quitting as the window will
be gone shortly
        gobject.idle_add(self.
window.add_text, "Server has
quit.\n")

    def handle_msg(self, data):
        if data == "QUIT":
# Server is quitting
            self.tidy_up()
        elif data == "":
# Server has probably closed
unexpectedly
            self.tidy_up()
        else:
# Tell the GTK thread to add some
text when it's ready
            gobject.idle_add(self.
window.add_text, data)
```

19 Starting the client

The main window is started by initialising an instance of the class. Notice that we don't need to store anything that is returned. We then start the GTK thread by calling **gtk.main()**.

```
if __name__ == "__main__":
# Create an instance of the main
window and start the gtk main loop
    MainWindow()
    gtk.main()
```

20 Trying it out

You'll want a few terminals: one to start the server, and some to run clients. Once you've started the server, open an instance of the client and enter `127.0.0.1:port`, where 'port' is the port you decided to use. The server will print the port it's listening on to make this easy. Then enter a username and click OK. Here is an example output from the server with two clients. You can use the client over a network by replacing 127.0.0.1 with the IP address of the server. You may have to let the port through your computer's firewall if it's not working.

```
[liam@liam-laptop Python]$ ./IM-
Server.py
Listening on port 59091
Listening for more clients
Starting client thread for
('127.0.0.1', 38726)
('127.0.0.1', 38726) sent: USERNAME
client1
echoing: client1 has joined.
Listening for more clients
Starting client thread for
('127.0.0.1', 38739)
('127.0.0.1', 38739) sent: USERNAME
client2
echoing: client2 has joined.
Listening for more clients
('127.0.0.1', 38739) sent: client2
says: Hi
echoing: client2 says: Hi
('127.0.0.1', 38726) sent: client1
says: Hi
echoing: client1 says: Hi
('127.0.0.1', 38726) sent: QUIT
echoing: client1 has quit.
Ending client thread for
('127.0.0.1', 38726)
^CTidying up
echoing: QUIT
Could not accept any more
connections
('127.0.0.1', 38739) sent:
echoing: client2 has quit.
Ending client thread for
('127.0.0.1', 38739)
```

21 That's it!

So, it's not perfect and could be a little more robust in terms of error handling, but we have a working instant messenger server that can accept multiple clients and relay messages between them. More importantly, we have learned a bunch of new concepts and methods of working.

Replace your shell with Python

Python is a great programming language, but did you know that it is even capable of replacing your primary shell (command-line interface)? Here, we explain all…

Resources

You will require a version of Python installed on your system. The good news is you don't have to do anything to get it installed. Most of the Linux distributions already ship with either Python 2.6 or Python 2.7

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, 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
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'
```

> You can easily write aliases to the equivalent of the Bash command that you intend to replace

2. Interfacing operating system & subprocesses

Python provides two modules to interface with the OS and to manage processes, called os and subprocess. These modules allow you to interact with the core operating system shell and let you work with the environment, processes, users and file descriptors.

The subprocess module was introduced to support better management of subprocesses (part of which already exists in the os module) 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/pgknm8b118
737mb8psz8x4z80000gn/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
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
```

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 you have learnt in section 1 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.
```

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/
```

“IPython also comes with its own Qt-based console”

```
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 preceding %.

To view a list of available magic functions, you can use ‘magic function %lsmagic’. Magic functions include functions that work with code such as %run, %edit, %macro, %recall etc; functions that affect shell such as %colors, %xmode, %autoindent etc; and other functions such as %reset, %timeit, %paste etc. Most of the 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. %edit will open the given Python script in the editor defined by the $EDITOR environment variable.

Shell command support: If you are in the mood to just run a shell command, you can do it very easily by prefixing the command with ! .

Example :

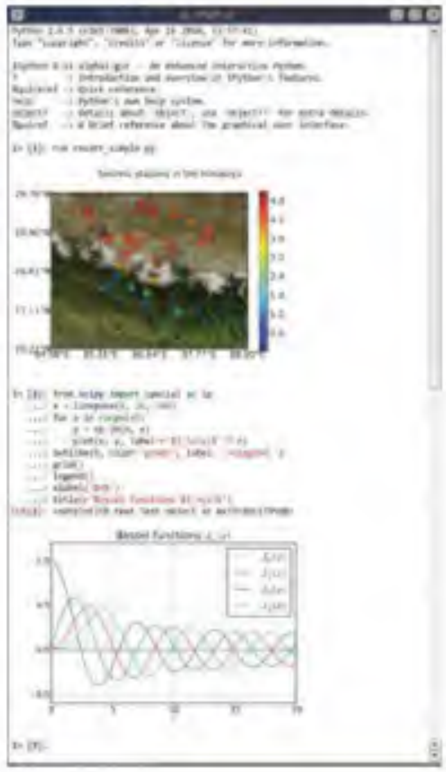
```
In [5]: !ps
  PID TTY           TIME CMD
 4508 ttys000    0:00.07 -bash
84275 ttys001    0:00.03 -bash
17958 ttys002    0:00.18 -bash

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 also comes with its own Qt-based console. This provides a number of features that are only available in a GUI, such as inline figures, multiline editing with syntax highlighting, and graphical calltips .

You can start the Qt console with:

```
$ ipython qtconsole
```

If you get errors related to missing modules, make sure that you have installed the dependent packages, such as PyQt, pygments, pyexpect and ZeroMQ.



IPython Qt console with GUI capabilities

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?

Python for system administrators

Learn how Python can help in system administration as it dares to replace the usual shell scripting...

Resources

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

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.
```

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).

```
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.

■ Generating the Yahoo App ID

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 above code 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 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 &amp;amp; Developer |
Linux User http://www.linuxuser.
co.uk/tag/linux-user-developer/
```

Gathering system information

One of the important jobs of a system administrator is gathering system information. In this section 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 it can provide lot of information, including the following:**

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
```

"Python doesn't limit your choice of interface"

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.

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

Note

If you are confused with the tab spacing of the code, look for the code files on FileSilo.

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:b
9:bb:69:64: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 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 easily be adopted into your GUI application (with software such as PyGTK or PyQt) or a web application (using a framework such as Django or Grok).

Writing a user interface using Python

Learn how to create a user-friendly 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
```

Multitask with your Pi

Learn how to add multitasking to your own Python code – perfect for those with multiple projects on the go

The majority of programmers will learn single-threaded programming as their first computational model. The basic idea is that instructions for the computer are processed sequentially, one after the other. This works well enough in most situations, but you will reach a point where you need to start multitasking. The classical situation for writing multi-threaded applications is to have them run on a multi-processor machine of some persuasion. In these cases, you would have some heavy, compute-bound process running on each processor. Since your Raspberry Pi is not a huge 16-core desktop machine, you might be under the assumption that you can't take advantage of using multiple threads of execution. This isn't true, though.

There are lots of problems that map naturally to the multiple thread model. You may also have IO operations that take a relatively large amount of time to complete. In these cases, it is well worth your programming effort to break your problem down into a multi-threaded model. Since Python is the language of choice for the Raspberry Pi, we will look at how you can add threads to your own Python code. For those of you who have looked into multi-threaded programming in Python, you may have run into the GIL (Global Interpreter Lock) before. This lock means that only one thread can actually be running at a time, so you don't get true parallel processing. But on the Raspberry Pi, this is actually okay.

The first bit of code we need is to import the correct module. For this article, we will be using the threading module. Once it is imported, you have access to all of the functions and objects that you would need to write your code. The first step is to create a new thread object with the constructor:

```
t = threading.Thread(target=my_func)
```

The Thread object takes some function that you have created, my_func in the above example, as the target code that needs to be run. When the thread object has finished its initialisation, it is alive but not running. You need to explicitly call the new thread's start() method. This will begin running the code within the function handed to the thread.

You can check to verify that this thread is alive and active by calling its is_alive() method. Normally, this new thread will run until the function exits normally. The other way a thread can exit is if an unhandled exception is raised. Depending on your experience of parallel programs, you may already have some ideas on what types of code you want to write. For example, in MPI programs, you typically have the same overall code running in multiple threads of execution. You use the thread's ID and a series of if or case statements to have each thread execute a different section of the code. To do something similar, you can use something like:

```
def my_func():
    id = threading.get_ident()
    if (id == 1):
       do_something()
  thread1 = threading.
Thread(target=my_func)thread1.
start()
```

This code works in Python 3, but the get_ident() function doesn't exist in Python 2. Threading is one of those modules that is a moving target when moving from one version of Python to another, so always check the documentation for the version of Python your are coding for.

Another common task in parallel programming is to farm out time-intensive IO into separate threads. This way, your main program can continue on with the core work and all of the computing resources are kept as busy as possible. But how do you figure out if the child thread is done yet or not? You can use the is_alive() function mentioned above, but what if you can't continue without the results from the child thread? In these cases, you can use the join() method of the thread object you are waiting on. This method blocks until the thread in question returns. You can include an optional parameter to have the method time-out after some number of seconds. This allows you to not get trapped into a thread that will never return due to some error or code bug.

Now that we have more than one thread of execution happening at the same time, we have a new set of issues to start worrying about. The first is accessing global data elements. What might happen if you have two different threads that want to read, or even worse write, to the same variable in global memory? You can have situations where changes to the value of variables can get out of sync with what you were expecting them to be.

These types of issues are called race conditions, because the different threads are racing with each other to see in what order their updates to variables will happen. There are two solutions to this type of problem. The first is to control access to these global variables and only allow one thread at a time to be able to work with them. The generic term describing this control is to use a mutex to control this access. A mutex is an object that a thread needs to lock before working with the associated variables. In the Python threading module, this object is called a lock. The first step is to create a new Lock object with:

```
lock = threading.Lock()
```

This new lock is created in an unlocked state, ready to be used. The thread interested in using it must call the acquire() method for the lock. If the lock is currently available then it changes state to the locked state and your thread can run the code that is meant to be protected. If the lock is currently in a locked state, then your thread will sit in a blocked state, waiting for the lock to become free. Once you are done with the protected code, you need to call the release() method to free the lock and make it available for the next thread. You could control a variable containing the sum of a series of results with:

Threads or processes?

What if you need to actually have truly parallel code, that has the ability to run on multiple cores? Because Python has the GIL, you need to move away from using threads and go to using separate processes to handle the different tasks. Luckily, Python includes a multiprocessing module that provides the process equivalent to the threading module. As with the threading module, you create a new process object and hand in a target function to be run. You then need to call the start() method to get it running. With threads, sharing data is trivial because memory is global and everybody can see everything.

However, different processes are in different memory scopes. In order to share data, we need to explicitly set up some form of communications. You can create a queue object where you can transfer objects. Processes can use the put() method to dump objects on the queue, and other processes can use the get() method to pull objects off. If you want a bit more control over who is talking to who, you can use pipes to create a two-way communication channel between two processes. When you use pipes and queues, you actually need to hand them in as arguments to your target function.

The other way you can share information is by creating a section of shared memory. You can create a single variable sharelocation with the Value object. If you have a number of variables you need to pass, you can put them in an Array object. As with pipes and queues, you will also need to pass them in as parameters to your target function. When you need to wait for the results from a process, you can use the join() method to get the main process to block until the sub-process finally finishes.

The processing module also includes the idea of a process pool that is different from the threading module. With a pool, you can pre-create a number of processes that can be used in a map function. This kind of construct is extremely useful if you are then going to be applying the same function to a number of different input values. For people who are actually using the concepts of mapping or applying functions from R, or Hadoop, this might turn out to be a bit more of an intuitive and beneficial model to use in your Python code.

```
lock.acquire()
sum_var += curr_val
lock.release()
```

This can lead to another common issue in parallel programs: deadlocks. These issues occur when you have multiple locks that are associated with different global variables. Say you have the variables A and B, and the associated locks lockA and lockB. If thread 1 tries to get lockA then lockB, while thread 2 tries to get lockB then lockA, you could have the situation where they each get their first requested lock, and then wait forever for the second requested lock.

The best way to avoid this type of bug is to code your program very carefully. Unfortunately, people are only human and messy code can creep in. You can try and catch this kind of bad behaviour by including the optional timeout parameter when you call the acquire() method. This tells the lock to only try and get the lock for some number of seconds. If the timeout is reached, the acquire method returns. You can tell whether or not it was successful by checking the returned value. If it was successful, acquire will return True. Otherwise, it will return False.

The second way you can deal with data access is by moving any variables that you can to within the local scope of the individual threads. The essential idea is that each thread would have its own local version of any required variables that nobody else can see. This is done by creating a local object. You can then add attributes to this local object and use them as local variables. Within the function being run by your thread, you would then have code that looks like:

```
my_local = threading.local()
my_local.x = 42
```

The last topic we will look at is synchronising your threads so that they can work together effectively. Bear in mind there will be certain times when a number of threads will need to talk to each other after working on their separate parts of a particular problem. The only way they can share their results is if they have all finished calculating their individual results. You can solve this problem by using a barrier, which each thread will stop at until all of the other threads have reached it. In Python 3, there is a barrier object that can be created for some number of threads. It will provide a point where threads will pause when they call the barrier's wait() method.

Because you actually need to explicitly tell the barrier object how many threads will be taking part in the barrier, this is another area where it is possible that you could actually be faced with the problem of having a bug. If you create five threads but create a barrier for ten threads, it will never actually be able to reach the point where all of the expected threads have reached the barrier. The other synchronisation tool is the timer object. A timer is a subclass of the thread class, and so takes a function to run after some amount of time has passed. As with a thread, you will need to call the timer's start() method in order to start the countdown to when the function gets executed. A new method, cancel(), will actually allow you to stop the countdown of the timer if it hasn't reached zero yet.

After following all of these steps you should now be able to have your code running even more efficiently by simply farming out any time intensive parts to other threads of execution. By following these steps in this way, the end result is that the main part of your program can remain as efficient and reactive as possible to interaction with the end user and you can also keep all parts of your Raspberry Pi as busy as possible.

> **“Bear in mind there will be certain times when a number of threads will need to talk to each other after working on their seperate parts of a particular problem”**

Use Pi to monitor audio

Since a Raspberry Pi is so compact, it can be used to power monitoring hardware. Learn how to use it for audio tasks

With such a small physical footprint and a low power requirement, the Raspberry Pi is a perfect platform that you can use to build your own scientific equipment. We will look at how you might be able to use your Pi to monitor and analyse sounds in your environment. This is useful if you are listening for particular sounds. You need a Raspberry Pi, some kind of USB microphone, and some kind of USB wireless connection if you want to check on the monitoring process remotely. The specifics of the hardware are up to you, but you should be able to use almost anything that is available to you. This article will focus on the Python code that you will need in order to record this audio and do some processing on it. Let's assume that you are using a Debian-based distribution on your Raspberry Pi, such as Raspbian, for the installation instructions below.

The first step is to make your microphone available to Python. PortAudio is a cross-platform library that can handle playing and recording audio on many different machines. PyAudio is the Python module that provides a wrapper around the PortAudio library. Luckily, both are available in the Raspbian package library. You can install them with the command

```
sudo apt-get install python-pyaudio
```

The module you need to import is called 'pyaudio'. The module consists of two key objects, PyAudio and Stream. PyAudio is the object that initialises the PortAudio library and allows you to start interacting with the audio devices on your Raspberry Pi. The boilerplate code to start your program would look like this:

```
import pyaudio
p = pyaudio.PyAudio()
```

Now that you have an instantiated PyAudio object, you can open your audio device and start recording from it. There are several parameters to the open function that control recording options like the sampling rate, the number of channels, the audio format, and the size of a temporary buffer. Since you will likely need these values in the processing step of your program, you will want to store them in meaningful variables. For example:

```
CHUNK = 1024
FORMAT = pyaudio.paInt16
CHANNELS = 2
RATE = 44100
```

The open function would look like this:

```
stream = p.open(format=FORMAT,
       channels=CHANNELS,
       rate=RATE, input=True,
       frames_per_buffer=CHUNK)
```

By default, this function call will try and open the default audio device. If you only have one plugged in, it should do the as you expect and open that particular device. But, what do you do if there are more than one microphone plugged in? You need to add an extra parameter, named 'input_device_index', that selects the device of interest. Unfortunately, the device index that PyAudio uses is kind of arbitrary. You can use the following code to get a list of the devices and their related indices:

```
for i in range(p.get_device_
count()):
ev = p.get_device_info_by_index(i)
print((i,dev['name'],dev['max
   InputChannels']))
```

Remote monitor access?

By using Raspberry Pi and building your own environmental monitor you can operate independently and away from any available amenities. This is fine if you are only interested in recording and storing the data that you are collecting to look at it later in an offline fashion. But what if you want to check in on your monitor and see what it is actually measuring? In this case, you should be able to connect some type of USB device that allows it to connect to the internet. This could be as simple as a Wi-Fi dongle, if you have a Wi-Fi network nearby, or something that operates over the cellular networks for more remote access. Once you have your Raspberry Pi connected to the internet, you can use IPython to provide a read-out of the monitor's activity. You will want to install:

```
sudo apt-get install ipython-notebook
```

This gets the notebook portion of IPython. You can start up the notebook with the command:

```
ipython notebook --ip=* --no-browser
```

This will then tell IPython to accept connections from anyone, and to not open a browser on the Raspberry Pi. IPython dumps information on the console where it got started, so you probably want to run it under tmux or screen. This way, IPython can continue to print output to the console while you are disconnected. If you are going to run any of the examples in the main article, then you probably will want to add the option '--pylab' to the ipython command. This tells IPython to pre-load the matplotlib and numpy modules.

Once this is done you can then connect to the IPython instance by opening the URL http://my.rpi.ip:8888 in a web browser on your local machine, where 'my.rpi.ip' is the IP address for your Raspberry Pi. If you have a working script, you can create a new notebook and import the code from it into the new notebook. Once it is running as required, you will then be able to disconnect and reconnect again and check in on the progress. There are also modules available that will actually help you to turn your IPython notebook into a proper dashboard. By doing it this way you will be able to create a fully-featured monitoring application on your Raspberry Pi.

From this output, you can find out what index value you should use in the open function call.
With an open stream connected to a microphone, you can start to read data. Since you defined the temporary buffer to be of size 'CHUNK', you use the same value in the read function from the stream. A convenient way to store this incoming data is to append them to a list. An example loop that you might want to use would look like:

```
frames = []
for i in range(0, END_TIME):
data = stream.read(CHUNK)
frames.append(data)
```

The recorded audio now exists in the frames list, ready to be processed. If you need to keep copies of this data, you can dump it into wave files. The wave Python module lets you work with files in WAV format. You can write out the data with :

```
import wave
wf = wave.open(WAVE_OUTPUT_
        FILENAME, 'wb')
wf.setnchannels(CHANNELS)
wf.setsampwidth(p.get_sample_
        size(FORMAT))
wf.setframerate(RATE)
wf.writeframes(b''.join(frames))
wf.close()
```

Where the filename you want to use is stored in the variable WAVE_OUTPUT_FILENAME. When the audio recording portion of your program is done, you need to clean up after yourself. You need to stop and shut down the stream first, then close down the audio device you were using. This can be done with:

```
stream.stop_stream()
stream.close()
p.close()
```

Once you have real data coming in, how do you process it? The easiest thing to do is to simply plot it as a time series of the amplitude of the sound in the environment. The matplotlib module is the go-to Python package to handle plotting and graphing of all kinds. It is a rather large module, and you will need to import whichever sub-module you need for the type of plotting you want to do. For most standard plots, you should be fine simply importing the following:

```
import matplotlib.pyplot as plt
```

You can then use 'plt' as the prefix for all of the plotting functions. If you just want to create a scatter plot where the 'x' values are the index of the data list, you can use:

```
plt.plot(DATA)
plt.show()
```

Here, the variable DATA has all of the sound information that you are interested in. This may be the data collected from one sampling, or a combination of many scans.
While the amplitude is one interesting quality of sound, a lot of information is lost when only looking at these values. Something that is usually of much more interest is the frequencies that exist in the sound sample. While amplitude can change quite dramatically from one scan to another, the frequencies being generated by whoever or whatever is making the sound changes very little.

So you can, theoretically, identify sources based on the spectrum of frequencies you measure. Mathematically, you can use the Fast Fourier Transform (FFT) to extract the frequencies that go into generating the sound you recorded. In Python, there are basic FFT functions in the numpy module, and more complicated sine and cosine transforms in the scipy module. Both of these modules are huge; you can import the sections you simply by using:

```
import numpy.fft as npfft
```

... or...

```
import scipy.fftpack as spfft
```

You can plot these frequency spectra with:

```
import numpy as np
import matplotlib plt
import scipy.fftpack as spfft
yf = spfft.fft(DATA)
plt.plot(yf)
plt.show()
```

Here, the variable DATA contains whatever number of samples you are processing this run. This will display what the major frequencies are in your recorded sound. Once you have all of this data generated, you can then start to make comparisons with known sound sources and potentially make identifications within the specific environment that you are monitoring.

Since all of the calls we have used above are blocking, we run the risk of getting stuck in one portion or another. To try and alleviate possible issues, you may want to look at using threads. While we are still stuck working with the GIL, it may still allow you to do more intensive processing while you are waiting for the next batch of sound data to come in. You can create a new thread with this code:

```
import thread
thread.start_new_thread(my_func,
(arg1, arg2, arg3, )
```

Here the function 'my_func' either handles the sound recording, or the data processing functionality. Since you are going to want to pass data in and out of these functions, an easy way to handle this is with a queue object. Setting one up is as simple as:

```
import Queue
queue1 = Queue.Queue(SIZE)
```

This will then create a queue that can hold SIZE elements in it. If you hand this in as one of the parameters to your thread functions, you can use it to pass data back and forth. You just need to also create and use locks to control access to the queue to be sure only one thread is accessing it at a time. Locks can be created by using the following function call:

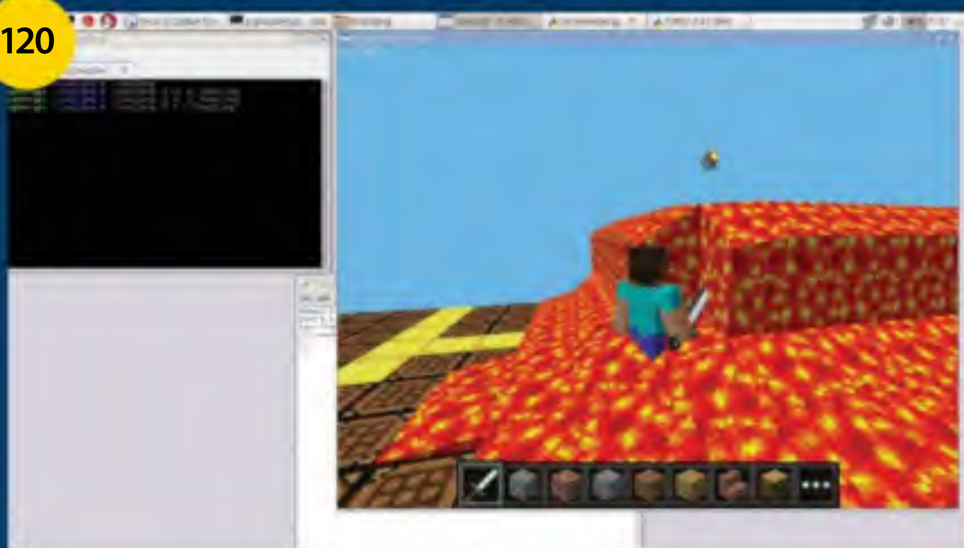
```
lock1 = thread.allocate_lock()
```

After completing these steps you will then be able to go on to use the lock methods 'acquire()' and 'release()' to manage access to the queue. After doing this you should then be ready to take the next step and actually build your own monitoring hardware for a variety of all different applications.

“You can identify sources based on their frequencies”

Create with Python

120



104

"You'll be surprised by the diversity of what you can make with Python"

108

112

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_instal via

```
easy_install kivy.
```

> **"You can develop cross-platform apps using various Python libraries"**

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':

```
def build(self):
```

The classic 'Hello World!' in Kivy GUI form, using the built-in Label widget

```
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.

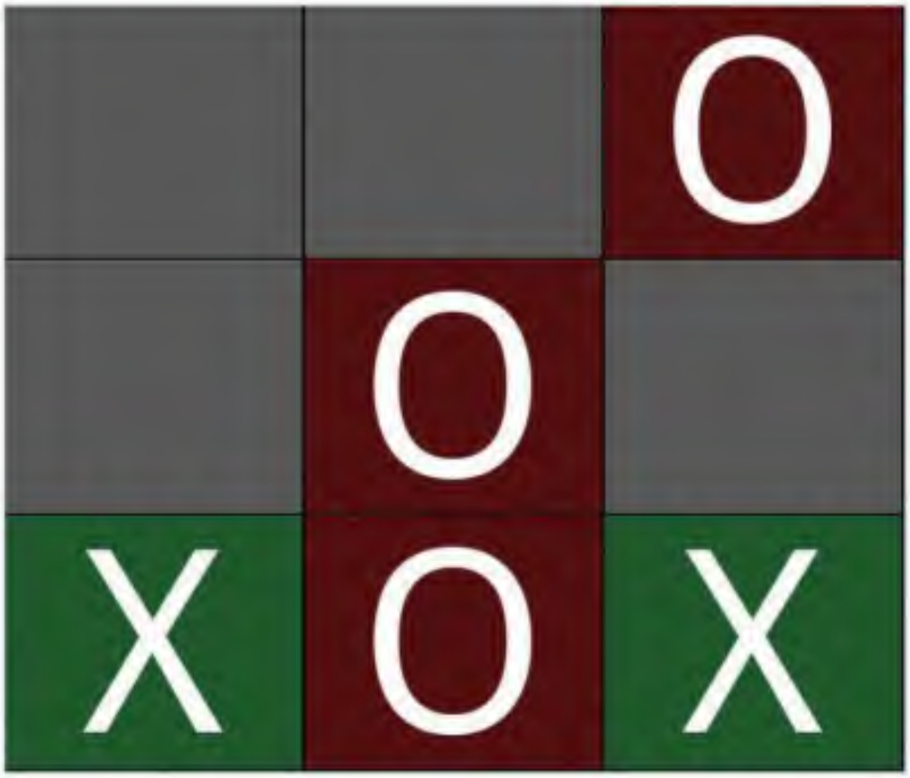
That's all you need to show some simple text, so run the program again to check that this does work. You can experiment with the parameters if it's unclear what any of them are doing.

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,



A tic-tac-toe grid now accepting input, adding a O or X alternately

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  # Number of columns
```

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

"Kivy comes with all the tools needed to use kv language"

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 **(Fig.01)**. 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, which is dispatched dynamically whenever 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.'

```
def build(self):
    return TicTacToeGrid()
    # Replaces the previous label
```

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.

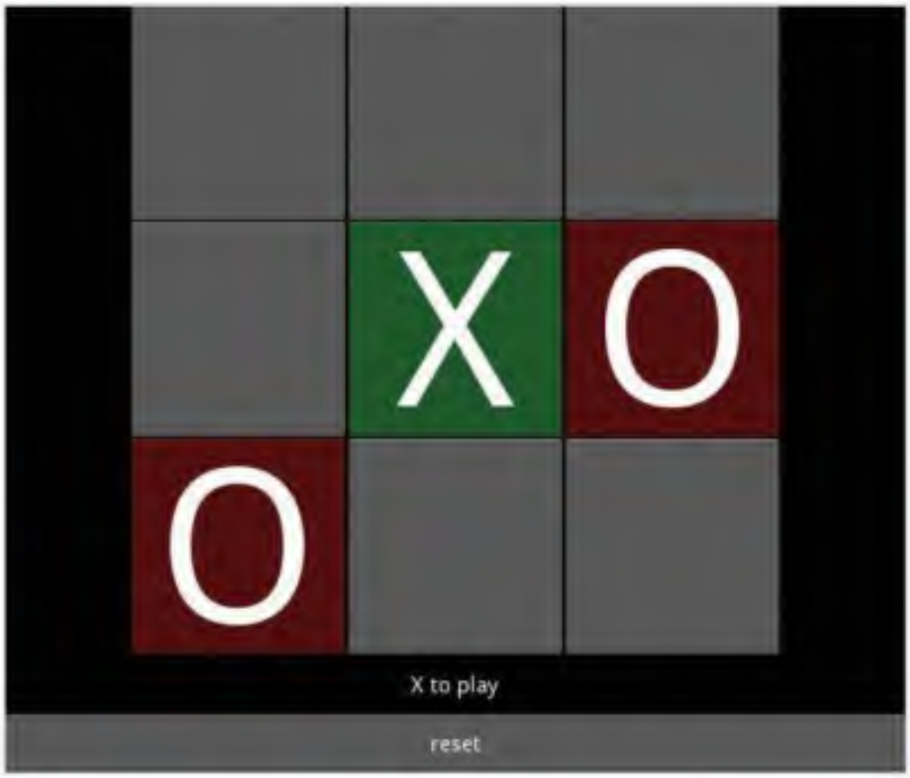
Has anyone won yet?

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 (Fig. 02). 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 keeping track of 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.



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

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

    # Sum each row, column and diagonal.
    # Could be shorter, but let's be extra
    # clear what's going on
    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])]
    # Sums can only be +-3 if one player
    # filled the whole line
    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 (Fig. 03). 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!

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, which you can reference 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 this way. You can 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 that it is always square to fit within its parent. You can play with all these settings to see exactly how it fits together, or maybe even try things like swapping out the different widget types to see how other widgets behave.

