NEW

CODING ACADEMY 2025

FULLY REVISED & UPDATED EDITION

LEARN TO CODE FAST TODAY!

• PYTHON • RUST • ERLANG
• MONGO • REDIS • GO • RIAK

164 PAGES OF EXPERT GUIDES & TUTORIALS

LEARN CORE CODING TECHNIQUES AND ADVANCED SKILLS

Digital Edition

FUTURE

TWELFTH EDITION

CODING ACADEMY 2025

You hold in your hands a gateway to the world of programming. In an era when the online world runs the world, the people that can code that world, rule it. Just take a few of the biggest online names in the world, from the creator of Facebook, Mark Zuckerberg, to Google founders Larry Page and Sergey Brin – they are all programmers.

But you don't need to aim to create the next billion-pound startup to have a lust for learning to code. Coding can be fun and entertaining; it can open the gates to creating amazing projects used online by entire communities, down to small physical installations running at home on a Raspberry Pi; the only limit is your imagination. **Coding Academy** takes you through the fundamentals of learning to code and helps you flex your new programming muscles with a host of practical Python projects, introducing the basics of database interactions and covering the cutting-edge languages of Erlang, Go, and Rust. Let's get coding!

FUTURE

CODING ACADEMY 2025

Future PLC Quay House, The Ambury, Bath, BA1 1UA

Editorial
Compiled by **Aiden Dalby & Steve Dacombe**
Senior Art Editor **Andy Downes**
Head of Art & Design **Greg Whitaker**
Editorial Director **Jon White**
Managing Director **Grainne McKenna**

Photography
All copyrights and trademarks are recognised and respected

Advertising
Media packs are available on request
Commercial Director **Clare Dove**

International
Head of Print Licensing **Rachel Shaw**
licensing@futurenet.com
www.futurecontenthub.com

Circulation
Head of Newstrade **Tim Mathers**

Production
Head of Production **Mark Constance**
Production Project Manager **Matthew Eglinton**
Advertising Production Manager **Joanne Crosby**
Digital Editions Controller **Jason Hudson**
Production Managers **Keely Miller, Nola Cokely, Vivienne Calvert, Fran Twentyman**

Printed in the UK

Distributed by Marketforce – www.marketforce.co.uk
For enquiries, please email: mfcommunications@futurenet.com

GPSR EU RP (for authorities only)
eucomply OÜ Pärnu mnt 139b-14 11317, Tallinn, Estonia
hello@eucompliancepartner.com, +3375690241

Coding Academy 2025 Twelfth Edition (TCB6779)
© 2025 Future Publishing Limited

We are committed to only using magazine paper which is derived from responsibly managed, certified forestry and chlorine-free manufacture. The paper in this bookazine was sourced and produced from sustainable managed forests, conforming to strict environmental and socioeconomic standards.

All contents © 2025 Future Publishing Limited or published under licence. All rights reserved. No part of this magazine may be used, stored, transmitted or reproduced in any way without the prior written permission of the publisher. Future Publishing Limited (company number 2008885) is registered in England and Wales. Registered office: Quay House, The Ambury, Bath BA1 1UA. All information contained in this publication is for information only and is, as far as we are aware, correct at the time of going to press. Future cannot accept any responsibility for errors or inaccuracies in such information. You are advised to contact manufacturers and retailers directly with regard to the price of products/services referred to in this publication. Apps and websites mentioned in this publication are not under our control. We are not responsible for their contents or any other changes or updates to them. This magazine is fully independent and not affiliated in any way with the companies mentioned herein.



Future plc is a public company quoted on the London Stock Exchange (symbol: FUTR)
www.futureplc.com

Chief Executive Officer **Kevin Li Ying**
Non-Executive Chairman **Richard Huntingford**
Chief Financial Officer **Sharjeel Suleman**

Tel +44 (0)1225 442 244





Contents

CODING ACADEMY

Fundamentals

10 **Different types of data**
How Python handles various variables

12 **More Python data types**
Beyond the previous ones, of course

14 **Reliability by abstraction**
Open your coding mind and think differently

16 **Files and modules done quickly**
The easy way to break things apart

18 **Write your own UNIX program**
Rewriting cat for fun and (no) profit

24 **Neater code with modules**
Use them wisely for super-clean programs

26 **Embrace storage & persistence**
Deal with persistent data in Python

28 **Lock down with data encryption**
Look after your data and it will look after you

Projects

32 **Python 3: How to get started**
We're moving to Python 3! About time

36 **Python 3: Using functions**
The essentials of modules, classes and more

40 **FTP: Build a client & server**
Create your own network code for programs

44 **Python 3: Making scripts**
Automate your system with scripts

48 **Python: times, dates & numbers**
Essential processing of complex dates

52 **NumPy & SciPy: For science!**
Become a data scientist



"Coding can be fun and entertaining; it can open the gates to creating amazing projects"

Contents

Databases

58 **Python and SQLite3**
Get started using databases

62 **Get to grips with MariaDB**
The open source alternative to MySQL

66 **MongoDB: Using native drivers**
NoSQL mixed with Python and Ruby

70 **MongoDB: An admin's guide**
Milk maximum power from Mongo

74 **MongoDB: Build a blog**
Get your words collected in a database

78 **Riak NoSQL**
Just what is the big deal about NoSQL?

82 **Redis data store**
High-speed, in memory NoSQL

Do more

88 **Program in Erlang: Introduction**
Get to grips with the Erlang language

94 **Erlang: Functions**
Discover functions and basic data types

100 **Rust: modules and cargo**
Get started with the hip new language

104 **Rust: functions and modules**
Go further tackling error handling

108 **Rust: file I/O**
Speed up your transfers

112 **Rust: networking**
Build a TCP-based client/server tool

116 **Rust: concurrency**
How to use Threads

120 **Go: Master Google's new programming language**
Get yourself familiar with the fundamentals

126 **Go: Data types**
Get to grips with Go's composite data types

132 **Go: Explore functions**
Learn how you can develop and use functions

Reference

140 **Get to grips with Python lists**
It's one thing after another, isn't it?

142 **Understanding functions and objects**
All about things and what things do

144 **Adapt and evolve with conditionals**
Because change is inevitable

146 **Variable scope of various variables**
There's a variety of variables, and they vary

148 **Recursion: round and round we go**
Repetition is no bad thing

150 **Super sorting algorithms**
Get everything orderly, and quick

152 **Hidden secrets of numbers**
Integers thoroughly demystified

154 **Using loops and using loops**
And using loops and using loops

156 **The magic of compilers**
How they translate your code

158 **Avoid common coding mistakes**
Get it right the first time

CODING ACADEMY

CODING ACADEMY

Fundamentals

Using Python, we'll take you through essential coding steps

10 **Different types of data**
How Python handles various variables

12 **More Python data types**
Beyond the previous ones, of course

14 **Reliability by abstraction**
Open your coding mind and think differently

16 **Files and modules done quickly**
The easy way to break things apart

18 **Write your own UNIX program**
Rewriting cat for fun and (no) profit

24 **Neater code with modules**
Use them wisely for super-clean programs

26 **Embrace storage & persistence**
Deal with persistent data using Python

28 **Lock down with data encryption**
Look after your data and it will look after you

Fundamentals

Different types of Python data

Functions tell programs how to work, but it's data that they operate on. Let's go through the basics of data in Python

In this section, we'll be using Python to broach many of the fundamental concepts of coding. Here we look at basic data types; in later articles, we'll look at a few more advanced topics that build on what we do here: data abstraction, fancy structures such as trees, and more.

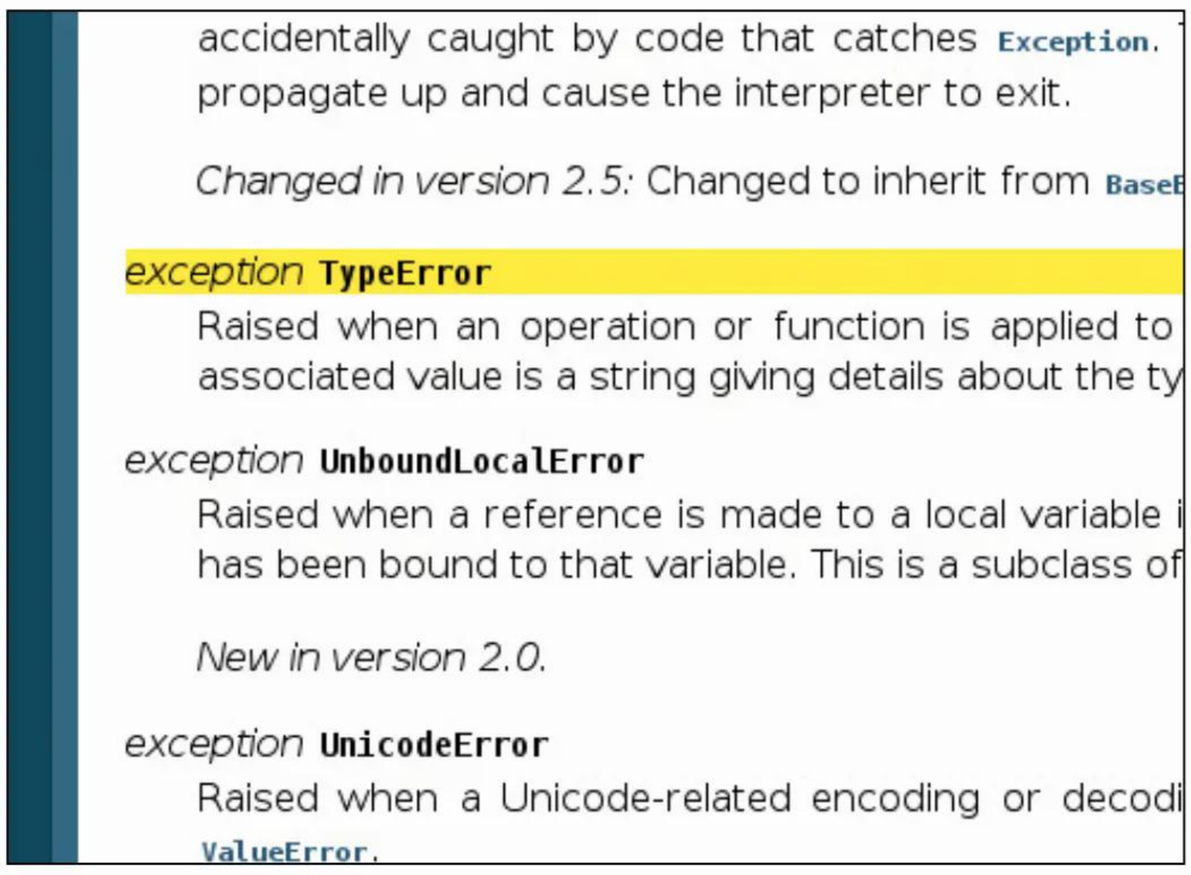
What is data?

In the world, and in the programs that we'll write, there's an amazing variety of different types of data. In a mortgage calculator, for example, the value of the mortgage, the interest rate and the term of the loan are all types of data; in a shopping list program, there are all the different types of food and the list that stores them – each of which has its own kind of data.

The computer's world is a lot more limited. It doesn't know the difference between all these data types, but that doesn't stop it from working with them. The computer has a few basic ones it can work with, and that you have to use creatively to represent all the variety in the world.

We'll begin by highlighting three data types: first, we have numbers. 10, 3 and 2580 are all examples of these. The particular numbers are ints, or integers. Python knows about other types of numbers, too, including longs (long integers), floats (numbers with a decimal point such as 10.35 or 0.8413) and complex (complex numbers). There are also strings, such as **'Hello World'**, **'Banana'** and **'Pizza'**. These are identified as a sequence of characters enclosed within quotation marks. You can use either double or single quotes. Finally, there are lists, such as **['Bananas', 'Oranges', 'Fish']**.

› While we're looking only at basic data types, in real programs getting the wrong type can cause problems, in which case you'll see a TypeError


accidentally caught by code that catches Exception.
propagate up and cause the interpreter to exit.

Changed in version 2.5: Changed to inherit from BaseE

exception **TypeError**

Raised when an operation or function is applied to
associated value is a string giving details about the ty

exception **UnboundLocalError**

Raised when a reference is made to a local variable i
has been bound to that variable. This is a subclass of

New in version 2.0.

exception **UnicodeError**

Raised when a Unicode-related encoding or decodi
ValueError.

In some ways, these are like a string, in that they are a sequence. What makes them different is that the elements that make up a list can be of any type.

In this example, the elements are all strings, but you could create another list that mixes different types, such as **['Bananas', 10, 'a']**. Lists are identified by the square brackets that enclose them, and each item or element within them is separated by a comma.

Working with data

There are lots of things you can do with the different types of data in Python. For instance, you can add, subtract, divide and multiply two numbers and Python will return the result:

```
>>> 23 + 42
65
>>> 22 / 11
2
```

If you combine different types of numbers, such as an int and a float, the value returned by Python will be of whatever type retains the most detail – that is to say, if you add an int and a float, the returned value will be a float.

You can test this by using the **type()** function. It returns the data type of whatever argument you pass to it.

```
>>> type(8)
<type 'int'>
>>> type(23.01)
<type 'float'>
>>> type(8 + 23.01)
<type 'float'>
```

You can also use the same operations on strings and lists, but they have different effects. The **+** operator concatenates, that is combines together, two strings or two lists, while the ***** operator repeats the contents of the string or list.

```
>>> "Hello " + "World"
"Hello World"
>>> ["Apples"] * 2
["Apples", "Apples"]
```

Strings and lists also have their own special set of operations, including slices. These enable you to select a particular part of the sequence by its numerical index, which begins from 0.

```
>>> word = "Hello"
>>> word[0]
'H'
>>> word[3]
'l'
>>> list = ['banana', 'cake', 'tiffin']
>>> list[2]
'tiffin'
```

Indexes work in reverse, too. If you want to reference the last element of a list or the last character in a string, you can use the same notation with a **-1** as the index. **-2** will reference the second-to-last character, **-3** the third, and so on. Note that when working backwards, the indexes don't start at **0**.

Methods

Lists and strings also have a range of other special operations, each unique to that particular type. These are known as methods. They're similar to functions such as **type()** in that they perform a procedure. What makes them different is that they're associated with a particular piece of data, and hence have a different syntax for execution. For example, among the list type's methods are **append** and **insert**.

```
>>> list.append('chicken')
>>> list
['banana', 'cake', 'tiffin', 'chicken']
>>> list.insert(1, 'pasta')
>>> list
['banana', 'pasta', 'cake', 'tiffin', 'chicken']
```

As you can see, a method is invoked by placing a period between the piece of data that you're applying the method to and the name of the method. Then you pass any arguments between round brackets, just as you would with a normal function. It works the same with strings and any other data object, too:

```
>>> word = "HELLO"
>>> word.lower()
'hello'
```

There are lots of different methods that can be applied to lists and strings, and to tuples and dictionaries (which we're about to look at). To see the order of the arguments and the full range of methods available, you'll need to consult the Python documentation.

Variables

In the previous examples, we used the idea of variables to make it easier to work with our data. Variables are a way to name different values – different pieces of data. They make it easy to manage all the bits of data you're working with, and greatly reduce the complexity of development (when you use sensible names).

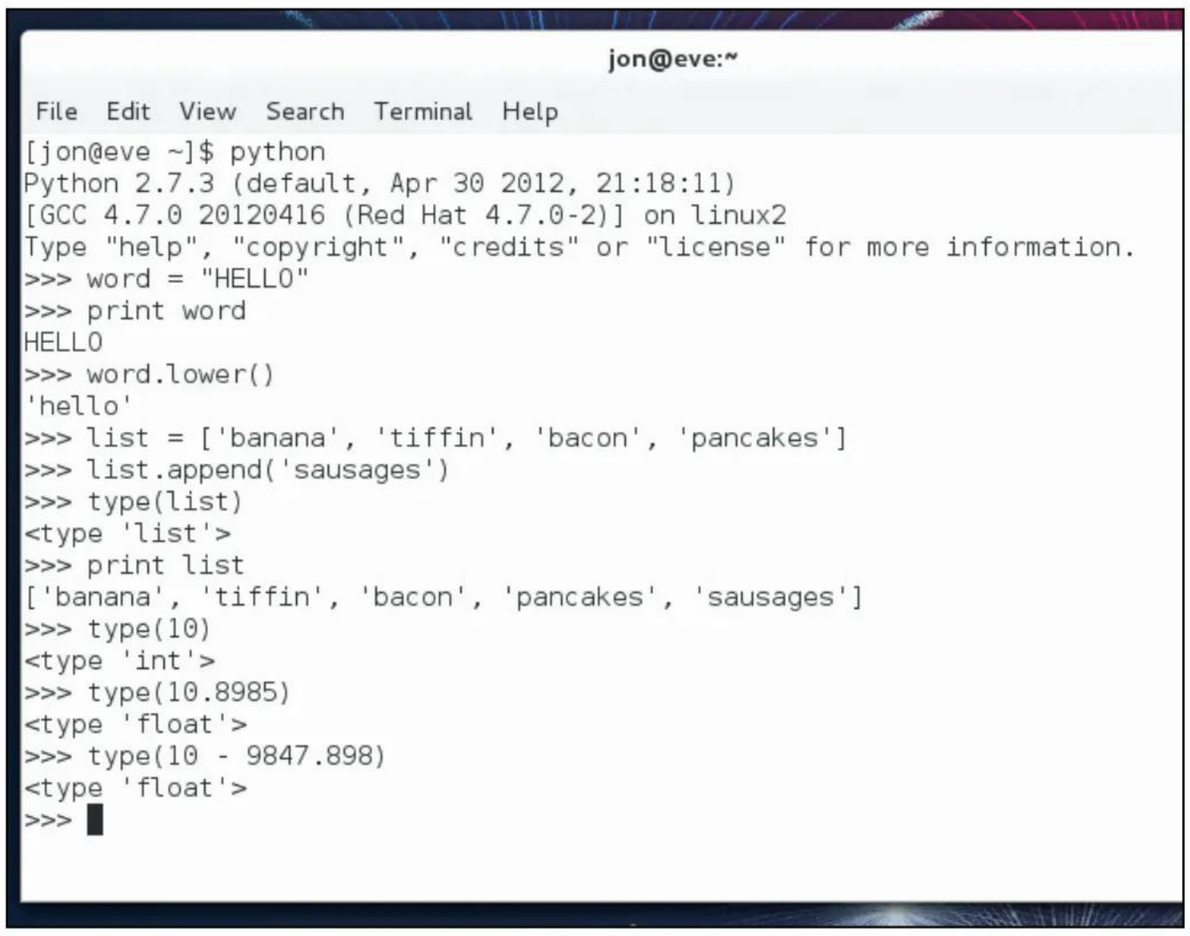
As we saw above, in Python you create a new variable with an assignment statement. First comes the name of the variable, then a single equals sign, followed by the piece of data that you want to assign to that variable.

From that point on, whenever you use the name assigned to the variable, you are actually referring to the data that you assigned to it. In the examples, we saw this in action when we referenced the second character in a string or the third element in a list by appending index notation to the variable name. You can also see this in action if you apply the **type()** function to a variable name:

```
>>> type(word)
<type 'str'>
>>> type(list)
<type 'list'>
```

Other data types

There are two other common types of data that are used by Python: tuples and dictionaries. Tuples are very similar to lists – they're a sequence data type, and they can contain elements of mixed types. The big difference is that tuples are immutable – that is to say, once you create a tuple you cannot change it – and that tuples are identified by round brackets, as opposed to square brackets: **('bananas', 'tiffin', 'cereal')**. Dictionaries are similar to a list or a tuple in that they contain a collection of related items. They differ in that the elements aren't indexed by numbers, but by 'keys' and are created with curly brackets: **{}**. It's quite like an English language dictionary. The key is the word that you're looking up, and the value is the definition of the word.



› The Python interpreter is a great place to experiment with Python code and see how different data types work together

With Python dictionaries, however, you can use any immutable data type as the key (strings are immutable, too), so long as it's unique within that dictionary. If you try to use an already existing key, its previous association is forgotten completely and that data lost for ever.

```
>>> english = {'free': 'as in beer', 'linux': 'operating system'}
>>> english['free']
'as in beer'
>>> english['free'] = 'as in liberty'
>>> english['free']
'as in liberty'
```

Looping sequences

One common operation that you may want to perform on any of the sequence types is looping over their contents to apply an operation to every element contained within. Consider this small Python program:

```
list = ['banana', 'tiffin', 'burrito']
for item in list:
    print item
```

First, we created the list as we would normally, then we used the **for... in...** construct to perform the **print** function on each item in the list. The second word in that construct doesn't have to be **item**, that's just a variable name that gets assigned temporarily to each element contained within the sequence specified at the end. We could just as well have written **for letter in word** and it would have worked just as well.

Once you have mastered these basic data types, you'll be ready to look at how you can put this knowledge to use when modelling real-world problems in later articles. In the meantime, read the Python documentation to become familiar with some of the other methods that it provides for the data types we've looked at before. You'll find lots of useful tools, such as **sort** and **reverse**! ■

Fundamentals

More Python data types

Learn how different types of data come together to solve a real problem as we write some code that counts words

In the previous tutorial, we introduced Python's most common data types: numbers (ints and floats), strings, lists, tuples and dictionaries. We demonstrated how they work with different operators, and explained a few of their most useful methods. We didn't, however, give much insight into how they might be used in real situations. In this article, we're going to fix that.

We're going to write a program that counts the number of times each unique word occurs in a text file. Punctuation marks will be excluded, and if the same word occurs but in different cases (for example, **the** and **The**), they will be taken to represent a single word. Finally, the program will print the results to the screen. It should look like this:

```
the: 123
you: 10
a: 600
...
```

As an example, we'll be using *The Time Machine*, by HG Wells, which you can download from Project Gutenberg, saving it in the same folder as your Python file under the name **timemachine.txt**.

As the program description suggests, the first thing we'll need to do is make the text accessible from inside our Python program. This is done with the **open()** function:

```
tm = open('timemachine.txt', 'r')
```

In this example, **open()** is passed two variables. The first is the name of the file to open; if it were in a different directory from the Python script, the entire path would have to be given. The second argument specifies which mode the file should be opened in: **r** stands for **read**, but you can also use **w** for write or **rw** for **read-write**.

Notice we've also assigned the file its own variable name, **tm**, so we can refer to it later in the program.

With a reference to the file created, we also need a way to access its contents. There are several ways to do this, but today we'll be using a **for... in...** loop. To see how this works, try opening **timemachine.txt** in the interactive interpreter and then typing:

```
>>> for line in tm:
        print line
...
```

The result should be every line of the file printed to the screen. By putting this code in to a **.py** file, say **cw.py**, we've got the start of our Python program.

103: is	200: but
105: came	215: with
111: so	216: for
111: upon	242: at
112: into	265: as
112: little	280: me
112: one	352: had
114: not	368:
115: all	416: it
118: he	436: that
121: have	438: my
121: they	541: in
122: from	548: was
122: there	678: to
129: his	803: a
132: then	1151: of
132: you	1230: and
137: on	1241: i
151: this	2253: the
157: were	
196: time	
200: but	

› Hardly surprisingly, our counting program, after being sorted, finds 'the' to be the most common word in *The Time Machine*, by HG Wells

Cleaning up

The program description also specified that we should exclude punctuation marks, consider the same word but in different cases as one word, and that we're counting individual words, not lines. As it stands, we have been able to read only entire lines as strings, however, with punctuation, strange whitespace characters (such as **\r\n**) and different cases intact.

Looking at the Python string documentation (find it at **http://docs.python.org/library**), we can see that there are four methods that can help us convert line strings into a format closer to that specified by the description: **strip()**, **translate()**, **lower()** and **split()**. Each of these are methods, and as such they're functions that are applied to particular strings using the dot notation. For example, **strip()**, which removes specified characters from the beginning and end of a string, is used like this:

```
>>> line.strip()
```

When passed with no arguments, it removes all whitespace characters, which is one of the jobs we needed to get done.

The function **translate()** is a method that can be used for removing a set of characters, such as all punctuation marks, from a string. To use it in this capacity, it needs to be passed two arguments, the first being **None** and the second being the list of characters to be deleted.

```
>>> line.translate(None, '!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~')
```

lower() speaks for itself, really – it converts every character in a string to lower-case. **split()** splits distinct elements

inside a string in to separate strings, returning them as a list. By passing an argument to **split()**, it's possible to specify which character identifies the end of one element and the start of another.

```
>>> line.split(‘ ‘)
```

In this example, we've passed a single space as the character to split the string around. With all punctuation removed, this will create a list, with each word in the string stored as a separate element. Put all of this in the Python file we started working on earlier, inside the **for** loop, and we've made considerable progress. It should now look like this:

```
tm = open(‘timemachine.txt’, ‘r’)
for line in tm:
    line = line.strip()
    line = line.translate(None, ‘!”#$%&\’()*+,-./:;<=>?@
    [\\]^_`{|}~’)
    line = line.lower()
    list = line.split(‘ ‘)
```

Because all of the string methods return a new, modified string, rather than operating on the existing string, we've re-assigned the line variable in each line to store the work of the previous step.

Uniqueness

Phew, look at all that work we've just done with data! By using the string methods, we've been able to remove all the bits of data that we weren't interested in. We've also split one large string, representing a line, into smaller chunks by converting it to a list, and in the process got to the exact, abstract concept we're most interested in: words. Our stunning progress aside, there is still some work to be done. We now need a way to identify which words are unique – and not just in this line, but in every line contained within the entire file.

The first thing that should pop into your head when thinking about uniqueness is a dictionary, the key-value store we saw in the previous article. It doesn't allow duplicate keys, so by entering each word as a key within a dictionary, we're guaranteed there won't be any duplicates.

What's more, we can use the value to store the number of times each word has occurred, incrementing it as the program comes across new instances of each key.

Start by creating the dictionary, and ensuring that it persists for the entire file – not just a single line – by placing this line before the start of the **for** loop:

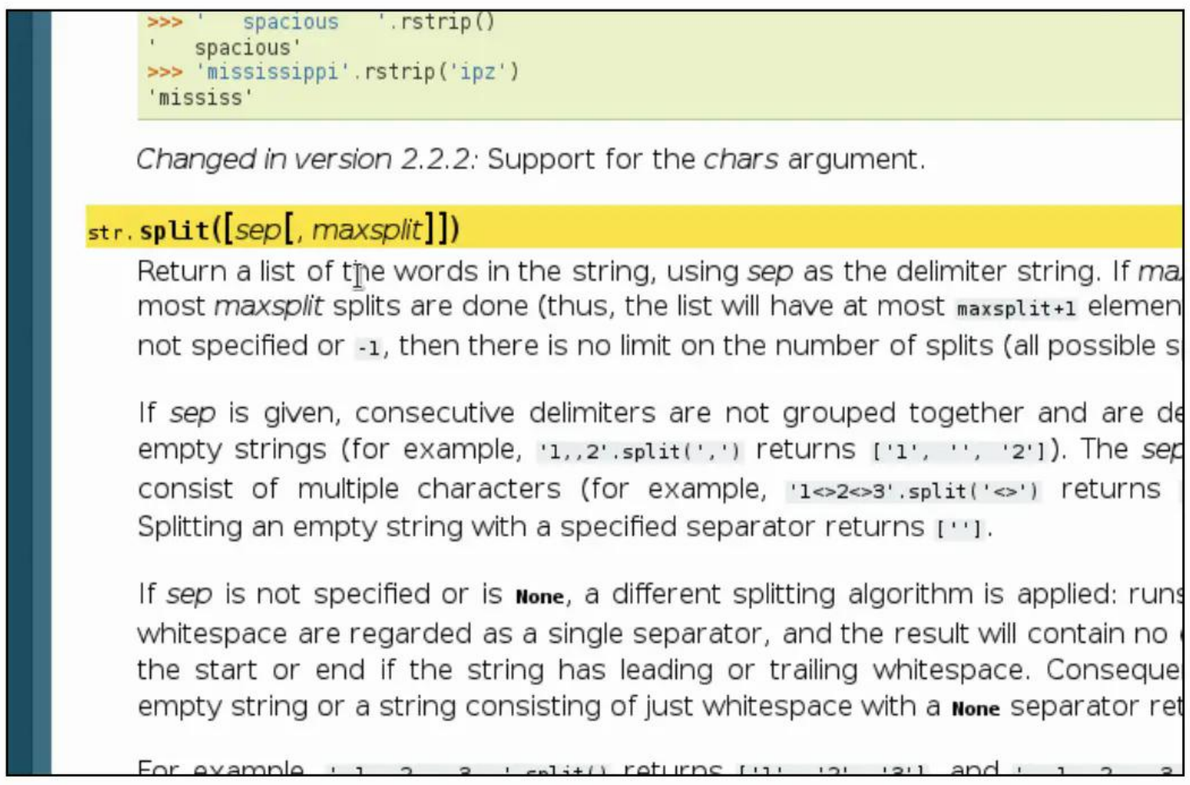
```
dict = {}
```

This creates an empty dictionary, which is ready to receive our words. Next, we need to think about a way to get each word into the dictionary. As we saw last time, ordinarily a simple assignment statement would be enough to add a new word to the dictionary. We could then iterate over the list we created above (using another **for** loop), adding each entry to the dictionary with a value of **1** (to represent that it has occurred once in the file).

```
for word in list:
    dict[word] = 1
```

But remember, if the key already exists, the old value is overwritten and the **count** will be reset. To get around this, we can place an **if-else** clause inside the loop:

> **“Another data type wrestled with, another step closer to our goal”**



```
>>> '   spacious   '.rstrip()
'   spacious'
>>> 'mississippi'.rstrip('ipz')
'mississ'
```

Changed in version 2.2.2: Support for the *chars* argument.

`str.`**split**([*sep*[, *maxsplit*]])

Return a list of the words in the string, using *sep* as the delimiter string. If *ma*
most *maxsplit* splits are done (thus, the list will have at most `maxsplit+1` elemen
not specified or `-1`, then there is no limit on the number of splits (all possible s

If *sep* is given, consecutive delimiters are not grouped together and are de
empty strings (for example, `'1,,2'.split(',')` returns `['1', '', '2']`). The *sep*
consist of multiple characters (for example, `'1<>2<>3'.split('<>')` returns
Splitting an empty string with a specified separator returns `['']`.

If *sep* is not specified or is **None**, a different splitting algorithm is applied: runs
whitespace are regarded as a single separator, and the result will contain no
the start or end if the string has leading or trailing whitespace. Conseque
empty string or a string consisting of just whitespace with a **None** separator ret

For example, `'1  2  3'.split()` returns `['1', '2', '3']` and `'1  2  3'`

```
if word in dict:
    count = dict[word]
    count += 1
    dict[word] = count
else:
    dict[word] = 1
```

This is a little bit confusing because **dict[word]** is being used in two different ways. In the second line, it returns the value and assigns it to the variable count, while in the fourth and seventh lines, **count** and **1** are assigned to that key's value, respectively. Notice, too, that if a word is already in the dictionary, we increment the **count** by 1, representing another occurrence.

› Python's Standard Library reference, http://docs.python.org/library, is an invaluable source for discovering what methods are available and how to use them

Putting it together

Another data type wrestled with, another step closer to our goal. At this point, all that's left to do is insert some code to print the dictionary and put it all together and run the program. The **print** section should look like this and be at the very end of the file, outside of the line-looping code.

```
for word,count in dict.iteritems():
    print word + “:“ + str(count)
```

This **for** loop looks different to what you've seen before. By using the **iteritems** method of the dictionary, we can access both the key (**word**) and value (**count**) in a single loop. What's more, we've had to use the **str()** function to convert **count**, an integer, into a string, as the **+** operator can't concatenate an integer and a string.Try running it, and you should see your terminal screen filled with lines like:

```
...
other: 20
sick: 2
ventilating: 2
...
```

Data everywhere!

That's all we planned to achieve in this particular tutorial and it's actually turn out to be quite a lot. As well as having had a chance to see how several different types of data and their methods can be applied to solve a real problem, we hope you've noticed how important it is to select the appropriate type for representing different abstract concepts. As a further programming exercise, why not look into sorting the resulting dictionary in order to see which words occur most frequently?

You might also want to consider writing the result to a file, one entry on a line, to save the fruits of your labour. ■

Reliability by abstraction

Think your code is solid? Perhaps it's a bit too lumpy. Creating abstractions can make code much easier to maintain

In the previous few tutorials, we have been looking at data. First, we introduced some of Python's core data types, and then we demonstrated how they can be put to use when solving a real problem. The next data-related topic we want to consider is abstraction, but before we get on to that, we're first going to look at abstraction – both in general and as it applies to procedures. So, this time we'll take a brief hiatus from data, before returning to it in a later article.

Square roots

To get our heads around the concept of abstraction, let's start by thinking about square roots and different techniques for finding them. One of these was discovered by Newton, and is thus known as Newton's method.

Newton's method states that when trying to find the square root of a number (x), we should start with a guess (y) of its square root; we can then improve upon that result by averaging our guess (y) with the result of dividing the number (x) by our guess (y). As we repeat this procedure, we get closer and closer to the square root. In most attempts, we'll never reach a definite result, we'll only make our guess more and more accurate. Eventually, we'll reach a level of accuracy that is good enough for our needs and then give up. Just to be clear about what is involved, take a look at the table below for how you would apply this method to find the square root of 2 (for example, x).

It's a lot of work just to find the square root of a number. Imagine if, when you were in school, every time you had to find a square root you had to do all these steps manually. Solving problems involving Pythagoras' theorem, for instance, would be much more unwieldy.

Luckily, assuming you were allowed to use calculators when you were at school, there's another, much easier method you can use to find square roots. Calculators come with a button marked with the square root symbol, and all you have to do is press this button once – much easier. This second approach is what's known as an abstraction. When working on problems, such as those involving Pythagoras' theorem, we don't care about how to calculate the square root, only that we can do it and get the correct result. We can treat the square root button on our calculator as a black box – we never look inside it, we don't know how it does what it does, all that matters is that we know how to use it and that it gives the correct result.

This a very powerful technique, one which can make programming a lot easier because it helps us to manage complexity. To demonstrate how abstraction can help us, consider the Python code below for finding the longest side of a right-angled triangle:

```
import math
def pythag(a, b):
    a2b2 = (a * a) + (b * b)
    guess = 1.0
    while (math.fabs((guess * guess) - a2b2) > 0.01):
        guess = (((a2b2 / guess) + guess) / 2)
    return guess
```

The first thing to note is that it's not in the least bit readable. Sure, with a piece of code this short, you can read through it reasonably quickly and figure out what's going on, but at a glance it's not obvious, and if it were longer and written like this you'd have a terrible time figuring out what on earth it was doing. What's more, it would be very difficult to test the different parts of this code as you go along (aka incremental development, vital for building robust software). For instance, how would you break out the code for testing whether or not a guess is close enough to the actual result, or the code for improving a guess, to check that it works? What if this function didn't return the expected results – how would you start testing all the parts to find where the error was?

> **"This is a very powerful technique that makes programming easier"**

Finally, there's useful code in here that could be reused in other functions, such as that for squaring a number, for taking an average of two numbers, and even for finding the square root of a number, but none of it is reusable because of the way it's written. You could type it all out again, or copy and paste it, but the more typing you have to do, the more obscure code you have to copy and paste, and the more likely mistakes are to make it in to your programming.

Let's write that code again, this time with some abstractions to fix the problems listed above. We haven't listed the contents of each new function we've created, leaving them for you to fill in.

Find a square root

Guess (y)	Division (x/y)	Average ((x/y) + y)/2)
1	2/1 = 2	(2 + 1)/2 = 1.5
1.5	2/1.5 = 1.33	(1.33 + 1.5)/2 = 1.4167
1.4167	2/1.4167 = 1.4118	(1.4118 + 1.4167)/2 = 1.4142

Reliability by abstraction

```
import math:
def square(x):
    ...
def closeEnough(x, guess):
    ...
def improveGuess(x, guess):
    ...
def sqrt(x, guess):
    ...
def pythag(a, b):
    a2b2 = square(a) + square(b)
    return sqrt(a2b2)
```

Here, we've split the code in to several smaller functions, each of which fulfils a particular role. This has many benefits. For starters, how much easier is the **pythag()** function to read? In the first line, you can see clearly that **a2b2** is the result of squaring two numbers, and everything below that has been consolidated in to a single function call, the purpose of which is also obvious.

What's more, because each part of the code has been split into a different function, we can easily test it. For example, testing whether **improveGuess()** was doing the right thing would be very easy – come up with a few values for x and guess, do the improvement by hand, and then compare your results with those returned by the function.

If **pythag()** itself was found not to return the correct result, we could quickly test all these auxiliary functions to narrow down where the bug was.

And, of course, we can easily reuse any of these new functions. If you were finding the square root of a number in a different function, for instance, you could just call the **sqrt()** function – six characters instead of four lines means there's far less opportunity to make mistakes.

One final point: because our **sqrt** code is now abstracted, we could change the implementation completely, but so long as we kept the function call and arguments the same, all

> **"This code can be improved by taking advantage of scope"**

code that relies on it would continue to work properly.

This means that if you come across a much more efficient way of calculating square roots, you're not stuck with working through thousands of lines of code, manually changing every section that finds a square root; you do it once, and it's done everywhere. This code can be improved still further by taking advantage of scope. **closeEnough()**

```
import math
def square(x):
    return x * x

def sqrt(x, guess):
    def closeEnough(x, guess):
        if math.fabs(square(guess) - x) > 0.01:
            return True
        else:
            return False

    def improveGuess(x, guess):
        return (((x / guess) + guess) / 2)

    while closeEnough(x, guess):
        guess = improveGuess(x, guess)
    return guess

def pythag(a, b):
    a2b2 = square(a) + square(b)
    return sqrt(a2b2, 1.0)

print pythag(2, 3)
~
```

and **improveGuess()** are particular to the **sqrt()** function – that is to say, other functions are unlikely to rely on their services.

To help keep our code clean, and make the relationship between these functions and **sqrt()** clear, we can place their definitions inside the definition of **sqrt()**:

```
def sqrt(x, guess):
    def closeEnough(x, guess):
        ...
    def improveGuess(x, guess):
        ...
    ...
```

These functions are now visible only to code within the **sqrt()** definition – we say they're in the scope of **sqrt()**. Anything outside of it has no idea that they even exist.

This way, if we later need to define similar functions for improving a guess in a different context, we won't face the issue of colliding names or the headache of figuring out what **improveGuess1()** and **improveGuess2()** do.

› Our final code for finding the longest side of a triangle is longer than what we had to start, but it's more readable, more robust, and generally better

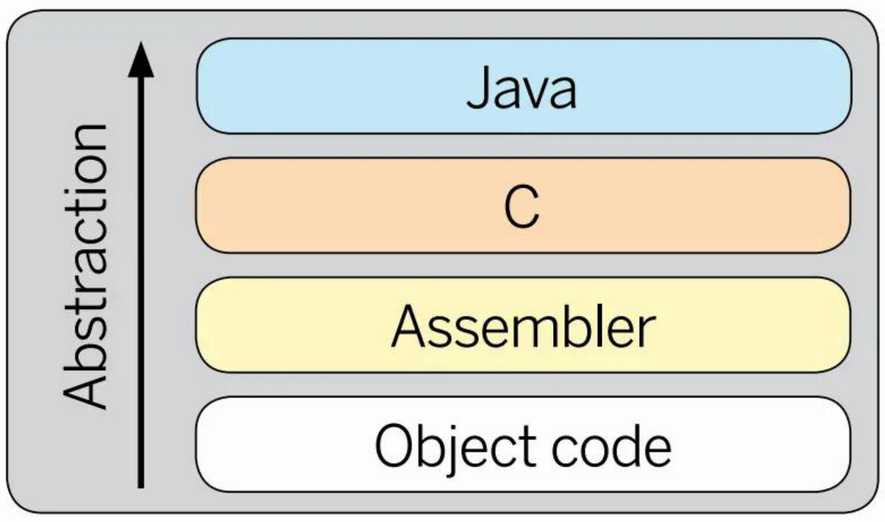
Layers of abstraction

Hopefully, this example has demonstrated how powerful a technique abstraction is. Bear in mind that there are many layers of abstraction present in everything you do on a computer that you never think of.

For instance, when you're programming do you know how Python represents integers in the computer's memory? Or how the CPU performs arithmetic operations such as addition and subtraction?

The answer is probably no. You just accept the fact that typing **2 + 3** in to the Python interpreter returns the correct result, and you never have to worry about how it does this. You treat it as a black box.

Think how much longer it would take you to program if you had to manually take care of what data went in which memory location, to work with binary numbers, and translate alphabetic characters in to their numeric representations – thank goodness for abstraction! ■



› There are layers of abstraction underneath everything you do on a PC – you just don't often think of them

Files and modules done quickly

It's time to expand your library of functions and grab external data with just two simple lines of Python

For the majority of programming projects, you don't get far before facing the age-old problem of how to get data into and out of your application. Whether it's using punched cards to get patterns into a 19th century Jacquard textile loom, or Google's bots skimming websites for data to feed its search engine, dealing with external input is as fundamental as programming itself.

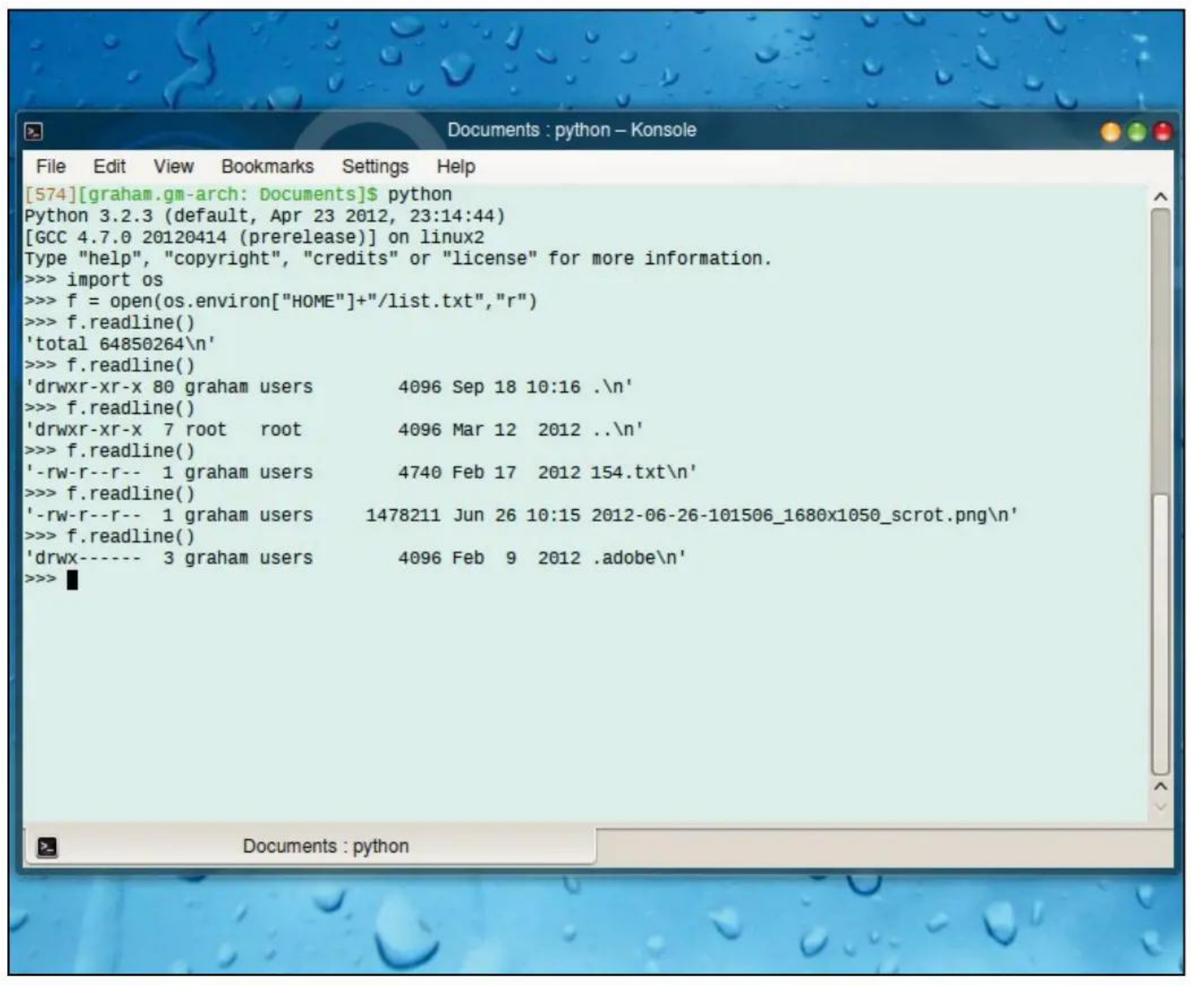
And it's a problem and a concept that you may be more familiar with on the Linux command line. When you type **ls** to list the contents of the current directory, for example, the command is reading in the contents of a file – the current directory – then outputting the contents to another: the terminal.

> **"If the filesystem knows the file is being changed, it won't allow access"**

Of course, the inputs and outputs aren't files in the sense most people would recognise, but that's the way the Linux filesystem has been designed – nearly everything is a file. This helps when you want to save the output of a command, or use that output as the input to another.

You may already know that typing **ls >list.txt** will redirect the output from the command to a file called **list.txt**, but you can take this much further because the output can be treated exactly like a file. **ls | sort -r** will pipe (that's the vertical bar character) the output of **ls** into the input of **sort** to create a reversed alphabetical list of a folder's contents.

› When you read a file, most languages will step through its data from the beginning to the end in chunks you specify. In this example, we're reading a line at a time



The complexity of how data input and output can be accomplished is entirely down to your programming environment. Every language will include functions to load and save data, for instance, but this can either be difficult or easy depending on how many assumptions the language is willing to make on your behalf. However, there's always a logical sequence of events that need to occur.

You will first need to open a file, creating one if it doesn't exist, and then either read data from this file, or write data to it, before explicitly closing the file again so that other processes can use it.

Most languages require you to specify a read mode when you open a file, because this tells the filesystem whether to expect file modifications or not. This is important because many different processes may also want to access the file, and if the filesystem knows the file is being changed, it won't usually allow access. However, many processes can access a read-only file without needing to worry about the integrity of the data it holds, because nothing is able to change it. If you know about databases, it's the same kind of problem you face with multiple users accessing the same table.

In Python, as with most other languages, opening a file to write or as read-only can be done with a single line:

```
>>> f = open("list.txt", "r")
```

If the file doesn't exist, Python will generate a "No such file or directory" error. To avoid this, we've used the output from our command line example to create a text file called **list.txt**. This is within the folder from where we launched the Python interpreter.

Environment variables

Dealing with paths, folders and file locations can quickly become complicated, and it's one of the more tedious issues you'll face with your own projects. You'll find that different environments have different solutions for finding files, with some creating keywords for common locations and others leaving it to the programmer.

This isn't so bad when you only deal with files created by your projects, but it becomes difficult when you need to know where to store a configuration file or load a default icon. These locations may be different depending on your Linux distribution or desktop, but with a cross-platform language such as Python, they'll also be different for each operating system. For that reason, you might want to consider using environment variables. These are similar to variables with

a global scope in many programming languages, but they apply to any one user's Linux session rather than within your own code. If you type **env** on the command line, for instance, you'll see a list of the environmental variables currently set for your terminal session. Look closely, and you'll see a few that apply to default locations and, most importantly, one called **HOME**. The value assigned to this environmental variable will be the location of your home folder on your Linux system, and if we want to use this within our Python script, we first need to add a line to import the operating system-specific module. The line to do this is:

```
import os
```

This command is also opening a file, but not in the same way we opened **list.txt**. This file is known as a module in Python terms, and modules like this 'import' functionality, including statements and definitions, so that a programmer doesn't have to keep re-inventing the wheel.

Modules extend the simple constructs of a language to add portable shortcuts and solutions, which is why other languages might call them libraries. Libraries and modules are a little like copying and pasting someone's own research and insight into your own project. Only it's better than that, because modules such as **os** are used by everyone, turning the way they do things into a standard.

Setting the standard

There are even libraries called **std**, and these embed standard ways of doing many things a language doesn't provide by default, such as common mathematical functions, data types and string services, as well as file input/output and support for specific file types. You will find the documentation for what a library does within an API. This will list each function, what it does, and what it requires as an input and an output. You should also be able to find the source files used by the import (and by **#include** in other languages). On most Linux systems, for example, **/lib/python2.x** will include all the modules. If you load **os.py** into a text editor, you'll see the code you've just added to your project, as well as which functions are now accessible to you.

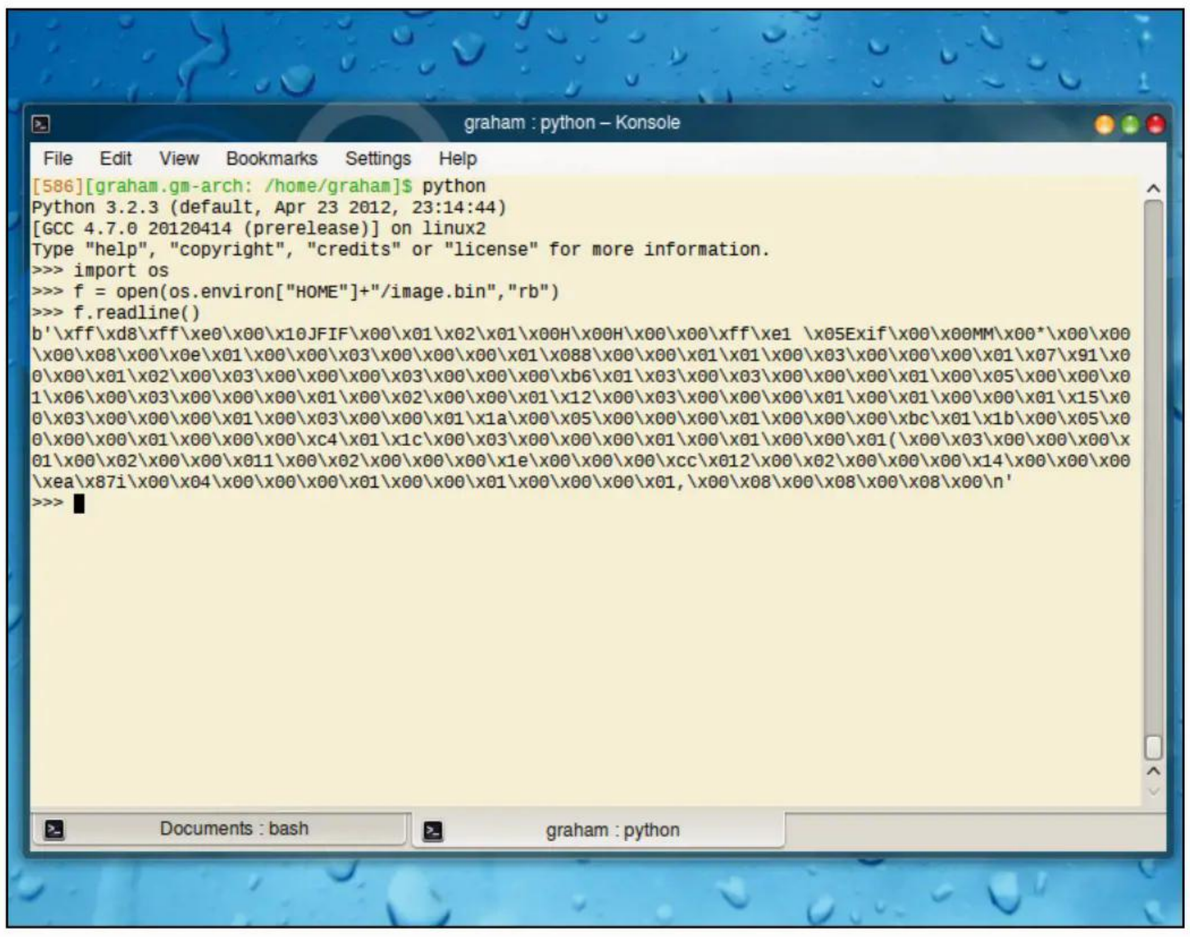
There are many, many different modules for Python – it's one of the best reasons to choose it over any other language, and more can usually be installed with just a couple of clicks from your package manager.

But this is where the ugly spectre of dependencies can start to have an effect on your project, because if you want to give your code to someone else, you need to make sure that person has also got the same modules installed.

If you were programming in C or C++, for example, where your code is compiled and linked against binary libraries, those binary libraries will also need to be present on any other system that runs your code. They will become dependencies for your project, which is what package managers do when you install a complex package.

The os module

Getting back to our project, the **os** module is designed to provide a portable way of accessing operating system-dependent functionality so that you can write multi-platform applications without worrying about where files should be placed. This includes knowing where your home directory might be. To see what we mean, add this code:



› Binary files have no context without an associated file type and a way of handling them. Which is why you get the raw data output when you read one

```
f = open(os.environ["HOME"]+"/list.txt","r")
```

This line will open the file **list.txt** in your home folder. Python knows which home folder is yours, because the **os.environ** function from the **os** module returns a string from an environmental variable, and the one we have asked it to return is **HOME**. But all we've done is open the file, we've not yet read any of its contents. This might seem counter-intuitive, but it's an historical throwback to the way that files used to be stored, which is why this is also the way nearly all languages work. It's only after a file has been opened that you can start to read its contents:

```
f.readline()
```

The above instruction will read a single line of the text file and output this to the interpreter as a string. Repeating the command will read the next line, because Python is remembering how far through the file it has read.

Internally, this is being done using something called a pointer and this, too, is common to the vast majority of languages. Alternatively, if you wanted to read the entire file, you could use **f.read()**. Because our file contains only text, copying the contents to a Python string is an easy conversion.

> **"If you load os.py into a text editor, you'll see the code you've just added"**

The same isn't true of a binary file. Rather than being treated as text, the organisation of the bits and bytes that make up a binary file are organised according to the file type used by the file – or no file type at all if it's raw data. As a result, Python (or any other programming language) would be unable to extract any context from a binary file, causing an error if you try to read it into a string. The solution, at least for the initial input, is to add a **b** flag when you first open the file, because this warns Python to expect raw binary. When you then try to read the input, you'll see the hexadecimal values of the file output to the display.

To make this data useful, you'll need to do some extra work, which we'll look at next; but first, make sure you close the open file, as this should ensure the integrity of your filesystem. As you might guess, the command looks like this:

```
f.close
```

And it's as easy as that! ■

Fundamentals

Write your own UNIX program

Try re-implementing classic Unix tools to bolster your Python knowledge and learn how to build real programs

In the next few pages, that's what we're aiming to do: get you writing real programs. Over the next few tutorials, we're going to create a Python implementation of the popular Unix tool **cat**. Like all Unix tools, **cat** is a great target because it's small and focused on a single task, while using different operating system features, including accessing files, pipes and so on.

This means it won't take too long to complete, but will also expose you to a selection of Python's core features in the Standard Library, and once you've mastered the basics, it's learning the ins-and-outs of your chosen language's libraries that will let you get on with real work.

> **"You now know more than enough to start writing real programs"**

Our goal for the project overall is to:

- Create a Python program, **cat.py**, that when called with no arguments accepts user input on the standard input pipe until an end of line character is reached, at which point it sends the output to standard out.
- When called with file names as arguments, **cat.py** should send each line of the files to standard output, displaying the whole of the first file, then the whole of the second file.
- It should accept two arguments: **-E**, which will make it put **$** signs at the end of each line; and **-n**, which will make it put the current line number at the beginning of each line.

This time, we're going to create a **cat** clone that can work with any number of files passed to it as arguments on the command line. We're going to be using Python 3, so if you want to follow along, make sure you're using the same version, because some features are not backwards-compatible with Python 2.

```
Activities  Terminal
#!/usr/bin/python3
import sys
from optparse import OptionParser

class CatCommand:
    def __init__(self):
        self.count = 1

    def run(self, i, options):
        #set default options
        e = ""

        for line in i:
            #modify printed line according to options
            if options.showend:
                line = line.rstrip()
                e = "$\n"
            if options.shownum:
                #line = str(self.count) + " " + line
                line = "{0} {1}".format(self.count, line)

            self.count += 1
            print(line, end=e)

def main():
    #set up options
    usage = "usage: %prog [OPTION]... [FILE]..."
    parser = OptionParser(usage=usage)
    parser.add_option("-E", dest="showend", action="store_true", help="Show $ at line endi
    parser.add_option("-n", dest="shownum", action="store_true", help="Show line numbers")
    (options, args) = parser.parse_args()

    c = CatCommand()

    #if filenames given
    if len(args) > 1:
        for a in args:
            f = open(a, "r")
            c.run(f, options)

    #otherwise read from stdin
    else:
        c.run(sys.stdin, options)

if __name__ == "__main__":
    main()
~
~
~
~
```

› The final program we'll be implementing. It's not long, but it makes use of a lot of core language features you'll be able to re-use time and again

Python files

Let's start with the easiest part of the problem: displaying the contents of a file, line by line, to standard out. In Python, you access a file with the **open** function, which returns a file-object that you can later read from, or otherwise manipulate. To capture this file-object for use later in your program, you need to assign the result of running the **open** function to a variable, like so:

```
file = open("hello.txt", "r")
```

This creates a variable, **file**, that will later allow us to read the contents of the file **hello.txt**. It will only allow us to read from this file, not write to it, because we passed a second argument to the open function, **r**, which specified that the file should be opened in read-only mode.

With access to the file now provided through the newly-created **file** object, the next task is to display its contents, line by line, on standard output. This is very easy to achieve, because in Python files are iterable objects.

Iterable objects, such as lists, strings, tuples and dictionaries, allow you to access their individual member elements one at a time through a **for** loop. With a file, this means you can access each line contained within simply by putting it in a **for** loop, as follows:

```
for line in file:
        print(line)
```

The **print** function then causes whatever argument you pass to it to be displayed on standard output.

If you put all this in a file, make it executable and create a **hello.txt** file in the same directory, you'll see that it works rather well. There is one oddity, however – there's an empty line between each line of output.

The reason this happens is that **print** automatically adds a newline character to the end of each line. Because there's already a newline character at the end of each line in **hello.txt** (there is, even if you can't see it, otherwise everything would be on one line!), the second newline character leads to an empty line.

You can fix this by calling **print** with a second, named argument such as: **print(line, end="")**. This tells **print** to put an empty string, or no character, at the end of each line instead of a newline character.

Passing arguments

This is all right, but compared to the real **cat** command, there's a glaring omission here: we would have to edit the program code itself to change which file is being displayed to standard out. What we need is some way to pass arguments on the command line, so that we could call our new program by typing **cat.py hello.txt** on the command line. Since Python has 'all batteries included', this is a fairly straightforward task, as well.

The Python interpreter automatically captures all arguments passed on the command line, and a module called **sys**, which is part of the Standard Library, makes this available to your code.

Even though **sys** is part of the Standard Library, it's not available to your code by default. Instead, you first have to import it to your program and then access its contents with dot notation – don't worry, we'll explain this in a moment. First, to import it to your program, add:

```
import sys
```

to the top of your **cat.py** file.

The part of the **sys** module that we're interested in is the **argv** object. This object stores all of the arguments passed on the command line in a Python list, which means you can access and manipulate it using various techniques we've seen in previous tutorials and will show in future ones.

There are only two things you really need to know about this. They are:

- The first element of the list is the name of the program itself – all arguments follow this.
- To access the list, you need to use dot notation – that is to say, **argv** is stored within **sys**, so to access it, you need to type **sys.argv**, or **sys.argv[1]** to get the first argument to your program.

Knowing this, you should now be able to adjust the code we created previously by replacing **hello.txt** with **sys.argv[1]**. When you call **cat.py** from the command line, you can then pass the name of any text file, and it will work just the same.

Many files

Of course, our program is meant to accept more than one file and output all their contents to standard output, one after another, but as things stand, our program can only accept one file as an argument.

To fix this particular problem, you need to loop over all the files in the **argv** list. The only thing that you need to be careful of when you do this is that you exclude the very first element, because this is the name of the program itself. If you think back to our previous article on data types and common list operations, you'll realise this is easily done with a slice.

This is just one line:

```
for file in sys.argv[1:]:
```

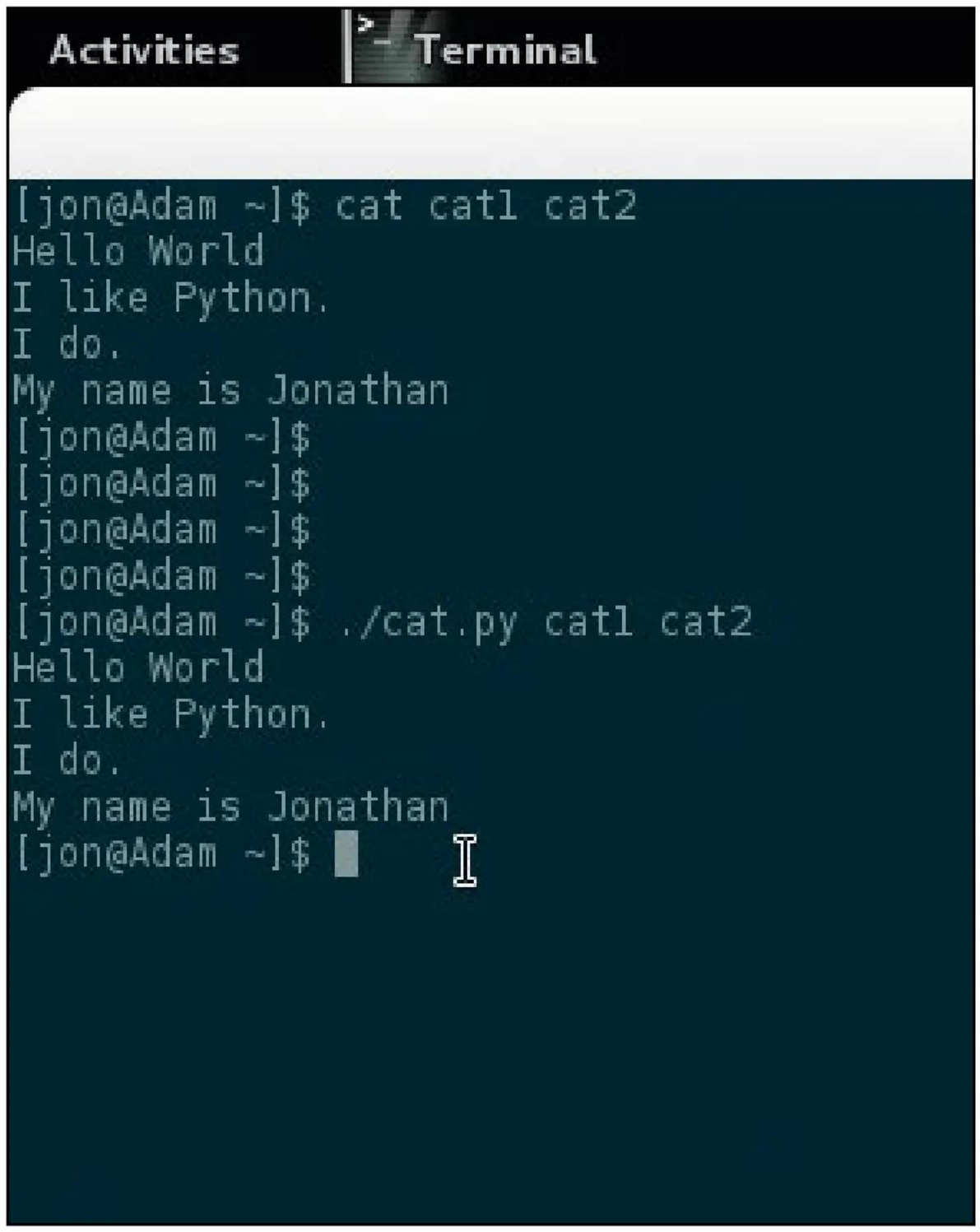
Because operating on all the files passed as arguments to a program is such a common operation, Python provides a shortcut for doing this in the Standard Library, called **fileinput**.