> "Try swapping out the different widget types to see how other widgets behave"

Fig 01

```
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 output just to see what's going on
        print('{} button clicked!'.format(instance.coords))
```

Code on FileSilo

Fig 02

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

        row, column = button.coords  # The pressed button is automatically
                                     # passed as an  argument

        # Convert 2D grid coordinates to 1D status index
        status_index = 3*row + column
        already_played = self.status[status_index]

        # If nobody has played here yet, make a new move
        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  # Switch current player
```

Fig 03

```
# Note the *args parameter! It's important later when we make a binding
# to reset, which automatically passes an argument that we don't care about
def reset(self, *args):
    self.status = [0 for _ in range(9)]

    # self.children is a list containing all child widgets
    for child in self.children:
        child.text = ''
        child.background_color = (1, 1, 1, 1)

    self.current_player = 1
```

Create a two-step authentication with Twilio

Increase security in access to your web services by building a simple two-step authentication with Twilio's SMS APIs to help you

Resources

Python 2.7+

Flask 0.10.0:
flask.pocoo.org/

Flask Github:
github.com/mitsuhiko/flask

A Twilio account:
twilio.com

Twilio's Python REST API Helper Library:
github.com/twilio/twilio-python/zipball/master

MySQLDB:
mysql-python.sourceforge.net

Telephony is one of the most versatile technologies in our households. Despite being invented over 100 years ago, we still use the same basic infrastructure that once only carried the voices of people to deliver a rich multitude of media content at incredible speeds. As is often the case with wonderful things, they can often be complex too – and yet phones are more important now to our daily lives than ever. So, what can we do to leverage some of that versatile technology?

Well, for starters we can use an API. Twilio has created a RESTful API that removes a great deal of that complexity of telephony so that we can write apps and services that are able to deliver and receive both phone calls and SMS using various endpoints and services. Neat! In this tutorial, we're going to look at using Twilio to help us create the basic flow for a two-step authentication system for logging into a service. We're also going to be using Flask to help us create our routes and generate our pages, but little of Flask's detail will be covered here.

■ The Twilio interface is kept nice and simple – no unnecessary complications here

01 Get a Twilio account and phone number

Signing up to Twilio is pretty easy. First, head over to http://twilio.com and click the 'Signup' button. At this point, the sign-up process doesn't really differ from any other service, but after you've entered an email address and password you'll be asked for a phone number. Given the nature of Twilio's API, it makes sense for them to ask whether we're human, and having them text us is a good way to confirm that. Hey, it's a two-step authentication, which is exactly what we're working towards.

You can enter any number you have access to, be it a landline or mobile, to confirm who you are, but at this point we suggest you authenticate using a phone that can accept SMS (instead of a landline). Having entered your number, you'll receive a text to authenticate your phone – enter it and you'll be presented with a Twilio phone number. This is your Twilio phone number and you'll be using it to send and receive our authentication texts.

02 Add credit

Just like a mobile phone operator, Twilio is not a free service – although it is very inexpensive. In order to continue, we'll need to add a card and some funds to our newly created Twilio account. On the main page of the dashboard, you'll see a big blue dialog asking to upgrade your trial account; click through and follow the instructions to add a card and the amount of credit you would like to use. The minimum amount of $20 (around £10 GBP) will be more than plenty for this and other projects. Once that's done, you're almost ready to start sending text messages – but first head back over to the Twilio dashboard and copy your account SID and auth token down somewhere, you'll need those a little later.

03 Install the Twilio Helper Library and MySQLDB

The Twilio helper library is a fantastic piece of code that lets you jump straight into sending and handling text messages in no time at all. There are a couple of ways to install the library: you can use either PIP or Easy_Install, like so

```
$ pip install twilio
$ easy_install twilio
```

Or you can download the source code for the helper library and run the 'setup.py' file. It really is as simple as that. Now, for storing the verification tokens we're going to use a MySQL database. To get Python talking to our SQL server, we'll use the Python module MySQLDB, the package for which you can grab like so…

```
apt-get install python-mysqldb
```

In the tutorial resources we have included an SQL dump with the table structure. Import it into a database of your choosing. Assuming everything so far has gone swimmingly, you can create a new project folder/environment and add a new file 'server.py'.

04 Server setup

Open the 'server.py' file for editing. The first thing we're going to do is import the libraries we need for our authentication flow, create the endpoints for our server and assign some of the variables needed to run our Flask server. **(Fig 01)**

You may have noticed the **account_sid** and **auth_token** variable we've set after the import statements. We'll use these with our Twilio client so we can interact with Twilio and our mobile phones. These settings can be found on the Twilio account dashboard, right below the header. We've also connected to our SQL database, so make sure your SQL server is running before you fire up the app, otherwise you'll have an error thrown. Save, now if you run your 'server.py' file, you should be able to access the index page of your server at 127.0.0.1:5000/.

Fig 01

```
import MySQLdb
from flask import Flask, redirect, request, session, render_template
from twilio.rest import TwilioRestClient as twilio
import string, random, time

db = MySQLdb.connect(host="127.0.0.1", user="SQLUSER",
passwd="SQLPASS", ↵ db="two-step", port=3306)

expirationLength = 300

account_sid = "YOUR ACCOUNT SID"
auth_token = "YOUR ACCOUNT AUTH TOKEN"
client = twilio(account_sid, auth_token)

@app.route('/')
def index():
    return "index page"

@app.route('/login', methods=['GET'])
def login():
    return "login page"

@app.route('/check-user', methods=['POST'])
def checkUser():
    return "check user page"

@app.route('/logout')
def logout():
    return "logout page"

@app.route('/verify', methods=['GET'])
def twoStep():
    return "verify page"

@app.route('/check-code', methods=['POST'])
def checkCode():
    return "check code page"

if __name__ == '__main__':

    app.secret_key = 'R4nDOMCRypt0gr4ph1cK3yf0R5355i0N'
    app.run(host='0.0.0.0', debug=True)
```

05 Server logic

If you've hit all of your server endpoints already, so far all you will see are the strings we returned at the end of endpoint declarations. These are not all that good-looking, so let's add some Flask templates to pretty things up a little. The focus of this tutorial is not on the intricacies of Flask and as such, included on the DVD is a folder called 'templates' and another called 'static'; copy them both to the root of your current project folder and amend your endpoints as in **Fig 02**. If you revisit the pages again, things might seem a little out of whack at the moment, but don't worry about that for the time being. It's mostly because we've not put together the server logic to help the templates figure out what to do.

Let's deal with the '/' path first. All we're doing here is checking the state of our session cookies and effecting how the index.html page renders according to that state. If the user isn't logged in, we'll give them a link to the login page, if the user is logged in but hasn't verified, then we'll give them a link to the code verification page. Before we deliver the template we need to check that our session has its particular variables set, otherwise we'll end up getting KeyErrors.

```
@app.route('/')
def index():

    checkSessionState()

    return render_template('index.
html')

def checkSessionState():
```

```
try:
    session['verified'] == True
except KeyError:
    session['verified'] = ''

try:
    session['loggedin'] == True
except KeyError:
    session['loggedin'] = ''

try:
    session['user'] == True
except KeyError:
    session['user'] = ''
```

06 Logging in

The first step in two-step authentication is logging in with a traditional username/email and password. Access your database and create a new user with the following query:

```
INSERT INTO users (username, password, ↵
phonenumber) VALUES ('A USERNAME', 'A ↵
PASSWORD', '+44YOURUSERSPHONENUMBER')
```

For the purposes of this tutorial, the password is plain text – but we implore you, when you're implementing passwords in a live environment, make sure that you hash them. Still, for now we're going to use a plain text password. Our login.html template has a form that's going to POST itself to check-user; here we'll check the validity of the credentials and then trigger the verification if needed. So we're going to use the MySQLDB module to get details from our database.

In order to query our database we need to create a cursor from which to execute our MySQL statements. We do this with cur = db.cursor():

```
@app.route('/check-user',
methods=['POST'])
def checkUser():

  #session.clear()

  if request.method == 'POST':
    #print request.form['username']

    cur = db.cursor()

    cur.execute("""SELECT * FROM ↵
users WHERE username = %s""", ↵
(request.form['username'],))
```

```
result = cur.fetchone()
returnedPassword = result[2]
returnedPhoneNumber = result[3]
```

We can then build an SQL statement using **cur.execute()**. Notice the **%s**; this will be replaced with the value passed through in the next variable. We execute the statement with **cur.fetchone()**, which will get us the first row that the query returns – if there is no result we'll get None and we can then return the user to the login page with an error message. Let's assume we've requested a valid user – we'll next check that the password assigned to that user is the same as the one submitted. If so, we'll generate the validation code to send to the user, which we'll store in the verification table of our database until it's used or expires. We'll need to create a new cursor to insert the verification code into our table. After we've executed the statement we need to commit the changes to the database, we do this with `db.commit()` – we'll then add the results of the query to our session so we can check against them later. (Fig 03)

07 Send the verification code

Now that we believe we're dealing with a valid user, it's time for the second step of our two-step process. On the line after where we stored a variable in our session, we make a call to `sendVerificationCode (VERIFICATION CODE, USER PHONE NUMBER)` and pass through the code we want to send to our user and the user's phone number. So what does that function actually look like? It must be big, long and complicated because it deals with the telecommunications network, right? Wrong. It's actually incredibly simple to send an SMS with Twilio. In fact, part of the inherent beauty of Twilio lies in its simplicity. To send a text, all we have to do is:

```
def sendVerificationCode(code,
number):

  text = client.messages.create(
    body="Your verification code
is:" ↵+ code,
    to=number,
    from_="+YOURTWILIONUMBER"
  )

return text.sid
```

Using the **client** variable we used to instantiate the Twilio REST module, we can access the **messages** class and execute the **create** method. All we need to pass through is the text that will make up the body of our message, the number that we want to send it to and the number that we want to send it from. When inputting the number that we want to send it to, it's best to use the +CountryCode type of phone number to avoid any ambiguity about the intended recipient. The number that we're sending from is our Twilio number; you can use any Twilio number you have assigned to your account, so long as it has credit. As soon as we execute that code, the message will be sent and your SMS will go straight through to the telephone. The SID is the unique identifier for the message/call sent; receiving it means the message has been executed successfully. After that, we can redirect our user to the verification page with `return redirect('/verify')` at the end of /check-user.

08 Check verification code

At this point the user will have received a text message with something along the lines of 'Your verification code is: 12cd56' and will be presented with the verification page. If, at this point, they choose to browse around our site, they won't be able to access anything that we don't want them to. Still, we'll know that they've logged in, so if they head back to the verification page, we can just let them input their code. Once they submit their code, it will be sent to the /check-code endpoint.

Just like before when we checked for our user's validity, we're going to attempt to retrieve the verification code and check it. (Fig 04)

First we're simply going to retrieve the code and check the user it has assigned to it. If that user assigned to the code matches the user in our session, then we can be certain that the right person is logging in with the right code – if not we can redirect them accordingly. Assuming the code is correct, we need to check it's still valid. Back in Step 6, we created an expiration time that was five minutes in the future from when the code was generated. If it's been more than five minutes (or whatever time you've set on it) then we're going to consider it invalid, delete the code from our table and then log out our user so they can start over, like so.

```
elif time.time() > expirationTime:

  expirySQL = db.cursor()

  expirySQL.execute("""DELETE FROM
↵
verification WHERE code=%s""", ↵
(codeToCheck,))
```

```
    expirySQL.close()
    session['loggedin'] == False

  return redirect('/logout')
```

If we manage to pass the tests so far, then we've two-step verified our user – hooray! Surprisingly easy, eh? Before we give our user free reign around our service, we still want to get rid of that token – we don't need it any more and we don't want to risk someone else using it maliciously in the future.

```
else:
    delSql = db.cursor()

    delSql.execute("""DELETE FROM ↵
verification WHERE code=%s""", ↵
(codeToCheck,))

    delSql.close()

    db.commit()

    session['verified'] = True

    return redirect('/')

else:
    return redirect('/
verify?error=true')
```

And that's it! Now we redirect our user to wherever we want them to be at the end of the process. In this instance we're sending them back to our index page, which will render a success message and give the user a link to log out whenever they like – but they could be redirected to their user page, and so on.

09 Conclusion

In every web-based service, security is king. Users entrust more and more personal data and trends to services every day and it's the responsibility of those services to maintain the privacy of that data as best they can. It's no wonder that services such as Amazon, Google and Facebook have all implemented two-step verification across their services. With two-step authentication, a user can tie their account to one of the most personal things they own: their phone. With services like Twilio and some simple code, they contain people's keys – or at least a part of them.

Fig 02

```
@app.route('/')
def index():
  return render_template('index.html')

@app.route('/login', methods=['GET'])
def login():
  return render_template('login.html')

@app.route('/check-user', methods=['POST'])
def checkUser():
  return "check user page"

@app.route('/logout')
def logout():
  return "logout page"

@app.route('/verify', methods=['GET'])
def twoStep():
  return render_template('verify.html')

@app.route('/check-code', methods=['POST'])
def checkCode():
  return "check code page"
```

Fig 03

```
verficationCode = generateVerificationCode(size=6)

ins = db.cursor()

expiration = int(time.time() + expirationLength)

sql = "INSERT INTO verification (code, expiration, username) VALUES ('%s',
'%s', ↵'%s')" % (verficationCode, expiration, request.form['username'])

ins.execute(sql)

ins.close()

db.commit()

session['user'] = request.form['username']
session['loggedin'] = True
```

Fig 04

```
@app.route('/check-code', methods=['POST'])
def checkCode():

  if request.method == 'POST':
    codeToCheck = request.form['code']

    if not 'user' in session:
      return redirect('/login')
    else:
      cur = db.cursor()

      cur.execute("""SELECT * FROM verification WHERE code = %s""", (codeToCheck,))

      result = cur.fetchone()

      cur.close()

      if result != None:
        returnedUser = result[3]
        expirationTime = int(result[2])

        if returnedUser != session['user']:
          return redirect('/verify?error=true')
```

Part one: Program a Space Invaders clone

Write your own RasPi shooter in 300 lines of Python

Resources

Raspbian: www.raspberrypi.org/downloads

Python: www.python.org/doc

Pygame: www.pygame.org/docs

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. While the Invaders meander their way down the screen towards you, it's your job to pick them off while dodging their random fire. When one wave is conquered, another faster, more aggressive wave appears. 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.

We hope you agree that this is an exciting game to play and a great tool to learn more about Python and Pygame, but our sensory system is far from overloaded here. Don't worry, as that will be covered in the next tutorial, adding animation and sound effects to our game to give it the spit and polish any self-respecting *Space Invaders*-inspired shooter demands…

01 Setting up dependencies

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
```

02 Downloading the project

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
```

```
#!/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()
```

Clean clode

Having all the most regularly used global variables clearly labelled here makes our code later on easier to read. Also, if we want to change the size of something, we only need to do it here and it will work everywhere.

Rain bullets

The Ammo class is short and sweet. We only need a few initialising attributes and the update method checks if it's still on the screen. If not, it's destroyed.

Get the code: bit.ly/11k5f2x

```
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()
    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))
```

Groups

This long list of groups we're creating are essentially sets.Each time we create one of these items, it's added to the set so it can be tested for collisions and drawn with ease.

Control

Taking care of keyboard input is the control method. It checks for key events and acts accordingly depending whether we're on the start screen or playing the game.

03 Testing Pivaders

With Pygame installed and the project cloned to your machine (you can also find the .zip on FileSilo – 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.

04 Creating your own clone

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.

05 Global variables & tuples

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.

06 Classes – part 1

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**.

07 Classes – part 2

In Pivader's classes, besides creating the required attributes – these are simply variables in classes – for the object (be it a player, an alien, some ammo or a block), 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 (we've called ours **main_loop()**) 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 the top or bottom of the screen, we destroy it (since we don't need it any more).

08 Ammo

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. This saves us creating extra unnecessary code and ensures consistent behaviour between objects we create.

We used widely available open source art and fonts to make the game

09 The game

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) and deciding if we need to return to the start screen, among other things.

10 The main loop

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 actual logic that dictates what happens within any one round of the game is actually 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. Starting at the top of **main_loop()** and working line-by-line down to its last line, you can see exactly what's being evaluated 20 times every second when you're playing the game.

11 Main loop key logic – part 1

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()**.

12 Main loop key logic – part 2

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.

"A class is essentially a blueprint"

```
    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)
        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()
```

Refreshing the screen You need to carefully consider the way in which you update the screen. Blitting the background between actor movements is vital for clean animation.

Guns 'n' ammo Bullets and missiles use the same parent class. We change a few key attributes originally initialised to create the behaviour we need; eg the vector for each is opposite.

Dead or alive Probably two of the most important questions are answered here – is the player dead or did you win the round?

Get the code: bit.ly/11k5f2x

```
  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()
```

Main loop This is the business end of our application. This loop executes 20 times a second. It needs to be logical and easy for another coder to understand.

Start the game The very last thing we do is create a Game object and call the main loop. Besides our constants, this is the only code that sits outside a class.

Part two: Add animation and sound to Pivaders

After writing a Space Invaders clone in just 300 lines of Python, now we expand it to include animation and sound

Resources

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, for the previous tutorial. One of the key challenges with the project was keeping it to a manageable size – just 300 lines of Python. Without the use of Pygame's strong set of features, that goal would likely have been overshot at least twofold. 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 length but in simplicity. 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, including all the images, music and sound effects we used, over at git.io/8QsK-w.

Even working within the clearly defined framework Pygame offers, there are still a thousand ways we could have approached adding animation and sound. We could have created any one of a dozen classes to create and manage containers of individual images, or read in a sprite sheet (a single image full of smaller, separate images) which we could then draw (or blit) to the screen. For the sake of simplicity and performance, 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.

01 Setting up dependencies

As we recommended with the last tutorial, 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 for your Pygame projects. 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. If you're unsure if you have them, though, type the following at the command line:

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

Pivaders.py listing from line 86 (continued on next page)

Get the code: bit.ly/1xPvY1F

```
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 = []
        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()
```

ship_sheet

We set the player image to be equal to one small segment of the sprite sheet by using the 'ani_pos' variable. Change the variable to change the picture

Set flags

We've added 'animate_left' and 'animate_right' Boolean flags to the control method. When they're true, the actual animation code is called via a separate method

fx.play()

Having already loaded the sound effect we want when we shoot, we now just need to call it when we press the space bar

02 Downloading pivaders

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 will create a folder called **pivaders** – go inside (**cd pivaders**) and take a look around.

03 Navigating the project

The project is laid out quite simply across a few subfolders. 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 to the main screen with the Esc key and press it again to exit the game completely.

04 Animation & sound

Compared with the game from last month's tutorial, you'll see it's now a much more dynamic project. The protagonist ship now leans into the turns as you change direction and corrects itself when you either press the opposite direction or lift your finger off the button. 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.

05 Finding images to animate

Before we can program anything, it's wise to have assets set up in a way we can use them. As mentioned, we've opted to use sprite sheets; these can be found online or created with GIMP with a little practice. Essentially they're a mosaic made up of individual 'frames' of equally sized and spaced images representing each frame. Find ready-made examples at **opengameart.org**, as used here.

06 Tweaking assets

While many of the assets on sites like **opengameart.org** 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 – as we did with our ship sheet asset to help us keep the code simple. 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.

07 Loading the sprite sheet

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.

Above The Freesound site is a good place to find free and open sound effects for projects

08 Animation flags

Down the list in the initialising code for the **Game** class, we set two flags for player animation: **self.animate_left** and **self.animate_right**. In the **Control** method of our Game class, we use these to 'flag' when we want animations to work using Boolean values. It allows us to 'automatically' animate the player sprite back to its resting state (otherwise the ship will continue to look as if it's flying left when it has stopped).

09 The animation method

These flags pop up again in the core animation code for the player: **animate_player()** within the Game class. Here we use nested **if** statements to control the animation and physically set the player image accordingly. Essentially 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 accordingly. The **Else** statement then animates the ship sprite back to its resting state and the same logic is then applied in the opposite direction.

10 Animating explosions

The **player_explosion()** and **alien_explosion()** methods that come after the player animation block in the Game class are similar but simpler executions of essentially 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 displayed. As the sprite sheet is vertical, the variables **alien_explode_pos** and **explosion_image** are set to a different part of **subsurface** than before.

11 Adding music to your project

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 with this code:

```
pygame.mixer.music.load('data/sound/10_Arpanauts.ogg')
pygame.mixer.music.play(-1)
pygame.mixer.music.set_volume(0.7)
```

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**.

12 Using sound effects

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(
    'data/sound/medetix__pc-bitcrushed-lazer-beam.ogg')
```

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.

If you're struggling to find free and open sound effects, we recommend **www.freesound.org**.

```
        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.animate_player()
        self.player_explosion()
        self.alien_explosion()
        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)
                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()
            self.level_up = 50
```



```
            self.explode = False
            self.alien_explode = False
            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()
                self.level_up = 50
                self.explode = False
                self.alien_explode = False
                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 next_round(self):
        self.explode = False
        self.alien_explode = False
        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 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)

        for z in pygame.sprite.groupcollide(
        self.bullet_group, self.alien_group, True, True):
            self.alien_explode = True
            self.explodey_alien.append(z.rect.x)
            self.explodey_alien.append(z.rect.y)
            self.score += 10
            self.explosion_fx.play()

        if pygame.sprite.groupcollide(
        self.player_group, self.missile_group, False, True):
            self.lives -= 1
            self.explode = True
            self.explosion_fx.play()

    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()
                self.calc_collisions()
                self.refresh_screen()
                if self.is_dead() or self.defenses_breached():
                    GameState.start_screen = True
                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()
                if self.win_round():
                    self.next_round()
            self.splash_screen()
        pygame.quit()

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

Create a Minecraft Minesweeper game

Use your Raspberry Pi and Python knowledge to code a simple mini-game in Minecraft

Resources

Jessie or Raspbian:
raspberrypi.org/downloads

Python:
www.python.org/

You may remember or have even played the classic Minesweeper PC game that originally dates back to the 60s. Over the years it has been bundled with most operating systems, appeared on mobile phones, and even featured as a mini-game variation on *Super Mario Bros*.

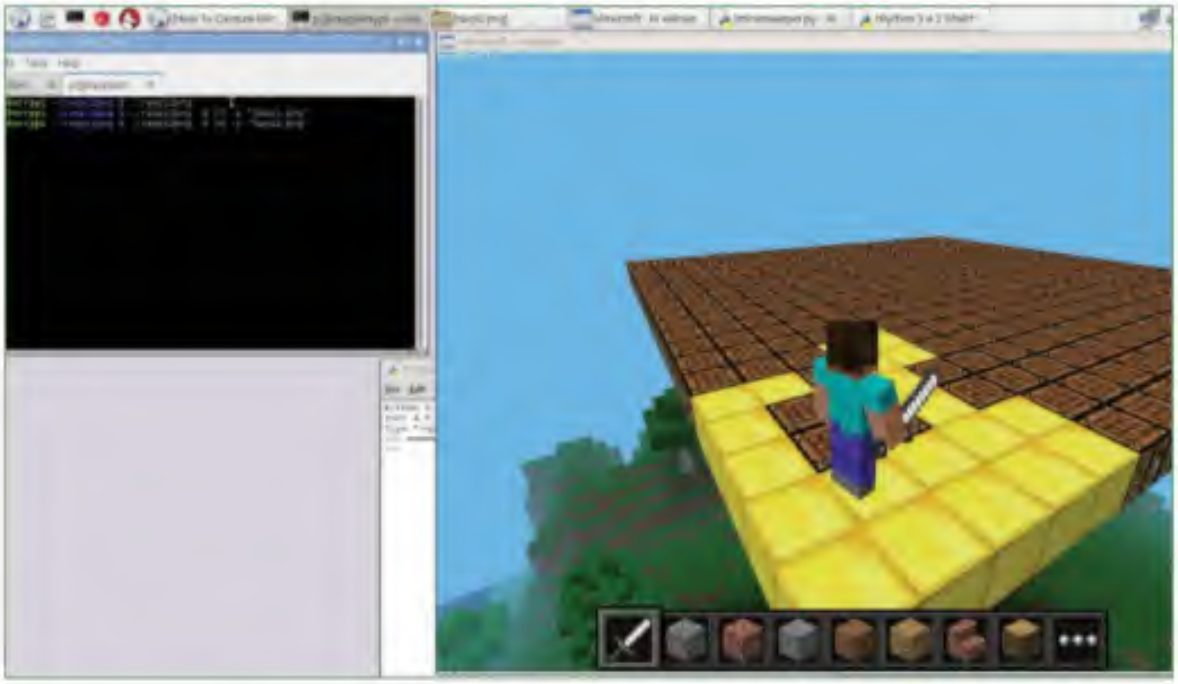
This project will walk you through how to create a simple version in *Minecraft*: it's *Minecraft* Minesweeper! You will code a program that sets out an arena of blocks and turns one of these blocks into a mine. To play the game, guide your player around the board. Each time you stand on a block you turn it to gold and collect points, but watch out for the mine as it will end the game and cover you in lava!

01 Update and install

To update your Raspberry Pi, open the terminal and type:

```
sudo apt-get upgrade
sudo apt-get update
```

The new Raspberry Pi OS image already has Minecraft and Python installed. The Minecraft API which enables you to interact with *Minecraft* using Python is also pre-installed. If you are using an old OS version, we would highly recommend you downloading and then updating your version to either the most recent Jessie or Raspbian image.



Right The safe blocks have been turned into gold – the rest are potential mines!

02 Importing the modules

Load up your preferred Python editor and start a new window. You need to import the following modules: import random to calculate and create the random location of the mine, and import time to add pauses and delays to the program. Next, add a further two lines of code: from mcpi import minecraft and mc = minecraft.Minecraft.create(). These create the program link between Minecraft and Python. The mc variable enables you to write "mc" instead of "minecraft.Minecraft.create()".

```
import random
import time
from mcpi import minecraft
mc = minecraft.Minecraft.create()
```

03 Grow some flowers

Using Python to manipulate *Minecraft* is easy; create the program below to test it is working. Each block has its own ID number, and flowers are 38. The x, y, z = mc.player.getPos() line gets the player's current position in the world and returns it as a set of coordinates: x, y, z. Now you know where you are standing in the world, blocks can be placed using mc.setBlock(x, y, z, flower). Save your program, open MC and create a new world.

```
flower = 38
while True:
 x, y, z = mc.player.getPos()
 mc.setBlock(x, y, z, flower)
 time.sleep(0.1)
```

04 Running the code

Reducing the size of the MC window will make it easier for you to see both the code and the program running; switching between both can be frustrating. The Tab key will release the keyboard and mouse from the MC window. Run the Python program and wait for it to load – as you walk around, you'll drop flowers! Change the ID number in line 1 to change the block type, so instead of flowers, try planting gold, water or even melons.

Switching to the shell

Switching between the Python Shell and Minecraft window can be frustrating, especially as MC overlays the Python window. The best solution is to half the windows across the screen. (Don't run MC full-screen as the mouse coordinates are off). Use the Tab key to release the keyboard and mouse from the MC window.

05 Posting a message to the Minecraft world

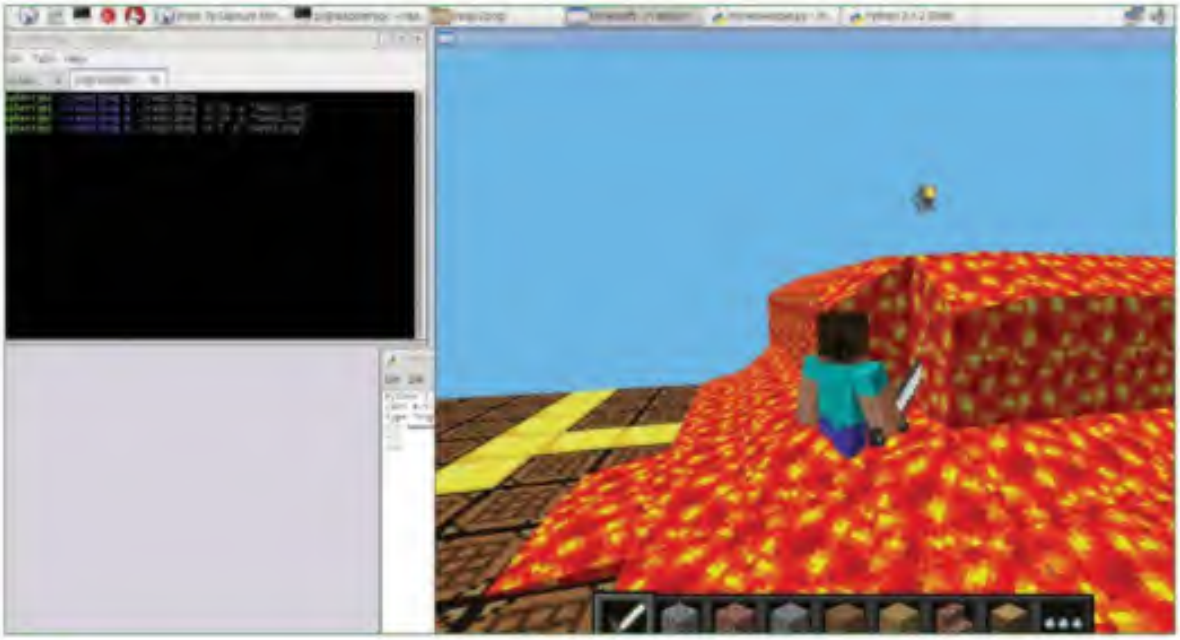
It is also possible to post messages to the *Minecraft* world. This is used later in the game to keep the player informed that the game has started and also of their current score. In your previous program add the following line of code under the flower = 38 line, making this line 2: mc.postToChat("I grew some flowers with code"). Now save and run the program by pressing F5 – you will see the message pop up. You can try changing your message, or move to the next step to start the game.

06 Create the board

The game takes place on a board created where the player is currently standing, so it is advisable to fly into the air or find a space with flat terrain before running your final program. To create the board you need to find the player's current location in the world using the code x, y, z = mc.player.getPos(). Then use the mc.setBlocks code in order to place the blocks which make up the board:

```
mc.setBlocks(x, y-1, z, x+20, y-1, z+20, 58).
```

The number 58 is the ID of the block that is a crafting table. You can increase or decrease the size of the board by changing the +20. In the code example above, the board size is 20 x 20 blocks, which gives you a 400-block arena to play within.



Right Nothing says "game over" quite like a huge eruption of lava

07 Creating the mine

In the previous step you found the player's location on the board. This x, y, z data can be reused to place the mine on the board. The code **mine = random.randrange(0, 11, 1)** generates a random number between 1 and 10. Combine this with the player's current x axis position and add the random number to the position – this creates a random mine block on the board.

```
mine_x = int(x+mine)
mine_y = int(y-1)
mine_z = int(z+mine)
```

Use setBlock to place the mine: **mc.setBlock(mine_x, mine_y, mine_z,58)**. Using **y-1** ensures that the block is placed on the same level as the board and is therefore hidden. The number 58 is the block ID, which you can change if you wish to see where the mine is; this is useful for testing that the rest of the code is working correctly. Remember to change it back before you play!

08 Create a score variable

Each second that you remain alive within the game, a point is added to your score. Create a variable to store the current score, setting it to a value of zero at the beginning of the game. Use the **postToChat** code to announce the score at the beginning of the game. Note that MC cannot print a value to chat, so the score is first converted into a string before it is displayed.

```
score = 0
mc.postToChat("Score is "+str(score))
time.sleep(10)
```

09 Check the player's position on the board

Next, you need to check the player's position on the board and see if they are standing on the mine. This uses a **while** loop to continually check that your player's position is safe, no mine, else game over. Since the player's coordinate position is used to build the original board and place the mine, you have to find the player's position again and store it as a new variable – x1, y1 and z1

```
while True:
  x1, y1, z1 = mc.player.getTilePos()
```

10 One point, please

Now that the player has moved one square they are awarded a point. This is a simple action of adding the value one to the existing score value. This is achieved using **score = score + 1**. Since it sits inside a loop, it will add one point each time the player moves.

```
time.sleep(0.1)
 score = score + 1
```

11 The tension increases…

Once you have been awarded the point, the next stage of the game is to check whether the block you are standing on is a safe block or if it is the mine. This uses a conditional to compare the coordinates of the block beneath you – x1, y1-1, z1 – with the mine_x, mine_y, mine_z position of the mine. If they are equal then you are standing on the mine. In the next step you will code the explosion:

```
if (x1, y1-1, z1) == (mine_x, mine_y, mine_z):
```

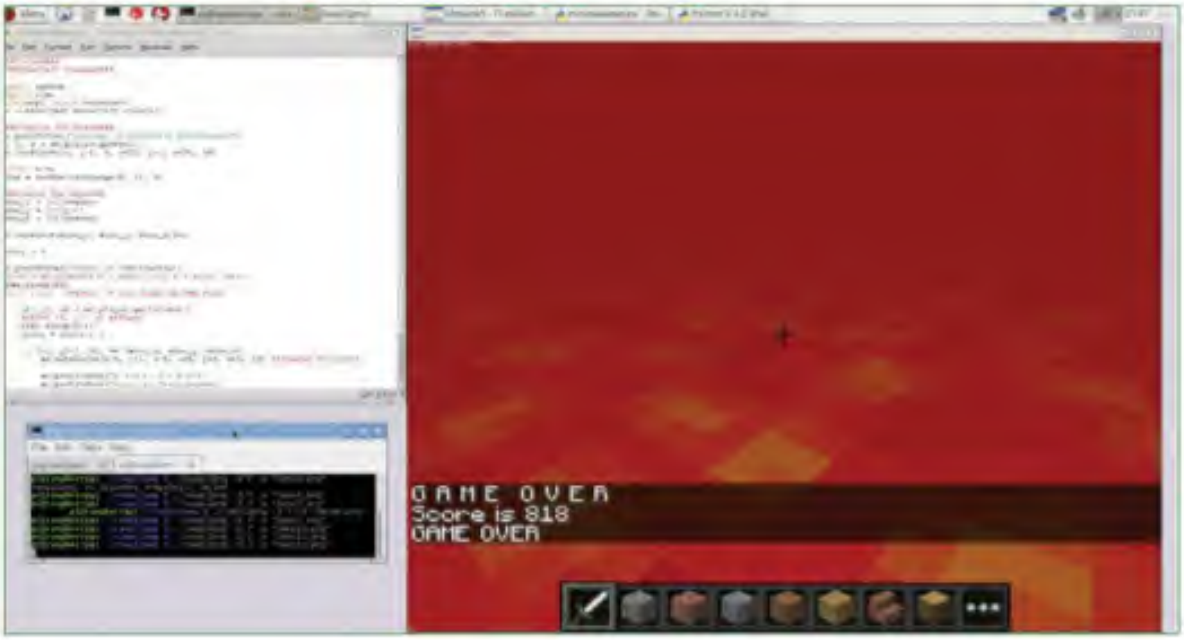
12 Setting the mine off

In the previous step a conditional checks whether you are standing on the mine or a safe block. If it is the mine then it will explode. To create this, use lava blocks, which will flow and engulf the player. You can use the **mc.setBlocks** code to set blocks between two points. Lava blocks are affected by gravity, so setting them higher than the player means that the lava flows down over the player.

```
mc.setBlocks(x-5, y+1, z-5, x+5, y+2, z+5, 10)
```

Other Minecraft hacks

If you enjoy programming and manipulating Minecraft then there are more great Raspberry Pi -based projects for you to check out. Our expert has a bunch of them at tecoed.co.uk/minecraft.html. The folks behind Adventures In Minecraft have some great guides over at stuffaboutcode.com/p/minecraft.html, as well.



Left Once finished, our mini-game uses the chat console to report your score

13 Game over

If you do stand on the mine, the game is over. Use the post to chat code to display a "Game Over" message in the *Minecraft* World.

```
mc.postToChat("G A M E  O V E R")
```

14 Final score

The last part of the game is to give a score. This uses the score variable that you created in Step 8 and then uses the **mc.postToChat** code. Convert the score to a string first so that it can be printed on the screen. Since your turn has ended, add a break statement to end the loop and stop the code from running.

15 Safe block

But what if you missed the mine? The game continues and you'll need to know where you have previously been on the board. Use the code **mc.setBlock(x1, y1-1, z1, 41)** to change the block you are standing on into gold or another material of your choice. In the code, the Y positon is Y – 1, which selects the block beneath the player's feet.

16 Increment the score

As well as living to play another turn, you also gain a point. This is achieved by incrementing the score variable by one each time you turn the block gold and return to the beginning of the loop to check the status of the next block you step on. The **postToChat** is to tell you that you have survived another move!

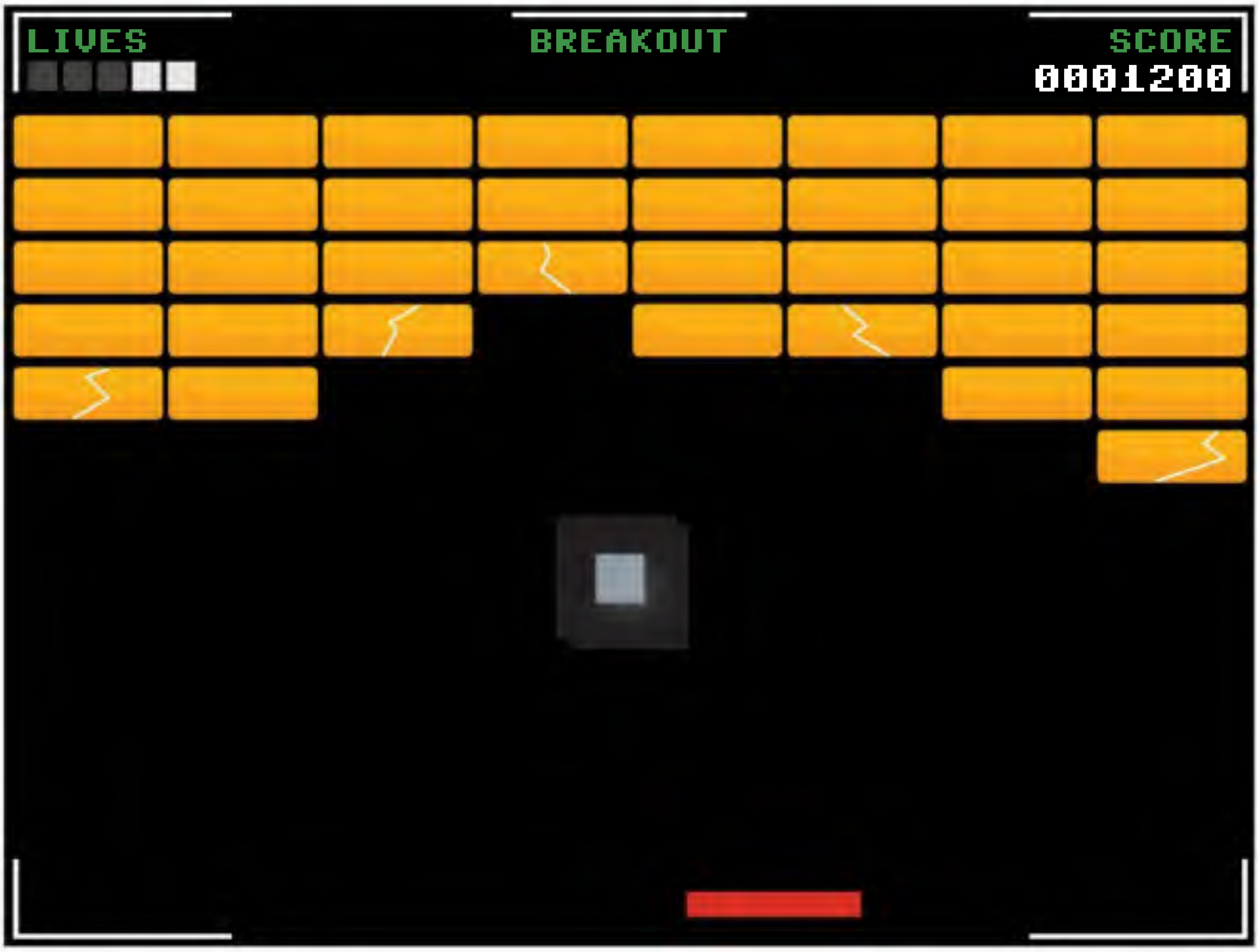
```
score = score + 1
mc.postToChat("You are safe")
```

17 Run the game

That completes the code for the program. Save it and then start a *Minecraft* game. Once the world has been created, run the Python program. Move back to the *Minecraft* window and you will see the board created in front of you.

Full code listing

```
import random
import time
from mcpi import minecraft
mc = minecraft.Minecraft.create()

###Creates the board###
mc.postToChat("Welcome to Minecraft MineSweeper")
x, y, z = mc.player.getPos()
mc.setBlocks(x, y-1, z, x+20, y-1, z+20, 58)

global mine
mine = random.randrange(0, 11, 1)

###Places the mine###
mine_x = int(x+mine)
mine_y = int(y-1)
mine_z = int(z+mine)
mc.setBlock(mine_x, mine_y, mine_z,58)
score = 0
mc.postToChat("Score is "+str(score))

time.sleep(5)
while True: ###Test if you are standing on the mine

  x1, y1, z1 = mc.player.getTilePos()
  #print x1, y1, z1 ###test
  time.sleep(0.1)
  score = score + 1

  if (x1, y1-1, z1) == (mine_x, mine_y, mine_z):
    mc.setBlocks(x-5, y+1, z-5, x+5, y+2, z+5, 10)
    mc.postToChat("G A M E O V E R")
    mc.postToChat("Score is "+str(score))
    break
  else:
    mc.setBlock(x1, y1-1, z1, 41)
mc.postToChat("GAME OVER")
```



Pygame Zero

Pygame Zero cuts out the boilerplate to turn your ideas into games instantly, and we'll show you how

Resources

Pygame Zero: pygame-zero.readthedocs.org

Pygame: pygame.org/download.shtml

Pip pip-installer.org

Python 3.2 or later www.python.org/

Code from FileSilo (optional)

Games are a great way of understanding a language: you have a goal to work towards, and each feature you add brings more fun. However, games need libraries and modules for graphics and other essential games features. While the Pygame library made it relatively easy to make games in Python, it still brings in boilerplate code that you need before you get started – barriers to you or your kids getting started in coding.

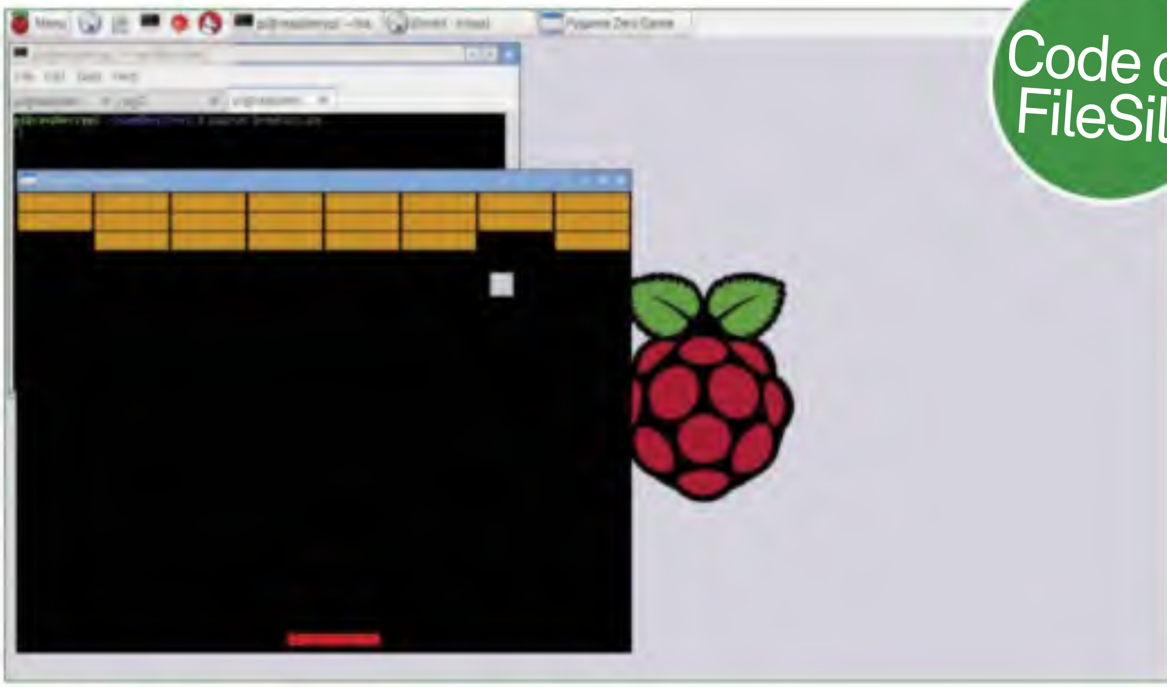
Pygame Zero deals with all of this boilerplate code for you, aiming to get you coding games instantly. Pg0 (as we'll abbreviate it) makes sensible assumptions about what you'll need for a game – from the size of the window to importing the game library – so that you can get straight down to coding your ideas.

Pg0's creator, Daniel Pope, told us that the library "grew out of talking to teachers at Pycon UK's education track, and trying to understand that they need to get immediate results and break lessons into bite-size fragments, in order to keep a whole class up to speed."

To give you an idea of what's involved here, we'll build up a simple game from a *Pong*-type bat and ball through to smashing blocks *Breakout*-style. The project will illustrate what can be done with very little effort. Pg0 is in early development but still offers a great start – and is now included on the Pi in the Raspbian Jessie image.

We'll look at installation on other platforms, but first let's see what magic it can perform.

Right *Breakout* is a classic arcade game that can be reimagined in Pygame Zero



Young and old

In situations where Pygame is used boilerplate and all with young people, great results can also be achieved (see Bryson Payne's book), but Pygame and Pg0, despite their use as powerful educational tools, are also good for creating games for coders no matter what stage of learning they are at.

Great games are all about the gameplay, driven by powerful imaginations generating images, animations, sounds and journeys through game worlds. Good frameworks open up this creative activity to people who are not traditional learners of programming, which is an area where Python has long excelled.

01 Zero effort

Although game writing is not easy, getting started certainly is. If you've got Raspbian Jessie installed on your Pi, you're ready to go. Open a terminal and type:

```
touch example.py
pgzrun example.py
```

And you'll see an empty game window open (Ctrl+Q will close the window). Yes, it's that easy to get started!

02 Python 3

If you haven't got Raspbian Jessie, chances are you'll have neither Pg0 nor Pygame installed. The Python's pip package installer will take care of grabbing Pg0 for you, but the preceding steps vary by distro. One thing you will need is Python 3.2 (or newer). If you've been sticking with Python 2.x in your coding (perhaps because it's used in a tutorial you're following), make Pg0 your chance for a gentle upgrade to Python 3.

03 Older Raspbian

If you're still running Raspbian Wheezy, you'll need to run the following steps to install Pygame Zero:

```
sudo apt-get update
sudo apt-get install python3-setuptools python3-pip
sudo pip-3.2 install pgzero
```

04 No Pi?

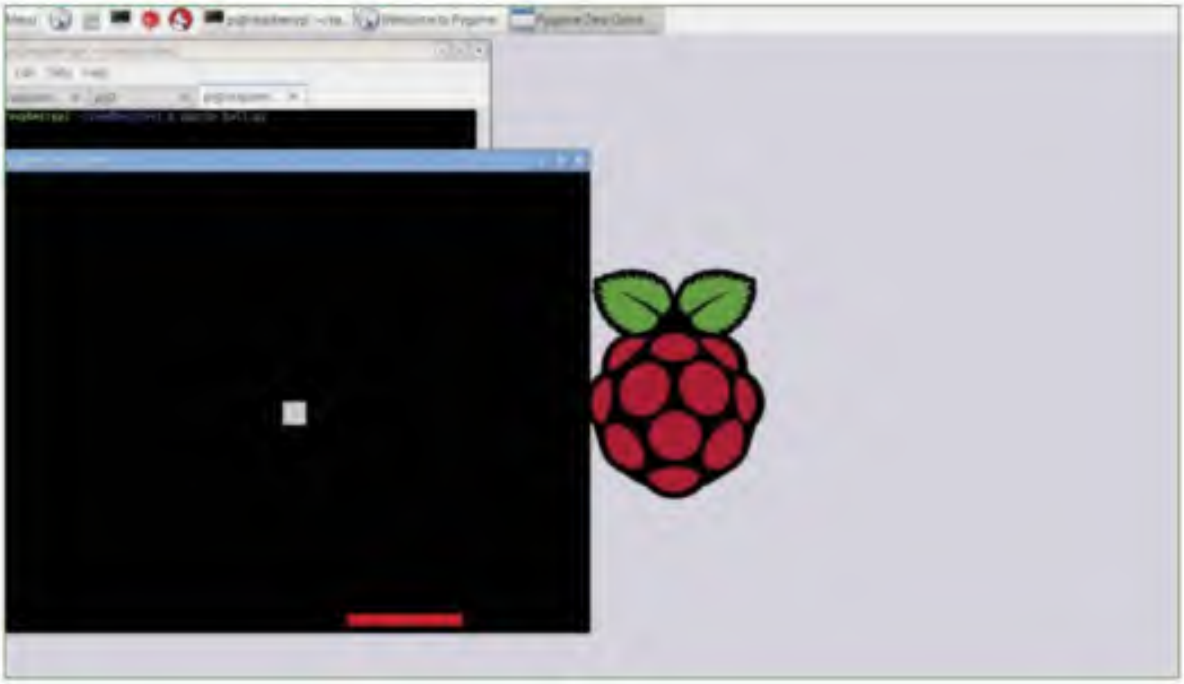
You don't even need a Raspberry Pi to install Pygame Zero – just install the Pygame library, then use pip to install Pygame Zero. Instructions vary by distro, but a good place to start is the documentation: **bit.ly/1GYznUB**.

05 Intro.py

That default black square of 800 by 600 pixels we saw in Step 1 can now be overridden manually. For example, we can use the following code to replace it with an oversized gold brick, in a sly nod to *Breakout*:

```
WIDTH = 1000
HEIGHT = 100
def draw():
  screen.fill((205, 130, 0))
```

That colour tuple takes RGB values, so you can quickly get colours off a cheatsheet; **screen** is built into Pg0 for the window display, with methods available for all sorts of different sprites…



Right The bat and ball come first – they're the cornerstones of *Pong* and *Breakout*

Program objects

David Ames, who uses Pg0 to teach younger children to code at events across the UK, told us: "One thing to avoid when it comes to teaching kids is Object Orientation." OOP (object-oriented programming) is partly abstracted away by Pg0, but it can't be ignored.

Perhaps the best approach is using Pg0 and some simple code to start, then dropping in a piece of OO when it's needed to solve a particular problem.

With the Code Club age group – about eight to eleven – feeding information to solve practical problems works well. It can work with adults, too – but there's always someone who's read ahead and has a few tricky questions.

06 Sprite

The intro example from the Pg0 docs expands on that with the **Actor** class, which will automatically load the named sprite (Pg0 will hunt around for a .jpg or .png in a subdirectory called images).

```
alien = Actor('alien')
alien.pos = 100, 56
WIDTH = 500
HEIGHT = alien.height + 20
def draw():
  screen.clear()
  alien.draw()
```

You can download the alien from the Pg0 documentation (**bit.ly/1Sm5lM7**) and try out the animation shown there, but we're taking a different approach in our game.

07 Breakout via Pong

While the Pi is something of a tribute to 1980s 8-bit computers, *Breakout* comes from the 1970s and is a direct descendant of the early arcade classic *Pong*. We'll follow the route from *Pong* to *Breakout* (which historically involved Apple founders Steve Wozniak and Steve Jobs) in the steps to creating our code, leaving you with the option of developing the *Pong* elements into a proper game, as well as refining the finished *Breakout* clone.

08 Batty

You can think of *Breakout* as essentially being a moving bat – that is, you're hitting a moving ball in order to knock out blocks. The bat is a rectangle, and Pygame's **Rect** objects store and manipulate rectangular areas – we use **Rect((left, top), (width, height))**, before which we define the bat colour and then call upon the **draw** function to put the bat on the screen, using the **screen** function.

```
W = 800
H = 600
RED = 200, 0, 0
bat = Rect((W/2, 0.96 * H), (150, 15))
def draw():
  screen.clear()
  screen.draw.filled_rect(bat, RED)
```

09 Mouse move

We want to move the bat, and the mouse is closer to an arcade paddle than the arrow keys. Add the following:

```
def on_mouse_move(pos):
  x, y = pos
  bat.center = (x, bat.center[1])
```

Use **pgzrun** to test that you have a screen, bat and movement.

"To get the ball to move we need to define move(ball) for each case where the ball meets a wall"

Full code listing

```
## Breakout type game to demonstrate Pygame Zero library
## Based originally upon Tim Viner's London Python Dojo
## demonstration
## Licensed under MIT License - see file COPYING

from collections import namedtuple
import pygame
import sys
import time

W = 804
H = 600
RED = 200, 0, 0
WHITE = 200,200,200
GOLD = 205,145,0

ball = Rect((W/2, H/2), (30, 30))
Direction = namedtuple('Direction', 'x y')
ball_dir = Direction(5, -5)

bat = Rect((W/2, 0.96 * H), (120, 15))

class Block(Rect):

    def __init__(self, colour, rect):
        Rect.__init__(self, rect)
        self.colour = colour

blocks = []
for n_block in range(24):
    block = Block(GOLD, ((((n_block % 8)* 100) + 2, ((n_block //
              8) * 25) + 2), (96, 23)))
    blocks.append(block)

def draw_blocks():
    for block in blocks:
        screen.draw.filled_rect(block, block.colour)

def draw():
    screen.clear()
    screen.draw.filled_rect(ball, WHITE)
    screen.draw.filled_rect(bat, RED)
    draw_blocks()

def on_mouse_move(pos):
    x, y = pos
    bat.center = (x, bat.center[1])

def on_mouse_down():
    global ball_dir
    ball_dir = Direction(ball_dir.x * 1.5, ball_dir.y * 1.5)
```

10 Square ball

In properly retro graphics-style, we define a square ball too – another rectangle, essentially, with the **(30, 30)** size making it that subset of rectangles that we call a square.

We're doing this because **Rect** is another built-in in Pg0. If we wanted a circular ball, we'd have to define a class and then use Pygame's **draw.filled_circle(pos, radius, (r, g, b))** - but **Rect** we can call directly. Simply add:

```
WHITE = 200,200,200
ball = Rect((W/2, H/2), (30, 30))
```

... to the initial variable assignments, and:

```
    screen.draw.filled_rect(ball, WHITE)
```

... to the **def draw()** block.

11 Action!

Now let's make the ball move. Download the tutorial resources in **FileSilo.co.uk** and then add the code inside the 'move.py' file to assign movement and velocity. Change the **5** in **ball_dir = Direction(5, -5)** if you want the ball slower or faster, as your processor (and dexterity) demands – but it's hard to tell now because the ball goes straight off the screen! Pg0 will call the **update()** function you define once per frame, giving the illusion of smooth(ish) scrolling if you're not running much else.

12 def move(ball)

To get the ball to move within the screen we need to define **move(ball)** for each case where the ball meets a wall. For this we use **if** statements to reverse the ball's direction at each of the boundaries. Refer to the full code listing on page 67.

Note the hardcoded value of 781 for the width of screen, minus the width of ball – it's okay to hardcode values in early versions of code, but it's the kind of thing that will need changing if your project expands. For example, a resizable screen would need a value of **W - 30**.

13 Absolute values

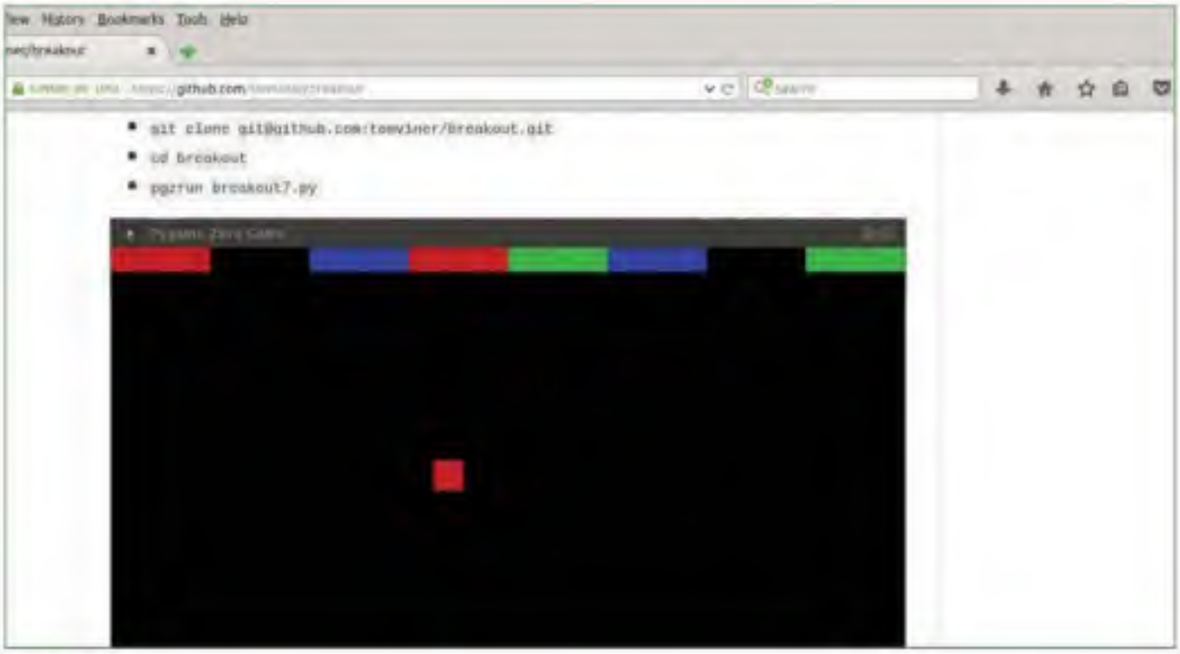
You might expect multiplying y by minus one to work for reversing the direction of the ball when it hits the bat:

```
ball_dir = Direction(ball_dir.x, -1 * ball_dir.y)
```

... but you actually need to use **abs**, which removes any minus signs, then minus:

```
ball_dir = Direction(ball_dir.x, - abs(ball_dir.y))
```

Try it without in the finished code and see if you get some strange behaviour. Your homework is to work out why.



Right Tom Viner's array of blocks negates the need for bordered rectangles

14 Sounds

Also upon bat collision, **sounds.blip.play()** looks in the sounds subdirectory for a sound file called blip. You can download the sounds (and finished code) from **FileSilo.co.uk**.

Actually, now we think about it, ignore the previous comment about homework – your real homework is to turn what we've written so far into a proper game of *Pong*! But first let's finish turning it into *Breakout*!

15 Blockhead!

If you're not very familiar with the ancient computer game *Breakout*, then:

```
apt-get install lbreakout2
```

... and have a play. Now, we haven't set our sights on building something quite so ambitious in just these six pages, but we do need blocks.

16 Building blocks

There are many ways of defining blocks and distributing them onto the screen. In Tom Viner's team's version, from the London Python Dojo – which was the code that originally inspired this author to give this a go – the blocks are sized in relation to number across the screen, thus:

```
N_BLOCKS = 8
BLOCK_W = W / N_BLOCKS
BLOCK_H = BLOCK_W / 4
BLOCK_COLOURS = RED, GREEN, BLUE
```

Using multicoloured blocks which are then built into an array means that blocks can join without needing a border. With its defining variables in terms of screen width, it's good sustainable code, which will be easy to amend for different screen sizes – see **github.com/tomviner/breakout**.

However, the array of colour bricks in a single row is not enough for a full game screen, so we're going to build our array from hard-coded values...

17 Going for gold

Create a Block class:

```
class Block(Rect):
  def __init__(self, colour, rect):
      Rect.__init__(self, rect)
      self.colour = colour
```

... and pick a nice colour for your blocks:

```
GOLD = 205,145,0
```

18 Line up the blocks

This builds an array of 24 blocks, three rows of eight:

```
blocks = []
for n_block in range(24):
  block = Block(GOLD, ((((n_block % 8)* 100) + 2,
    ((n_block // 8) * 25) + 2), (96, 23)))
  blocks.append(block)
```

19 Drawing blocks

Draw_blocks() is added to **def draw()** after defining:

```
def draw_blocks():
  for block in blocks:
      screen.draw.filled_rect(block, block.colour)
```

20 Block bashing

All that remains with the blocks is to expand **def move(ball)** – to destroy a block when the ball hits it.

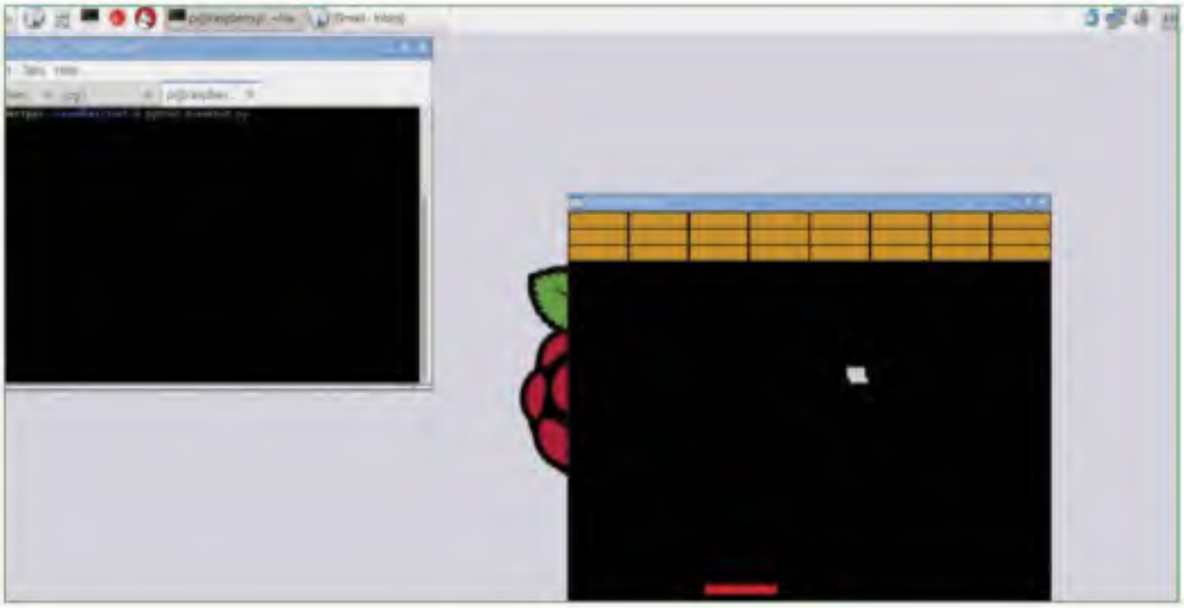
```
to_kill = ball.collidelist(blocks)
if to_kill >= 0:
  sounds.block.play()
  ball_dir = Direction(ball_dir.x, abs(ball_dir.y))
  blocks.pop(to_kill)
```

Pg0 +1

There's a new version of Pg0 in development – it may even be out as you read this. Pg0 creator Daniel Pope tells us "a tone generation API is in the works," and that at the Pg0 PyConUK sprint, "we finished Actor rotation."

Contributions are welcome – not only to the Pg0 code, but more examples are needed not just to show what can be done, but to give teachers tools to enthuse children about the creative act of programming.

Pg0 has also inspired GPIO Zero, to make GPIO programming easier on the Raspberry Pi, with rapid development occurring on this new library as we go to press.



Left Test your game once it's finished – then test other people's *Breakout* games to see how the code differs

Full code listing (cont.)