In order to use this shortcut, you must first import it by putting **import fileinput** at the top of your code. You will then be able to use it to recreate the rest of our **cat** program so far, as follows:

```
for line in fileinput.input():
        print(line, end="")
```

This simple shortcut function takes care of opening each file in turn and making all their lines accessible through a single iterator.

That's about all that we have space for in this tutorial. Although there has not been much code in this particular example, we hope you have started to get a sense for how much is available in Python's Standard Library (and therefore how much work is available for you to recycle), and how a good knowledge of its contents can save you a lot of work when implementing new programs. ■



› The output of the real Unix command, cat, and our Python re-implementation, are exactly the same in this simple example

> **"The part of the sys module we're interested in is the argv object"**

Enhance your UNIX program

Our tour of the Python programming language continues, as we continue our clone of the Unix cat command

The previous tutorial showed you how to build a simple **cat** clone in Python. In this guide, we're going to add some more features to our program, including the ability to read from the standard input pipe, just like the real **cat**, and the ability to pass options to your **cat** clone. So, without further delay, let's dive in.

Fortunately, you already know everything you need to interact with the standard input pipe. In Linux, all pipes are treated like files – you can pass a file as an argument to a command, or you can pass a pipe as an argument. It doesn't matter which you do, because they're basically the same thing.

"Python provides us with a much more powerful alternative to sys.argv"

In Python, the same is true. All you need to get to work with the standard input pipe is access to the sys library, which if you followed along last time, you already have. Let's write a little sample program first to demonstrate:

```
import sys
for line in sys.stdin:
        print(line, end="")
```

The first line imports the **sys** module. The lines that follow are almost identical to those that we had last time. Rather than specifying the name of a file, however, we specified the name of the file-object, **stdin**, which is found inside the **sys** module. Just like a real file, in Python the standard input pipe is an iterable object, so we use a **for** loop to walk through each line.

You might be wondering how this works, though, because standard input starts off empty. If you run the program, you will see what happens. Rather than printing out everything that's present straightaway, it will simply wait. Every time a new line character is passed to standard input (by pressing [Return]), it will then print everything that came before it to standard output.

Right, now we have two modes that our program can operate in, but we need to put them together into a single program. If we call our program with arguments, we want it to work like last time – that is, by concatenating the files' contents together; if it's called without any arguments, we want our program to work by repeating each line entered into standard input. We could easily do this with what we have learned so far – simply check to see what the length of the **sys.argv** array is. If it's greater than 1, then do last lesson's version, otherwise do this version:

```
if len(sys.argv) > 1:
        [last month...]
else:
        [this month...]
```

Pretty straightforward. The only point of interest here is the use of the **len()** function, seeing as we're on a journey to discover different Python functions. This function is built in to Python, and can be applied to any type of sequence object (a string, tuple or list) or a map (like a dictionary), and it always tells you how many elements are in that object.

There are more useful functions like this, which you can find at **http://docs.python.org/3/library/functions.html**.

Parsing arguments and options

This is quite a simplistic approach, however, and Python actually provides us with a much more powerful alternative to **sys.argv**. To demonstrate this, we are going to add two options to our program that will modify the output generated by our program.

You may not have realised it, but **cat** does in fact have a range of options. We're going to implement the **-E**, which shows dollar symbols at the end of lines, and **-n**, which displays line numbers at the beginning of lines.

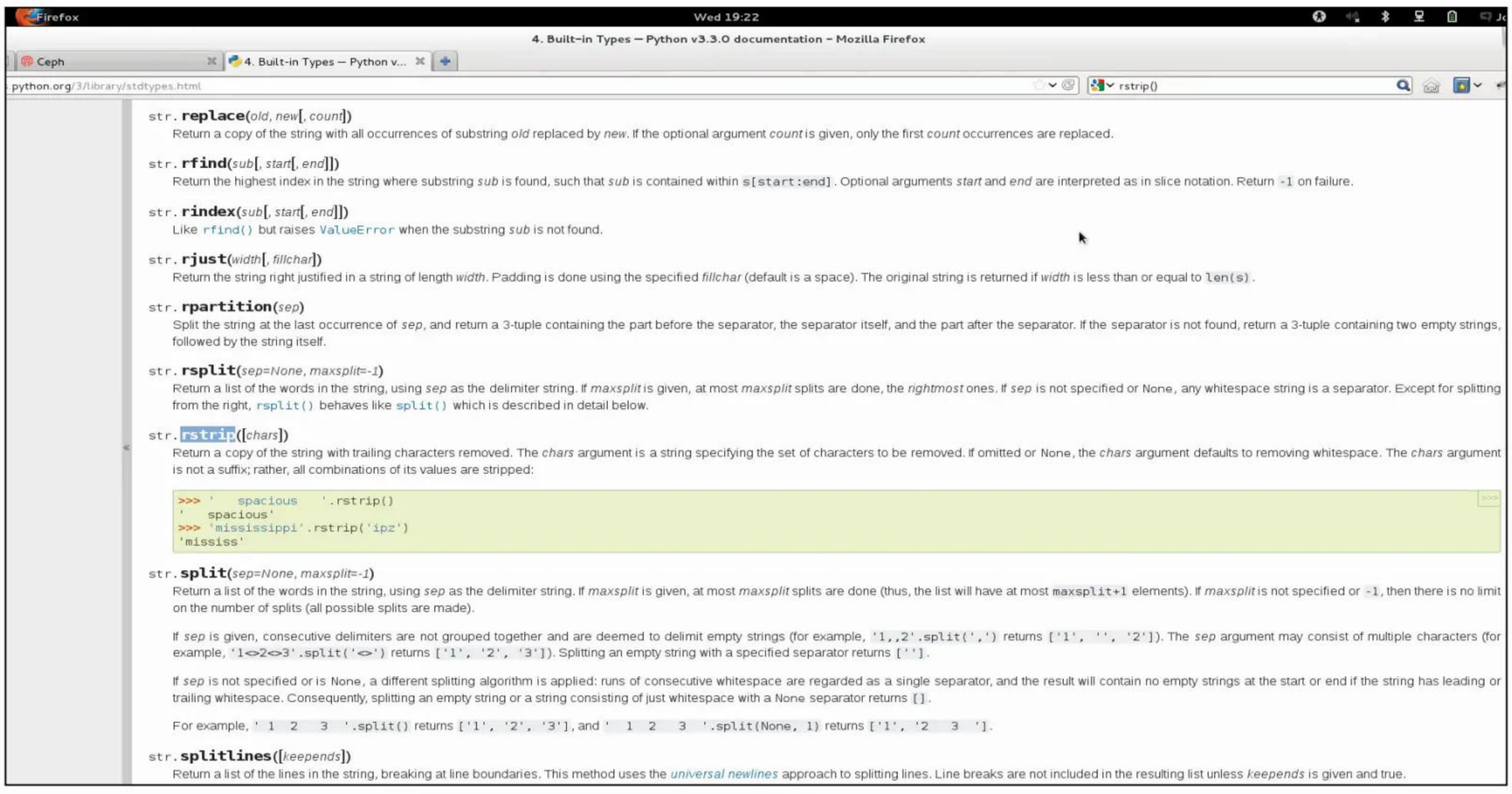
To do this, we'll start by setting up an **OptionParser**. This is a special object, provided as part of the **optparse** module, which will do most of the hard work for you. As well as automatically detecting options and arguments, saving you a lot of hard work, **OptionParser** will automatically generate help text for your users in the event that they use your program incorrectly or pass **--help** to it, like this:

```
[jon@LT04394 ~]$ ./cat.py --help
Usage: cat.py [OPTION]... [FILE]...
Options:
```

```
>>> print_lots("I will not use white space.")
'I will not use white space.I will not use white space.I will not use white sp
ill not use white space.I will not use white space.I will not use white space.
not use white space.I will not use white space.I will not use white space.I wi
use white space.I will not use white space.I will not use white space.I will n
white space.I will not use white space.I will not use white space.I will not u
e space.I will not use white space.I will not use white space.I will not use w
ace.I will not use white space.I will not use white space.I will not use white
I will not use white space.I will not use white space.I will not use white spa
ll not use white space.I will not use white space.I will not use white space.I
ot use white space.I will not use white space.I will not use white space.I wil
se white space.I will not use white space.I will not use white space.I will no
hite space.I will not use white space.I will not use white space.I will not us
 space.I will not use white space.I will not use white space.I will not use wh
ce.I will not use white space.I will not use white space.I will not use white
>>> print_lots("I will not use white space.").replace(" ", "")
'IwillnotusewhitespaceIwillnotusewhitespaceIwillnotusewhitespaceIwillnotuse
```

› The Python language comes with all the bells and whistles you need to write useful programs. In this example, you can see the replace method applied to a string in order to remove all white space – in the tutorial, we used the rstrip method for a similar purpose

Enhance your UNIX program



› The Python 3 website provides excellent documentation for a wealth of built-in functions and methods. If you ever wonder how to do something in Python, http://docs.python.org/3/ should be your first port of call

```
-h, --help  show this help message and exit
-E      Show $ at line endings
-n      Show line numbers
```

Just like a real program! To get started with **OptionParser**, you first need to import the necessary components:

```
from optparse import OptionParser
```

You may notice that this looks a bit different from what we saw before. Instead of importing the entire module, we're only importing the **OptionParser** object. Next, you need to create a new instance of the object, add some new options for it to detect with the **add_option** method, and pass it a **usage** string to display:

```
usage = "usage: %prog [option]... [file]..."
parser = OptionParser(usage=usage)
parser.add_option("-E", dest="showend", action="store_
true", help="Show $ at line
endings")
parser.add_option("-n", dest="shownum", action="store_
true", help="Show line numbers")
```

The **%prog** part of the **usage** string will be replaced with the name of your program. The **dest** argument specifies what **name** you'll be able to use to access the value of an argument once the parsing has been done, while the **action** specifies what that value should be. In this case, the action **store_true** says to set the **dest** variable to **True** if the argument is present, and **False** if it is not. You can read about other actions at **http://docs.python.org/3/library/optparse.html**.

Finally, with everything set, you just need to parse the arguments that were passed to your program and assign the results to two array variables:

```
(options, args) = parser.parser_args()
```

The **options** variable will contain all user-defined options, such as **-E** or **-n**, while **args** will contain all positional arguments left over after parsing out the options. You can call these variables whatever you like, but the two will always be set in the same order, so don't confuse yourself by putting the variables the other way around! With the argument-parsing code written, you'll next want to start implementing the code that will run when a particular option is set. In both cases, we'll be modifying the string of text that's output by the program, which means you'll need to know a little bit about Python's built-in string editing functions. Let's think about the **-E**, or **showend**, option first. All we want this to do is remove the invisible line break that's at the end of every file (or every line of the standard input pipe, as implied by pressing [Return]), and replace it with a dollar symbol followed by a line break.

The first part, removing the existing new line, can be achieved by the **string.rstrip()** method. This removes all white space characters by default, at the right-hand edge of a string.

If you pass a string to it as an argument, it will strip those characters from the right-hand edge instead of white space. In our case, just white space will do.

Completing the job

The second part of the job is as simple as setting the end variable in the **print** statement to the string **$\n** and the job is almost complete. We say "almost complete" because we still need to write some more logic to further control the flow of the program based on what options were set, as well as whether or not any arguments are passed.

> **"Don't confuse yourself by putting the variables the other way around!"**

The thing is, this logic needs to be a little more complicated than it ordinarily would be because we need to maintain a cumulative count of lines that have been printed as the program runs to implement the second **-n**, or **shownum**, option.

While there are several ways you could achieve this, in the next tutorial we're going to introduce you to a bit of object-oriented programming in Python and implement this functionality in a class.

We'll also introduce you to a very important Python convention – the **main()** function and the **name** variable. In the meantime, you can keep yourself busy by investigating the **string.format()** method and see whether you can figure out how you can append a number to the beginning of each line. ■

Fundamentals

Finish up your UNIX program

Our guide to the Python programming language continues. This tutorial, we're going to finish our clone of cat

We've come quite a long way over the last two tutorials, having implemented the ability to echo the contents of multiple files to the screen, the ability to echo standard input to the screen and the ability to detect and act upon options passed by the user of our program. All that remains is for us to implement the line number option and to gather together everything else we've written into a single, working program.

Objects

Last time, we ended by saying that there are many ways we could implement the line counting option in our program. We're going to show you how to do it in an object-oriented style, because it gives us an excuse to introduce you to this aspect of Python programming. You could, however, with a little careful thought, implement the same function with some nested **for** loops, although they're not nearly as readable as object-oriented code.

> **"It's a very natural way of thinking because it mirrors the real world"**

When building complicated programs, figuring out how to organise them so they remain easy to read, easy to track which variables are being used by which functions, and easy to update, extend, or add new features, can be challenging. To make this easier, there are various paradigms that provide techniques for managing complexity.

One of these paradigms is the concept of object-oriented programming. In object-oriented programming, the elements of the program are broken down into objects which contain state – variables, in other words – that describe the current condition of the object, and methods, that allow us to perform actions on those variables or with that object.

It's a very natural way of thinking, because it mirrors the real world so closely. We can describe a set of properties about your hand, such as having five fingers that are in certain locations, and we can describe certain methods or things you can do with your hand, such as moving one finger to press a key, or holding a cup. Your hand is an object, complete with state and methods that let you work with it.

We're going to turn our **cat** program into an object, where its state records how many lines have been displayed, and its methods perform the action of the **cat** program – redisplaying file contents to the screen.

```
[jon@Adam Linux Format]$ ./cat.py cat1 cat2
Hello World
I like Python.
I do.
My name is Jonathan
[jon@Adam Linux Format]$ ./cat.py --help
Usage: cat.py [OPTION]... [FILE]...

Options:
  -h, --help  show this help message and exit
  -E          Show $ at line endings
  -n          Show line numbers
[jon@Adam Linux Format]$ ./cat.py -nE cat1 cat2
1 Hello World$
2 I like Python.$
3 I do.$
4 My name is Jonathan$
[jon@Adam Linux Format]$ ./cat.py
hello
hello
world
world
hello
hello
lxf
lxf
^CTraceback (most recent call last):
  File "./cat.py", line 46, in <module>
    main()
  File "./cat.py", line 43, in main
    c.run(sys.stdin, options)
  File "./cat.py", line 13, in run
    for line in i:
KeyboardInterrupt
[jon@Adam Linux Format]$
```

› **Just to prove that it works, here's our cat implementation, with all of the options being put to use**

Python objects

Python implements objects through a class system. A class is a template, and an object is a particular instance of that class, modelled on the template. We define a new class with a keyword, much like we define a new function:

```
class catCommand:
```

Inside the class, we specify the methods (functions) and state that we want to associate with every instance of the object. There are some special methods, however, that are frequently used. One of these is the **init** method. This is run when the class is first instantiated into a particular object, and allows you to set specific variables that you want to belong to that object.

```
def __init__(self):
    self.count = 1
```

In this case, we've assigned **1** to the **count** variable, and we'll be using this to record how many lines have been displayed. You probably noticed the **self** variable, passed as the first argument to the method, and wondered what on earth that

was about. Well, it is the main distinguishing feature between methods and ordinary functions.

Methods, even those that have no other arguments, must have the **self** variable. It is an automatically populated variable, which will always point to the particular instance of the object that you're working with. So **self.count** is a count variable that is exclusive to individual instances of the **catCommand** object.

The run method

We next need to write a method that will execute the appropriate logic depending on whether certain options are set. We've called this the **run** method:

```
def run(self, i, options):
    #set default options
    e = “”
    for line in i:
        #modify printed line according to options
        if options.showend:
            [...last time]
        if options.shownum:
            line = “{0} {1}”.format(self.count, line)
        self.count += 1
        print(line, end=e)
```

Notice that we've passed the **self** variable to this method, too. The two other arguments passed to this function are arguments that we'll pass when we call the method later on, just like with a normal function. The first, **i**, is going to be a reference to whichever file is being displayed at this moment, while the **options** variable is a reference to the options decoded by the **OptParse** module.

The logic after that is clear – for each line in the current file, modify the line depending on what options have been set. Either we do as last time, and modify the end character to be **“$\n”** or we modify the line, using the **.format** method that we suggested you research last time, to append the **count** variable, defined in the **init** method, to the rest of the line. We then increment the count and print the line.

The most important part is the use of **self**. It lets us refer to variables stored within the current instance of the object. Because it's stored as part of the object, it will persist after the current execution of the **run** method ends. As long as we use the **run** method attached to the same object each time we **cat** a new file in the argument list, the count will remember how many lines were displayed in the last file, and continue to count correctly.

It might seem more natural – given the description of methods as individual actions that can be taken by our objects – to split each argument into a different method, and this is a fine way to approach the problem.

> **“The last thing to do is call the main function when the program runs”**

The reason we've done it this way, though, is that we found it meant we could re-use more code, making it more readable and less error-prone.

Now all that's left to do is to tie everything together. We are going to do this by writing a main function. This isn't actually required in Python, but many programs follow this idiom, so we will too:

```
def main():
    [option parsing code ...]
    c = catCommand()
    if len(args) > 1:
        for a in args:
            f = open(a, “r”)
            c.run(f, options)
    else:
        c.run(sys.stdin, options)
```

We haven't filled in the object parsing code from the previous tutorial, because that hasn't changed. What is new is the **c = catCommand()** line. This is how we create an instance of a class – how we create a new object. The **c** object now has a variable, **count**, which is accessible by all its methods as the **self.count** variable. This is what will enable us to track the line numbers.

We then check to see whether any arguments have been passed. If they have, then we call the **run** method of the object **c** for each file that was passed as an argument, passing in any options extracted by **OptParse** along the way. If there weren't any arguments, though, we simply call the **run** method with **sys.stdin** instead of a file object.

The final thing we need to do here is actually call the main function when the program is run:

```
if __name__ == “__main__”:
    main()
```

These last two lines are the strangest of all, but quite useful in a lot of circumstances. The **name** variable is special – when the program is run on the command line, or otherwise as a standalone application, it is set to **main**; however, when it is imported as an external module to other Python programs, it's not.

In this way, we can automatically execute **main** when run as a standalone program, but not when we're importing it as a module. ■

cat.py

```
#!/usr/bin/python3
import sys
from optparse import OptionParser

class CatCommand:
    def __init__(self):
        self.count = 1

    def run(self, i, options):
        #set default options
        e = ""

        for line in i:
            #modify printed line according to options
            if options.showend:
                line = line.rstrip()
                e = "$\n"
            if options.shownum:
                #line = str(self.count) + " " + line
                line = "{0} {1}".format(self.count, line)

            self.count += 1
            print(line, end=e)

def main():
    #set up options
    usage = "usage: %prog [OPTION]... [FILE]..."
    parser = OptionParser(usage=usage)
    parser.add_option("-E", dest="showend", action="store_true", help="Show $ at line endings")
    parser.add_option("-n", dest="shownum", action="store_true", help="Show line numbers")
    (options, args) = parser.parse_args()

    c = CatCommand()

    #if filenames given
    if len(args) > 1:
        for a in args:
            f = open(a, "r")
            c.run(f, options)

    #otherwise read from stdin
    else:
        c.run(sys.stdin, options)

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

› The completed program isn't very long, but it has given us a chance to introduce you to many different aspects of the Python language

Fundamentals

Neater code with modules

Untangle the horrible mess of code that you've made and add coherent structure to your programs

In our previous tutorials, we have mentioned how programming is all about managing complexity, and we've introduced you to quite a few of the tools and techniques that help programmers do this. From variables to function definitions or object-orientation, they all help. One tool we've yet to cover, in part because you don't start to come across it until you're writing larger programs, is the idea of modules and name spaces.

Yet, if you've written a program of any length, even just a few hundred lines, this is a tool you're no doubt desperate for. In a long file of code, you'll have noticed how quickly it becomes more difficult to read. Functions seem to blur into one another, and when you're trying to hunt down the cause of the latest error, you find it difficult to remember exactly where you defined that all-important variable.

These problems are caused by a lack of structure. With all your code in a single file, it's more difficult to determine the dependencies between elements of your program – that is, which parts rely on each other to get work done – and it's more difficult to visualise the flow of data through your program.

> **"As your programs grow in length, there are other problems that occur"**

As your programs grow in length, too, there are other problems that begin to occur: for instance, you may find yourself with naming conflicts, as two different parts of your program require functions called **add** (for example, adding integers or adding fractions in a mathematics program), or you may have written a useful function that you want to share with other programs that you're writing, and the only tool you have to do that is boring and error-prone copy and pasting.

Untangling the mess

Modules are a great way to solve all of these problems, enabling you to put structure back into your code, letting you avoid naming conflicts, and making it easier for you to share useful chunks of code between programs.

You've no doubt been using them all the time in your code as you've relied on Python built-in or third-party modules to provide lots of extra functionality. As an example, remember the **optparse** module we used before. We included it in our program along with the **import** statement, like so:

```
import optparse
```

After putting this line at the top of our Python program, we magically got access to a whole load of other functions that automatically parsed command line options. We could access them by typing the module's name, followed by the name of the function we wanted to execute:

```
optparse.OptionParser()
```

This was great from a readability perspective. In our **cat** clone, we didn't have to wade through lots of code about parsing command line arguments; instead we could focus on the code that dealt with the logic of echoing file contents to the screen and to the standard output pipe. What's more, we didn't have to worry about using names in our own program that might collide with those in the **optparse** module, because they were all hidden inside the **optparse** namespace, and reusing this code was as easy as typing **import optparse** – no messy copy and pasting here.

How modules work

Modules sound fancy, and you might think they're complicated, but – in Python at least – they're really just plain old files. You can try it out for yourself. Create a new directory and inside it create a **fact.py** file. Inside it, define a function to return the factorial of a given number:

```
def factorial(n):
        result = 1
        while n > 0:
                if n == 1:
                        result *= 1
                else:
                        result *= n
                n -= 1
        return n
```

Create a second Python file called **doMath.py**. Inside this, first import the module you just created and then execute the **factorial** function, printing the result to the screen:

```
import fact
print fact.factorial(5)
```

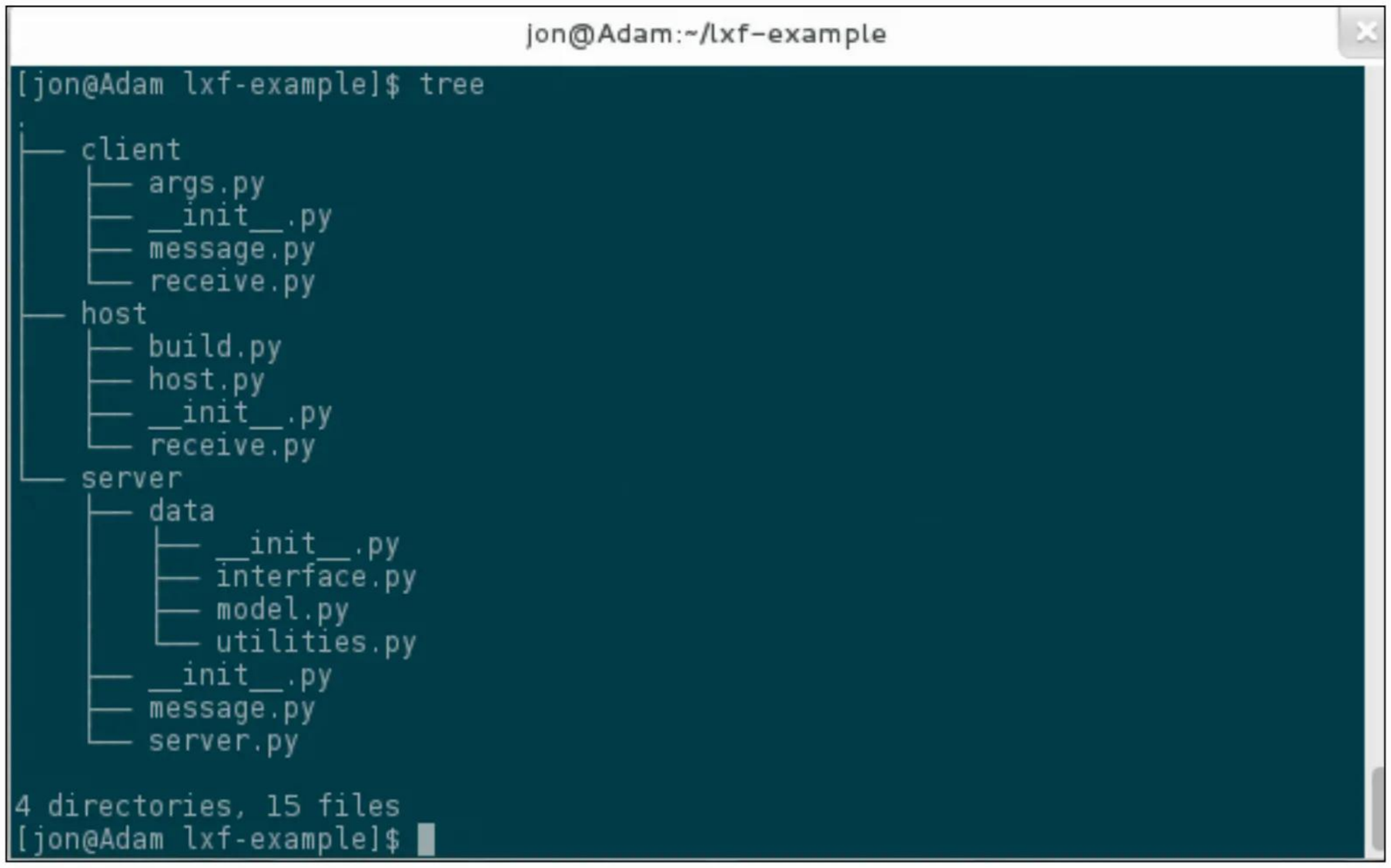
Now, when you run the **doMath.py** file, you should see 120 printed on the screen. You should notice that the name of the module is just the name of the file, in the same directory, with the extension removed.

We can then call any function defined in that module by typing the module's name, followed by a dot, followed by the function name.

The Python path

The big question that's left is, how does Python know where to look to find your modules?

The answer is that Python has a pre-defined set of locations that it looks in to find files that match the name specified in your import statements. It first looks inside all of the built-in modules, the location of which are defined when



› By splitting your code up in to smaller chunks, each placed in its own file and directory, you can bring order to your projects and make future maintenance a lot easier

you first install Python; it then searches through a list of directories known as the path.

This path is much like the Linux *Bash* shell's **$PATH** environment variable – it uses the same syntax, and serves exactly the same function. It varies, however, in how the contents of the Python path are generated. Initially, the locations stored in the path consist of the following two locations:

» The directory containing the script doing the importing.

» The **PYTHONPATH**, which is a set of directories pre-defined in your default installation.

You can inspect the path in your Python environment by importing the **sys** module, and then inspecting the path attribute (typing **sys.path** will do the trick). Once a program has started, it can even modify the path itself and add other locations to it.

Variable scope in modules

Before you head off and start merrily writing your own modules, there is one more thing that you need to know about: variable scope.

We have touched upon scope as a concept before, but as a quick refresher, scope refers to the part of a program from which particular variables can be accessed. For instance, a single Python module might contain the following code:

```
food = [“apples”, “oranges”, “pears”]
print food
def show_choc():
    food = [“snickers”, “kitkat”, “dairy milk”]
    print food
show_choc()
print food
```

If you run that, you'll see that outside the function the variable **food** refers to a list of fruit, while inside the function, it refers to a list of chocolate bars. This demonstrates two different scopes: the global scope of the current module, in which food refers to a list of fruit, and the local scope of the function, in which food refers to a list of chocolate.

When looking up a variable, Python starts with the innermost variable and works its way out, starting with the immediately enclosing function, and then any functions enclosing that, and then the module's global scope, and then finally it will look at all the built-in names.

In simple, single-file programs, it is a bad idea to put variables in the global scope. It can cause confusion and subtle errors elsewhere in your program. Using modules can help with this problem, because each module has its own global scope. As we saw above, when we import a module, its contents are all stored as attributes of the module's name, accessed via dot notation. This makes global variables somewhat less troublesome, although you should still be careful when using them. ■

Python style

While many people think of Python as a modern language, it's actually been around since the early 1990s. As with any programming language that's been around for any length of time, people who use it often have learned a lot about the best ways to do things in the language – in terms of the easiest way to solve common problems, and the best ways to format your code to make sure it's readable for co-workers and anyone else working on the code with you (including your future self).

If you're interested in finding out more about these best practices in Python, there are two very useful resources from which you can start learning:

» **http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html**

» **www.python.org/dev/peps/pep-0008**

Read these and you're sure to gain some deeper insight into the language.

Embrace storage and persistence

Deal with persistent data and store your files in Python to make your programs more permanent and less transient

Storage is cheap: you can buy a 500GB external hard drive for less than £35 these days, while even smartphones come with at least 8GB of storage, and many are easily expandable up to 128GB for only the price of a pair of jeans.

It's no surprise, then, that almost every modern application stores data in one way or another, whether that's configuration data, cached data to speed up future use, saved games, to-do lists or photos. The list goes on and on.

With this in mind, this programming tutorial is going to demonstrate how to deal with persistent data in our language of choice – Python.

The most obvious form of persistent storage that you can take advantage of in Python is file storage. Support for it is included in the standard library, and you don't even have to import any modules to take advantage of it.

To open a file in the current working directory (that is, wherever you ran the Python script from, or wherever you were when you launched the interactive shell), use the **open()** function:

```
file = open("lxf-test.txt", "w")
```

The first argument to open is the filename, while the second specifies which mode the file should be opened in – in this case, it's write, but other valid options include read-only (**r**) and append (**a**).

> **"Almost every modern application stores data in one way or another"**

In previous tutorials, we've shown you that this file object is in fact an iterator, which means you can use the **in** keyword to loop through each line in the file and deal with its contents, one line at a time. Before reviewing that information, however, let's look at how to write data to a file.

Writing to files

Suppose you're writing your own RSS application to deal with your favourite feeds. You've already got some way to ask users to enter in a list of feeds (perhaps using **raw_input()**, or perhaps using a web form and CGI), but now you want to store that list of feeds on disk so you can re-use it later when you're checking for new updates. At the moment, the feeds are just stored in a Python list:

```
feeds = ["http://newsrss.bbc.co.uk/rss/newsonline_uk_
edition/front_page/rss.xml", "http://www.tuxradar.com/rss"]
```

To get the feeds into the file is a simple process. Just use the **write** method:

```
for feed in feeds:
        file.write("{0}\n".format(feed))
```

Easy! Notice how we used the **format string** function to add a new line to the end of each string, otherwise we would end up with everything on one line – which would have made it harder to use later.

Re-using the contents of this file would be just as simple. Using the file as an iterator, load each line in turn into a list, stripping off the trailing new line character. We'll leave you to figure this one out.

When using files in your Python code, there are two things that you need to keep in mind. The first is that you need to convert whatever you want to write to the file to a string first. This is easy, though, because you can just use the built-in **str()** function – for example, **str(42) => "42"**.

The second is that you have to close the file after you have finished using it – if you don't do this, you risk losing data that you thought had been committed to disk, but that had not yet been flushed. You can do this manually with the **close** method of file objects. In our example, this would translate to adding **file.close()** to our program. It's better, however, to use the **with** keyword:

```
with open("lxf-test.txt", "a") as file:
        feeds = [line.rstrip("\n") for line in f]
```

This simple piece of Python code handles opening the file object and, when the block inside the **with** statement is finished, automatically closes it for us, as well. If you are unsure what the second line does, try looking up Python list comprehensions – they're a great way to write efficient, concise code, and to bring a modicum of functional style into your work.

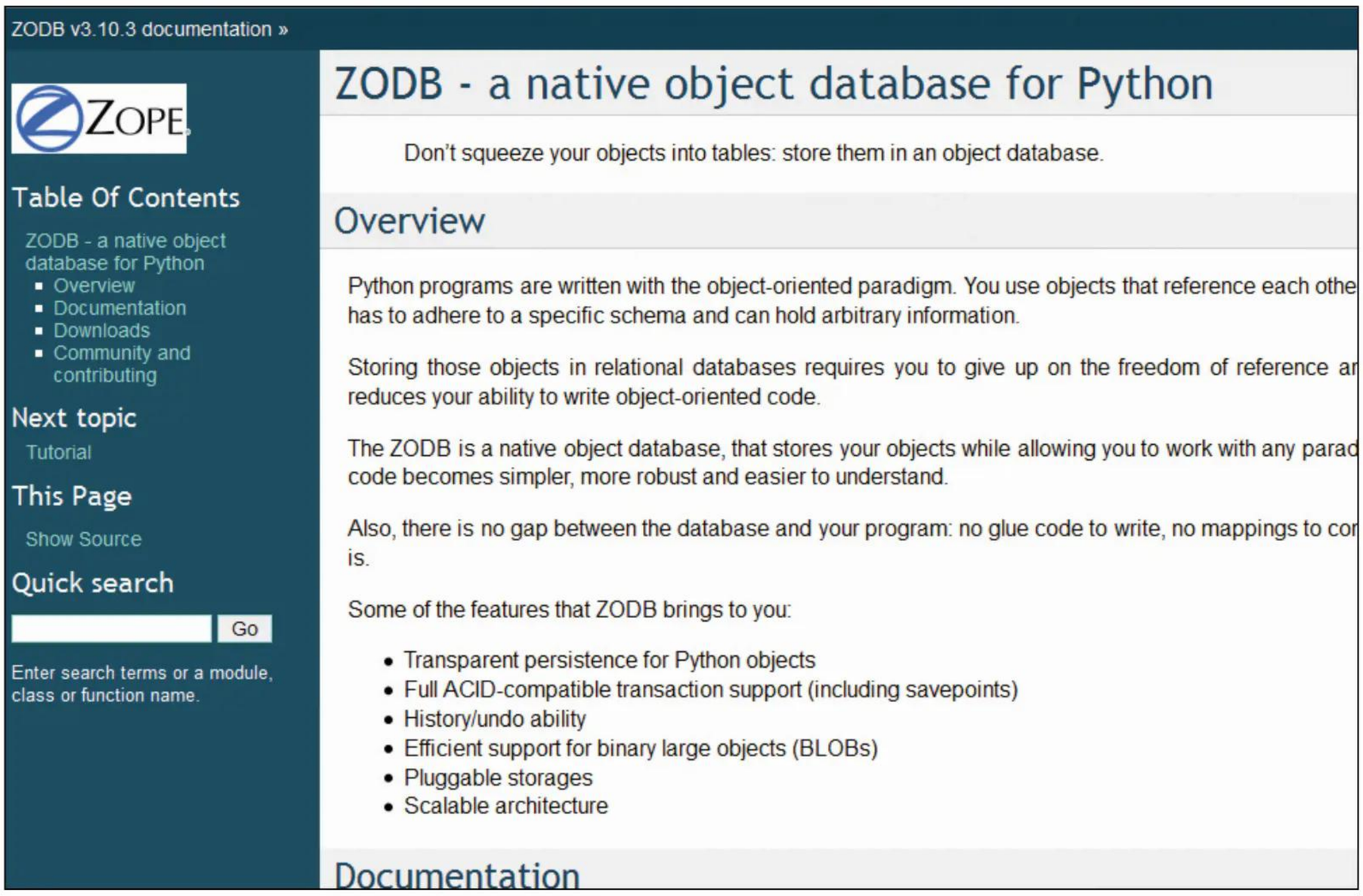
Serialising

Working with files would be much easier if you didn't have to worry about converting your list (or dictionary, for that matter) into a string first of all – for dictionaries in particular, this could get messy. Fortunately, Python provides two tools to make this easier for us.

The first of these tools is the **pickle** module. **Pickle** accepts many different kinds of Python objects, and can then convert them to a character string and back again. You still have to do the file opening and closing, but you no longer have to worry about figuring out an appropriate string representation for your data:

```
import pickle
...
with open("lxf-test.txt", "a") as file:
        pickle.dump(feeds, file)
...
with open("lxf-test.txt", "r") as file:
        feeds = pickle.load(file)
```

Embrace storage and persistence



› If you're interested in persistent data in Python, a good next stopping point is the ZODB object database. It's much easier and more natural in Python than a relational database engine (www.zodb.org)

This is much easier, and it has other applications outside of persisting data in files, too. For example, if you wanted to transfer your feed list across the network, you would first have to make it into a character string, too, which you could do with **pickle**.

The problem with this is that it will only work in Python – that is to say, other programming languages don't support the pickle data format. If you like the concept of pickling (more generically, serialising), there's another option that does have support in other languages, too: JSON.

You may have heard of JSON – it stands for JavaScript Object Notation, and is a way of converting objects into human-readable string representations, which look almost identical to objects found in the JavaScript programming language. It's great, because it's human readable, and also because it's widely supported in many different languages, largely because it has become so popular with fancy web 2.0 applications.

In Python, you use it in exactly the same way as **pickle** – in the above example, replace **pickle** with **json** throughout, and you'll be writing interoperable, serialised code!

Shelves

Of course, some code bases have many different objects that you want to store persistently between runs, and keeping track of many different pickled files can get tricky. There's another Python standard module, however, that uses **pickle** underneath, but makes access to the stored objects more intuitive and convenient: the **shelve** module.

Essentially, a shelf is a persistent dictionary – that is to say, a persistent way to store key-value pairs. The great thing about shelves, however, is that the value can be any Python object that **pickle** can serialise. Let's take a look at how you can use it. Thinking back to our RSS reader application, imagine that as well as the list of feeds to check, you wanted to keep track of how many unread items each feed had, and which item was the last to be read. You might do this with a dictionary, for example:

```
tracker = {"bbc.co.uk":          {"last-read": "foo",
                                   "num-unread": 10,},
           "linuxformat.com":     {"last-read": "bar",
                                   "num-unread": 5,}}
```

You could then store the list of feeds and the tracking details for each in a single file by using the **shelve** module, like so:

```
import shelve
shelf = shelve.open("lxf-test")
shelf["feeds"] = feeds
shelf["tracker"] = tracker
shelf.close()
```

There are a few important things that you should be aware of about the **shelve** module:

» The **shelve** module has its own operations for opening and closing files, so you can't use the standard **open** function.

» To save some data to the shelf, you must first use a standard Python assignment operation to set the value of a particular key to the object you want to save.

» As with files, you must close the shelf object once finished with, otherwise your changes may not be stored.

Accessing data inside the shelf is just as easy. Rather than assigning a key in the shelf dictionary to a value, you assign a value to that stored in the dictionary at a particular key: **feeds = shelf["feeds"]**. If you want to modify the data that was stored in the shelf, modify it in the temporary value you assigned it to, then re-assign that temporary value back to the shelf before closing it again.

That's about all we have space for this tutorial, but keep reading, because we'll discuss one final option for persistent data: relational databases (for example, *MySQL*). ■

Lock down with data encryption

Learn the principles behind encryption – we unpack the details of how it prevents snoopers from reading your data

When writing a program, we usually assume that the other applications are friendly. We don't normally try to hide our data from them. In fact, we normally try to write our files so that other programs can read them. We use text encodings, XML and other standards to make sure that our files play nicely with other software. However, there are times when we want to keep our information to ourselves. Perhaps we need to transmit it across an insecure network, or put it on a USB key that could be lost. Whatever the case, we need to make sure that prying eyes can't see what it is, and for this we use encryption.

> **"The first rule of data encryption is this: never to try to create your own"**

Before we get started, we should say that the first rule of data encryption is never try to create your own. Unless, that is, you have a PhD in the appropriate area of mathematics and plenty of experience. There are a number of standards that are generally considered unbreakable, and most languages have a good set of encryption libraries that support these. These libraries will be far more secure than any you can create yourself; just make sure you read the documentation properly to avoid using insecure options. Here we're going to break this rule, but only to show how encryption works.

All encryption starts with data you want to hide. It then applies some method of rendering that data unreadable. Ideally, it should have some method of recovering the data that only the original user can perform. Usually this is done with a password (when the password is a string of binary information rather than an alpha-numeric word, it's referred to as a 'key', but the basic principal is the same). There are two different types of encryption: symmetric key and asymmetric key. The former uses the same key to decrypt information as it used to encrypt it, while the latter uses two different keys.

Symmetric encryption

Here we're going to take a look at a simple (and fairly insecure) method of symmetric encryption (also known as private key encryption) for text that uses the XOR (exclusive OR) function. XOR takes two binary digits as an input (which can each be either 0 or 1). It outputs a 1 if one of the two inputs is a 1, and a 0 if either none or both is (See the XOR Truth Table, right).

We can XOR strings of data by XORing each item in turn. That simple XOR provides all we need for our encryption method which we'll get onto in a minute. First we'll look at the information we're going to encrypt.

Text, like all computer data, is stored in binary as 1s and 0s. ASCII text encoding stores each character as a string of eight 'bits' of binary information. For example, 'B' is 01000010, 'e' is 01100101 and 'n' is 01101110. ASCII text, then is just a chain of these characters, each one eight bits long. 'Ben', therefore is 010000100110010101101110.

Now, back to our encryption method. We're going to use a password that's just a single character (we told you it wasn't going to be secure!). It can be any single character. And our encryption method is to XOR each letter of our text with our key (See XORing characters box, above right).

That ciphertext is our unreadable text. Without knowing the key, there's no way a program can read it... or is there?

We know the key (it's 'A'), but how can we use this to get the original text back? Actually, it's really simple. Back in

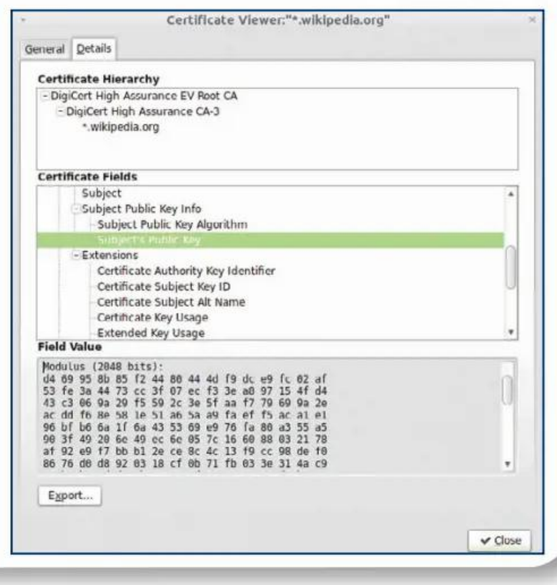
Asymmetric encryption

Symmetric encryption is great for securing files, but it has some problems when securing communication. For starters, you need a secure way of sharing the keys with everyone. If you wanted an encrypted communication with, for example, Google, you'd somehow need to obtain a key, and this key would need to be different for every person Google communicated with otherwise they'd be able to eavesdrop.

To get around this, we have asymmetric encryption (which is also known as public key encryption). In this method there are two keys, one public and the other private. Anything that is encrypted with the public key can only be decrypted with the private key and visa versa.

Now, for example, if you need an encrypted communication with Google, you only need to know Google's public key. Using this public key encryption, you can exchange a single-use key for a session of symmetric encryption.

For further info and an example of an asymmetric implementation, see Neil Bothwick's tutorial at **www.techradar.com/news/internet/data-privacy-how-safe-is-your-data-in-the-cloud--1170332/1**



XORing characters

	b								e								n							
Text	0	1	0	0	0	0	1	0	0	1	1	0	0	1	0	1	0	1	1	0	1	1	1	0
Key A	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	1
	XOR								XOR								XOR							
Cypher text	0	0	0	0	0	0	1	1	0	0	1	0	0	1	0	0	0	0	1	0	1	1	1	1

› We can encrypt a stream by XORing each character in turn with our key

school you may have learned that (a + b) + c = a + (b + c). That is, with addition at least, it doesn't matter which order you do the addition; you always get the same result. Well, it turns out that the same thing is true of XOR. But before we can use that, there are two more things we need to know: anything that is XOR'd with itself is 0, and anything XOR'd with 0 is unchanged.

So, key XOR key is 0 and text XOR 0 is text. That means that text XOR (key XOR key) is text, and we've just learned that this is the same as (text XOR key) XOR key. Since our cyphertext is just text XOR key, we now know that cyphertext XOR key is our original text. Basically, that's just a really long way of saying that we can decrypt something exactly the same way we encrypted it.

Statistical attacks

We now have our method of encryption and decryption, but it's not very secure. First of all, there are only 256 possible keys (if we include all eight bit strings, and not just the ones that have ASCII characters), so it's perfectly feasible for an attacker to check every one in turn. However, it turns out they don't have to.

In English text, some characters are very common, and others are quite rare. For example, the space character usually makes up 15–20% of all the characters in a piece of text and the lower-case E about 10% while lower-case Z can be as little as 0.02% and capital Z almost never.

Since a given character in our text will always evaluate to the same character in our cyphertext, these proportions will come across. For example, given the cyphertext:

```
00001001 00011110 00000011 01110001 00110010
00111000 00100001 00111001 00110100 00100011
00100010 01111101 01110001 01100001 00100010
01110001 00101000 00111110 00100100 01110001
00100010 00110100 00110100 01111101 01110001
00110000 00100011 00110100 01110001 00111110
00111110 00100101 01110001 00100010 00110100
00110010 00100100 00100011 00110100
```

We can see that the most common 8-bit string is: 01110001. We can take a guess that this is the encrypted form of the space character.

We know that text XOR text is 0 and key XOR 0 is key. Therefore we know that (text XOR key) XOR text is key. So, if we XOR the most common character with space (00100000), we get the key. In this case, it's 01010001 which corresponds to the ASCII character 'Q'. Using this, we can decrypt the whole text to:

```
XOR ciphers, as you see, are not secure
```

This is known as a statistical attack and is one of the most common ways of attacking encryption. The simplest way to prevent your data falling victim to it is to use an encryption method that's well known and been tested by the best minds in the business. AES, Two-fish, Three-fish and Serpent are all good choices. As we mentioned at the start, you should find a library that implements one or more of them in your language of choice.

You may have noticed a slight flaw in this plan. How do you find out a server's public key in the first place? After all, if an attacker could trick you into using their public key, they could read all the supposedly secure communications.

We get around this by using certificates. When you install a web browser, it will come with public keys for a number of trusted certificate authorities. A website can then get one of these authorities to sign their public key. When you go to an encrypted web page, the server sends you a certificate (which contains both their public key, and the signature from the authority) and the encrypted method. Since you trust the authority, you can now trust this certificate, and read the page safe in the knowledge that it's been securely transmitted and not tampered with.

This method isn't perfect. First, it requires you to trust a range of authorities (you can see how many authorities you trust in your web browser. For example, in Firefox, go to Edit > Preferences > Advanced > View Certificates > Authorities). It's quite a few, and most are probably companies you've never heard of, much less trust. If a hacker or disgruntled employee gets into the computer systems at any one of them, they could intercept almost any encrypted web traffic they wanted. But that's not the only way these certificates can be subverted. If an attacker can find a way of forging the digital signature, they can trick a computer into thinking a communication comes from a particular source when it doesn't. In fact, it was this method that allowed the Flame malware to get into Iranian computers. Microsoft had used the insecure MD5 (rather than the more secure SHA) hash, and attackers were able to use this to make the computer think something was signed when it wasn't. Be careful! ■

XOR truth table

Input 1	Input 2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

CODING ACADEMY

CODING ACADEMY

Projects

Take your new-found Python skills and put them to good use

32 **Python 3: How to get started**
We're moving to Python 3! About time

36 **Python 3: Using functions**
The essentials of modules, classes and more

40 **FTP: Build a client & server**
Create your own network code for programs

44 **Python 3: Making scripts**
Automate your system with scripts

48 **Python: times, dates & numbers**
Essential processing of complex dates

52 **NumPy & SciPy: For science!**
Become a data scientist

Python 3: How to get started

Join us as we investigate what is probably one of the least loved and disregarded sequels in the whole history of programming languages

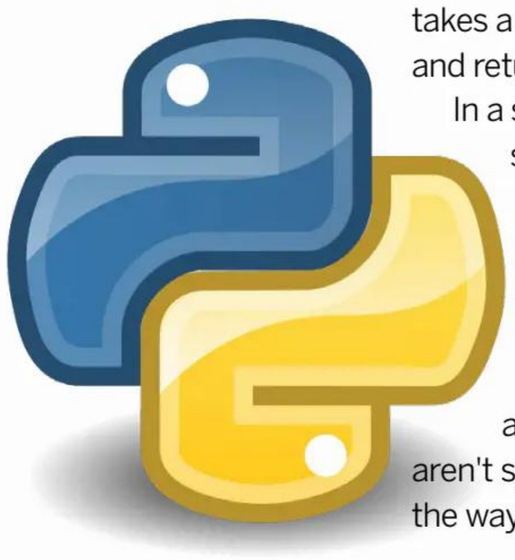
Way back in December 2008, Python 3.0 (also known as Py3k or Python 3000) was released. Yet here we are, ten years later, and most people are still not using it. For the most part, this isn't because Python programmers and distribution maintainers are a bunch of laggards, and the situation is very different from, for example, people's failure/refusal to upgrade (destroy?) Windows XP machines. For one thing, Python 2.7, while certainly the end of the 2.x line, is still regularly maintained, and probably will continue to be until 2020. Furthermore, because many of the major Python projects (also many, many minor ones) haven't been given the 3 treatment, anyone relying on them is forced to stick with 2.7.

Early on, a couple of big projects – NumPy and Django – did make the shift, and the hope was that other projects would follow suit, leading to an avalanche effect. Unfortunately, this didn't happen and most Python code you find out there will fail under Python 3. With a few exceptions, Python 2.7 is forwards-compatible with 3.x, so in many cases it's possible to come up with code that will work in both, but still programmers stick to the old ways. Indeed, even in the excellent monthly magazine *Linux Format,* certain authors, whether by habit, ignorance or affection for the past, continue to provide code that is entirely incompatible with Python 3. We won't do that in this article. We promise.

So let's start with what might have been your first ever Python program:

```
print 'Hello world'
```

Guess what – it doesn't work in Python 3 (didn't you just promise...?). The reason it doesn't work is that **print** in Python 2 was a statement, while in Python 3 **print** is a function, and functions are, without exception, called with brackets. Remember that functions don't need to return anything (those that don't are called void functions), so **print** is now a void function which, in its simplest form, takes a string as input, displays that string as text to *stdout*, and returns nothing.



In a sense, you can pretend **print** is a function in Python 2, since you can call it with brackets, but a decision was made to offer its own special syntax and a bracketless shorthand. This is rather like the honour one receives in mathematics when something named after its creator is no longer capitalised – for example, abelian groups. But these kind of exceptions are not a part of the Python canon ("Special cases aren't special enough to break the rules"), so it's brackets all the way. On a deeper level, having a function-proper print does allow more flexibility for programmers – as a built-in function, it can be replaced, which might be useful if you're into defying convention or making some kind of Unicode-detecting/defying wrapper function. Your first Python program should have been:

```
print ('Hello world')
```

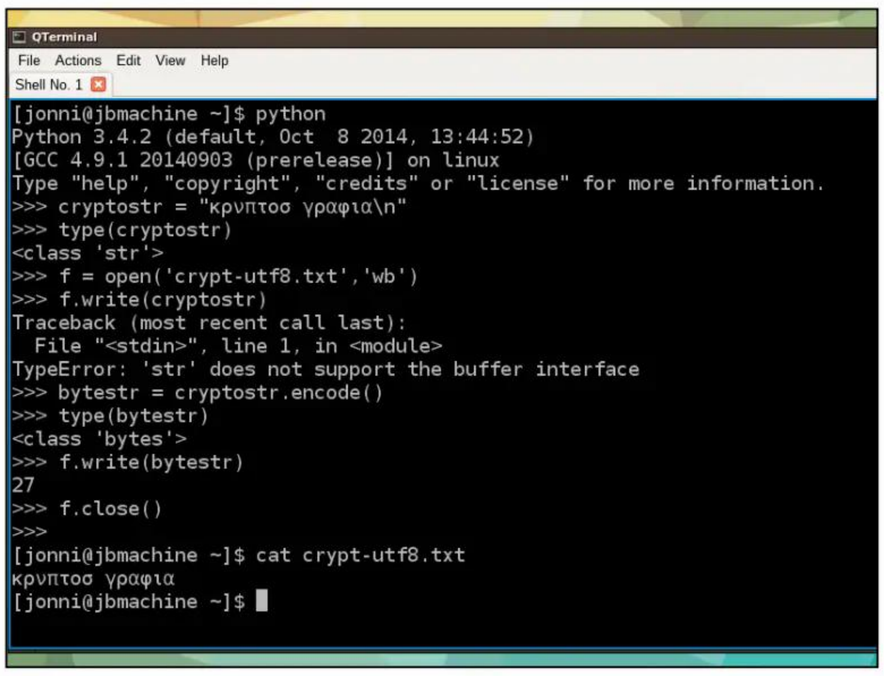
which is perfectly compatible with Python 2 and 3. If you were a fan of using a comma at the end of your print statements (to suppress the newline character), then sad news: this no longer works. Instead, we use the **end** parameter, which by default is a new line. For example:

```
print ('All on', end=" ")
print ('one line')
```

Print in Python 3

A significant proportion of Python programs could be made compatible with 3 just by changing the **print** syntax, but there are many other, far less trivial, things that could go wrong. To understand them, we must first be au fait with what really changed in Python 3.

Most of the world doesn't speak English. In fact, most of the world doesn't even use a Latin character set; even those regions that do tend to use different sets of accents to decorate the characters. As a result, besides the ASCII standard, numerous diverse and incompatible character encodings have emerged. Each grapheme (an abstraction of a character) is assigned a codepoint, and each codepoint is assigned a byte encoding, sometimes identically. In the past, if you wanted to share a document with foreign characters in



› The Greek kryptos graphia, which translates as 'hidden writing', followed by a new line using the correct script

The Unicode revolution

Traditionally, text was encoded in ASCII, in which each character is encoded as a 7-bit codepoint, which gives you 128 characters to play with. Some of these characters are invisible teletype codes (ASCII originated in the 1960s), and once we've counted the familiar alphanumeric characters, there isn't really much room left. Because we like things to be bytes, several 256-character extensions of the ASCII encoding emerged. The most notorious of these is ISO-8859-1, sometimes called Latin-1. This widely-used character set (and the related Windows-1252) contains almost all the accents required for the Latin-scripted languages, as well as the characters used in the romanisation of other languages. As a result, it's fairly common in the western hemisphere, but doesn't really solve the problem elsewhere.

The correct solution would be a standard encoding (or maybe a couple of them) that accounts for as many as possible of the set of characters anyone on earth might conceivably wish to type. Obviously, this will require many more than 256 characters, so we'll have to do away with one character encoding to one byte (hence the divergence of codepoints and byte encodings), but it's for a greater good. Fortunately, all the wrangling, tabulating and other rigmarole has been done, and we have an answer: Unicode. This accounts for over 100,000 characters, bidirectional display order, ligature forms and more. Currently there are two encodings in use: UTF-8, which uses one byte for common characters (making it entirely backwards compatible with ASCII), and up to four bytes for the more cosmopolitan ones; and UTF-16, which uses two bytes for some characters and four bytes for others. Unicode has been widely adopted, both as a storage encoding standard and for internally processing tests. The main raison d'être of Python 3 is that its predecessor did not do the latter.

it, then plain ASCII wouldn't help. You could use one of the alternative encodings, if you knew the people you were sharing it with could do the same, but in general you needed to turn to a word processor with a particular font, which just moves the problem elsewhere. Thankfully, we now have a widely adopted standard: Unicode (*see The Unicode Revolution box, above*) that covers all the bases, and is backwards compatible with ASCII and (as far as codepoints are concerned) its Latin-1 extension. We can even have Unicode in our domain names, although internally these are all still encoded as ASCII, via a system called Punycode.

Python 2 is far from devoid of Unicode support, but its handling of it is done fairly superficially (Unicode strings are sneakily re-encoded behind the scenes) and some third-party modules still won't play nicely with it.

Strings in Python 2 can be of type **str** (which handles ASCII fine, but will behave unpredictably for codepoints above 127) or they can be of type **unicode**. Strings of type **str** are stored as bytes and, when printed to a terminal, are converted to whichever encoding your system's locale specified (through the **LANG** and **LC_*** environment variables in Linux). For any modern distro, this is probably UTF-8, but it's definitely not something you should take for granted.

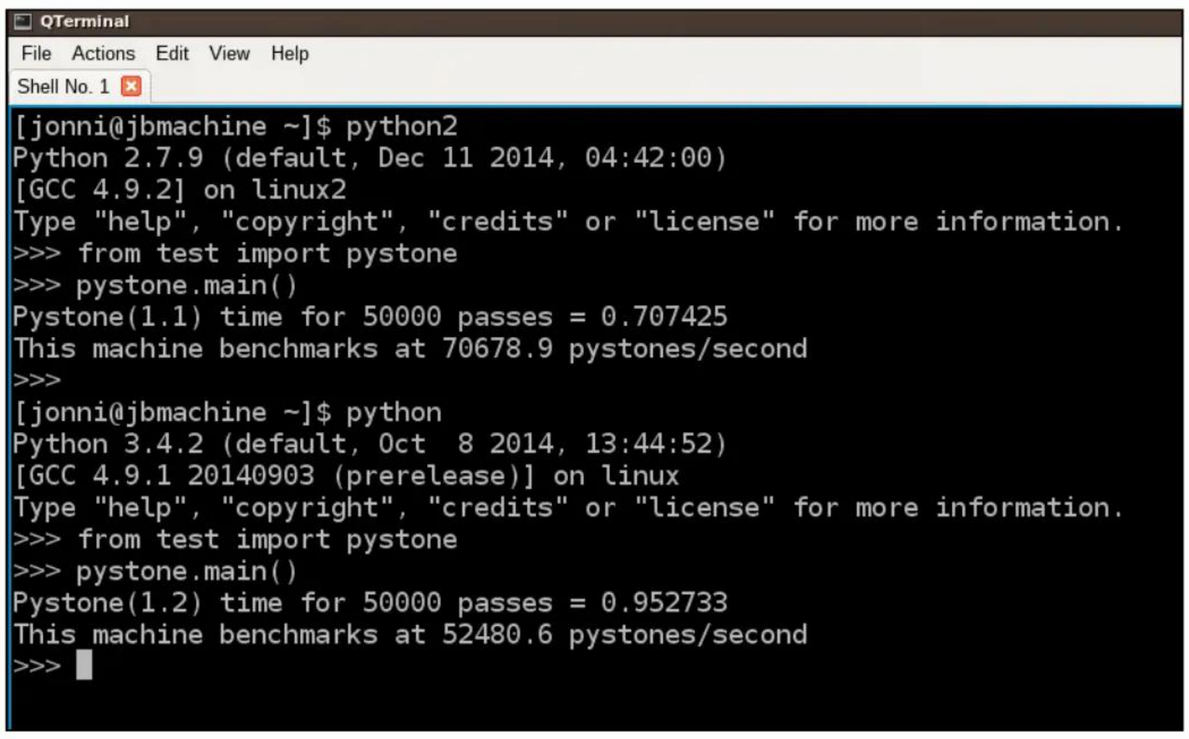
The **unicode** type should be used for textual intercourse – finding the length of, slicing or reversing a string. For example, the Unicode codepoint for the lowercase Greek letter pi is **03c0** in hex notation. So we can define a unicode string from the Python console like so, provided our terminal can handle Unicode output and is using a suitable font:

```
>>> pi = u'\u03c0'
>>> print(pi)
π
>>> type(pi)
<type 'unicode'>
>>> len(pi)
1
```

However, if we were to try this on a terminal without Unicode support, things will go wrong. You can simulate such a scenario by starting Python with:

```
$ LC_ALL=C python
```

Now when you try to print the lowercase character pi, you will run into a **UnicodeEncodeError**. Essentially, Python is trying and failing to coerce this to an ASCII character (the only type supported by the primitive C locale). Python 2 also tries to perform this coercion (regardless of current locale settings) when printing to a file or a pipe, so don't use the **unicode** type for these operations, instead use **str**.



› The *PyStone* benchmark will likely be slower in Python 3, but the same won't be true for all code. Don't be a Py3k refusenik without first trying your code

The **str** type in Python 2 is really just a list of bytes corresponding to how the string is encoded on the machine. This is what you should use if you're writing your strings to disk or sending them over a network or to a pipe. Python 2 will try and convert strings of type **unicode** to ASCII (its default encoding) in these situations, which could result in tears. So we can also get a funky pi character by using its UTF-8 byte representation directly. There are rules for converting Unicode codepoints to UTF-8 (or UTF-16) bytes, but it will suffice to simply accept that the pi character encodes to the two bytes **CF 80** in UTF-8. We can escape these with an **\x** notation in order to make Python understand bytes:

```
>>> strpi = '\xCF\x80'
>>> type(strpi)
<type 'str'>
>>> len(strpi)
2
```

So π apparently now has two letters. The point is: if your Python 2 code is doing stuff with Unicode characters, you'll need to have all kinds of wrappers and checks in place to take account of the localisation of whatever machine may run it. You'll also have to handle your own conversions between »

Quick tip

Arch Linux is one of few distributions to use Python 3 by default, but it can live happily in tandem with its predecessor (available in the **python2** package).

» **str** and **unicode** types, and use the codecs module to change encodings as required. Also, if you have Unicode strings in your code, you'll need to add the appropriate declaration at the top of your code:

```
# -*- coding: utf-8 -*-
```

The main driving force for a new Python version was the need to rewrite from the ground up how the language dealt with strings and their representations to simplify this process. Some argue that it fails miserably (see Armin Ronacher's rant at **http://bit.ly/UnicodeInPython3**), but it really depends on your purposes. Python 3 does away with the old unicode type entirely, since everything about Python 3 is now Unicode-orientated. The str type now stores Unicode codepoints, the language's default encoding is UTF-8 (so no need for the **-*- coding** decorator above) and the new **bytes** object stores byte arrays, like the old str type. The new str type will need to be converted to bytes if it's to be used for any kind of file I/O, but this is trivial via the **str.encode()** method. If you're reading Unicode text files, you'll need to open them in binary 'rb' mode and convert the bytes to a string using the converse **bytes.decode()** method (*see the picture for details bottom left*).Python 3, however, brings about many other changes besides all this Unicode malarkey, some of these are just the removal of legacy compatibility code (Python 3, unlike 2.7, doesn't need to be compatible with 2.0), some of them provide new features, and some of them force programmers to do things differently.

But wait, there's more: for example, functions can now be passed keyword-only arguments, even if they use the ***args** syntax to take variable length argument lists, and catching exceptions via a variable now requires the **as** keyword. Also, you can no longer use the ugly **<>** comparison operator to test for inequality – instead use the much more stylish **!=** which is also available in Python 2.

Automating conversion with 2to3

For reasonably small projects, and those that aren't going to encounter any foreign characters, it's likely that your Python 2 code can be automatically made Python 3 compatible using the *2to3* tool. This helpful chap runs a predefined set of fixers