```
def move(ball):
    global ball_dir
    ball.move_ip(ball_dir)

    if ball.x > 781 or ball.x <= 0:
        ball_dir = Direction(-1 * ball_dir.x, ball_dir.y)
    if ball.y <= 0:
        ball_dir = Direction(ball_dir.x, abs(ball_dir.y))
    if ball.colliderect(bat):
        sounds.blip.play()
        ball_dir = Direction(ball_dir.x, - abs(ball_dir.y))

    to_kill = ball.collidelist(blocks)
    if to_kill >= 0:
        sounds.block.play()
        ball_dir = Direction(ball_dir.x, abs(ball_dir.y))
        blocks.pop(to_kill)

    if not blocks:
        sounds.win.play()
        sounds.win.play()
        print("Winner!")
        time.sleep(1)
        sys.exit()

    if ball.y > H:
        sounds.die.play()
        print("Loser!")
        time.sleep(1)
        sys.exit()

def update():
    move(ball)
```

21 Game over

Lastly, we need to allow for the possibility of successfully destroying all blocks.

```
if not blocks:
    sounds.win.play()
    sounds.win.play()
    print("Winner!")
    time.sleep(1)
    sys.exit()
```

22 Score draw

Taking advantage of some of Pygame Zero's quickstart features, we've a working game in around 60 lines of code. From here, there's more Pg0 to explore, but a look into Pygame unmediated by the Pg0 wrapper is your next step but one.

First refactor the code; there's plenty of room for improvement – see the example 'breakout-refactored.py' which is included in your tutorial resources. Try adding scoring, the most significant absence in the game. You could try using a global variable and writing the score to the terminal with **print()**, or instead use **screen.blit** to put it on the game screen. Future versions of Pg0 might do more for easy score keeping.

23 Class of nine lives

For adding lives, more layers, and an easier life-keeping score, you may be better defining the class **GameClass** and enclosing much of the changes you wish to persist within it, such as **self.score** and **self.level**. You'll find a lot of Pygame code online doing this, but you can also find Pg0 examples, such as the excellent pi_lander example by Tim Martin: **github.com/timboe/pi_lander**.

24 Don't stop here

This piece is aimed at beginners, so don't expect to understand everything! Change the code and see what works, borrow code from elsewhere to add in, and read even more code. Keep doing that, then try a project of your own – and let us know how you get on.

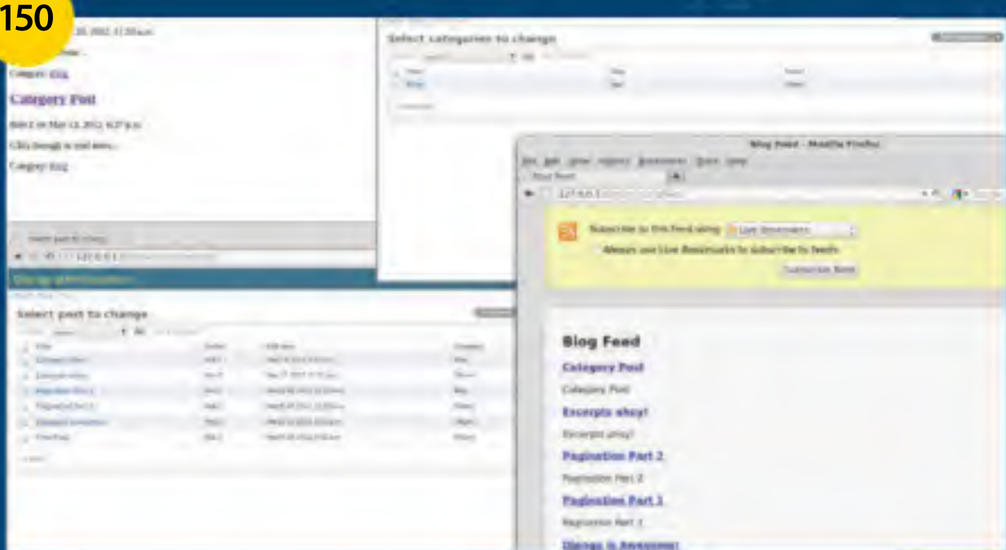
Web development

138

132

"Python is a versatile language, perfect for making websites"

150




lenovo
LG
id="
>var z =

DEVELOP WITH PYTHON

Don't be fooled into thinking Python is a restrictive language or incompatible with the modern web. Explore options for building Python web apps and experience rapid application development

Why?

First released in 1991, companies like Google and NASA have been using Python for years

Thanks to the introduction of the Web Server Gateway Interface (WSGI) in 2003, developing Python web apps for general web servers became a viable solution as opposed to restricting them to custom solutions.

Python executables and installers are widely available from the official Python site at **www.python.org**.

Mac OS X users can also benefit greatly from using Homebrew to install and manage their Python versions. Whilst OS X comes bundled with a version of Python, it has some potential drawbacks. Updating your OS may clear out any downloaded packages, and Apple's implementation of the library differs greatly from the official release. Installing using Homebrew helps you to keep up to date and also means you get the Python package manager pip included.

Once Python is installed the first package to download should be virtualenv using 'pip install virtualenv', which enables you to create project-specific shell environments. You can run projects on separate versions of Python with separate project-specific packages installed.

Check out the detailed Hitchhiker's Guide to Python for more information: **docs.python-guide.org/en/latest**.

Frameworks

Let's take a look at some of the frameworks available when developing Python web applications

Django djangoproject.com

GOOD FOR: Large database-driven web apps with multiuser support and sites that need to have heavily customisable admin interfaces

Django contains a lot of impressive features, all in the name of interfaces and modules. These include autowiring, admin interfaces and database migration management tools by default for all of your projects and applications. Django will help to enable rapid application development for enterprise-level projects, whilst also enabling a clear modular reuseable approach to code using subapplications.

Tornado tornadoweb.org

GOOD FOR: Web socket interaction and long polling due to its ability to scale to manage vast numbers of connections

Tornado is a networking library that works as a nonblocking web server and web application framework. It's known for its high performance and scalability and was initially developed for friendfeed, which was a real-time chat system that aggregated several social media sites. It closed down in April 2015 as its user numbers had declined steadily, but Tornado remains as active and useful as ever.

Flask flask.pocoo.org

GOOD FOR: Creating full-featured RESTful APIs. Its ability to manage multiple routes and methods is very impressive

Flask's aim is to provide a set of commonly used components such as URL routing and templates. Flask will also work on controlling the request and response objects, all-in-all this means it is lightweight but is still a powerful microframework.

PyramiD pylonsproject.org

GOOD FOR: Highly extensible and adaptable to any project requirement. Not a lightweight system either

Heavily focused on documentation, Pyramid brings all the much needed basic support for most regular tasks. Pyramid is open source and also provides a great deal of extensibility – it comes with the powerful Werkzeug Debugger too.

Werkzeug werkzeug.pocoo.org

GOOD FOR: API creation, interacting with databases and following strict URL routes whilst managing HTTP utilitie

Werkzeug is the underlying framework for Flask and other Python frameworks. It provides a unique set of tools that will enable you to perform URL routing processes as well as request and response objects, and it also includes a powerful debugger.

Create an API

Let us explore the Flask microframework and build a simple yet powerful RESTful API with minimal code

01 Install Flask

Create a new directory inside of which your project will live. Open a Terminal window and navigate to be inside your new directory. Create a new virtual environment for this project, placed inside a new directory called 'venv', and activate it. Once inside the new virtual shell, proceed to installing Flask using the 'pip install Flask' command.

```
virtualenv venv
. venv/bin/activate
pip install Flask
```

02 Create Index

Create a new file in the root of the project location called 'index.py'. The sample API will use a SQLite database, so we need to import that module for use in the application. We'll also import some core components from the Flask module to handle request management and response formatting as well as some other functions. The minimum import for a Flask application is Flask itself.

```
import sqlite3
from flask import Flask, request, g,
redirect, url_for, render_template,
abort, jsonify
```

03 Declare Config

For a small application we can declare configuration options as upper-case name value pairs inside the main module, which we'll do now. Here we can define the path and name of the SQLite database and also set the Flask debug output to True for development work. Initialise the Flask application to a namespace and then import the config values set directly above it. We then run the application. All routes must be placed above these last two lines.

```
# Config
DATABASE = '/tmp/api.db'
DEBUG = True
app = Flask(__name__)
app.config.from_object(__name__)
# Add methods and routes here
if __name__ == '__main__':
app.run()
```

04 Connect to Database

With the database path defined, we need a way to create connection to the database for the application to obtain data. Create a new method called 'connet_db' to manage this for us. As a method we can call it when we set up a prerequest hook shortly. This will return a new open connection using the database details set in the configuration object.

```
def connect_db():
return sqlite3.connect(app.
config['DATABASE'])
```

05 Database Schema

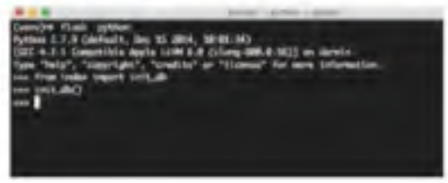
Our SQLite database will only contain one table. Create a new file called 'schema.sql' in the root of the project directory. This fill will contain the SQL commands required to create the table and populate it with some sample bootstrapped data.

```
drop table if exists posts;
create table posts (
    id integer primary key autoincrement,
    title text not null,
    text text not null
);
insert into posts (title, text) values
('First Entry', 'This is some text');
insert into posts (title, text) values
('Second Entry', 'This is some more text');

insert into posts (title, text) values
('Third Entry', 'This is some more text
(again)');
```

06 Instantiate the Database

To populate the database with the new table and any associated data, we will need to import and apply the schema to the database. Add a new module import at the top of the project file to obtain the 'contextlib.closing()' method. What we will do next is create a method that will initialise the database by reading the contents of schema.sql and executing it against the open database.

```
from contextlib import closing
def init_db():
  with closing(connect_db()) as db:
    with app.open_resource('schema.sql',
mode='r') as f:
      db.cursor().executescript(f.read())
    db.commit()
```



07 Populate the Database

To populate the database you can now run the init_db inside an active python shell. To do so enter a shell by typing 'python' inside your environment, and then running the command below. Alternatively, you can use the sqlite3 command and pipe the schema.sql file into the database.

```
# Importing the database using the
init_db method
python
>>> from index import init_db
>>> init_db()
# Piping the schema using SQLite3
sqlite3 /tmp/api.db < schema.sql
```

08 Request DB Connection

With the database created and populated we need to be able to ensure we have an open connection and close it accordingly when finished. Flask has some decorator methods to help us achieve this. The before_request() method will establish the connection and stores it in the g object for use throughout the request cycle. We can then close the connection after the cycle using the teardown_request() method.

```
@app.before_request
def before_request():
  g.db = connect_db();
```

"World-renowned image sharing service Instagram and social pin board Pinterest have also implemented Python as part of their web stack, opting for Django"

```
@app.teardown_request
def teardown_request(exception):
  db = getattr(g, 'db', None)
  if db is not None:
    db.close()
```

09 Display Posts

Create your first route so that we can return and display all available posts. To query the database we execute a SQL statement against the stored db connection. The results are then mapped to values using Python's dict method and saved as the posts variable. To render a template we then call render_template() and pass in the file name and the variable to display as the second argument. Multiple variables can be passed through as a comma-separated list.

```
@app.route('/')
def get_posts():
  cur = g.db.execute('select title, text
from posts order by id desc')
  posts = [dict(title=row[0], text=row[1])
for row in cur.fetchall()]
  return render_template('show_posts.
html', posts=posts)
```

10 Template Output

Flask expects templates to be available within the templates directory in the root of the project, so make sure that you create that directory now. Next, add a new file called 'show_posts.html'. The dynamic values are managed using Jinja2 template syntax, the default templating engine for Flask applications. Save this file in the templates directory.

```
<ul class=posts>
  {% for post in posts %}
  <li><h2>{{ post.title }}</h2>{{ post.
text|safe }}
  {% else %}
  <li>Sorry, no post matches your
request.
  {% endfor %}
</ul>
```

11 Make an API Response

To create an API response we can define a new route with a specific API endpoint. Once again, we query the database for all posts. The data is then returned as JSON, using the JSONify method to do so. We can add specific values such as post count and a custom message if you wish, as well as the actual posts variable, formatted as JSON.

```
@app.route('/api/v1/posts/',
methods=['GET'])
def show_entries():
  cur = g.db.execute('select title, text
from posts order by id desc')
  posts = [dict(title=row[0],
text=row[1]) for row in ur.fetchall()]
  return jsonify({'count': len(posts),
'posts': posts})
```

12 Get a specific Post

CTo obtain a specific post from the API we need to create a new route, which will accept a dynamic value as part of the URI. We can also choose to use this route for multiple request methods, which are in this case GET and DELETE. We can determine the method by checking the request.method value and run it against a conditional if/else statement.

```
@app.route('/api/v1/posts/<int:post_id>',
methods=['GET', 'DELETE'])
def single_post(post_id):
  method = request.method
  if method == 'GET':
    cur = g.db.execute('select title,
text from posts where id =?', [post_id])
  posts = [dict(title=row[0],
text=row[1]) for row in cur.fetchall()]
    return jsonify({'count': len(posts),
'posts': posts})
  elif method == 'DELETE':
    g.db.execute('delete from posts
where id = ?', [post_id])
    return jsonify({'status': 'Post
deleted'})
```

13 Run the application

To run your Flask application, navigate using the active Terminal window into the root of the project. Ensuring you are in an active virtual environment Python shell, enter the command to run the main index file. The built-in server will start and the site will be accessible in the browser on default port local address http://127.0.0.1:5000.

```
python index.py
```

14 API JSON Output

The root of the application will render the template we previously created. Multiple routes can be generated to create a rich web application. Visiting an API-specific URL in the browser will return the requested data as cleanly formatted JSON. The ability to define custom routes like a versioned RESTful endpoint is incredibly powerful.

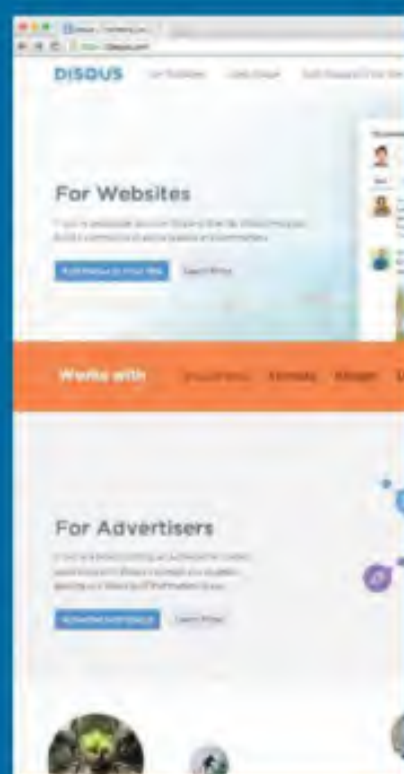
Python in the wild

Interested in Python development? You'd be in good company with big names currently using it

Disqus, the popular social interaction comment service provider, has been implementing their production applications in Python for a very long time. Python's benefit for the development team was its ability to scale effectively and cater for a large number of consumers whilst also providing an effective underlying API for internal and external use. The company are now starting to run some production apps in Go, but the majority of code still runs on Python.

World-renowned image sharing service Instagram and social pin board Pinterest have also implemented Python as part of their web stack, opting for Django to assist with the functionality and ability to cater for the many thousands of content views and requests made to their services.

Mozilla, Atlassian's Bitbucket repository service, and popular satire site The Onion have all been noted as using Django for their products.



Django application development

Django is a full Python web-app framework with impressive command-line tools

Installing Django

The installation of Django is relatively easy once you have python installed. See for yourself as we build a simple app here

01 Create Virtual Environment

Create a new directory for your project and navigate inside it using a new Terminal window. Create a new virtual environment for this project, opting to use the latest Python 3. Your Python 3 location may vary, so be sure to set the correct path for the binary package.

```
virtualenv -p /usr/local/bin/python3 venv
```

02 Activate and Install

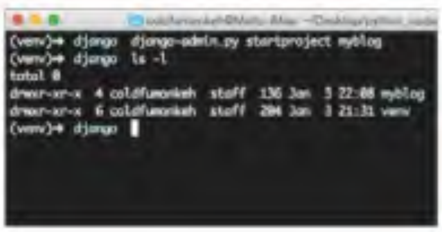
Using your Terminal window, activate the virtual environment to start the project-specific shell. VirtualEnv has a local version of the Python package manager pip installed, so it's fairly straight forward to run the command to install Django.

```
. venv/bin.activate
pip install Django
```

03 Create Core Project

The Django install contains some incredibly useful command-line tools, which will help you to run a number of repetitive and difficult tasks. Let's use one of them to create a fresh project structure for us. Run the django-admin.py script with the name of the project that you want created.

```
django-admin.py startproject myblog
```



04 Initial Migration

Navigate into the project directory via the Terminal window. Some of the installed apps included in the project generation require database tables.

Using the helper. run a migration command to create all of these automatically. The Terminal window will keep you informed of all of your progress and what has been applied from the migration.

```
cd myblog
python manage.py migrate
```

05 Create App

Each Django project is made up of at least one application or module. Run the startapp command to create a new blog app module, which will generate the required code adjacent to the main project structure.

```
python manage.py startapp blog
```

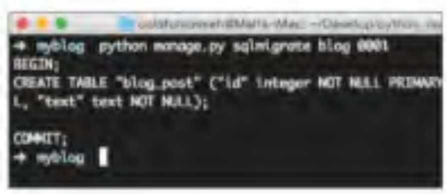
Database models & migration

Django's ability to manage the migration and maintenance of database schema and project models is very impressive

01 Generate the model

Open blog/models.py and create the first model class, providing the property names and types for each. You can dig deeper into field types via the docs here: bit.ly/1yln1kn. Once complete, open myblog/settings.py and add the blog app to the list of allowed installed applications so that the project will load it.

```
# blog/models.py
class Post(models.Model):
  title = models.CharField(max_
length=200)
  text = models.TextField()
# myblog/settings.py
INSTALLED_APPS = ('django.contrib.admin',
..., 'django.contrib.staticfiles', 'blog')
```



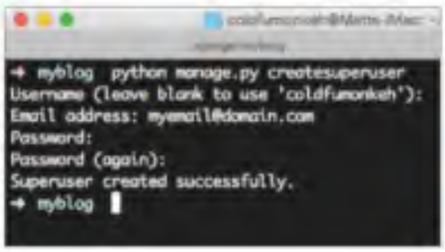
02 Data Migration

Any creation of models or changes to data need to be migrated. To do so we need to make migration files from the model data, which generate sequentially numbered files. Then we run a specific migration to generate the required SQL and the final migrate command performs the database execution.

```
python manage.py makemigrations blog
python manage.py sqlmigrate blog 0001
python manage.py migrate
```

Autowiring the admin interface

Admin sections can be problematic in their own right. Django provides an extensible admin interface for you



01 Create Admin User

Django makes content administration incredibly easy and has an admin section available in a default project as standard at http://127.0.0.1:8000/admin. To log in you need to create a superuser account. Run the associated command and specify user details as required to then proceed and log in.

```
python manage.py createsuperuser
```

02 Switch on blog management

Having logged in to the administration interface you will be greeted with features to manage users and group roles and privileges, which alone are very powerful and provided for you by Django. There is not yet, however, any access to manage our blog posts so let's turn that on.

Using the dev server

Django ships with a very helpful built-in development server, which will help you out by autocompiling and reloading after you have completed all of your file changes. All you have to do to start the server is to run the 'python manage.py runserver' command from your Terminal window within the project directory.

03 Enable Admin Management

To enable our module and associated models to be managed through the admin interface, we need to register them with the admin module. Open blog/admin.py and then go on to import and register the models in turn (we only have one of these currently though). Save the file and refresh the admin site to see the posts that are now available to manage.

```
from django.contrib import admin

# Register your models here.
```

04 Create a View

With the admin interface accepting new submissions for our post class we'll create a view page to display them. Open blog/views.py and import the Post class from the models. Create a method to obtain all posts from the database and output them as a string.

```
from django.http import HttpResponse
from blog.models import Post
def index(request):
  post_list = Post.objects.order_by('-id')
[:5]
  output = '<br />'.join([p.title for p
in post_list])

return HttpResponse(output)
```

05 Manage the URLs

Create 'blog/urls.py' and add the code to import the views that were just made in the module and the accompanying URL patterns. Open myblog/urls.py and add the URL function call to implement a new URL for the app to display the view. Visit http://127.0.0.1:5000/blog in your browser to render the new view.

```
# blog/urls.py

from django.conf.urls import patterns,url
from blog import views
urlpatterns = patterns('',
  url(r'^$', views.index, name='index'),
)

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

Hosting Python apps

Admin sections can be problematic in their own right. Django provides an extensible admin interface for you

Heroku heroku.com

This app is perhaps one of the most well-known cloud hosting providers. Their stack server environments support a number of core web app languages including Python as standard. Their unique Toolbelt command-line features and integration with Git repositories, as well as being incredibly quick and easy to scale and improve performance, makes them an obvious choice. A free account will let you run a Python web app on one dyno instance without any cost.

Python Anywhere www.pythonanywhere.com

Another hosted option, and one created specifically for Python applications in general is Python Anywhere. The free basic option plan has enough weight and power behind it to get you up and running with a Python web app without having to scale, but as soon as your project gains traction, you can switch plans and boost your plans performance.

It offers an incredibly impressive range of modules as standard, available to import into your application immediately to get you started, including Django and Flask should you need them.

Creating dynamic templates with Flask, Jinja2 and Twitter

Create a dynamic webpage with Twitter and Flask's rendering engine, Jinja2

Resources

Python 2.7+

Flask 0.10.0: flask.pocoo.org

Flask GitHub: github.com/mitsuhiko/flask

A Twitter account

Your favourite text editor

Code downloaded from FileSilo

The template uses a loop to generate a list of Twitter tweets

Python and Flask are a great combination when you're looking to handle the Twitter OAuth process and build requests to obtain tokens. We've used Twitter here because of the large amount of easily digestible data available on it. However, since Twitter adheres to the standards set out by OAuth 1.0, the code we've used to sign and build requests can be modified to work with any third-party API using the same standard without a great deal of work. For years PHP has been a mainstay of template generation, but now with well-documented frameworks such as Flask, Sinatra and Handlebars, the ability to use powerful scripting languages greatly improves our ability to make great web services. Here, we're going to use Python, Flask and its templating engine to display tweets. Flask comes with the super-nifty Jinja2 templating engine, If you're familiar with Node.js or front-end JavaScript, the syntax will look very similar to the Handlebars rendering engine. But, before we dive into that, we need to organise some of the example code that we're using for this.

01 Rearranging our code

Server code can get messy and unmaintainable quickly, so the first thing we're going to do is move our helper functions to another file and import them into our project, much like you would a module. This way, it will be clear which functions are our server logic and endpoints and which are generic Python functions. Open the **TwitterAuthentication** file downloaded from FileSilo (stored under **Twitter OAuth files**) and locate the getParameters, sign_request and create_oauth_headers functions. Cut and paste them into a new file called **helpers.py** in the root of your project folder. At the top of this file we want to import some libraries.

```
import urllib, collections, hmac, ↵
binascii, time, random, string

from hashlib import sha1
```

Now we can head back over to server.py and import the helper functions back into our project. We do this by simply calling `import helpers`. Because Python is smart, It will look in the current directory for a helpers.py file before it looks for a system module. Now every function included in helpers.py is accessible to our project. All we need to do to call them is prepend our the methods we called before with helper.function_name and it will execute. For sign_request, we'll need to pass our oauth_secret and consumer_secret for each request rather than accessing it from the session. Adjust the function declaration like so:

```
def sign_request(parameters, method,
↵baseURL, consumer_secret, oauth_
secret):
```

02 server.py modules

With a lot of the modules needed in this project having been moved to **helpers.py**, we can now remove most of them from **server.py**. If we amend our first import statement to be...

```
import urllib2, time, random, json
```

...our project will continue to function as it did before. Note the addition of the json module: we'll be using that later as we start handling Twitter data.

Having Flask use a rendering engine is super-simple. Flask comes packaged with the Jinja2 template rendering engine, so we've nothing to install – we just need to import the package into the project. We can do this by adding render_template to the end of our `from flask import […]` statement.

03 Our first template

Now that we have a rendering engine, we need to create some templates for it to use. In the root of our project's folder, create a new folder called **templates**. Whenever we try to render a template, Flask will look in this folder for the template specified. To get to grips with templating, we'll rewrite some of our authentication logic to use a template, rather than manually requesting endpoints. In **templates**, create an **index.html** file. You can treat this HTML file like any other – included in the resources for this tutorial is an index.html that includes all of the necessary head tags and <!DOCTYPE> declarations for this file.

04 Rendering our template

In **server.py**, let's create a route for '/' to handle the authorisation process.

```
@app.route('/')
def home():

    if not 'oauth_token' in session:
        session.clear()
        session['oauth_secret'] = ''
        session['oauth_token'] = ''
    return render_template('index.html')
```

It's a simple function: all we want to do is check whether or not we have an oauth_token already and create those properties in the Flask session so we don't throw an error if we try to access it erroneously. In order to send our generated template in response to the request, we `return render_template('index.html')`.

05 Template variables

We can choose to send variables to our template with `render_template('index.htm', variableOne=value, variableTwo=Value)` but in this instance we don't need to as each template has access to the request and session variables.

Fig 01

```
{% if session['oauth_token'] != "" %}
    <h1>Already Authorised</h1>
    <div class="dialog">
<p>Hello, You've authenticated!<br>Let's <a href="/get-tweets">get some tweets</a></p>
    </div>
{% else %}
    <h1>Authorisation required</h1>
    <div class="dialog">
        <p>We need to <a href="/authenticate">authenticate</a></p>
    </div>

{% endif %}
```

Code on FileSilo

The BSD-licensed Flask is easy to set up and use – check out the website for more info

Open **index.html**. All code executed in a Flask template is contained within {% %}. As this is our homepage, we want to direct users accordingly, So let's check if we've got an access token **(Fig 01)**.

Between the **ifs** and **else** of the template is standard HTML. If we want to include some data – for example, the access token – we can just add **{{ session['oauth_token'] }}** in the HTML and it will be rendered in the page. Previously, in our / authorised endpoint, we would display the OAuth token that we received from Twitter; however, now that we have a template, we can redirect our users back our root URL and have a page rendered for us that explains the progress we've made.

06 Getting lost (and then found again)

With every server, some things get misplaced or people get lost. So how do we handle this? Rather than defining a route, we can define a handler that deals with getting lost.

```
@app.errorhandler(404)
def fourOhFour(error):
return render_template('fourohfour.html')
```

If a page or endpoint is requested and triggers a 404, then the **fourOhFour** function will be fired. In this case, we'll generate a template that tells the user, but we could also redirect to another page or dump the error message.

07 Static files

Pretty much every webpage uses JavaScript, CSS and images, but where do we keep them? With Flask we can define a folder for use with static content. For Flask, we create a **static** folder in the root of our project and access files by calling **/static/css/styles.css** or **/static/js/core.js**. The tutorial resources include a CSS file for styling this project.

08 Let's get some tweets

So now we know how to build templates, let's grab some tweets to display. In **server.py** define a new route, **get-tweets**,like so:

```
@app.route('/get-tweets')
@app.route('/get-tweets/<count>')
def getTweets(count=0):
```

You'll notice that unlike our other authentication endpoints, we've made two declarations. The first is a standard route definition: it will intercept and handle the path **get-tweets**. The second lets us define a parameter that we can use as a value in our **getTweets** function. By including **count=0** in our function declaration, we ensure that there will always be a default value when the function is executed; this way we don't have to check the value is present before we access it. If a value is included in the URL, it will override the value in the function. The string inside the **<variable name>** determines the name of the variable. If you want the variable passed to the function to have a specific type, you can include a converter with the variable name. For example, if we wanted to make sure that **<count>** was always an integer instead of a float or string, we'd define our route like so:

> "Now we know how to build templates, let's grab some tweets to display"

```
@app.route('/get-tweets/<int:count>')
```

09 Checking our session and building our request

Before we start grabbing tweets, we want to run a quick check to make sure we have the necessary credentials and if not, redirect the user back the authorisation flow. We can do this by having Flask respond to the request with a redirection header, like so:

```
if session['oauth_token'] == "" or ↵
session['oauth_secret'] == "":
        return redirect(rootURL)
```

Assuming we have all we need, we can start to build the parameters for our request **(Fig 02)**.

You'll notice that the nonce value is different from that in our previous requests. Where the nonce value in our authenticate and authorise requests can be any random arrangement of characters that uniquely identify the request, for all subsequent requests the nonce needs to be a 32-character hexadecimal string using only the characters a-f. If we add the following function to our **helpers.py** file, we can quickly build one for each request.

```
def nonce(size=32, chars="abcdef" +
↵string.digits):
  return ''.join(random.choice ↵
(chars) for x in range(size))
```

10 Signing and sending our request

We've built our parameters, So let's sign our request and then add the signature to the parameters **(Fig 03)**.

Before we create the authorisation headers, we need to remove the **count** and **user_id** values from the **tweetRequestParams** dictionary, otherwise the signature we just created won't be valid for the request. We can achieve this with the **del** keyword. Unlike our token requests, this request is a GET request, so instead of including the parameters in the request body, we define them as query parameters.

```
?count=tweetRequestParams['count'] ↵
&user_id=tweetRequestParams['user_id']
```

11 Handling Twitter's response

Now we're ready to fire off the request and we should get a JSON response back from Twitter. This is where we'll use the json module we imported earlier. By using the **json.loads** function, we can parse the JSON into a dictionary that we can access and we'll pass through to our **tweets.html** template.

```
tweetResponse = json. ↵
loads(httpResponse.read())
return render_template('tweets.html',
↵data=tweetResponse)
```

Whereas before, we accessed the session to get data into our template, this time we're explicitly passing a value through to our template.

12 Displaying our tweets

Let's create that template now, exactly the same as **index.html** but this time, instead of using a conditional, we're going to create a loop to generate a list of tweets we've received.

First, we check that we actually received some data from our request to Twitter. If we've got something to render, we're ready to work through it, otherwise we'll just print that we didn't get anything.

Once again, any template logic that we want to use to generate our page is included between

{% %}. This time we're creating a loop; inside the loop we'll be able to access any property we have of that object and print it out. In this template we're going to create an <li> element for each tweet we received and display the user's profile picture and text of the tweet (Fig 04).

In our template we can access properties using either dot notation (.) or with square brackets ([]). They behave largely the same; the [] notation will check for an attribute on the dictionary or object defined whereas the . notation will look for an item with the same name. If either cannot find the parameter specified, it will return undefined. If this occurs, the template will not throw an error, it will simply print an empty string. Keep this in mind if your template does not render the expected data: you've probably just mis-defined the property you're trying to access.

Unlike traditional Python, we need to tell the template where the **for** loop and **if/else** statements end, so we do that with **{% endfor %}** and **{% endif %}**.

13 Flask filters

Sometimes, when parsing from JSON, Python can generate erroneous characters that don't render particularly well in HTML. You may notice that after **tweet['text']** there is **|forceescape**, This is an example of a Flask filter; it allows us to effect the input before we render – in this case it's escaping our values for us. There are many, many different built-in filters that come included with Flask. Your advisor recommends a full reading of all the potential options.

14 Wrapping up

That's pretty much it for templating with Flask. As we've seen, it's insanely quick and easy to build and deploy dynamic sites. Flask is great tool for any Python developer looking to run a web service. Although we've used Twitter to demonstrate Flask's power, all of the techniques described can be used with any third-party service or database resource. Flask can work with other rendering engines, such as Handlebars (which is superb), but Jinja2 still needs to be present to run Flask and conflicts can occur between the two engines. With such great integration between Flask and Jinja2, it makes little sense to use another engine outside of very specific circumstances.

Fig 02

```
tweetRequestParams = {
    "oauth_consumer_key" : consumer_key,
    "oauth_nonce" : helpers.nonce(32),
    "oauth_signature_method" : "HMAC-SHA1",
    "oauth_timestamp" : int(time.time()),
    "oauth_version" : "1.0",
    "oauth_token" : session[‚Äòoauth_token'],
    "user_id" : session['user_id'],
    "count" : str(count)
}
```

Fig 03

```
tweetRequest = helpers.sign_request(tweetRequestParams, "GET",
"https://api.twitter.com/1.1/statuses/user_timeline.json", consumer_secret,
session['oauth_secret'])

tweetRequestParams["oauth_signature"] = tweetRequest

makeRequest=urllib2.Request("https://api.twitter.com/1.1/statuses/
user_timeline.json?count=" + tweetRequestParams['count'] + "&user_id="
+ tweetRequestParams['user_id'])

del tweetRequestParams['user_id'], tweetRequestParams['count']

makeRequest.add_header("Authorization", helpers.create_oauth_
headers(tweetRequestParams))

try:
    httpResponse = urllib2.urlopen(makeRequest)
except urllib2.HTTPError, e:
    return e.read()
```

Fig 04

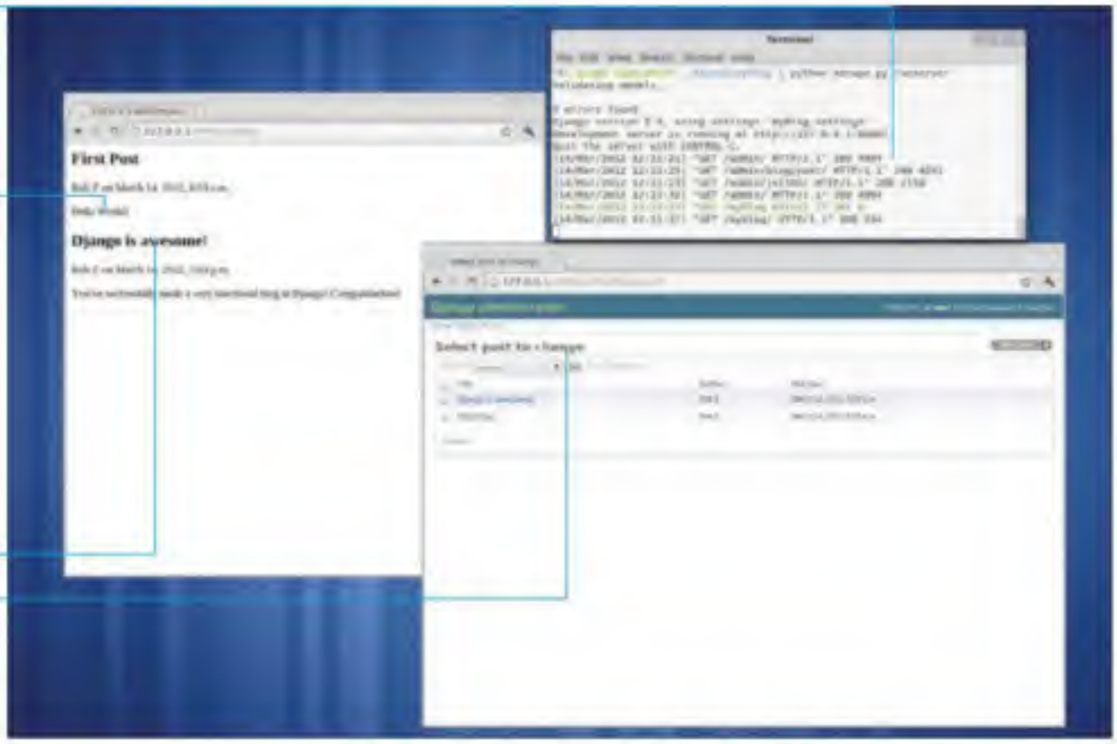
```
{% if data %}

    <ul id="tweets">
        {% for tweet in data %}
            <li>
                <div class="image">
                    <img src="{{ tweet['user']['profile_image_url_https']
}}" alt="User Profile Picture">
                </div>
                <div class="text">
                    <a>{{ tweet['text']|forceescape }}</a>
                </div>
            </li>
        {% endfor %}
    </ul>

{% else %}
  <p>We didn't get any tweets :(</p>
{% endif %}
```

Django comes with a lightweight development server so you can test all your work locally

Django is of course able to read and write to SQL databases, but it needs very little prior knowledge to succeed in doing so

Using HTML and CSS in conjunction with Django is clear and straightforward; it's much easier to bug-fix than PHP

Django comes with a generic back-end site that is set up in seconds, and easily customisable after that



Resources

Python Source Code
www.python.org/download/releases/2.7.2

Django Source Code
www.djangoproject.com/download

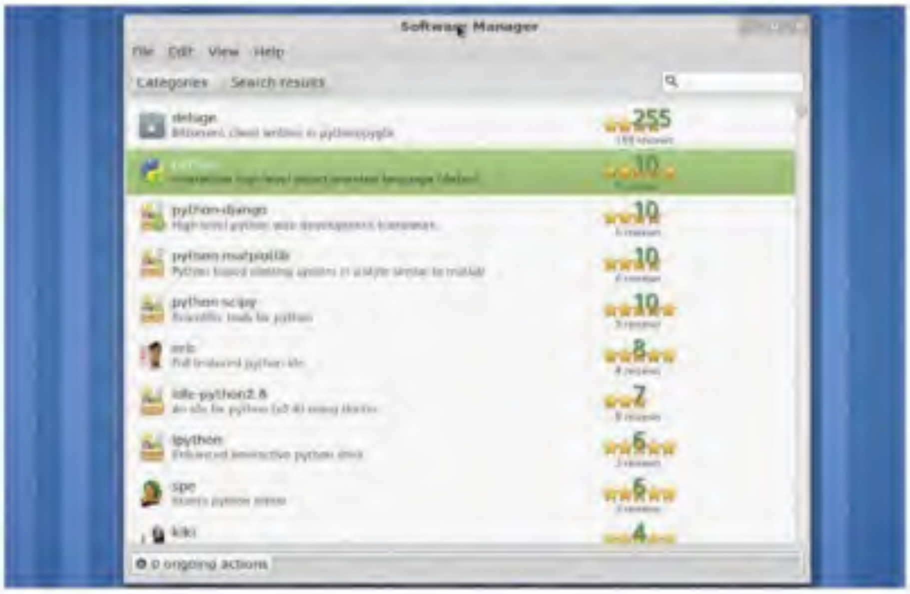
Build your own blog with Django

Learn how to use this extremely powerful Python-based web framework to create a complete blog from scratch in record time

Creating your own blog always feels like a great accomplishment. Sure, you could use the fantastic WordPress if you need a complete blog with every feature you'd ever need right now. And Tumblr exists for people who just want to write something, or post pictures of corgis in space.

You don't have full control from start to finish with a prefabricated blog, though, and neither of these is written in the fantastic Django. Django is of course based on Python, the object-orientated programming language designed to have clearly readable syntax. Due to its Python base, it's an incredibly powerful and simple-to-use language for web development with a vast array of applications.

So let's use it to make a blog. In this first section of the process we will explore how to set up Django, writing and reading to a database, creating a front- and back-end, and some interactions with HTML.



01 Install Python

Django is based on Python, and requires it to be installed to develop on. Python 2.7 is the recommended version, and this is installed with the python package. If you want to check your version, start the Python shell by typing 'python' into the terminal.

02 Install Django

Most operating systems will have a Django package available in the repository, like python-django in Debian. The Django website has a list if you have trouble finding it, or you could build it from source. Make sure you install version 1.3.

03 Verify your Django

To make sure Django installed properly, and that you have the right version, enter the Python shell by typing 'python' and **enter the following:**

```
import django
print django.get_version()
```

It will return a version number if it has installed correctly, which should be 1.3.

04 Start a new project

In the terminal, cd to the folder you want to develop the blog in, and then **run the next command:**

```
django-admin startproject myblog
```

Here, 'myblog' can be replaced by whatever you wish to name the project, but we'll use it for the upcoming examples.

05 Start the development server

Django comes with a lightweight development server to test out work locally. We can also use it to check our work, so cd to the myblog folder and **then use:**

```
python manage.py runserver
```

If all goes well, it should return zero errors. Use Ctrl+C to exit the server.

06 Configure the database

The database settings are kept in the settings.py file. Open it up with your favourite editor and go to the Databases section. **Change ENGINE to:**

```
'ENGINE': 'django.db.backends.sqlite3',
```

And in NAME, put the absolute path – for example:

```
'NAME': '/home/user/projects/myblog/sqlite.db',
```

Save and exit.

07 Create the database

The database file will be generated by **using the command:**

```
python manage.py syncdb
```

During the creation, it will ask you to set up a superuser, which you can do now.

The SQLite database file will be created in your myblog folder.

08 Create your blog

Now it's time to create a blog app in your project. **Type:**

```
python manage.py startapp blog
```

This creates the models file which is where all your data lives. You can change 'blog' to another name, but we'll use it in our examples.

09 Start your blog model

We can now take the first steps in creating our blog model. Open models.py and change it so it **says the following:**

```
from django.db import models
class Post(models.Model):
    post = models.TextField()
```

This creates the Post class, which has a subclass that contains your blog text.

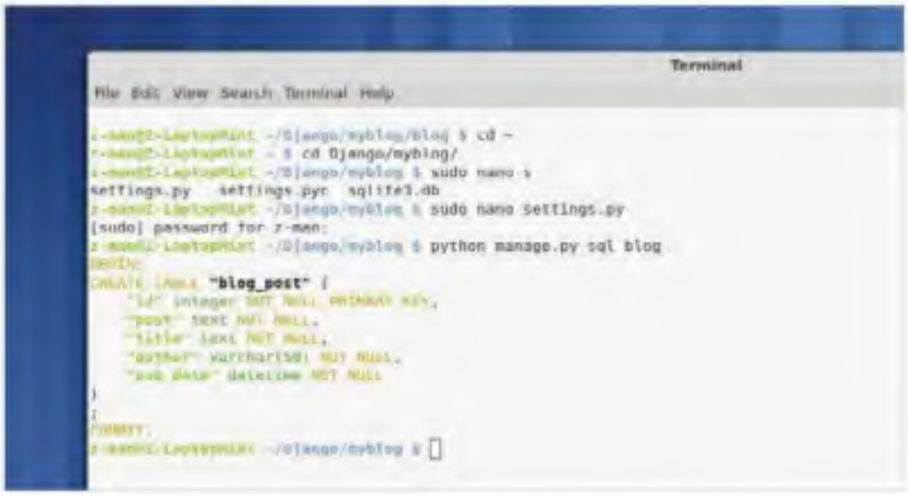
10 Customise your blog

Let's now expand the blog model a bit so it resembles a more classic blog:

```
class Post(models.Model):
    post = models.TextField()
    title = models.TextField()
    author = models.CharField(max_
length=50)
    pub_date = models.DateTimeField()
```

A CharField needs to have a character limit defined, and DateTimeField holds the time values.

> **"You don't have full control from start to finish with a prefabricated blog – but you will with Django"**



11 Install your app

Your app needs to be installed to your project, which is very simple. Open the settings.py file again, go to the INSTALLED_APPS section **and add:**

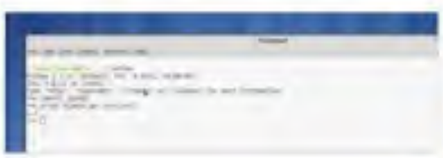
```
'blog',
```

Then run the following to create the database tables:

```
python manage.py sql blog
```

And finally:

```
python manage.py syncdb
```



12 Set up to post

Now we can create a post and test out our code. First though, **enter the Python shell:**

```
python manage.py shell
```

Then execute these commands to add all the necessary fields and data:

```
from blog.models import Post
import datetime
```

13 Let's blog

Create the post. **For this example, we will call it test_post:**

```
test_post = Post()
```

Now let's add the blog content:

```
test_post.post = 'Hello World!'
test_post.title = 'First Post'
test_post.author = 'Me'
test_post.pub_date = datetime.
datetime.now()
```

And then save it with:

```
test_post.save()
```

14 Start the site back-end

To create the admin site, edit urls.py from the myblog directory, and uncomment or **add the following lines:**

```
from django.contrib import admin
admin.autodiscover()
url(r'^admin/', include(admin.site.
urls)),
```

Save and exit, then edit settings.py and uncomment this line from INSTALLED_APPS:

```
'django.contrib.admin',
```

The admin site is now at 127.0.0.1:8000/admin/.

15 Setup the admin page

The admin page has a generic, usable template, but you need to configure it to view, edit, create and delete posts. First, create a new file admin.py in the blog directory **and enter:**

```
from blog.models import Post
from django.contrib import admin

admin.site.register(Post)
```

To have the posts display nicely on the site, edit models.py **and add:**

```
class Post (models.Model):
    …
    def __unicode__(self):
        return self.title
```

Save, and run:

```
python manage.py syncdb
```

The admin page is now usable! You should be able to see the other posts, and it's now a lot easier to add more.

16 Activate the front-end

Open up urls.py from the myblog directory in your editor and **add the following to the urlpatterns section:**

```
url(r'^myblog/', 'blog.urls.index')),
```

One of the examples in the file can be uncommented and edited to this as well. It points to a model we will now create.

17 Create another urls file

You need to create another urls file in the app directory, in our case blog/urls.py. Create it and **add the following:**

```
from django.template import Context, loader
from blog.models import Post
from django.http import HttpResponse
def index(request):
    post_list = Post.objects.all()
    t = loader.get_template('blog/index.html')
    c = Context({
        'post_list': poll_list,
    })
    return HttpResponse(t.render(c))
```

> **"Django is an incredibly powerful and simple-to-use language"**

18 Start the template

The code we've just written looks for a template that currently doesn't exist. We first need to tell Django where templates are to be looked for in settings.py:

```
TEMPLATE_DIRS = (
    '/home/user/projects/templates',
)
```

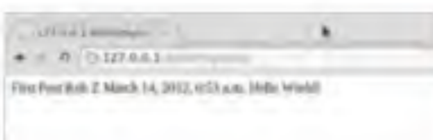
You can put the template directory wherever you want, as long as it's referenced here.

19 Write a template

Now to write the site template. In our example, we're using index.html:

```
{% for post in post_list %}
    {{ post.title }}
    {{ post.author }}
    {{ post.pub_date }}
    {{ post.post }}
{% endfor %}
```

This needs to be located in a folder with the same name as your app within the template directory.



20 View your handiwork

Let's make sure this worked. Start the developer **server with:**

```
python manage.py runserver
```

And navigate to 127.0.0.1:8000/myblog/.

It's not pretty, but you should have successfully called upon your stored data. We'll spend the next steps tidying it up a bit.

21 Format the front page

Go back into the template file, index.html, and **add the following html tags:**

```
{% for post in post_list %}
    <h2>{{ post.title }}</h2>
    {{ post.author }} on {{ post.pub_date }}
    <p>{{ post.post }}</p>
{% endfor %}
```

This is just an example – the post can be in any order with any tags.

22 Spruce up the admin list

We'll do this in the admin.py file in our blog directory; open it in your editor and make the **following changes:**

```
from blog.models import Post
from django.contrib import admin
class Admin(admin.ModelAdmin):
    list_display = ['title', 'author', 'pub_date']
admin.site.register(Post, Admin)
```

In this case 'list_display' is a fixed variable name.

23 A logical post page

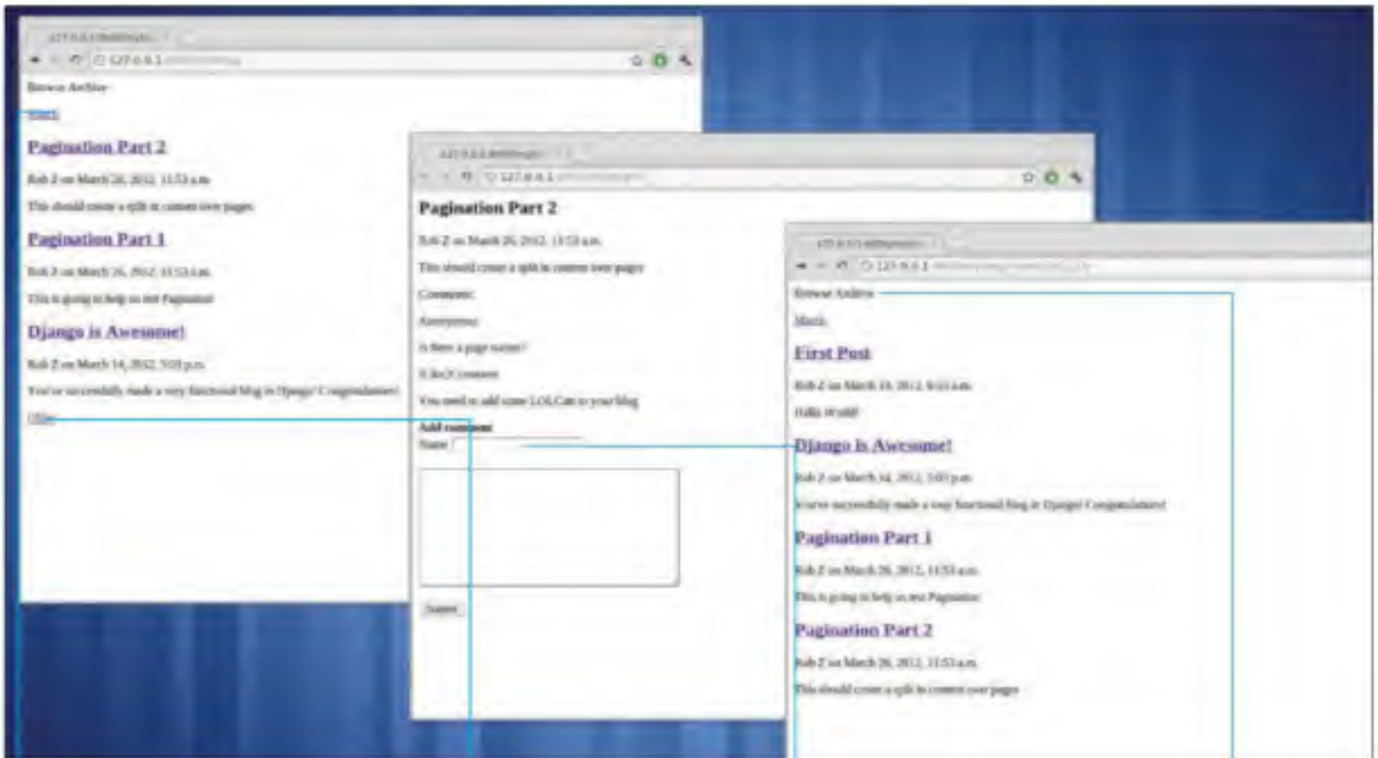
The new post page on the site might not be in an order you're comfortable with. We'll change that now in admin.py with the **following additions:**

```
class Admin(admin.ModelAdmin):
    list_display = ['title', 'author', 'pub_date']
    fields = ['title', 'pub_date', 'author', 'post']
admin.site.register(Post, Admin)
```

Remember to save!

24 A functional blog

So there you have it! Navigating to 127.0.0.1:8000/admin/ or 127.0.0.1:8000/myblog/ will show off the fine work you've created. Django is dead easy to use once you know how, and there are plenty of tweaks you should be able to make after this tutorial.



With Django we can make simple sidebars that list archives by month

Django has built-in code to deal with pagination very cleanly and effectively

Allow your readers to give you feedback, and moderate them in the admin panel

With minimal extra code, our template can display the month archive from the sidebar

Deliver content to your blog

We continue building an awesome blog using the powerful Django framework, and this tutorial is all about the front-end content delivery

Resources

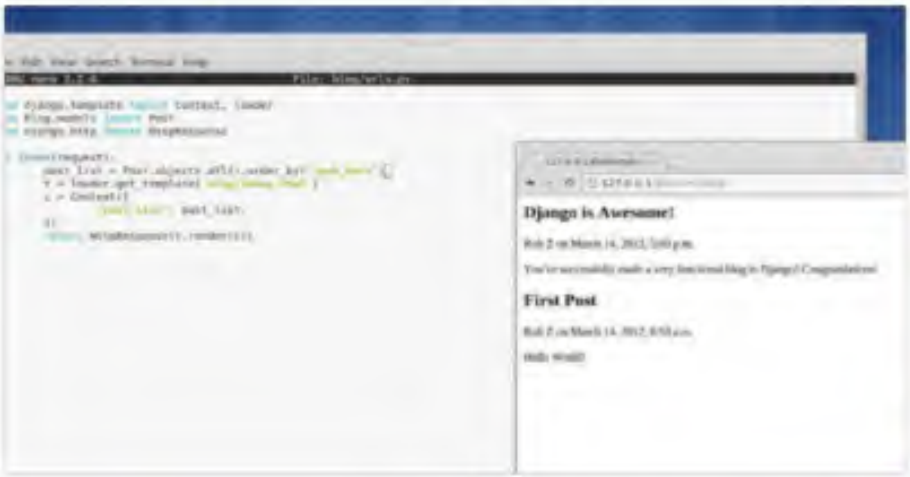
Python base: http://www.python.org/download/

Django source: https://www.djangoproject.com/download/

In the last tutorial we began to build the most basic of blogs, and learned how to use a bit of Django in the process. We can now set up a new project, create a database and write basic code to read and write to the database. All simple stuff, but of course it's core to building websites where Django might be called upon.

Here we will give the front end of the site an overhaul, making it more of the standard you would expect from a modern blog. This will include a sidebar, pages, post pages and the ability to add and moderate comments. In the process we will learn some more of the benefits that come with using Django to develop websites.

You should keep using Django 1.3 for this tutorial, as we did before.



01 New blog order

We left off last time with the blog displaying posts in chronological order, which isn't very helpful to readers. To correct this, open up urls.py in the blog folder and edit the **following line:**

```
post_list = Post.objects.all().order_
by("-pub-date")
```

This makes sure that posts are displayed in reverse order (newest first).

02 A view to a page

You'll want to be able to link specific pages, of course, and to do that we first have to define what goes into these pages in the urls.py file in the **blog folder:**

```
def post_page(request, post_id):
    post_page = Post.objects.
get(pk=post_id)
    return render_to_response('blog/
post.html', {'post_page': post_page})
```

03 Clean up your code

You may notice that we used a different return command to the index definition – this is a shortcut that makes writing the code a bit easier. To get it working, **add:**

```
from django.shortcuts import render_to_
response
```

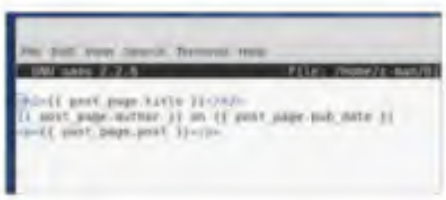
We recommend that you edit the index code to match post_page.

04 Edit URLs

In urls.py in myblog we need to make some additions and modifications for the website to direct to the **post correctly:**

```
url(r'^myblog/$', 'blog.urls.index'),
url(r'^myblog/(?P<post_id>\d+)/$',
'blog.urls.post_page'),
```

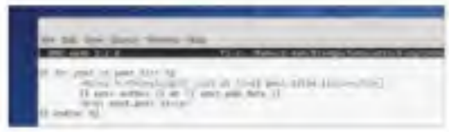
The post_id is the number of the post, which is auto-generated. The '$' is important to make the redirection work.



05 A post template

We told the post_page to point towards a template we now need to create. In the same location as index.html, create post.html with the following formatting to resemble the **front page:**

```
<h2>{{ post_page.title }}</h2>
{{ post_page.author }} on {{ post_page.
pub_date }}
<p>{{ post_page.post }}</p>
```



06 Link to the page

Let's get these links working from the main page. Open up the index.html file and make the **following change:**

```
<h2><a href=/myblog/{{ post.pk }}>{{
post.title }}</a></h2>
```

This is a very simple addition using an absolute link, and requires no fiddling with the views or model.

07 Pagination

To get blog posts split up over pages, we need to make some additions to urls.py in the **blog folder:**

```
post_list = Post.objects.all().order_
by("-pub_date")
paginator = Paginator(post_list, 3)
try: list_page = request.GET.get("list_
page", '1')
except ValueError: list_page = 1
post_list = paginator.page(list_page)
return render_to_response('blog/index.
html', {'post_list': post_list})
```

08 Please turn over

Now we need to add the navigation links to the blog, so open the index template **for editing:**

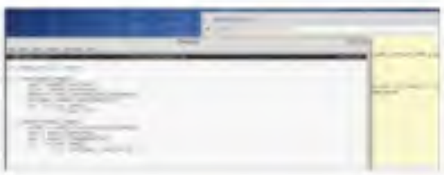
```
{% if post_list.has_previous %}
    <a href="?list_page={{ post_list.
previous_page_number }}">Newer </a>
{% endif %}
{% if post_list.has_next %}
    <a href="?list_page={{ post_list.
next_page_number }}"> Older</a>
{% endif %}
```

09 Wrong page

Let's add a quick bit of code to return somebody to the previous page if they get the URL wrong:

```
from django.core.paginator import
Paginator, EmptyPage, InvalidPage

try:
    post_list = paginator.page(list_
page)
except (EmptyPage, InvalidPage):
    post_list = paginator.
page(paginator.num_pages)
```

The last part replaces 'post_list = paginator.page(list_page)'.