```
Shell No. 1
[jonni@jbmachine ~]$ python2 aleph.py
א ב ג ד ה ו ז ח ט י ך כ ל ם מ | נ ס ע ף פ ץ צ
[jonni@jbmachine ~]$ 2to3 aleph.py
RefactoringTool: Skipping implicit fixer: buffer
RefactoringTool: Skipping implicit fixer: idioms
RefactoringTool: Skipping implicit fixer: set_literal
RefactoringTool: Skipping implicit fixer: ws_comma
RefactoringTool: Refactored aleph.py
--- aleph.py    (original)
+++ aleph.py    (refactored)
@@ -1,3 +1,3 @@
 for j in range(0x5d0, 0x5eb):
-    char = unichr(j)
-    print '%c ' % char,
+    char = chr(j)
+    print('%c ' % char, end=' ')
RefactoringTool: Files that need to be modified:
RefactoringTool: aleph.py
[jonni@jbmachine ~]$
```

› Aleph, beth, gimel... The *2to3* program shows us how to refactor our Python 2 code for Python 3. Shalom aleikhem

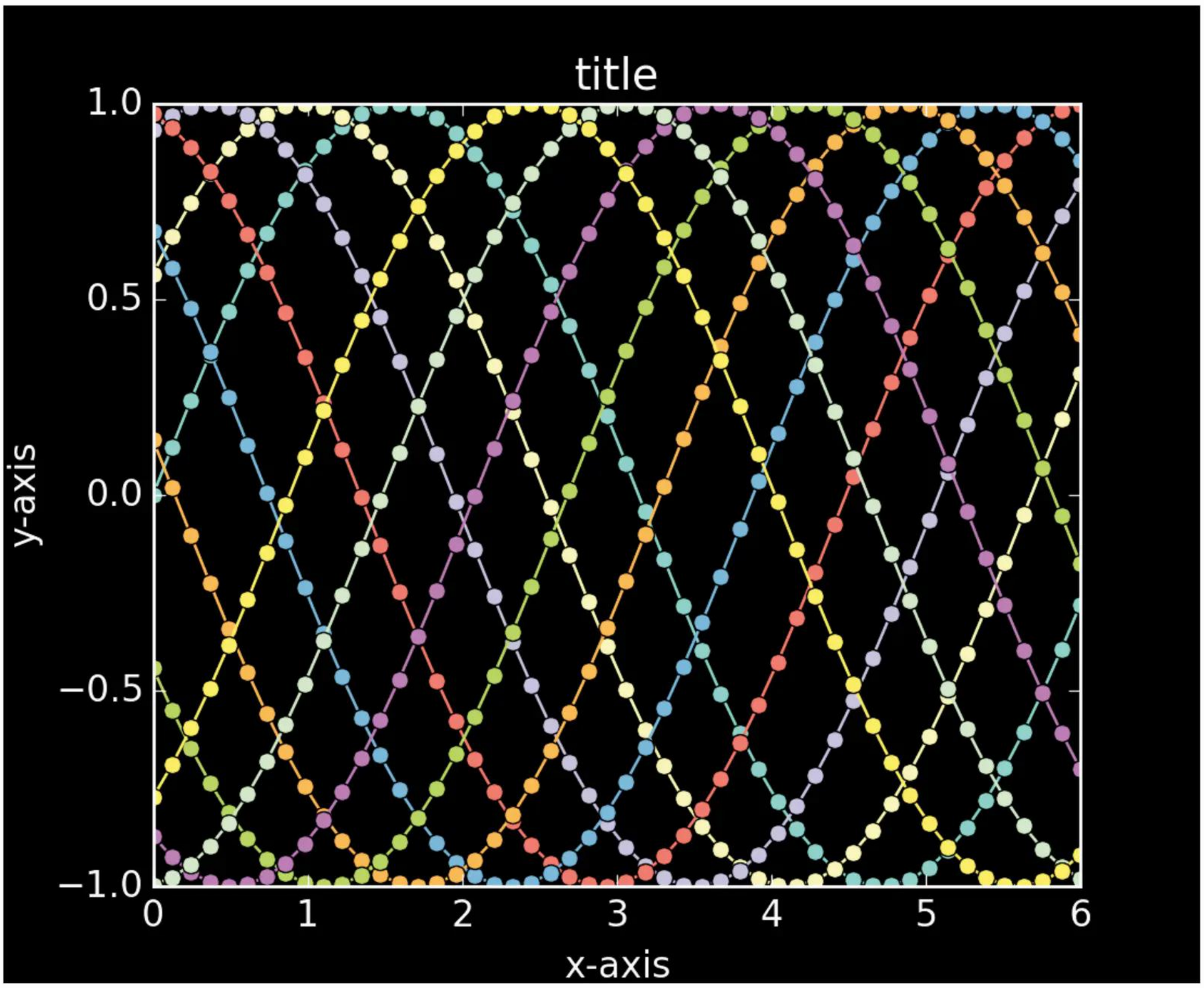
The division bell

One change in Python 3 that has the potential to flummox is that the behaviour of the division operator **/** is different. For example, here's an excerpt from a Python 2 session

```
>>> 3/2
1
>>> 3./2
1.5
>>> 3.//2
1.0
```

This shows the first command operating on two ints, and returning an int, thus in this case the operator stands for integer division. In the second example, the numerator is a float and the operator acts as floating point division, returning what you'd intuitively expect half of three to be. The third line uses the explicit floor division operator **//** which will return the rounded-down result as a float or an int, depending on the arguments. The latter operator was actually backported to 2.7, so its behaviour is the same in Python 3, but the behaviour of the classic division operator has changed: if both numerator and denominator are ints, then an int is returned if one divides the other, otherwise a float is returned with the correct decimal remainder. If at least one of the arguments is a float (or complex), then the behaviour is unchanged.

This means the **/** operator is now as close to mathematical division proper as we can hope, and issues involving unexpected ints will no longer cause harm.



› **The popular matplotlib module has been Python 3 compatible since v1.2, for all your graphing and plotting requirements**

on your Python 2 code, with the goal of emitting bona fide Python 3 code. Using it can be as simple as:

```
$ 2to3 mypy2code.py
```

This will output a diff of any changes against the original file. You can also use the **-w** option to overwrite your file – don't worry, it will be backed up first. Some fixers will only run if you specify them explicitly, using the **-f** option. An example is **buffer** which will replace all **buffer** types with **memoryviews**. These two aren't entirely compatible, so some tweaking might be necessary in order to successfully migrate. Using *2to3* can save you a whole heap of effort, because searching and replacing **print** statements manually, for instance, is a drudgerous task. The pictured example (*over on the left, on p38*) shows the changes to a simple three-line program – the **unichr()** function is now **chr()**, because Unicode is now implicit, and the print line is reworked, even if it uses **%** to format placeholders.

Parlez-vous Python Trois?

Another option is to create 'bilingual' code that is compatible with both Python 2 and Python 3. While you should really only target one specific Python version, this middle ground is very useful when you're porting and testing your code. You might need to rework a few things in Python 2 while still enjoying the new features and learning the new syntax. Many Python 3 features and syntaxes have already been backported to 2.7, and many more can be enabled using the __future__ module. For example, you can get the new-fangled **print** syntax by using the following:

```
>>> from __future__ import print_function
```

This is used with the **from** in this way, and __future__ doesn't behave like a standard module – instead it acts as a compiler directive, which in this case provides the modern print syntax. We can likewise import division to get the new style division, or unicode_literals to make strings Unicode by default. There is another module, confusingly called future, which isn't part of the standard library, but can help ease transition issues.

When Python 3.0 was released, it was generally regarded as being about 10% slower than its predecessor. This was not surprising, because speed was not really a priority for the new version and many special-case optimisations were removed in order to clean up the code base. Now that we're up to version 3.5.1 (which was released in December 2015), it would be nice if performance had since been improved. Unfortunately, this hasn't been the case, which you can verify for yourself using the *PyStone* benchmark. We tested it (see the previous spread) on an aging machine in the office, which has now come back from the dead twice and hence possesses supernatural powers far beyond what its dusty 2.3GHz processor would suggest.

But don't be misled; *PyStone* tests various Python internals, of which your code may or may not make extensive use. It's important to test your code in both versions to get a more accurate picture. You can always use Cython (maintained at **http://cython.org**) to speed up code that is amenable to C translation (loops, math, array access), or the bottleneck module.

Guido van Rossum, the author of Python, says that Python 2.7 will be supported until 2020, but that's no excuse to postpone learning the new version now. Python 3 adoption is increasing, so you won't be alone. Half the work is retraining your fingers to add parentheses. ■

Python 3: Using functions

Everything you need to know to start using functions and developing classes, modules, packages and objects in Python 3

Quick tip

You can also create a new function using the lambda keyword, which also creates an object but returns it as a result, which is sometimes called an anonymous function. This is an advanced feature of Python 3.

This tutorial will teach you about Python 3 functions, classes, objects, modules and packages. Its main purpose is to clarify all these terms by using small examples that illustrate how to create and use them all, and a large part of the tutorial is about namespaces, which are used everywhere in Python 3.

Functions do two very important things: first, they enable us to reuse code and therefore reduce code repetition, which, in turn, allows us to write less code. Second, functions offer a way to solve a big program by solving smaller components of that problem.

The following Python code declares a function that is named aFunction:

```
def aFunction():
    print("Hello!")
```

As you can see, a function definition begins with the `def` keyword followed by the function name. It's not mandatory for a Python function to return something or have a number of arguments. You can now use aFunction as follows:

```
aFunction()
```

Despite the fact it prints the same message all the time, the previous function takes no input and returns nothing. However, most of the time a function takes one or more parameters as input and returns one or more values. The next function, we've called timesString, takes two parameters: an integer and a string, as input and returns a string:

```
def timesString(times, text):
    string = ""
    for x in range (0, times):
        string += text
    return string
```

You can use `timesString` as follows:

```
myString =  timesString(3, "Hi! ")
print(timesString(2, "Hi! "))
```

However calling `timesString()` as `timesString("Hi", "Hi! ")` will cause an error message because the first parameter cannot be used as an integer inside the function.

As you might be aware, when a function returns something, you should take care of it and it's quite common to save the return value of a function using a variable. The proper way for a function to return a value is with the `return` keyword. It's very important to remember that Python has no rules about the types of the arguments and the return value of a function, i.e. they can be whatever you want. However, it's easy to forget that when a function works with integer values, it will now work when you pass a string instead. You should be very careful because improper argument types can be the source of nasty bugs. Although it's not mandatory to declare the types of the function arguments, you are free to do so. (See the screenshot, left for more information.)

Now a quick mention of function declarations in Python 3. What the def keyword does is create a new object and assign a name to it. So def creates a new function object and its name is used for referencing it. Now that you know the basics of functions, we can start talking about classes.

```
2. mtsouk@mail: ~/docs/article/working/TGG14p1/tut2/code (ssh)
code$ python3 myFunctions.py
Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
Hi! Hi! Hi!
Hi! Hi!
Hi! Hi! Hi! Hi! Hi!
Traceback (most recent call last):
  File "myFunctions.py", line 29, in <module>
    print(timesString("Hi", "Hi! "))
  File "myFunctions.py", line 8, in timesString
    for x in range (0, times):
TypeError: 'str' object cannot be interpreted as an integer
code$ cat myFunctions.py

def aFunction():
    print("Hello!")


def timesString(times, text):
    string = ""
    for x in range (0, times):
        string += text
    return string

def withTypes(times: int, text: str) -> str:
    string = ""
    for x in range (0, times):
        string += text
    return string

aFunction()

for x in range(0, 5):
    aFunction()
```

› This is the output of the myFunctions.py Python 3 script. The last statement of the script causes the error message you see on the screen because timesString() cannot use a string variable as an integer

A Python class

A Python class is a way of grouping functions and variables using an object oriented method. A Python class provides all the expected features of object oriented programming, such

as inheritance and method override etc. The following Python code defines a new Python class that we are calling Linux:

```
class Linux:
    def __init__(self, version):
        self.version = version
```

As you can see, a class definition begins with the `class` keyword followed by the name of the class. When the definition of a class contains a method with the name of `__init__()`, the `__init__()` method is automatically executed for every instance of the class. The `__init__()` method can also have arguments like a normal function. You can see a full example of the Linux class in **myClass.py**.

It is now time for a warning. The following class definition contains a class variable that's shared by all instances of the class. This is quite a powerful development as it means that it can also be changed by all instances of the class:

```
class Linux:
    data = []

    def addData(self, aNumber):
        self.data.append(aNumber)
```

So, imagine the following Python code, saved as **sharedVariable.py**:

```
if __name__ == '__main__':
    old = Linux()
    new = Linux()
    old.addData(1.0)
    new.addData(1.2)

    print('Old: ', old.data)
    print('New: ', new.data)
```

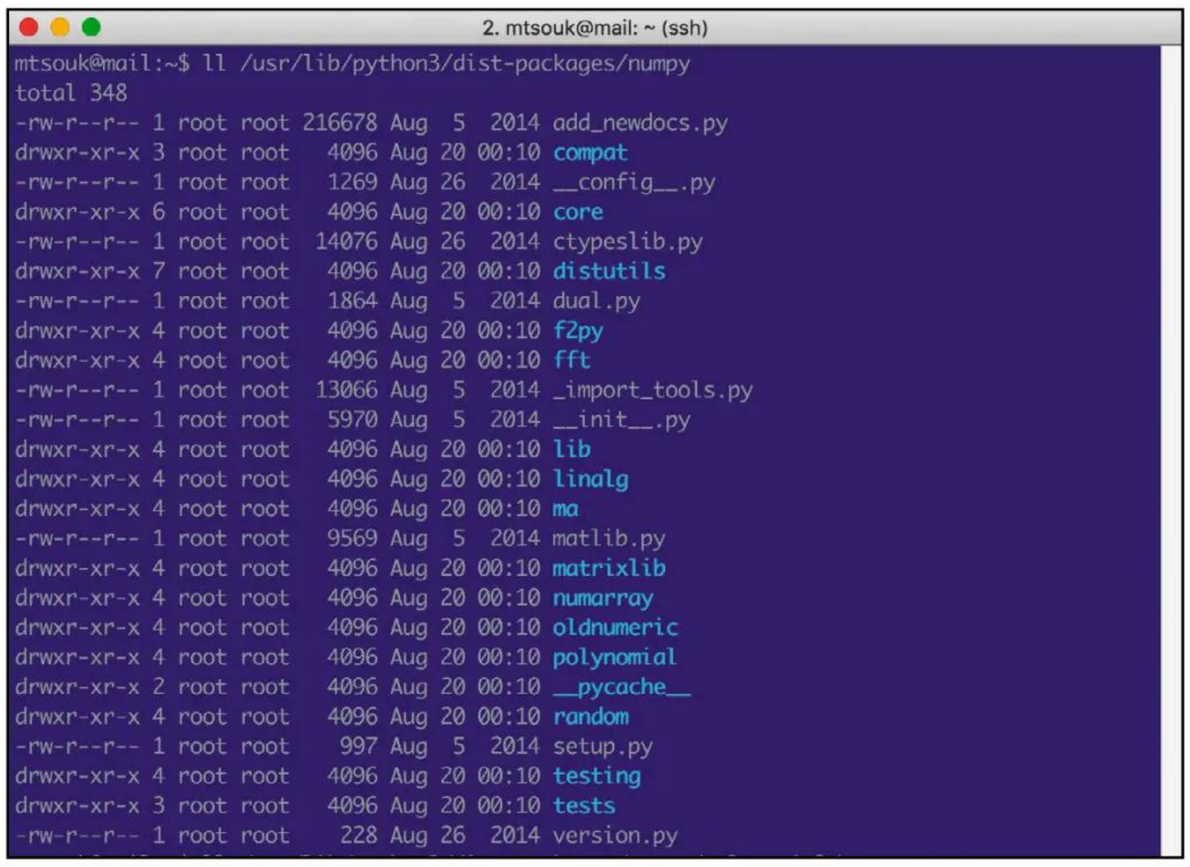
Its output will be the following:

```
$ python3 sharedVariable.py
Old:  [1.0, 1.2]
New:  [1.0, 1.2]
```

The correct design for such a variable would be to declare it inside the `__init__()` method, so that it would not be shared by all instances of the class. Now, another big secret: everything in Python 3 is a class!

A Python object

Put simply, an object is a variable of a given class. While a class is just a definition, an object is something more specific that exists and can be used for storing data or calling its methods. So, without an object existing, a class is essentially just a theory. As you saw in the previous section, you can declare an object as easily as writing:



› This is the directory structure of the famous NumPy package, which contains about 1,100 files and directories!

```
old = Linux()
new = Linux(2.3)
```

Lets move on to modules. A module in Python is a single file imported under one import statement. The presented Python module, saved as **myModule.py**, will show you how to group different elements and put them into a module that's named LXF.

```
def hello():
    print("Hello World!")

myVersion = 123
```

A Python module

The presented module contains one function and one variable. As the file that holds the code of the module is named **myModule.py**, the name of the module will be myModule. You can then use it in your programs as follows:

```
import myModule

myModule.hello()
print(myModule.myVersion)
```

If you execute **useMyModule.py** then this will generate the following output:

```
$ python3 useMyModule.py
```

»

Quick tip

Object oriented programming is a programming paradigm where data and related operations are organised into objects. Each object is an instance of a class, which is either offered by the programming language or defined by the developer. Classes can be combined and create more complex classes.

Programming and debugging

While programming is not as difficult as you may think, it's not a trivial task, especially when you have to program critical applications. You should learn the basic things first and then try to build something real on your own. The one thing to remember is that you shouldn't be afraid of learning slowly; as long as you keep making new mistakes and developing new programming skills, you will finally learn how to code.

As programs tend to contain errors, debugging is also a job for the programmer. First of all, you have to know that debugging code that you or someone else wrote is more perplexing and difficult than the actual writing of the code but it can also be very exciting. The hardest part of the debugging process isn't fixing the bugs; it's reproducing them on your machine. The fixing part is usually the easiest part of the process.

When debugging code it's helpful to have the following things in mind: never debug code when you are sleep deprived or exhausted and that sometimes it is better to leave debugging for tomorrow or take a walk to get some fresh air first. Additionally, while you should try to isolate a bug as early as possible, sometimes it's better to write a program, or a part of it, from scratch instead of trying to find out why it does not work—especially when you have to deal with a relatively small amount of code.

Last, you should always bear in mind that the code that is the hardest to debug is the code that you think cannot possibly be wrong. Obviously, the best debugging technique is trying to write clean code from the beginning because clean code contains less bugs and is easier to debug.

Python is a great programming language to learn programming and create useful applications but only practice can make you a real programmer!

```
Hello World!
123
```

After using myModule, you might find a new directory named **__pycache__**. The **__pycache__** directory contains Python 3 bytecode compiled and ready to be executed and is automatically created by Python 3:

```
$ ls -l __pycache__/
total 4
-rw-r--r-- 1 mtsouk mtsouk 293 Aug 23 10:17 myModule.
cpython-34.pyc
$ file __pycache__/myModule.cpython-34.pyc
__pycache__/myModule.cpython-34.pyc: python 3.4 byte-
compiled
```

If your Python program cannot find the requested module, it will generate the following kind of error message:

```
$ python3 useMyModule.py
Traceback (most recent call last):
  File "useMyModule.py", line 1, in <module>
    import myModule
ImportError: No module named 'myModule'
```

Python doesn't look everywhere on your Linux system for modules. The following Python code presents a technique that allows you to dynamically add a directory to the list of directories that Python uses to look for modules and packages:

```
import sys
if '/a/path' not in sys.path:
    sys.path.insert(0, '/a/path')
```

Additionally, you can find the full path of an imported module or package as follows:

```
>>> import numpy
>>> print(numpy.__file__)
/usr/lib/python3/dist-packages/numpy/__init__.py
```

This happens because each module has a `__file__` variable that is used to hold the relative path from the current working directory.

```
from .testing import Tester
test = Tester().test
bench = Tester().bench

from . import core
from .core import *
from . import compat
from . import lib
from .lib import *
from . import linalg
from . import fft
from . import polynomial
from . import random
from . import ctypeslib
from . import ma
from . import matrixlib as _mat
from .matrixlib import *
from .compat import long

# Make these accessible from numpy name-space
#  but not imported in from numpy import *
if sys.version_info[0] >= 3:
    from builtins import bool, int, float, complex, object, str
    unicode = str
else:
    from __builtin__ import bool, int, float, complex, object, unicode, str

from .core import round, abs, max, min

__all__.extend(['__version__', 'pkgload', 'PackageLoader',
            'show_config'])
__all__.extend(core.__all__)
__all__.extend(_mat.__all__)
__all__.extend(lib.__all__)
__all__.extend(['linalg', 'fft', 'random', 'ctypeslib', 'ma'])

# Filter annoying Cython warnings that serve no good purpose.
import warnings
warnings.filterwarnings("ignore", message="numpy.dtype size changed")
warnings.filterwarnings("ignore", message="numpy.ufunc size changed")
mtsouk@mail:~$ wc /usr/lib/python3/dist-packages/numpy/__init__.py
 197  762 5970 /usr/lib/python3/dist-packages/numpy/__init__.py
mtsouk@mail:~$
```

This screenshot shows off some of the Python code you'll find in the __init__.py file of the NumPy Python 3 package

About Python packages

A package might look similar to a module but it's a different beast: a package is a collection of modules stored in different directories and as a result a package has a hierarchy. Additionally, when you import a package only what is declared inside the **__init__.py** file of that package is directly visible to the rest of the world. As the documentation of Python 3 states "Technically, a package is a Python module with a __path__ attribute". The __path__ attribute of a package is used during the import process of its subpackages. The image (top, previous page) shows the root directory of the **NumPy** package as well as the contents of its **polynomial** and **random** subdirectories. The next output shows that **NumPy** contains a lot of files and about 91 directories:

```
$ ls -lR /usr/lib/python3/dist-packages/numpy | wc
   1073    7482   55047
$ find /usr/lib/python3/dist-packages/numpy  -type d | wc
     91      91    5267
```

As you can realise developing such a powerful and handy package requires a lot of work. The screenshot (*see left*) shows a part of the Python code of the **__init__.py** file of the **NumPy** package. However, discussing more detail about Python packages is beyond the scope of this tutorial.

Python 3 namespaces

A namespace is a technique found in many programming languages for generating containers that map names to objects. As you might expect, each namespace is isolated from the rest of the world. Put simply, namespaces allow you to use the same function, variable or class name multiple times without causing any problems.

When Python searches for something, it first checks the local namespace before going to search the global namespace. However, you cannot use the same variable name multiple times in the same namespace—as you can probably image this would either cause a conflict or be the root of potential bugs. Things can be very tricky with namespaces as you can see by looking at the following Python 3 code:

```
aGlobal = 1
```

Object oriented programming

In Python 3 everything is an object so object oriented programming plays a key role. This kind of programming is a very useful paradigm that can make your life easier because it keeps both data and code in the same place, which is the object (of a class). Additionally, the developer is allowed to change the implementation of an existing class method without worrying because the actual implementation doesn't really matter as long as you keep the method name the same.

You are also allowed to reuse an existing class and create a bigger one using Inheritance. This is when you 'inherit' the attributes and the methods of one or more existing classes, called super classes, in order to create a new class—the new class can have additional attributes and methods. The good thing is that you can alter the implementation of some of the methods of the super classes if needed while keeping the implementation of the rest of the methods the same.

```
def aFunction():
    aGlobal = aGlobal + 1
    print(aGlobal)

aFunction()
print(aGlobal)
```

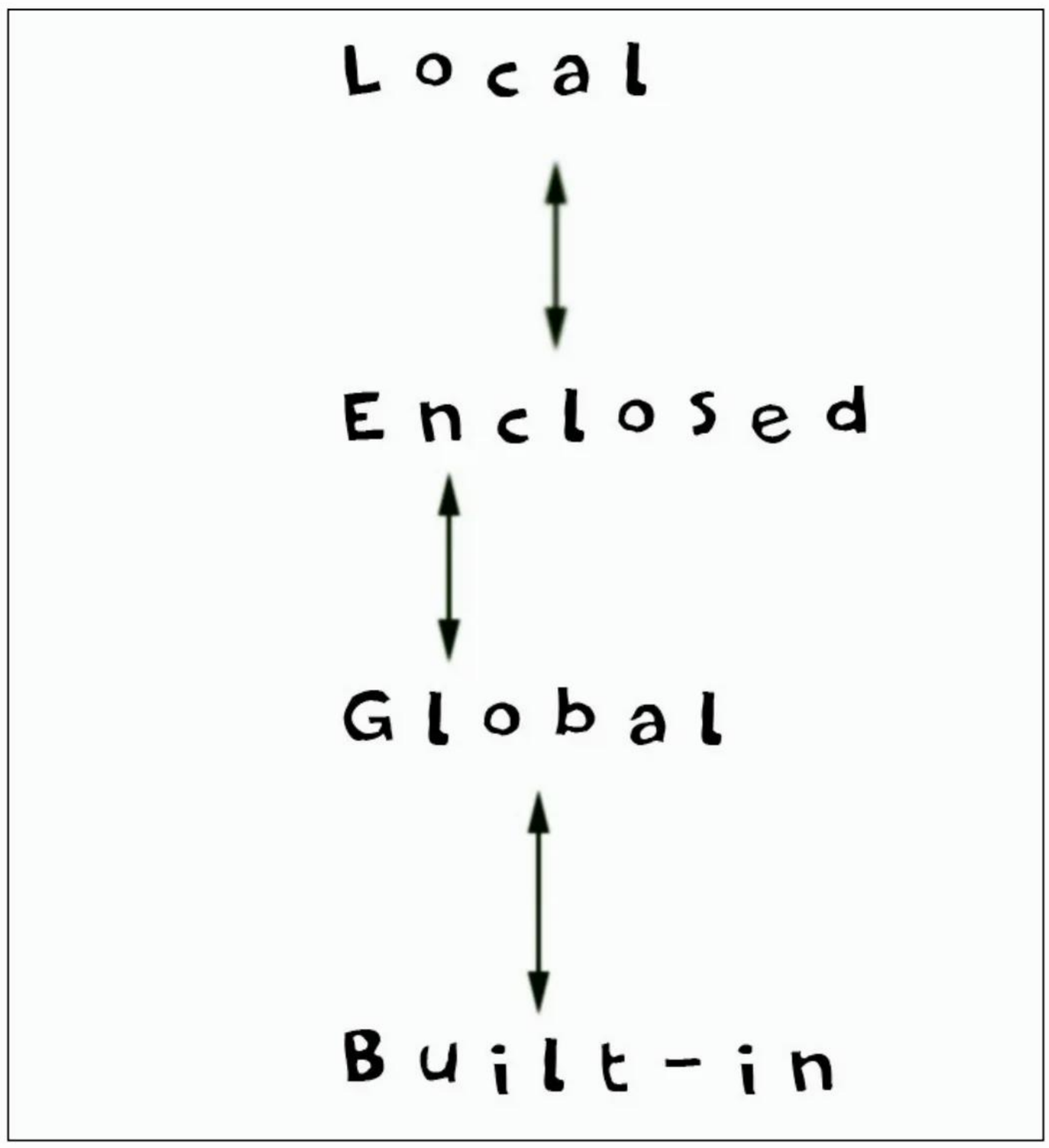
Executing the previous code will generate the following error message, which clearly reveals the root of the problem:

```
UnboundLocalError: local variable 'aGlobal' referenced
before assignment
```

In that case, you want to use the global variable named `aGlobal`. However, the current version of `aFunction()` raises an `UnboundLocalError` error because Python 3 thinks that you are trying to change the value of a local variable named `aGlobal` that's automatically created, which doesn't have an initial value and therefore cannot be read, causing the error message. You can easily solve the previous problem by using the `global` keyword, which tells Python 3 that you are interested in the `aGlobal` variable that exists in the global namespace:

```
def aFunction():
    global aGlobal
    aGlobal = aGlobal + 1
    print(aGlobal)
```

The correct version of the code is saved as **namespaceProblem.py**. As you can see in order to avoid conflicts and errors, you should try to specify the namespace you want to use.



This diagram shows the rule that Python uses in order to find out whether a variable exists in a namespace or not

Globals() and locals()

If you find yourselves in a difficult position where you want to tell whether a variable belongs to the local namespace or not, you should use the `globals()` and `locals()` functions. The following Python code showcases the use of the `globals()` and `locals()` functions for revealing the namespace of a variable:

```
#!/usr/bin/python3

aGlobalariable = 1

def aFunction():
    aLocalVariable = -10
    print('The aLocalVariable inside aFunction: ',
'aLocalVariable' in locals())
    print('The aGlobalariable inside aFunction: ',
'aGlobalariable' in locals())

aFunction()
print('aLocalVariable in main:', 'aLocalVariable' in globals())
print('aGlobalariable in main:', 'aGlobalariable' in globals())
```

Executing the previous code, saved as **globalLocal.py**, generates the following output:

```
$ ./globalLocal.py
The aLocalVariable inside aFunction:  True
The aGlobalariable inside aFunction:  False
aLocalVariable in main: False
aGlobalariable in main: True
```

What the previous output says is that there's a variable named `aLocalVariable` inside the local variables of the current namespace whereas there's not a variable `aGlobalariable` in the local namespace. Similarly, there's a variable named `aGlobalariable` in the global namespace whereas a variable named `aLocalVariable` doesn't exist in the global namespace. Being able to distinguish between local and global variables can make your life easier.

There are times, however, when it's compulsory to use the name of the namespace when you want to access a variable or a constant:

```
import math
import numpy
import scipy

print(math.e, ' :math')
print(numpy.e, ' :numpy')
print(scipy.e, ' :scipy')
```

Trying to use `e` without a namespace will generate an error message:

```
NameError: name 'e' is not defined
```

The rule that Python follows for the resolution of variable names can be summarised in 'Local then Enclosed then Global then Built-in', which should be pretty clear by now. (The rule is presented in a more pleasant way, above.)

Now, one more big secret: when you declare a class, a function, a method function, a package or a module, Python automatically creates a new namespace for you in order to avoid any conflicts. Additionally, when a function or method function finishes, its namespace is destroyed along with its variables.

The whole thing with namespaces might look difficult, tedious and generally not interesting but knowing how namespaces work can save you a lot of time in the long run. Now you will be able to easily understand how a Python 3 program works as well as its various components. ■

Quick tip

There is no better way to organise your Python scripts than by using functions, classes and modules. If you have a small script use functions. If you have a bigger script, you'll most likely use one or more classes. For really big scripts, the best way to organise them is with the help of modules that can be reused by you and others.

Projects

FTP: Build a client & server

Start developing FTP clients and servers in Python 3 with this opening guide to creating your own network code for programs

The subject of this tutorial is network programming in Python 3. The network protocol that will be used is FTP, mainly due to its simplicity; however most of the things that you will learn here can be applied to other protocols as well.

After reading this tutorial you should be more comfortable with network programming in Python and the FTP protocol in particular, you will have written an FTP client with a GUI, as well as a basic FTP server written in Python 3.

The FTP protocol

FTP (File Transfer Protocol) used to be one of the most popular protocols because of its usefulness. Its main disadvantage is that it uses unencrypted traffic, so anyone can steal and use your data if they get the chance. Public FTP servers allow you to log in to them using Anonymous FTP, which is a way for archive sites to allow free access to them. There's a special FTP account with limited access called anonymous, most of the times you cannot upload anything on the anonymous account. You can use any string as its password; however, it is a tradition to use your email address as the password, which mainly happens for letting the administrators of the FTP site get an idea of who is accessing the FTP site. Please have in mind that the password does not appear on screen when logging to an FTP site from the command line; however, FTP clients with a GUI might give you the choice to view the password.

This tutorial will use anonymous FTP for all the presented connections; however, it is relatively easy to change the GUI application and enable it to make regular FTP connections that require a valid username and password combination.

This tutorial will use a Greek anonymous FTP server for testing purposes, please find a public FTP server close to your geographical area and use that.

21.13. ftplib — FTP protocol client

Source code: Lib/ftplib.py

This module defines the class FTP and a few related items. The FTP class implements the client side of the FTP protocol. You can use this to write Python programs that perform a variety of automated FTP jobs, such as mirroring other FTP servers. It is also used by the module urllib.request to handle URLs that use FTP. For more information on FTP (File Transfer Protocol), see Internet RFC 959.

Here's a sample session using the ftplib module:

```
>>> from ftplib import FTP
>>> ftp = FTP('ftp.debian.org')     # connect to host, default port
>>> ftp.login()                     # user anonymous, passwd anonymous@
'230 Login successful.'
>>> ftp.cwd('debian')               # change into "debian" directory
>>> ftp.retrlines('LIST')           # list directory contents
-rw-rw-r--    1 1176     1176         1063 Jun 15 10:18 README
...
drwxr-sr-x    5 1176     1176         4096 Dec 19  2000 pool
drwxr-sr-x    4 1176     1176         4096 Nov 17  2008 project
drwxr-xr-x    3 1176     1176         4096 Oct 10  2012 tools
'226 Directory send OK.'
>>> ftp.retrbinary('RETR README', open('README', 'wb').write)
'226 Transfer complete.'
>>> ftp.quit()
```

The module defines the following items:

class ftplib.**FTP**(*host=''*, *user=''*, *passwd=''*, *acct=''*, *timeout=None*, *source_address=None*)

Return a new instance of the FTP class. When *host* is given, the method call `connect(host)` is made. When *user* is given, additionally the method call `login(user, passwd, acct)` is made (where *passwd* and *acct* default to the empty string when not given). The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if is not specified, the global default timeout setting will be used). *source_address* is a 2-tuple `(host, port)` for the socket to bind to as its source address before connecting.

The FTP class supports the with statement, e.g.:

```
>>> from ftplib import FTP
>>> with FTP("ftp1.at.proftpd.org") as ftp:
...     ftp.login()
...     ftp.dir()
...
'230 Anonymous login ok, restrictions apply.'
dr-xr-xr-x   9 ftp      ftp           154 May  6 10:43 .
dr-xr-xr-x   9 ftp      ftp           154 May  6 10:43 ..
dr-xr-xr-x   5 ftp      ftp          4096 May  6 10:43 CentOS
dr-xr-xr-x   3 ftp      ftp            18 Jul 10  2008 Fedora
>>>
```

› **Figure 1: This is the web page with the documentation of the ftplib Python 3 module that helps you implement FTP clients. This should be the first place to look for help about the ftplib module**

Python FTP

The most useful module for creating FTP clients is called ftplib. If you are a hardcore developer you can also use the socket module (**https://docs.python.org/3/library/socket.html**). If you don't want to write too much Python code, you can take a look at the Twisted system (**https://twistedmatrix.com/trac/**). However, ftplib is pretty good for learning purposes.

A very common way for installing a Python module is by using the *pip* utility; note that the Python 3 version of the *pip* utility is named *pip3*:

```
# pip3 install ftplib
```

You should execute the previous command with root privileges in order to make the Python module available to all the users on your Linux machine. Alternatively, you can ask your system administrator to install it for you.

However, most of the standard Python 3 modules will have a ready to install package for your Linux distribution, which you should choose. For the ftplib module, you should install the ftplib3 Debian package:

```
# apt-get install ftplib3
```

A sample app

It's time to knock together a simple Python 3 script that will show you how to connect to an FTP server as well as the various errors that might come up. As you already know, the ftplib module will be used, Figure 1 shows the online documentation of ftplib.

Create an autonomous Python script

Suppose that you have the following Python 3 code, named **aScript.py**, that you want to convert into an autonomous program:

```
print("Hello Linux Format!")
```

Doing the trick is pretty simple: the first thing that you will need to do is finding out where the Python 3 interpreted is located on your Linux file system:

```
$ which python3
/usr/bin/python3
```

Then, you will need to put the next line as the first line of your script, which depends on the output of the which command:

```
#!/usr/bin/python3
```

If you try to execute it now you will get the following kind of error message:

```
$ ./aScript.py
-bash: ./aScript.py: Permission denied
```

This is because **aScript.py** does not have the correct permissions that are needed in order to become an executable file. So, you will need to change the permissions of the file with the help of the *chmod* command:

```
$ chmod 755 aScript.py
```

From now on you can execute your Python 3 script as follows:

```
$ ./aScript.py
Hello Linux Format!
```

Please note that if you misspell the name of the Python 3 executable or its location you will get the following kind of error message:

```
$ ./FTPgui.py
-bash: ./FTPgui.py: /usr/bin/local/python3: bad interpreter: No such file or directory
```

If the directory that contains one or more autonomous Python scripts is in the `$PATH` environment variable, you will be able to find and execute your Python 3 scripts from anywhere on your Linux system.

But first, let us check the current value of the `$PATH` variable:

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
```

Now, create a directory named **bin** inside your home directory and add its full path to the `$PATH` variable:

```
$ mkdir ~/bin
$ export PATH="$HOME/bin:$PATH"
$ echo $PATH
/home/mtsouk/bin:/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
```

Then, put **aScript.py** inside the bin directory and use the *which* command to find it:

```
$ mv aScript.py ~/bin/
$ which aScript.py
/home/mtsouk/bin/aScript.py
$ aScript.py
Hello Linux Format!
```

You are done!

The following Python code, saved as **justConnect.py**, it connects to the specified FTP server that is given as a command line argument:

```
#!/usr/bin/python3

import os, sys
from ftplib import FTP

site =  str(sys.argv[1])
connection = FTP(site)
connection.login("anonymous", "aMail@gmail.com")
print ("Connected to %s" % site)
connection.close()
```

If **justConnect.py** cannot connect to an FTP server for some reason you will get the following kind of output:

```
socket.gaierror: [Errno 8] nodename nor servname provided, or not known
socket.gaierror: [Errno -2] Name or service not known
ConnectionRefusedError: [Errno 111] Connection refused
ConnectionRefusedError: [Errno 111] Connection refused
```

The good thing is that the error message will reveal the reason the connection failed. The first connection failed because the Linux machine was not connected to the Internet, the second one failed because the hostname was not valid, th0e third attempt failed because the hostname you gave does not support FTP and the fourth attempt failed because the username, the password or both were incorrect.

Reading data

We will now write a script that lists the contents of the root directory of the FTP server and another one that gets a file from an FTP server and saves it to your Linux machine.

The following Python code, when put after the Python code of **justConnect.py** but before the call to the `close()` method, shows how to get a listing of all files in the current directory of the FTP server:

```
files = []
files = connection.nlst()

for f in files:
    print(f)
```

The following Python code, saved as **getFile.py**, shows how to retrieve a file from the FTP server:

```
#!/usr/bin/python3

import os, sys
from ftplib import FTP

site =  str(sys.argv[1])
connection = FTP(site)
connection.login("anonymous", "aMail@gmail.com")
print ("Connected to %s" % site)

# Change current directory
newDirectory = '/pub/www'
connection.cwd(newDirectory)

file = '00_index.txt'
localfile = open(file, 'wb')
```

Quick tip

Cron jobs

A common use of FTP is for transferring log files between Linux machines. Using a Python script to do so allows you to run it as a cron job during weekends or late at night when Linux servers are under less stress.

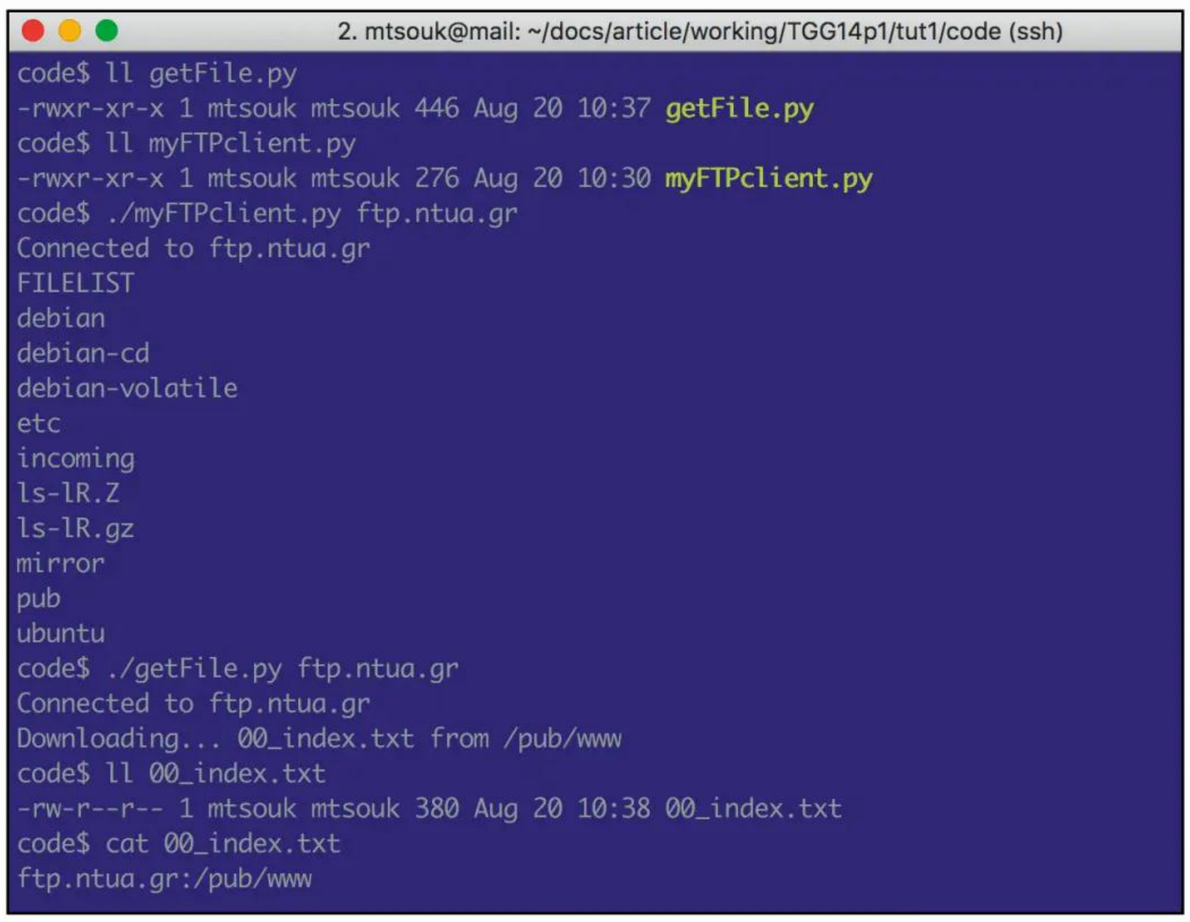


Figure 2: This figure shows getFile.py and myFTPclient.py in action. The first script retrieves a file from the FTP server whereas the second script lists the contents of the root directory of the FTP server

```
print("Downloading... %s from %s" %(file, newDirectory))

connection.retrbinary('RETR ' + file, localfile.write, 1024)
connection.close()
localfile.close()
```

The cwd() method is used for changing the current working directory of the FTP server. Additionally, you need to specify the name of the local file and open it for writing as a binary file using the `open()` call. Last, you can see that in order to retrieve a file, you need to call the `retrbinary()` function, which retrieves files using the binary format even if they are plain text files. Figure 2 (*on the previous page*) shows both **myFTPclient.py** and **getFile.py** in action.

Now that you know the basics, it is time to create an application with a GUI.

Quick tip

Useful resources
Find out more about the FTP protocol and its commands by reading RFC 959 found at: **www.ietf.org/rfc/rfc959.txt**. RFC 959 is the official specification of the File Transfer Protocol.

About the GUI application

As you can guess, the most important decision that you will have to make when designing a GUI application is about the widgets that you are going to use. First, you will need to select the desired FTP server from a List box. Then, you will need one Button widget named Connect in order to make the application to connect using the anonymous FTP account and get the list of files from the FTP server.

Finally, you will need a big area to display the contents of the root directory of the FTP server, a Text widget will be used for that. Now that we know the way the application is going to work, it is time to start the development. The name of the program will be **FTPgui.py**.

The application reads the list of available FTP servers from an external text file, named **FTP.txt**, where each line is the address of an FTP server, and displays it on screen using a List box, the `listOfServers` variable holds the name of the text file. Should you wish to change the list of the FTP servers, you should edit **FTP.txt** using your favourite text editor.

The Python 3 code that reads the text file is the following:

```
with open(listOfServers) as f:
    for line in f:
        lbox.insert(END, line.rstrip())
```

As you can see in the previous code, each line is inserted in the Listbox Widget after removing the newline character at the end of it. If you forget to remove the newline you will get the following kind of error when you press the Connect button:

```
socket.gaierror: [Errno 8] nodename nor servname provided, or not known
```

However, having just the GUI without implementing the actual functionality is useless! The next section will talk more about the implementation.

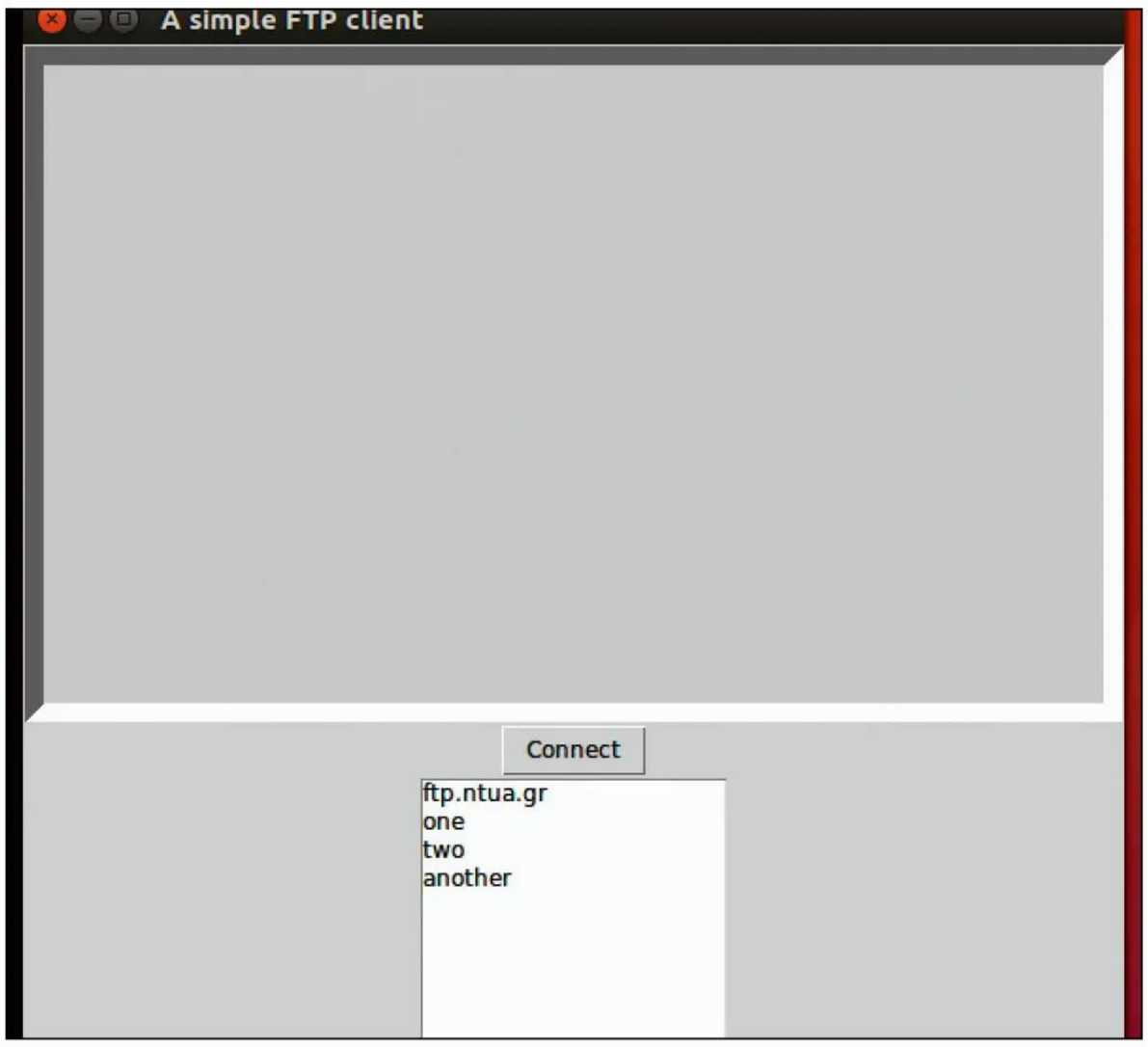


Figure 3: This is the initial screen of the FTPgui.py application. The list of the FTP servers is read by an external text file named FTP.txt that you can edit on your own

Implementing the GUI

The crucial code of the **FTPgui.py** application is the following because this is where the GUI is constructed:

```
root = Tk()
root.title("A simple FTP client")
c = Canvas(root, width=800, height=500)
c.pack()
text = Text(c, borderwidth=10, background='lightgray')
text.pack(side=RIGHT, expand=True)
cc = Button(root, text="Connect", command=connect)
cc.pack()
lbox = Listbox(root)
lbox.pack(side=LEFT, expand=1)
```

Examining FTP traffic

The best two ways to examine FTP traffic, which is unencrypted and therefore not secure, is by using *tcpdump* or *WireShark*. *WireShark* comes in two versions: as a GUI application named *wireshark* and as a command line application named *tshark*. For reasons of simplicity, what is going to be used here is *tcpdump*. The following *tcpdump* command filtered by *grep* shows all traffic that contains either PASS or USER, which is how the FTP protocol passes username and password information:

```
# tcpdump -A port ftp | grep -i 'PASS\|USER'
```

The next *tcpdump* command captures just two network packets because of the **-c** command line option:

```
# tcpdump -c 2 -A port ftp
```

Figure 5 (*right*) shows some FTP traffic, where you can see the username and password used for an FTP connection.

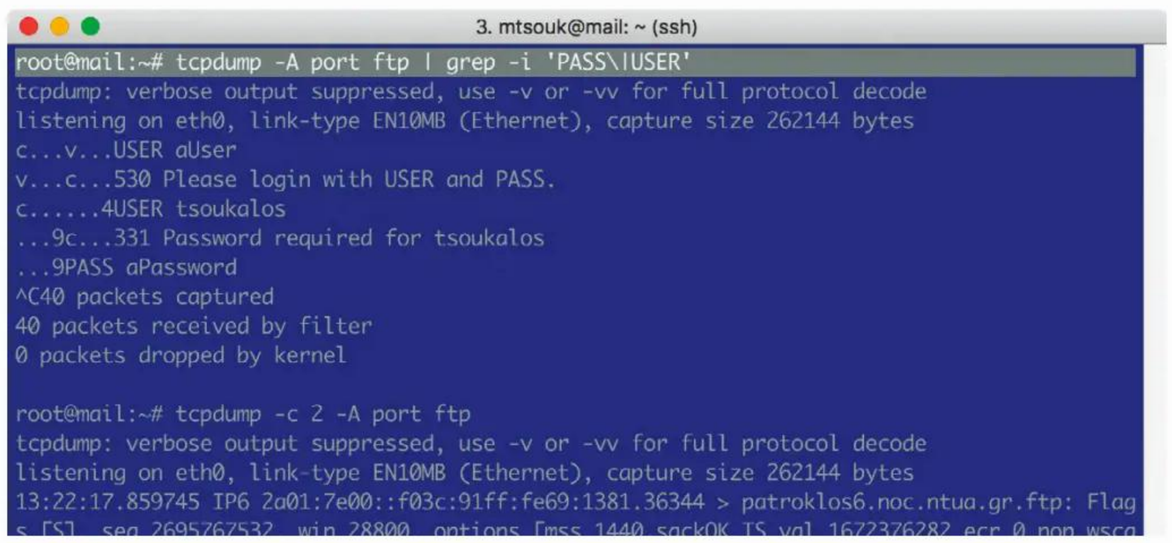


Figure 5: This figure illustrates the use of the tcpdump command line utility for capturing FTP network traffic. It also shows how easy it is to capture the username and password data of an FTP connection

As you can see from the definition of the cc variable, each time you press the Button Widget, Python calls the connect() function. The implementation of connect() is the following:

```
def connect():
    text.delete('1.0', END)
    selection = lbox.curselection()
    site = lbox.get(selection[0])
    connection = FTP(site)
    connection.login("anonymous", "aMail@gmail.com")
    print ("Connected to %s" % site)
    files = []
    files = connection.nlst()
    for f in files:
        text.insert('1.0', f)
        text.insert('1.0', "\n")

    connection.close()
    text.focus()
```

The first line of connect() clears the area of the Text widget. The second and third lines of connect() show how to read the current Listbox selection as an index and use the index to get the full text of the selection, which is saved in the site variable. Then, you read the list of files and directories from the FTP server, which you put in the Text widget area in order to be displayed and you are done!

Using the GUI

The application is pretty simple yet functional. The good thing is that you are allowed to edit the text from the Text Widget or copy it to another application. Figure 3 (*see left*) shows the initial screen of the GUI application whereas Figure 4 (above right) shows the GUI application in action.

The current version of **FTPgui.py** is far from perfect: it performs no error handling, it only supports the anonymous FTP account, it does not allow you to get a file and it cannot change the working directory of the FTP server. However, should you wish to write your own FTP client, **FTPgui.py** can be used as a starting point.

Creating an FTP server

The last section of the tutorial will show you how to create an FTP server in Python 3 using pyftpdlib.

The Python code of **myServer.py** is the following:

```
#!/usr/bin/python3

import os
from pyftpdlib.authorizers import DummyAuthorizer
from pyftpdlib.handlers import FTPHandler
from pyftpdlib.servers import FTPServer

auth = DummyAuthorizer()
auth.add_user('LXF', 'aPassword12345', '.',
perm='elradfmwM')
auth.add_anonymous(os.getcwd())
handler = FTPHandler
handler.authorizer = auth
handler.banner = "A simple FTP server."

address = ('0.0.0.0', 1234)
server = FTPServer(address, handler)

server.max_cons = 16
server.max_cons_per_ip = 2
server.serve_forever()
```

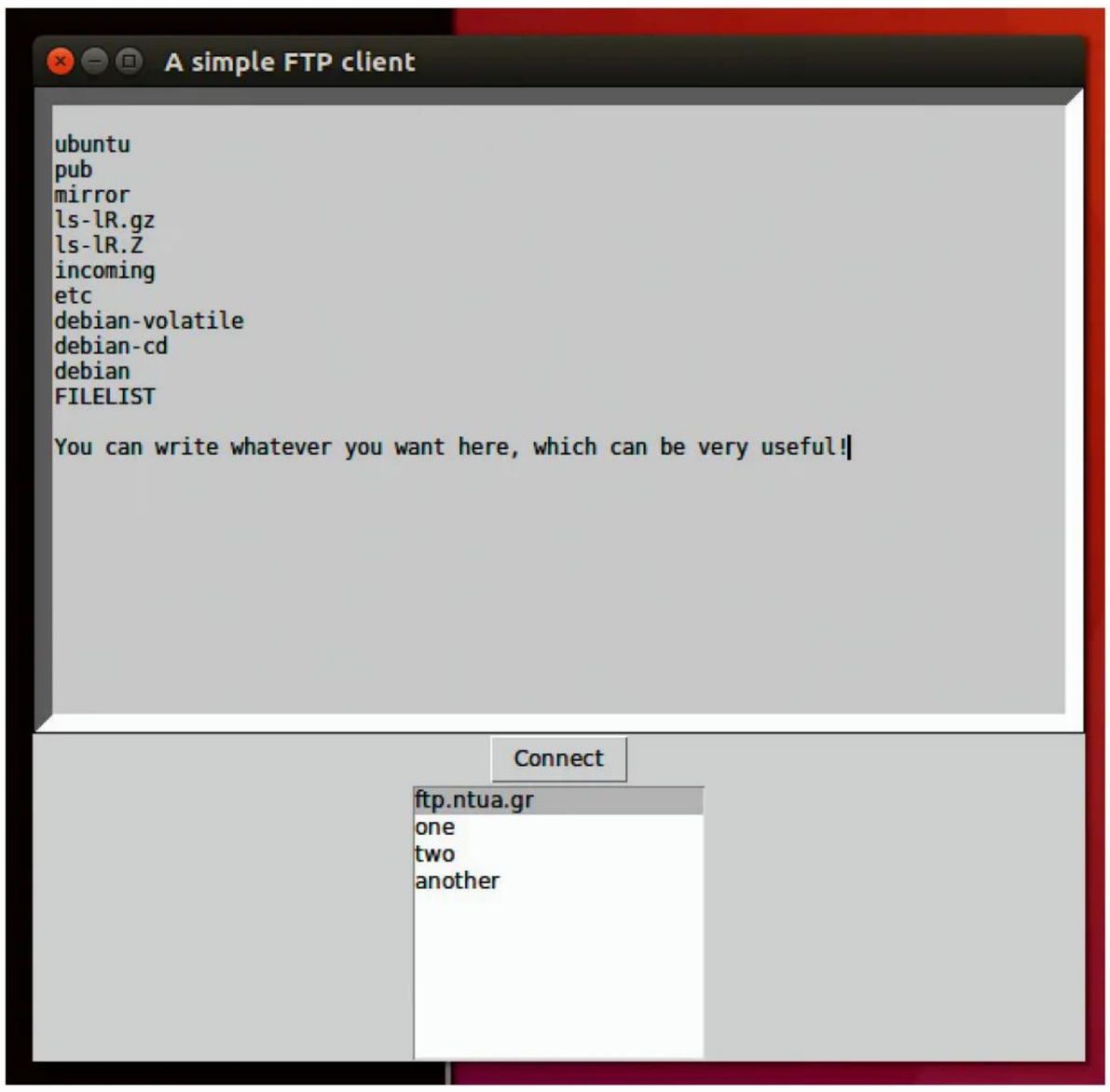
Before executing **myServer.py**, you should install the pyftpdlib Python 3 package (**python3-pyftpdlib** on Debian) if it is available for your Linux distribution. Otherwise, you can download and install pyftpdlib for Python 3 using *pip*:

```
# pip3 install pyftpdlib
```

Executing **myServer.py** and connecting to it produces the following type of output:

```
$ ./myServer.py
[I 2016-08-20 11:35:04] >>> starting FTP server on
0.0.0.0:1234, pid=18460 <<<
[I 2016-08-20 11:35:04] concurrency model: async
[I 2016-08-20 11:35:04] masquerade (NAT) address: None
[I 2016-08-20 11:35:04] passive ports: None
[I 2016-08-20 11:35:09] 2.86.6.235:50303-[] FTP session
opened (connect)
[I 2016-08-20 11:35:16] 2.86.6.235:50303-[anonymous]
USER 'anonymous' logged in.
[I 2016-08-20 11:35:18] 2.86.6.235:50303-[anonymous] FTP
session closed (disconnect).
[I 2016-08-20 11:35:22] 2.86.6.235:50305-[] FTP session
opened (connect)
[I 2016-08-20 11:35:29] 2.86.6.235:50305-[] USER 'LXF'
failed login.
[I 2016-08-20 11:35:39] 2.86.6.235:50305-[] FTP session
closed (disconnect).
[I 2016-08-20 11:35:40] 2.86.6.235:50306-[] FTP session
opened (connect)
[I 2016-08-20 11:35:43] 2.86.6.235:50306-[LXF] USER 'LXF'
logged in.
[I 2016-08-20 11:37:30] received interrupt signal
[I 2016-08-20 11:37:30] >>> shutting down FTP server (2
active socket fds) <<<
[I 2016-08-20 11:37:30] 2.86.6.235:50306-[LXF] FTP session
closed (disconnect).
```

You can find more information about pyftpdlib at **http://pythonhosted.org/pyftpdlib/tutorial.html**. ■



› Figure 4: This figure shows the FTPgui.py script in action. Although you can turn off the editing capabilities of the Text Widget, it is pretty handy to be able to edit the list of files!

Quick tip

Python 2 vs 3

There exist two major Python versions still in active use: Python 2 and Python 3. This tutorial uses Python 3 because this is where things are going. So, if you can, stop using Python version 2...

Python 3: Making scripts

Explore how to start making useful Python 3 scripts for various system administration tasks, including security checks and automating jobs

Python can easily replace *bash*, AWK and shell for system administration tasks—Python can even replace Perl! This tutorial will present various types of Python 3 scripts that will help you do many sysadmin jobs.

One thing that you should keep in mind is that you should break down big tasks into smaller ones and implement the smaller tasks instead—this will not only make the debugging process easier but it will allow many people to get involved in the process. The other important thing that you should remember is that most of the scripts will require root privileges to run because they deal with system devices and files, so be extra careful with your code.

Look at the following Python code, saved as **simpleSA.py**:

```
#!/usr/bin/python3
import sys
import re
total = 0
counter = dict()
for line in sys.stdin:
   total += 1
   line = line.rstrip('\n')
   parsed = list(map(‘’.join, re.findall(r'\"(.*?)\"|\[(.*?)\]|(\S+)', line)))
   host = parsed[0]
   if host in counter:
      counter[host] += 1
   else:
      counter[host] = 1
print("Processed ", total, " log records!")
for key in counter:
   print(key, ':', counter[key])
```

Quick tip

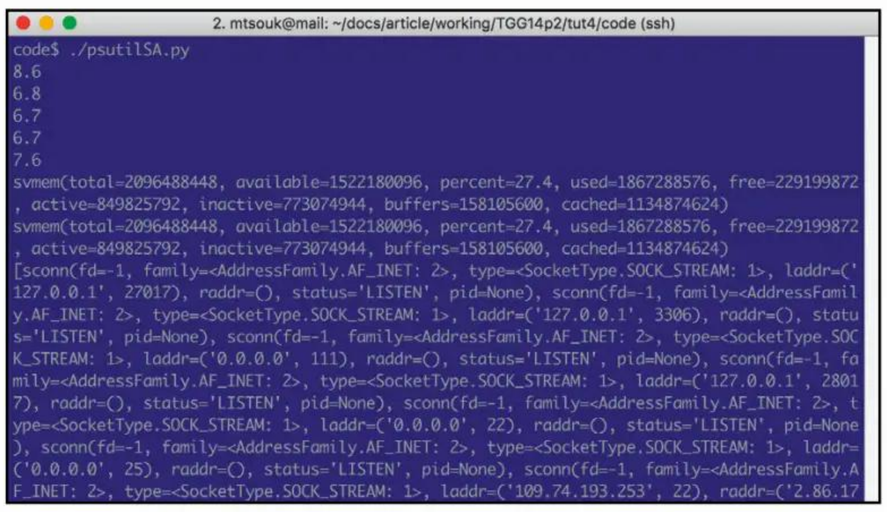
When writing scripts, you should have in mind that if you can’t solve a problem manually, you will not be able to generate a script for it. Put simply, always start from a manual solution before trying to automate a task with a script.

This script reads Apache log files and counts the number of connections per IP address—this job can be done using AWK but AWK has many limitations including the fact it can’t communicate with databases, send emails and create plots etc.

Our sample script

The script reads its input line by line and uses a dictionary to keep its data, which at the end is printed on screen. The regular expression that’s used separates the fields of each log entry in order to get the first one, which is the IP address of the machine that gets the data from the web server.

Unfortunately, the output isn’t ordered from the IP with the biggest number of appearances to the IP with the smallest number of appearances. This can be easily fixed by



› This is the output of the psutilSA.py script that uses the psutil module to get system information about a Linux system, including network status and memory usage

modifying the code that does the printing with:

```
for w in sorted(counter, key=counter.get, reverse=True):
        print (w, counter[w])
```

Executing **simpleSA.py** produces the next kind of output:

```
$ cat ./access.log | ./simpleSA.py
Processed  11273  log records!
xxx.55.aaa.245 979
zzz.121.bbb.13 340
...
ttt.46.rrr.nnn 1
ooo.ggg.161.mmm 1
```

The next script, named **storeMongo.py**, stores some of the data of an Apache log file to a MongoDB database:

```
#!/usr/bin/python3
import sys
import pymongo
import re
total = 0
connMongo = pymongo.Connection('mongodb://localhost:27017')
db = connMongo.LXF
logs = db.aSite
for line in sys.stdin:
        line = line.rstrip('\n')
        parsed = list(map(‘’.join, re.findall(r'\"(.*?)\"|\[(.*?)\]|(\S+)', line)))
        total = total + 1
        log = {
                'host': parsed[0],
                'date': parsed[3],
                'document': parsed[4],
```

The duties of a system admin

A system administrator has to perform many different tasks on a Linux system. First of all they have to keep the system up and running all the time, especially when the system is a critical server. Second, they have to deal with the generated log files and make sure that they are kept for as long as needed. Third, they have to create reports that give a general overview of the system status and keep track of the disk space of the system in order to avoid unpleasant surprises. Additionally, sysadmins have to automate as many tasks as possible; put simply, they have to develop scripts that automate the desired tasks. Last, a sysadmin has to overview the security of the system, detect security threats and install updates.