10 Have your say

Everyone has their opinion on the internet. You can give your readers the ability to comment, and we'll start **by editing models.py:**

```
class Comment(models.Model):
    author = models.CharField(max_
length=50)
    text = models.TextField()
    post = models.ForeignKey(Post)
    def __unicode__(self):
          return (self.post, self.text)
```

We've made it so they can put their name with a comment.

11 Back to the comment

We now need to add a small line to the urls.py file in myblog so the comment can be posted then sent **back to the original page:**

```
url(r'^myblog/add_comment/(\d+)/$',
'blog.urls.add_comment'),
```

This URL pattern calls the ID of the page that you're on.

"We need to be able to process the data and metadata in the forms"

12 Form a comment

We need to be able to process the data and metadata in the forms, so let's add a class to urls.py in the blog folder with the **following additions:**

```
from django.forms import ModelForm
from blog.models import Post, Comment
class CommentForm(ModelForm):
    class Meta:
          model = Comment
          exclude = ['post']
```

13 In the post

We need to attribute the comments to the post they're being made on, so update the **post_page definition:**

```
from django.core.context_processors
import csrf
def post_page(request, post_id):
    post_page = Post.objects.
get(pk=post_id)
    comments = Comment.objects.
filter(post=post_page)
    d = dict(post_page=post_page,
comments=comments, form=CommentForm())
    d.update(csrf(request))
    return render_to_response('blog/
post.html', d)
```

The CSRF tag is to prevent cross-site request forgery.

14 Comment template

Let's get the post page ready for comments by **adding this to post.html:**

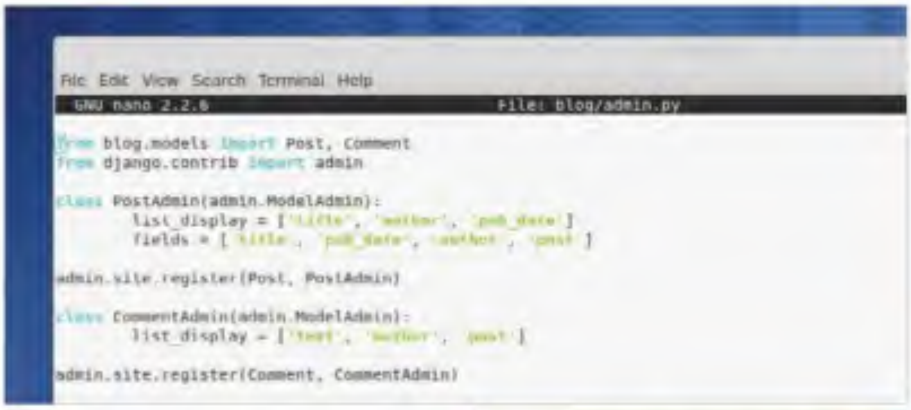
```
<p>Comments:</p>
{% for comment in comments %}
    {{ comment.author }}
    <p>{{ comment.text }}</p>
{% endfor %}
<strong>Add comment</strong>
<form action="{% url blog.urls.
add_comment post_page.id %}"
method="POST">{% csrf_token %}
    Name {{ form.author }}
    <p>{{ form.text }}</p>
    <input type="submit" value="Submit">
</form>
```

15 Define your comments

The final step is defining the comments in blog/urls.py, **and it's a big one:**

```
def add_comment(request, comment_id):
    p = request.POST
    if p.has_key('text') and p['text']:
          author = 'Anonymous'
          if p['author']: author =
p['author']
          comment = Comment(post=Post.
objects.get(pk=comment_id))
          cf = CommentForm(p,
instance=comment)
          cf.fields['author'].required =
False
          comment =
cf.save(commit=False)
          comment.author = author
          comment.save()
    return HttpResponseRedirect(reverse
('blog.urls.post_page', args=[comment_
id]))
```

This ensures text has been entered, and if not specified author is 'Anonymous'. Before testing, run syncdb so comment tables can be created.



16 Administrate

Like the posts, we can get the Admin page to see comments. Start editing blogs/admin.py to get this **feature added:**

```
from blog.models import Post, Comment
from django.contrib import admin
class PostAdmin(admin.ModelAdmin):
    list_display = ['title', 'author',
'pub_date']
    fields = ['title', 'pub_date',
'author', 'post']
admin.site.register(Post, PostAdmin)
```

17 Comment-specific admin features

Now we can add the comment-specific admin features without causing any clashes:

```
class CommentAdmin(admin.ModelAdmin):
    list_display = ['text', 'author',
'post']
admin.site.register(Comment,
CommentAdmin)
```

This will show the comments on the admin site, and you can see the comment, the author and the post it's connected to.

18 Sidebar beginnings

Django makes it pretty easy to order posts by years and months, but first we need to import some new models into blog/urls.py:

```
import time
from calendar import month_name
```

We're going to define two new functions, month_timeline and month, to make the sidebar.

19 Start to define month_timeline

First we need to get all the information from the posts:

```
def month_timeline():
    year, month = time.localtime()[:2]
    begin = Post.objects.order_by('pub_
date')[0]
    month_begin = begin.pub_date.month
    year_begin = begin.pub_date.year
    month_list = []
```

The '[:2]' makes sure we only get the time information we need.

20 Finish your definition

Now we will order the posts by month and year starting from our first month.

```
for y in range(year, year_begin-1, -1):
    start, end = 12, 0
    if y == year: start = month
    if y == year_begin: end = month_
begin-1
    for m in range(start, end, -1):
        month_list.append((y, m,
month_name[m]))
    return month_list
```

21 Return to reader

With the list organised, we can now define month so we can display it on the blog:

```
def month(request, year, month):
    post_list = Post.objects.
filter(pub_date__year=year, pub_date__
month=month)
    return render_to_response('blog/
index.html', dict(sidebar_list=post_
list, month_list=month_timeline()))
```

Now we need to link it up to the index template.

22 Finalise your sidebar definition

Edit the return command on the index function to include the sidebar information:

```
return render_to_response('blog/index.
html', dict(post_list=post_list,
sidebar_list=post_list.object_list,
month_list=month_timeline()))
```

Then add this line to urls.py in myblog so a month page can be rendered:

```
url(r'^myblog/month/(\d+)/(\d+)/$',
'blog.urls.month'),
```

All we need to do now is display the information on the site.

23 Sidebar on the web

Go to the index template. First of all, change the first line of the post forloop to:

```
{% for post in sidebar_list %}
```

Simple enough. Now we need to add the sidebar information:

```
{% for month in month_list %}
    <p><a href="{% url blog.urls.month
month.0 month.1 %}">{{ month.2 }}</a></
p>
{% endfor %}
```

24 Sidebar finale

Obviously it's not at the side right now – that's a job for the HTML and CSS. The info is there, though, and you can manipulate it any way you want. However, your blog is now a lot more friendly to your readers.

Resources

Python base: http://www.python.org/download/

Django source: https://www.djangoproject.com/download/

Enhance your blog with extra features

To add to the previous tutorials, we'll cover some of the more advanced features you can utilise with the power of Django

We've been building our Django blog to create and display posts, allow people to make comments, and filter posts by month like a classic blog sidebar. We still have a bit of a way to go until it looks and behaves more like a classic blog, though.

In this tutorial, we're going to add in summaries, excerpts, categories and finally an RSS feed. This allows us to look at a few things – firstly we'll get a better understanding of cross-model referencing and how that works in the admin site. We will also go through how to make changes to the database, and how Django helps when creating an SQL query.

Finally, the RSS feed is part of a standard feed library in Django itself. We will learn how to import and use it to create a simple list of the latest entries that click through to the posts. By the end of the tutorial your Django blog will be finally finished!

01 Summarise

On a normal blog we're going to have much longer articles. We can generate a summary of each of these on the index page template like so:

```
<p>{{ post.post|truncatewords:3 }}</p>
```

This automatically takes the first three words of the post – of course, you can use any number.

> **"We're going to add summaries, excerpts and an RSS feed"**

02 Manual excerpt

If you don't want an automatic summary, we can add an excerpt field to our post model so you can craft one manually:

```
excerpt = models.TextField()
```

To limit the characters in your excerpt, use a CharField like for our author section.

03 Write an excerpt

To write the excerpt, or append it to the previous posts, we'll have to add it to the admin page. Open up admin.py and edit the fields section of the AdminPost class to add excerpt:

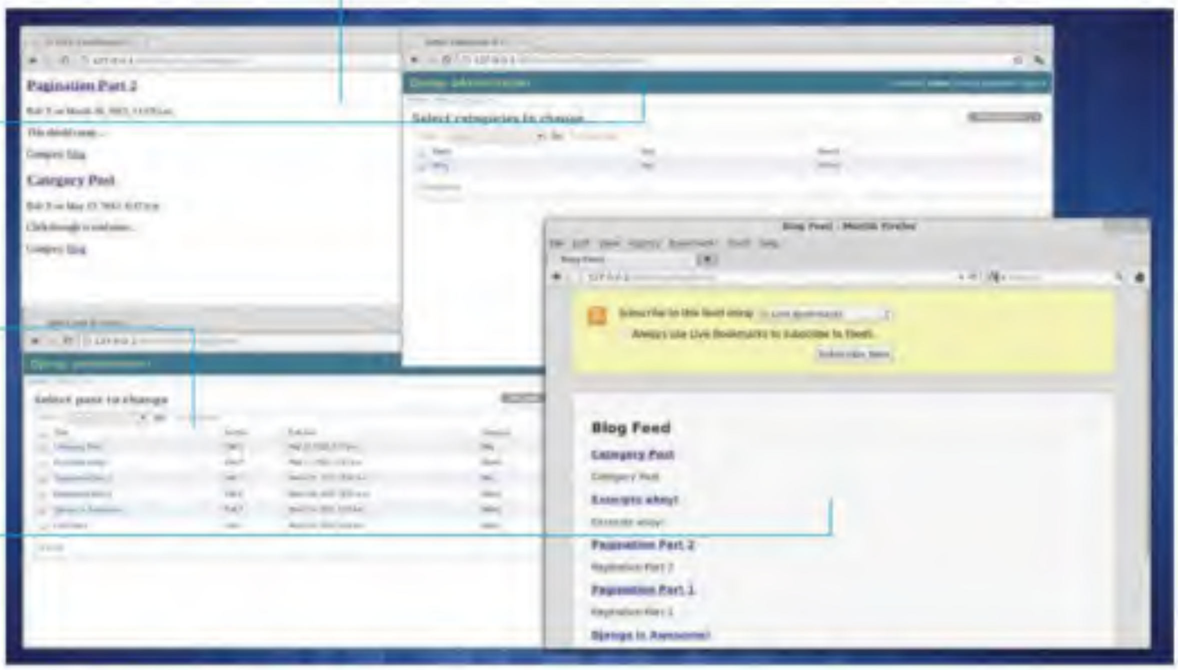
```
fields = ['title', 'pub_date', 'author', 'post', 'excerpt']
```

Have automatic summaries or manually crafted excerpts for your blog posts

Create and manage parent and child categories as a separate function of the blog

Learn how to alter the database to create posts with categories, and add them to other posts

Create custom RSS feeds using built-in Django functions



04 Excerpt or summary

You can replace the post content in the index template with the excerpt, but we can keep it as a backup for if the excerpt is empty:

```
{% if post.excerpt %} <p>{{ post.
excerpt }}</p> {% else %} <p>{{ post.
post|truncatewords:3 }}</p> {% endif %}
```

05 Database error

If you've decided to test the changes, you'll have noticed our web server has stopped working. This is because there is no excerpt column in our database. Therefore we need to add the excerpt column. To find out how, run:

```
$ python manage.py sqlall blog
```

06 Database query

The output will show you what the SQL code is to add the models to the database. We want to add the excerpt field specifically, which should look something like this:

```
"excerpt" text NOT NULL
```

Make a note of it.

07 Alter table

To get into the database shell and add the field, run: `$ python manage.py dbshell`

Then we need to use an ALTER TABLE query:

```
ALTER TABLE "blog_post".
```

And then enter the code we noted down like so:

```
ADD "excerpt" text;
```

08 Save the changes

We've removed NOT NULL as we already have entries that won't have an excerpt, and want to make it so an auto summary can be made. Save the changes with: `COMMIT;` and then exit the shell with: `.quit`

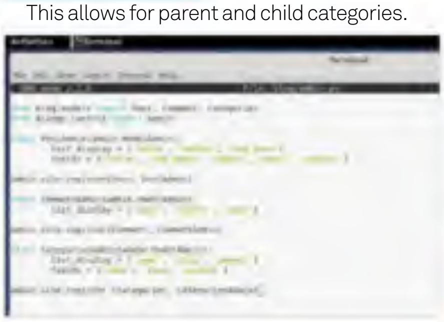
09 Test it out

Now we can test out the excerpt code – create a new post or edit an existing one to have an excerpt. If you've followed our steps correctly it should work; if not, you may need to do a bit of bug fixing.

10 Category model

We can add a model for blog categories: `class Categories(models.Model): name = models.CharField(unique=True, max_length=200) slug = models.SlugField(unique=True, max_length=100) parent = models.ForeignKey('self', blank=True, null=True, related_name='child') def __unicode__(self): return (self.name)`

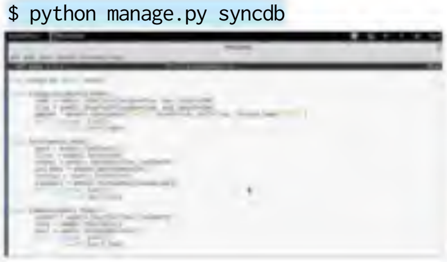
This allows for parent and child categories.



11 Administrate categories

We can add it to the admin site by creating a Categories section in admin.py: `class CategoriesAdmin(admin.ModelAdmin): list_display = ['name', 'slug', 'parent'] fields = ['name', 'slug', 'parent'] admin.site.register(Categories, CategoriesAdmin)`

Before we can make categories, though, we need to create the database table: `$ python manage.py syncdb`



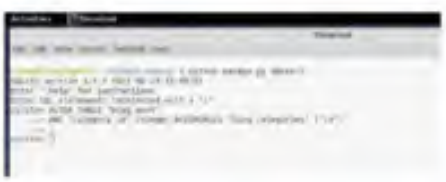
12 Categorise the posts

Similarly to what we did with the comments, we want to add a ForeignKey to the Post model so we can attribute a post to a category. Add this line: `category = models.ForeignKey(Categories)`

And move Categories to the top of models.py.

13 Database category

Like before, we'll find out the SQL needed to alter the table: `$ python manage.py sqlall blog` Which for our example returns a somewhat different code than before: `"category_id" integer NOT NULL REFERENCES "blog_categories" ("id")` It's an ID we're getting, not text, from the categories table.



14 Alter table – part 2

Again let's enter the database shell: `python manage.py dbshell` We'll continue much like before, but with the new code: `ALTER TABLE "blog_post" ADD "category_id" integer REFERENCES "blog_categories" ("id");` And finally, to save: `COMMIT;`

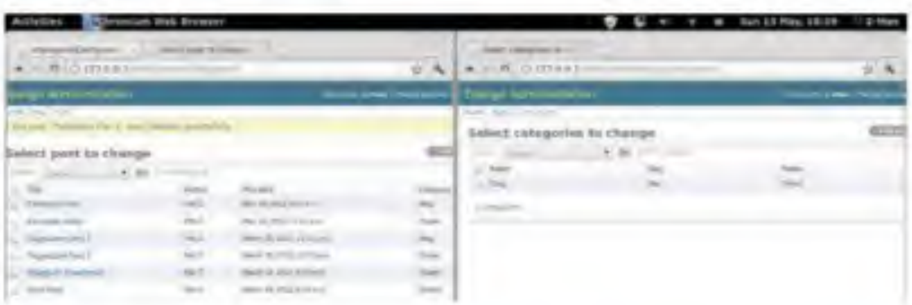
> **"We can now create categories separately"**

16 Category display

As our urls.py in the blog directory gets all the post fields, to the index template we just add: `<p>Category: {{ post.category }}</p>` And to the post template: `<p>Category: {{ post_list.category }}</p>`

17 Category page

First we need to define our category in blog/urls.py. Import Categories and then add: `def blog_categories(request, category_id): categories = Categories.objects.get(pk=category_id)` We need the category_id to call the corresponding posts.



15 Administrate categories – part 2

Now we can go back to admin.py and add the new category fields to the PostAdmin model: `list_display = ['title', 'author', 'pub_date', 'category'] fields = ['title', 'pub_date', 'author', 'post', 'excerpt', 'category']` Our previous blog posts with no category have disappeared! To fix this, go back to models.py and make this change to the Post model: `category = models.ForeignKey(Categories, blank=True, null=True)` So we can now create categories separately, assign them to posts, and view posts without a category.

18 Category definition

Finish the definition by using the parent_id to filter the correct Posts, then render the response:

```
category_posts = Post.objects.filter(category=categories) return render_to_response('blog/categories.html', dict(category_posts=category_posts, categories=categories))
```

Again we're calling a new template that we'll construct shortly.

19 Category URLs

We'll create the URL in urls.py as for the post page, only it'll give the slug of the category instead of an ID in the link:

```
url(r'^myblog/category/(?P<category_id>\d+/$', 'blog.urls.blog_categories'),
```

20 Category template

We'll use something similar to the Index and Post template to create a category page template:

```
{% for post in category_posts %} <h2><a href=/myblog/{{ post.pk }}>{{ post.title }}</a></h2> {{ post.author }} on {{ post.pub_date }} % if post.excerpt %} <p>{{ post.excerpt }}</p> {% else %} <p>{{ post.post|truncatewords:3 }}</p> {% endif %} <p>Category: {{ post.category }}</p> {% endfor %}
```

21 Category clickthrough

Finally, let's make the categories click through to the relevant page by changing the category display to be:

```
<p>Category: <a href=/myblog/category/{{ categories.pk }}>{{ post.category }}</a></p>
```

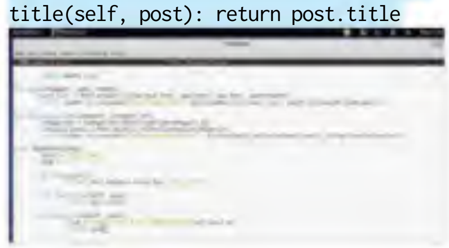
This can go on the categories, post and index template.

> **"Finally, let's make the categories click through to the relevant page"**

22 RSS

Django has a built-in RSS framework. In blog/urls.py add:

```
from django.contrib.syndication.views import Feed class BlogFeed(Feed): title = "Blog Feed" link = "/" def items(self): return Post.objects.order_by("-pub_date") def item_title(self, post): return post.title
```



23 RSS links

We need to define item_link for the feed so that the feed items can link to the right place. We have to give the complete URL and the post ID for it work:

```
def item_link(self, post): link = "http://127.0.0.1:8000/myblog/"+str(post.pk) return link
```

24 RSS URLs

The final step is adding the feed URL to urls.py:

```
url(r'^myblog/feed/$', BlogFeed()),
```

And now your blog is now fully functional. With a bit more tweaking and theming, you can get it online and blog away!

Use Python with Pi

158

168

Activate motion tracking with your Pi

Learn how to track motions with your Raspberry Pi, a camera and some Python code

By now you will know that it is possible for you to capture images using a camera and a Raspberry Pi. This lets you include image capture functionality within your own Python program, but there is so much more you can do once you add vision to your code. Here we will look at how you can add motion detection to your Python program.

This kind of advanced image processing is extremely difficult to do, so we will definitely be building on the hard work of others. Specifically, we will be using the excellent OpenCV Python package. This package is constantly being improved, with more functionality being added with every update.

The first thing you will need to do is install the various Python packages that you will need to talk to the camera and use OpenCV. Installing the packages can be done with:

```
sudo apt-get install python-picamera
python-opencv
```

This will then also automatically begin to install all of the required dependencies. This project will assume that you will use the camera module for the Raspberry Pi. Check out the boxout to the right for other options if you want to try using a USB webcam. To talk to the camera module, you need to import the PiCamera class from the picamera Python module. You will also need the PiRGBArray class so that you can store the raw data from the camera. To talk to the camera, you instantiate a new instance of the PiCamera class. You can then adjust and set the resolution and the frame rate before you begin to start capturing images.

```
from picamera import PiCamera
from picamera import PiRGBArray
camera = PiCamera()
camera.resolution = tuple([640,480])
camera.framerate = 16
rawImage = PiRGBArray(camera,
tuple([640,480]))
```

You now have your camera ready, and a memory buffer available to store the captured images in. There are several different methods that you can use to do motion tracking. One of the simpler ones is to try and notice when something within the image field changes. There is a Python module, called imutils, that provides several basic image processing functions that are useful in the pre-processing steps. There is no package for it within Raspbian, however, so you will actually need to install it with:

```
sudo pip install imutils
```

To look at image changes, we need to see what the background image looks like. You can take a series of images and look at the average of them to get an idea of the general background. Then, if a new image differs from the averaged background, we know that something has changed. This change is most probably due to something moving within the field of the image. To simplify the process, we will greyscale the image and then blur it slightly to get rid of any high-contrast regions. You will then want to simply run a continuous loop, pulling an image from the camera and running this process:

```
import imutils
import cv2
for f in camera.capture_
continuous(rawImage, format='bgr',
        use_video_port=True):
  frame = imutils.resize(f.array,
                width=500)
  gray = cv2.cvtColor(frame, cv2.
                COLOR_BGR2GRAY)
  gray = cv2.GaussianBlur(gray, (21,
                21), 0)
```

Here we start using the OpenCV functions to handle the image processing steps. You may have noticed that we are actually working with the array representation of the raw image data for the captured frame. There is no meta-data wrapping this image information, so it is your responsibility to remember what you are working with. The next step within the loop is to check whether we have an averaged image yet, and to initialise it if we don't. So the first time through the loop, then the following code will execute:

```
if avg is None:
  avg = gray.copy().astype("float")
  rawImage.truncate(0)
  continue
```

Now that we have an averaged image, we can add every subsequent captured image to the weighted average. We also need to find how different the current image is from this weighted average.

```
cv2.accumulateWeighted(gray, avg, 0.5)
imgDiff = cv2.absdiff(gray, cv2.
        convertScaleAbs(avg))
```

By using this weighted average, we should be able to deal with false positive hits due to environment changes like fluctuations in the lighting. Now that you have what is different from the average, what can you do with it? How do you decide how different it is from the average? We need to set some threshold difference that signifies a "real" difference in the image from the average. If you then dilate this thresholded image, you will then find that you can then go on to apply the findContours function to identify the contours of the objects that are different from the calculated averaged background:

```
imgThresh = cv2.threshold(imgDiff, 5,
        255, cv2.THRESH_BINARY)[1]
imgThresh = cv2.dilate(imgThresh,
        None, iterations=2)
(conts, _) = cv2.findContours
(imgThresh.copy(),      cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
```

What about webcams?

In the main article, we have been using the Raspberry Pi module that plugs into the IO bus of the Pi. But what if you don't have easy access to one of these? It can be said that almost everyone has an old webcam sitting around the house somewhere, and the Raspberry Pi has a perfectly useful USB port. The image quality and frame per second count is not as good as what you can get with the actual Pi Module. The key is getting the image data off the camera in the format that the OpenCV image analysis functions is expecting. The VideoCapture() function can not only take a video file name to read in, but can also take device IDs for cameras attached to the Raspberry Pi. Assuming that you only have one camera attached, you can connect to it with:

```
camera = cv2.VideoCapture(0)
```

Making sure that your USB webcam is correctly connected and that Linux can properly talk to it is always where you may run into issues. But, if everything works the way it should, you can use the ideas from the main body of the article to use it for motion detection. While You may want to use some other framework to handle this, a good framework that is also very fast is pygame. You can use OpenCV to handle all of the image processing steps and build your user interface with pygame. The only issue is that the internal formats used by OpenCV and pygame to store image data are different, so you will need to do a translation back and forth. You only really need to worry about translating from OpenCV to pygame, since that is the direction that information will flow. There are a few helper functions in circulation that you can use to convert the OpenCV image to a string format, and then a pygame function to import this string into a pygame image. You could use something like:

```
pygameImg = pygame.image.frombuffer(cv2Img.
tostring(), cv2Img.shape[1::-1], "RGB")
```

This takes images from OpenCV (stored in cv2Img) into a pygame format (stored in pygameImg). If you have to, you can even do a similar transformation using strings back from pygame to OpenCV format.

This dumps all of the contours from the current image into the list 'conts'. You probably aren't very interested in tiny objects within the list of contours. These might simply be artifacts within the image data. You should loop through each of these and ignore any that are below some area limit. You probably want to highlight any remaining object contours by placing a bounding box around them. Luckily, OpenCV provides a function that will give the corner coordinates and the width and height. You can then draw a box on the image using this information:

```
for c in conts:
  if cv2.contourArea(c) < 5000:
     continue
  (x, y, w, h) = cv2.boundingRect(c)
  cv2.rectangle(frame, (x, y), (x+w,
             y+h), (0, 255, 0), 2)
```

You should now have an image with all of the moving objects highlighted by red bounding boxes. What can you do with these annotated images, though? If you have a graphical environment available, you can display these results directly on the screen. OpenCV includes several functions to display the results of your image analysis. The simplest is to use the imshow(), which will pop up a window to display the image and also add a title.

```
cv2.imshow("Motion detected", frame)
```

If you aren't monitoring the results of your motion detector in real time, you probably still want to capture images when something moves in the environment. Luckily, OpenCV also includes a pretty exhaustive list of IO functions. You will probably want to timestamp these images first, though. Using the Python module timestamp and the function "putText()", you can get the current time and date and add it to the image itself with:

```
import timestamp
ts = timestamp.strftime("%A %d %B %Y
%I:%M:%S%p")
cv2.putText(frame, ts, (10, frame.
shape[0] - 10), cv2.FONT_HERSHEY_SIMPLEX,
0.35, (0, 0, 255), 1)
```

Now you have an image with the current time and date on it, and the parts of the image that show up as having movement bounded in red boxes. You can use the OpenCV IO functions to write out these images so that you can check them out later. The following code is an example:

```
cv2.imwrite("filename.jpg", frame)
```

The function imwrite() uses the file name extension in order to figure out what format to use when writing out the image. It can handle JPEG, PNG, PBM, PGM, PPM and TIFF. If the particular format you want to use also takes options, you can include them in the call to imwrite() as well. For example, you can set the JPEG quality by including CV_IMWRITE_JPEG_QUALITY and then setting it to some value between 0 and 100. Everything we have looked at has been focused on the idea of analysing the images in real time, and this is great if you can put the Raspberry Pi in the same location as the camera. If you can't fit it in, though, you can still use the ideas here to post-process the video recorded by your micro-camera. You can use the same OpenCV IO functions to load the video file with:

```
camera = cv2.VideoCapture("filename.
avi")
```

You can then run through the same process to analyse each of the image frames within the video file. The VideoCapture() function can also read in a series of image files if your camera is simply grabbing a series of still images rather than a video. Once your program finishes, you need to remember to clean up after yourself. You should release the camera that you were using, and if you had OpenCV display any images on the desktop then you should clean those up, too.

```
camera.release()
cv2.destroyAllWindows()
```

After completely this you should now have enough information now to be able to add some basic motion detection to your own Python programs. If you explore the OpenCV documentation, you will find many other, more complex, image processing and analysing tools that are available to play with. Also, more functionality is constantly being added to the OpenCV project.

Handle OpenGL with Pi

The Raspberry Pi includes a basic GPU. Learn how to use it to generate 3D graphics on board

Many of the examples we have looked at in this column have been programs that run on the console. Here we will be taking a look at how you can include 3D graphics within your Python code. The hardware on the board includes a GPU, so you should be able to take advantage of everything this computer is capable of. Since we are going to be playing with graphics, this means that you will probably want to have an actual monitor connected to your Raspberry Pi. If you don't have one, you can connect to your Raspberry Pi over SSH with X11 forwarding turned on. As long as you have an X11 server on your desktop, you can see the display on your local machine. To get access to OpenGL from Python, you will need to install the correct package, with:

```
sudo apt-get install python-opengl
```

This is assuming that you are using a Debian-based distribution, such as Raspbian. It will also install the OpenGL libraries that actually talk to the GPU and handle the 3D operations. For those who don't already know, OpenGL is the open source, multi-platform library to handle 3D graphics. For those of you coming from the Microsoft world, it is essentially Direct3D for everything non-Microsoft. All of the actual functionality is encoded within libraries written in C, while python-opengl provides a set of Python wrappers allowing you to access these libraries from your code.