Dealing with all these tasks manually is impossible because it requires too much time. The solution to this problem is creating scripts. These offer another important advantage: when a script is fully debugged, it can run safely without any human intervention.

A very good book that can help you be productive and effective with your time is *Time Management for System Administrators* by Thomas A Limoncelli. If you want to learn more about the various tasks of a sysadmin, you should read *The Practice of System and Network Administration*, 2nd edition, by Thomas A Limoncelli, Christina J Hogan and Strata R Chalup, and *Unix and Linux System Administration Handbook*, 4th edition, by Evi Nemeth, Garth Snyder, Trent R Hein and Ben Whaley.

```
                'StatusCode': parsed[5],
                'size': parsed[6]
            }
            log_id = logs.insert(log)
            print("The _id of the inserted post is", log_id)
connMongo.close()
print("Processed ", total, " log records!")
```

The **pymongo** module is used for communicating with the *MongoDB* database. As we did in **simpleSA.py,** a regular expression is used for parsing each log entry. Running it would generate the following kind of output:

```
$ cat /srv/www/www.mtsoukalos.eu/logs/access.log.1 | ./
storeMongo.py

The _id of the inserted post is 57bff466b83a88414a29186d
...
Processed  11273  log records!
```

Please note that you might need to install the **python3-pymongo** package in order to get PyMongo for Python 3. You can now check the *MongoDB* server to see the format of the saved data:

```
$ mongo
MongoDB shell version: 2.4.10
connecting to: test
> use LXF
switched to db LXF
> db.aSite.findOne()
{
        "_id" : ObjectId("57bff463b83a88414a28ec6f"),
        "size" : “2578”,
        "host" : "8.29.198.27",
        "document" : "GET /rss.xml HTTP/1.1",
        "StatusCode" : "200",
        "date" : "14/Aug/2016:06:29:12 +0300"
}
```

We will not continue talking about what you can do with the *MongoDB* data but as long as you have your log files stored in a database the possibilities are endless. Of course, you can use any database you want including *MySQL*, *PostgreSQL* and *Oracle*—it is just that *MongoDB* might be more appropriate for unstructured data.

The cross-platform **psutil** module, a very important module for administrators, allows you to retrieve information on running processes and system utilisation. You can install **psutil** with `# apt-get install python3-psutil`. Alternatively, you can use **pip3** to install it. You can then find its version:

```
import psutil
psutil.__version__
‘2.1.1’
```

The **psutilSA.py** script illustrates some of the capabilities of the **psutil** module:

```
#!/usr/bin/python3
import psutil
for x in range(5):
   print(psutil.cpu_percent(interval=1))
for x in range(2):
   print(psutil.virtual_memory())
print(psutil.net_connections())
p = psutil.Process(1)
print(p.name(), p.uids())
```

First, you get information about the CPU usage, after that you get information about the virtual memory, you see all network connections and last, you get information about the process with process ID of 1. You can find more information about **psutil** at **https://github.com/giampaolo/psutil**. (See the output of the script top left.)

Quick tip

When you want to draw something and create a PNG file? Use *Matplotlib*! This can create impressive plots and graphs and can be easily used in scripts.

```
2. mtsouk@mail: ~/docs/article/working/TGG14p2/tut4/code (ssh)
tut4$ ./code/findBig.py .
./figures/psutilSA.png
Size:  931901
tut4$ ./code/duPy.py .
               Start from: /home/mtsouk/docs/article/working/TGG14p2/tut4
        Checking: /home/mtsouk/docs/article/working/TGG14p2/tut4/figures
1331808 bytes
        Checking: /home/mtsouk/docs/article/working/TGG14p2/tut4/code
10063 bytes
tut4$ ./code/duPy.py /home
               Start from: /home
        Checking: /home/mtsouk
722331270 bytes
tut4$ ./code/duPy.py /home/mtsouk/code
               Start from: /home/mtsouk/code
        Checking: /home/mtsouk/code/Haskell
887 bytes
        Checking: /home/mtsouk/code/python
18398220 bytes
        Checking: /home/mtsouk/code/perl
158 bytes
        Checking: /home/mtsouk/code/C
545206 bytes
tut4$ ./code/findBig.py /home
/home/mtsouk/.cabal/packages/hackage.haskell.org/00-index.tar
Size:  207514624
tut4$ cd code
code$ ./findBig.py .
```

› **Here’s the output from the two scripts, findBig.py and duPY.py, which work with files and directories**

Scripting for security

This section will explain how to use Python 3 to check the security of your Linux system. The following Python script, saved as **notWWW.py**, finds all files that are located inside the directory of a website and don’t belong to the www-data user or the root user:

```
#!/usr/bin/python3
import os
import sys
import pwd
if len(sys.argv) >= 2:
   directory = str(sys.argv[1])
else:
   print('Not enough arguments!')
   sys.exit(0)
```

»

```
noBytes = 0
fileName = ''
owner = 'www-data'
for root, dirs, files in os.walk(directory):
    for file in files:
        pathname = os.path.join(root,file)
        if os.path.exists(pathname):
            st = os.stat(pathname)
            try:
                userinfo = pwd.getpwuid(st.st_uid)
                ownername = pwd.getpwuid(st.st_uid).pw_name
            except:
                print("Unexpected error:", sys.exc_info()[0],
pathname)
            if (ownername != owner) and (ownername != 'root'):
                print(pathname)
                print ('Owner: ', ownername)
```

Executing the above script generates the following output:

```
$ ./notWWW.py /a/directory
/a/directory/aSite/public_html/
speeches/20152016/28-02-2016.mp3
Owner:  mtsouk
```

It might not be clear, but this is a truly useful script!

Creating disk utilities

Next, we'll develop two small scripts that work on directories and files. The following script, saved as **duPy.py**, calculates the sizes of the subdirectories that exist in a root directory:

```
#!/usr/bin/python3
import os
import sys
def directorySize(path):
    total = 0
    for entry in os.listdir(path):
        fullPath = os.path.abspath(os.path.join(path, entry))
        if os.path.isdir(fullPath):
            total += directorySize(fullPath)
        elif os.path.isfile(fullPath):
            total += os.path.getsize(fullPath)
    return total
if len(sys.argv) >= 2:
    directory = os.path.normpath(os.path.abspath((sys.
argv[1])))
    print('\t\tStart from:', directory)
else:
    print('Not enough arguments!')
    sys.exit(0)
dirList = next(os.walk(directory))[1]
for current in dirList:
    current = os.path.join(directory, current)
    print('\tChecking:', current)
    print(directorySize(path = current), 'bytes')
```

The root directory is given as a command-line argument—if your current directory contains no subdirectories, the script won't generate output. The `directorySize()` function visits all the subdirectories of the directory with the help of the `os.listdir()` function. If you're comfortable with *Matplotlib*, you can generate a plot from the output. The following script, called **findBig.py**, finds the biggest file in a directory:

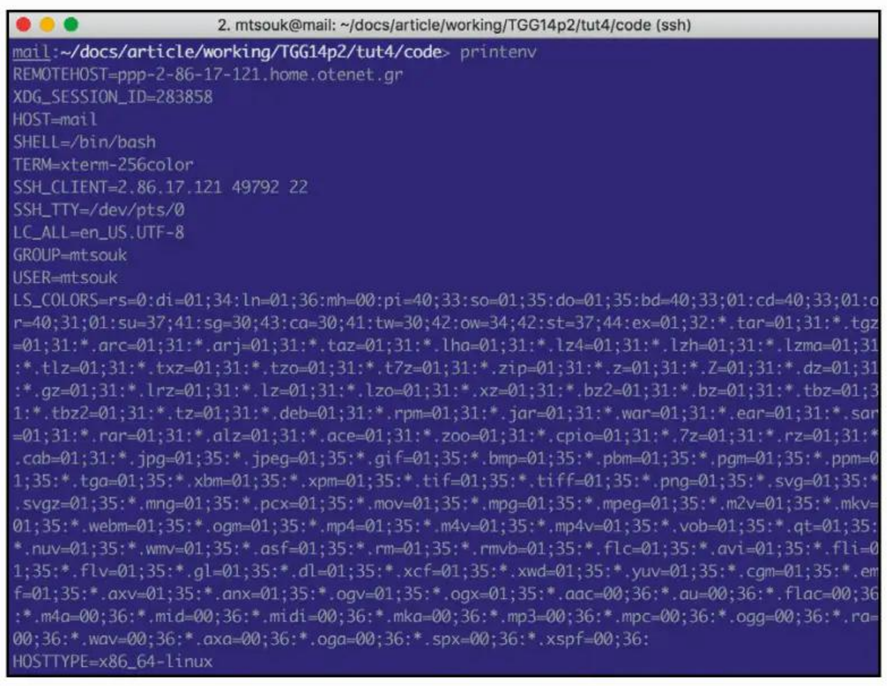
```
#!/usr/bin/python3
import os
import sys
if len(sys.argv) >= 2:
    directory = str(sys.argv[1])
else:
    print('Not enough arguments!')
    sys.exit(0)
noBytes = 0

fileName = ''
for root, dirs, files in os.walk(directory):
    for file in files:
        pathname = os.path.join(root,file)
        if os.path.exists(pathname):
            if noBytes < os.path.getsize(pathname):
                noBytes = os.path.getsize(pathname)
                fileName = pathname
print(fileName)
print ('Size:', noBytes)
```

The call to `os.path.exists(pathname)` is made to ensure the file exists, in case it's a symbolic link that points to nowhere or in the rare case that it's been deleted. (See the output from

Quick tip

Apart from Python, both Perl and Ruby are good candidates for writing sysadmin scripts. Choosing a programming language for writing your scripts is not an easy task however; when you choose one of them, stick with it!



This is a small part from the output of the export Bash shell command and the printenv tcsh shell command. Both commands show the defined environment variables

Environment variables demystified

A good sysadmin must know about environment variables. First of all, you should know that there are multiple shells; therefore, not everyone is using the *Bash* shell, which means that you might use a different way to define and alter environment variables. The best way to find the available shells is by looking at **/etc/shells**, which lists the full path of each shell.

As different shells use different ways to set environment variables, we'll quickly deal with the two most popular shells: *Bash* and *C* shell. Defining an environment variable in *Bash* is done as follows:

```
$ export AVAR='123'
```

Most often you will need to add a path to the PATH variable, which is pretty easy:

```
$ export PATH="$HOME/bin:$PATH"
```

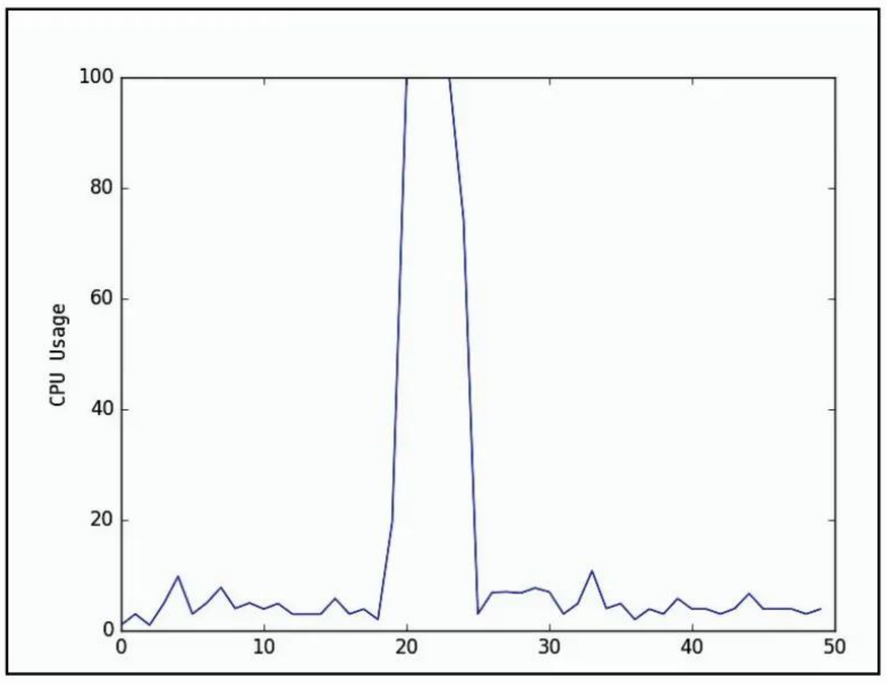
The previous command adds the **bin** directory located inside your **home** directory, as defined by the `HOME` environment variable, to the existing value of the PATH variable! On the other hand, the C shell uses the next way:

```
$ setenv EDITOR vi
```

Nowadays, most Linux systems use *TENEX C* Shell instead of the *C* Shell, which is an enhanced version of Berkeley *csh*.

Please bear in mind that the most important reason for using full path names for the commands you want to run in your crontab entries and your scripts is that a wrongly defined PATH variable can make you execute a different executable than the one that you intended to, which can be a major security threat.

The screenshot (above) in the main tutorial shows a sample output from the `export` command on the *Bash* shell and the `printenv` command on the *tcsh* shell. Almost all environment variables come predefined so you won't have to deal with them very often.



› This is the output of plotCPU.py, a Python 3 script that uses both *Matplotlib* and psutil to create a handy plot

both **duPy.py** and **findBig.py**, on the previous page). Error-handling code should be present in every Python 3 program, but especially when writing sysadmin scripts where an error can cause unpredictable results and affect the stability of your system. Imagine that you want to execute this Python command: `f = open('aFile.txt')` .

Opening a file for reading might fail for various reasons including not having the appropriate file permissions and trying to read a file that doesn't exist. So, a better version of the previous code, saved as **errorHandle.py**, would be:

```
#!/usr/bin/python3
import sys
try:
    f = open('./mongodb.conf')
    s = f.readline()
    s = s.rstrip('\n')
    print(s)
except IOError as e:
    errno, strerror = e.args
    print("I/O error({0}): {1}".format(errno,strerror))
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

The **errorHandle.py** script just reads and prints the first line of the file **mongodb.conf**. Depending on the situation, the output from the error handling code might look like this:

```
$ ls -l mongodb.conf
---------- 1 mtsouk mtsouk 2259 Aug 26 18:01 mongodb.
 conf
$ ./errorHandle.py
I/O error(13): Permission denied
$ ls -l mongodb.conf
ls: cannot access mongodb.conf: No such file or directory
$ ./errorHandle.py
I/O error(2): No such file or directory
```

Note: The first try failed because we didn't have the right file permissions whereas the second failed because **mongodb.conf** could not be found.

Visualising using Python 3

This section will demonstrate how to visualise data using *Matplotlib*. The following script, **plotCPU.py**, creates a beautiful and handy visualisation that shows the CPU usage on your Linux machine over a small period of time:

```
#!/usr/bin/python3
import psutil
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
x = []
y = []
for k in range(50):
    x.append(k)
    temp = psutil.cpu_percent(interval=1)
    y.append(temp)
plt.ylabel('CPU Usage', family = 'monospace')
plt.plot(x, y)
plt.savefig('./CPUplot.png')
plt.show()
plt.close()
```

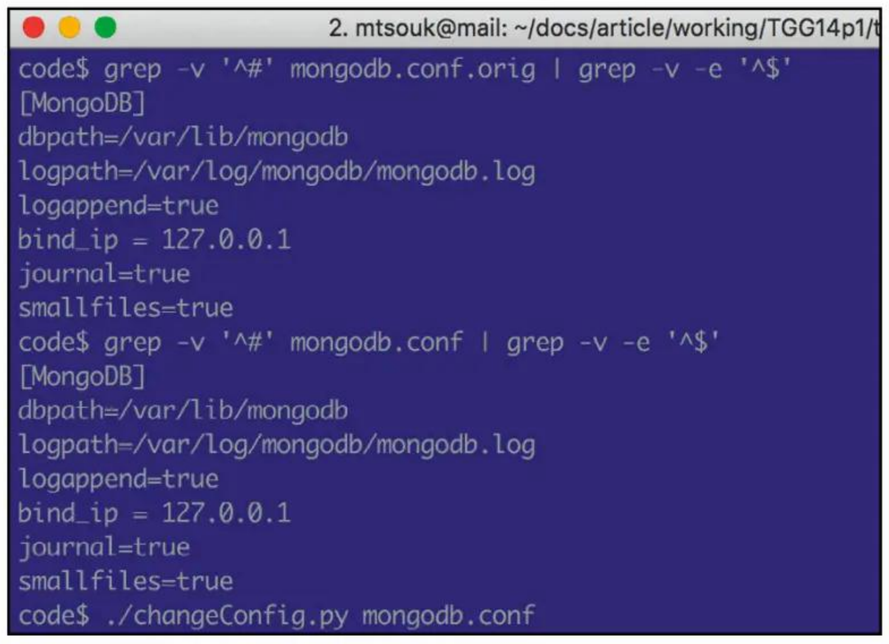
You can change the number of samples you get, which is defined in the for loop. (The output of plotCPU.py when executed on a not so busy Linux system is pictured, left.) The Python module we're using to make changes to configuration (config) files is called **ConfigParser**, a module that comes with Python 3. The contents of **changeConfig.py** are:

```
#!/usr/bin/python3
import configparser
from itertools import chain
parser = configparser.ConfigParser()
with open("./mongodb.conf") as lines:
    parser.read_file(lines)
print("dbpath: ", parser.get('MongoDB', 'dbpath'))
parser.set('MongoDB', 'dbpath', '/tmp')
print("New dbpath: ", parser.get('MongoDB', 'dbpath'))
print("smallfiles: ", parser.get('MongoDB', 'smallfiles'))
parser.remove_option('MongoDB', 'smallfiles')
with open('./mongodb.conf', 'w') as configfile:
    parser.write(configfile)
```

If you try to read a key that doesn't exist, you will get the following kind of error message:

```
configparser.NoOptionError: No option 'smallfiles' in section:
'MongoDB'
```

If your config files don't have sections, you should insert a dummy section header to satisfy the requirements of **ConfigParser.** For the *mongodb.conf* file the name of the dummy section can be [MongoDB]. This can also be done with Python code but it's simpler this way—if your config file has sections, then you don't need to deal with it.(Pictured below is the before and after status of the **/etc/mongodb.conf** config file.) Note: You shouldn't make changes to a config file that's in use; always work with a backup file first. This next *grep* command is also handy as it doesn't show empty lines and lines that begin with an '#' in a text file, which means you can see the effects of the script: `$ grep -v '^#' mongodb.conf | grep -v -e '^$'` . As a sysadmin you'll deal with config files all the time—learn a module that can work with them and create your own scripts and you'll save lots of time. ■



› The file changeConfig.py shows you how to read a config file, make changes to it and save the changes—knowing how to parse configuration files is really important

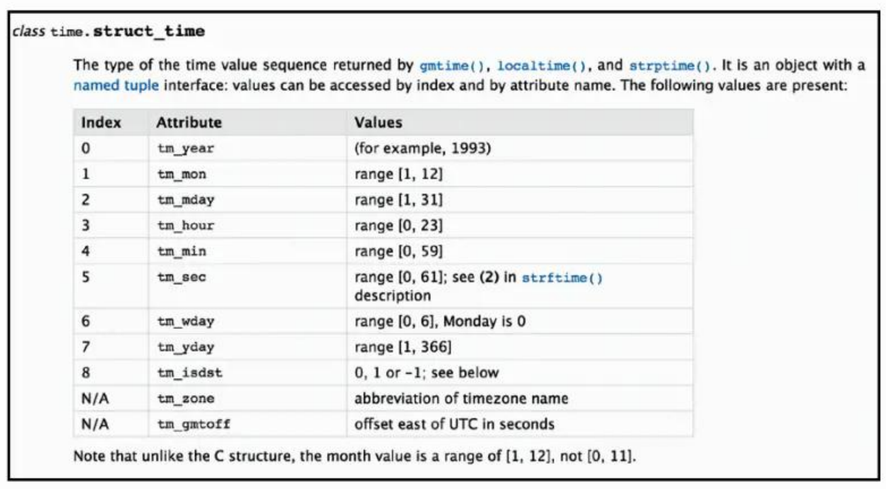
Python: times, dates & numbers

Times and dates are tricky things, more than your might realise. We look at how you can deal with them within your Python 3 programs

Dealing with dates and times can seem a simple task, but if you consider that some months have 30 days, others have 31, and February might have 29 days instead of the usual 28, things get more complicated.

This tutorial is about using numbers, dates and times as well as doing conversions and calculations with dates and times in Python 3, along with using Matplotlib for plots.

There exist two main modules for dealing with dates and times: time and datetime. So, depending on what you want to do, you should choose the appropriate Python 3 module; when in doubt use datetime. Both are standard Python modules so there's no installation required. You can find the documentation page for both at **https://docs.python.org/3/library/datetime.html** and **/time.html**, respectively.



class `time.struct_time`

The type of the time value sequence returned by `gmtime()`, `localtime()`, and `strptime()`. It is an object with a named tuple interface: values can be accessed by index and by attribute name. The following values are present:

Index	Attribute	Values
0	tm_year	(for example, 1993)
1	tm_mon	range [1, 12]
2	tm_mday	range [1, 31]
3	tm_hour	range [0, 23]
4	tm_min	range [0, 59]
5	tm_sec	range [0, 61]; see (2) in `strftime()` description
6	tm_wday	range [0, 6], Monday is 0
7	tm_yday	range [1, 366]
8	tm_isdst	0, 1 or -1; see below
N/A	tm_zone	abbreviation of timezone name
N/A	tm_gmtoff	offset east of UTC in seconds

Note that unlike the C structure, the month value is a range of [1, 12], not [0, 11].

› Figure 2: Various fields of the time.struct_time class, which has a named tuple interface. This means that its values can be accessed by index and by attribute name

The datetime continuum

First let's learn perform some basics using the Python 3 shell:

```
>>> from datetime import datetime
>>> datetime.today()
datetime.datetime(2016, 9, 3, 22, 24, 44, 913316)
>>> now = datetime.today()
>>> print(now)
2016-09-03 22:25:39.824785
>>> from datetime import timedelta
>>> print(now + timedelta(days=1))
2016-09-04 22:25:39.824785
>>> then = now + timedelta(days=1)
>>> then - now
datetime.timedelta(1)
>>> (then - now).days
1
```

The `datetime.today()` function returns the current date and time as a datetime object whereas the datetime.timedelta module allows you to do some simple calculations with dates. The next two sections will continue illustrating how to do calculations with dates and times.

Calculations with dates

Most of the times just knowing a date is not enough - it is how you use that date that make it really useful. Here's how to calculate the number of days until Christmas:

```
>>> from datetime import date
>>> today = date.today()
>>> print(today)
2016-09-05
>>> Christmas = date(2016, 12, 25)
>>> print(Christmas - today)
111 days, 0:00:00
```

The next shows which date will be in 90 days time:

```
>>> from datetime import timedelta
>>> print(today+timedelta(days=90))
2016-12-04
>>> print((today+timedelta(days=90)).weekday())
6
```

The `timedelta()` function does all the work. Using the Calendar module, you can also print the name of the day:

```
>>> import calendar
>>> dayIndex = (today+timedelta(days=90)).weekday()
>>> calendar.day_name[dayIndex]
'Sunday'
```

So, up to now you know how to do some basic calculations with dates, let's switch to times. This is how to do time calculations in Python 3:

```
>>> from time import time
>>> now = time()
>>> print(now)
1473090180.730119
>>> then = time()
>>> timeDifference = then - now
>>> print(timeDifference)
8.078937768936157
>>> print(minutes, seconds)
0.0 8.078937768936157
>>> print("The time difference is %.2f seconds" %
timeDifference)
The time difference is 8.08 seconds
```

So, if you want to print the time difference in a better format, you can use the `print()` function as shown in the last command. Another very useful operation is converting the local time to UTC format:

```
>>> import time
>>> import datetime
>>> LocalTime = datetime.datetime.now()
>>> print(LocalTime)
2016-09-05 18:49:09.149180
>>> EpochSecond = time.mktime(LocalTime.timetuple())
>>> print(EpochSecond)
1473090549.0
>>> utcTime = datetime.datetime.
utcfromtimestamp(EpochSecond)
>>> print(utcTime)
2016-09-05 15:49:09
>>> LocalTime.ctime()
'Mon Sep  5 18:49:09 2016'
>>> utcTime.ctime()
'Mon Sep  5 15:49:09 2016'
```

The last example shows how to convert UTC to local time:

```
>>> from datetime import datetime
>>> UTCtime = datetime.utcnow()
>>> type(UTCtime)
<class 'datetime.datetime'>
>>> print(UTCtime)
2016-09-05 15:54:23.554440
>>> from dateutil import tz
>>> HERE = tz.tzlocal()
>>> print(HERE)
tzlocal()
>>> UTC = tz.gettz('UTC')
>>> gmt = UTCtime.replace(tzinfo=UTC)
>>> gmt.astimezone(HERE)
datetime.datetime(2016, 9, 5, 18, 54, 23, 554440,
tzinfo=tzlocal())
```

As you can see, the UTCtime variable is a datetime object, which means that you can manually put any date and time in order to convert it to another timezone.

Working with Epoch time

The simplest way to get the current date expressed in Epoch time is with the date command line utility:

```
$ date +%s
1472933779
```

Two operations are important here: being able to convert a date and time to Epoch time and vice versa. The following commands executed in the Python 3 shell show how:

```
>>> epochTime = 1472933779
>>> import datetime
>>> newDateTimeObject = datetime.datetime.
fromtimestamp(epochTime)
>>> print(newDateTimeObject)
2016-09-03 23:16:19
>>> type(newDateTimeObject)
<class 'datetime.datetime'>
>>> newDateTimeObject.timestamp()
1472933779.0
```

Converting epoch time to datetime object can be done with the datetime.datetime.fromtimestamp() method. As soon as you have a datetime object, you can use the timestamp() method to convert it to epoch time. Once you know these two operations, everything is simple:

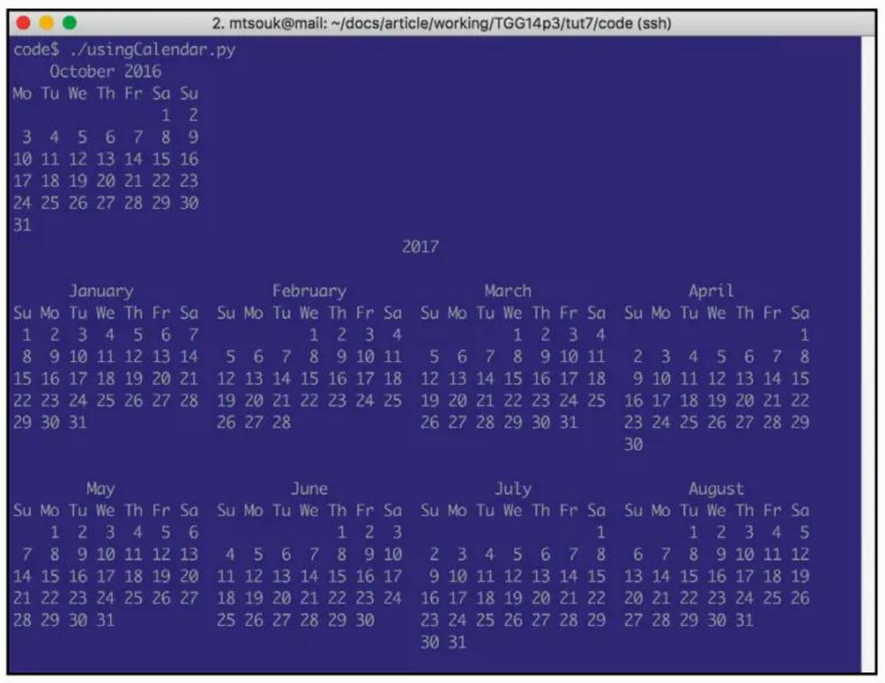
```
>>> import time
>>> time.strftime('%Y-%m-%d %H:%M:%S', time.
localtime(epochTime))
'2016-09-03 23:16:19'
>>> newDate = time.strftime('%Y-%m-%d %H:%M:%S',
time.localtime(epochTime))
>>> type(newDate)
<class 'str'>
```

The newDate object is a string but the next section will show how to convert a string to a datetime object.

Date to time and back again

Most of the times you will have to parse a string to create a datetime object. The following output shows how to convert a string to a datetime format and vice versa:

```
>>> from datetime import datetime
>>> myString = '2016-09-23 21:30:15'
>>> datetime.strptime(myString, "%Y-%m-%d %H:%M:%S")
datetime.datetime(2016, 9, 23, 21, 30, 15)
>>> aDate = datetime.strptime(myString, "%Y-%m-%d
%H:%M:%S")
>>> type(aDate)
<class 'datetime.datetime'>
>>> aDate.strftime('%Y')
'2016'
>>> aDate.strftime('%Y-%m-%d')
'2016-09-23'
```



› Figure 3: The output of usingCalendar.py, which demonstrates how to print a month's calendar or a year's calendar in text and HTML format

Quick tip

Epoch time

Epoch Time (aka UNIX time) is the number of seconds that have elapsed since Thursday 1st of January 1970, 00:00:00 UTC without counting the leap seconds.

Working with fractions

A fraction is a number of the form a/b, where a is the Numerator and b is the Denominator, which cannot be zero. Both of these are integers. Python 3 offers the Fractions module to help you work with fractions:

```
>>> import fractions
>>> a = fractions.Fraction(4, 5)
>>> b = fractions.Fraction(3, 5)
>>> print(a+b)
7/5
>>> print(a/b)
4/3
>>> print(b/a)
3/4
>>> float(a/b)
1.3333333333333333
>>> int(a/b)
1
>>> aa = 4/5
>>> bb = 3/5
>>> print(aa+bb)
1.4
>>> print(aa/bb)
1.3333333333333335
>>> print(bb/aa)
0.7499999999999999
>>> float(b/a)
0.75
```

Although the use of fractions is not always necessary, when you need to make calculations with fractions, the fractions module will save the day. In the example shown here, you can see that the output of the aa/bb division is less useful than the output of b/a; additionally 3/4 is exactly 0.75 where the output of bb/aa approaches this number.

```
>>> aDate.strftime('%d-%m-%Y')
'23-09-2016'
>>> type(aDate.strftime('%d-%m-%Y'))
<class 'str'>
```

The datetime.datetime.strptime() method converts a string to a datetime object; its second argument is the format for your string whereas the first argument is the string itself. The strftime() method of a datetime object is used for converting it to a string and you only need to provide the string format to it.

If your format does not match the actual data you are going to get the following kind of error message:

```
ValueError: time data '2016-09-23 21:30:15' does not match format '%m-%d %H:%M:%S'
```

The directives for the module enable you to specify the format of a date and time string in order to convert it to a date and time object as well as the string format of the output when you want to convert a datetime object into a string.

Quick tip

Complex Numbers
Python has easy support for complex numbers thought the use of the complex() function. However, should you wish to do advanced things with complex numbers, you should use the cmath module **https://docs.python.org/3.3/library/cmath.html**.

Format to format

It is not unusual to want to convert one date format to another. Put simply, you want to convert a date format of Day-Month-Year to Year-Month-Day. Look at the following interaction with the Python 3 shell to learn more about it:

```
>>> aDate = '5-9-2016'
>>> import datetime
>>> aDTobject = datetime.datetime.strptime(aDate, "%d-%m-%Y")
>>> print(aDTobject.strftime('%Y-%m-%d'))
2016-09-05
```

The way to do it, is to convert the first string into a datetime object that you then convert to the desired format using the strftime() method. Although the technique is not difficult, you should be very careful with all the available directives.

Time.struct_time objects

Python 3 offers the time.struct_time object that comes with a named tuple interface, which means that its values can be accessed both by index and by attribute name. You can work with a time.struct_time object as follows:

```
>>> import time
>>> myTime = time.gmtime()
>>> type(myTime)
<class 'time.struct_time'>
>>> print(myTime)
time.struct_time(tm_year=2016, tm_mon=9, tm_mday=5, tm_hour=19, tm_min=12, tm_sec=4, tm_wday=0, tm_yday=249, tm_isdst=0)
>>> time.strftime('%H:%M of %Y-%m-%d', myTime)
'19:12 of 2016-09-05'
>>> myTime[1]
9
>>> myTime.tm_mon
9
```

Figure 2 shows the fields of the time.struct_time class as found in the Python 3 documentation of the time module.

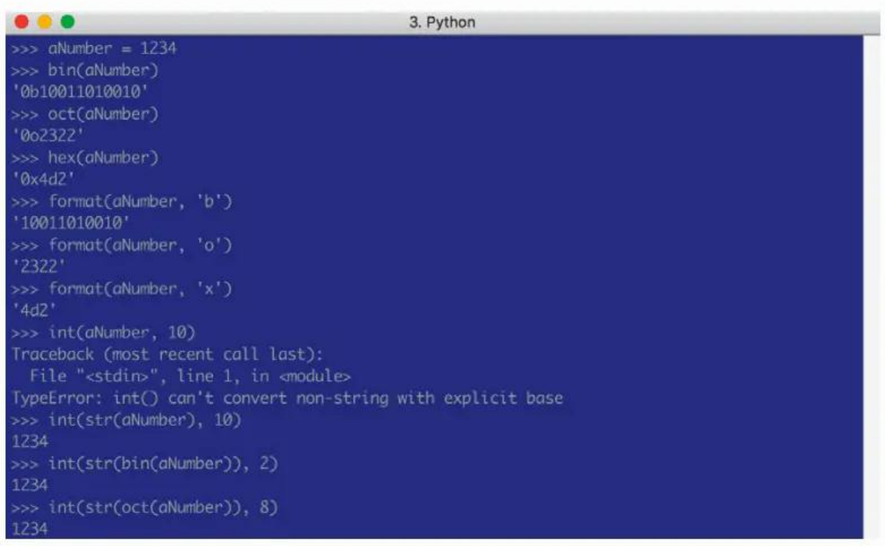
Calendar module

The Calendar module tries to mimic the cal(1) utility. The good thing is that the calendar module can generate both plain text and HTML output. However, using text output is a better choice because the HTML output might be ugly. The following Python 3 code, saved as **usingCalendar.py**, illustrates the use of the calendar module:

```
#!/usr/bin/python3
import calendar

# Plain Text
textCalendar = calendar.TextCalendar(calendar.MONDAY)
textCalendar.prmonth(2016, 10)
textCalendar = calendar.TextCalendar(calendar.SUNDAY)
print(textCalendar.formatyear(2017, 2, 1, 1, 4))
# HTML output
htmlCalendar = calendar.HTMLCalendar(calendar.MONDAY)
print(htmlCalendar.formatmonth(2016, 10, withyear=True))
# print(htmlCalendar.formatyear(2017, width=3))
```

The formatyear() function takes five parameters. The first one is the desired year, the second declares the width of the date column, the third one defines the number of lines per week. The fourth one is the number of spaces between month columns, the last parameter denotes the number of columns in the output, where each column is a month. You can format the text output as you want. The formatmonth() function generates a month's calendar in HTML format and requires three parameters, the year, the month and whether you want to print the year on the output or not. Figure 3 (*see the previous spread*) shows the output from **usingCalendar.py**. You can find more information about the calendar module at **https://docs.python.org/3/library/calendar.html**.Another interesting problem is finding leap years. The



› Figure 4: Here's the use of the bin(), oct() and hex() functions as well as the use of format() and int() functions in order to convert integers between the various numeral systems

Working with Binary, Octal and Hexadecimal numbers

Binary, Octal and Hexadecimal numbers are widely used in Computer Science. The following output shows how to convert between the various numeral systems using the bin(), oct() and hex() functions:

```
>>> aNumber = 10
>>> bin(aNumber)
'0b1010'
>>> oct(aNumber)
'0o12'
>>> hex(aNumber)
'0xa'
>>> int('10', 16)
16
>>> int('11101010', 7)
958349
```

The int() function converts a string to an integer, the second argument of int() is the base of the numeral system. This can be very handy when you want to convert a number to a numeral system other than Binary, Octal and Hexadecimal. So, the number 11101010 in base-7 numeral system is equal to 958349 in base-10 numeral system.

The format() function also converts a number into Binary, Octal and Hexadecimal without displaying the 0b, 0o and 0x prefixes.

Figure 4 (*above*) shows the Python 3 shell running operations that allow you to handle Binary, Octal and Hexadecimal integers. It also shows the use of the format() and int() functions for converting integers and strings to the various numeral systems.

following Python 3 script, saved as **leapYears.py**, accepts a list of years as input and checks each one of them using a relatively simple expression; if it finds a leap year it prints it:

```
#!/usr/bin/python3
import calendar
import sys

def isLeapYear(n):
   if n % 400 == 0:
      return True
   if n % 100 == 0:
      return False
   if n % 4 == 0:
      return True
   else:
      return False
# Get command line arguments
# and process them one by one
for argument in sys.argv[1:]:
   year = int(argument)
   if isLeapYear(year):
      print(year, 'is a leap year!')
```

Please note that the order you put the if statements in the `isLeapYear()` function is significant; therefore, if you change their order, the algorithm will not work and generate erroneous results. Executing **leapYears.py** produces this:

```
$ ./leapYears.py 2016 400 2100 2000 2020
2016 is a leap year!
400 is a leap year!
2000 is a leap year!
2020 is a leap year!
```

Alternatively, you can use the `isleap()` function of the Calendar module. Note that **leapYears.py** does not perform any exception handling, so don't give it a string to check!

```
if calendar.isleap(year):
```

Finding Monday

Let's code a function to find the date of the last Monday of a month. The following Python 3 code, saved as **lastMonday.py**, calculates the last Monday and Wednesday of a month:

```
#!/usr/bin/python3

import calendar
cal = calendar.Calendar(0)
month = cal.monthdatescalendar(2016, 9)
lastWeek = month[-1]
print('Last Monday', lastWeek[0])
print('Last Wednesday', lastWeek[2])
```

Executing **lastMonday.py** generates this:

```
$ ./lastMonday.py
Last Monday 2016-09-26
Last Wednesday 2016-09-28
```

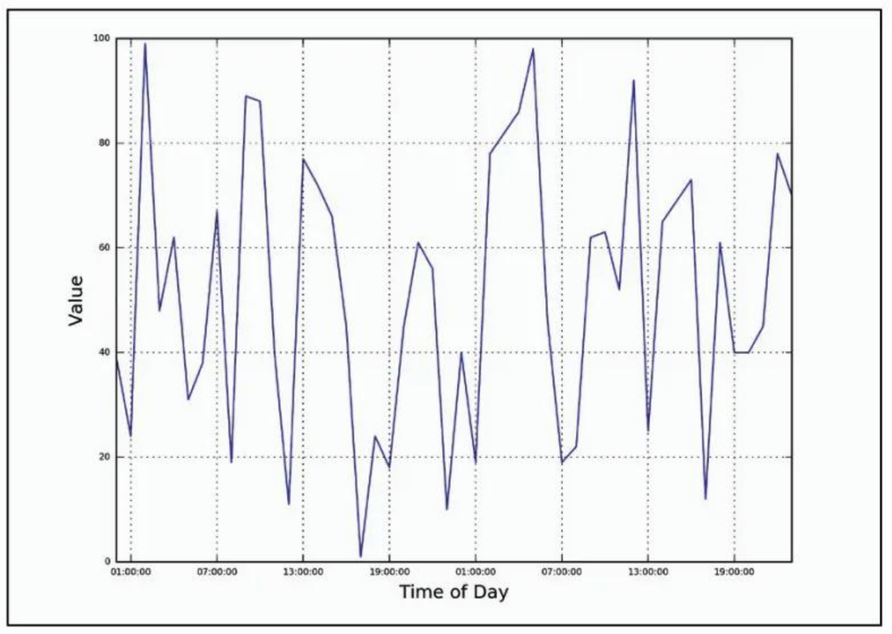
What the Python 3 code does is find the last week of a month using the Calendar module and then the last week of a month. After that, you just print the desired days of the week you want (0 is Monday). Once again, we are using the Calendar module, which, as it turns out, can do many things!

Plotting time series

Most of the times you'll get your data from a text file; as a result the date and the time will be in plain text format, which you will need to convert to datetime format. The format of the data file that you are going to plot is the following:

```
$ head -1 timeSeriesData.txt
00:00 2016-09-27                39
```

The total number of samples is 48, which is one sample per hour for two days. You can use as many samples as you want because Python and Matplotlib can handle a large number of data. There exist two columns in the datafile, which are separated by a tab character: the left column contains the date and the time, the right column is the value you want to plot.



› Figure 5: This is the output of the timeSeries.py script that shows how to plot time series data using Matplotlib

The **timeSeries.py** script will need to read both columns and create one list for each column. The two lists will be given as input to the Matplotlib `plot()` function. The Python 3 code of **timeSeries.py** is:

```
#!/usr/bin/python3
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import datetime
import numpy as np
import sys

if len(sys.argv) >= 2:
        filename = str(sys.argv[1])
else:
        print('Not enough arguments!')
        sys.exit(0)

x = np.empty((0))
y = np.empty((0))
# Read text file
try:
   f = open(filename, 'r')
except IOError as e:
   errno, strerror = e.args
   print("I/O error({0}): {1}".format(errno,strerror))

for line in f:
   newLine = line.rstrip().split('\t')
   dateTimeObject = datetime.datetime.
strptime(newLine[0], "%H:%M %Y-%m-%d")
   x = np.append(x, dateTimeObject)
   y = np.append(y, newLine[1])

plt.xlabel('Time of Day')
plt.ylabel('Value')
plt.grid(True)
plt.tick_params(axis='both', which='major', labelsize=6)
plt.plot(x,y)
plt.savefig('./timeSeries.png', dpi=300)
plt.show()
plt.close()
```

In **timeSeries.py** we've used Matplotlib to plot the data but you could also use software such as R or *Octave* to do the same. Figure 5 (above) shows the results file generated by **timeSeries.py**. Now go experiment! ■

Quick tip

Bitwise Operations

Python 3 supports bitwise operations like shift left and shift right, bitwise OR and bitwise AND. However bitwise operators are more appropriate for programming languages such as C so if you do lots of them, maybe you should write that part of your code in C itself!

NumPy & SciPy: For science!

Start using NumPy, SciPy and other Python 3 packages related to mathematics and plotting so you can become a data scientist!

While we're looking at NumPy and SciPy we will be using the plotting capabilities of the Matplotlib package. Both packages can do pretty advanced things; however, they can also perform basic mathematical operations. Although you can use *pip3* to install NumPy and SciPy, you will most likely find a ready to install package for your Linux distribution for both NumPy and SciPy, which is considered a better solution. Make sure that you install the appropriate package for the version of Python you are using: **python-scipy** for Python 2 and **python3-scipy** for Python 3.

After this tutorial you'll be able to do many things with NumPy and SciPy, including creating plots, saving data to disk, working with arrays and matrices and generating random numbers.

Using NumPy

Let's start using NumPy within the Python 3 shell:

```
$ python3
Python 3.4.2 (default, Oct  8 2014, 10:45:20)
[GCC 4.9.1] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>> import numpy
>>> print(numpy.version.version)
1.8.2
```

So, you are using NumPy version 1.8.2, which is defined in the `numpy.version.version` variable of the NumPy module. You can perform simple array calculations with NumPy after defining two arrays named `a` and `b`:

```
>>> from numpy import *
>>> a = array([1, 2, 3, 4, 5])
>>> b = array([1, 0, -1, 0, -1])
>>> a * 4
array([ 4,  8, 12, 16, 20])
>>> 4 + a
array([5, 6, 7, 8, 9])
>>> sin(a)
array([ 0.84147098,  0.90929743,  0.14112001, -0.7568025 ,
-0.95892427])
```

The `sin(a)` command applies the `sin()` function to all the elements of the `a` array and shows the result. However, if you do not save the output using a variable, it will be lost. The `4 + a` command adds the number 4 to every element of the `a` array whereas `a * 4` multiplies each element with number 4. Doing the same calculations programmatically would require some for loops and many other Python commands. NumPy does the same thing faster using fewer commands. Let's store the existing arrays on disk to read them afterwards:

```
import numpy
anArray = numpy.array([ [1, 2, 3], [4, 5, 6] ])
print('Existing array:\n', anArray)
numpy.save('usingSave.txt', anArray)
numpy.savetxt('usingSavetxt.txt', anArray)
loadArray = numpy.load('usingSave.txt.npy')
loadArraytxt = numpy.loadtxt('usingSavetxt.txt')
print('Using load()\n', loadArray)
print('Using loadtxt()\n', loadArraytxt)
```

There exist two methods for saving an array on disk, named numpy.save() and numpy.savetxt(). The former uses a binary file format whereas the latter uses a text format to

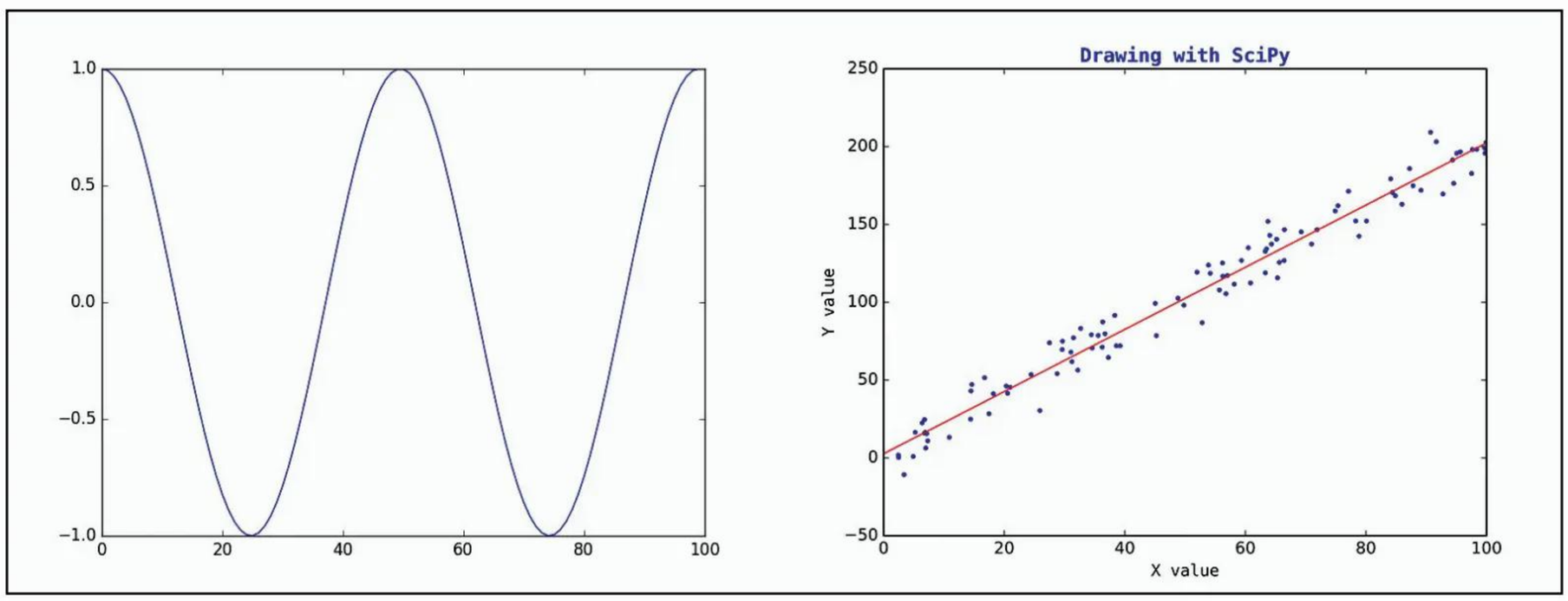


Figure 1: On the left is the output from simpleNum.py whereas on the right is the output from simpleSci.py. Both plots where made using Matplotlib

About NumPy

NumPy is the primary Python 3 package for performing scientific computing. NumPy brings two main things to Python: NDArray, which is a multi-dimensional array object and Universal Functions, which are very fast math operations between array objects. Although arrays are not everyday objects, they are extensively used in maths and in the computer graphics area. On a Debian system you can install NumPy for Python 3 as follows:

```
# apt-get install python3-numpy
```

After successfully installing NumPy, you can use it in your programs as follows:

```
import numpy
```

A major advantage of NumPy is that it can be integrated into Python programs, which means that you might get your data from a database, another program, an external file or an HTML page and process it using NumPy! NumPy is also needed by PyGame, a powerful Python package for creating games.

Figure 5 shows the installation process for both NumPy and SciPy. You can find more about NumPy at **http://www.numpy.org**

save the array. You can then read the generated disk files using numpy.load() and numpy.loadtxt(), respectively. Please have in mind that numpy.save() automatically attaches the **npy** extension to the provided filename.

Executing **saveArray.py** generates the following output:

```
$ python3 saveArray.py
Existing array:
 [[1 2 3]
 [4 5 6]]
Using load()
 [[1 2 3]
 [4 5 6]]
Using loadtxt()
 [[ 1.  2.  3.]
 [ 4.  5.  6.]]
$ ls -l usingSave*
-rw-r--r-- 1 mtsouk mtsouk 128 Aug 24 12:11 usingSave.txt.
npy
-rw-r--r-- 1 mtsouk mtsouk 150 Aug 24 12:11 usingSavetxt.
 txt
```

Please note that numpy.savetxt() automatically converts the integer elements of the array to floating point numbers:

```
$ cat usingSavetxt.txt
1.00000000000000000e+00 2.00000000000000000e+00
3.00000000000000000e+00
4.00000000000000000e+00 5.00000000000000000e+00
6.00000000000000000e+00
```

Using SciPy

Checking SciPy using the Python 3 shell:

```
$ python3
Python 3.4.2 (default, Oct  8 2014, 10:45:20)
[GCC 4.9.1] on linux
Type “help”, “copyright”, “credits” or “license” for more
information.
>>> import scipy
>>> print(scipy.version.version)
0.14.0
```

As you can see from the previous output, you are using SciPy version 0.14.0, which is stored in the `scipy.version.version` variable. Let us now see some of the advanced things that SciPy can do:

```
>>> from scipy import constants, special
>>> print(constants.c)
299792458.0
>>> print(special.gamma(1))
1.0
>>> print(special.gamma(-10))
inf
```

As you can see, although there is some functionality in the root of the SciPy package, most of methods that you are going to need are located in sub-packages, which you must import separately. The **scipy.special** package includes mathematical functions not found in the standard Python math package whereas the **scipy.constants** package contains constants used in Physics like the speed of light and the number of particles found in one mole of a substance.

Saving the output

Although working from the Python 3 shell is good, most of the times you would like to create a Python script that saves its graphical output to a file. As you will learn in a while, you will need the Matplotlib package for this job. This section will present one script that uses NumPy (**simpleNum.py**) and another that uses SciPy (**simpleSci.py**). **SimpleNum.py** is:

```
import math
import numpy
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt

values = numpy.linspace(-2*numpy.pi, 2*numpy.pi, 100)
plt.plot(numpy.cos(values))
plt.savefig('./numpyDraw.png')
plt.show()
plt.close()
```

It would be good to explain the way **simpleNum.py** works in case you want to make changes to it to match your needs. In this case, the Matplotlib `plot()` function takes an array as its argument and plots it. The array has as many points as the last parameter of the `numpy.linspace()` function.

There are many subtle points that can generate many kinds of error messages. You can see that **simpleNum.py** uses `numpy.cos()` instead of the `math.cos()` function offered by Python 3. This happens because functions that belong to the math package do not work nicely with NumPy arrays and generate the following kind of error message:

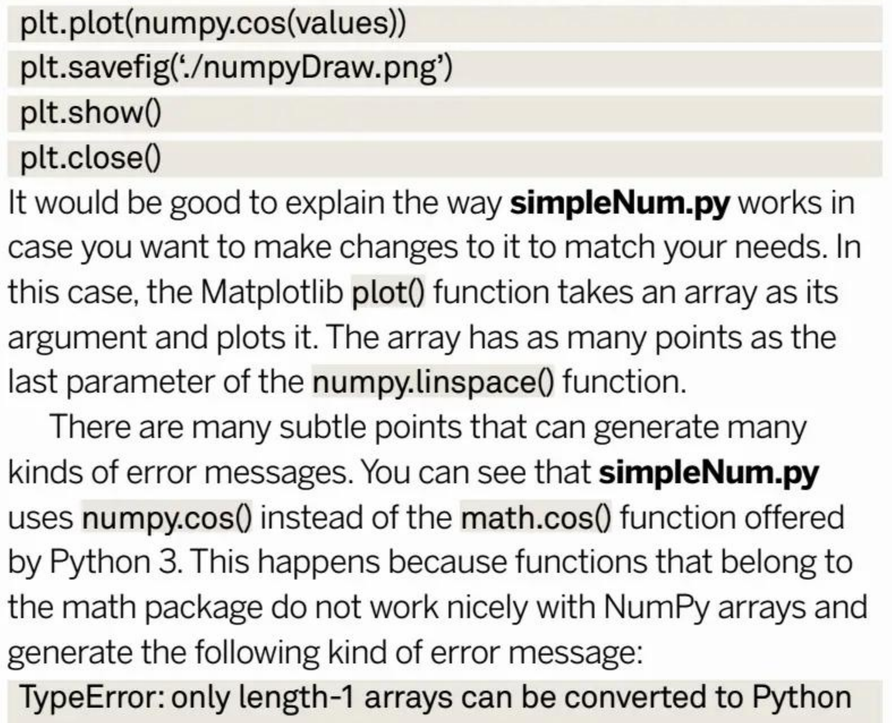
```
TypeError: only length-1 arrays can be converted to Python
scalars
```

The `matplotlib.use('Agg')` command forces Matplotlib to not »

Quick tip

Python 3 vs Python 2

As there exist many incompatibilities between the two Python versions, it is important that you install the version of a package that matches the version of Python you are using. Please keep in mind that this tutorial uses Python 3.



› **Figure 2: The code of plotFunction.py shows how to plot multiple functions at once using subplots**

» use any graphical backend, which is the desired behaviour when generating Python scripts that you want to execute remotely or as *cron* jobs. It is important for this command to go right after the `import matplotlib` command. The `values` variable is a NumPy array that holds the number of points that will be drawn on screen. The code of s**impleSci.py** is:

```
from scipy.optimize import curve_fit
import numpy
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
def line(x, a, b):
    return a * x + b
x = numpy.random.uniform(0., 100., 100)
y = 2. * x + 3. + numpy.random.normal(0., 10., 100)
plt.plot(x,y, '.')
plt.title('Drawing with SciPy', color = 'blue', family =
'monospace', fontweight='bold')
plt.xlabel('X value', family = 'monospace')
plt.ylabel('Y value', family = 'monospace')
error = numpy.repeat(10., 100)
popt, pcov = curve_fit(line, x, y, sigma=error)
print(popt)
xvalues = numpy.linspace(0., 100., 100)
plt.plot(xvalues, line(xvalues, popt[0], popt[1]), 'r-')
plt.savefig('scipyDraw.eps', format='eps', dpi=1000)
plt.show()
plt.close()
```

First, you have a sample that you got from the numpy.random.uniform() method. Then, you want to find the best line that fits your data with the help of the scipy.optimize.curve_fit() method. Last, you plot both the sample and the line to see how close they are.

Executing **simpleSci.py** will generate the following output:

```
$ python3 simpleSci.py
[ 1.99700629  2.71577275]
$ ls -l scipyDraw.eps
-rw-r--r-- 1 mtsouk mtsouk 28917 Aug 24 17:38 scipyDraw.
eps
```

The output is prettier now because you have titles, a grid on the output, some colour and different fonts. One other difference is that this time the output is in the EPS format and that is uses a much higher resolution. Apart from these changes, creating and saving graphics in both NumPy and SciPy is exactly the same!

As you can understand, even if you want to take advantage of the capabilities that SciPy offers, you still have to use some of the capabilities of NumPy. Additionally, if you need to create a plot, then the use of Matplotlib is required. Figure 1 (on the previous spread) shows the output of both **simpleNum.py** and **simpleSci.py**. On the left is the output from **simpleNum.py** and on the right the output from **simpleSci.py**.

Quick tip

MATLAB and Octave

SciPy offers the scipy.io package that contains functions that let you load MATLAB or *Octave* matrices and arrays of numbers or strings in a Python program and vice versa.

About SciPy

SciPy is built on top of NumPy and is more advanced than NumPy. It supports numerical integration, optimisations, audio processing, signal processing, image processing and statistics.

On a Debian system you can install SciPy for Python 3 as follows:

```
# apt-get install python3-scipy
```

You can use SciPy on your programs as follows but you should keep in mind that SciPy contains many sub-packages that you should explicitly import, which is to do with name space reuse:

```
import scipy
```

You can find more about SciPy at **http://www.scipy.org.**

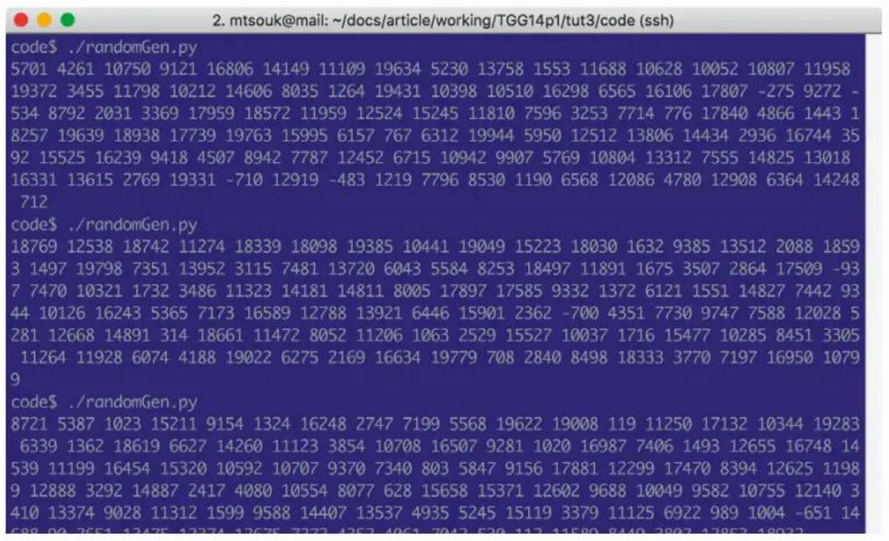


Figure 3: This Figure presents sample output from the randomGen.py script that generates random numbers

Plotting plots

As you already know, in order to plot a function you will need to install Matplotlib:

```
# pip3 install matplotlib
# python3
>>> import matplotlib
>>> matplotlib.__version__
'1.5.2'
```

There is no particular reason for using *pip3* instead of *apt-get* to install Matplotlib. Generally speaking, it might be wiser to install software using the packages offered by your Linux distribution. The only downside of this is that you might not get the latest version of a package, especially when you are using a Linux distribution that is targeted at server machines.

Although you saw how to plot a function in **simpleNum.py**, this time you are going to learn how to plot multiple functions at once. The code is as follows:

```
import numpy
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
x = numpy.linspace(-10, 10, 500)
y1 = x
y2 = x ** 2

y3 = x ** 3

y4 = x ** 4
f, axarr = plt.subplots(2, 2)
axarr[0, 0].plot(x, y1)
axarr[0, 0].axhline(0, color='red')
axarr[0, 0].axvline(0, color='red')
axarr[0, 0].set_title('[0,0]')
axarr[0, 1].plot(x, y2)
axarr[0, 1].axhline(0, color='red')
axarr[0, 1].axvline(0, color='red')
axarr[0, 1].set_title('[0,1]')
axarr[1, 0].plot(x, y3)
axarr[1, 0].set_title('[1,0]')
axarr[1, 0].axhline(0, color='red')
axarr[1, 0].axvline(0, color='red')
axarr[1, 1].plot(x, y4)
axarr[1, 1].set_title('[1,1]')
axarr[1, 1].axhline(0, color='red')
axarr[1, 1].axvline(0, color='red')
plt.suptitle('Plotting multiple functions!', color = 'blue',
family = 'monospace', fontweight='bold')
```

```
plt.savefig('multipleDraw.eps', format='eps', dpi=1000)
plt.show()
plt.close()
```

Executing **plotFunction.py** generates no output but creates a new file:

```
$ python3 plotFunction.py
$ ls -l multipleDraw.eps
-rw-r--r-- 1 mtsouk mtsouk 45138 Aug 24 18:48
multipleDraw.eps
```

The trick here is that you create a 2x2 array with subplots. Then, you manipulate each subplot individually and plot the function or functions you want. In order to set a common title, you will need to use the plt.suptitle() method. Last, you draw axis lines using axhline() and axvline() for the x axis and the y axis, respectively. Plotting two functions at the same subplot is as easy as the following code:

```
axarr[0, 1].plot(x, y1)
axarr[0, 1].plot(x, y2)
```

The first statement plots the (x, y1) pair of data whereas the second statement plots the (x, y2) pair of data on top of the first one. You can add as many pairs of data as you want. You can see the output of **plotFunction.py** in Figure 2 (*see the previous spread*) and you can find more information about Matplotlib at **http://matplotlib.org**.

You should also use Matplotlib for plotting time series data. Another tutorial in this book talks more about handling time and date data. What you should remember after reading this, is that the most important thing is to know the exact format of date and time, because without it you will have trouble plotting any time series data.

Doing random things

There are times when you need some sample data to test something. This is the perfect job for a random number generator. The next Python code, saved as **randomGen.py**, shows how to generate random numbers in Python:

```
#!/usr/bin/python3
import sys
import random
MIN = -1000
MAX = 20000
TOTAL = 100
for i in range(0, TOTAL):
   sys.stdout.write("%d " % random.randrange(MIN, MAX))
print()
```

The current version of **randomGen.py** generates 100 random numbers from -1000 to 20,000. If you call the name of the file **random.py** and you try to execute it, you will get an error message because there will be a conflict between your own filename and the actual random module. This happens because Python tries to find the methods of the random module inside your own file:

```
AttributeError: 'module' object has no attribute 'randrange'
```

So, in order to avoid similar problems, it is better to include uppercase letters in the filenames with the Python code.

The following code, executed inside the Python 3 shell, shows how to pick elements from a list at random:

```
>>> import random
>>> aList = [0, 1, -1, 2, -2, 3, 4, 5, 6, 7]
>>> random.choice(aList)
1
>>> random.sample(aList, 2)
[5, -1]
>>> random.shuffle(aList)
>>> aList
[7, 0, 1, 6, 4, -2, -1, 3, 5, 2]
```

The random.choice() method picks one element from the aList. This can be pretty handy when you want to find the winner of a lottery, pick a student from a class, examine an element from a given set, etc. The random.sample() method returns a new list containing N elements from the original list. Last, the random.shuffle() method changes the order of the items in aList. The NumPy package offers the numpy.random.choice() method that works similarly to the random.sample() method. Figure 3 (see left) shows a sample output from the **randomGen.py** program.

Arrays and matrices

There are many ways to create an array using NumPy; the simplest one is by using the array() function as follows:

```
>>> myArray = numpy.array([ [1,2,3], [4,5,6], [7,8,9] ]);
```

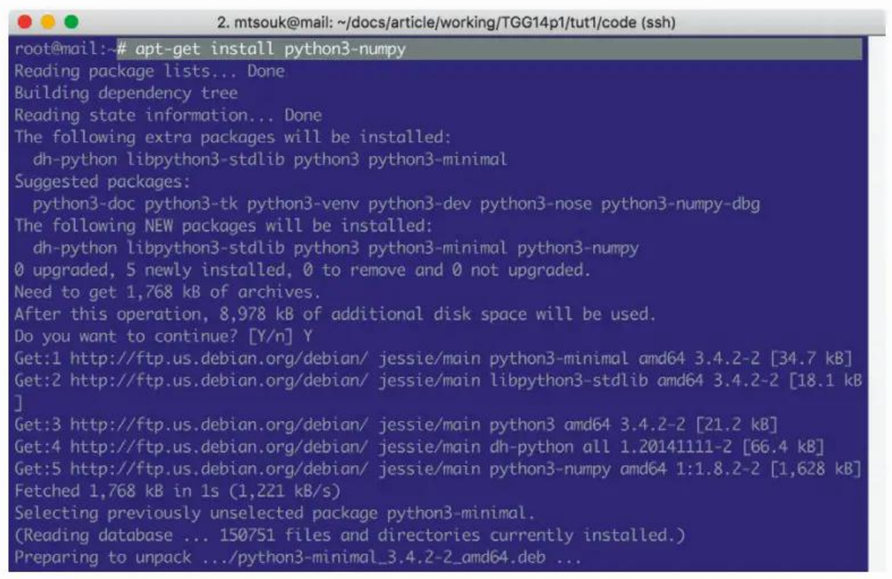
This creates a 3x3 array. You can find the maximum and minimum elements of myArray as easy as:

```
>>> myArray.min()
1
>>> myArray.max()
9
```

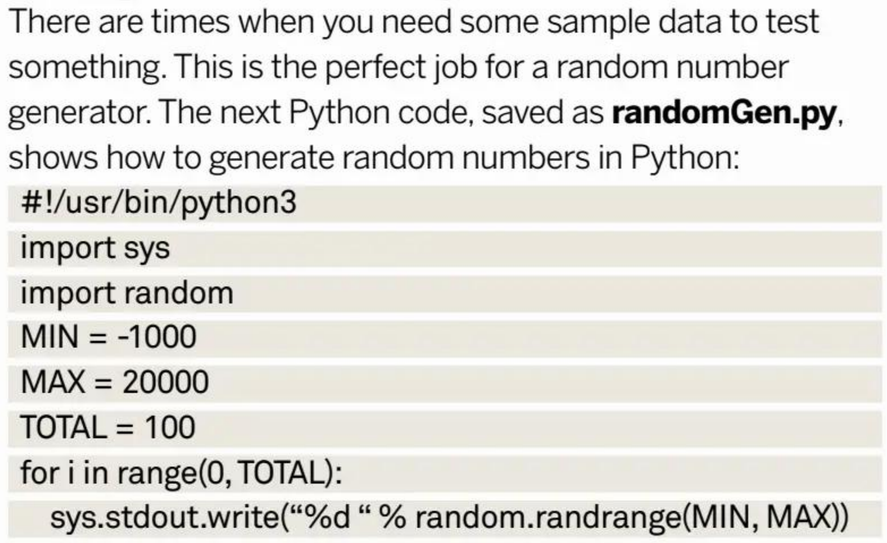
A special type of a two-dimensional NumPy array is called a matrix. A matrix looks like an array except that matrix multiplication follows the ordinary rules of linear algebra. You create matrices using the numpy.matrix function:

```
>>> m1 = numpy.matrix('1 2 3; -1 -2 -3; -1 0 1')
>>> m2 = numpy.matrix('0 2 3; -2 0  -3; -1 0 0')
>>> m1.shape
(3, 3)
```

Figure 4 shows the results of array and matrix calculations when executed from the Python 3 shell. If you still have questions that is understandable, it's part of the process of learning. The general idea is to start by mastering the basics first before going into more advanced topics. ■



› Figure 5: Installing NumPy and SciPy on a Debian Linux system is straightforward because Debian offers ready to install packages for both packages



› Figure 4: NumPy can perform calculations with matrices and arrays. You can declare an array using numpy.array() and a matrix using numpy.matrix()

Quick tip

Random numbers

Although the concept of generating random numbers looks simplistic, it is not. It is very difficult to generate truly random numbers, which is the reason that random number generation is a science with lots of theory behind it.

CODING ACADEMY

CODING ACADEMY

Databases

Manage your data efficiently and learn the ins and outs of SQL

58 **Python and SQLite3**
Get started using databases

62 **Get to grips with MariaDB**
The open source alternative to MySQL

66 **MongoDB: Using native drivers**
NoSQL mixed with Python and Ruby

70 **MongoDB: An admin's guide**
Milk maximum power from Mongo

74 **MongoDB: Build a blog**
Get your words collected in a database

78 **Riak NoSQL**
Just what is the big deal about NoSQL?

82 **Redis data store**
High-speed, in-memory NoSQL

SQLite3: Easy Python access

If you're working with serious data, you'll need a proper database. Here's how to use the SQLite3 database with the Python Programming Language

This article uses both Python 3.4.x and Python 3.5.x but it would be easy to make the presented code to work in the older Python 2.7.x version. The full path of the Python 3 executable used in this tutorial is either /usr/local/bin/python3 or /usr/bin/python3. If your Linux system uses a different one, change the scripts accordingly.

About SQLite3

SQLite has a couple of main advantages: it does not need a server process to run and it is a single file database, meaning that one single file has all the necessary database data. It also offers great performance, reliability, portability, simplicity, low memory usage, it can be embedded in many devices including mobile phones and it can, funnily enough, be accessed using the SQL language.

SQLite does not offer authentication or authorisation. The usual UNIX file permissions – defined by the **chmod(1)** command – are used in order to determine the three SQLite-supported access levels: read/write access, read access and no access. Also, SQLite is not suitable for very large datasets even though modern file systems support files with sizes bigger than a terabyte. Last, SQLite does not support replication – you can backup a database by simply copying the database file!

You can execute SQLite3 on your Linux machine by typing "sqlite3" on your shell prompt. If SQLite is not already installed, you might need to install it using your favourite package manager.

If at any time you need to learn more information about your active SQLite3 database, you can look at the **sqlite_master** table that holds the key information about your database tables. The first command shows the schema of **sqlite_master** whereas the second command displays the same information in a different format:

```
sqlite> .schema sqlite_master
CREATE TABLE sqlite_master (
  type text,
  name text,
  tbl_name text,
  rootpage integer,
  sql text
);
sqlite> PRAGMA TABLE_INFO(sqlite_master);
0|type|text|0||0
1|name|text|0||0
2|tbl_name|text|0||0
3|rootpage|integer|0||0
4|sql|text|0||0
```

The schema of a table defines the columns of a table and their attributes. The schema of a database is a way of visualising the tables and their relationships.

Connecting to a SQLite3 database

The following Python code presents an autonomous example that shows how you can connect to SQLite3 and execute a SELECT query:

```
#!/usr/local/bin/python3

import sqlite3
conn = sqlite3.connect('LXF.sqlite')
c = conn.execute("SELECT count(*) from sqlite_master")

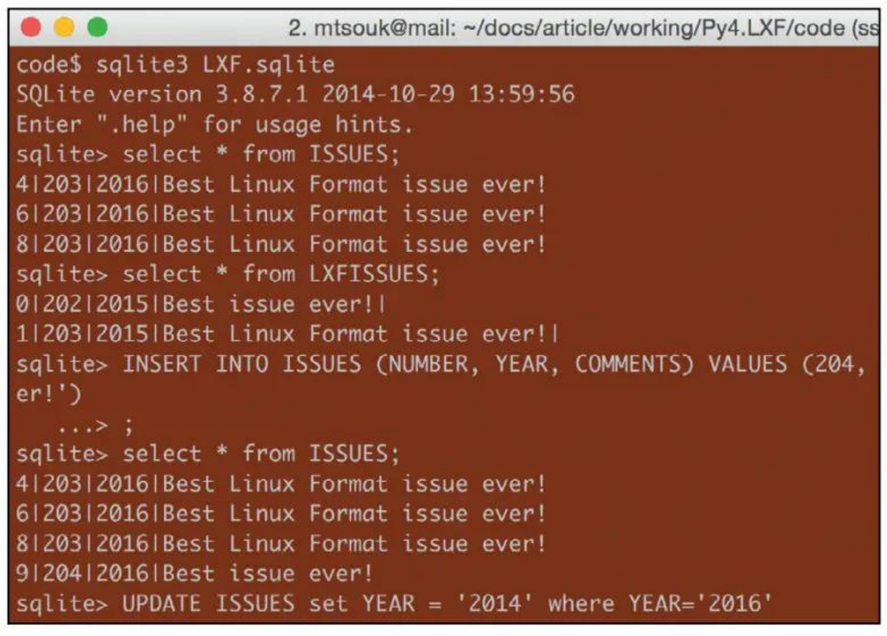
for row in c:
   print(row)

conn.close()
```

As you can see from the database query, you are accessing the **sqlite_master** table. The output you are going to get should look similar to the following:

```
$ ./connect.py
(0,)
```

The "sqlite3" module is used for communicating with a SQLite3 database. Then, you should define the database you want to connect to by specifying the filename of the database file. In order to execute SQL queries you will need to create a cursor – the presented query reads a database table and returns the number of records that can be found



› Figure 1: This figure shows examples of the basic CRUD operations as executed in the LXF.sqlite database

The supported data types

SQLite3 supports the INTEGER, REAL, TEXT, BLOB data types and the NULL value. INTEGER tholds integer values up to 8 bytes depending on the value. REAL holds floating point numbers using 8 bytes. The TEXT data type holds text strings which are stored using the database encoding – most of the times, the database encoding will be UTF-8. The Binary Large Object (BLOB) data type is used for storing binary data like images and compressed files. Last, the NULL value is used for representing missing values or empty cells. As there is no Boolean data type, you can use the INTEGER data type instead. Additionally, as there is no data type dedicated to storing dates and times, SQLite3 uses the INTEGER, REAL or TEXT data type to store such data depending on their representation. As an example, the UNIX time is stored using the INTEGER data type. The built-in functions used for date and time manipulation automatically convert between the various formats.

on the table. Please note that if the **LXF.sqlite** database file does not exist, it will be automatically created; therefore typographical errors might give you strange results. Additionally, if the database file you want to access is not in the current working directory, you should use its full path.

The aforementioned code will be extensively used in this tutorial so execute it before continuing with the rest of the tutorial to make sure that everything is working.

Additionally, the following output shows an interaction with the Python shell where the commands produce error messages:

```
>>> c = conn.execute("SELECT count(*) from sqlite")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
sqlite3.OperationalError: no such table: sqlite
>>> conn.close()
>>> c = conn.execute("SELECT count(*) from sqlite_master")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
sqlite3.ProgrammingError: Cannot operate on a closed database.
```

The first command shows that if the table you are trying to query does not exist, your program will fail. The last shows that if you accidentally close a database connection and try to access it again, your script will fail as well.

Creating a table

As the first action you are going to make is creating a new table, this section will show you various ways of creating tables in SQLite3:

```
$ sqlite3 LXF.sqlite
SQLite version 3.8.7.1 2014-10-29 13:59:56
Enter ".help" for usage hints.
sqlite> CREATE TABLE LXFISSUES (
   ID INT PRIMARY KEY     NOT NULL,
   NUMBER        INT     NOT NULL,
   YEAR          INT     NOT NULL,
   COMMENTS      CHAR(50)
);
sqlite> CREATE TABLE ISSUES (
   ID INTEGER PRIMARY KEY     AUTOINCREMENT,
   NUMBER        INT     NOT NULL,
   YEAR          INT     NOT NULL,
   COMMENTS      CHAR(50)
);
```

Creating a primary key is very important for performance reasons so choose it wisely. The AUTOINCREMENT keyword can also be very handy, especially when you want to create unique ID fields – as you can imagine, you do not have to provide a value to a field with the AUTOINCREMENT key because it is given by SQLite3. The use of AUTOINCREMENT is only allowed on an INTEGER PRIMARY KEY field.

The main different between LXFISSUES and ISSUES is that when you insert a record to LXFISSUES you will need to provide four values whereas the ISSUES table requires only three values because the ID field is automatically provided by SQLite3.

Another very handy SQL command is "ALTER TABLE" which allows you to add new columns to an existing database table:

```
sqlite> ALTER TABLE LXFISSUES ADD COLUMN EDITOR CHAR(100);
sqlite> .schema LXFISSUES
CREATE TABLE LXFISSUES (
   ID INT PRIMARY KEY     NOT NULL,
   NUMBER        INT     NOT NULL,
   YEAR          INT     NOT NULL,
   COMMENTS      CHAR(50),
   EDITOR CHAR(100));
```

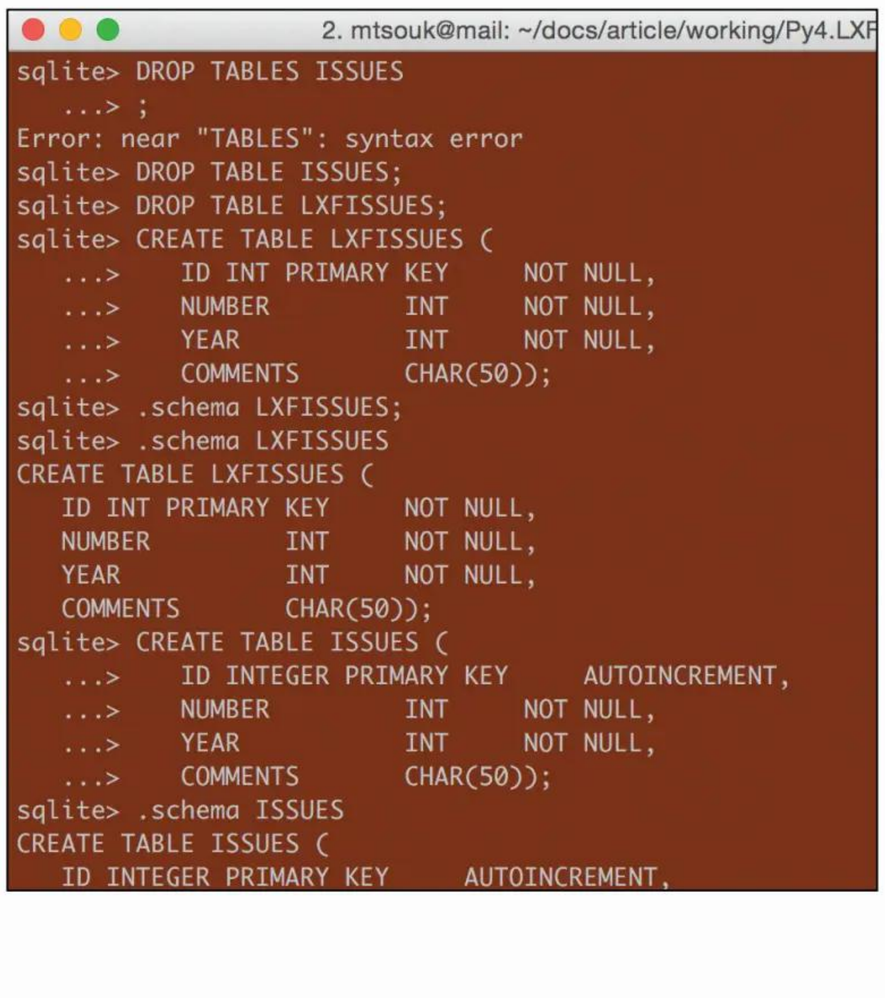
The **alter.py** script shows how to perform the same actions using Python and the **sqlite3** module:

```
conn.execute('''DROP TABLE LXFISSUES;''')
conn.execute('''DROP TABLE ISSUES;''')

conn.execute('''CREATE TABLE LXFISSUES (
   ID INT PRIMARY KEY     NOT NULL,
   NUMBER        INT     NOT NULL,
   YEAR          INT     NOT NULL,
   COMMENTS      CHAR(50));''')

conn.execute('''CREATE TABLE ISSUES (
   ID INTEGER PRIMARY KEY     AUTOINCREMENT,
   NUMBER        INT     NOT NULL,
   YEAR          INT     NOT NULL,
   COMMENTS      CHAR(50));''')

conn.execute('''ALTER TABLE LXFISSUES ADD COLUMN
EDITOR CHAR(100);''')
```



› Figure 2: This figure shows how you can create a table in SQLite3. Additionally, it shows how to alter an existing table by adding a new field to it

» As you can see, you are using the same commands as before embedded in the **conn.execute()** function. The **execute()** function that is used for creating and altering a table does not require a cursor as it returns no output. It is a common practice to use the DROP command to delete a table before trying to create it; however, if the table does not exist, the DROP command will fail. Figure 2 (pictured on the previous page) shows various operations related to tables as executed inside the SQLite3 shell.

Using the Python driver with SQLite3

Now that you know how to both connect to a SQLite3 database and create a table using Python, it is time to learn how to perform the other basic operations – insert, select, update and delete – with the help of the **sqlite3** module. As you saw in **connect.py**, in order to read the results of a SELECT query, you will have to iterate over your cursor.

The Python code of **insert.py** is the following:

```
conn.execute("INSERT INTO ISSUES (NUMBER, YEAR, COMMENTS) VALUES (202, 2015, 'Best issue ever!') ")
conn.execute("INSERT INTO ISSUES (NUMBER, YEAR, COMMENTS) VALUES (203, 2015, 'Best Linux Format issue ever!') ")
conn.commit()
conn.execute("INSERT INTO LXFISSUES (ID, NUMBER, YEAR, COMMENTS) VALUES (0, 202, 2015, 'Best issue ever!') ")
conn.execute("INSERT INTO LXFISSUES (ID, NUMBER, YEAR, COMMENTS) VALUES (1, 203, 2015, 'Best Linux Format issue ever!') ")
conn.commit()
```

Trying to run **insert.py** more than once will generate an error message because of the table constraints; you are not allowed to insert a record with an ID field that already exists, because the ID field is the primary key and must be unique. This is not true for the ISSUES table because of the AUTOINCREMENT field. The **conn.commit()** command is required for actually writing the records to the database.

The Python code of **fetch.py**, which showcases the use of the SELECT command, is the following:

```
c = conn.execute("SELECT * from ISSUES;")
for record in c:
    print(record)
```

As you can imagine, you cannot be sure about the number of records a SELECT statement returns; therefore, you will need to iterate over a cursor to get all results from a SELECT statement.

The Python code of **delete.py** is the following:

```
conn.execute("DELETE FROM ISSUES where ID='2'")
conn.commit()
print ("Total number of rows deleted :", conn.total_changes)
```

Once again, the use of the **conn.commit()** command in compulsory. If the DELETE command does not match any records, you are going to get no warnings or error messages. However, you can use the **conn.total_changes** property to find out what happened.

In the code that follows, saved as **update.py**, you will see how to perform SQLite3 updates using Python:

```
conn.execute("UPDATE ISSUES set YEAR = '2016' where YEAR='2015'")
conn.commit()
print ("Total number of rows updated :", conn.total_changes)
```

Figure 3 (pictured below) shows the four Python scripts in action. The following Python code, named **tableNames.py**, shows how you can retrieve the names of the columns of a given table:

```
import sys

# First argument is database name
database = sys.argv[1]
# Second argument is table name
tableName = sys.argv[2]

conn = sqlite3.connect(database)
c = conn.cursor()
query = "select * from " + tableName + " where 1=0"
c.execute(query)
# This will be empty because of the query
rs = c.fetchall()
field_names = [r[0] for r in c.description]

for f in field_names:
    print("*", f)
```

Keep this code in mind – you are going to use similar code in the GUI application you are going to develop.

The name of the table you want to get its column names is given as a command line argument. Executing **tableNames.py** produces the following kind of output:

```
$ ./tableNames.py LXF.sqlite ISSUES
* ID
* NUMBER
* YEAR
```

```
rMacBook:code mtsouk$ ./alter.py
rMacBook:code mtsouk$ ./insert.py
rMacBook:code mtsouk$ ./insert.py
Traceback (most recent call last):
  File "./insert.py", line 10, in <module>
    conn.execute("INSERT INTO LXFISSUES (ID, NUMBER, YEAR, COMMENTS) VALUES (0, 202,
2015, 'Best issue ever!') ")
sqlite3.IntegrityError: UNIQUE constraint failed: LXFISSUES.ID
rMacBook:code mtsouk$ ./fetch.py
(1, 202, 2015, 'Best issue ever!')
(2, 203, 2015, 'Best Linux Format issue ever!')
(3, 202, 2015, 'Best issue ever!')
(4, 203, 2015, 'Best Linux Format issue ever!')
rMacBook:code mtsouk$ ./update.py
Total number of rows updated : 4
rMacBook:code mtsouk$ ./fetch.py
(1, 202, 2016, 'Best issue ever!')
(2, 203, 2016, 'Best Linux Format issue ever!')
(3, 202, 2016, 'Best issue ever!')
(4, 203, 2016, 'Best Linux Format issue ever!')
rMacBook:code mtsouk$ ./delete.py
Total number of rows deleted : 2
rMacBook:code mtsouk$ ./fetch.py
```

› Figure 3: This figure shows the various Python scripts developed for this tutorial in action. Some of them might generate error messages when executed multiple times

The SQL Language

CRUD stands for Create, Read, Update and Delete which are the basic operations that can be performed on a database such as SQLite3. In order to be able to follow this tutorial, it would be very helpful to have the basic CRUD commands in mind so that you can check what the Python scripts do.The following SQL commands illustrate the four basic operations:

```
sqlite> INSERT INTO ISSUES (NUMBER, YEAR, COMMENTS) VALUES (204, 2016, 'Best issue ever!');
sqlite> SELECT * FROM ISSUES;
sqlite> UPDATE ISSUES set YEAR = '2014' where YEAR='2016';
sqlite> DELETE FROM ISSUES where ID='8';
```

Figure 1 shows the output from the previous operations using an example database.

To find all tables of a SQLite3 database, you should execute ".tables" from the SQLite3 shell which, all being well, produces output that should look a little something like this...

```
sqlite> .tables
ISSUES    LXFISSUES
```

```
* COMMENTS
```

About Tkinter

The presented application will need a menu to display the available tables of a database. A rational choice for this will be the OptionMenu class that creates a popup menu and a button to display it. You put the list of available tables into a dictionary data structure and you pass it as a parameter to the OptionMenu class. To get the current value of an OptionMenu you can use the **get()** method on the variable you used when you defined the OptionMenu class.

For displaying the required text, the handy ScrolledText class will be used. You should use the **delete()** method to delete the current data of the ScrolledText class and the **insert()** method to print the new data.

As you can understand, the logic behind the application is pretty simple yet very effective.

About the GUI

The presented application will read a database file and get all table names from it. Afterwards, it will allow you to select a table from a list in order to present you with the fields of the selected table. The database file will be given as a command line argument to the Python script. The initial version of the application is saved as **gui.py**:

```
#!/usr/bin/python3

import sqlite3
import tkinter
from tkinter import *
import tkinter.scrolledtext as ST
import sys

def callBack():
   print ("value is", variable.get())
   text.delete('1.0', END)
   myText = "Table name: " + variable.get()
   text.insert('insert', myText)

# First argument is database name
database = sys.argv[1]

conn = sqlite3.connect(database)
c = conn.cursor()

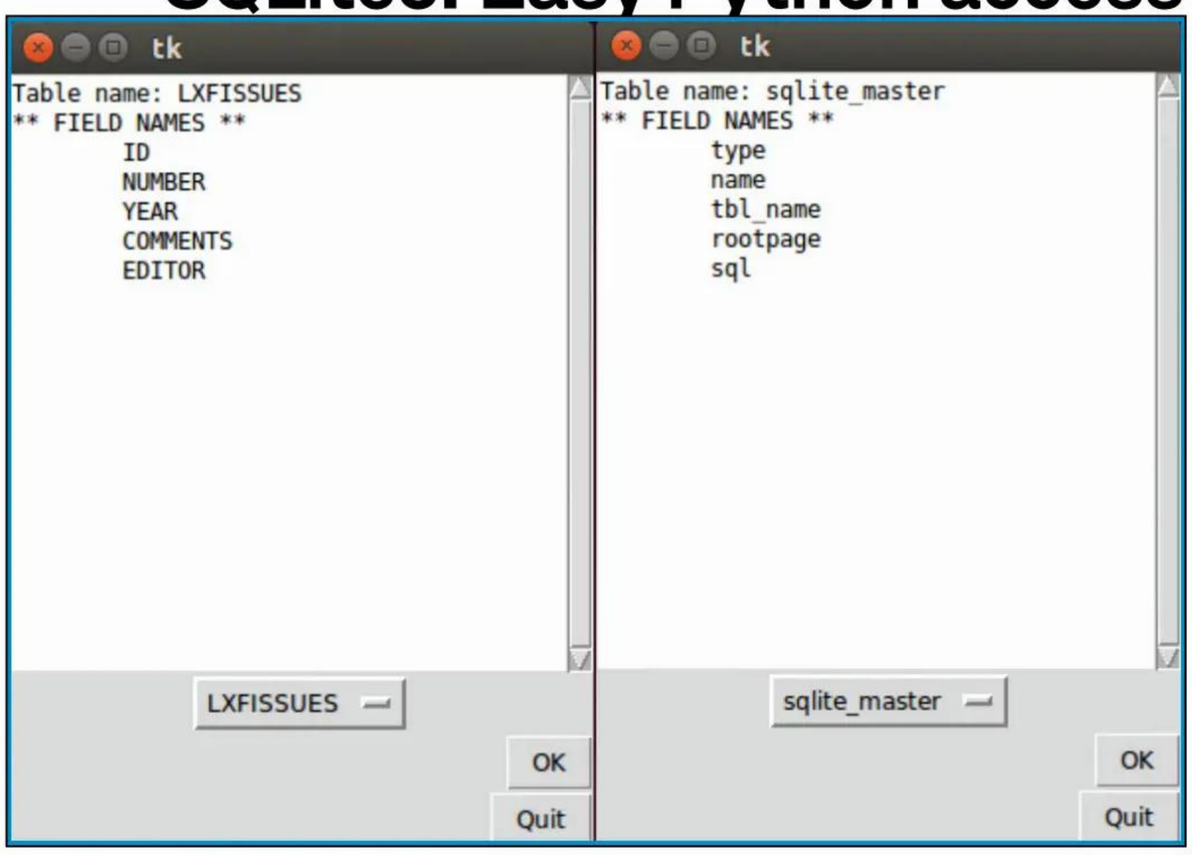
root=Tk()
canvas=Canvas(root,width=810,height=600)
canvas.pack()

text = ST.ScrolledText(canvas, width=35, height=20,
borderwidth=0)
text.pack()

# This is a dummy list of tables
listOfTables = {"sqlite_master",
          "one",
          "two",
          "three"}

variable = StringVar(root)
variable.set("sqlite_master") # default value
w = OptionMenu(root, variable, *listOfTables)
w.pack()
Button(root, text='Quit', command=root.quit).
pack(side=BOTTOM, anchor=SE)
Button(root, text='OK', command=callBack).pack(side=TOP,
anchor=SE)

mainloop()
conn.close()
```



› **Figure 4: This Figure shows the actual application in action using the LXF.sqlite database**

The final version of the application

The working version of the application is named **finalGUI.py**.The following code shows the implementation of the **callBack()** function because this is where the real action happens:

```
def callBack():
   print ("value is", variable.get())
   text.delete('1.0', END)
   tableName = variable.get()
   myText = "Table name: " + tableName
   text.insert('insert', myText + '\n')
   text.insert('insert', "** FIELD NAMES **\n")
   # Get the desired data!
   query = "select * from " + tableName + " where 1=0"
   c.execute(query)
   # This will be empty because of the query
   rs = c.fetchall()
   field_names = [r[0] for r in c.description]
   for f in field_names:
      text.insert('insert', '\t' + f + '\n')
```

In order to access the table names of a SQLite3 database from the **sqlite_maste**r table you should enter the following command:

```
SELECT name FROM sqlite_master WHERE type='table';
```

Once you've done that, the following Python code shows how the **listOfTables** variable is populated using the previous SELECT command:

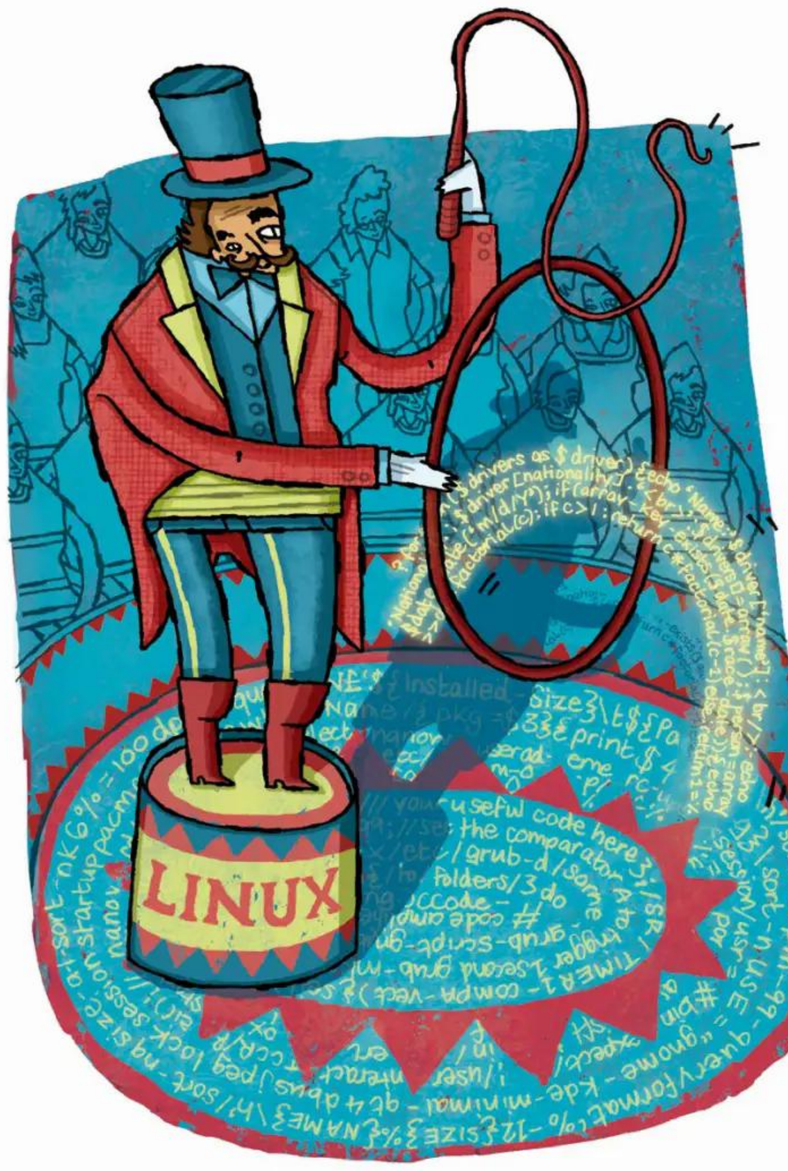
```
listOfTables = {}
listOfTables['sqlite_master'] = 0

c.execute("SELECT name FROM sqlite_master WHERE
type='table';")
for record in c:
   # print(record[0])
   listOfTables[record[0]] = 0
```

Figure 4 (at the top of the page) shows the **finalGUI.py** script in action. If you want to take this further you can learn more information about the sqlite3 module at **https://docs.python.org/3/library/sqlite3.html**. The official SQLite3 site is **https://www.sqlite.org/**. ■

MariaDB: How to get started

We turn the tables on Oracle's MySQL by dabbling in the wholesome open-source fork that is MariaDB



The *MySQL* RDBMS (Relational database management system) first appeared in 1995, and its then-custodian, Swedish firm MySQL AB, became one of the earliest examples of a successful open source company. While the database was entirely available under the GPL, clients unwilling to pander to its clauses (ie those who wanted to implement *MySQL* databases in closed-source products) could acquire the software under a monetary licence. In 2008, MySQL AB was acquired by Sun Microsystems, giving them a competitor to arch-rival Oracle's database. However, two years after, Sun was acquired by Oracle, so that *MySQL* was now, in a sense, in the hands of the enemy. As one may imagine, consternation ensued in the *MySQL* community, the main concern being that, under Oracle's stewardship, *MySQL* would become a closed product. The very day that the takeover was announced, Michael Widenius (former CTO of MySQL AB) and other developers forked *MySQL*, and the result was *MariaDB*.

Quick tip

For experimenting with compatibility between different versions of MySQL/MariaDB, checkout the handy MySQL Sandbox: **http://mysqlsandbox.net**

Since taking the helm Oracle has certainly added some closed source extensions to *MySQL*, but it also continues to develop the Community Edition alongside the Enterprise offering. Many in the industry, Canonical in particular, are happy with the attention Oracle continues to give to *MySQL*. In a *ZD-Net* interview, Mark Shuttleworth (Founder of Canonical) dismissed the surrounding exodus as a symptom of the "tendency to imagine conspiracies" on the part of certain players in the open source community.

While *MariaDB* is nowhere near as popular as its rival Oracle (the world's most used RDBMS), its parent *MySQL*, or Microsoft's SQL Server (numbers 2 and 3 on the podium), it is certainly gaining ground. A ranking on **db-engines.com** puts it at #21 as of April 2016. Being (almost) a drop-in replacement for *MySQL*, but at the same time offering improved performance (and a more wholesome provenance), means that obstacles to adoption are low. Many distros have already adopted *MariaDB* as their default *MySQL* implementation, among the first to do so were Arch and OpenSUSE 12.3 (March 2013), followed by Fedora 19 (July 2013), now joined by Mageia, Chakra and RHEL 7. Most other distros, while still having packages for Oracle's *MySQL* Community Server, will also have *MariaDB* packages that you can use instead.

This Strange Engine

The guts of any RDBMS – the parts that do all the creating, reading, updating and deleting of data – is known as the database engine. Older versions of *MySQL* used the MyISAM engine, but this was later replaced with InnoDB. Up until version 10.0.9 *MariaDB* also used InnoDB, but it has since replaced it with a fork called XtraDB, which is a drop-in replacement for InnoDB with extra features. XtraDB is maintained by Percona, who maintain Percona Server, which is, confusingly, another drop-in replacement for *MySQL*.

XtraDB takes greater advantage of modern hardware, offers better scalability, is highly configurable and more memory-efficient. What's not to like? Of course, there are many other database engines supported by *MariaDB* and many places where it's technically superior to its parent. You can read about them on the *MariaDB* website (see **http://bit.ly/MariadbVSMySQL**) if you want to find out more about these features.

MariaDB: How to get started

The first stable version of *MariaDB* to be released, in February 2010, was numbered 5.1.42, in an apparent defiance of convention. The rationale here is that it was based on the then current 5.1.42 version of *MySQL*. Version 5.5, was likewise based on *MySQL 5.5*. However, since the current stable branch of *MariaDB* evolved independently of *MySQL 5.6*, it's numbered differently – they chose a nice round 10.0 for this series. To be clear, this is not to say that *MariaDB 10.0* isn't compatible with *MySQL 5.6* as that would be almost entirely untrue. Rather, many of the extra features and bugfixes introduced by Oracle in 5.6 had already been implemented by the community, and the extensive refactoring of code undertaken by Oracle was deemed unnecessary to replicate. *MariaDB* is its own entity, and has added many of its own features and revamped many under-the-hood components, some of which we'll explore here. However, *MySQL* is also its own thing and while new features added to it will be reflected in *MariaDB* where appropriate, this will take time and so there are edge-cases where the two are incompatible, and this gap may widen in the future.

That said, this is an introductory tutorial and so most of it will work just fine in both the *Maria* and *My* databases (these are the names of Widenius' daughters incidentally, he also has a son called Max, and another database called *MaxDB*). Actually it'll mostly work for any SQL database.

The box [*next page*] gives some hopefully distro-agnostic instructions for installing *MariaDB*, but do check your own distro's documentation in case some new-fangled install method has been concocted since this was written. Once you've got it installed, the next thing is to set up the database server, a helpful script is provided for this purpose. It needs to be run as root:

```
# mysql_secure_installation
```

This script (which might be run automatically on install, in which case there's no need to run it again) will prompt for the current *MySQL* root password (not your system's root password) which will be blank if you've just done a clean install. You will then be prompted to set a new root password; offered the opportunity to delete anonymous users and test databases, and restrict root access to local logins. All of which you should do, unless you have some reason not to.

We can start the *MariaDB* shell as the root user with:

```
$ mysql -u root -p
```

The **-p** means that we will be prompted for the password. Once we're in, you should see the *MariaDB* prompt **MariaDB [(none)]>**. SQL (Structured Query Language) is a fairly easy to follow language; don't forget the semicolons at the end of every command though. We can see any databases currently in service with a simple:

```
SHOW DATABASES;
```

While the uppercase is optional, and hard on the eyes, you should still use it for SQL commands as it helps to differentiate reserved words from database names and the like. You'll see three databases here, one called **mysql** which is to do with *MariaDB*'s internals and not for you to touch, as well as **information_schema** and **performance_schema** which hold metadata about databases and can be used to diagnose issues. We can create a new database called **lxfdata** and then connect to it with:

```
CREATE DATABASE lxfdata;
USE lxfdata;
```

After the first command, you'll see a friendly acknowledgement that everything went OK:

```
Query OK, 1 row affected (0.00 sec)
```

And note that the prompt changes to **MariaDB [(lxfdata)]>** after the second. You can also tack the database name onto the end of the **mysql** shell command to dive straight into it.

Inside the database, data is stored in tables, but we don't have any yet. So let's rectify this by creating an example table called linuxes, which will house some data about various Linux distros. Tables are formed of columns, and each column has an associated data type. SQL queries will span newlines so you can format them nicely.

```
CREATE TABLE linuxes (
    id int(5) NOT NULL AUTO_INCREMENT,
    name varchar(32) DEFAULT NULL,
    current_version varchar(32) DEFAULT NULL,
    easy bool DEFAULT NULL,
    PRIMARY KEY(id)
);
```

We've set up four columns, so here's an overview of how we'll use them:

- **id** An id number, this will automatically incremented for each entry.
- **name** The name of the distro, a 32 character string, defaults to the value NULL if not stated.
- **current_version** The distro's version number, or codename. Same datatype as **name**.
- **easy** Whether or not the distro is suitable for beginners. A boolean (true or false) value.

Quick tip

One of the big names that have switched from *MySQL* to *MariaDB* is Wikipedia. You can read all about its transition here **http://bit.ly/WikipediaAdoptsMariaDB**.

```
Copyright (c) 2000, 2014, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [lxfdata]> SHOW COLUMNS FROM linuxes;
+-----------------+-------------+------+-----+---------+----------------+
| Field           | Type        | Null | Key | Default | Extra          |
+-----------------+-------------+------+-----+---------+----------------+
| id              | int(5)      | NO   | PRI | NULL    | auto_increment |
| name            | varchar(32) | YES  |     | NULL    |                |
| current_version | varchar(32) | YES  |     | NULL    |                |
| easy            | tinyint(1)  | YES  |     | NULL    |                |
+-----------------+-------------+------+-----+---------+----------------+
4 rows in set (0.00 sec)

MariaDB [lxfdata]>
```

The columns of our linuxes table; you can extend this to include all kinds of data – even, using the blob types, binary data, like a logo image for each distro

» We'll use **id** as a primary key, which is a canonical index for the table. Each table must have at least one primary key and said keys must be unique. You can quit the *MariaDB* shell with **quit** or Ctrl-D.

The screenshot (pictured below) shows a summary of our empty database. Booleans are referred to as **tinyints**, since they are stored as the integers 1 or 0. You can read all about the manifold datatypes in *MariaDB* at **http://bit.ly/MariaDBDataTypes** . For example, the **blob** datatype stores up to (but not including) 64K of binary data, and has its counterparts **tinyblob**, for storing up to 256 bytes, as well as **mediumblob** (16MB) and **longblob** (4GB).

Don't hurt yourself

Just like the situation for the rest of Linux, it's generally not a good idea to use the **root** user except when necessary. This is especially true if your database is connected to a web application – if this connects as the root user then all your databases could be at risk if that application becomes compromised. So lets create a less privileged user.

First reconnect to our **lxfdata** database:

```
$ mysql -u root lxfdata -p
```

We'll call our user **lxfuser** and give them a weak password:

```
CREATE USER 'lxfuser'@'localhost' IDENTIFIED BY
'password1';
```

The **@ 'localhost'** syntax means this user can only log in locally. You can also specify a hostname here, or use **@'%'** to enable users to login from anywhere else. For instance, if we want to allow **lxfuser** to add rows to our table, and select it:

```
GRANT INSERT, SELECT on lxfdata.linuxes TO
'lxfuser'@'localhost';
```

› **We empowered our user and all they did was mess with our data. Tsk, tsk**

```
File  Actions  Edit  View  Help
Shell No. 1
MariaDB [lxfdata]> UPDATE linuxes SET easy=1 WHERE name='Gentoo';
ERROR 1142 (42000): UPDATE command denied to user 'lxfuser'@'localhost' for table 'linuxe
MariaDB [lxfdata]> Bye
[jonni@jbmachine ~]$ mysql -u root lxfdata -p
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 5
Server version: 10.0.16-MariaDB-log MariaDB Server

Copyright (c) 2000, 2014, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [lxfdata]> GRANT UPDATE, DELETE on lxfdata.linuxes TO 'lxfuser'@'localhost';
Query OK, 0 rows affected (0.00 sec)

MariaDB [lxfdata]> Bye
[jonni@jbmachine ~]$ mysql -u lxfuser lxfdata -p
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 6
Server version: 10.0.16-MariaDB-log MariaDB Server

Copyright (c) 2000, 2014, Oracle, MariaDB Corporation Ab and others.
```

Now we can connect to our database as **lxfuser**, and start adding some data. So exit the shell and log in again with:

```
$ mysql -u lxfuser lxfdata -p
```

Each entry in our table (ie each distro) will occupy a row. Our first row will be, hey, why not, let's be all subversive: the DPRK's Red Star Linux (the OSX-ey Desktop version is now floating around the darker corners of the internets) **http://bit.ly/RedStar3Desktop**.

```
INSERT INTO linuxes (name, current_version, easy) VALUES
('Red Star Linux', '3.0', 0);
```

Red Star is definitely not easy if you don't read Korean. Note that we don't need to specify the **id** column since it auto-increments. Other columns which we don't specify in the first set of brackets will get the value **NULL**, or whatever other default value you specified above. Thus, for our next two favourite distros, which are rolling releases and hence have no version number, we can do:

```
INSERT INTO linuxes (name, easy) VALUES
  ('Arch Linux', 0),
  ('Gentoo', 0);
```

And we conclude this exercise in data entry with some more Linuxes we love:

```
INSERT INTO linuxes (name, current_version, easy) VALUES
  ('Ubuntu', '14.10 Utopic Unicorn', 1),
  ('Linux Mint', '17.1 Rebecca', 1),
  ('Debian', '7.8 Wheezy', 0),
  ('openSUSE', '13.2', 1),
  ('Fedora', '21', 0),
  ('Mageia', '4.1', 1),
  ('Elementary OS', '0.2 Luna', 1),
  ('Bodhi Linux', '3.0.0', 1);
```

We can view the fruits of our labour using the following SQL query:

```
SELECT * FROM linuxes;
```

As you can imagine, tables can become unwieldy and viewing them in their entirety like this will not always be possible, or pleasant. It's handy, then to restrict the rows and columns that are displayed. Suppose we are only interested in the names and version numbers of our distros. This query will show only those columns:

```
SELECT name, current_version FROM linuxes;
```

On the other hand, we might only be interested in 'easy' distributions, which we can display with:

```
SELECT * FROM linuxes WHERE easy = 1;
```

We can also specify a sort order here, rather than the order in which the data were entered. All we need do is add an **ORDER BY** directive, for example to sort the last query lexicographically by distribution name, do:

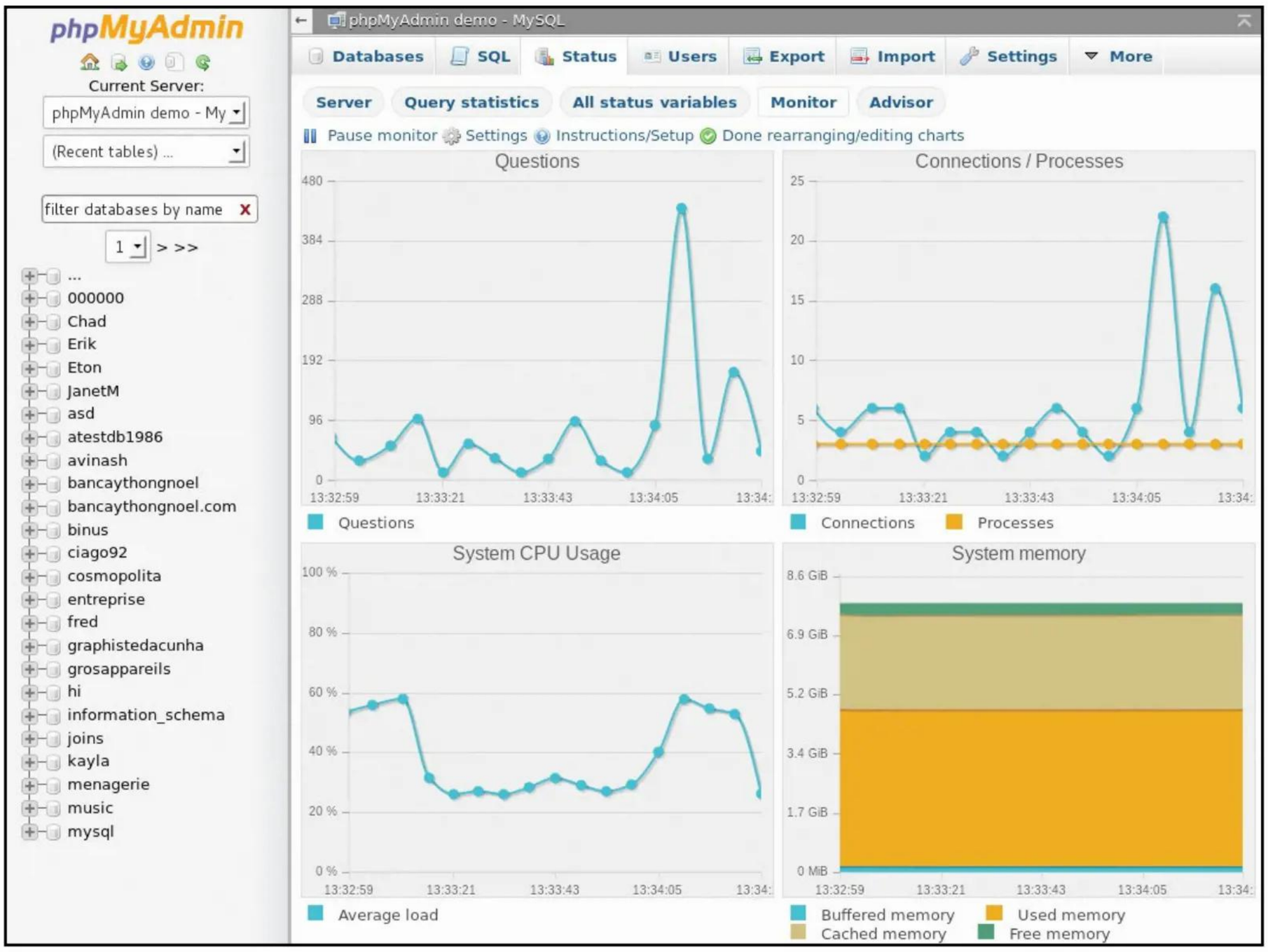
Installation differences

As mentioned elsewhere, a few of the distros including Fedora, Arch, openSUSE and some others have chosen *MariaDB* as their default RDBMS, meaning that there will be a **mariadb** package (and on Fedora a **mariadb-server** package as well) waiting in the repositories for you. Ubuntu has chosen to stick with *MySQL*, so the following code will add the MariaDB repo and install everything:

```
$ sudo apt-get install software-properties-
common
$ sudo apt-key adv --recv-keys --keyserver
hkp://
keyserver.ubuntu.com:80 $
0xcbcb082a1bb943db
$ sudo add-apt-repository 'deb http://mirrors.
coreix.net/mariadb/repo/10.0/ubuntu utopic
main'
$ sudo apt-get update
$ sudo apt-get install mariadb-server
```

This will also work for Linux Mint. For Debian the procedure is much the same, just replace **software-properties-common** with **python-software-properties** in the first command and change the repository in the third to something like **'deb http://mirrors.coreix.net/mariadb/repo/10.0/debian wheezy main'**. For other distributions and bleeding-edge releases from *MariaDB*'s own repositories, visit **https://downloads.mariadb.org/mariadb/repositories**.

Once installed the service will automatically start on some distros, and other distros, however, you will need to start it manually. For *Systemd* peeps, this will be done with the command **systemctl start mysqld** – *MariaDB* is so compatible that it didn't even change the name of the daemon here.



› ***PHPMyAdmin*** **is a popular tool for web-based database administration. It's also a popular target for hackers, so be careful**

```
SELECT * FROM linuxes WHERE easy = 1 ORDER BY name;
```

Queries can be nested, and can include regular expressions via the **REGEXP** operator.

We never granted our humble **lxfuser** the privileges required to change or delete rows, but we can do that if we first logout of the client shell and, while logged in as the root user, do:

```
GRANT UPDATE, DELETE on lxfdata.linuxes TO
'lxfuser'@'localhost';
```

Now if we log in to the client as **lxfuser** we can start messing with our data. We might, if we were to follow the advice of 4chan's ***/g/*** board, get the idea that Gentoo is easy.

We can then update the database to reflect our new opinion with:

```
UPDATE linuxes SET easy=1 WHERE name='Gentoo';
```

Deleting rows is straightforward, we could delete Ubuntu (not that we have anything against it, after all Canonical pay our wages, if you believe the whispers) with a simple:

```
DELETE FROM linuxes WHERE name = 'Ubuntu';
```

Columns can be deleted (technically **DROP**-ped) too, but this requires a different privilege, **ALTER**.

The second most destructive thing one can do with your database is DROP one or more tables, this requires DROP privileges:

```
DROP TABLE linuxes;
```

The most destructive thing that we can do is to callously delete whole databases:

```
DROP DATABASE lxfdata;
```

In some cases it's possible to (partially) recover accidentally dropped tables, but it's not something you can count on. Rather, you should keep regular backups, which we shall now explore.

Tux on

Nobody likes losing data, and databases are known for housing lots of it, so performing regular back ups is a good idea. We can use the **mysqldump** client to do this. Rather than saving the data as stored by the engine, **mysqldump** will, by default, output the SQL commands necessary to recreate the databases and tables, so that they can be imported by some other relational database management system if needs be. It can also output CSV or XML data.

To save our simple **linuxes** table, do:

```
$ mysqldump lxfdata linuxes -u root -p
```

You need to do this as *MariaDB*'s root user since tables are locked prior to dumping, and this requires privileges beyond those assigned to our humble **lxfuser**. Hopefully that works and you will see some messy looking SQL. Of course, it's more convenient to redirect this to a file, by adding something like **> linuxes.sql** to the command above.

You could then import the data like this:

```
$ mysql lxfdata < linuxes.sql
```

Note that this will replace, rather than add to, the extant **linuxes** table, but it will not recreate the whole **lxfdata** database in which it is housed. That functionality can be achieved by omitting the table name from the **mysqldump** command. Indeed, if you want to back up all the databases (except the internal information schema one) you can omit the database name too. For more details on **mysqldump** check **https://mariadb.com/kb/en/mariadb/mysqldump**.

And that concludes our primer on the wholesome SQL database that is *Maria*. Besides the basic SQL we've shown here, you can use it in conjunction with any number of popular applications, including *OwnCloud*, *Wordpress* and *Drupal*. So don't delay, snub Larry today. ■

MongoDB: Using native drivers

Let's jump into popular NoSQL database MongoDB with a guide to getting started using the Ruby and Python drivers

NoSQL databases are designed for the web and don't support joins, complex transactions and other features of the SQL language. *MongoDB* is an open source *NoSQL* database written in C++ by Dwight Merriman and Eliot Horowitz which has native drivers for many programming languages, including C, C++, Erlang, Haskell, Perl, PHP, Python, Ruby and Scala. In this tutorial, we'll cover the *MongoDB* drivers for Python and Ruby.

The *MongoDB* document format is based on JSON, and the JSON structures consist of key and value pairs and can nest arbitrarily deep. If you're not already familiar with JSON, you can think of JSON documents as dictionaries and hash maps that are supported by most programming languages.

The following instructions will help you install *MongoDB* on an Ubuntu Linux system:

```
$ sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10
$ echo "deb http://repo.mongodb.org/apt/ubuntu trusty/mongodb-org/3.0 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-3.0.list
$ sudo apt-get update
$ sudo apt-get install -y mongodb-org
```

The last command installs the latest *MongoDB* version which, at the time of writing this tutorial, is 3.0.7. On an Ubuntu
Linux system, you can install the Ruby interface to the *MongoDB* database in the following way (provided that Ruby is already installed):

```
$ sudo gem install mongo
```

Please make sure that you use `gem` to install the Ruby *MongoDB* driver because your Linux distribution (distro) might have an older driver version that uses different functions for connecting to the database.

You can install the Python driver by executing `sudo apt-get install python-pymongo`. If you're using Python 3 you should run `sudo apt-get install python3-pymongo` instead of the previous command. Alternatively, you can install the Python driver with the `sudo pip install pymongo` command provided that the *pip* utility is already installed.

You might need to execute the following JavaScript code from the *MongoDB* shell in order to insert sample data in your *MongoDB* database to experiment more while working your way through this tutorial:

```
> use LXF
switched to db LXF
> for (var i=0; i<10000; i++) { db.sampleData.insert({x:i, y:i/2}); }
WriteResult({ "nInserted" : 1 })
> db.sampleData.count();
10000
```

What the JavaScript code does is select the `LXF` database – if `LXF` doesn't already exist, it will be automatically created – and insert 10,000 documents in the `sampleData` collection of the `LXF` database. You are free to change the name of the database, which is defined in the `use LXF` command, and the collection, which is defined in the `db.sampleData.insert()` command; however, all presented Ruby and Python code uses the `LXF` database.

The `db.sampleData.count()` command verifies that the `sampleData` collection has indeed 10000 documents. Should you wish to delete the entire `sampleData` collection, you should execute the next command:

```
> db.sampleData.drop();
true
> db.sampleData.count();
0
```

All presented Python and Ruby examples are autonomous and will work without any changes, assuming, of course, that the appropriate collections and databases exist in your *MongoDB* installation.

Quick tip

MongoDB is schemaless, which means that two documents belonging to the same collection can have a different number of keys with the exception of the _id key.
This is very important when writing code for MongoDB because a misspelled collection name will create a new collection not an error message!

```
2. mtsouk@mtsouk-ubuntu: ~/code/mongoDB (ssh)
mongoDB$ cat connect.py
from pymongo import MongoClient
client = MongoClient()

client = MongoClient("127.0.0.1", 27017)

db = client.LXF

# Choose a collection
sampleData = db.sampleData

item = sampleData.find_one()
print item['_id']
print item['x']

mongoDB$ python connect.py
564d920ba5c76a7e7fd229c9
0.0
mongoDB$
```

› This Python script uses MongoClient() to specify the desired machine and the port number of the server

The Ruby driver

The Ruby *MongoDB* driver is written in Ruby and is officially supported by *MongoDB*. Although it can be used on its own, it is also used by object mapping libraries, such as Mongoid. The driver supports all *MongoDB* versions, including versions 3.0.x and 2.6. You can find the source code of the driver at **https://github.com/mongodb/mongo-ruby-driver**.

The following Ruby code (**connect.rb**, see **http://bit.ly/1RZp1WS**) checks whether you can connect to a *MongoDB*

GridFS and Ruby

If you want to find all GridFS files that are stored in a database, you can query the system table that holds this information. GridFS uses two system tables, one for holding the filenames and another that holds the actual binary data of a chunk. The default behaviour of GridFS is to use two collections with names prefixed by fs bucket: **fs.chunks** and **fs.files**.

The following Ruby code inserts both a existing binary file and a text file that's created on the fly as GridFS objects:

```
fs = $client.database.fs
$file = File.open("image.png")
$file_id = fs.upload_from_stream("image.png", $file)
$file.close

# To create a file with raw data and insert it
file = Mongo::Grid::File.new('I am a NEW file stored in GridFS', :filename => 'aFile.txt')
$client.database.fs.insert_one(file)
```

You should look at **storeGridFS.rb** in the source code archive (**http://bit.ly/1RZp1WS**) for more. The next section of Ruby code, which can be found in **retrieveGridFS.rb**, retrieves a previously inserted GridFS file using its file_id:

```
# Upload a text file
fs = $client.database.fs
$file = File.open("connect.rb")
$file_id = fs.upload_from_stream("connect.rb", $file)
$file.close

# Download a file
$file_to_write = File.open('perfectCopy', 'w')
fs.download_to_stream($file_id, $file_to_write)
```

As you can see from the Ruby code (above), the new copy of the GridFS file will be named **perfectCopy**. (The screenshot shows both Ruby examples, **storeGridFS.rb** and **retrieveGridFS.rb** in action).

Please note that the first program (**storeGridFS.rb**) blindly inserts two files, therefore if you run it multiple times, both files will be inserted multiple times .

You can only differentiate between the various copies of the same GridFS file using the **_id field.**

You will discover that the *MongoDB* documentation that shows how to use the Ruby driver to retrieve GridFS files from a MongoDB database is a little unclear, but you'll find that **retrieveGridFS.rb** is

server and prints the version of the Ruby driver:

```
require 'rubygems'
require 'mongo'
include Mongo

$client = Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'LXF')
Mongo::Logger.logger.level = ::Logger::ERROR
$collection = $client[:someData]
puts 'Connected with version:'
puts Mongo::VERSION
```

If you can successfully execute the code (*above*), then you are ready to continue with the rest of the tutorial. Otherwise, try to correct the errors before continuing. The generated output from **connect.rb** is the following, which means that you are using the 2.1.2 version of the Ruby *MongoDB* driver:

```
$ ruby connect.rb
D, [2015-11-19T10:28:57.085526 #2542] DEBUG -- : MONGODB | Adding 127.0.0.1:27017 to the cluster.
Connected with version:
2.1.2
```

The `Mongo::Client.new()` function specifies the IP address of the machine that runs *MongoDB* as well as the port number that *MongoDB* listens to – you can use a hostname instead of the IP address. The last parameter (`:database`) defines the name of the database you want to connect to. There are a number of other useful supported parameters, such as :user, :password, :connect_timeout, :replica_set, etc.

We've supplied a similar program (**connect.py**) in the source code archive (**http://bit.ly/1RZp1WS**) written in Python (see image), which uses the official Python *MongoDB* driver. The program connects to a *MongoDB* database, randomly reads a document from the sampleData collection of the LXF database and prints the _id and x fields of the document. As you'll see from the code supplied, both drivers work in an analogous way.

Insert, Update and Select Operations

Both **connect.rb** and **connect.py** will be used again and again in this tutorial because without a proper connection to *MongoDB*, you won't be able to perform any other operation so make sure that you understand them well, especially their various parameters and variables, before going any further.



› When used correctly, indexes can greatly improve the performance of your applications

The following code presents a complete example in Ruby (without the required code for connecting to the database), where you can insert multiple documents on the `someData` collection of a *MongoDB* database:

```
$collection = $client[:someData]
500.times do |n|
  doc = {
    :username => "LinuxFormat",
    :code => rand(4), # random value between 0 and 3, inclusive
    :time => Time.now.utc,
    :n => n*n
  }
  $collection.insert_one(doc)
end
```

The loop inserts 500 documents, using `n` as the iterator. The `insert_one()` function is used for writing a JSON document, »

Quick tip

For more info on the Ruby MongoDB driver head to **http://bit.ly/RubyMongoDB**. Similarly, there's info about the Python driver at **http://bit.ly/PythonMongoDB**.

which is generated using Ruby code, to the desired collection of the selected database. As you can see, you can insert multiple documents one by one.

The following version of `find()` (*below*) uses both `$gt` and `$lt` operators to query the someData collection of the LXF database and select documents that match certain criteria:

```
$collection.find({"n" => {"$gt" => 205000} ,"code" => {"$lt" => 1}}).each do |doc|
  puts doc
end
```

As you can see, the `find()` function might return multiple documents. Therefore, you will have to use `.each` to access the returned documents one by one. The `find_one()` function returns just one JSON document which means that you should not use an iterator to read its results.

The code (above) finds all documents where the `n` key has a value greater than `205000` and the `"code"` key value is less than 1 and iterates through all the documents that match the conditions in the `find()` function.

There exist two functions that allow you to perform updates on a MongoDB database: update_one() and update_many(). The first one updates just one document – if many documents match the criteria only the first one gets updated – whereas the second one updates all documents that match the given criteria. The Ruby example (below) shows how to update a document using `update_many()`:

```
result = $collection.find({"n" => {"$gt" => 205000}}).update_many({ "$inc" => { :code => 10 }})
# The next command returns the number of documents that were updated.
puts result.n
```

As you can see, you will first need to use `find()` to select the documents you want to update and then use `update_many()` to actually update the documents.

Listing indexes

The following Ruby code lists all indexes of the someData collection that belong to the LXF database:

```
collection.indexes.each do |index|
  p index
end
```

As a collection can have multiple indexes, you must use iteration to get all results from the `.indexes` function. The full Ruby code can be found inside the **indexes.rb** file in the code archive. If the someData collection doesn't exist, you will get an error message similar to the following:

```
D, [2015-11-21T18:06:32.078495 #18166] DEBUG -- : MONGODB | 127.0.0.1:27017 | LXF.listIndexes | FAILED | no collection (26) | 0.000233s
```

Please note that the index for the _id field is automatically generated for each collection on a MongoDB database as soon as you insert some data on the collection. Additionally, you can also create new users using the Ruby MongoDB driver. The following code shows the technique:

```
$client.database.users.create(
    'linuxFormat',
    password: 'aPass',
    roles: [ Mongo::Auth::Roles::READ_WRITE ])
```

The complete Ruby code can be found in **newUser.rb**. You'll note that the output of **newUser.rb** changes when executed two times in a row (*pictured, below*). The first time the user is successfully generated without any errors; however, the second time the process fails because the user already exists. Note that there is a special database on every *MongoDB* instance that keeps user-related data called 'database'. Also, bear in mind that in order to view the information of another user, you must have the viewUser action on the database of the other user.

The Python driver

The Python driver works with the latest as well as older versions of *MongoDB*. Its source code can be found at **https://github.com/mongodb/mongo-python-driver**. You can find the exact version of your Python *MongoDB* driver by executing the following code:

```
$ python -c "import pymongo; print(pymongo.version)"
2.6.3
```

It's now time to go back to the Python program (pictured in the screenshot below) and explain it a little more. You first call the `MongoClient()` function with two parameters. The first parameter specifies the IP of the desired *MongoDB* server and

```
2. mtsouk@mtsouk-ubuntu: ~/code/mongoDB (ssh)
> use LXF
switched to db LXF
> db.fs.files.find().pretty()
{
        "_id" : ObjectId("5650474fbd6569222e000001"),
        "chunkSize" : 261120,
        "uploadDate" : ISODate("2015-11-21T10:28:31.233Z"),
        "contentType" : "binary/octet-stream",
        "filename" : "image.png",
        "length" : 7864,
        "md5" : "c583db040256dd1a7cb64cef247f908c"
}
{
        "_id" : ObjectId("5650474fbd6569222e000003"),
        "chunkSize" : 261120,
        "uploadDate" : ISODate("2015-11-21T10:28:31.234Z"),
        "contentType" : "binary/octet-stream",
        "filename" : "aFile.txt",
        "length" : 32,
        "md5" : "eba5fcda08a667446469cd3cb3ce9e8a"
}
> db.fs.chunks.find().pretty()
{
        "_id" : ObjectId("5650474fbd6569222e000002"),
        "data" : BinData(0,"SSBhbSBhIE5FVyBmaWxlIHN0b3JlZCBpbiBHcmlkRlM="),
        "files_id" : ObjectId("5650474fbd6569222e000003"),
```

› GridFS stores its files in two system tables named fs.files and fs.chunks. (Details are in the box on the previous page)

› This is the output of the newUser.rb Ruby script when executed two times in a row. The second time the script fails as the user already exists

```
mongoDB$ ruby newUser.rb
D, [2015-11-21T18:23:43.830336 #9031] DEBUG -- : MONGODB | Adding 127.0.0.1:27017 to the cluster.
/var/lib/gems/1.9.1/gems/mongo-2.1.2/lib/mongo/operation/result.rb:226:in `validate!': User "linuxFormat@LXF" already exists (11000) (Mongo::Error::OperationFailure)
        from /var/lib/gems/1.9.1/gems/mongo-2.1.2/lib/mongo/operation/write/write_command_enabled.rb:48:in `execute_write_command'
        from /var/lib/gems/1.9.1/gems/mongo-2.1.2/lib/mongo/operation/write/write_command_enabled.rb:38:in `execute'
```

the second specifies the desired port number. You then define the database you want to work with using `"db = client.LXF"`. Last, you select the desired collection with `sampleData = db.sampleData`. After that, you are free to use the sampleData variable to interact with the *MongoDB* server. More or less, the names of the Python methods you need to call are the same as the Ruby names. The find_one() function used randomly selects a JSON document from the currently selected collection.

It is time to see something more practical. Should you wish to find all JSON documents of the someData collection, you would use the following version of `find()` as long with a for loop:

```
# Choose a collection
someData = db.someData
for document in someData.find():
        print document
```

As you've already learnt, the main difference between find() and find_one() is that the former returns a cursor that you will have to iterate in order to get all returned documents, whereas find_one() randomly returns a single JSON document that you just need to store using a variable.

The following version of `find()` (*below*) uses `$gt` to query the `someData` collection of the LXF database and sort the results by the `n` field using the `sort()` method:

```
for doc in someData.find({"code": {"$gt": 30}}).sort("n"):
        print doc
```

Other useful *MongoDB* operators include $ne (not equal), $lte (less than or equal), $gte (greater than or equal) etc.

The next Python code (**update.py**) shows how to update an existing document:

```
# Choose a document.
print someData.find_one({"n" : 1})
# Update it
someData.update({"n" : 1}, {'$set':{'newField': 10}})
# Print it again.
print someData.find_one({'newField': 10})
```

```
mongoDB$ mongofiles -d LXF list
2015-11-21T12:37:59.886+0200    connected to: localhost
image.png       7864
aFile.txt       32
image.png       7864
aFile.txt       32
mongoDB$ ruby storeGridFS.rb  | grep SUCCEEDED
D, [2015-11-21T12:38:10.638629 #8831] DEBUG -- : MONGODB | 127.0.0.1:27017 | LXF.find | SUC
CEEDED | 0.00018091399999999998s
D, [2015-11-21T12:38:10.639588 #8831] DEBUG -- : MONGODB | 127.0.0.1:27017 | LXF.insert | S
UCCEEDED | 0.000271322s
D, [2015-11-21T12:38:10.640080 #8831] DEBUG -- : MONGODB | 127.0.0.1:27017 | LXF.insert | S
UCCEEDED | 0.00018853700000000001s
D, [2015-11-21T12:38:10.640509 #8831] DEBUG -- : MONGODB | 127.0.0.1:27017 | LXF.find | SUC
CEEDED | 0.000140543s
D, [2015-11-21T12:38:10.640932 #8831] DEBUG -- : MONGODB | 127.0.0.1:27017 | LXF.insert | S
UCCEEDED | 0.000162581s
D, [2015-11-21T12:38:10.641331 #8831] DEBUG -- : MONGODB | 127.0.0.1:27017 | LXF.insert | S
UCCEEDED | 0.00015727000000000002s
mongoDB$ mongofiles -d LXF list
```

› Both storeGridFS.rb and retrieveGridFS.rb in action, and mongofiles verifies that both Ruby scripts are working

The following Python code (**drop.py**) drops the moreData collection from the LXF database and should be used with great care:

```
# Choose a collection
moreData = db.moreData
# Drop the entire collection!
print moreData.drop()
```

Please note that the following two calls are equivalent:

```
>>> db.aCollection.drop()
>>> db.drop_collection("aCollection")
```

Creating MongoDB Indexes

In order to create a single key ascending index for the `n` key, you should use the following Python code:

```
someData.create_index("n")
```

It is considered a good practice to check the correctness of your programs before using them in production. (There's an example of this pictured on the left, where, using the mongoDB shell , the **getIndexes()** function verifies that the new index was successfully created.) The **index.py** file in the source code archive contains the full Python code. Similarly, you can drop an existing index using Python as follows:

```
# Choose a collection
someData = db.someData
# Drop an index
someData.drop_index("n_1")
```

The **dropIndex.py** file contains the Python code. If the index you specified does not exist, you'll get an error message similar to the following:

```
pymongo.errors.OperationFailure: command
SON([('dropIndexes', u'someData'), ('index', 'n_1')]) failed: index
not found with name [n_1]
```

Not everything should be done with Ruby or Python as administrative tasks are better done using the Mongo shell and the JavaScript programming language. Therefore, in another tutorial we'll teach you how to administer *MongoDB* and create Replica sets from the *MongoDB* shell.

However, knowing how to use *MongoDB* with your favourite programming language can be very handy and enjoyable, so in a third tutorial we will teach you how to create a blog site using *MongoDB*, the Python driver, the knowledge from this tutorial and the Bottle framework.

MongoDB is a great and modern database, ripe for exploitation! ■

Quick tip

GridFS works by splitting a large file into smaller files that are stored as separate documents in *MongoDB*. Each chunk has a maximum size of 255 KB.

The mongofiles command line utility

In case you don't want to write any code yourself, you can save and retrieve GridFS files with the help of the *mongofiles* command line tool that comes with the *MongoDB* distribution.