The first step in trying to use OpenGL in Python is to import all of the code that you will need. Since OpenGL can be a bit messy and convoluted, there are several categories of helper functions available. These are grouped under the GLU and GLUT sub-modules. The imports that will likely be part of your boilerplate are:

```
from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
```

All of the functions that you will likely ever need should be imported to your namespace. One of the really useful functions that GLUT provides is the ability to create windows and to draw within them. In order to use it, you need to initialise the subsystem first:

```
glutInit()
```

```
glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_ALPHA | GLUT_DEPTH)
```

The first thing you will need is a window to draw within. Windows need a size and a position to define where they should be drawn on the physical screen. A basic window would be defined by:

```
glutInitWindowSize(500, 400)
glutInitWindowPosition(0, 0)
```

The last step is to actually create the window. This then creates a new window object and displays it on the display device in use. You can do this with:

```
window1 = glutCreateWindow('window1')
```

The newly created window will have a title on the main window bar defined by the string handed in as an option. At this point we have a window, but there is nothing in it yet. The next step is to tell OpenGL what you want to draw on this canvas. You can do this by setting a callback function that OpenGL can use to draw with. A boilerplate of a draw function looks like:

How to handle 3D on OpenGL

"But, what about 3D?" we hear you ask. Luckily, we can reuse much of what we have already covered here to move into three dimensions. The first change is that vertices now are defined with three values rather than two, giving you a set of x, y and z coordinates. The second change is in the category of functions that you need to use. You can see that the drawing functions used are explicitly two-dimensional in their names. For example, you should have noticed the '2' in names like 'glVertex2f()'. To draw three-dimensionally, this changes to names like 'glVertex3fv()'. The third change is more conceptual. Once we move to three dimensions, we need to worry about where we are in relation to the object that is being drawn, and what the field of view and depth of clipping panes need to be. The essential definitions can be set with:

```
gluPerspective(45, (width/height), 0.1, 50.0)
```

The first parameter is the field of view in degrees, the second parameter is the aspect ratio, and the last two parameters are the near and far clipping panes. OpenGL needs this to figure out how to project your three-dimensional object onto a two-dimensional surface (the screen) so that it looks correct. You start off located right next to the object, which is too close to see it clearly. Luckily, you can move relatively easily. There is a function that can apply a transformation matrix to objects within OpenGL. In order to move back, say five units, you would use:

```
glTranslatef(0.0, 0.0, -5)
```

There are also other types of transformations, like rotations, which can be applied within OpenGL. If we extend the example rectangle to a cuboid, we could draw it the same way by going from vertex to vertex. We can then apply a rotation to it with:

```
glRotatef(1, 3, 1, 1)
```

This takes an angle and x, y and z coordinates and rotates the matrix by that much. You can then clear the screen and redraw the rotated cube.

```
def draw1():
    glClear(GL_COLOR_BUFER_BIT | GL_DEPTH_BUFFER_BIT)
    glLoadIdentity()
    # Other drawing
    glutSwapBuffers()
```

The first line blanks out the contents of the window. The second line resets the position of the window, ready for further drawing. The commented line is where you would place any other drawing commands needed by the draw1 function. The last line is important for double-buffering. The technique of double-buffering is where you draw on an in-memory buffer region, and then dump the entire contents of this region to the physical display device. This is done to get the fastest possible drawing speed. Once you have your function defined, you can then set it as the callback function with the GLUT function:

```
glutDisplayFunc(draw1)
```

The last step is to tell OpenGL to start drawing. You can do this with:

```
glutIdleFunc(draw1)
glutMainLoop()
```

The first line tells OpenGL to run the draw1 function whenever the system is otherwise idle, In this case, this means all the time. The second line starts the processing loop for the OpenGL process. You now have a black window, which is rather boring.

Now we need to actually start drawing within this new window. In OpenGL, drawing simple objects is done by using vertices and connecting lines between them. As a simple example, we will look at drawing a basic rectangle. Any drawing commands need to be bracketed by a begin command and an end command to tell OpenGL what functions need to be paid attention to. This looks like:

```
glBegin(GL_QUADS)
# drawing commands
glEnd()
```

The actual drawing commands consist of telling OpenGL where the vertices are located within the window. For this simple case, the coordinate system of the window starts at (0, 0) in the bottom left corner. The x-coordinate increases towards the right, and the y-coordinate increase towards the top. If we have the coordinates of the bottom-left corner of the rectangle in the variables x and y, with a width and a height defined, we can set the vertices with the code:

```
glVertex2f(x, y)
glVertex2f(x + width, y)
glVertex2f(x + width, y + height)
glVertex2f(x, y + height)
```

Notice that you need to define the vertices in order, going around your rectangle. You probably also want to draw your rectangle is some other colour than the black background of the window. You can do this by changing the colour used, just before you start drawing. The command is:

```
glColor3f(0.0, 0.0, 1.0)
```

The parameters are floating point numbers, from 0.0 to 1.0, to define how much red, green and blue to use in the colour. So the above command will give you a bright blue rectangle. If you were to run this code as is, however, you still won't see your rectangle; this is because we haven't told OpenGL that we are drawing in 2D yet. A function that contains all of the required steps looks like:

```
def prepare2d(width, height):
  glViewport(0, 0, width, height)
  glMatrixMode(GL_PROJECTION)
  glLoadIdentity()
  glOrtho(0.0, width, 0.0, height, 0.0, 1.0)
  glMatrixMode(GL_MODELVIEW)
  glLoadIdentity()
```

Digging into the details of what these functions are and why you need them would probably fill another full article. For now, consider these functions as further reading for the student.

> **"The actual drawing commands consist of telling OpenGL where the vertices are located within the window"**

Turn your Raspberry Pi into a stop-motion studio

Build your own animation studio by using your Raspberry Pi as a stop-motion camera

Resources

Hard drive

OSMC:
osmc.tv/

Home network

Another Linux computer, less than eight years old

What have you done with your Raspberry Pi camera lately? While it gives us plenty of new ways to use the Pi, unless you've got your computer set up as a security webcam or you're particularly a fan of time-lapse photography, the chances are that you've overlooked the Pi camera module for a while.

If you're a fan of animation or you simply want to extend the possibilities of the module, why not build a stop-motion camera? By using Python and an external button to capture images, the Raspberry Pi can actually be the perfect tool for animators.

Better still, you can go beyond animating toys or bits of LEGO and go old school by mounting the Pi on a rostrum and creating a cartoon. Even if you can't buy or build one, you can mount the stop motion Pi camera with a smartphone mount for stability.

01 Mount your stop-motion Pi camera

Before you get started, think about the type of animation you're going to be capturing. If you're using the traditional top-down method, as used by classic cartoon animators, then you'll need a rostrum to mount the Raspberry Pi.

Alternatively, you may be animating something on a desk, table or perhaps the floor, but you'll need your Pi camera mounted in a similar way, looking across rather than looking down.

Various options are available, such as smartphone tripods and dashboard mounts. Most of these should be suitable for securely mounting your Raspberry Pi.

02 Find somewhere to shoot

For your first attempts at shooting a stop-motion video, you should use a wide and uncluttered space. This might be a desk, a kitchen work surface or even the floor, but it should be a hard and flat area in most cases (unless you have need for a bumpy carpeted environment for your video) to aid with the creation of your stop-motion film.

As time progresses and your skill develops, other surfaces can prove useful alternatives, but keep it simple for now and stick with flat surfaces while you get to grips with the art form using the Raspberry Pi stop-motion camera.

03 Connect the Pi camera module

Next you'll need to connect the Pi camera module to your Raspberry Pi. All models have the necessary connector, although where it is found on the device will depend on the version of your Raspberry Pi.

Below Our home-made antenna may look a little rough around the edges, but it works great!

The Model A has the Pi-camera connector next to the Ethernet port, as does the Model B. On the B+ and the Raspberry Pi 2, the connector is in a similar position, but it's a little further from the Ethernet port between the audio-out and HDMI ports.

Connecting the camera module can be tricky. Begin with taking your Pi out of its case or remove the top where possible and disconnect all cables. Take precautions before removing the device from its antistatic bag, as the camera module is very sensitive to static electricity.

On the Pi, lift the plastic catch on the connector and slot the camera module flex into place with the shiny contacts facing away from the Ethernet port. Once the flex is fully slotted in, push the plastic catch back into place.

04 Test your Pi camera module

After connecting the Pi camera, check that it works by booting the Raspberry Pi (we're assuming you're running Raspbian) and entering this in the command line:

```
sudo raspi-config
```

With the keyboard arrows, move down to option five, 'Enable Camera', and tap Enter. In the following screen, hit Enter again to enable the camera and exit. If you're not already signed into the GUI, do so now (if you're in the command line interface, enter **startx** to launch the desktop view). Open the terminal and enter:

```
raspistill -o image1.jpg
```

You can review the resulting image in your Home directory.

05 Straighten out the image

With the Pi camera up and running, you may notice that it's outputting the image with the axes flipped. We can fix this using Python, so open the terminal and enter:

```
sudo apt-get install python-picamera python3-
picamera
sudo idle3
```

In the Python editor, open File>New Window and enter the code below, setting the camera.vflip and camera.hflip as True or False as required. Save (perhaps as 'camflip.py'), then press F5 to run the script and view the correctly outputted image.

To save time, however, you might try rotating the position of your camera or Pi camera module!

```
import picamera
from time import sleep

with picamera.PiCamera() as camera:
    camera.vflip = True
    camera.hflip = True
    camera.start_preview()
    sleep(3)
    camera.capture('/home/pi/image2.jpg')
    camera.stop_preview()
```

06 Set up the breadboard and button

We have two ways to add a button to the Raspberry Pi, but before proceeding, ensure you have switched the computer off and disconnected it from the mains. You should also disconnect any cables and hardware.

The simplest method of adding a button is to employ a solder-free breadboard and a single-state pushbutton. Connect the button to the breadboard with two male-to-female wires running to GPIO pins GND and 17. With a script designed to detect action from the button on the GPIO, each frame of your animation can be captured with a single button push.

Left Consider the angle you'll be shooting from as you are setting up

Right With the camera module, ensure the shiny side faces away from the Ethernet port

"Don't want to build your own rostrum? Why bother when a camera tripod can be positioned as needed?"

07 Code for stop motion

Once satisfied with the results of your Pi camera, it's time to turn it into a stop-motion camera. The first step is to type up the code shown below, which will capture an image of your subject and save it into a folder called 'Stop motion'. Each image is numbered sequentially and they can all be stitched together once your animation is complete. Save the code as animation.py:

```
import picamera
from RPi import GPIO

button = 17

GPIO.setmode(GPIO.BCM)
GPIO.setup(button, GPIO.IN, GPIO.PUD_UP)

with picamera.PiCamera() as camera:
    camera.start_preview()
    frame = 1
    while True:
        GPIO.wait_for_edge(button, GPIO.
FALLING)
        camera.capture('/home/pi/animation/
frame%03d.
                                jpg' % frame)
        frame += 1
    camera.stop_preview()
```

Then, in a new terminal window, enter the following:

```
sudo python3 animation.py
```

Press the button to capture each frame, moving the subject as needed. When you're all done, hit Ctrl+C to terminate the script.

08 Stitch together your stop-motion animation

For true animation the collected images will need to be compiled into one single file. In the terminal, install ffmpeg:

```
sudo apt-get install ffmpeg
```

Once installed, you will then need to convert your images into a video clip, as follows:

```
ffmpeg -y -f image2 -i /home/pi/Desktop/stop-
motion/frame%03d.jpg -r 24 -vcodec libx264 -profile
high -preset slow /home/pi/Desktop/stop-motion/
animation.mp4
```

With this file created, open with the command:

```
omxplayer animation.mp4
```

The video will then be played in full-screen mode.

09 Use an app instead

Don't fancy using the script? Try this stop-motion application. Begin by installing the raspicam-extras package that includes the UB4L drives for the Pi:

```
wget http://www.linux-projects.org/listing/uv4l_
repo/lrkey.asc && sudo apt-key add ./lrkey.asc
sudo sh -c 'echo "deb http://www.linux-projects.
org/listing/uv4l_repo/raspbian/ wheezy main" >> /
etc/apt/sources.list'
sudo apt-get update
sudo apt-get install uv4l uv4l-raspicam uv4l-
raspicam-extras
```

With that done, enter:

```
sudo apt-get install stopmotion
```

Launch with the **stopmotion** command to open a GUI with a live camera for you to line up each shot. This is a more elegant solution and captured images can be stitched together using the 'Number of images' slider and the camera button above it.

Tripods and suction holders

Don't want to build your own rostrum? Why bother when a camera tripod can be positioned as needed and other items, like smartphone suction holders and grips, can be employed to hold your Raspberry Pi case and camera module in place?

For top-down animation, suction-pad smartphone holders (available for under £10) that use a sticky gel for a stronger grip are perfect for holding your stop-motion Pi camera and attaching to a flat surface above the animation subject.

Above Here's the stopmotion program in action – it's a simple enough GUI to get your head around and gives you a nice preview window

10 Put it all together

Now you have the camera set up, a device for keeping it steady (whether a DIY rostrum or a tripod), and you've constructed a button or plan to capture each frame via SSH. Your stop-motion Raspberry Pi camera is finally ready!

By now you're probably aching to get started, so with your stop-motion Pi camera ready to use (and close to a power supply), it's time to start building your film set. While this might simply be an empty table top, there might equally be a few props you would like to include.

11 Storyboard your shoot

It's easy to get tied up with the idea of creating a stop-motion camera and forget all about a subject and how it will act.

You can avoid any problems here by taking the time to carefully plan what will happen in your film: your story. Remember, each second of the video will require 26 frames!

The best way to plan at this level is to simply write up an outline, but beyond this you may prefer to storyboard instead by making pencil sketches to help you progress the story.

12 Cast your stop-motion shoot

You'll also need a good idea of what your subject will be; this means who or what you're going to be using the stop-motion camera to capture frames of. Typically, amateur stop-motion films make use of household objects, toys and child's play clay.

The beauty of this kind of animation is that you can use almost anything that you can get your hands on, from a cup and saucer to an Action Man, as long as you have a way to support the subject(s) in the positions you wish them to take throughout.

13 Stop-motion with toys

If you cast toys as your stop-motion stars, you will get a much better result from something that is built to stand up on its own than toys that tend to sit or fall over.

LEGO sets and Minifigs appear in many stop-motion productions on YouTube. This is with good reason, as they're really easy to place in a desired position. The construction element of the bricks is also a major attraction. Another popular option is Transformers toys. These are both good places to start, but you should aim to develop your own approach over time.

14 People in stop-motion films

It isn't only inanimate objects that you can include in stop-motion films. People can feature too! Pop videos such as Peter Gabriel's 1985 hit *Sledgehammer* have taken advantage of stop motion (that video was produced by Aardman Animations, the eventual creators of *Wallace and Gromit*) and the technique can be used on humans to create surreal effects. If you want your subject to be moving around a room too, they can appear to be floating or gliding. The results can be strange, but useful if you know what you want.

15 Make your own *Wallace and Gromit*

Known as 'claymation', the practice of animating lumps of clay has been a popular form of animation for years in the UK, but there's more to it than just clay. These forms, whether they're cheese-loving old men or remarkably clever dogs, have a wire skeleton that is used to keep movement in the desired position.

This makes it much easier to capture the frames efficiently, but for the best results you should also have several versions of the same figures available. This is just in case one gets deformed and damaged during production!

16 From stop motion to time lapse

Similar to stop motion, time lapse is a technique that automatically captures images on a preset timer. We can use a Python script to control this, saving the captures in a directory and using ffmpeg to compile them into a film.

However, what you may not want for this project is a mains cable trailing all over, especially if you're attempting to capture the movement of the stars at night or nature activity. We suggest employing a Pi-compatible battery pack to make your time-lapse Pi camera truly mobile, using SSH to run the script remotely:

```
import time
import picamera

VIDEO_DAYS = 1
FRAMES_PER_HOUR = 60
FRAMES = FRAMES_PER_HOUR * 24 * VIDEO_DAYS

def capture_frame(frame):
  with picamera.PiCamera() as cam:
    time.sleep(2)
    cam.capture(‘/home/pi/Desktop/frame%03d.jpg’ % frame)

# Capture the images
for frame in range(FRAMES):

  # Note the time before the capture
  start = time.time()
  capture_frame(frame)

  # Wait for the next capture. Note that we take into
  # account the length of time it took to capture the
  # image when calculating the delay
  time.sleep(
    int(60 * 60 / FRAMES_PER_HOUR) - (time.time() - start)
)
```

17 Take your stop-motion studio to the next level

At the risk of encouraging you to become the next Ivor Wood (creator of *The Wombles*, *Paddington* and *Postman Pat*, among others), it is possible to use the Raspberry Pi's camera module for ambitious projects as well as small ones. After all, this device photographs in high resolution so there is no reason not to adopt this setup and incorporate it into a working stop-motion studio with a miniature set.

Sharing your work through YouTube is a great idea too, especially as it will make it simple to add a soundtrack using YouTube's browser-based editor.

Send an SMS from your Raspberry Pi

Create a program that combines Twilio and simple Python code to enable you to send an SMS (text message) from your Pi to a mobile phone

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

Resources

Raspberry Pi

Twilio account

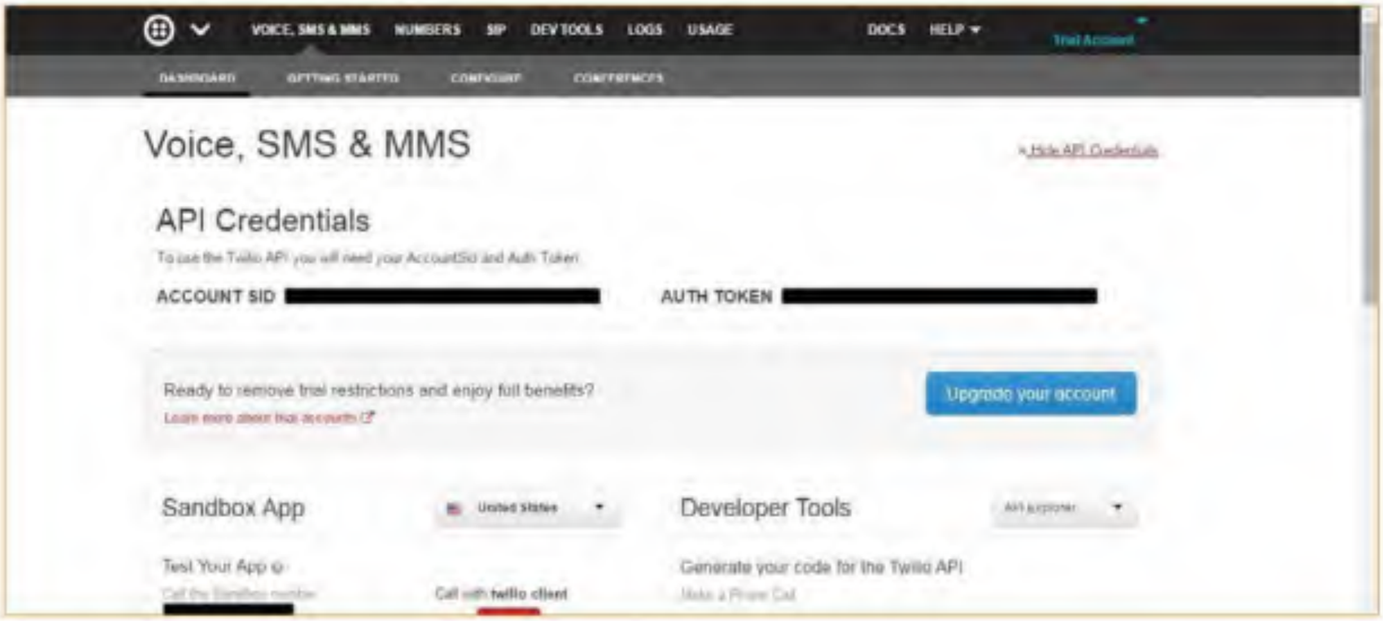
Text messaging, or SMS (Short Message Service), has become a staple of everyday communication. What began life as a 40 pence message service is now offered by most tariff providers as an unlimited service. Twilio, a cloud communications company, enables you to send SMS messages for free from your Raspberry Pi to a mobile phone using just six lines of code.

01 Set up your Twilio account

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.

02 Register and verify mobile numbers

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.



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

03 The dashboard

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.

04 Install the software

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**.)

05 Twilio authentication

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 = "XXXXXXXXXXXXXXXXXXXX"
# Enter Yours
auth_token = "XXXXXXXXXXXXXXXXXXXX"
# Enter Yours
client = TwilioRestClient(account_sid, auth_
token)
```

06 Create your message

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")
```

> **"Twilio enables you to send SMS messages for free"**

07 Add your numbers

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)
```

08 Send the message

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!"
```

09 Other API and codes

Twilio provides a wide range of API codes and reference documents to create other communication programs, such as making phone calls, recording a call, and retrieving data including caller IDs and call duration. The API also complements a wide range of languages, including Ruby, PHP, Java and Node.js (**twilio.com/api**).

Code on FileSilo

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.

Use Pi to recognise Faces

Wouldn't it be nice to have your Raspberry Pi be able to keep a look out for persons of interest?

In a previous piece, we looked at how you could use your Raspberry Pi and some Python code to look at a scene and figure out whether there was any motion there. Now we will take it one step further and see if we can get our Raspberry Pis to actually recognise faces.

We'll assume here that you are either using the camera module for the Raspberry Pi, or a USB web cam. In either case, you have a stream of images being handed to your code, ready to be processed. Again, as in previous articles, we will be using the OpenCV Python module to handle all of the heavy image processing involved. The first step is to be sure you have Python and the OpenCV module installed with:

```
sudo apt-get install python
python-opencv
```

To identify a face within an image, your code needs to check each sub-region of the image for the features of a face. To successfully identify a face, you may need to check as many as 600 features, or even more. To do this for each sub-region of the image, the amount of work involved would quickly bring any machine to dead stop. OpenCV gets around this problem by using a technique called a cascade. For each sub-region, OpenCV does some simple, quick, general checks to see if this particular sub-region contains something that might be a face.

If these initial tests are positive, then it will cascade down to more detailed tests to verify this initial conclusion. This also means that OpenCV can quickly decide that some sub-regions does not contain a face and can save time and skip the more detailed tests. This technique brings facial recognition to real-time speeds. In order to actually do these cascades, you need to supply a Haar cascade file. This file is an XML file that contains the details from a training session where many images were processed to figure out how best to do the cascading tests. While you can generate these files yourself through a training process, there are a number of files available which will identify many common targets. You can get them with the command:

```
sudo apt-get install opencv-data
```

This will install a number of these cascade files within the directory /usr/share/opencv/haarcascades.

Now that we have everything ready, we can look at how to identify faces within images. To simplify things, we will start by assuming that we have a static image in the file 'faces.jpg' and that we are going to try and process it. The first step is to create a new cascade classifier constructed around one of the Haar cascade files. Assuming that you are only interested in forward facing faces, you could use code like:

```
import cv2 face_cascade = cv2.
CascadeClassifier('/usr/share/opencv/
haarcascades/haarcascade_frontalface_
default.xml')
```

One thing to be aware of is that 'cv2.CascadeClassifier()' will not complain if the filename handed doesn't exist. So be careful of typos. The next step is to load the image to be processed. Since OpenCV does most processing in greyscale, you will also need to convert the image if it is in colour. You could use something along the lines of:

```
face_img = cv2.imread('faces.jpg')
gray_img = cv2.cvtColor(face_img, cv2.
COLOR_BGR2GRAY)
```

You can now get OpenCV to start looking for faces. A good starting point could be using the function below:

```
faces = face_cascade.
detectMultiScale(gray_img,scaleFactor=1.1,
minNeighbors=5, minSize=(30,30),
flags=cv2.cv.CV_HAAR_SCALE_IMAGE)
```

This will give you a list of objects defining where OpenCV has found what looks like a face. Something to keep in mind is that these algorithms are performed using machine learning, so they are not 100 per cent correct. You will likely get false positives, as well as missed faces. Depending on the quality of the images being processed, you will need to change the parameters of the 'detectMultiScale()' function to match your situation. There are five in the above example, but you could use up to as many as nine parameters.

In the above example, the scaleFactor tries to account for the fact that the subject in the image may be closer or farther away than those used in training the classifier. The minNeighbors defines how many nearest objects are detected around the currently analysed one before the classifier decides that it has actually found a face. The minSize parameter defines the window size used in the cascade classifier. The returned objects are 4-tuples that contain the x and y coordinates of a bounding rectangle, along with the width and height of this rectangle.

Now that we have the basics of detecting a face, we need to move on and see how to do this on a stream of images, such as though coming from a camera. Depending on the camera, you might be able to use the video capture functionality within OpenCV itself. For example, you could use something like:

```
video_capture = cv2.VideoCapture(0)
```

If you are using something else, like pygame, to read in the image stream, it should be usable with minimal preprocessing, if any. You then need to enter a loop where you are reading in successive frames from the image stream and checking each one for faces. A very simple boilerplate loop might look like this while True:

```
ret, frame = video_capture.read() gray
= cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)
faces = face_cascade.
```

Generating your own cascades

We have seen that the key part of object detection within an image is the cascade file. When you open one up, you will notice that it doesn't seem to be very complicated. The complicated bit is in the encoded data that defines that features within the cascade algorithm that are used to identify the object. This data needs to be discovered through a training step where the relevant OpenCV functions look at images that both have the object in question and images that do not have the object. You need a much larger set of negative images compared to positive images.

For example, the author of the blog entry at http://coding-robin.de/2013/07/22/train-your-own-opencv-haar-classifier.html needed to use a set of 40 positive images and 600 negative images in order to identify bananas within an image. The positive images also need to be preprocessed so that they are cropped as closely as possible to the object of interest. Once you have all of the positive and negative images, you need to create samples that can be used for the training step. To do this, you will need to install an extra package with

```
sudo apt-get install libopencv-dev
```

This package installs a number of utilities, including opencv_createsamples. This utility takes the training images, along with many potential parameters to produce annotated results that can then be used in the training step. The training is done by using the utility program opencv_traincascade. This is the newer trainer, that can operate as a multi-threaded application. Unfortunately, the format of the cascade XML file that gets written out is different from that generated by the older utility 'opencv_haartraining'. Depending on who you may want to share this cascade file with, means that you may need to use the older slower method.

The same as with opencv_createsamples, opencv_traincascade takes a large number of command line options to control how the training progresses. This process can take a long time, and a huge amount of memory. A long time is on the order of days, rather than just on the order of hours. And large amounts of memory is almost literally as much as you can throw at the problem. This is why you should definitely check Google before going through the massive amounts of work needed to generate your own cascade file.

> **"You can identify the entire body, or just the upper or lower portions of a body. You can even identify whether there is a smile within the image"**

```
detectMultiScale(gray, scaleFactor=1.1,
minNeighbors=5, minSize(30, 30),
flags=cv2.cv.CV_HAAR_SCALE_IMAGE)
```

This code assumes that the face_cascade object has already been instantiated using the correct Haar cascade file. This loop is not very useful however, as it doesn't do anything if any faces are found. If your program is an interactive one, you could simply have the identified frames popped up on the screen. A simple addition to this loop would be for:

```
(x, y, w, h) in faces:
cv2.rectangle(frame, (x, y), (x+w, y+h),
(0, 255, 0), 2)
cv2.imshow('Video', frame)
```

This code grabs the x and y coordinates and the width and height of the bounding box around the feature that was identified as a potential face. The function 'cv2.rectangle()' then takes these values to draw a green rectangle of the given width and height at the given location on the captured frame. It will loop through all of the detected faces and draw the bounding boxes on the image frame. Once they have all been drawn on, the function 'cv2.imshow()' will pop up a window in which to display the captured frame along with the highlighted faces. This is only an example of something you could do when a face is detected within an image frame. You may decide to do something else.

You may actually only be interested in whether a persons face was detected or not. In this case, you could simply decide to check the size of the faces list, and if it is not empty you would then need to trigger some other action. Continuing with the above example code, we need some way to get out of the infinite loop. Since you are in the middle of using OpenCV to do the image processing, you can also use the included utility functions to capture key presses using code like:

```
if cv2.waitKey(1) & 0xFF == ord('q'):
break
```

This will break out of the infinite while loop when the q key is pressed. After dropping out of the loop, there is still cleanup that must be done. Both the connection to the camera and all of the displayed image windows need to be destroyed. You can do this with the code below:

```
video_capture.release()
cv2.destroyAllWindows()
```

At this point, you can add basic facial recognition to any of your Python programs that have access to a video stream. The OpenCV library is efficient enough that you should be able to do this kind of recognition in real time, as the video stream is coming in. But, this isn't all that you can do at this point. If you look at the Haar cascade files that were installed when you installed the opencv-data package, you will see that there are several others available to help you identify other features. There are cascade files to identify a generic eye, or specifically a left eye or a right eye. You can identify the entire body, or just the upper or lower portions of a body. You can even identify whether there is a smile within the image frame. If there are other objects that you are interested in keeping an eye out for, a quick Google search may bring up a cascade file for exactly what you are trying to detect. If not, the other option is to then check out the side box for a quick look at what is involved if you then need to generate your own cascade file for some odd object.

Master essential Sense HAT skills: Part 1

Develop your Sense HAT skills and start a coding a version of Rock, Paper, Scissors, Lizard, Spock

In November 2015, two Raspberry Pi computers were each fitted with an Astro Pi, which is an add-on board boasting an array of sensors and an 8x8 LED matrix. These were flown to the ISS and delivered to Major Tim Peake with a selection of school children's experiments. In late December, the Astro Pi was released for sale to the public as the rebranded 'Sense HAT'. This consists of exactly the same hardware and sensors set found on the Astro Pi but with a new 'Sense HAT' API. The first part of this two-part tutorial introduces you to the Sense HAT hardware and walks you through the skills required to create a Sense HAT version of the updated classic Rock, Paper, Scissors, Lizard, Spock game. These skills also stand alone and you can adapt them for use in your own projects.