```
$ mongofiles -d LXF put abinaryFile
2015-11-18T21:45:51.567+0200 connected to: localhost
added file: abinaryFile
$ mongofiles -d LXF get abinaryFile
2015-11-18T21:45:56.674+0200 connected to: localhost
finished writing to: abinaryFile
```

```
$ mongofiles -d LXF list
2015-11-18T21:46:01.635+0200 connected to: localhost
abinaryFile             7168
$ mongofiles -d LXF delete aBinaryFile
2015-11-18T21:49:37.759+0200 connected to: localhost
successfully deleted all instances of 'aBinaryFile' from GridFS
```

The `list` command allows you to see all the GridFS files that are stored in the `LXF` database. Should you wish to get a binary file using its _id you'll need to use the `get_id` command. Similarly, you can delete a GridFS file using the `delete` command – please bear in mind that if a file is stored multiple times, the `delete` command deletes all instances of that particular filename.

The `mongofiles` command supports a number of other options that you can view when you execute it with the `--help` parameter. If you need more information about *mongofiles*, you can check **http://docs.mongodb.org/manual/reference/program/mongofiles**.

MongoDB: An admin's guide

Now it's time to cover all the groundwork for administering a MongoDB database, including how to create a Replica Set

Let's dive deeper into *MongoDB*, a document-oriented NoSQL database that has become very popular. We'll show you how to perform various administrative tasks, with the help of a dummy table we'll create and populate with data, and we'll also look at how to create a Replica Set. We won't deal with Storage Engines as the commands for database administration are the same regardless of the storage engine used. However, if you're using *MongoDB* on a production server you should study WiredTiger and MMAPv1. Additionally, beginning with version 3.2 of *MongoDB*, WiredTiger is the default storage engine.

You can find information about installing the latest version of *MongoDB* at **www.mongodb.org/downloads**. At the time of writing, 3.2 is the current version and we'll install this on a Ubuntu 14.04 system from the following archive:

```
$ wget https://fastdl.mongodb.org/linux/mongodb-linux-x86_64-ubuntu1404-3.2.0.tgz
```

Once extracted, you'll have to run

```
$ tar zxvf mongodb-linux-x86_64-ubuntu1404-3.2.0.tgz
$ cd mongodb-linux-x86_64-ubuntu1404-3.2.0/
$ cd bin
$ sudo mkdir -p /data/db
$ sudo chown `id -u -n` /data/db
$ sudo ./mongod
```

to start a *MongoDB* server. The last command will execute a *MongoDB* server that will print its log messages in your terminal. Later on we'll cover how to run a *MongoDB* server in the background without an attached terminal, but putting the *MongoDB* **bin** directory in your PATH will make life easier:

```
$ export PATH="/home/mtsouk/mongodb-linux-x86_64-ubuntu1404-3.2.0/bin":"${PATH}"
```

Where `/home/mtsouk/mongodb-linux-x86_64-ubuntu1404-3.2.0/bin` should be replaced with your path. Next, you should be able to tell the version of *MongoDB* you're using which can be done with two commands:

```
$ ./mongod --version
db version v3.2.0
git version: 45d947729a0315accb6d4f15a6b06be6d9c19fe7
OpenSSL version: OpenSSL 1.0.1f 6 Jan 2014
allocator: tcmalloc
modules: none
build environment:
    distmod: ubuntu1404
    distarch: x86_64
    target_arch: x86_64
$ ./mongo --version
MongoDB shell version: 3.2.0
```

The `mongod` refers to the name of the *MongoDB* server process, whereas `mongo` is the name of the *MongoDB* shell which mainly accepts JavaScript commands. As you saw, both executables give you the same information about the *MongoDB* version. Most of the tasks presented in this article will be performed using the Mongo shell, which starts by executing the `mongo` command. If you run mongo without specifying a port number, it will automatically try to connect to localhost using port number 27017 which is the default port number used by the MongoDB server process. Should you wish to specify a different port number, you can run mongo with the **--port** option followed by the desired port number. (The screenshot opposite shows how you can connect to a MongoDB server that runs on your local machine using its default port number as well as the contents of the **/data/db** directory and some log messages.)

Before you try any more commands, you should run the following JavaScript code from the MongoDB shell in order to add some data to your database and have something to experiment with:

```
use LXF
switched to db LXF
for (var i=0; i<100000; i++) { db.myData.insert({x:i, y:i*i}); }
WriteResult({ "nInserted" : 1 })
db.myData.count();
100000
```

The first command switches to the LXF database and if the database doesn't exist, it will be auto-created. It then uses a JavaScript for loop in order to insert 100,000 documents into the myData collection of the LXF database. Once again, if the myData collection isn't present it will be automatically created. The last command shows how you can find out the total number of records that exist in a collection.

CRUD commands

This section will teach you how to insert, delete, update and select documents from a *MongoDB* collection, which is what *MongoDB* calls a table. If you try to insert a document that has a different set of keys to an existing collection then it will be created without any complaints. This means that typos aren't detected very easily. Each time you insert a BSON document without specifying the value of the **_id** field, *MongoDB* automatically generates this field which acts as the Primary Key for the BSON document. When created automatically, an _id field is 12 bytes long and looks like this:

Quick tip

To learn more about MongoDB you can check *MongoDB: The Definitive Guide*, 2nd Edition by Kristina Chodorow and *MongoDB in Action*, 2nd Edition by Kyle Banker, Peter Bakkum, Shaun Verch, Douglas Garrett and Tim Hawkins.

Sharding

Sharding is a method for storing data across multiple machines. Shards store the data whereas config servers store the metadata of the cluster. Query Routers, which are also called mongos instances, interface with client applications and direct operations to the appropriate shard or shards.

MongoDB divides the shard key values into chunks – they aren't related to GridFS chunks– and distributes the chunks evenly across the various shards. *MongoDB* distributes data at the collection level. In order to shard a collection, you need to define a shard key, which is either an indexed field or an indexed compound field that exists in every document in the collection.

MongoDB uses range-based sharding because it's better for answering queries and, sometimes, is better for keeping your data sorted. You can combine sharding with replication. As you do not want clients to talk directly to a shard server, you must configure mongos to listen to the default MongoDB port (27017). The only reason for connecting directly to anything except mongos is for DBA work. Please note that three config servers can handle a MongoDB cluster with 1000 total machine. Also, all config servers hold the same data.

```
"_id" : ObjectId("55e063b1b1983f50f79646be")
```

You can also get a random document from a collection using `findOne()`:

```
db.myData.findOne()
```

The main difference between `find()` and `findOne()` is that the former can return multiple documents with the help of a cursor, whereas the latter randomly returns a single BSON document. You can insert a document with:

```
db.myData.insert( { "x": 123, "y":500 } )
WriteResult({ "nInserted" : 1 })
```

Similarly, you can delete a document that matches certain criteria as follows:

```
db.myData.remove({"y": 500})
WriteResult({ "nRemoved" : 1 })
```

You can also update an existing document using:

```
db.myData.update({"x": 123}, {$set: { "z": 123}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1
})
```

To update multiple documents at the same time, set the `multi:` option (below) to true while using `update()`:

```
> db.myData.update({"x": 123}, {$set: { "z": 123}},
{multi:true})
```

Although you can omit the quotes around the names in the "key": value pairs (eg `"x":123`) when working in the Mongo shell, they are still a part of the JSON specification so keep in mind that in JSON they have to be there. There's also a very handy utility that comes with *MongoDB* called *mongostat*. This displays basic and live server statistics in a similar way to the *vmstat* utility.

You can find all available databases on the current *MongoDB* server as follows:

```
show databases
LXF   0.003GB
local  0.000GB
```

Similarly, you can find the available collections on the current database with:

```
> show collections
myData
```

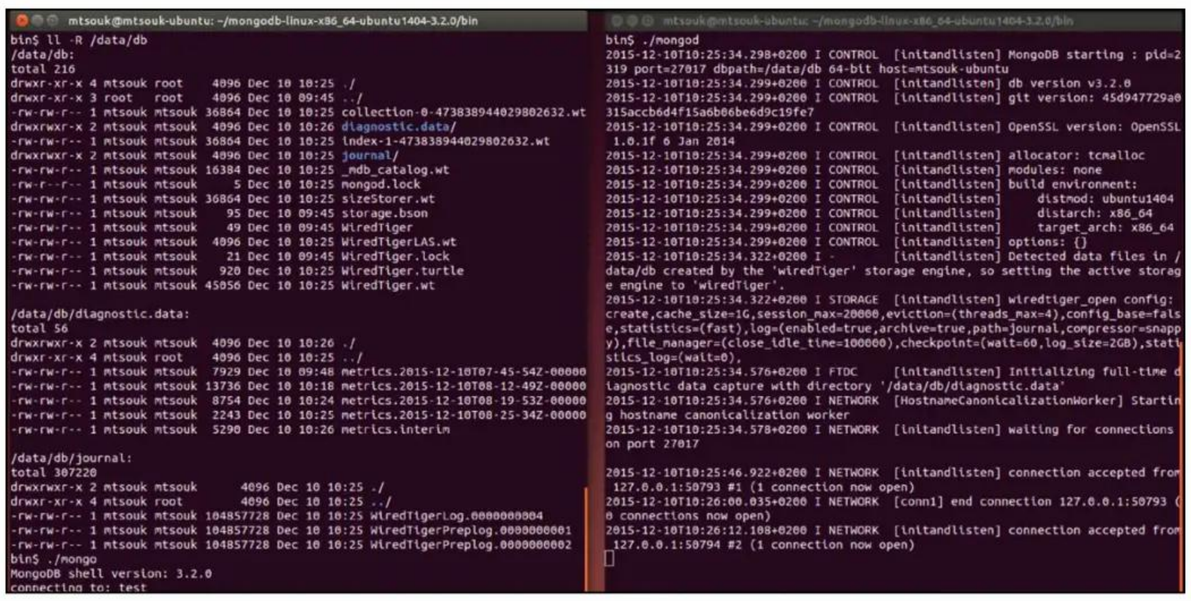
If you wish to delete the entire myData collection, use:

```
db.myData.drop();
true
db.myData.count();
0
```

You can drop a database, including its data files, with:

```
use LXF
switched to db LXF
> db.runCommand( { dropDatabase: 1 } )
{ "dropped" : "myData", "ok" : 1 }
```

This command doesn't delete the users associated with the database. Should you wish to drop the associated users, you need to run `dropAllUsersFromDatabase` in the database you are deleting.



› In the left terminal is the contents of the /data/db directory and In the right one are the log files generated by the running *MongoDB* server

You can manually start a *MongoDB* server process from the command line with:

```
$ mongod --fork --logpath a.log --smallfiles --oplogSize 50
--port 27101 --dbpath w1 --replSet w --logappend
```

Here the `--port` parameter defines the port number that the server will listen to; the `--dbpath` value defines the directory that will contain the database files; the value of the `--logpath` parameter shows the file that will hold the log messages; and the `--fork` parameter tells the OS that the process will run in the background without an associated terminal. The `--replset` parameter defines the name of the replica set and should only be included when you want to define a replica set. The `--logappend` parameter tells the *MongoDB* process to append to the log file instead of over-writing it. The `--oplogSize` parameter specifies the maximum size in megabytes for the replication operation log and should only be included when you want to create a replica set.

If you try to start a *MongoDB* server without a proper log file, you'll get this kind of error message:

```
2015-11-29T12:01:54.349+0200 F CONTROL  Failed global
initialization:
2015-12-29T16:02:25.725+0200 F CONTROL  [main] Failed
global initialization: FileNotOpen Failed to open "/a/path/a.
log"
```

Quick tip

TheMongoDB manual is available at **https://docs.mongodb.org** but the MongoDB University website also offers free online courses (**https://university.mongodb.com**).

Queries and Indexes

Analysing a query is a very good way to find out why a query runs slowly as well as how your query is executed:

```
> db.myData.find({ "x": { $gt: 99990} }).
explain("executionStats")
```

The left part of the screenshot (*below*) shows the output of `explain("executionStats")` on the query (above). »

» The query uses an execution plan that does a collection scan (COLLSCAN), which means that all documents of the collection are going to be visited. Therefore, if the winning plan has the COLLSCAN keyword, then no index has been used which means that the DB admin has to rethink the currently defined indexes and define more appropriate ones. A properly defined index can speed up your queries a lot. We'll now show how to define an index and execute the previous query. The index for the "x" key will be created with:

```
> db.myData.createIndex({"x":1})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
```

You can now execute the previous `find()` command with `explain("executionStats")`. The right part of screenshot (*below*) shows the output of `explain()` when an index is used. As you can see the output is a little different now because this query returns the results by scanning the index keys (IXSCAN) and is significantly faster than the previous one because of it. Additionally, the query now looks at only nine documents whereas the query without the index looked at all the collection's documents. This is more obvious in the following output:

```
        "executionTimeMillis" : 0,
        "totalKeysExamined" : 9,
        "totalDocsExamined" : 9,
```

The previous output verifies that with the help of an index it took the `find()` command 0 milliseconds to complete because it just examined nine documents (denoted by `totalDocsExamined`) in order to return nine documents.

The `getIndexes()` function returns the indexes of a collection whereas the `dropIndex()` function deletes an existing index. However, you can't drop the index of the _id field.

```
> db.myData.getIndexes()
[
        {
                "v" : 1,
                "key" : {
                        "_id" : 1
                },
                "name" : "_id_",
                "ns" : "LXF.myData"
        },
        {
                "v" : 1,
                "key" : {
                        "x" : 1
                },
                "name" : "x_1",
                "ns" : "LXF.myData"
        }
]
```

MongoDB automatically creates an index for the _id field that can be found on every collection. You can find all indexes that belong to the active database with:

```
db.getCollectionNames().forEach(function(collection) {
  indexes = db[collection].getIndexes();
  print("Indexes for " + collection + ":");
  printjson(indexes);
});
```

Although defining indexes is a simple process, you should have in mind that indexes take disk space and should be used only when needed.

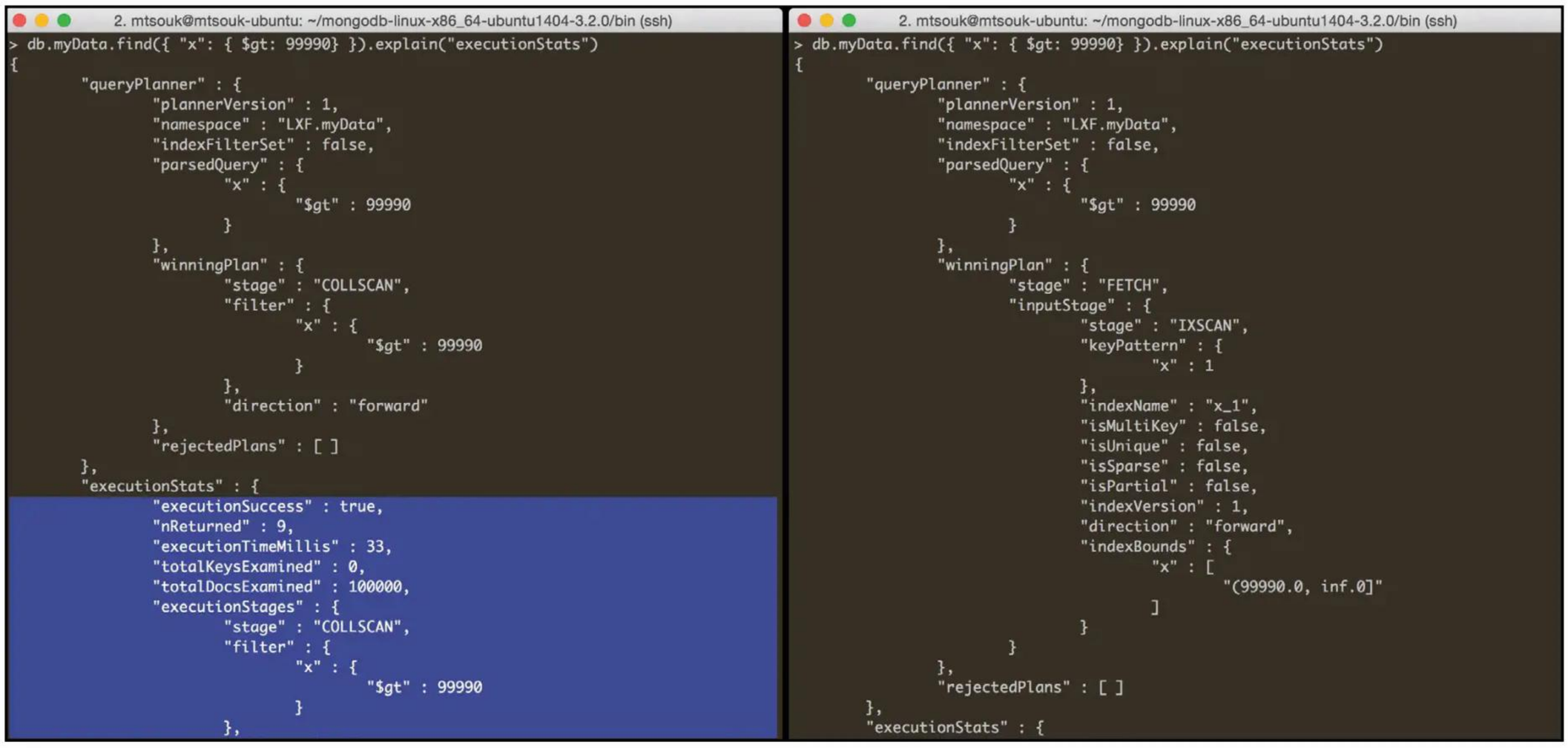
Quick tip

As always, the best way to learn *MongoDB* is to use it on a project. In the next part we'll cover how to create a blog site with the help of *MongoDB*, Python and Bottle.

Replication

Basically replication means two things: the first meaning is durability; the changes must be propagated and not get lost if something bad happens. The second meaning is availability, which refers to the servers being available as much as possible; when a server goes down, another one must take its place without any user action.

Most replica sets have three nodes: primary, secondary



› An index can tremendously improve the execution time of a query. The explain(executionStats) function will tell you if a query uses an index or not

and arbiter. The arbiter is a special kind of node that doesn't save any data and cannot become primary; it just participates in voting for a new primary. It's also common for a replica set to have one primary node and two secondaries. Additionally, it's also common for a replica set to have one or more hidden nodes which aren't visible to the driver, don't answer any user queries, and can't become primary nodes. However, they can be used for backups and analytics, and they can also vote.

You will need to start at least three *MongoDB* instances to showcase a *MongoDB* replication scenario. The good thing is that all instances can run on the same machine, although this isn't good practice for production. So, you can start the three different processes as follows:

```
$ mkdir node1; mkdir node2; mkdir arbiter
$ mongod --replSet LXF --dbpath node1 --port 30000
--logappend --logpath a.log --fork
$ mongod --replSet LXF --dbpath node2 --port 30001
--logappend --logpath b.log --fork
$ mongod --replSet LXF --dbpath arbiter --port 30002
--logappend --logpath c.log --fork
```

Then, you must connect to the machine that you want to act as the primary node and initiate the replica set with:

```
$ mongo --port 30000
> rs.initiate()
> rs.add("mtsouk-ubuntu:30001")
> rs.add("mtsouk-ubuntu:30002", {arbiterOnly: true})
```

The `rs.status()` command can help you learn more about the status of a replica set. When the primary node goes down for some reason, the following things will happen: The remaining nodes will detect the failure, as they can't find the missing node. After confirming that the node is actually down, the nodes will vote for a new primary node. After a successful election, a secondary node becomes the new primary and accepts writes.

Executing the `rs.status()` command after stopping the primary node, which can be done by terminating its process with the `kill` command will produce a slightly different output than the one shown.

System tables

MongoDB stores system information in collections that use the <database>.system.* namespace, which *MongoDB* reserves for internal use. Therefore, you should not create any collections that begin with system. The 'admin' database stores many system collections, including admin.system.roles, which stores custom roles that administrators create and assign to users to provide access to specific resources. The authentication credentials of every user as well as any roles assigned to the user are stored in admin.system.users, and the schema version of the user credential documents are stored in admin.system.version.

Each individual database has some collections that keep track of data related to each specific database. The <db>.system.namespaces collection contains information about the collections of the database, the <db>.system.indexes contains information about the indexes of the database. However, <db>.system.indexes shouldn't be accessed directly but with the help of the getIndexes() function. The <db>.system.profile collection contains information about database profiling. Database profiling can help you identify inefficient queries and operations.

Last but not least, the <db>.system.js collection holds special JavaScript code for use in server-side JavaScript.

Each mongod instance has its own 'local' database. The 'local' database stores data used in the replication process and other instance-specific data. The 'local' database isn't replicated as it's invisible to the replication process. The *MongoDB* system tables can also give you information about stored GridFS files. There exist two system tables, one for holding the filenames and another that holds the various chucks with the actual binary data that compose a file. The default behaviour of GridFS is to use two collections with names prefixed by the fs bucket: fs.chunks and fs.files. If you want to find all stored GridFS files kept on a database, you can query the system table that holds this information. This is the output from a query to the fs.files table:

```
use LXF
switched to db LXF
> db.fs.files.find()
{ "_id" : ObjectId("5682af61bd65690bcc000001"),
"chunkSize" : 261120, "uploadDate" : ISODate("2015-12-
29T16:05:53.642Z"), "length" : 28549, "md5" :
"1494011900cf195d44165191d042a124", "filename" : "/
home/mtsouk/aFile.zip" }
{ "_id" : ObjectId("5682af7cbd65690bdb000001"),
"chunkSize" : 261120, "uploadDate" : ISODate("2015-12-
29T16:06:20.882Z"), "length" : 28549, "md5" :
"1494011900cf195d44165191d042a124", "filename" :
"aBinaryFile.zip" }
{ "_id" : ObjectId("5682af85bd65690be8000001"),
"chunkSize" : 261120, "uploadDate" : ISODate("2015-12-
29T16:06:29.642Z"), "length" : 35444349, "md5" :
"92c7ae236c2ea6326a603c9cffb9437b", "filename" :
"mongosniff" }
```

Alternatively, you can use the *mongofiles* tool to list all GridFS files that a present on a database:

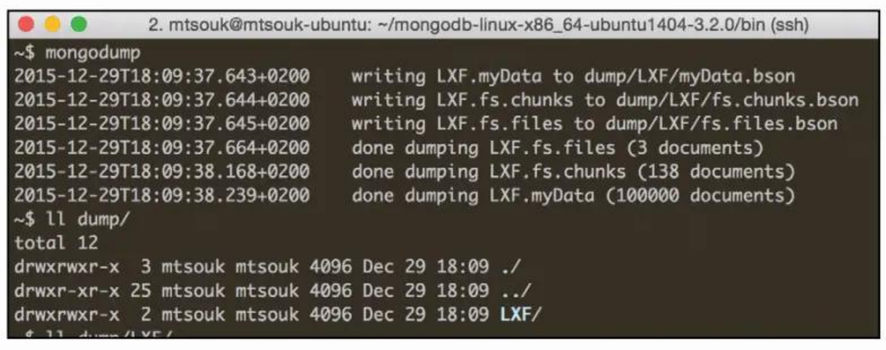
```
$ mongofiles -d LXF list
2015-12-29T18:07:54.895+0200    connected to:
localhost
/home/mtsouk/aFile.zip  28549
aBinaryFile.zip     28549
mongosniff      35444349
```

If you need more information about *mongofiles*, visit **http://bit.ly/MongofilesDocs**. ■

Backup and recovery

Having a backup strategy is essential because all systems will eventually fail. Although using *MongoDB* sharding and replication can minimise the need for a backup, bad things can happen at any time. Additionally, if you're running a single server process, then it's mandatory to take regular backups. The two main utilities are: *mongodump* and *mongorestore* and they can help you backup and restore a *MongoDB* database, respectively.

The screens*hot (pictured, left)* briefly shows off the use of *mongodump* and *mongorestore*. The `mongorestore` command was executed with the `--quiet` parameter in order to hide all log output.



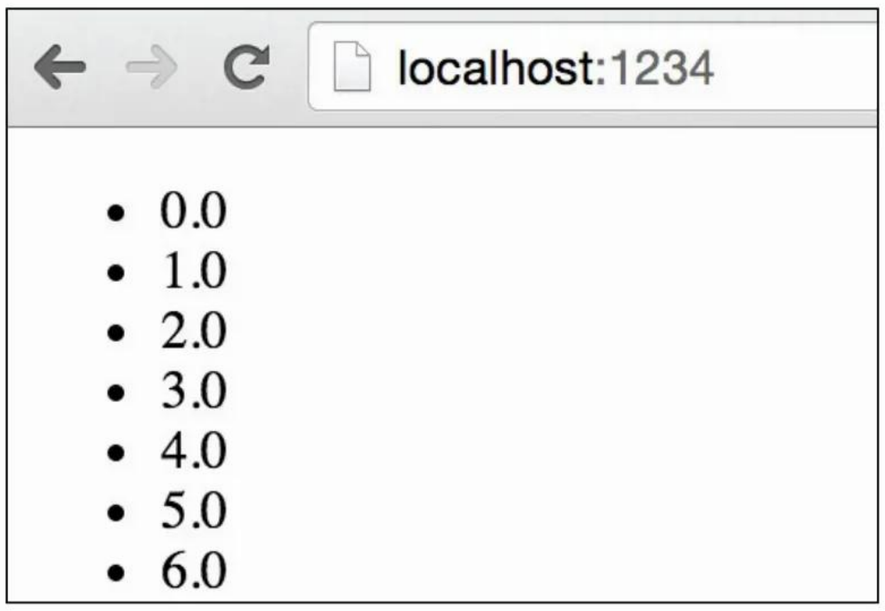
› The *mongodump* and *mongorestore* utilities can help you backup and restore a *MongoDB* database

Databases

MongoDB: Build a blog

Add the Bottle Python framework to MongoDB and get posting

This tutorial will use the *MongoDB* knowledge you already have and combine it with Bottle, a Python framework, in order to create a blog site that stores its data on a *MongoDB* database. The Bottle framework will be mainly responsible for the user interface of the website as well as its logic. We're assuming that you'll have both *MongoDB* and PyMongo already installed on your Linux distro. If you don't feel comfortable with *MongoDB* and *PyMongo* – the Python *MongoDB* driver – we'd recommended revisiting the articles on the previous few pages to learn more about using the driver and *MongoDB* administration, respectively). At the end of this tutorial, you'll have a beautiful blog site written in Python that uses *MongoDB* to store its data and Bottle to display it.



This is an extract of the generated output from the sampleData.py script that prints the values of the 'x' key from the documents of the 'sampleData' collection

The Bottle framework

Bottle is a fast, simple and lightweight WSGI micro web-framework written in Python. The whole framework is a single file module that has no dependencies other than the Standard Library of Python. At the time of writing, the latest stable Bottle version is 0.12.9 and you can install Bottle by executing the following command as root with `$ pip install bottle`. On a Debian system this will install Bottle at **/usr/local/lib/python2.7/dist-packages/bottle.py** and at **/usr/local/bin/bottle.py**. However, as Bottle doesn't depend on any external Python libraries, you can download **bottle.py** and put it inside the directory that you use for development:

```
$ wget http://bottlepy.org/bottle.py
$ wc bottle.py
  4107   15384  156332 bottle.py
```

The "Hello World!" program in Bottle looks like this:

```
from bottle import route, run, template

@route('/hello/<user>')
def index(user):
    return template('<h2>Hello World from {{user}}!</h2>', user=user)
run(host='localhost', port=1234)
```

The run method that's imported can be used to run the application in a development server, which is the best way to test your application as you write it. The route method tells the application about the supported URL requests as well as how to handle them using Python functions. Routing in Bottle applications is implemented by calling a single Python function for each supported URL. This isn't a dumb Hello World program because it also displays the name of the user which is included in the URL. As you can see, you define the user variable and then you pass it to the `template()` function. You can test this simple web page by executing the Python script as follows:

```
$ python hw.py
Bottle v0.12.8 server starting up (using WSGIRefServer())...
Listening on http://localhost:1234/
Hit Ctrl-C to quit.
127.0.0.1 - - [25/Jan/2016 16:28:26] "GET /hello/tsoukalos
HTTP/1.1" 200 36
```

You should now go to your favourite web browser and point it at **http://localhost:1234/hello/tsoukalos** to see the web page you've just created! Please note that trying to get the **http://localhost:1234/hello/tsoukalos/** URL (that ends with the '/' character) will fail as it's not configured.

As you might have guessed, the host name and the port number of the simple web server are defined inside **hw.py** with the help of the following code:

```
run(host='localhost', port=1234)
```

Should you wish to print debugging information, you should turn on the debug mode as follows:

```
run(host='localhost', port=1234, debug=True)
```

Connecting Bottle with MongoDB

The connection between Bottle and *MongoDB* is possible with the help of the Python *MongoDB* driver. The following example (**sampleData.py**) uses Bottle to read a *MongoDB* server and display the values of the 'x' key that can be found in the documents of the 'sampleData' collection of the LXF

Basic CRUD commands

CRUD stands for Create, Read, Update and Delete, which are the basic operations that can be performed on any database. In order to be able to follow this *MongoDB* tutorial, it would be very helpful to bear the basic CRUD commands in mind so that you can check what the Python scripts do or don't do.

You can get one random document from a collection using `findOne()`:

```
> db.sampleData.findOne()
```

In the code below, the first command returns all documents from a collection whereas the second command returns all documents that have the `n` key set to the `324` value:

```
> db.sampleData.find()
> db.sampleData.find({n: 324})
```

The main difference that you'll discover between find() and findOne() is that the former function can return multiple documents with the help of a cursor, whereas the latter will randomly return a single BSON document.

The next example sorts the output of `find()` based on the `x` field in descending order:

```
> db.sampleData.find().sort( { x: -1 } )
```

You can also insert a document into *MongoDB* using the follow:

```
> db.sampleData.insert( { "x": 23, "y":13 } )
```

Similarly, you can delete a document that matches certain criteria in the following way:

```
> db.sampleData.remove({"y": 13})
WriteResult({ "nRemoved" : 1 })
```

Please bear in mind that the safest way for you to identify and delete a document is by using its _id field. You can also update an existing document using the following command:

```
> db.sampleData.update({"x": 123}, {$set: { "z": 123}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

A useful way of updating multiple documents all at once is to set the `multi` option to true (`{multi:true}`) while using the `update()` function, in the following way:

```
> db.sampleData.update({"x": 23}, {$set: { "anotherKey": 54321}}, {multi:true})
```

database. The *MongoDB* server used in the Python script is located on the same machine and the script uses the default *MongoDB* port which is 27017. The Python code of **sampleData.py** is the following:

```
import pymongo
from pymongo import MongoClient
from bottle import route, run, template

# Get data from MongoDB
myData = []
client = MongoClient('localhost', 27017)
db = client.LXF
cursor = db.sampleData.find({}, {'_id':0, 'y':0})
for myDoc in cursor:
    myData.append(myDoc['x'])

@route('/')
def rootDirectory():
    return template('listContents', data=myData)

run(host='localhost', port=1234)
```

The `myDoc['x']` code is used for getting the actual value of the 'x' key, which is then put it in the `myData` list – this list is then passed as a parameter to the `template()` function. The `find()` command used doesn't return the values of the `'y'` and `'_id'` fields to save CPU time, which is very handy when you have lots of data on your *MongoDB* database.

As you can see, the **sampleData.py** script needs one external file in order to run – this is how Bottle organises its projects. Although it might be possible to include the code of an external Bottle file inside the main Python script, it's better to save such code separately, especially when you are dealing with large projects that support multiple URLs.

The use of the `'listContents'` template, with the help of the `template()` method, simplifies the code of **sampleData.py** – the small price that you pay for this is that you have to edit multiple files. The contents of the **listContents.tpl** file, the TPL extension is automatically used, are the following:

```
<!DOCTYPE html>
<html><head>
<title>LXF.sampleData: Contents of 'x' key</title>
</head>
<body>
<ul>
%for myX in data:
<li>{{myX}}</li>
%end
</ul>
</body></html>
```

Although template files mainly contain HTML code, they also have access to variables, such as data, which makes them able to generate dynamic content. The contents of the internal data variable are accessed using a for loop that's coded using a pretty straightforward way. The data is formatted using an HTML list. Before running **sampleData.py**, as you did with **hw.py**, make sure that the sampleData collection of the LXF database has data in it. If not, run the following code from the *MongoDB* shell to insert some data:

```
> use LXF
switched to db LXF
> for (var i=0; i<100; i++) { db.sampleData.insert({x:i, y:i*i}); }
WriteResult({ "nInserted" : 1 })
> db.sampleData.count();
100
```

If you wish to delete the entire sampleData collection to start with an empty one, you should execute:

```
> db.sampleData.drop();
true
```

As saving data on a database takes disk space, it's good to know how to delete entire collections and databases in a *MongoDB* database. The following command deletes the entire LXF database along with its data files:

```
> use LXF
switched to db LXF
> db.runCommand( { dropDatabase: 1 } )
{ "dropped" : "myData", "ok" : 1 }
```

Despite the various similarities between **sampleData.py** and **hw.py**, the former is a more advanced program because it connects to a *MongoDB* database to get its data. If you can successfully execute **sampleData.py** and get its output on your favourite web browser, you're good to continue with the rest of the tutorial; otherwise, you should try to correct any errors on your Python code or your *MongoDB* configuration.

Quick tip

You can find more information about the Bottle framework at **http://bottlepy.org**. For more on the Python MongoDB driver go to **https://docs.mongodb.org/ecosystem/drivers/python**.

Handlers, Views, Forms and Cookies

Bottle follows the MVC (Model-View-Controller) software pattern in order to separate the different functions of the user interface.

The Model is responsible for storing, querying and updating data whereas the View shows how the information »

» will be displayed onscreen, and finally, the Controller holds the logic of the application.

Using Python code, Bottle breaks its sites into Handlers, Views and Forms. A URL Handler allows you to specify web pages and bind them to Python code. You can have as many URL Handlers as you want. However, if you believe you're defining too many of them, you should rethink the design of your web application. As you saw in **hw.py**, Bottle also supports dynamic routes and regular expressions in its routes. URL handlers are defined with the use of the `@bottle.route()` command.

A View is what the user sees. This is code concerned with displaying information and is usually dynamic content. The View shouldn't contain any code that should be inside the Controller part and they are implemented using Bottle templates. By default, Bottle searches in **./** and **./views/** for templates. A Form is a way of getting input from the user. Bottle also supports Cookies which can be very useful.

Now that you know the basics about Bottle and the structure of its projects, it's time to start building the blog site. The structure of the blog site will be created by executing the following commands:

```
$ mkdir blog
$ cd blog
$ mkdir views
$ touch blog.py
$ touch views/menu.tpl
$ touch views/listPosts.tpl
$ touch views/write.tpl
$ touch views/showsinglepost.tpl
```

So, the root directory that contains all project files and directories is named **blog**. The main file which contains the URL rules is called **blog.py**. Its Python code is the following:

```
import pymongo
from pymongo import MongoClient
from bottle import route, run, template, request, get, post, redirect
import datetime
from bson import ObjectId

@route('/')
def rootDirectory():
    return template('menu')

@route('/list/')
@route('/list')
def listAllPosts():
    # Get data from MongoDB
    myData = []
    client = MongoClient('localhost', 27017)
    db = client.LXF
    cursor = db.blogposts.find()
    for myDoc in cursor:
        myData.append(myDoc)
    return template('listPosts', data=myData)

@get('/write/')
@get('/write')
def writeNewPost():
    return template('write', dict(subject="", body="", tags=""))

@post('/presentnewpost/')
@post('/presentnewpost')
def presentNewPost():
    # Extract the data from the write.tpl FORM
    title = request.forms.get("subject")
    post = request.forms.get("body")

    if title == "":
        title = "There is no title!"
    if post == "":
        post = "This is the text of the post."
    postDocument = { "title": title,
                "text": post,
                "date": datetime.datetime.utcnow()}

    client = MongoClient('localhost', 27017)
    db = client.LXF
    # Write them to the database
    db.blogposts.insert_one(postDocument)
    # Get the postID which is the _id field of the document
    postid = str(postDocument['_id'])
    # print(postid)
    # Present the document
```



› Being able to create new blog posts is essential for a blog site. This image shows the web page that allows you to write new blog posts

Quick tip

Bottle isn't the only Python web framework. Other Python frameworks that you can consider are Django (**www.djangoproject.com**) and Flask (**http://flask.pocoo.org**).

Creating a static site using Bottle

Although the Bottle framework was developed with dynamic sites in mind, it can also be used for creating static sites as well. The following Python code (**static.py**) demonstrates this:

```
from bottle import route, run
@route('/hello/')
@route('/hello')
def hello():
    return "<h1>Hi to you too!<h1>"

run(host='localhost', port=1234, debug=True)
```

The code above defines the handlers for two specific URLs ('/hello' and '/hello/') which are the only URLs that are supported by **static.py**. The actual HTML code is embedded inside the PY file so you don't have to look into too many files when you want to make changes to your site. Additionally, **static.py** shows that you can support multiple routes with one function, as **static.py** does with '/hello' and '/hello/' which is very handy. If you try to access a URL that's not supported by **static.py**, you'll get the usual 404 error. Should you wish to print your own error message, you should use the following route which catches everything that can't be matched by the previous rules:

```
@route('<mypath:path>')
def doSomething(mypath):
    print mypath
    return "Your URL is %s but it does not exist!"
% mypath
```

The main advantage of this approach is that you can easily convert your website from being a static to dynamic one simply by changing the way that you deal with the URL handlers. We've also found that Bottle is a handy and quite a sophisticated way to quickly create a prototype for a dynamic site before you start the actual implementation process.

```
    redirect('/post/' + postid)

# Prints an entire post
@get("/post/<postid>")
def showPost(postid):
    client = MongoClient('localhost', 27017)
    db = client.LXF
    query = {'_id':ObjectId(postid)}
    post = db.blogposts.find_one(query)
    return template('showsinglepost', post=post)
```

As you can see, **blog.py** defines various URL handlers. The **menu.tpl** template file displays the main page of the blog site. The **listPosts.tpl** template briefly lists all blog posts and allows you to select and view any one of them. The **write.tpl** template allows you to write a new blog post using a form – you can customise it any way you want. The **showsinglepost.tpl** view is automatically presented after you've created a new post with **write.tpl**. As you can guess, **showsinglepost.tpl** is used for showing single posts in general.

Before continuing with Bottle, it's time to talk a little about the schema of the *MongoDB* database that will hold the data. As you might already know, *MongoDB* is schemaless which means that two documents that belong to the same collection can have a totally different number of keys with the exception of the _id key, which is mandatory.

This is very important when you are writing code for *MongoDB* because if you misspell the name of a collection in your code, you will get no error messages, instead a new collection will be created! The same thing will happen if you misspell the name of a key of a document.

Creating blog posts

As usual, a template file will deal with the writing of new blog posts. The Bottle template file (**write.tpl**) for creating new blog posts is as follows:

```
<!DOCTYPE html>
<html><head>
<title>Write a Blog Post.</title>
</head>
<body>
<h2>If you do not want to write a new post select one of the following:</h2>
<ul>
<li><a href="/">home Page</a> </li>
<li><a href="/list">List all posts</a></li>
<li><a href="/login">User login (not implemented)</a></li>
<li><a href="/newuser">User Signup (not implemented)
</a></li>
</ul>
<form action="/presentnewpost" method="POST">

<h2>Subject:</h2>
<input type="text" name="subject" size="120"
value="{{subject}}"><br>
<h2>Blog Text:<h2>
<textarea name="body" cols="120" rows="20">{{body}}</
textarea><br><p>
<input type="submit" value="Submit">

</body></html>
```

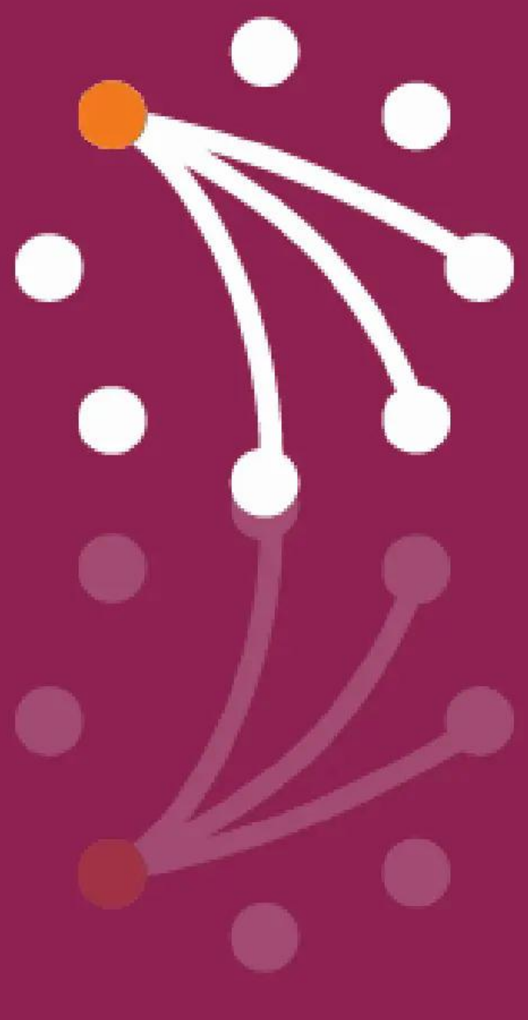
Each blog post is saved on the *MongoDB* database. After pressing the 'Submit' button, your data is automatically passed to the **/presentnewpost** URL where the blog post is saved. Last, you're being redirected to the **/post/postid** URL to see the post. The postid of each post is the string value of its _id key, which is unique to that collection. The Bottle template for getting a list of all available blog posts:

```
<!DOCTYPE html>
<html><head>
<title>List Blog Posts Page!</title>
</head>
<body>
<h1>This page shows all existing blog posts.</h1>
<ul>
%for myX in data:
<%
  link = str(myX['_id'])
%>
<li><a href="/post/{{link}}">{{myX['title']}}</a></li>
%end
</ul>
</body></html>
```

The following trick allows you to execute Python code inside a Bottle template file:

```
<%
  link = str(myX['_id'])
%>
```

This code is used for creating the correct link for each blog post. It would be interesting to see how your data is stored on the *MongoDB* collections. As *MongoDB* is schemaless, you can add new keys to the blogposts collection at any time without any downtime. ■



riak

Everything you wanted to know about NoSQL, but were too afraid to ask. Discover why admins love this high-speed system

The world of databases moves slowly (but perhaps not when transacting), so when a revolution hits it can take decades for the repercussions to be felt. Coined back in 1998, NoSQL are databases that started life not using the then-standard query language SQL. But more revolutionary the actual database designs themselves, moved away from the standard relational model altogether for speed and ease of design.

As you might expect by the name, even though NoSQL databases were originally designed not to use SQL, they can now; but generally, they use various different query languages instead. While these originally might have appeared in 1998, NoSQL didn't actually gain any significany prevalence among DBAs until the late Noughties when it was adopted as a rallying Twitter hashtag for a group of non-relational distributed database projects that were after something small and unique for their various implementations.

If you are wondering whether or not it's worth considering NoSQL databases, you should be aware that according to DB-Engines Ranking (**https://db-engines.com/en/ranking**), *MongoDB*, a popular NoSQL database, is currently the fifth most popular after Oracle, *MySQL*, Microsoft SQL Server and *PostgreSQL* – and even Oracle has a NoSQL version of its famous database.

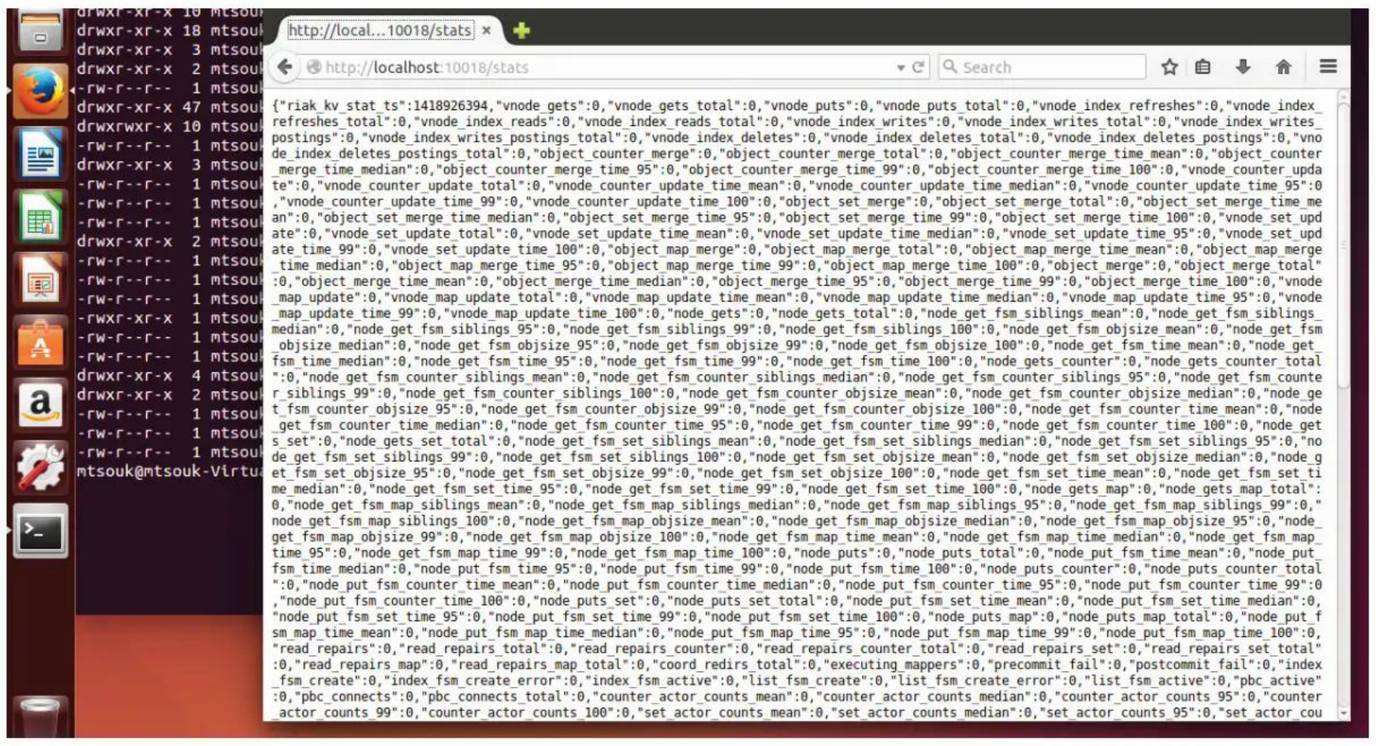
The problem with relational databases is that in order to store complex information you have to deconstruct it into bits and fields, and store it in lots of different tables. Likewise, in order to restore the data, you have to retrieve all those bits and fields and put them back together. Neither of those two tasks is efficient, particularly if you have a big and busy website that's storing and querying data all the time.

> **"NoSQL DBs are designed for the web and don't support joins and complex transactions…"**

The next logical step is to use many machines to run your database, but that also creates a problem because relational databases were originally designed to run as single-node systems. So, large companies, such as Google and Amazon, developed their own database systems, *Bigtable* and *Dynamo* respectively, that were quite different from traditional relational database systems, and which inspired the NoSQL movement.

It's quite difficult to define what a NoSQL database is but there are common characteristics among NoSQL databases: they are non-relational; open source (although not always); schema-less; easy to be distributed on many machines (again, not always) and trying to serve data from the 21st century web culture.

So, NoSQL databases are designed for the web and don't support joins, complex transactions and other features of the SQL language. Their terminology is also a little different, but lets dive into the details.



› Every Riak server has a web interface. In this case, we're accessing the server statistics, using http://localhost:10018/stats/ The port number and IP are defined in riak.conf

The main advantage of a NoSQL database is that they are suited to and efficient for big data and real-time web applications. They also offer easy scalability, and enable you to implement high availability painlessly. They are also generally easier to administer, set up and run, and they can store complex objects. Additionally, it's easier to develop applications for and with NoSQL databases. Their schema can change easily without any downtime because, in reality, they have no schema. Most of them, with the exception of Oracle NoSQL, are open source projects.

Key disadvantages of NoSQL databases include the fact that they require a totally new way of thinking and that you still need a DBA on large and/or critical projects. If your company needs to use both SQL and NoSQL databases, you will have two entirely different systems to program and administer and therefore will need even more people. Being relatively new, they are not as mature as relational databases; therefore choosing a NoSQL database for a critical problem may not always be the safest solution, but this will not be a problem in a couple of years. The last disadvantage is the fact that although they look like they have no schema, you will need to assume an implicit schema in order to do some serious work with your data. This isn't unexpected because as long as you are working with data, you cannot get away with having a schema, even an informal one.

There are several kinds of NoSQL database each of them being good in one or more areas but not all. You can categorise NoSQL databases according to their data model:

» **Document** This is a very common data model. It thinks of the database as a big storage for documents where each document is a multipart data structure that's usually represented in forms of JSON. You can still store documents in any format you want. *MongoDB*, *CouchDB* and *RavenDB* are representative document NoSQL databases.

» **Key-Value** This is also a common data model that's similar to the hash map data structure, where you have a key and you ask the database to return the value stored for that particular key. The value can be anything from a single number to a whole document. The database knows nothing about the stored data. Examples of key-value NoSQL databases include *Riak*, *Redis* and Project Voldemort.

» **Column-family** This is a rather complex data model. You have a 'row key' that enables you to store and access multiple column families. Each column family is a combination of columns that fit together. Row keys must be unique within a column family. The data model might be more complicated than the others but it results in faster retrieval times. Examples of column-family NoSQL databases include *Cassandra* and *Apache HBase*.

» **Graph** This model is totally different from the other three as it is based on the Graph structure. As a logical consequence Graph NoSQL databases handle hierarchies and relationships between things very well; doing similar things with a relational database would be an extremely challenging and slow task. *Neo4j* is a graph NoSQL database. For this article, we'll be using *Riak* as our NoSQL database test case.

Installing Riak

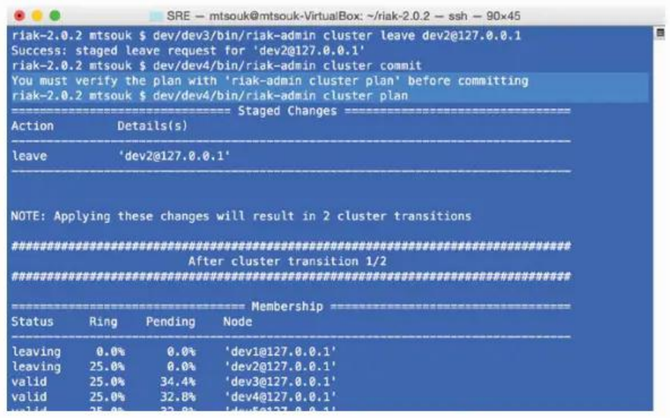
The first thing you should know before installing *Riak* is that you need Erlang on your system. The best way to install *Riak* is by compiling it from source because you have better control and a totally autonomous build of *Riak*. Follow the next steps:

```
$ wget http://s3.amazonaws.com/downloads.basho.com/riak/2.0/2.0.2/riak-2.0.2.tar.gz
$ tar zxvf riak-2.0.2.tar.gz
$ cd riak-2.0.2
$ make rel
```

Alternatively, you can get the *Riak* source code from GitHub and compile it as before:

```
$ git clone git://github.com/basho/riak.git
$ cd riak
$ make rel
```

Both ways should work without any particular problems; we used the first way to compile *Riak*. After successfully compiling *Riak*, you can find its main binary files inside the **./rel/riak/bin** directory. In the same directory that you build Riak, you can run **make devrel** and get ready to run eight Riak »



› Generating a *Riak* cluster with five nodes is pretty easy, see the next page for details

Map and Reduce

MapReduce is an advanced querying technique and a tool for data aggregation used in NoSQL databases. It's an alternative technique for querying a database that differs from the usual declarative querying techniques. You give instructions to the database on how to find the data you are looking for and MapReduce tries to find the data. (See the top of the next page for a simple example of how MapReduce works.) Using MapReduce can be very tricky sometimes. Nevertheless, it enables you to create queries that would have been extremely challenging to create using SQL.

Once you understand the MapReduce process and practice it, you will find it both very reliable and handy. The MapReduce solution takes more implementation time but it can expand better than an SQL solution. It provides some flexibility that's not currently available in the aggregation pipeline. The tricky thing is deciding whether or not the MapReduce technique is appropriate for the specific problem you are trying to solve. This kind of knowledge comes with experience!

» databases that we will use as example servers. This is the main reason to get the *Riak* source code and compile it for yourself.

Before we continue with the rest of the article, we need to introduce you to some terms. First, a *Riak* node is analogous to a physical server. A *Riak* cluster is a 160-bit integer space which is divided into equally-sized partitions. Partitions are, in turn, the spaces into which a *Riak* cluster is divided. Each vnode in *Riak* is responsible for a partition. Vnodes coordinate requests for the partitions they control. A *Riak* cluster can have many nodes that reside on the same or different physical machines. A ring is a 160-bit integer space equally divided into partitions, and a bucket is a namespace for data stored in *Riak*. Internally, *Riak* computes a 160-bit binary hash of each bucket/key pair and maps this value to a position on an ordered ring of all such values.

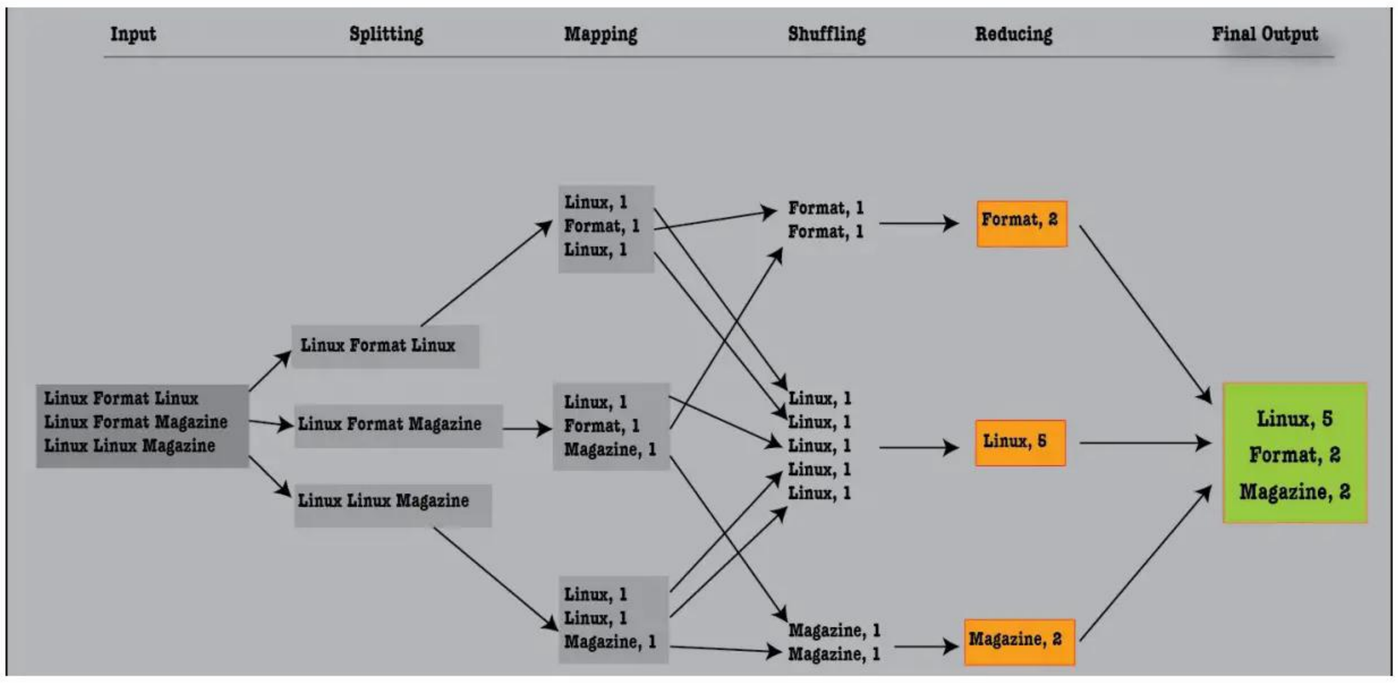
As you will see later in this article, any client interface to *Riak* interacts with objects in terms of the bucket and key in which a value is stored, as well as the bucket type that is used to set the properties of the bucket.

The default mode of operation for *Riak* is to work as a cluster consisting of multiple nodes. *Riak* nodes are not clones of one another by default.

You can start three example *Riak* database servers – you don't have to start all eight Riak servers – by executing the next commands:

```
$ ./dev/dev1/bin/riak start
$ ./dev/dev2/bin/riak start
$ ./dev/dev3/bin/riak start
$ ./dev/dev1/bin/riak start
Node is already running!
```

Each *Riak* server offers a web interface (see the previous page for an example of what you will see after connecting to a Riak server). The port number and the server IP address are defined inside the **riak.conf** file. This is a plain text file that you can edit. This command reveals the IP and the port number that each Riak server listens to:



› A MapReduce example. It may look simplistic but MapReduce is a very powerful technique. Attempting the same with SQL would be extremely difficult

```
$ grep listener.http.internal `find ./dev -name
riak.conf`
./dev/dev2/etc/riak.conf:listener.http.internal
= 127.0.0.1:10028
./dev/dev1/etc/riak.conf:listener.http.internal
= 127.0.0.1:10018
./dev/dev3/etc/riak.conf:listener.http.internal
= 127.0.0.1:10038
```

And so on... Every node in *Riak* has a name associated with it. You can change the name by changing the nodename variable of the **riak.conf** file. The first server (dev1) uses port number 10018, the second *Riak* server (dev2) uses port number 10028 and the third (dev3) uses port number 10038. *Riak* versions prior to 2.0 used a configuration file called **app.config** which has been replaced by **riak.conf**.