01 Install the Sense HAT software

First, attach the board to the GPIO header and install the Sense HAT software. This is pre-installed on the latest Raspbian images; for older images it can be downloaded – boot up your Raspberry Pi, load the LX Terminal and type in **sudo apt-get install sense-hat** to install the software. On completion, make sure you reboot your Raspberry Pi:

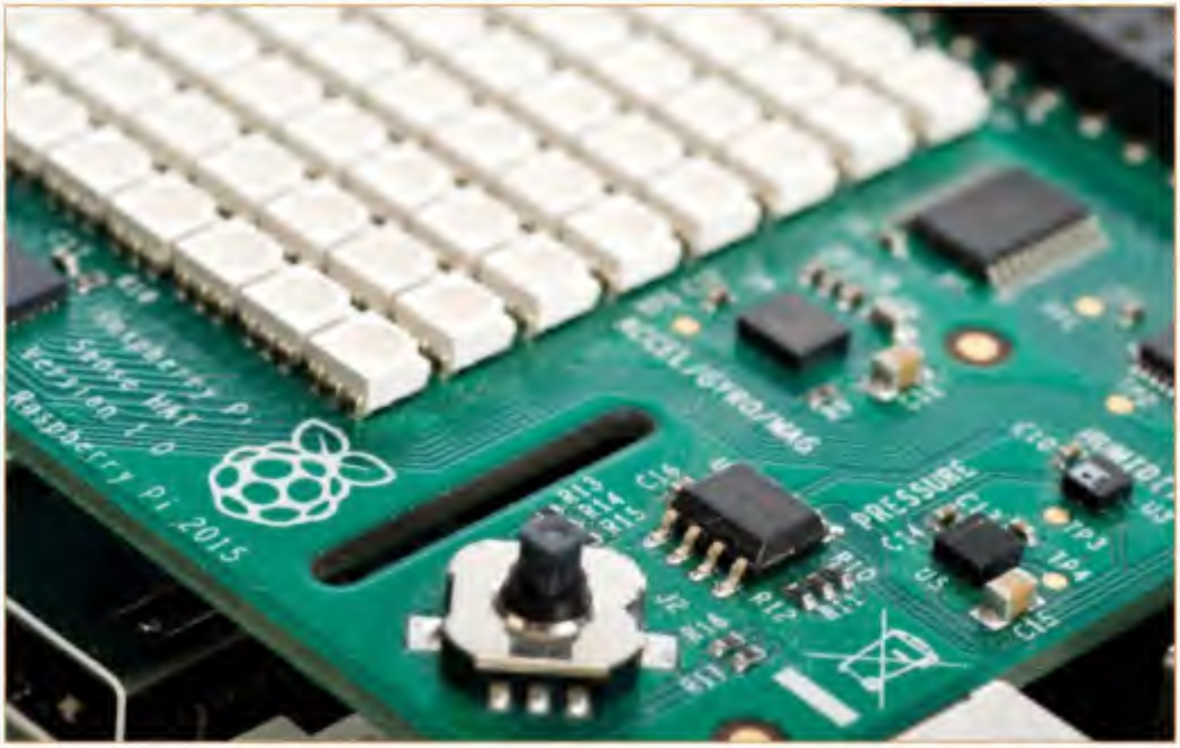
```
sudo apt-get update
sudo apt-get install sense-hat
sudo reboot
```

02 Scrolling a message

Writing code to scroll text on LCD / LED displays can be challenging and frustrating. The Sense HAT API removes the difficulties and simplifies the whole procedure to a simple line of code: **sense.show_message("This is a test message")**. Open your Python editor and enter the code at the bottom of this step, save and then run it. Your message will be scrolled across the Sense HAT LEDs. Change the text between the quotation marks to add your own message. Adjust the colour of the message and the time it takes to scroll by including the lines text_colour=[255, 0, 0]) (setting the RGB value) and scroll_speed=(0.05). Try experimenting with the example code below:

```
from sense_hat import Sense HAT
sense = SenseHat()
sense.show_message("Linux User and
```

Below The Sense HAT isn't just about the LED array – it's also packed full of useful sensors



Right Each of the LED squares in this grid can be individually controlled, so you can draw images

```
Developer",
text_colour=[255, 0, 0])
```

03 Taking a temperature reading

The Sense HAT has a built-in heat sensor that can be used to read and return the current temperature (line 3). The sensor is fairly close to the CPU and this may pick up some of the residual heat. However, on the whole the reading is sound. To measure the temperature, return to your Python editor and type in the code below, then save and run the file. This will return the current temperature reading and print it out:

```
from sense_hat import SenseHat
sense = SenseHat()
temp = sense.get_temperature()
print("Temperature: %s C" % temp)
```

04 Compass reading

One of the more nifty sensors is the magnetometer, which can be used as a compass. This returns a measurement of the Sense HAT's position in relation to magnetic north. Again the code is easy to use: **sense.get_compass()** in line 3 returns the position, which is stored in a variable called north. The value that is measured is then printed out in line 4. Use the code example below to test the compass sensor and the readings:

```
from sense_hat import SenseHat
sense = SenseHat()
north = sense.get_compass()
print("North: %s" % north)
```

05 Mapping an LED image from a picture

Images are built up of pixels which combine to create an overall picture. Each LED on the matrix can be automatically set from an image file. For example, an image of a lizard can be loaded, the colours and positions calculated, and then the corresponding LEDs enabled. The image needs to be 8 x 8 pixels in size so that it fits the LED Matrix. Download the supplied test picture file, lizard.png, and save it into the same folder as your program. Use the code below to open and load the image of the lizard (line 3). The Sense HAT will do the rest of the hard work for you:

```
from sense_hat import SenseHat
sense = SenseHat()
sense.load_image("lizard.png")
```

06 Make your own 8 x 8 image

There are two further methods to create an image with the LEDs. The first is a superb on-screen program that enables you to manipulate the LEDs in real time. You can change the colours, rotate them and then export the image as code or as an 8 x 8 PNG file. Install 'Python PNG library' by opening the terminal and typing **sudo pip3 install pypng**. After this has finished, type **git clone https://github.com/jrobinson-uk/RPi_8x8GridDraw**. Once the installation has completed, move to the RPi folder with **cd RPi_8x8GridDraw**, then type **python3 sense_grid.py** to run the application.

07 Create and export your image

The Grid Editor enables you to select from a range of colours displayed down the right-hand side of the window. Simply choose the colour and then click the location of the LED on the grid, select 'Play on LEDs' to display the colour on the Sense HAT LED. Clear the LEDs using the 'Clear Grid' button and then start over. Finally, when exporting the image, you can either save as a PNG file and apply the code in the previous step to display the picture, or you can export the layout as code and import that into your program.

The origins of RPSLS

Sheldon Cooper from *The Big Bang Theory* famously brought the game into the mainstream media when he explained how it was played to his friend, though it wasn't actually invented on the show. The original expansion of the classic Rock, Paper, Scissors game is attributed to Sam Kass and Karen Bryla.

Python version of RPSLS

If you can't wait for the next part of this tutorial, then you can play around with this Python version of the RPSLS game. The Sense HAT version uses similar mechanics to calculate the winner and losers of each round played: **https://trinket.io/python/46302ff1af**

08 Code each LED: Part 1

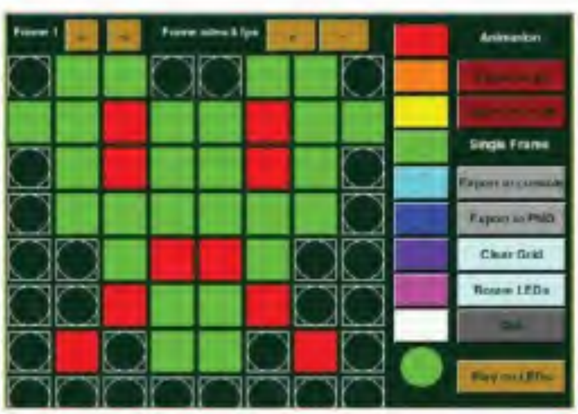
The second method to create an image is to individually code each LED and combine these. To set the colour of an LED, create a variable for a colour and assign an RGB value to it. Add additional colours by creating additional variables. Now create a representation of the image using the variable names: in this example, the X and O symbols combine to make a question mark. Set the LEDs with the code **sense.set_pixels(question_mark)**:

```
from sense_hat import SenseHat
sense = SenseHat()
X = [255, 0, 0]  # Red
O = [255, 255, 255]  # White
question_mark = [
O, O, O, X, X, O, O, O,
O, O, X, O, O, X, O, O,
O, O, O, O, O, X, O, O,
O, O, O, O, X, O, O, O,
O, O, O, X, O, O, O, O,
O, O, O, X, O, O, O, O,
O, O, O, O, O, O, O, O,
O, O, O, X, O, O, O, O
]
sense.set_pixels(question_mark)
```

09 Code each LED: Part 2

To add more colour variations to your image, create a new variable and assign the RGB values using the same format. Replace your previous code with the example below to create a new image. Lots of websites can convert the RGB values for all 16,581,375 colours – try **http://www.colorpicker.com**.

```
X = [255, 0, 0]  # Red
O = [255, 255, 255]  # White
B = [0, 255, 0] #Green
new_image = [
B, B, O, O, O, O, X, X,
B, B, O, O, O, O, X, X,
B, B, O, O, O, O, X, X,
B, B, O, O, O, O, X, X,
B, B, O, O, O, O, X, X,
B, B, O, O, O, O, X, X,
B, B, O, O, O, O, X, X,
B, B, O, O, O, O, X, X
]
```



Right The Sense HAT Grid Editor makes it easy to generate image code

10 Create an image of a lizard

Now you can begin to customise your own images to use. To get started, create a simple lizard using the LED layout. Then experiment with your own ideas and concepts. Remember, you will also need images for Rock, Paper, Scissors and Spock. Johan Vinet has some excellent and inspirational examples of 8 x 8 pixel art **http://johanvinet.tumblr.com/image/127476776680**.

11 Selecting a random image

Next you need a code to scroll through your images. Begin by renaming each of the images starting from one – for example 1.png, 2.png. Create a new variable called playersChoice which will store the current picture filename. Next, load the image using the code **sense.load_image(str(playersChoice) + ".png")** – note that playersChoice needs to be converted into a string before it is used. Once the picture is displayed, the variable is incremented, which loads the next picture. Finally, add a conditional to check if you have reached your last picture. For example, if you have five images then use:

```
if playersChoice == 5:
        playersChoice = 0
```

... to check when the variable reaches the last image and to reset the variable to zero. This loads the first picture and the loop starts again from the beginning:

```
import time
while True:
sense.load_image(str(playersChoice) + ".png")
playersChoice = playersChoice + 1
time.sleep(1)
if playersChoice == 5:
        playersChoice = 0
```

12 Clearing the image

When cycling through the images, the LEDs will turn on and off to the corresponding requirements of your code. However, there will be times when you need to turn off all the LEDs at once. This is referred to as clearing them. To clear the LEDs and set them all to 'off', use the line **sense.clear()**.

13 Setting the joystick up in PyGame

The Sense HAT is equipped with a small multi-directional joystick that can be programmed to respond to direction. The RPSLS game makes use of the joystick to cycle through your images and to select one. In a new Python file, set up the PyGame window by importing the PyGame modules at the top of your file (lines 1 and 2). Next, initialise with **pygame.init()**. PyGame runs in a separate window, but since the game takes place on the Sense HAT, the window is obsolete. You can therefore minimise the size by using: **pygame.display.set_mode((140, 180)) line 5**.

```
import pygame
from pygame.locals import *
###Set up PyGame Screen###
pygame.init()
pygame.display.set_mode((140, 180))
```

14 Add joystick controls

To add the joystick movement, create a variable called "running" which is set to True (line 1). Then add a **while** statement (line 2) to continually check whether the joystick has

Left We'll use the tiny joystick in the bottom-right corner as the input for our selection interface

been moved. The program checks for an event on line 4 and the picture cycle restarts after the fifth image. Line 5 checks for two conditions being met: a key being pressed and the playersChoice being less than a value of 5. If the condition is met, it checks for a joystick movement. The up arrow on the keyboard corresponds to K_UP, which in essence is the joystick moved up (line 6). Finally, add a print statement to test that code is responding to the joystick movement and that it works correctly (line 7).

```
running = True
while running == True:
 for event in pygame.event.get():
    if event.type == KEYDOWN and playersChoice < 5:
           if event.key == K_UP:
print("UP")
```

15 Selecting a picture

You have now coded PyGame to wait for 'events' and check and respond if the joystick is moved 'up'. In the RPSLS game, this procedure is used to enable the player to select a picture that represents their turn – for example, Paper. Replace the **print("UP")** on line 7 with the image section code used in step 11. As before, use **sense.load_image(str(playersChoice) + ".png")** to load the image onto the LEDs, then increment the playersChoice variable using **playersChoice = playersChoice + 1** to select the next picture. Line 6 checks if you have reached the end of the picture cycle and then resets it back to a value of zero (line 8), so the cycle begins again starting form picture one:

```
if event.type == KEYDOWN and playersChoice < 5:
      if event.key == K_UP:
          print (playersChoice)
          sense.load_image(str(playersChoice) + ".png")
          playersChoice = playersChoice + 1
  if playersChoice == 5:
      playersChoice = 0
```

> **"Adapt them for use in your own projects"**

16 Confirming your selection

Once you have scrolled through and chosen your picture then you need to be able to select it. This is achieved pressing the whole joystick button down, which acts as if the Return key has been pressed. On line 2, the code **if event.key == K_RETURN:** checks for this action and then responds by breaking the loop. The loop stops cycling through and the current image number is then stored inside the playersChoice variable, where it is used later in the game to compare with the Computer's choice and calculate the winner.

```
'''Checks for a 'select / Enter' Choice'''
  if event.type == KEYDOWN:
      if event.key == K_RETURN:
          running = False
          break
          '''Ends loop and moves onto main game'''
```

17 RPSLS

You now have a basic structure for the start of the RPSLS game. The program enables you to scroll through up to five images, which each represent one of the hands that you can play: Rock, Paper, Scissor, Lizard or Spock. In the next issue, part two of this tutorial will code the mechanics of the gameplay and combine this to create the finished RPSLS game. A sneak video of the final game in action can be viewed here: **www.youtube.com/watch?v=T_ZvWkMgVFM**. In the meantime, work on creating your 8 x 8 pixel art.

Master essential Sense HAT skills: Part 2

Use your skills from the last tutorial to create a Sense HAT version of RPSLS

By now you will probably have played Rock, Paper, Scissors in real life. The issue with this version is that there are only three possible outcomes other than a tie. Sam Kass and Karen Bryla invented an alternative version which adds "Spock" and "lizard". "Spock" is signified with the Star Trek Vulcan hand sign, while "lizard" is shown by forming the hand into a mouth. Spock smashes scissors and vaporises rock; he is poisoned by lizard and is disproved by paper. Lizard poisons Spock and eats paper; it is crushed by rock and decapitated by scissors. This tutorial walks you through creating your own SenseHAT version of the game.

01 How RPSLS works with the Sense HAT

When you complete this tutorial you will have a folder which contains five 8 x 8 images, each representing one of the items: rock, paper, scissors, lizard or Spock. These images are named 0.png, 1.png, 2.png, 3.png and 4.png. When the program runs, it will welcome you and ask you to use the joystick to select an item. Each time you push the joystick up, the item will change. Press enter to select. The computer then selects a random item. Each picture is assigned a value and a formula is used to find the modulus (the remainder) of dividing the two values. The value of the remainder determines the outcome: win, lose or draw.:

02 Import the modules

Boot up your Raspberry Pi and open the LX terminal. Type **sudo idle3** to load the Python 3 editor. Import the pygame module (line one) and the SenseHAT module (line three). The program uses the random module to select the computer's choices in the game. Import the random module (line four):

```
import pygame
import pygame.locals import *
from sense_hat import SenseHat
import random
import time
```

03 Scrolling a message

Throughout the game the player is updated via messages which are scrolled across LED matrix. The SenseHAT API simplifies the whole procedure to a simple line of code: **sense.show_message("This is a test message")**. Your message will be scrolled across the SenseHAT LEDs. Change the text between the quotation marks and add your own message. Adjust the colour of the message and the time it takes to scroll by including the lines, **text_colour=[255, 0, 0]**) (setting the RGB value) and **scroll_speed=(0.05)**, Try experimenting with the example code:

```
sense = SenseHat()
sense.show_message("Linux User & Developer", text_colour=[255,
0, 0])
```

04 Mapping an LED image

Images are made of pixels that combine to create an overall picture.

Each LED on the matrix can be automatically set from an image file. For example, an image of a lizard can be loaded, the colours and positions calculated and then the corresponding LEDs enabled. The image needs to be 8 x 8 pixels so that it fits the LED matrix. Download the test picture file – lizard.png – and save it into the same folder as your program:

```
from sense_hat import SenseHat
sense = SenseHat()
sense.load_image("lizard.png")
```

05 Create the game variables and initialise PyGame

Next, create variables to store the player's choice, line one, the computer's choice, and also to track the picture number that is currently displayed on the LED matrix, global count. These are set as global variables that enable them to be accessed within other parts of the program and to return the values that are stored. Now, set up the PyGame window typing **pygame.init()** and **pygame.display.set_mode((140, 180))**. The window is not used in the game so set it to a small size. Load the first image with the code **sense.load_image("0.png")**, this loads picture zero from your folder. You may find that the LEDs are too bright, add sense.**low_light = True**, to reduce the brightness. Finally set **playersChoice** to **0**, the first image, and then set the game running with **gameRunning = True**

```
global playersChoice
global computer_choice
global count
###Set up PyGame Screen###
pygame.init()
pygame.display.set_mode((140, 180))

###Prepare Sense Hat###
sense = SenseHat()
sense.load_image("0.png")
sense.low_light = True #save your eyes!

playersChoice = 0
gameRunning = True
```

06 Convert the values / numbers into items

During the RPSLS gameplay the program uses numbers to identify the items instead of their names. This means you can select a picture and also use a modulus operation to calculate the outcome of the game. Create a function that converts and assigns the value into the respective item name. A simple conditional checks what the number is and returns the name of the item. The value of 'scissors' is set twice – during the gameplay a loop checks that the fifth image has been loaded and then resets the value to zero which would result in the fifth image not being displayed.

```
def number_to_name(number):
if number == 0:
  return "Rock"
elif number == 1:
  return "Spock"
elif number == 2:
  return "Paper"
elif number == 3:
  return "Lizard"
elif number == -1: ### because value is 5 so re-sets
to 0, zero - 1 = -1 ###
  return "Scissors"
elif number == 4: ### because value is 5 so re-sets
to 0, zero - 1 = -1 ###
  return "Scissors" ### for the computer
```

07 Check for joystick movement

Create another function that holds the main mechanics of the gameplay. Start by adding the global variables **sense.set_rotation(90)**, which positions the image correctly. Next use **for event in pygame.event.get():** to check for a joystick movement. The joystick is used to cycle through a loop of the five LED images. The line **if event.type == KEYDOWN and playersChoice < 5:** checks that the player has moved the joystick up and also that the picture number is less than 5. If it is equal to five then the loop resets. If the **playersChoice value** is less than five then the corresponding picture number is loaded to the LEDs using **sense.load_image(str(playersChoice) + ".png")** line 13 and the variable incremented. The loop restarts from the beginning and checks for a joystick movement and then displays the relevant picture file:

```
def mainGame():
###PLAYER SELECTION###
###Loops while running variable is True###
running = True
global playersChoice
global computer_choice
while running == True:
  sense.set_rotation(90)
  for event in pygame.event.get():
    if event.type == KEYDOWN and playersChoice < 5:
      if event.key == K_UP:
        print (playersChoice)
        sense.load_image(str(playersChoice) + ".png")
        playersChoice = playersChoice + 1
    if playersChoice == 5:
      playersChoice = 0
```

08 Select your item

When you are scrolling through the images, you need to be able to select an item to play. Press the joystick down to act

Pixel inspiration

You will need an image for Rock, Paper, Scissors and Spock. Johan Viet has some excellent and inspirational examples of 8 x 8 pixel art. Check them out at **http://johanvinet.tumblr.com/image/127476776680**

Full code listing

```
import pygame
from pygame.locals import *
from sense_hat import SenseHat
import random
import time

global playersChoice
global computer_choice
global count

###Set up PyGame Screen###
pygame.init()
pygame.display.set_mode((140, 180))

###Prepare Sense Hat###
sense = SenseHat()
sense.load_image("0.png")
sense.low_light = True #save your eyes!

playersChoice = 0
gameRunning = True

'''Converts the Number into the choice i.e. lizard,
spock etc '''
def number_to_name(number):
  if number == 0:
    return "Rock"
  elif number == 1:
    return "Spock"
  elif number == 2:
    return "Paper"
  elif number == 3:
    return "Lizard"
  elif number == -1: ### because value is 5 so re-
sets to 0, zero - 1 = -1 ###
    return "Scissors"
  elif number == 4: ### because value is 5 so re-
sets to 0, zero - 1 = -1 ###
    return "Scissors" ### for the computer

def mainGame():
  ###PLAYER SELECTION###
  ###Loops while running variable is True###
  running = True
  global playersChoice
  global computer_choice
  while running == True:
    sense.set_rotation(90)
    for event in pygame.event.get():

      if event.type == KEYDOWN and playersChoice
< 5:
        if event.key == K_UP:
          print (playersChoice)
          sense.load_image(str(playersChoice) +
".png")
          playersChoice = playersChoice + 1
      if playersChoice == 5:
        playersChoice = 0

      '''Checks for a 'select / enter' Choice '''
      if event.type == KEYDOWN:
        if event.key == K_RETURN:
          running = False
          break
          '''Ends loop and moves onto main game'''

  '''Message for player about their choice'''
  #print ("Your Choice is", playersChoice) #test
```

as the Return key. Add the line **if event.key == K_RETURN:** line three, to respond to the joystick being pressed down, then end the picture cycle loop using **running = False**. Finally 'break' out of the loop to move onto the next part of the game:

```
'''Checks for a 'select / Enter' Choice '''
    if event.type == KEYDOWN:
      if event.key == K_RETURN:
        running = False
        break
```

09 Update the status

Once you have made your selection, scroll a confirmation message across the LEDs. Create a new variable called number and assign the playersChoice value to it. Next, use the function in step six to convert the number into the name of the item, **playerMessage = number_to_name(number)**. Finally scroll the message across using the line, **sense.show_message(playerMessage, text_colour=[0, 0, 255], scroll_speed = 0.08)**. You can change the text colour by altering the values within the square brackets and also the speed of the scroll between the values of zero and one:

```
'''Message for player about their choice'''
number = playersChoice - 1
playerMessage = number_to_name(number)
print playerMessage
sense.set_rotation(0)
sense.show_message("You = ", text_colour=[0, 255,
255], scroll_speed = 0.08)
  sense.show_message(playerMessage, text_colour=[0, 0,
255], scroll_speed = 0.08)
```

10 The computer's selection

Now it's the computer's turn. First the computer selects a random number between 5 and 50, **count = random.randrange(5,50)**. This is a random number of times that it will cycle the choices to give the impression of 'thinking'. Then the computer selects a random number between 0 and 5, **computer_choice = random.randrange(0,5)**. This is the computer's 'item' selection. Then one is subtracted from the cycle and the computer selects a new picture. This continues while the value of count is greater than zero. In each cycle the picture is displayed on the LED matrix with **sense.load_image(str(computer_choice) + ".png")**

```
###COMPUTER SELECTION
  '''Computer selects a random choice from the
options'''
  count = random.randrange(5,50)
sense.set_rotation(90)
while count > 1:
  computer_choice = random.randrange(0,5)
  print computer_choice
  time.sleep(0.1)
  sense.load_image(str(computer_choice) + ".png")
  count = count - 1
```

11 Game status update

The next part is to use messages to update the player. Use the function in step six to look up and convert the number into an item: **computerMessage = number_to_name(number)**. Now use **sense.show_message("Computer = ", text_colour=[0, 150, 255], scroll_speed = 0.06)** to display the computer's final choice:

```
#time.sleep(1)
  '''Message for player about the computer's choice'''
print ("The computers choice is", computer_choice)
number = computer_choice
computerMessage = number_to_name(number)
print computerMessage ##test
sense.set_rotation(0)
sense.show_message("Computer = ", text_colour=[0,
150, 255], scroll_speed = 0.06)
  #sense.show_message(computerMessage, text_colour=[0,
0, 255], scroll_speed = 0.08)
sense.load_image(str(computer_choice) + ".png")
```

12 Did you win?

The value of the computer's choice is subtracted from the value of the player's choice and divided by five, returning the remainder to determine the outcome of the game. Create a new variable called result and assign the calculation to it, **result = (int(computer_choice - (playersChoice-1))) % 5**. If the result is zero, then the player and the computer have picked the same item: the game is a tie. Use an IF statement on line five to check the value and then display a message using **sense.show_message("Player and Computer Tie!", text_colour=[0, 0, 255], scroll_speed = 0.08)**:

```
###WINNER CALCULATED###
'''Calculates the Winner'''
result = (int(computer_choice - (playersChoice-1))) % 5
  #print result
if result == 0:
    sense.show_message("Player and Computer Tie!",
text_colour=[0, 0, 255], scroll_speed = 0.08)
```

To win, the value of the remainder has to be greater than or equal to three. Use an ELSE IF statement to check this, **elif result >=3**. Any other value means that the computer has won this round. Add an else statement and use **sense.show_message("Computer Wins!", text_colour=[255, 0, 0], scroll_speed = 0.08** to display the winning or losing messages.:

```
elif result >=3:
    sense.show_message("Player Wins!", text_
colour=[0, 255, 0], scroll_speed = 0.08)
else:
    time.sleep(1)
    sense.show_message("Computer Wins!", text_
colour=[255, 0, 0], scroll_speed = 0.08)##??
```

13 Introducing the game

Now to add an intro message and instructions. You could add these into the main game loop but each time you start a new round you have to wait for them to repeat. Type **sense.show_message("Welcome to RPSLS!", text_colour=[155, 100, 30], scroll_speed = 0.08)** to scroll the messages and load the first image onto the LEDs with **sense.load_image("0.png")**:

```
###START THE GAME##
sense.show_message("Welcome to RPSLS!", text_
colour=[155, 100, 30], scroll_speed = 0.08)
sense.show_message("Pleases use 'Up' to select",
```

Creating the 8 x 8 images

A simple method to create your images is to use the RPi Grid Draw program. This enables you to manipulate the LEDs in real time. You can change the colours, rotate them and then export the image as code or as an 8 x 8 png file.

Install 'Python PNG library', open the Terminal window and type: sudo pip3 install pypng. After this has finished type, git clone **https://github.com/jrobinson-uk/RPi_8x8GridDraw**. Once the installation has completed move to the RPi folder, type cd RPi_8x8GridDraw, type python3 sense_grid.py to run the application

```
text_colour=[155, 255, 255], scroll_speed = 0.05)
sense.load_image("0.png")
```

14 Detecting play

Create a while loop that checks that the game is running. Next set a variable called play_again to a value of one. Then call the function created in steps seven which holds the game mechanics, on line three, type mainGame(). When the round finishes scroll a message across the LEDs asking the player if they want to play again, sense.show_message("Play Again?", text_colour=[255, 255, 255], scroll_speed = 0.08) line four:

```
while gameRunning == True:
play_again = 1
mainGame()
sense.show_message("Play Again?", text_colour=[255,
255, 255], scroll_speed = 0.08)
```

15 Play again?

Use an IF statement to select the option to play again. This is enabled by moving the joystick up. Use for event in pygame.event.get(): on line two and check for a 'keydown' event, the joystick being moved up, if event.type == KEYDOWN:, if event.key == K_UP: lines three and four. If the Up is selected then the play again variable is set to zero, line five, which ends the IF statement, and starts the game again:

```
while play_again == 1:
  for event in pygame.event.get():
    if event.type == KEYDOWN:
          if event.key == K_UP:
            play_again = 0
```

16 End the game and exit

After following all of these steps, check for a joystick movement down, **if event.key == K_DOWN**: Then you will need to type a message by using **sense.show_message("Bye Bye", text_colour=[255, 255, 255], scroll_speed = 0.08)** and then change the gameRunning variable to False. After doing this, wait and the program will automatically loop back to the start where it will then find that the loop from step 14 has now become False, meaning that the game will stop running, as you can see in the last section of code.

Full code listing (continued)

```
  number = playersChoice - 1
  playerMessage = number_to_name(number)
  print playerMessage
  sense.set_rotation(0)
  sense.show_message("You = ", text_colour=[0, 255,
255], scroll_speed = 0.08)
  sense.show_message(playerMessage, text_colour=[0,
0, 255], scroll_speed = 0.08)

  ###COMPUTER SELECTION###
  '''Computer selects a random choice from the
options'''
  count = random.randrange(5,50)
  sense.set_rotation(90)
  while count > 1:
    computer_choice = random.randrange(0,5)
    print computer_choice
    time.sleep(0.1)
    sense.load_image(str(computer_choice) + ".png")
    count = count - 1

      '''Message for player about the computer's
choice'''
  print ("The computers choice is", computer_choice)
  number = computer_choice
  computerMessage = number_to_name(number)
  print computerMessage ##test
  sense.set_rotation(0)
  sense.show_message("Computer = ", text_colour=[0,
150, 255], scroll_speed = 0.06)
  #sense.show_message(computerMessage, text_
colour=[0, 0, 255], scroll_speed = 0.08)
  sense.load_image(str(computer_choice) + ".png")

  print computerMessage
  time.sleep(1)

  ###WINNER CALCULATED###
  '''Calculates the Winner'''
  result = (int(computer_choice - (playersChoice-1)))
% 5

  if result == 0:
      sense.show_message("Player and Computer Tie!",
text_colour=[0, 0, 255], scroll_speed = 0.08)
      #print "tie"
  elif result >=3:
      sense.show_message("Player Wins!", text_
colour=[0, 255, 0], scroll_speed = 0.08)
      #print "Player wins!"
  else:
      time.sleep(1)
      sense.show_message("Computer Wins!", text_
colour=[255, 0, 0], scroll_speed = 0.08)##??
      #print "Computer wins!"
  print " "

###START THE GAME##
sense.show_message("Welcome to RPSLS!", text_
colour=[155, 100, 30], scroll_speed = 0.08)
sense.show_message("Pleases use 'Up' to select",
text_colour=[155, 255, 255], scroll_speed = 0.05)
sense.load_image("0.png")

while gameRunning == True:
  play_again = 1

  mainGame()
  sense.show_message("Play Again?", text_
colour=[255, 255, 255], scroll_speed = 0.08)
  while play_again == 1:
    for event in pygame.event.get():
      if event.type == KEYDOWN:
            if event.key == K_UP:
              play_again = 0

      if event.type == KEYDOWN:
            if event.key == K_DOWN:
              print("Bye")
              sense.show_message("Bye Bye", text_
colour=[255, 255, 255], scroll_speed = 0.08)
              play_again = 0
              gameRunning = False
```

HOW TO USE FileSilo

EVERYTHING YOU NEED TO KNOW ABOUT ACCESSING YOUR NEW DIGITAL REPOSITORY

To access FileSilo, please visit filesilo.co.uk/bks-960

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. 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 available on FileSilo, from great video tutorials and online guides to superb downloadable resources. And the more bookazines you purchase, the more your instantly accessible collection of digital content will grow.

03 You can access FileSilo on any desktop, tablet or smartphone device using any popular browser (such as Safari, Firefox or Google Chrome). 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 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 and Open Source 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

OVER 2 HOURS OF VIDEO TUTORIALS

The Python Book

The ultimate guide to coding with Python

Get to grips with the basics

Learn Python the right way and complete basic projects with our simple guides

Put Python to work

Supercharge your system and make life easier with handy coding tutorials

Use Python with Raspberry Pi

Work on any Raspberry Pi model using its officially recognised language

Over 20 projects