The easiest way of finding out if a *Riak* node is up or down is via the **curl** command and the web interface of the node to ping it:

```
$ curl http://localhost:10018/ping
OK
$ curl http://localhost:10038/ping
curl: (7) Failed to connect to localhost port
10038: Connection refused
```

Alternatively, you can use the following:

```
$ ./dev/dev1/bin/riak ping
pong
$ ./dev/dev6/bin/riak ping
Node 'dev6@127.0.0.1' not responding to
pings.
```

The advantage of the 'curl way' is that you can run it from a remote machine – provided that the *Riak* server also listens to an external IP address – without having to login to the machine that runs the *Riak* node. You can stop the dev1 *Riak* server by executing the **./dev/dev1/bin/riak stop** command.

Riak uses epmd – the Erlang Port Mapper server – which plays a crucial part in the whole *Riak* operation. The epmd process starts automatically by the **erl** command if the node is to be distributed and there's no running instance present. The process enables *Riak* nodes to find each other. It's an extremely lightweight and harmless process that can continue to run even after all *Riak* nodes have stopped. You may kill it manually after you stop all *Riak* nodes, but this isn't compulsory. This command lists all names registered with the currently running epmd process:

```
$ epmd -names
epmd: up and running on port 4369 with data:
name dev3 at port 49136
name dev1 at port 55224
name dev2 at port 48829
```

Storing and retrieving data

You can connect to *Riak* dev1 server and store a document using the web interface:

```
$ curl -v -X PUT http://127.0.0.1:10018/riak/
LXF/test -H "Content-Type: text/html" -d
"<html><body><h1>This is a test.</h1></
body></html>"
```

Riak benchmarking

Basho (**http://basho.com**) offers a benchmarking tool for *Riak* written in Erlang. You can get and install it with:

```
$ git clone git://github.com/basho/basho_
bench.git
$ cd basho_bench
$ make
```

You should then run **./basho_bench myconfig.config** to get the tool collecting data, and either create a **myconfig.config** yourself or modify an existing config. Existing files reside in the **examples** directory. We used the **examples/basho_bench_ets.config** file as a starting point and added the **{riakclient_nodes, ['dev1@127.0.0.1, 'dev2@127.0.0.1']}."** line. *Basho Bench* creates one Stats process and workers based on what's defined in the concurrent configuration setting in **myconfig.config** file. As soon as these processes are created and initialised, **Basho Bench** sends a **run** command to all worker processes and this initiates the testing. The Stats process is notified every time an operation completes. It also gets the elapsed time of the completed operation and stores it in a histogram.

All the results are inside the **tests** directory. The latest results can be found using the **./tests/current/** soft link. To generate a graph against the current results, run **make results**. See the next page for sample output.

What is actually stored in the **/riak/LXF** test location is what follows the **-d** option. When you successfully insert a new value, *Riak* will return a 204 HTTP code. As you already know, *Riak* is a key-value store, therefore in order to retrieve a value you need to provide a key to *Riak*. You can connect to *Riak* dev1 server and ask the previously stored document by going to the **http://127.0.0.1:10018/riak/LXF/test** URL. Every URL follows the **http://SERVER:PORT/riak/BUCKET/KEY** pattern. The following command returns the list of available buckets:

```
$ curl -i 'http://127.0.0.1:10018/riak?buckets=true'
HTTP/1.1 200 OK
Vary: Accept-Encoding
Server: MochiWeb/1.1 WebMachine/1.10.5 (jokes are better explained)
Date: Fri, 19 Dec 2014 21:13:37 GMT
Content-Type: application/json
Content-Length: 33

{"buckets":["LXF","linuxformat"]}
```

The following command returns the list of keys in a bucket:

```
$ curl 'http://127.0.0.1:10018/buckets/LXF/keys?keys=true'
{"keys":["test2","test","test3"]}
```

Most of the times, you are going to use a script written in a programming language to access a *Riak* database. The following is a Python script that connects to a *Riak* database, stores and retrieves a document:

```
import riak
# Connect to the cluster
client = riak.RiakClient(pb_port=10017, protocol='pbc')
# The name of the bucket
bucket = client.bucket('python')
# "myData" is the name of the Key that will be used
aRecord = bucket.new('myData', data={
 'Name': "Mihalis",
 'Surname': "Tsoukalos"
 })
# Save the record
aRecord.store()
# Define the key for the record to retrieve
myRecord = bucket.get('myData')
# Retrieve the record!
dictRecord = myRecord.data
# Now print it to see if all this actually worked.
print dictRecord
$ python myRiak.py
{u'Surname': u'Tsoukalos', u'Name': u'Mihalis'}
```

The pb_port value of 10017 is defined in the **./dev/dev1/etc/riak.conf** file using the listener.protobuf.internal parameter. This is the Protocol Buffers port that is used for connecting to the *Riak* Cluster.

Due to the flexibility in the way that a NoSQL database stores data, inserting, querying and updating a NoSQL database is more complex than a database that uses SQL.

Generating a Riak cluster

Creating and manipulating clusters in *Riak* is relatively easy with the help of the **riak-admin** command. If you try to add a node that's not already running to a cluster, you will fail with the following error message:

```
$ dev/dev2/bin/riak-admin cluster join dev1@127.0.0.1
Node is not running!
$ ./dev/dev2/bin/riak start
$ dev/dev2/bin/riak-admin cluster join dev1@127.0.0.1
Success: staged join request for 'dev2@127.0.0.1' to 'dev1@127.0.0.1'
$ dev/dev2/bin/riak-admin cluster join dev1@127.0.0.1
Failed: This node is already a member of a cluster
```

Similarly, if you try to join a node to itself, you will get an error message:

```
$ dev/dev1/bin/riak-admin cluster join dev1@127.0.0.1
Failed: This node cannot join itself in a cluster
```

The following command shows the members of an existing cluster:

```
$ dev/dev2/bin/riak-admin status | grep members
ring_members : ['dev1@127.0.0.1','dev2@127.0.0.1']
$ dev/dev1/bin/riak-admin status | grep members
ring_members : ['dev1@127.0.0.1','dev2@127.0.0.1']
$ dev/dev3/bin/riak-admin status | grep members
Node is not running!
```

Another useful command that shows the status of the nodes is the following:

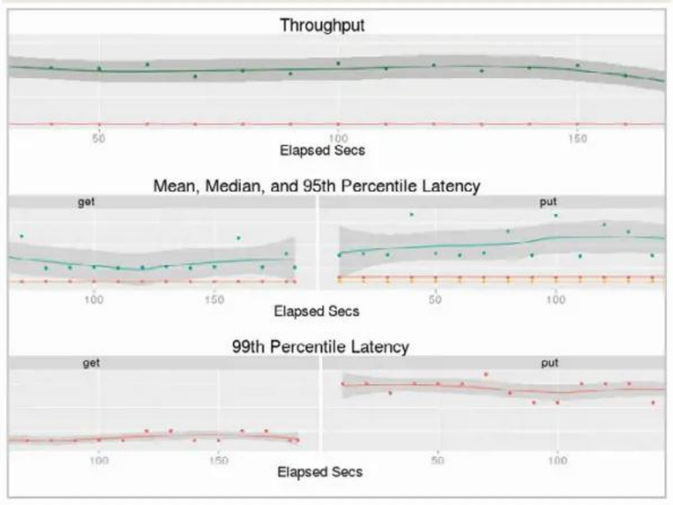
```
$ ./dev/dev1/bin/riak-admin member-status
```

The joining status is a temporary status and will become valid when all changes that are waiting in a queue will be applied and committed. If you want to force changes, you should execute the **riak-admin cluster commit** command.

If you run the **riak-admin member-status** command again you will see the new status of the dev3 node, and the **riak-admin cluster plan** command displays the changes that are about to be applied.

For a node to actually leave the cluster (see bottom of p111, to see what an interaction with a cluster of five nodes looks like), you must first review the changes using the **riak-admin cluster plan** command and then commit them with **riak-admin cluster commit**.

So far, you won't have seen any security when interacting with a *Riak* database. Nevertheless, *Riak* supports users and passwords. You can find a lot more information on how *Riak* deals with authentication and authorisation at **http://bit.ly/RiakDocsAuthz**. ■



› ***Riak* offers a benchmarking tool called *Basho Bench*. The graph is produced with R**

Data consistency

Data consistency in databases is critical. ACID (Atomicity, Consistency, Isolation and Durability) is a set of properties that guarantee that database transactions perform reliably. Atomicity means that when you do something to change a database, the change should work or fail as a whole. Isolation means that if other things are taking place at the same time on the same data, they should not be able to see half-finished data. Durability refers to the guarantee that once the user has been notified of the success of a transaction, the transaction will persist, and won't be undone even if the hardware or the software crashes afterwards.

Graph databases perform ACID transactions by default, which is a good thing. On the other hand, not every problem needs 'perfect' ACID compliance. *MongoDB* is ACID-compliant at the single document level, but it doesn't support multiple-document updates that can be rolled back. Sometimes, you may be OK with losing a transaction or having your DB in an inconsistent state temporarily in exchange for speed.

Carefully check the characteristics of a NoSQL database and decide if it fits your needs. Nevertheless, if data consistency is absolutely critical, you can always implement it in code if it's not fully supported by your NoSQL DB. Keep in mind that this might be non-trivial especially on distributed environments.

Redis: NoSQL data stores

It's time to explain all you the essentials you'll need to know about getting started with Redis, the in-memory data structure store

Redis (REmote Dictionary Service) is a *NoSQL* key-value data store, or more precisely a data structure server. Although *Redis* is similar to *Memcached* in functionality, it differs from *Memcached* because *Redis* is persistent, which means that data is saved on your hard disk, and supports more data types. *Redis* stores its entire data set in RAM, so it is extremely fast. The latest major version of *Redis* has support for clustering, which allows *Redis* to scale better. In this tutorial, we'll look at *Redis* installation and usage as well as *Redis* data types. Then, we'll examine an example in Python and talk about administering a *Redis* database. The best place for learning more about Redis is **http://redis.io**.

Installing Redis

The latest stable version of Redis is 3.2.0. If your Linux system has a ready-to-install package, we'd highly recommended using it to install *Redis*; otherwise, you can follow the instructions at **http://redis.io/download**. You can install *Redis* on Ubuntu using the following command:

```
$ sudo apt-get install redis-server redis-tools
```

To start the *Redis* server use the following command:

```
$ sudo /etc/init.d/redis-server start
```

Starting redis-server: `redis-server` .

```
$ ps ax | grep redis | grep -v grep
 2779 ?        Ssl    0:00 /usr/bin/redis-server 127.0.0.1:6379
```

As you can see from the output of `ps` , *Redis* listens to port number 6379 using the **127.0.0.1** IP address. You can check that everything works as expected using `redis-cli` :

```
$ redis-cli ping
PONG
```

If the answer is `PONG` then everything is OK. However, if the answer is similar to the following, then *Redis* client cannot connect to the *Redis* server:

```
$ redis-cli ping
Could not connect to Redis at 127.0.0.1:6379: Connection refused
```

Stopping the Redis server can be done as follows:

```
$ sudo /etc/init.d/redis-server stop
Stopping redis-server: redis-server.
```

The configuration files of *Redis* can be found inside **/etc** but you will not have to deal with them very often:

```
$ ls -l /etc/redis
total 56
drwxr-xr-x  2 root root  4096 Jun  6 11:00 ./
drwxr-xr-x 137 root root 12288 Jun  6 11:00 ../
-rw-r--r--  1 root root 30176 Jan 14  2014 redis.conf
-rw-r--r--  1 root root  5662 Jan 13  2014 sentinel.conf
```

Finally, finding what version of *Redis* you are using is as easy as executing the following:

```
$ redis-server --version
Redis server v=2.8.4 sha=00000000:0 malloc=jemalloc-3.4.1 bits=64 build=a44a05d76f06a5d9
```

The basics

A *Redis* database doesn't have a schema; therefore you cannot create a table as if you were using *MySQL* or any other relational database. However, you can create as many key-value pairs as you want! Additionally, *Redis* doesn't support the SQL language.

Redis comes with its own client, named *redis-cli*. If you execute `redis-cli` without any command-line arguments, you will go to the *Redis* interactive shell, which shows the IP address and the port number of the *Redis* server process you are connected to:

```
$ redis-cli
127.0.0.1:6379>
```

However, if you give a valid *Redis* command as a command-line argument to *redis-cli*, it will execute the command and put you back to the Linux shell:

```
$ redis-cli TIME
1) "1465797399"
2) "913202"
```

You can also get the current time in UNIX timestamp format as follows:

```
127.0.0.1:6379> TIME
1) "1465797446"
2) "201615"
```

Quite strangely, the `TIME` command returns two things: the UNIX time in seconds and the number of microseconds elapsed in the current second. If you want to find information about a command, you can do it as follows:

```
$ redis-cli help @generic
$ redis-cli
127.0.0.1:6379> help LPUSH
```

The first command (*above*) finds help about a given category whereas the second command finds help about a specific command. The supported categories are `@generic` , `@list` , `@set` , `@sorted_set` , `@hash` , `@pubsub` , `@transactions` , `@connection` , `@server` , `@scripting` and `@hyperloglog` . *Redis* supports an unusual and somewhat cryptic set of commands that you need to get used to.

Quick tip

There are various applications that can help you use *Redis* from a web browser, such as *Redsmin* (**www.redsmin.com**), *Redis Desktop Manager* (**http://redisdesktop.com**) and *redis-browser* (**http://bit.ly/RedisBrowser**).

What makes Redis special?

Redis has two unique characteristics: it runs in memory and it supports many unusual data types. One of the benefits of *Redis* running in memory is that it's really fast!

This can be proven by the fact that most operations on *Redis* data types have a time complexity of O(1), which is a constant time regardless of the amount of data stored in the *Redis* database.

Additionally, the data types supported by *Redis* offer unique design capabilities to the developer. *Redis* is better than Memcached because *Redis* offers built-in persistence. *Redis* requires you to think in a different and a more contemporary way when designing applications, which can give a new perspective to your applications, and adds support for clusters, which makes it scalable. Finally, *Redis* uses a simple configuration file, so instead of dealing with the various configuration options you can start developing your application.

If you want to find out where *Redis* keeps its data files, you can execute the following command:

```
$ redis-cli config get dir
1) "dir"
2) "/var/lib/redis"
```

If the *Redis* server isn't running, you can find the data directory of *Redis* by looking at its configuration file:

```
$ grep dir /etc/redis/redis.conf | grep -v '^#'
dir /var/lib/redis
```

You can insert a new key-value pair to *Redis* as follows:

```
127.0.0.1:6379> SET aKey "Mihalis"
OK
```

You can retrieve the value of aKey using GET:

```
127.0.0.1:6379> GET aKey
"Mihalis"
```

All *Redis* commands are case insensitive, which means that the `GET` command can also be entered as `get`, `Get`, `gET` etc. However, it's good practice to type *Redis* commands in uppercase to differentiate them from the rest of the input.

Data types

Redis supports many different data types: strings, lists, sets, sorted sets, hashes, hyperloglogs and bitmaps.

The string is the most common data type in *Redis*. The good thing is that it's safe to store binary data on a string variable. The maximum size of a string variable is 512MB. A list is a list of strings, sorted by insertion order. It's possible to insert additional elements into an existing list on the head or on the tail of it, using `LPUSH` and `RPUSH` respectively. A new list is created by using `LPUSH` or `RPUSH` on an empty list. As you will see, a List can contain duplicate values.

A set is an unordered collection of strings. *Redis* sets can't have repeated members. *Redis* enables you to perform unions, intersections and differences of existing sets. The `SADD` command enables you to add new elements to a set. A sorted set has a score associated with each one of its members that's used for ordering the members of the set, starting from the smallest score up to the biggest score, and this is the most advanced *Redis* data type.

As you might expect, a *Redis* hash is a mapping between string fields and string values, which makes it the perfect candidate for storing objects. Finally, both bitmaps and hyperLogLogs are data types that are based on the String base type with different semantics.

You can create a new list in the following way:

```
127.0.0.1:6379> LPUSH aList "123"
(integer) 1
127.0.0.1:6379> LPUSH aList "Mihalis"
(integer) 2
127.0.0.1:6379> RPUSH aList "Tsoukalos"
(integer) 3
127.0.0.1:6379> RPUSH aList Tsoukalos
(integer) 4
```

The return value of `RPUSH` and `LPUSH` is the total number of list elements. You can store a set as follows:

```
127.0.0.1:6379> SADD aSet 32 21
(integer) 2
127.0.0.1:6379> SADD aSet 11 21
(integer) 1
```

The first command added both `32` and `21` to the set, which is why it returned the value `2`.

The second command added only `11` to the set because `21` is already a member of the set and therefore returned the value `1`.

Similarly, you can store a hash using the following: »

Quick tip

You can find the source code of *Redis* at **https://github.com/antirez/redis**. You can also find Redis documentation at **http://redis.io/documentation**.

```
127.0.0.1:6379> LPUSH aList foo
(integer) 8
127.0.0.1:6379> RPUSH aList foo1
(integer) 9
127.0.0.1:6379> LRANGE aList 0 -1
1) "foo"
2) "321"
3) "123"
4) "Mihalis"
5) "123"
6) "Tsoukalos"
7) "Tsoukalos"
8) "Tsoukalos"
9) "foo1"
127.0.0.1:6379> SADD aSet 0 1 2 3 2 1
(integer) 3
127.0.0.1:6379> SMEMBERS aSet
1) "0"
2) "1"
3) "2"
4) "3"
5) "21"
6) "32"
127.0.0.1:6379> SADD aSet 0 1 2 3 2 1
(integer) 0
127.0.0.1:6379> SMEMBERS aSet
1) "0"
2) "1"
3) "2"
4) "3"
5) "21"
6) "32"
127.0.0.1:6379> HMSET anotherHash one 1 two 2 three 3
OK
127.0.0.1:6379> HGETALL anotherHash
1) "one"
2) "1"
3) "two"
4) "2"
```

› Some exciting insert, update and delete operations using various Redis data types. No need to thank us!

```
mtsouk@mtsouk-ubuntu:~/code/redis$ ls -al
total 24
drwxrwxr-x  2 mtsouk mtsouk 4096 Jun 13 13:34 .
drwxrwxr-x 11 mtsouk mtsouk 4096 Jun 10 16:41 ..
-rw-rw-r--  1 mtsouk mtsouk  251 Jun 10 20:13 connect.py
-rw-rw-r--  1 mtsouk mtsouk  282 Jun 13 13:34 delete.py
-rw-rw-r--  1 mtsouk mtsouk  523 Jun 13 12:03 insert.py
-rw-rw-r--  1 mtsouk mtsouk  299 Jun 13 12:07 query.py
mtsouk@mtsouk-ubuntu:~/code/redis$ python ./query.py 132
Telephone not found!
mtsouk@mtsouk-ubuntu:~/code/redis$ python ./query.py 123
Telephone not found!
mtsouk@mtsouk-ubuntu:~/code/redis$ python ./insert.py 123 Mihalis Tsoukalos
mtsouk@mtsouk-ubuntu:~/code/redis$ python ./query.py 123
['123', 'Mihalis', 'Tsoukalos']
mtsouk@mtsouk-ubuntu:~/code/redis$ python ./insert.py 123 Mike Tsoukalos
mtsouk@mtsouk-ubuntu:~/code/redis$ python ./query.py 123
['123', 'Mihalis', 'Tsoukalos', '123', 'Mike', 'Tsoukalos']
mtsouk@mtsouk-ubuntu:~/code/redis$ python ./delete.py 123
mtsouk@mtsouk-ubuntu:~/code/redis$ python ./query.py 123
Telephone not found!
mtsouk@mtsouk-ubuntu:~/code/redis$
```

› Here we have insert.py, query.py and delete.py, and it demonstrates how they interact with a *Redis* database server

```
127.0.0.1:6379> HMSET myHash:1 CC 2000 Turbo 1 make VW colour Black
OK
```

Retrieving all previously stored values can be done as:

```
127.0.0.1:6379> LRANGE aList 0 -1
1) "Mihalis"
2) "123"
3) "Tsoukalos"
4) "Tsoukalos"
127.0.0.1:6379> SMEMBERS aSet
1) "11"
2) "21"
3) "32"
127.0.0.1:6379> HGETALL myHash:1
1) "CC"
2) "2000"
3) "Turbo"
4) "1"
5) "make"
6) “VW”
7) "colour"
8) "Black"
```

The `LRANGE` command allows you to print certain elements of a list. The first parameter is the name of the list, the second parameter is the index of the first element you want to display and the third parameter is the last element you want to display. A value of `-1` denotes the last element of the list. You can print all elements of a set using the `SMEMBERS` command—you can see that, for instance, number `21` appears only once. Similarly, you can print all elements of a hash using the `HGETALL` command. Please note that `myHash:1` is different from `myHash:2`, so this technique gives you a way of having multiple hashes with similar names. However, since you cannot query on the properties of a hash key, you will need to build and maintain indexes on your own in order to perform queries, which might complicate things.

Knowing the name of a key, you can find its type using the `TYPE` command:

```
127.0.0.1:6379> TYPE foo
string
127.0.0.1:6379> TYPE myHash:1
hash
```

As you'll hopefully begin to see, *Redis* is pretty flexible about what kind of data you can store (pictured on the previous page are various insert, update and delete operations with various data types as performed on a Redis database) and you can find more about Redis data types at **http://redis.io/topics/data-types-intro**.

Python and Redis

In order to let Python talk to a *Redis* server, you should first install a Python package:

```
$ sudo pip install redis
```

After installing the **redis** package, you should execute the following Python code to make sure that you can connect to the server properly:

```
import redis
r = redis.StrictRedis(host='localhost', port=6379, db=0)
myVal = r.set('One', 'Two')

if myVal == True:
        print “Set operation was successful.”
    else:
        print “Operation failed!”

myVal = r.get('One')
print “The value of One is: " + myVal
```

Executing **connect.py** will generate the following output:

```
$ python connect.py
Set operation was successful.
The value of One is: Two
```

Now that you know how to connect, insert and query a *Redis* database using Python, it's time to create a small application. The application will be a basic phone book application. Each entry will have a name, a surname and a telephone number. The allowed operations will be insert, query and delete, supported by three Python scripts called **insert.py**, **query.py** and **delete.py** respectively. Please note that the unique key for each 'entry' is the telephone number. (You can also see the three scripts in action, above).

Three Python scripts

The contents of **insert.py** is:

```
import redis
import sys
r = redis.StrictRedis(host='localhost', port=6379, db=0)
    if len(sys.argv) != 4:
        print 'Wrong Number of arguments'
        print 'insert.py phone name surname'
        sys.exit(1)

telephone = sys.argv[1]
name = sys.argv[2]
surname = sys.argv[3]

myTel = r.rpush(telephone, telephone)
myName = r.rpush(telephone, name)
mySur = r.rpush(telephone, surname)
```

The Python code of **query.py** is:

```
import redis
import sys
if len(sys.argv) != 2:
        print 'Wrong Number of arguments'
        print 'query.py phone'
        sys.exit(1)
```

When not to use Redis

The people who develop *Redis* strongly suggest running production *Redis* servers on Linux because the Linux platform is where they do most of the *Redis* development and testing. However, *Redis* also runs on macOS, FreeBSD and OpenBSD.

As you already know, *Redis* can replace *Memcached* and can be used for caching operations. *Redis* is also really good at storing frequently-updated real-time data. The interesting thing is that it handles writes faster than it handles reads, so if you know you are going to have lots of write operations, you should consider using *Redis*. Additionally, the data types supported by Redis will make your life easier if they suit your needs.

However, if a traditional database, such as *MySQL* or *PostgreSQL* does your job, don't consider replacing it with *Redis*. The reason being that Redis doesn't have support for sharding, so if you need sharding you might want to consider using *MongoDB* instead. As you can easily understand, you have to have enough RAM on your Linux machine to hold the entire data set.

Finally, if you need to perform many queries on your data, *Redis* may not be a good choice because it doesn't support traditional query operations and requires a complete redesign of your application.

```
r = redis.StrictRedis(host='localhost', port=6379, db=0)
telephone = sys.argv[1]
aList = r.lrange(telephone, 0, -1)

if aList:
        print aList
else:
        print 'Telephone not found!'
```

The **delete.py** script is similar to **query.py** but it uses the `r.delete()` function to delete an entry. All three Python scripts have a common preamble, which is the code that's needed to connect to the *Redis* server and check that the user gave the correct number of command line arguments. **insert.py** uses the `r.rpush()` function, **query.py** uses the `r.lrange()` function and **delete.py** uses the `r.delete()` function.

Redis admin

When using a *Redis* database, you will need to be able to administer it and perform various important tasks. You can change the number of available databases using the 'databases' setting in the Redis configuration file:

```
$ grep databases /etc/redis/redis.conf | grep -v '^#'
databases 16
```

You can get the same information using the Redis client:

```
127.0.0.1:6379> CONFIG GET databases
1) "databases"
2) "16"
```

As you can see, the default *Redis* configuration creates `16` databases that you can access as 0, 1, ..., 15. You can select the database you want with the help of the `SELECT` command. However, note that using multiple databases in one *Redis* process is not recommended. *Redis* offers no way of finding the index number of the current database.

The next command shows all the keys that exist in the current database:

```
127.0.0.1:6379> KEYS *
1) "One"
2) "foo"
127.0.0.1:6379> SELECT 1
OK
127.0.0.1:6379[1]> KEYS *
(empty list or set)
```

As you can see, the initial database has two keys whereas the database with index number `1` has no keys; in other words, the database with index 1 is empty. Please note that executing `KEYS *` in production environments is highly discouraged because it can return too much output. Filtering the output is highly recommended: eg, the `KEYS a*` command will return all keys that begin with `a`.

The following command shows useful statistics about your Redis server process:

```
$ redis-cli INFO
```

You can see a part of the previous output (*pictured below*) processing the output of the `INFO` command using a utility such as *grep* will help you find what you are looking for. You can find more information about the meaning of each field at **http://redis.io/commands/INFO**.

You can back up a *Redis* server by making a copy of its data file. It's safer to copy a *Redis* data file when *Redis* isn't running because *Redis* doesn't save its data immediately but does so from time to time. Hence, if you want all your data to persist and survive a hardware failure, you should make sure that the `SAVE` command is manually executed quite frequently. You can restore a backup by just copying the data file to the desired location. This means that backing up *Redis* can be easily automated using a script and the *Cron* utility.

The last command shows all of the *Redis* databases that are in use:

```
$ redis-cli INFO | grep ^db
db0:keys=6,expires=0,avg_ttl=0
```

The **http://redis.io/topics/admin** page contains useful Redis setup hints, have fun and enjoy! ■

```
mtsouk@mtsouk-ubuntu:~/code/redis$ redis-cli INFO
# Server
redis_version:2.8.4
redis_git_sha1:00000000
redis_git_dirty:0
redis_build_id:a44a05d76f06a5d9
redis_mode:standalone
os:Linux 3.19.0-59-generic x86_64
arch_bits:64
multiplexing_api:epoll
gcc_version:4.8.2
process_id:1077
run_id:86eb52d9c1b35143d8a963c2c219adc78608dceb
tcp_port:6379
uptime_in_seconds:69558
uptime_in_days:0
hz:10
lru_clock:1860052
config_file:/etc/redis/redis.conf

# Clients
connected_clients:1
client_longest_output_list:0
client_biggest_input_buf:0
blocked_clients:0

# Memory
used_memory:502240
used_memory_human:490.47K
used_memory_rss:3424256
used_memory_peak:501344
used_memory_peak_human:489.59K
used_memory_lua:33792
mem_fragmentation_ratio:6.82
mem_allocator:jemalloc-3.4.1
```

› The INFO command returns information and statistics about the current Redis server process in a plain text format that can be easily read by humans and parsed by command line utilities

CODING ACADEMY

CODING ACADEMY

Do more

Python isn't the only language; try Erlang, Rust, Go, Swift, and R

88 **Program in Erlang: Introduction**
Get to grips with the Erlang language

94 **Erlang: Functions**
Discover functions and basic data types

100 **Rust: modules and cargo**
Get started with the hip new language

104 **Rust: functions and modules**
Go further tackling error handling

108 **Rust: file I/O**
Speed up your transfers

112 **Rust: networking**
Build a TCP-based client/server tool

116 **Rust: concurrency**
How to use Threads

120 **Go: Master Google's new programming language**
Get yourself familiar with the fundamentals

126 **Go: Data types**
Get to grips with Go's composite data types

132 **Go: Explore functions**
Learn how you can develop and use functions

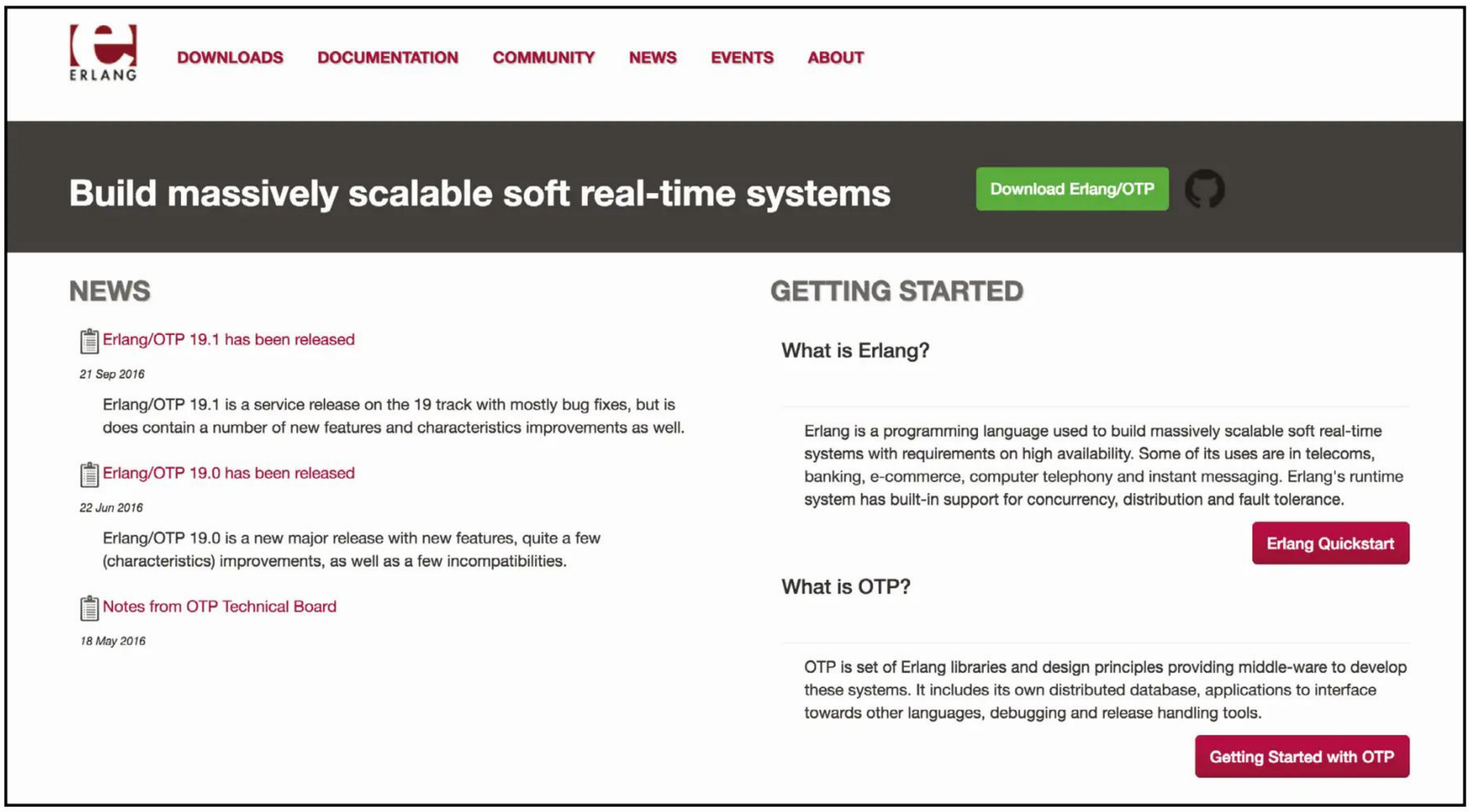
Program in Erlang: Introduction

Learn how to get to grips with the Erlang programming language in the following tutorials

This tutorial will be the first one in a series of tutorials about the Erlang programming language. Erlang is a functional programming language with many unique and interesting characteristics, and was developed at Ericsson in order to serve the switching of telephone equipment. Although it may look like a special-purpose programming language, it is not. Erlang can do many things, but it's most appropriate for developing highly available server processes. In other words, it's better to develop the server side of an application using Erlang than the client side. The purpose of this tutorial is to present you with a quick overview of how Erlang does things by giving you small yet complete code examples. The rest of the tutorials will offer more detailed information about significant Erlang topics, so once you've got the hang of this one, start practicing these more advanced skills!

About Erlang

We can't say that Erlang is going to be the next big thing in programming, because Erlang has been around for more than 20 years! Neither can we say that functional programming is the next big thing in programming, because it has existed for even longer! However, we can say that people are still learning how to take advantage of functional programming languages. An Erlang program should be implemented as a large number of small processes that do simple tasks and communicate with each other using code that has no side effects. Erlang processes interact with each other by exchanging messages without sharing any data, therefore allowing Erlang code to utilise multiple CPU cores without complications. As a result of the above, an Erlang program might run four times faster on a four-core CPU without changing a single line of code!



Getting Erlang

Most Linux distributions do not come with the latest Erlang version. So, if you want to use the latest Erlang version you have two options: downloading a binary package of Erlang or building it from the source code. Building from the source code is relatively difficult so it is better to avoid it when you can; downloading a binary package is a superior solution. Figure 8 shows the process in more detail but you can find more information at **https://www.erlang-solutions.com/resources/download.html**.

If you would like to learn more about Erlang, you will find both *Programming Erlang, Second Edition* by Joe Armstrong from Pragmatic Bookshelf, and *Learn You Some Erlang for Great Good! A Beginner's Guide* by Fred Hébert from No Starch Press very useful. Last, there is *Erlang and OTP in Action* by Eric Merritt, Martin Logan, and Richard Carlsson from Manning Publications, which talks about OTP in more detail. All of these titles will help you to broaden and enhance your Erlang knowledge.

Figure 1

```
-module(practical).
-export([fibo/1, for/2, start/0]).

fibo(0) -> 0;
fibo(1) -> 1;
fibo(N) -> fibo(N - 1) + fibo(N - 2).

for(0,_) ->
    [];
for(N,Term) when N > 0 ->
    io:fwrite("~p: ~p~n", [N, fibo(N)]),
    [Term|for(N-1,Term)].

start() ->
```

This is the Erlang code of practical.erl, which contains a function that calculates numbers that belong to the Fibonacci sequence

About OTP

OTP is the most important part of Erlang. OTP stands for Open Telecom Platform and is a collection of open source libraries and tools. Despite its name, OTP is not just about telecommunications; OTP was designed for developing big projects with big teams and allows you to take all the generic components, put them into libraries, make sure they work fine and reliably, and then reuse that code as often as possible. What the programmer has to do is to deal with things that change from application to application, which mainly is the actual functionality of the application. If you are going to write real world Erlang software, you will eventually have to learn the OTP Framework.

Put simply, OTP allows you to supervise existing Erlang code. In order to supervise an existing module, you will have to write an additional Erlang module but you will not need to make any changes to the original module! Once the supervisor process detects that the supervised process is down, it automatically restarts it.

OTP is the most important and vital part of Erlang. The two next tutorials will talk more about OTP and present interesting code examples, so keep reading!

A small example

You can find the exact version of Erlang you are using as follows, which you might find pretty bizarre:

```
$ erl -eval '{ok, Version} = file:read_
file(filename:join([code:root_dir(), "releases",
erlang:system_info(otp_release), "OTP_VERSION"])),
io:fwrite(Version), halt().' -noshell
```

Quick tip

Functional programming is a programming paradigm where computations are treated as an evaluation of mathematical functions. Functional programming tries to eliminate side effects in order to make the code safer. Other functional programming languages are Elixir, which is based on Erlang, Haskell, F#, Common List and Racket, to name a few. You will learn more about functional programming during the Erlang series of tutorials so stay tuned!

Figure 2

```
3> mtsouk@mail:~$
mtsouk@mail:~$ erl
Erlang/OTP 19 [erts-8.1] [source-4cc2ce3] [64-bit] [async-threads:10] [hipe] [kernel-poll:f
alse]

Eshell V8.1  (abort with ^G)
1> b().
ok
2> h().
1: b()
-> ok
ok
3> Str1 = "Linux User and Developer".
"Linux User and Developer"
4> length(Str1).
24
5> Lenght = length(Str1).
24
6> b().
Lenght = 24
Str1 = "Linux User and Developer"
ok
7> W.
* 1: variable 'W' is unbound
8> date().
{2016,10,28}
9> math1:factorial(10).
** exception error: undefined function math1:factorial/1
10> i().
Pid                   Initial Call                       Heap     Reds Msgs
Registered            Current Function                  Stack
<0.0.0>               otp_ring0:start/2                   376      574    0
init                  init:loop/1                           2
<0.1.0>               erts_code_purger:start/0            233        4    0
erts_code_purger      erts_code_purger:loop/0               3
<0.4.0>               erlang:apply/2                     2586   113133    0
erl_prim_loader       erl_prim_loader:loop/3                5
<0.30.0>              gen_event:init_it/6                 376      226    0
error_logger          gen_event:fetch_msg/5                 8
<0.31.0>              erlang:apply/2                     1598      416    0
application_controlle gen_server:loop/6                     7
```

This shows some of the capabilities of the Erlang shell, including the i() command that displays information about the system

19.1.3

Before presenting a real example, it is time to have a look at the Erlang version of the Hello World! program, saved as hw.erl in this instance.

```
$ cat hw.erl
%% Programmer: Mihalis Tsoukalos
 % hello world program
 % hello world program
 -module(hw).
 -export([helloWorld/0]).
 helloWorld() ->
  io:fwrite("Hello, world!\n").
```

As you can see, comments in Erlang always start with %. You should always have in mind that all Erlang code must be included in modules and that the Hello World! program is no exception. Additionally, all the functions of an Erlang module are private by default, so if you are planning on calling them from outside your module, you should add them in the

Figure 3

```
% Add List items
addList([]) -> 0;
addList(List) -> addList(List, 0).
addList([], Total) -> Total;
addList([Head|Tail], Total) ->  addList(Tail, Total + Head).
```

The Erlang code of the addList() function illustrates how to iterate over the elements of a list in order to add them

Generating Documentation

Erlang offers a handy way to generate documentation that goes near the code. As an example, you can look at Figure 6 for the extra commands used for documenting the code of hw.erl. In order to create the documentation files, you will need to execute the following command from the Erlang shell:

```
5> edoc:files(["hw.erl"], [{dir,
"documentation"}]).
```

The results of the previous command can be seen reflected in the next directory:

```
$ ls -l documentation/
total 28
-rw-r--r-- 1 mtsouk mtsouk   35 Oct 29 20:19
edoc-info
-rw-r--r-- 1 mtsouk mtsouk 2109 Oct 29 20:19
erlang.png
-rw-r--r-- 1 mtsouk mtsouk 1952 Oct 29 20:19
hw.html
-rw-r--r-- 1 mtsouk mtsouk  471 Oct 29 20:19
index.html
-rw-r--r-- 1 mtsouk mtsouk  406 Oct 29 20:19
modules-frame.html
-rw-r--r-- 1 mtsouk mtsouk 1065 Oct 29 20:19
overview-summary.html
-rw-r--r-- 1 mtsouk mtsouk  869 Oct 29 20:19
stylesheet.css
```

Another upcoming tutorial will talk more about writing documentation in Erlang.

Figure 4

```
patternMatch(Number) ->
    case Number of
        10 -> io:fwrite("10!");
        20 -> io:fwrite("20!");
        _ -> io:fwrite("Unknown!")
    end.
```

This naïve function shows the use of the case statement in Erlang, which allows you to match a variable according to some conditions

export() list, as it happens with the helloWorld() function. The number 0 after the function name states the number of arguments that a function has – helloWorld() has no arguments, hence the number 0. After saving the previous code, you can compile it and execute it as follows:

```
$ erlc hw.erl
$ erl -V -noshell -s hw helloWorld -s init stop
Hello, world!
```

The first command compiles the Erlang code using erlc, which is the Erlang compiler, and generates a beam file. BEAM stands for Bodgan's Erlang Abstract Machine, which is a virtual machine responsible for interpreting optimised BEAM code. Each time you compile an Erlang module, the compiler converts its code into a BEAM file. The second command uses erl, which is the binary file for the Erlang interactive shell, and the beam file, which is automatically named hw.beam, in order to execute the helloWorld function from the hw module, which prints the desired message.

The reason you do not enter the Erlang interactive shell can be found in the last part of the command (-s init stop), which tells erl to exit after executing the desired function from the desired module. Alternatively, you can execute a function from the Erlang shell, which you can enter by typing erl with no command line arguments:

```
1> c(hw).
{ok,hw}
2> hw:helloWorld().
Hello, world!
ok
```

The first command compiles the code of the package, whereas the second command executes the desired function. As you can see, each command should end with a full stop/period character (.).

Now it's time to see a more practical example. The Erlang code in practical.erl defines a function that allows you to calculate natural numbers that belong to the Fibonacci sequence. Figure 1 shows the Erlang code of the practical module, where you can also see how to imitate a for loop, which is not defined in Erlang! Additionally, you will be able to see the use of the io:fwrite() function for printing your output on your screen. The io:fwrite() function works similarly to the printf() C function where you need to use control sequences in order to print the desired output. Executing practical.erl generates the following kind of output:

```
$ erlc practical.erl
$ erl -noshell -s practical start -s init stop
5: 5
4: 3
3: 2
2: 1
1: 1
```

Recursion is very frequently used in various functional programming languages because it fits the way functional programming works. However more efficient ways to implement the Fibo() function in Erlang using other Erlang data structures exist – you will learn more about them in a forthcoming tutorial.

In order to save magazine space, all Erlang code that will be developed from now on will be included in a module named lud, which as expected can be found in a file

Figure 5

```
readlines(FileName) ->
    {ok, Device} = file:open(FileName, [read]),
    try process_line(Device)
      after file:close(Device)
    end.

process_line(Device) ->
    io:fwrite(".").
```

The Erlang code of two functions. The readlines() function opens a file for reading whereas the process_line() function can be used for processing the file separately

named lud.erl, so you will only see the implementation of the relevant functions. Please note that both module names and filenames must be written using lowercase letters.

The Erlang shell

The Erlang interactive shell is a powerful component of Erlang that can be used in many different situations. The first thing you should learn about the Erlang shell is how to properly exit it! So, if you want to exit the Erlang shell, you should type "q()." and press the Enter key – do not forget to properly end the previous command with a full stop/period character. If you try to exit the Erlang shell by pressing Ctrl+C you will get a relatively complex message so it is better to avoid this method.

The next output verifies that when you define a new variable in Erlang, you cannot change its value afterwards, which means that Erlang variables are immutable:

```
4> OneVar = 123.
123
5> OneVar = 1234.
** exception error: no match of right hand
side value 1234
```

This happens because Erlang uses the single assignment model, which seems strange at first, but it can save you from nasty bugs. Additionally, the fewer side effects your code has, the most likely it is that it can be executed concurrently on its own process using one of the available CPU cores. Seeing the current value of a variable is easy:

```
8> OneVar.
123
```

Figure 2 shows another interaction with the Erlang shell to make you more comfortable with it – you should try analogous commands on your own.

Variables and numbers

Variables in Erlang begin with a capital letter or an underscore. If you try to declare a variable differently, you will get the following kind of error message:

```
2> aVar = 12.
** exception error: no match of right hand
side value 12
3> _aVar = 12.
12
```

Erlang looks like using some kind of algebra to evaluate its expressions, which is evident in the next example:

```
2> MyVar = 12.4.
12.4
3> MyVar = MyVar + 1.
** exception error: no match of right hand
side value 13.4
4> 12.4 = MyVar.
12.4
```

The second statement is false because when you replace the value of MyVar in the expression, the expression becomes '12.4 = 12.4 + 1', which from a mathematical perspective is false! Similarly, the last expression is true because if you replace the value of MyVar, it becomes '12.4 = 12.4', which is definitely right. After such a correct command Erlang returns a value, which is this case is the value of the variable.
Getting to know Erlang error messages will definitely help you understand the idiosyncrasies of Erlang quicker, so do not underestimate error messages!

Erlang uses two kinds of numbers, integers and floats. However, keep in mind that when defining floats, you should always have a number to the left of the decimal point. A forthcoming tutorial will talk about the various data types of Erlang in more detail.

Iterating in Erlang

Look at the following code as executed in the Erlang shell:

```
6> L1 = [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5].
[-5,-4,-3,-2,-1,0,1,2,3,4,5]
7> c(lud).
{ok,lud}
8> lud:addList(L1).
0
9> lud:addList([2, 10, 12, 22]).
46
```

The code of addList() shows how to iterate over the elements of a list – lists play a key role in Erlang – in order to find out the summary of all list elements. The L1 list is defined inside the shell but you can also give the list directly to the function without having to define a list variable first. You can also see

Figure 7

```
writeFibo(Filename) ->
    {ok, FILE} = file:open(Filename, write),
    io:fwrite(FILE, "15: ~p~n", [fibo(15)]),
    io:fwrite(FILE, "10: ~p~n", [fibo(10)]),
    io:fwrite(FILE, "25: ~p~n", [fibo(25)]).
```

› **This is a function implementation that contains documentation information. You can also see how to write documentation about the module**

that you need to prefix the name of the function with the name of the module in order to execute it. Figure 3 shows the relevant Erlang code, as found in the lud.erl module.

Pattern matching

Pattern matching plays a key role in functional programming languages and Erlang is no exception. Most things in Erlang work with pattern matching techniques, which should be exhaustive and cover all possible values. Most of the times, the last case in a matching scenario matches what is left. Please have in mind that the order of the branches in pattern matching is significant as only the first match gets executed, so you will have to go from the most special cases to the most generic ones when trying to match something.
Figure 4 shows a naïve function named patternMatch(), that works with a case statement. Executing patternMatch() generates the following kind of output:

```
2> lud:patternMatch(10).
10!ok
3> lud:patternMatch(20).
20!ok
4> lud:patternMatch(30).
Unknown!ok
```

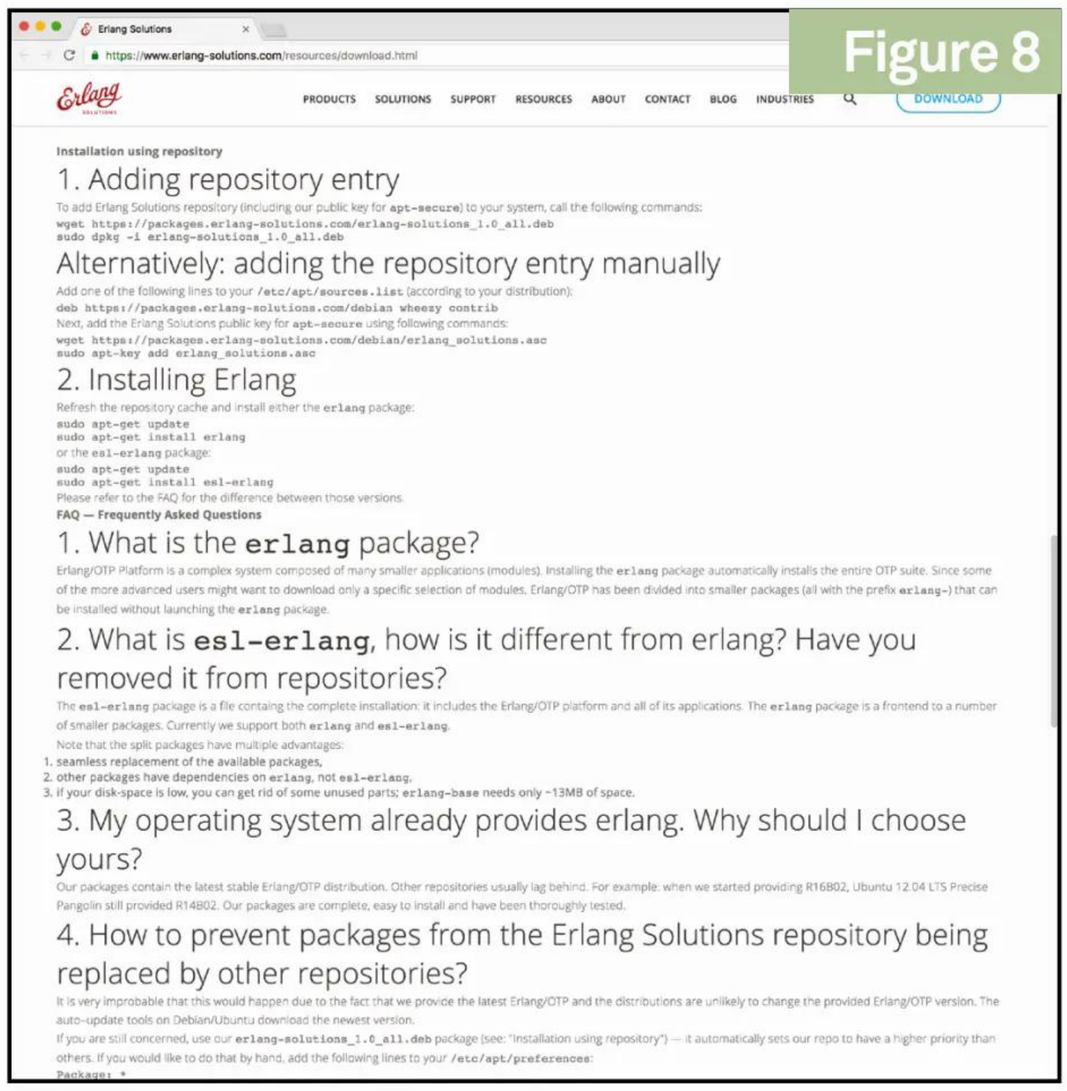
If your code fails to catch all cases and such a case comes up, you will get an error message similar to the following:

```
4> lud:patternMatch(50).
** exception error: no case clause matching 50
   in function  lud:patternMatch/1 (lud.erl,
line 25)
```

Erlang functions

Functions in Erlang are first class citizens and can do many interesting things. If you examine Figure 5 you will see an Erlang function named readlines() that opens a file for reading in order to process it by another function. Figure 6 shows the helloWorld() function from hw.erl with a preamble that contains documentation information and has nothing to do with the way the function works. A forthcoming tutorial in a future issue will talk more about Erlang functions and how you can use them in your programming.

Figure 8



› **You can get precompiled packages for installing the latest Erlang version by visiting https://www.erlang-solutions.com/**

File I/O

This next section will develop a function that accepts one argument, which is the name of a particular file, in order to create it and write three integers of the Fibonacci sequence to it. As you can probably already guess, it will use the fibo() function from the practical.erl module, which also shows you how to get the output of other functions. The Erlang code of the writeFibo() function can be seen in Figure 7. This section is just a quick introduction to File I/O in Erlang so there is no point in explaining all the code of writeFibo(); a forthcoming tutorial will talk more about file input and output operations in Erlang.
This tutorial was an introduction to the capabilities of Erlang. If you want to make your life easier, start writing Erlang code as soon as possible to get used to Erlang because in the next tutorial, we are going to learn more about specific aspects of Erlang.

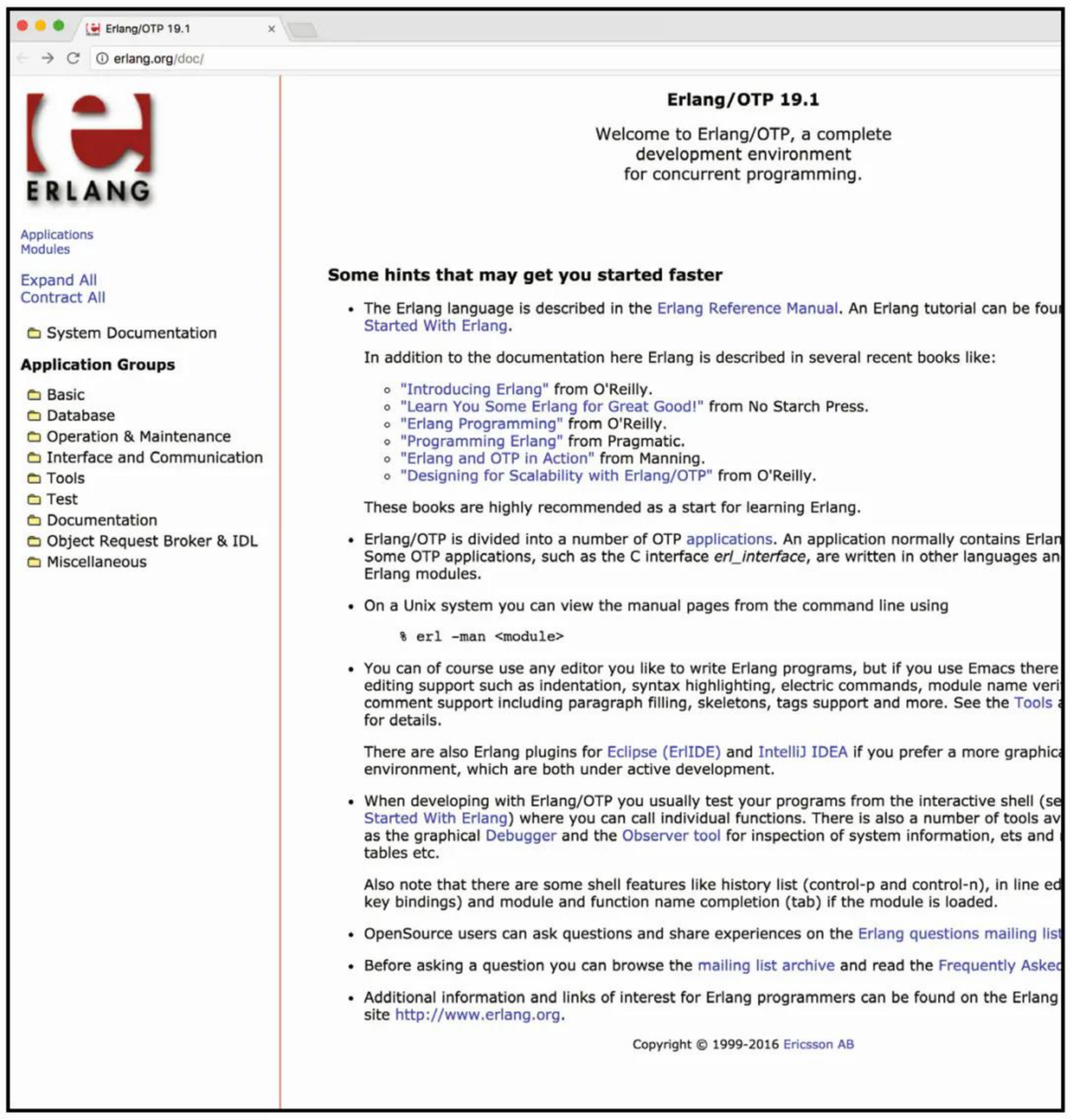
Do more

Erlang: Functions

Discover Erlang functions and basic Erlang data types as well as other interesting and helpful Erlang topics

This tutorial is the second one in the series of tutorials about the Erlang programming language. The main subject of this tutorial that we'll be looking at in depth is Erlang data types and functions.

As you might remember from the previous tutorial, all Erlang code comes in modules unless you are experimenting in the Erlang shell; as a result, all Erlang code comes in functions. Erlang has a pretty unusual way of defining functions, especially if you are used to programming languages such as C or Python, which will be explained here. Additionally, as Erlang is a functional programming language, it also supports anonymous functions, which are also going to be illustrated. You will also learn about atoms, lists, maps and tuples, so start reading!



More About Erlang

Concurrency is a central part of Erlang. As a result, Erlang processes, which should not be confused with Linux processes, are lightweight. Put simply, Erlang processes are easy to create, much easier than Linux processes, as they require a very small amount of time and have a small memory overhead. Erlang processes do not communicate with each other using memory, which is a risky thing, but by using messages. Furthermore, as processes are independent, the memory space of each process can be garbage-collected individually. Last, the failure of a process cannot do any damage to other processes, therefore allowing them to continue their jobs.

More About OTP

OTP is a central part of Erlang and the Erlang way of thinking because it allows you to make your Erlang applications highly available. This section will talk a little bit more about OTP in order to get a better understanding of it.

OTP is unique among programming languages and allows teams to work and develop distributed, fault-tolerant, scalable and highly available systems.

Despite its name (Open Telecom Platform), OTP is domain independent, which means that you can program applications for many different areas.

OTP consists of three main parts. These are the Erlang language itself, various tools that come with Erlang, and the design rules, which are generic behaviours and abstract principles that allow you to focus on the logic of the system. The behaviours can be worker processes that do the dirty work, while supervisor processes monitor workers as well as other supervisors. In order to do this right, the developer should structure the processes appropriately.

That is enough information about OTP for this tutorial; if you want to learn even more details about OTP then we recommended some excellent books in the previous tutorial.

Variables and Numbers

As expected Erlang supports two kinds of numbers, integers and floats.

When defining floats, you should always have a number on the left of the decimal point, even if it is zero. If you forget to do so, you will get the following kind of error message:

```
11> MyFloat = .987.
* 1: syntax error before: ','
```

If the statement is correct, Erlang will reply by printing the float value:

```
11> MyFloat = 0.987.
0.987
```

Figure 1 shows an interaction with the Erlang shell where many variables are declared and used. You should pay special attention to the b() function that prints all defined variables and the f() function that clears all the bound

Figure 1

```
5> mtsouk@mail:~$ erl
Erlang/OTP 19 [erts-8.1] [source-e7be63d] [64-bit] [async-threa
alse]

Eshell V8.1  (abort with ^G)
1> 10 - 5.
5
2> 10.4 - 0.5.
9.9
3> W = 1.
1
4> B = 2.
2
5> b().
B = 2
W = 1
ok
6> W - B.
-1
7> 2 = B.
2
8> 3 = B.
** exception error: no match of right hand side value 2
9> MyFloat = .987.
* 1: syntax error before: '.'
9> MyFloat = 0.987.
0.987
10> b().
B = 2
MyFloat = 0.987
W = 1
ok
11> f(MyFloat).
ok
12> f(MyFloat).
ok
13> b().
B = 2
W = 1
ok
14> f(DoesNotExist).
ok
15> f().
ok
16> b().
ok
17> W-B.
* 1: variable 'W' is unbound
18> 3 = B.
* 1: variable 'B' is unbound
19> B = 3.
3
20> B = 4.
** exception error: no match of right hand side value 4
21>
```

› **The use of b() and f() functions as well as the declaration of numeric variables in Erlang**

Figure 2

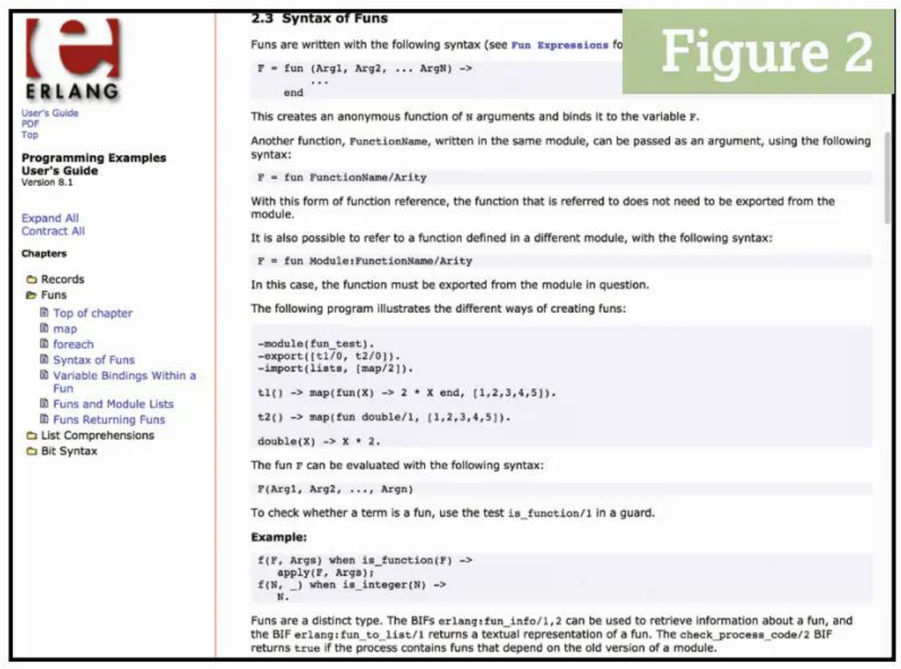


Figure 3

```
Eshell V8.1  (abort with ^G)
1> MYMAP = #{country=>greece, city=>athens, year=>2016, d
#{city => athens,country => greece,date => {nov,18},year
2> MYMAP.
#{city => athens,country => greece,date => {nov,18},year => 2016}
3> maps:get(city,MYMAP).
athens
4> maps:update(city,patras,MYMAP).
#{city => patras,country => greece,date => {nov,18},year => 2016}
5> map_size(MYMAP).
4
6> LIST1 = [a, b, 3, {a,b}].
[a,b,3,{a,b}]
7> [H|T] = LIST1.
[a,b,3,{a,b}]
8> H.
a
9> T.
[b,3,{a,b}]
10> LIST2 = [d|T].
[d,b,3,{a,b}]
11> LIST3 = [{a,b}|T].
[{a,b},b,3,{a,b}]
12> length(LIST3).
4
13> length(LIST2).
4
14>
```

› **This is a small part of the Erlang reference about the fun keyword**

variables when executed without any parameters or a specific variable when the variable is given as an argument.

Erlang data types

Erlang supports many data types including atoms, maps, lists and funs. An atom is used for representing a constant value. Atoms have a global scope and always start with lowercase letters:

```
1> linux.
linux
2> 12.
12
```

As you can see, the value of an atom is the atom itself! Although it looks strange to discuss the value of an atom or an integer, the functional nature of Erlang requires that each expression has a value, which also applies to atoms and integers, despite the fact that they are naïve expressions.

A fun is a functional object that also allows you to create anonymous functions, which you can pass as arguments to other functions as if they were variables, without having to use their names. Figure 2 shows a part of the use Erlang reference about the fun keyword – there will be more about anonymous functions going forward.

A map is a compound data type that can contain a variable number of key-value pairs. Each pair is called an element – the total number of elements is called the size of the map. The following shell command shows how to create a map:

```
1> MYMAP = #{country=>greece, city=>athens,
year=>2016, date=>{nov,18}}.
#{city => athens,country => greece,date =>
{nov,18},year => 2016}
```

As you can understand, there are many functions that allow you to manipulate maps – you can see some of them in action in Figure 3.

A list is another compound data type with a variable number of elements. You can define a new list in the Erlang shell as follows:

```
1> LIST1 = [a, b, 3, {a,b}].
[a,b,3,{a,b}]
```

Please also bear in mind that behind the scenes Erlang treats strings as lists, so everything that can work on a list can also be used for strings.

A unique process ID identifies each Erlang process. A PID has the following form and its own data type, which means that you cannot use a process ID as if it was a string:

```
1> self().
<0.57.0>
```

The self() function returns the process ID of the calling process. Similarly, the spawn() function returns the process ID of the new process, which is used for sending messages to it:

```
1> c(hw).
{ok,hw}
2> spawn(hw, helloWorld, []).
Hello, world!
<0.65.0>
```

Figure 4

```
-module(tuples).
-export([main/0]).

process_tuple(TUPLE) when is_tuple(TUPLE) ->
    N = tuple_size(TUPLE),
    W = element(1, TUPLE),
    io:format("Size: ~w First element: ~w ~n", [N, W]);

process_tuple(_) ->
    io:format('This is not a tuple!~n').

main() ->
    T1 = {[1, 2], 2, 3, 4, 5, 6},
    T2 = {a, b, c, d},
    process_tuple(T1),
    process_tuple(T2).
```

› How to define a map and a list in Erlang shell, including functions that help you deal with maps and lists

› The implementation of the process_tuple() function that illustrates how to process tuples

Figure 5

```
-module(more_fun).
-export([check_temp/1]).

check_temp(Temp) ->
    case Temp of
        {celsius, N} when N >= 20, N =< 45 ->
            'Hot';
        {celsius, N} when N >= 46 ->
            'Way too hot!';
        {celsius, N} when N =< 19 ->
            'It is getting cold...';
        {kelvin, N} ->
            'Cannot tell about Kelvin!';
        {fahrenheit, N} ->
            'Do not know about Fahrenheit!';
        _ ->
            'What?'
    end.

```

» As you can see, spawn() takes three parameters, which are the name of the module the function belongs to, the name of the function, and the parameters of the function, which are passed as a list. If the function takes no parameters, you should pass an empty list. Figure 3 shows you how to define and process maps and lists inside the Erlang shell.

Tuples and guards

A guard will allow you to specify the kind of data a given function will accept. Although it might just look a little fuzzy at the moment, you will see Erlang code that uses guards later on in this tutorial. The when keyword indicates a guard. The condition of a guard is relatively simple and will allow you to do pattern matching based on the content of the argument and not just on its shape.

A tuple is a composite data type, which means that a tuple allows you to combine multiple items into a single data type and store them using a single variable. Most of the times Erlang tuples group two to five items. Moreover, the first atom of a tuple usually identifies the purpose or the category of the tuple. You can declare a tuple as follows:

```
2> T1 = {linux, 1}.
{linux,1}
```

The element() function allows you to access a given item of a tuple:

```
3> element(1, T1).
Linux
```

As you can see, the first element of a tuple has an index number of 1. Similarly, the setelement() function allows you to change the value of an existing tuple item in order to create a new tuple:

```
6> T2 = setelement(1, T1, unix).
{unix,1}
```

Tuples are handy as they let you group data and are frequently used in Erlang as well as other programming languages.

Erlang functions

A function in Erlang is a sequence of function clauses that are separated by semicolons and terminated using a period/full stop (.). A function clause has the following form:

```
1> F=fun(X) -> 2*X end.
#Fun<erl_eval.6.52032458>
```

The previous code creates an anonymous function with one argument. The anonymous function is bound to a

variable named F, which is also an atom. You can use it as follows:

```
3> F(4).
8
4> F(4.5).
9.0
```

The number of the arguments of a function is called the arity of the function. It is the combination of the module name (m), the function name (f) and the arity (N) that uniquely identifies a function as m:f/N. As you might remember, you have to export a function in order to be able to use it outside of the module that it belongs to.

The next example shows how to pass functions as arguments to other functions! Type the following at the Erlang shell:

```
1> F=fun(X) -> 2*X end.
#Fun<erl_eval.6.52032458>
2> Five = fun(N, Function) -> 5 * Function(N)
end.
#Fun<erl_eval.12.52032458>
3> Five(10,F).
100
4> F(10).
20
```

Here, you define two anonymous functions, named F() and Five(). The Five() function takes two arguments, which is an integer 'N' and a function 'Function' and multiplies the numeric result of Function(N) with the number 5!

More about functions

Erlang supports much more complex functions than the ones you saw in the previous section. The following example will illustrate how to use a function to process a tuple. Please have in mind each tuple counts as a single function argument. A very common way to process tuples is by using pattern matching. Figure 4 shows the code of the process_tuple() function that processes tuples as found in tuples.erl. Executing tuples:main/0 generates the next output:

```
15> c(tuples).
{ok,tuples}
16> tuples:main().
Size: 6 First element: [1,2]
Size: 4 First element: a
Ok
```

The last example will show a function that processes tuples using guards and a case statement, which is a pretty common practice in Erlang. Figure 5 shows the relevant Erlang code as found in more_fun.erl. Using more_fun:check_temp/1 generates the following kind of output:

```
11> c(more_fun).
more_fun.erl:12: Warning: variable 'N' is unused
more_fun.erl:14: Warning: variable 'N' is unused
{ok,more_fun}
12> more_fun:check_temp({fahrenheit, 50}).
'Do not know about Fahrenheit!'
13> more_fun:check_temp({kelvin, 50}).
'Cannot tell about Kelvin!'
14> more_fun:check_temp({celsius, 50}).
'Way too hot!'
15> more_fun:check_temp({celsius, 10}).
'It is getting cold...'
```

As you can see, not all case statements must have a guard.

More information about the Erlang shell

The q() function is the easiest way to exit the Erlang shell, but keep in mind that the q() function quits everything Erlang is doing. If you are working locally then there is no problem, but if you are working on a remote system you'd better quit by typing Ctrl+G and then Q. The reason is that you may shut down the Erlang runtime on the remote machine when quitting using q()! The built-in line editor of erl is a subset of well known editor Emacs.

In Figure 6 you will see some more advanced commands of the Erlang shell including the declaration of a function, the use of the h() function to print the history list, and two alternative ways to exit the Erlang shell, the init:stop() function as well as the halt() function.

Getting user input

Although Erlang is primarily used for server applications, it can also allow you to interact with users. This section will teach you how to get user input from Erlang, which is pretty handy when you are developing small interactive programs or other command line utilities.

Although it is relatively easy to get user input from the Erlang shell, the tricky part is verifying that the input is valid in order to avoid exceptions:

```
1> {ok, [VAR]} = io:fread("input : ", "~d").
input : 123
{ok,"{"}
2> VAR.
123
3> {ok, [OTHER]} = io:fread("input : ", "~d").
input : abc
** exception error: no match of right hand
side value {error,{fread,integer}}
```

Figure 7 shows sample Erlang code that teaches you how to get user input and make sure that you take what you want, which you might find more complicated than expected, especially if you are familiar with other programming languages. Executing userInput.erl generates the following kind of output:

```
1> c(userInput).
{ok,userInput}
Please give {Name, Surname} >> asd, asd.
There is an error somewhere.
Please give {Name, Surname} >> {'Mihalis',
'Tsoukalos'}.
Hello 'Mihalis' 'Tsoukalos'!
Please give {Name, Surname} >> 12.
A tuple is needed!
Please give {Name, Surname} >> quit.
Bye!
ok
```

Figure 6

```
mtsouk@mail:~$ erl
Erlang/OTP 19 [erts-8.1] [source-e7be63d] [64-bit] [async-thread
alse]

Eshell V8.1  (abort with ^G)
1> init:stop().
ok
2> mtsouk@mail:~$ erl
Erlang/OTP 19 [erts-8.1] [source-e7be63d] [64-bit] [async-threads:10] [hipe] [kernel-poll:f
alse]

Eshell V8.1  (abort with ^G)
1> halt().
mtsouk@mail:~$ erl
Erlang/OTP 19 [erts-8.1] [source-e7be63d] [64-bit] [async-threads:10] [hipe] [kernel-poll:f
alse]

Eshell V8.1  (abort with ^G)
1> F=fun(X) -> 2*X end.
#Fun<erl_eval.6.52032458>
2> F(2).
4
3> F(4).
8
4> F(4.5).
9.0
5> OTHER = fun(X) ->
5>     2 * X
5> end.
#Fun<erl_eval.6.52032458>
6> OTHER(5).
10
7> h().
1: F = fun(X) ->
           2 * X
       end
-> #Fun<erl_eval.6.52032458>
2: F(2)
-> 4
3: F(4)
-> 8
4: F(4.5)
-> 9.0
5: OTHER =
       fun(X) ->
           2 * X
       end
-> #Fun<erl_eval.6.52032458>
6: OTHER(5)
-> 10
ok
8>
```

As you can imagine, the key role is played by the function that checks the return value of the io:read() function, which is a tuple. Additionally, the user needs to end each input with a dot (full stop/period).

As you will see, Erlang is not particularly good at dealing with strings. Additionally, it is the job of the developer to check that the input is in the right form and of the desired data type because improper data might create troubles when you attempt to process it.

There are other ways to get user input, including reading characters and reading entire lines of text – you will learn more about it in the next Erlang tutorial.

Creating Erlang scripts

The escript binary file allows you to create Erlang scripts, which is an attractive capability of Erlang found in most scripting languages. The following Erlang script accepts one command line argument and asks the user for their name, using a simplified version of the code found in userInput.erl:

```
#!/usr/bin/env escript
%% -*- erlang -*-

main([String]) ->
    try
      N = list_to_integer(String),
        I = io:read("Please give {Name} >> "),
        process_input(I),
        io:format(": ~w ~n", [N])
    catch
      _:_ ->
        usage()
    end;
main(_) ->
    usage().

usage() ->
    io:format("usage: scriptName integer\n"),
    halt(1).

process_input({ok, Data}) when is_tuple(Data) ->
      Name = element(1, Data),
      io:format("~w", [Name]);

process_input({error, _}) ->
      io:format("There is an error somewhere.~n").
```

After you have created the script file, you should then change its permissions:

```
$ chmod 755 aFile.erl
$ ls -l aFile.erl
-rwxr-xr-x  1 mtsouk  staff  0 Nov 14 15:30
aFile.erl
```

“It is the job of the developer to check that the input is in the right form and of the desired data type, because improper data might create troubles when you attempt to process it”

How to define a function inside the Erlang shell and the init:stop() and halt() functions that help you exit it

Filesystems and the Erlang shell

There will be times when you would like to move to another directory while you are working in the Erlang shell. Look at the following interaction with the Erlang shell:

```
1> pwd().
/home/mtsouk
OK
2> cd("..").
/home
OK
3> pwd().
/home
OKYou'll learn more about the .unwrap()
```

So, the cd() function allows you to change the current directory, whereas pwd() prints the current working directory.

Infinite loops

The following Erlang code, when combined with a negative integer as an argument, generates a function that keeps running and never ends:let args2 = os::args();

```
fibo(0) -> 0;
fibo(1) -> 1;
fibo(N) -> fibo(N - 1) + fibo(N - 2).
```

In other words, you should be very careful when defining the arguments that a function can accept, and put guards where necessary in order to avoid such bugs. A more appropriate and secure definition of fibo() would be the following example:

```
fibo(N) when N > 0 -> fibVar(N).
fibVar (0) -> 0;
fibVar (1) -> 1;
fibVar (N) -> fibVar (N - 1) + fibVar (N - 2).
```

Figure 7

```
-module(userInput).
-export([getInput/0]).

getInput() ->
    I = io:read("Please give {Name, Surname} >> "),
    process_input(I).

process_input({ok, Data}) when is_tuple(Data) ->
    Name = element(1, Data),
    Surname = element(2, Data),
    io:format("Hello ~w ~w! ~n", [Name, Surname]),
    getInput();

process_input({ok, quit}) ->
    io:format("Bye!~n");

process_input({ok, _}) ->
    io:format("A tuple is needed!~n"),
    getInput();

process_input({error, _}) ->
```

The code of userInput.erl illustrates one way of getting user input

Figure 8

```
code$ ll fibo1.*
-rw-r--r-- 1 mtsouk mtsouk 252 Nov 17 21:52 fibo1.erl
code$ rl -noshell -s fibo1 main 100 -s init stop
-bash: rl: command not found
code$
code$ ls -l fibo1.*
-rw-r--r-- 1 mtsouk mtsouk 252 Nov 17 21:52 fibo1.erl
code$ erl -noshell -s fibo1 main 100 -s init stop
{"init terminating in do_boot",{undef,[{fibo1,main,[['100']],[]},{init,start_em,1,
,do_boot,3,[]}]}}
init terminating in do_boot ()

Crash dump is being written to: erl_crash.dump...done
code$ ls -l erl_crash.dump
-rw-r----- 1 mtsouk mtsouk 334184 Nov 17 21:53 erl_crash.dump
code$ rm erl_crash.dump
code$ erc fibo1.erl
-bash: erc: command not found
code$ erlc fibo1.erl
code$ erl -noshell -s fibo1 main 100 -s init stop
100: 354224848179261915075
code$ time erl -noshell -s fibo1 main 100 -s init stop
100: 354224848179261915075

real    0m1.170s
user    0m0.133s
sys     0m0.020s
code$
```

The use of the fibo() function found in fibo1.erl

Figure 9

```
-bash: rls: command not found
code$
code$ ls -l fibo2.*
-rw-r--r-- 1 mtsouk mtsouk 296 Nov 17 21:52 fibo2.erl
code$ erlc fibo2.erl
code$ time erl -noshell -s fibo2 main 100 -s init stop
100: 354224848179261915075

real    0m1.200s
user    0m0.143s
sys     0m0.030s
code$ cat fibo2.erl
-module(fibo2).
-export([main/1]).

fibo(0) -> 0;
fibo(1) -> 1;
fibo(N) ->
    fibo(N + 1, [1,0]).

fibo(End, [H|_]=L) when length(L) == End -> H;
fibo(End, [A,B|_]=L) ->
    fibo(End, [A+B|L]).

main([Arg|_]) ->
    N = list_to_integer(atom_to_list(Arg)),
        io:format("~w: ~w~n", [N, fibo(N)]).

code$
```

The use of the fibo() function found in fibo2.erl as well as its entire Erlang code

Next, you can execute aFile.erl just as if it was a regular shell script:

```
$ ./aFile.erl 123
Please give {Name} >> {'Mihalis'}.
'Mihalis': 123
```

As you can see, you just have to embed your Erlang code into a text file in a specific way and read the command line arguments as a list using the list_to_integer() function. This is a very pretty way of creating small Erlang programs that do specific but relatively small tasks. The next tutorial will talk about formatting output, lists, maps, records and message passing between Erlang processes. This will give you a much greater insight into actually developing working programs with Erlang and taking advantage of its features in order to make them run better and more efficiently. Until then, write as much Erlang code as you can!

Quick tip

Although the Erlang code of fibo1.erl is different from the code you saw in the previous tutorial, there is another way to calculate Fibonacci numbers in Erlang, which is presented in fibo2.erl. The implementation of the fibo() function, which uses a list, as well as its performance, can be seen in Figure 9. Just remember to compile the code first using erlc, which is the Erlang compiler. As you can see, not all Fibonacci implementations are equal!

Calculating Fibonacci numbers

We will now learn how to calculate Fibonacci numbers in a different way than the one you saw in the previous tutorial. The code of the fibo() function, which can be found in fibo1.erl, is the following:

```
fibo(N) when N > 0 -> fibVar(N, 0, 1).
fibVar(0, F1, _F2) -> F1;
fibVar(N, F1,  F2) -> fibVar(N - 1, F2, F1 + F2).
```

You can see its performance using the time(1) command in Figure 8. The implementation of fibo() in fibo1.erl uses another function that is named fiboVar() and takes three arguments instead of just one. However, as fiboVar() is only used internally, it does not need to be in the export() list of the module. Therefore, for this tutorial, the only function in the export() list is main/1.

Rust: Modules and Cargo

It's time to kick off a new series on Rust by covering the basics, such as data types, modules, variables and the package manager, Cargo

This tutorial is a gentle introduction to the Rust programming language. In later tutorials, we'll talk about more specific areas of Rust but for now we will cover the foundations. By the end of it, through calculating Fibonacci numbers and writing a small Rust program that calculates the number of characters in a text file you should have an understanding of the basics and get a good sense of the language. Additionally, you'll learn about the various data types and how the *Cargo* package manager can help you build Rust projects.

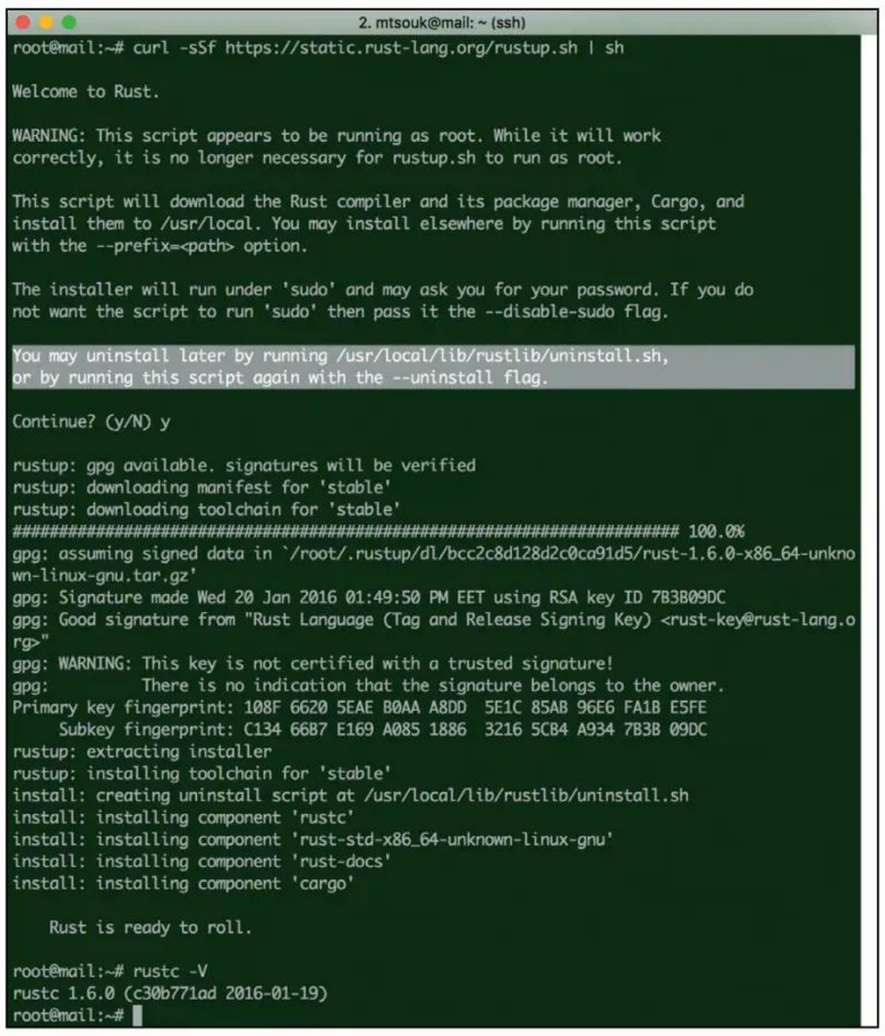
Installing Rust on a Linux distribution (distro), in our case Debian, is as simple as executing the following command (note: which you'll need to do as root):

```
# curl -sSf https://static.rust-lang.org/rustup.sh | sh
```

This command not only installs Rust but also downloads and installs *Cargo*. (Top right, shows the installation process.) You can uninstall Rust by executing the **/usr/local/lib/rustlib/uninstall.sh** script. As Rust is changing rapidly you can check version of the language you're using with:

```
$ rustc -V
rustc 1.6.0 (c30b771ad 2016-01-19)
```

The Rust compiler (rustc) is clever enough to find errors and is also responsible for enforcing the safety rules. As a consequence, it might prevent your Rust code from compiling until you make the necessary modifications to your code. This isn't necessarily a bad thing because this makes your code more robust. At the time of writing this tutorial, Rust 1.6.0 is the latest stable version.



The best way to install the latest stable Rust version, as well as the *Cargo* tool, on your Linux machine is to use the rustup.sh script which also provides you with a shell script in rustlib for an easy uninstall

Hello World

Now let's dive in with Hello Word code for Rust:

```
fn main() {
println!("Hello World!");
}
```

By convention, Rust code is saved in files that have an .rs extension, so the "Hello World!" program can be saved as **helloW.rs**. As you can see, the definition of every Rust function, including `main()`, begins with the `fn` keyword. Any autonomous Rust program must have a `main()` function. Additionally, the `println!` macro is similar in functionality to the printf() C function and is responsible for printing your output. You can tell that `println!` is a macro because of its form: it uses an identifier followed by an exclamation point. Macros are useful in Rust because they provide syntactic abstraction. You can compile **helloW.rs** in the following way:

```
$ rustc helloW.rs
```

```
$ ls -l helloW
-rwxr-xr-x 1 mtsouk mtsouk 574184 Feb 25 21:08 helloW
$ file helloW
helloW: ELF 64-bit LSB shared object, x86-64, version 1
(SYSV), dynamically linked, interpreter /lib64/
ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=f9
9c6bf5f6c8ad0b551787ce573c615ccec1167a, not stripped
$ ./helloW
Hello World!
$ strip helloW
$ ls -l helloW
-rwxr-xr-x 1 mtsouk mtsouk 308016 Feb 25 21:09 helloW
```

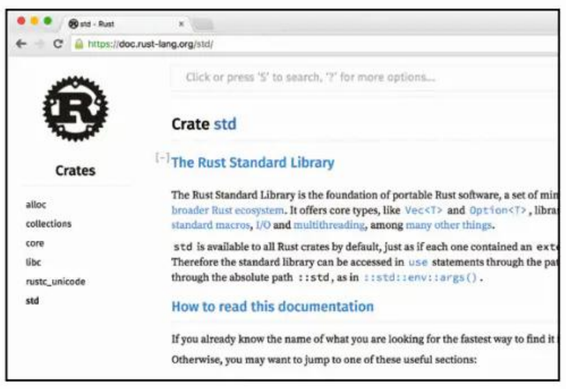
Once you've entered this code, the `strip` command will discard symbols from object files and usually makes an executable file smaller in size. Here, it made the helloW executable 266,168 bytes smaller!

Quick tip

You can find the full Rust documentation at **https://doc.rust-lang.org**. The book is a good introduction to Rust. Find more information about *Cargo* **http://doc.crates.io**.

Rust modules

Rust comes with a plethora of modules that can speed up your development process. All modules that belong to the Standard Library are under the std:: tree. Some important and handy modules are: » **std::env** For inspecting a process's environment and manipulating it.
» **std::fs** For cross-platform filesystem manipulation operations;
» **std::net** For TCP and UDP communication;
» **std::os** For OS specific operations.
» **std::path** For cross-platform path manipulation operations.
» **std::process** For dealing with Linux processes;
» **std::threads** For working with threads
» **std::time** For dealing with time.
Please note that std::os has three subcategories: std::os::linux, std::os::raw and std::os::unix. Head to **https://doc.rust-lang.org/std** for more about the Standard Library and the std module is available to all Rust crates by default.



» **The Rust Standard Library web details Rust's standard functionality**

Rusty Fibonacci

Let's go on and find out how to calculate Fibonacci numbers in Rust. We've called the file **fibonacci.rs** and the code is:

```
fn fibonacci(n: i32) -> i32
{
        if n == 0 {
                return 0;
        }

        if n <= 1 {
                return 1;
        }
        else {
                return fibonacci(n - 1) + fibonacci(n - 2);
        }
}

fn main() {
        for x in 0..10 {
                println!("Fibonacci number {} is {} ", x,
fibonacci(x));
        }
}
```

You also see a for loop is defined in Rust in a different way from how you may know it from programming languages, such as C and C++. Additionally, you don't have to enclose the condition of an if statement inside parentheses. But what's really important about this code is that it teaches you how to define a function that gets one argument as input and returns one value:

```
fn fibonacci(n: i32) -> i32
```

Bear this function definition in mind as we're going to learn more about Rust functions, modules, etc in the next tutorial, but for now we'll learn more about the i32 keyword in the next section. You can print dynamic text that uses the value of a variable using `println!` as follows:

```
println!("Fibonacci number {} is {} ", x, fibonacci(x));
```

As Rust already knows the type of the variable, you don't have to define it inside `println!`.

Variables in Rust

You can declare a new variable using the `let` keyword; it's not compulsory to specify the type of a variable because the Rust compiler will try to find out the type. As is the case with most modern programming languages, variables are immutable by default. Should you wish to define a mutable variable, you should use `let mut` instead of just `let`. It's considered good practice to manually give a type to a variable when you declare it and not let the compiler do the dirty work for you. Rust also supports pointers but you won't have to use them as often as in C. (In the next tutorial you'll learn more about dealing with pointers in Rust.)

The following code (variables.rs) defines four variables:

```
let a1 = -78912312;
let a2:i16 = 2;
let mut a3:u32 = 123;
let mut a4:f32 = -123.321;
```

The `a1` variable is of integer type whereas `a2` is a signed integer that uses 16 bits to store its data. Both `a1` and `a2` are immutable so their values cannot change. Variables `a3` and `a4` are an unsigned 32-bit integer and a 32-bit floating point number, respectively. You can alter the value of both `a3` and `a4` as they have been declared with `let mut`. If a variable isn't being used in your program, the Rust compiler will print a warning message:

```
variables.rs:3:6: 3:8 warning: unused variable: `a1`,
#[warn(unused_variables)] on by default
```

Similarly, the Rust compiler will warn you when a variable doesn't need to be mutable:

```
variables.rs:5:6: 5:12 warning: variable does not need to be
mutable, #[warn(unused_mut)] on by default
```

If you want to find out the size of a variable in bytes, you can use the std::mem library as follows:

```
println!("The size of a1 is {} bytes", mem::size_of_val(&a1));
```

The output of the previous code will be `The size of a1 is 4 bytes` which means that a1 occupies 32 bits.

Rust has built-in support for a plethora of constants, such as pi and e, through the std::f64::consts module, but you can also declare your own constant values. Please note that it's mandatory to define the data type of a constant value. Constant values should be defined outside of all function definitions as follows:

```
const LINESIZE:i8 = 1024;
```

Constant values in Rust are pretty much like using #define in C (eg `#define LINESIZE 1024`) and cannot be modified. Another way of defining global variables is with the 'static' keyword which we'll cover in a later tutorial.

Rust data types

Rust comes with many built-in data types including u8, u16, u32, u64 for unsigned integers and i8, i16, i32, i64 for signed integers where the number (8, 16, 32 and 64) after the 'u' or 'i' »

Quick tip

The special area of systems programming is focused on programs that deal with file I/O, files and directories, threads, system files and information, server processes, network programming, signal handling and inter-process communication etc.

A program with bugs in these areas can crash your Linux system so be careful if you decide to start developing systems software.

Do more

```
2. mtsouk@mail: ~/docs/article/working/rustIntro.LXF210/code (ssh)
WC$ ll
total 12
-rw-r--r-- 1 mtsouk mtsouk   38 Feb 29 17:42 Cargo.lock
-rw-r--r-- 1 mtsouk mtsouk   77 Feb 29 17:41 Cargo.toml
drwxr-xr-x 2 mtsouk mtsouk 4096 Feb 29 17:38 src
WC$ wc Cargo.toml
 6 11 77 Cargo.toml
WC$ cargo build
   Compiling WC v0.1.0 (file:///home/mtsouk/docs/article/working/rustIntro.LXF210/code/WC)
warning: crate `WC` should have a snake case name such as `wc`, #[warn(non_snake_case)] on
by default
WC$ cargo run Cargo.toml
     Running `target/debug/WC Cargo.toml`
File to open: Cargo.toml
Total number of characters: 77
WC$
WC$ cargo run
     Running `target/debug/WC`
thread '<main>' panicked at 'called `Option::unwrap()` on a `None` value', ../src/libcore/o
ption.rs:367
Process didn't exit successfully: `target/debug/WC` (exit code: 101)
WC$ cat src/main.rs
use std::io::{BufReader,BufRead};
use std::fs::File;
use std::env;

fn main() {
        let mut count = 0;
        let filename = &env::args().nth(1).unwrap();
        println!("File to open: {}", filename);

        // No Error Checking!!!
        let f = File::open(filename).unwrap();
```

» The WC project counts the characters of a text file that is given as a command line argument.

» character shows the minimum number of bits used for storing a number of this type. Similarly, you can use f32 and f64 for storing floating point numbers.

As you might expect Rust has support for arrays. You can declare an array and find its size as follows:

```
let an_array = [3.5, 1.0, 2.0];
println!("The an_array array has {} elements", an_array.len());
```

If the first element of an array is an integer, then the compiler will automatically think that all the elements of the array should be integers. So the following code, for instance, won't compile: `let an_array = [1, 3.5, 1.0, 2.0];`. The error message you will get is:

```
variables.rs:13:31: 13:34 error: mismatched types:
 expected `_`,
    found `_`
(expected integral variable,
    found floating-point variable) [E0308]
```

If you name an array variable as anArray, the Rust compiler will generate a warning message:

```
variables.rs:13:6: 13:13 warning: variable `anArray` should
have a snake case name such as `an_array`, #[warn(non_
snake_case)] on by default
```

This happens because Rust prefers the `an_array` notation when declaring variables. However, your program will still compile. You can manually define the type and the size of an array in the following way:

```
let another_array: [f32; 5] = [1.0, 3.5, 1.0, 2.0, 2.0];
let init_array: [f32; 5] = [2.0; 5];
```

The `init_array` variable uses a handy Rust shorthand for initialising each element of an array to the 2.0 value. You can visit all array elements using an iterator as follows:

```
for x in another_array.iter() {
        println!("{}", x);
}
```

The `.take(n)` method (*below*) will only return n elements whereas the `.skip(n)` method will skip the first n elements and continue with the remaining ones. You can also combine `.take(n)` with `.skip(m)` and use both of them as `.skip(n).take(m)` to skip the first n elements and return the next m elements:

```
for x in another_array.iter().skip(2).take(2) {
        println!("{}", x);
}
```

Rust also supports Boolean variables with the help of the true and false keywords:

```
let it_is_true = true;
let it_is_false = false;
```

You can declare a variable for holding a single character with `let ch = 'a';`. Similarly, you can declare a variable for holding a string with `let str = "This is a string!";`.

You can process a string character by character using the `.chars()` method:

```
for x in aLine.chars() {
        count += 1;
}
```

We'll use the previous code in the WC project later on. As you can see (pictured, above left) there exists a `.lines()` method for processing a file line by line.

Quick tip

Apart from C, you can write systems software in Go, Perl and Python etc. It's important to understand, however, that all programming languages implement their own systems functions using the well-known C system calls.

Command line arguments

Here's how to read and process the command line arguments in Rust. The full Rust code can be found in argV.rs, where you'll see that you need to use the std::env package and the env::args() method to read the arguments. Note: argV.rs performs no error checking (we'll be covering error checking in Rust in the next part).

Should you wish to use the command line arguments as numbers, you can use the following Rust code which shows how to properly convert a string to an integer:

```
let args2 = os::args();
let mut sum = 0;
for x in args2.skip(1)
{
   let number: u32 =  x.parse().unwrap();
   println!("{}", number)
   sum += number;
}
println!("Sum: {}", sum);
```

You'll learn more about the .unwrap() method in the next issue, but you can see (above) a handy use of the `skip()` method to skip the first command line argument, which is the name of the executable. If you enter a non-numerical value as a command line argument, even the minus sign, the program will crash.

```
2. mtsouk@mail: ~/docs/article/working/rustIntro.LXF210/code (ssh)
helloWorld$ cargo build
   Compiling regex-syntax v0.2.5
   Compiling utf8-ranges v0.1.3
   Compiling libc v0.2.7
   Compiling memchr v0.1.10
   Compiling aho-corasick v0.5.1
   Compiling regex v0.1.54
   Compiling helloWorld v0.1.0 (file:///home/mtsouk/docs/article/working/rustIntro.LXF210/c
ode/helloWorld)
warning: crate `helloWorld` should have a snake case name such as `hello_world`, #[warn(non
_snake_case)] on by default
helloWorld$ cat Cargo.toml
[package]
name = "helloWorld"
version = "0.1.0"
authors = ["mtsouk"]

[dependencies]
regex = "0.1.41"
helloWorld$ ll target/debug/deps/
total 13200
-rw-r--r-- 1 mtsouk mtsouk  726718 Feb 25 20:59 libaho_corasick-eea45523c8483aac.rlib
-rw-r--r-- 1 mtsouk mtsouk  728752 Feb 25 20:59 liblibc-c862fb1c783dd674.rlib
-rw-r--r-- 1 mtsouk mtsouk   99632 Feb 25 20:59 libmemchr-545f7c18e4fb5174.rlib
-rw-r--r-- 1 mtsouk mtsouk 6764620 Feb 25 20:59 libregex-65b13a44d3660542.rlib
-rw-r--r-- 1 mtsouk mtsouk 4750680 Feb 25 20:59 libregex_syntax-46f1c6091c03f7ce.rlib
-rw-r--r-- 1 mtsouk mtsouk  395116 Feb 25 20:59 libutf8_ranges-afda04de2a52d0c6.rlib
helloWorld$ ll ~/.cargo/
total 4
drwxr-xr-x 5 mtsouk mtsouk 4096 Feb 25 20:56 registry
helloWorld$ ll ~/.cargo/registry/
total 12
drwxr-xr-x 3 mtsouk mtsouk 4096 Feb 25 20:56 cache
drwxr-xr-x 3 mtsouk mtsouk 4096 Feb 25 20:56 index
drwxr-xr-x 3 mtsouk mtsouk 4096 Feb 25 20:56 src
helloWorld$ ll ~/.cargo/registry/src/
total 4
drwxr-xr-x 8 mtsouk mtsouk 4096 Feb 25 20:56 github.com-88ac128001ac3a9a
helloWorld$ ll ~/.cargo/registry/src/github.com-88ac128001ac3a9a/
total 24
drwxr-xr-x 5 mtsouk mtsouk 4096 Feb 25 20:56 aho-corasick-0.5.1
drwxr-xr-x 4 mtsouk mtsouk 4096 Feb 25 20:56 libc-0.2.7
drwxr-xr-x 4 mtsouk mtsouk 4096 Feb 25 20:56 memchr-0.1.10
drwxr-xr-x 7 mtsouk mtsouk 4096 Feb 25 20:56 regex-0.1.54
drwxr-xr-x 3 mtsouk mtsouk 4096 Feb 25 20:56 regex-syntax-0.2.5
drwxr-xr-x 4 mtsouk mtsouk 4096 Feb 25 20:56 utf8-ranges-0.1.3
helloWorld$
```

› **This shows how *Cargo* deals with the dependencies of a projects which are automatically being downloaded and linked in order to generate the requested executable file**

About Cargo

Cargo is the Rust package manager, which can help you generate new packages and deal with their dependencies. Apart from the Rust code itself, the **Cargo.toml** file that's found in every *Cargo* project is the most important file of a project, because it contains the metadata that *Cargo* needs in order to compile your project. As an example, the default **Cargo.toml** file for a project named helloWorld would be:

```
$ cargo new helloWorld --bin
code$ cd helloWorld/
helloWorld$ cat Cargo.toml
[package]
name = "helloWorld"
version = "0.1.0"
authors = ["mtsouk"]

[dependencies]
```

If you want to make a binary executable, the `--bin` parameter should be used, but you don't need the parameter when you want to develop a library. As you can see (*pictured above*) *Cargo* automatically generates a file named **main.rs** inside the **src** directory where you should put your Rust code. By default, **main.rs** contains the source code of the "Hello World!" program. You're free to edit **Cargo.toml** and change the values of the name, version and authors variables.

You can compile a Cargo project by executing the `cargo build` command. For the helloWorld project, the executable file with debug information will be located at **target/debug/helloWorld**. Should you wish to build the release version, you would run `cargo build --release` and the executable file will be **target/release/helloWorld**.

Alternatively, if you want to execute your program in one step, you can run the `cargo run` command, and clean your project by executing `cargo clean` command. If you put your command line arguments after the entire `cargo` command, you can pass them to your executable.

You can add a dependency to your project, too. The good thing is that Cargo automatically downloads the required dependencies as well as any dependencies that your dependencies might have. (See p111 for screenshot of how Cargo builds a project when there are dependencies defined inside **Cargo.toml**.)

Despite the fact that your project needs just one library (regex), you'll notice that Cargo also downloaded and used five more libraries. As you can see, you can also define the minimum version of the library you need. (Screenshot, left, shows that version 0.1.54 of the regex library that was actually downloaded.)

Cargo puts the source code of the required packages inside the **~/.cargo** directory but compiles them and generates the needed libraries when it builds a project.

The Cargo tool is the recommended way of building and maintaining your Rust projects and will be used extensively in this series of tutorials about the Rust programming language.

Counting characters

The WC project (which you can find at **linuxformat.com/archives?issue=210**) can be created with:

```
$ cargo new WC --bin
$ cd WC/
$ vi src/main.rs
```

You won't need to make any changes to the contents of **Cargo.toml** for the WC project. The Rust code in **./src/main.rs** counts the total number of the characters in a text file. The name of the text file is given as a command line argument to the Rust program. (*See far left, for the WC project running in Rust.*) As you can also see (*again, far left*) the Rust compiler is complaining because it prefers project names to be in lowercase characters.

Congratulations, you just developed a naive version of the *wc* command line utility. Together we'll learn more about File I/O in Rust in the next part. ■

Rust: Functions and modules

It's time to brush off a few more flakes of Rust and uncovers error handling, functions and module in the latest trendy language

This time, we'll be using the tutorial to help you become more comfortable with Rust Functions, Modules and Error Handling as well as some other Rust features. We're going to cover how to develop and package your functions and make them autonomous entities that you can easily transfer to other computers. You'll also learn how to deal with errors, because bad things (typos! – Ed) happen all the time. Last, we'll cover pointers and vectors.

In the previous Rust tutorial, we used Rust 1.6 but this one will use Rust version 1.7 which has just come out. The first function that you will have to implement is main() and the function has the following structure:

```
fn main() {
... Your Rust Code Here ...
}
```

A function declaration in Rust always begins with the `fn` keyword. As you can see (*above*), the `main()` function has the simplest possible declaration as it neither takes any input nor returns any values. Every autonomous Rust program should have a main() function because it is where the execution of the program begins. The following couple of lines of Rust code show a function example that returns the distance between two points in the plane given their x and y coordinates:

```
fn distance_2D( x1: i32, x2: i32, y1: i32, y2: i32) -> f64 {
   let distance: f64 = (((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1)) as f64).sqrt();
   return distance;
}
```

```
rMacBook:code mtsouk$ rustc functions.rs
functions.rs:1:1: 4:2 warning: function `distance_2D` should have a snake case name such as `distance_2_d`, #[warn(non_snake_case)] on by default
functions.rs:1 fn distance_2D( x1: i32, x2: i32, y1: i32, y2: i32) -> f64 {
functions.rs:2     let distance: f64 = (((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1)) as f64).sqrt();
functions.rs:3     return distance;
functions.rs:4 }
rMacBook:code mtsouk$ vi functions.rs
rMacBook:code mtsouk$ rustc functions.rs
functions.rs:39:39: 39:50 error: unresolved name `distance_2D` [E0425]
functions.rs:39     println!("The distance is {:.5}", distance_2D(x1, x2, y1, y2));
                                                     ^~~~~~~~~~
<std macros>:2:25: 2:56 note: in this expansion of format_args!
<std macros>:3:1: 3:54 note: in this expansion of print! (defined in <std macros>)
functions.rs:39:5: 39:68 note: in this expansion of println! (defined in <std macros>)
functions.rs:39:39: 39:50 help: run `rustc --explain E0425` to see a detailed explanation
error: aborting due to previous error
rMacBook:code mtsouk$ vi functions.rs
rMacBook:code mtsouk$ rustc functions.rs
rMacBook:code mtsouk$ ./functions
Current value of pair: (3, 1)
Min Max returns: (1, 3)
The sum is: 4
Does nothing!
do_nothing returns nothing!
The distance is 1.41421
0 * 2 = 0
1 * 2 = 2
```

Quick tip

Error handling is important because instead of letting the compiler decide what to do in case of an error, it lets you handle the error situation the way you want it!

› **This screenshot shows the output as well as the compilation process of functions.rs**

The `distance_2D()` function takes four arguments as input and returns one. The key point here is realising that the square root of a positive integer is rarely an integer so the return value of `distance_2D()` should be a floating point number. As the `sqrt()` function is imported by default, you don't need to take any extra action to use it. Using `distance_2D` with an i64 variable instead of a i32 one will produce this error:

```
functions.rs:47:63: 47:65 error: mismatched types:
 expected `i32`,
    found `i64` [E0308]
```

In other words, Rust has strict rules! Last, if a function isn't used, the Rust compiler will generate a warning message:

```
functions.rs:6:1: 9:2 warning: function is never used:
`distance_2D`, #[warn(dead_code)] on by default
```

Please note that the declaration of the arguments of a function is compulsory. Additionally, Rust functions return exactly one value but you can get away with this limitation with the help of tuples and other data structures.

You also need to bear in mind that functions don't return an actual value, return the unit type `()`. You can check such a return value as follows:

```
if do_nothing() == () {
    println!("do_nothing returns nothing!");
}
```

Rust tuples

A tuple is an ordered list of a fixed size of values of different types that offers a convenient way to pack and use multiple variables as a set. Defining a tuple is as simple as writing the following code:

```
let pair = (1, 3);
println!("Current value of pair: {:?}", pair);
```

The last command shows that you can print a tuple as a usual variable. A function that returns a tuple should be defined as follows:

```
fn sum_ab(numbers: (i32, i32)) -> i32 {
    return numbers.0 + numbers.1;
}
```

The code also reveals that you can access the two individual values of the numbers tuple as `numbers.0` and `numbers.1`. Additionally, a function that both requires a tuple as its first argument and returns a tuple should be defined as follows:

```
fn min_max(pair: (i32, i32)) -> (i32, i32) {
    let (a, b) = pair;
    if a > b {
        return (b ,a);
    }
    return pair;
}
```

The first command of the `min_max()` function demonstrates that you can access the fields of a tuple using a destructuring `let`, which allows you to assign them into new variables. The **tuples.rs** file contains all of the examples related to tuples.

A more complex function

In this section you will learn about function pointers.

The following code illustrates the process where `double_int()` is a real function and `x` is the function pointer:

```
fn double_int(x: i32) -> (i32) { x+x }
let x: fn(i32) -> i32 = double_int;
for i in 0..10 {
    println!("{} * 2 = {}", i, x(i));
}
```

From now on you can use `x` to call the `double_int()` function. The problem with function pointers is that they usually reduce the readability of code and can create nasty bugs, especially when you are trying to figure out where exactly the variable points to. All the example code related to functions can be found in the **functions.rs** file. (See left to see the **functions.rs** code in action.) As you will see from the warning messages of the Rust compiler, Rust would prefer to name `distance_2D` as `distance_2_d` which is left as an exercise for the reader.

Error handling

Although the error handling capabilities of Rust are marvellous, sometimes compiler warnings can be very useful (as you can see, below). However, warning messages don't prevent the compiler from generating an executable file that can crash. Imagine that you are trying to open a file for reading but for some reason the operation fails. In such situations, you usually want the program to be able to handle »

The .unwrap() method

The unwrap() method takes a value out of an option type and allows you to use the actual value. This method is mainly used for getting strings, vectors and unique pointers out of option types without the need to copy them.

The following slice of Rust code demonstrates the usefulness of the .unwrap() method to read a command line argument:

```
let filename = &env::args().nth(1).
unwrap();
```

Note: If the value is 'None', your program will panic. The similar unwrap_or() allows you to handle related situations as it provides a default value.

Quick tip

Modules are useful but don't create one just for storing a single function. Also, don't use them when there's no relation between the functions you have developed, eg don't put a sorting function and a Fibonacci-calculating function in the same module. In all other cases, you are free to use a module!

```
2. bash
rMacBook:code mtsouk$ rustc functions.rs
functions.rs:1:1: 6:2 warning: function is never used: `distance_2D`, #[warn(dead_code
)] on by default
functions.rs:1 fn distance_2D( x1: i32, x2: i32, y1: i32, y2: i32) -> i32 {
functions.rs:2     let mut distance = 0;
functions.rs:3     distance = 1;
functions.rs:4
functions.rs:5     return distance;
functions.rs:6 }
functions.rs:1:17: 1:19 warning: unused variable: `x1`, #[warn(unused_variables)] on b
y default
functions.rs:1 fn distance_2D( x1: i32, x2: i32, y1: i32, y2: i32) -> i32 {
                               ^~
functions.rs:1:26: 1:28 warning: unused variable: `x2`, #[warn(unused_variables)] on b
y default
functions.rs:1 fn distance_2D( x1: i32, x2: i32, y1: i32, y2: i32) -> i32 {
                                        ^~
functions.rs:1:35: 1:37 warning: unused variable: `y1`, #[warn(unused_variables)] on b
y default
functions.rs:1 fn distance_2D( x1: i32, x2: i32, y1: i32, y2: i32) -> i32 {
                                                 ^~
functions.rs:1:44: 1:46 warning: unused variable: `y2`, #[warn(unused_variables)] on b
y default
functions.rs:1 fn distance_2D( x1: i32, x2: i32, y1: i32, y2: i32) -> i32 {
                                                          ^~
functions.rs:2:9: 2:21 warning: value assigned to `distance` is never read, #[warn(unu
sed_assignments)] on by default
functions.rs:2     let mut distance = 0;
                       ^~~~~~~~~~~~
functions.rs:1:1: 6:2 warning: function `distance_2D` should have a snake case name su
ch as `distance_2_d`, #[warn(non_snake_case)] on by default
functions.rs:1 fn distance_2D( x1: i32, x2: i32, y1: i32, y2: i32) -> i32 {
```

The warning messages generated by the Rust compiler are very useful

» the error and print information to help you understand why the error happened. The following example code resolves such problems:

```
let file = match File::open(&filename) {
            Err(why) => panic!("couldn't open {}: {}", filename,
Error::description(&why)),
   Ok(file) => file,
};
```

All the methods related to the File structure return the io::Result<T> type that's an alias for Result<T, io::Error>. The complete Rust code can be found at **error_handle.rs**.

You can see three executions of **error_handle.rs** (pictured p116), where the file you want to open is OK, cannot be read due to inadequate Linux permissions or doesn't exist at all.

Developing modules and crates

A crate is a package that you can download and compile on your own Linux machine. The definitive place to search existing crates is **https://crates.io**. Once you find a package that you want to use, you can include it in the **Cargo.toml** file of your Cargo project. In contrast, a module is a way of grouping elements of related functionality including functions and constants. The presented module will include simple functions just to illustrate the development of a module but it will also show you how to include a module into another module! The following commands show how you can create the skeleton of a module named lxf using Cargo:

```
$ cargo new lxf
$ cd lxf
$ vi src/lib.rs
$ vi src/lxf.rs
```

Modules are libraries, which means that instead of **main.rs**, you will have to edit **lib.rs** and execute `cargo` without the `--bin` command line option.

Note that by default all items in a module are visible inside the module only. The 'pub' modifier can make an item visible to the outside world. (The screenshot top right shows the source code that can be found in `lib.rs` and `lxf.rs`, where you can also see the use of the 'pub' keyword.) The use of a separate file that keeps the Rust code of the lxf module (named after the module name) isn't mandatory, but it makes your life easier when you have big projects. You will now have to create a new project that will use the lxf module:

```
$ cargo new aa --bin
$ cd aa
$ vi src.main.rs
```

The contents of lxf directory (pictured top right) includes the compilation process of the 'aa' project, the contents of **Cargo.toml** and **main.rs** and the output from the executable file.

Please pay special attention to the *Cargo.toml* file of the aa project because this is where you tell your project where to find the desired module (lxf) on your local machine. Also note the `extern crate lxf;` line inside the **main.rs** file.

Pointers in Rust

If you are familiar with the C programming language, you will already know that pointers can create unsafe code and bugs. The good news is that Rust is developed in a way that you will not need to use pointers very often even if they are supported. Rust supports four pointer types that can be categorised in two sets: references and raw pointers.

De-referencing a pointer in Rust is done with an asterisk (*) whereas de-structuring a pointer can be done by using an ampersand (&), ref or ref mut. Both types of pointers can be de-referenced using the * operator. However, for raw pointers you will have to use an unsafe block. Additionally, both references and raw pointers can be created using the & operator. In the next Rust tutorial, we will talk more about the different kinds of pointers but you will see pointers in action in the next section.

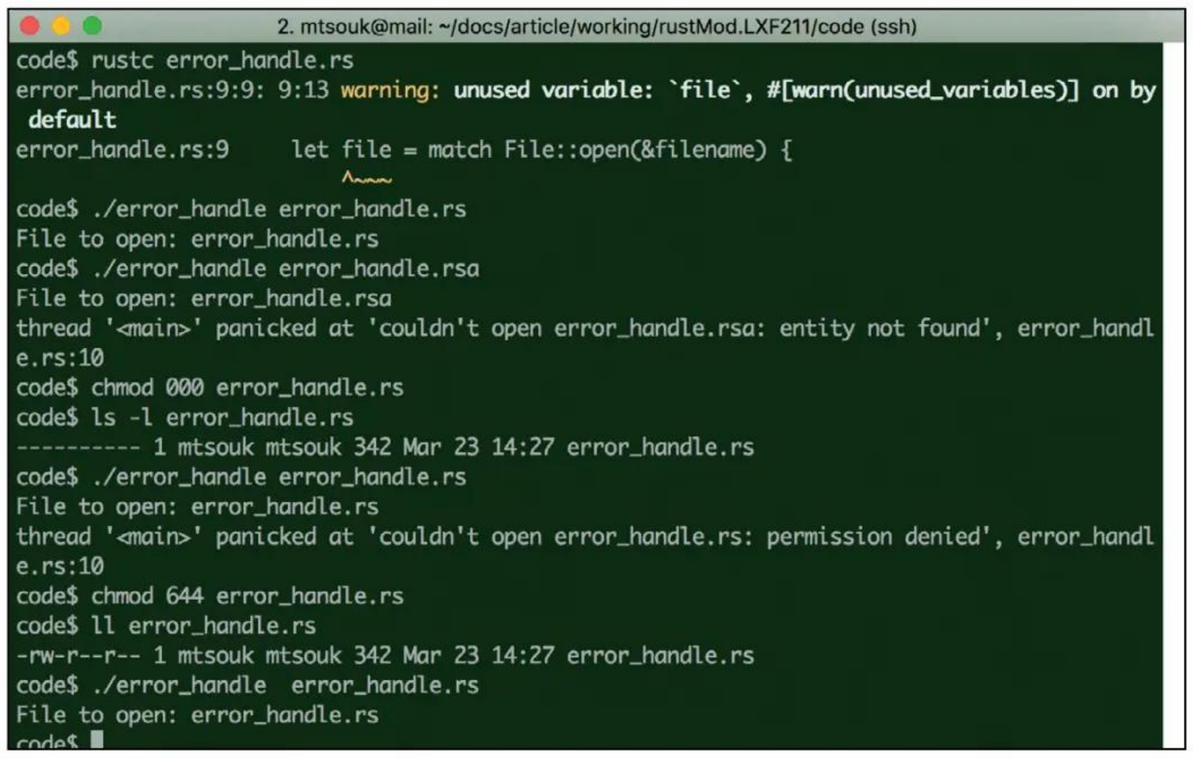
Ownership and borrowing

Rust includes the notion of ownership and borrowing, which can be found in research languages. Both can help you achieve memory safety because they aid in defining who can use what and where. Ownership means that the pass of a variable also passes the ownership of the value to the new variable. Borrowing is taking control over a variable for a while in order to do something with it and then returning the ownership of the variable back. This allows you to have multiple references to a variable; however, only one reference can be mutable at any given time!

Look at the following code (below) which appears fine:

```
let my_vector = vec![1, 2, 3, 4, 5];
let another_vector = my_vector;
println!("my_vector is {:?}", my_vector);
```

However, if you try to compile this code it will generate an error message like this:



› This screenshot shows the error handling capabilities of Rust when trying to open a text file for reading

Static global variables

Static global variables are similar to constants, but the program uses the memory address of the variable instead of its value. Due to their nature, static global variables can be mutable but this is an 'unsafe' operation and should be used in an 'unsafe' block of code. The following code defines a Static Global variable:

```
static AUTHOR: &'static str = "Mihalis Tsoukalos";
```

As you can see, the developer must specify the type of a static global variable for it to be valid. Most of the time you're only going to need to use a 'regular' const variable, which behaves in a similar way to a C language #define statement.

```
2. mtsouk@mail: ~/docs/article/working/rustMod.LXF211/code (ssh)
lxf$ cat src/lib.rs
pub mod lxf;
lxf$ cat src/lxf.rs
pub static AUTHOR: &'static str = "Mihalis Tsoukalos";

pub mod issue211 {
        pub fn pages() {
                println!("Issue 211 has 100 pages!");
        }
}

pub fn message() {
        println!("Hello from LinuxFormat magazine!");
        println!("The author of this module is {}", AUTHOR);
}
lxf$ ll
total 12
-rw-r--r-- 1 mtsouk mtsouk   39 Mar 23 15:13 Cargo.lock
-rw-r--r-- 1 mtsouk mtsouk   89 Mar 23 15:51 Cargo.toml
drwxr-xr-x 2 mtsouk mtsouk 4096 Mar 23 16:09 src
lxf$
```

```
2. mtsouk@mail: ~/docs/article/working/rustMod.LXF211/code (ssh)
lxf$ ll
total 12
-rw-r--r-- 1 mtsouk mtsouk   39 Mar 23 15:13 Cargo.lock
-rw-r--r-- 1 mtsouk mtsouk   89 Mar 23 15:51 Cargo.toml
drwxr-xr-x 2 mtsouk mtsouk 4096 Mar 23 16:09 src
lxf$ cargo build
   Compiling lxf v0.1.0 (file:///home/mtsouk/docs/article/working/rustMod.LXF21
lxf$ cd ../aa/
aa$ cat Cargo.toml
[package]
name = "aa"
version = "0.1.0"
authors = ["mtsouk"]

[dependencies]
lxf = { path = "./../lxf" }
aa$ cat src/main.rs
extern crate lxf;
```

› This screenshot shows the contents of the Cargo.toml and ./src/main.rs files as well as the output of the generated executable from the aa project

› This screenshot shows the contents of the ./src/lib.rs and ./src/lxf.rs files used in our lxf project

```
pointers.rs:10:32: 10:41 error: use of moved value: `my_vector` [E0382]
...
pointers.rs:8:6: 8:20 help: if you would like to borrow the value instead, use a `ref` binding as shown:
pointers.rs: let ref another_vector = my_vector;
```

For primitive data types, which are cheaper, Rust performs a full copy but for other types, such as a vector, it's not allowed to use a variable after you've assigned it to another variable. So, the 'correct' version of the code is the following:

```
let my_vector = vec![1, 2, 3, 4, 5];
let ref another_vector = my_vector;
println!("my_vector is {:?}", my_vector);
```

However, if you want the same effect for a primitive type that is, not being able to use a variable after it has been assigned to another variable you will need to `Box` it first:

```
let integer = Box::new(12345);
let my_integer = integer;
// You cannot use this!
println!("integer is {}", *integer);
```

The following code illustrates the notion of borrowing:

```
let mut a_var = 3.14;
{
        let b_var = &mut a_var;
        *b_var = 3.14159;
}
println!("a_var is now {}", a_var);
```

If you remove the curly braces ({}), the code will not compile because you will have two variables (`a_var` and `b_var`) pointing to the same memory location! The curly braces make `b_var` a local variable that references `a_var`, changes its value and returns the ownership back to `a_var` as soon as you reach the end of the block. Again, all these happen in order to be safe and not allow the misuse of a variable. The **pointers.rs** file contains Rust code that showcases ownership and borrowing.

Vectors in Rust

Arrays are good but have a major limitation: they have a fixed size. When you want a Rust data structure that can grow dynamically, you should use a vector. The following Rust code, included in **vectors.rs**, shows how to initialise a vector as well as how to access all its elements:

```
let mut a_vector = Vec::new();
a_vector.push(2);
a_vector.push(-2);
for x in a_vector.iter() {
        println!("{}", x);
}
```

```
for i in 0..a_vector.len()
{
        println!("{}", a_vector[i]);
}
```

You need the `.push()` method to add a value to a vector and you need the `Vec::new()` method to declare a new vector variable. The elements of a vector variable can be accessed as if the vector was an array so `a_vector[0]` is still the first element of the `a_vector` variable. Additionally, the code shows how to access all elements of a vector using two methods: an iterator or as an array using an index. The following code can crash your program because the `a_vector` vector has only four elements:

```
println!("{}", a_vector[5]);
```

The error message you will get is:

```
thread '<main>' panicked at 'index out of bounds: the len is 4 but the index is 5', src/libcollections/vec.rs:1134
```

The next Rust code presents a solution to the problem:

```
match a_vector.get(5)
{
        Some(x) => println!("The value of a_vector[5] is: {}", x),
        None => println!("a_vector[5] is not defined!")
}
```

This code will not crash your program because it checks whether the requested vector element exists or not. You can also use the len() function to get the size of a vector and see if an index is out of bounds, but the previous technique is more elegant. You can remove an element from a vector using the pop() method which is pretty straightforward. However, the next output shows that the return data type of pop() can be quite strange:

```
an_element is Some(12)
```

When the given index is valid, the pop() function returns `Some` with the element under that index. If the index is not valid it returns `None`; `None` and `Some` are the variants of the enum. In other words, a value with type Option<T> can either be a None or a Some.

The `Some` type contains that actual value of type T that you need to extract. So, the correct way to pop() an element from a vector is like this:

```
match a_vector.pop()
{
        Some(x) => println!("x is {:?}", x),
        None => println!("It is not defined!")
}
```

As a result, if the vector is empty and you try to use `pop()` then you had better use the match block to make things more robust. You might think that you will need to write more code in Rust, but in reality you'll be writing much higher quality code! ■

Rust: Speeding up file I/O

In this part of the Rust series, we explain how to use file I/O in your Rust programs for such things as processing text files

Hopefully, you'll be feeling more comfortable with Rust, which means it's time to deal with a subject that's very important in systems programming and Operating Systems: file I/O. Rust's strict programming rules come in handy when dealing with files because it reduces errors and bugs and allows you to concentrate on what really matters—which is the algorithm and the operations that you want to perform. First, we'll present some simple examples before continuing with more advanced topics, such as processing text files and memory-mapped file I/O. This tutorial will also cover some important Rust topics, including pointers, pattern matching and the usize and isize types. If at any time you feel that you need some help, you should review the previous two Rust tutorials [over the last eight pages] or visit the Rust documentation web page. So, without further ado, let's get started!

Command-line arguments

The following Rust code, we've called **arguments.rs**, prints all command-line arguments given to a program including the name of the executable:

```
use std::env;
fn main()
{
        let mut counter = 0;
        for argument in env::args()
        {
                counter = counter + 1;
           println!("{}: {}", counter, argument);
        }
}
```

As you will see in the rest of the tutorial, you can directly access a specific command-line argument. However, the presented technique is generic, simple and elegant.

Quick tip

File I/O is a crucial part of every OS because slow and unreliable file I/O can destroy an OS or a database server. This means that you not only need to make I/O operations fast, but, most importantly, reliable. The key to speed and reliability is simplicity!

A simple example

The following Rust code, called **simple.rs**, demonstrates how to open a file for reading and creating an empty file for writing, which are the two simplest forms of file I/O:

```
use std::env;
use std::fs::File;
use std::path::Path;
use std::error::Error;

fn main()
{
   let args: Vec<_> = env::args().collect();
   let open = ::std::env::args().nth(1).unwrap();
   let write = ::std::env::args().nth(2).unwrap();

   let path = Path::new(&open);
   let display = path.display();

   let mut file = match File::open(&path)
                        {
                        Err(why) => panic!("couldn't open {}: {}",
display, Error::description(&why)),
      Ok(file) => file,
   };

   let write_path = Path::new(&write);
   let write_display = path.display();
   let mut write_file = match File::create(&write_path)
                        {
                        Err(why) => panic!("couldn't create {}:
{}", write_display, Error::description(&why)),
            Ok(write_file) => write_file,
            };
}
```

Both filenames are given as command-line arguments to **simple.rs** and converted to Path variables. Our example also includes error-handling code, which is necessary because bad things can happen when dealing with files and file permissions. The `File::create()` method opens a file in write-only mode. If the file already exists, then its contents will be completely deleted, otherwise an empty file will be created (see the bottom of p116 for **simple.rs** in action). There exist many crates that allow you to perform file I/O. The general rule is to use the library functions that you know best.

The presented program will use command-line arguments to get the name of the file you want to copy as well as the filename of the new file. In the Rust code of **cp.rs** (on p116), you can see various executions of it including situations where error messages are produced as a result of various error conditions that are caught by Rust. As you can see, **cp.rs** uses fs::copy to create the copy of a file.

Pattern matching

We can now go on to learn about pattern matching in Rust as it can be very useful in many situations and especially when you want to process text files and examine their contents. Pattern matching in Rust happens with the help of the 'match' keyword, as you briefly saw in error handling code. A match statement must catch all possible values of the variable used

The usize and isize types

Put simply, both usize and isize types are pointer-sized integer types—their actual size in bits depends on the architecture used.

The problem that usize and isize types solve is having a type that's big enough to hold any pointer or any offset in a data structure. This is an elegant solution to the presented problem because both usize and isize types are related to the architecture of the current system. On the other hand, an u64 variable is always 64 bits.

However, bear in mind that the range of numbers that usize and isize can represent can change because it's based on the platform you are compiling your code for so don't make any assumptions about their size.

so having a default branch at the end is very common.

The default branch is defined with the help of the underscore character (' `_` ') which is a 'catch all'. However, in some rare situations, such as when you examine a condition that can be either true of false, a default branch isn't needed. The Rust code (below) showcases the use of pattern matching in Rust:

```
match an_age
{
        0 => “Too young!!”,
        -1 | -2 => “at the age of cheating!”,
        1...18 => “Still very young!”,
        18...25 => “At the age of studying!”,
        ready_for_work @ 26...40 => “At the age of working!”,
        _ if (an_age % 10 == 0) => “at a new decade!”,
        _ => “Unknown age!”
}
```

As you can see, you can use ranges with the help of an ellipsis (' `...` ') or multiple values as a list with the help of the ' `|` ' character. Additionally, you can give a range a name, as it happens in the 26-40 range that has the name `ready_for_work` . Pattern matching also works on tuples and other types of variables. Executing the previous code, saved in **reg_exp.rs**, produces the following output:

```
25: you are At the age of studying!
26: you are At the age of working!
...
40: you are At the age of working!
41: you are Unknown age!
...
50: you are at a new decade!
```

As you can see from the output, the order you put the various conditions plays a key role because only the first match gets executed as happened with the age of 40. If you're interested in regular expressions, then things are a little more complicated because you will have to use a separate create, called regex, to do your job. Regular expressions and pattern matching play a key role in modern programming languages, so you should try as many examples as possible in order to really understand how both concepts work in Rust.

Working with plain text files

This section will teach you how to process text files. The following code, which is part of **match_me.rs**, shows you a handy technique for iterating over a text file line by line:

```
let input_path = ::std::env::args().nth(1).unwrap();;
let mut file = BufReader::new(File::open(&input_path).
unwrap());
for line in file.lines()
{
        let myLine = line.unwrap();
        println!("{}”, myLine);
}
```

The following Rust code demonstrates how to select the lines that contain a given static string that's provided as a command-line argument to the program:

```
if my_line.contains(&string_to_match)
{
        println!("Found it!");
}
```

This code shows that sometimes a different technique can be simpler and more efficient than the use of pattern matching. In this case, the `contains()` function does the work for you much more elegantly by trying to find the given text inside its input. Using a regular expression or pattern matching would have been unnecessarily complicated.

All lines that match the given static text are saved in another file that's also given as a command line argument to **match_me.rs**. The full code is in the code pack download. The following output shows **match_me.rs** in action when processing itself:

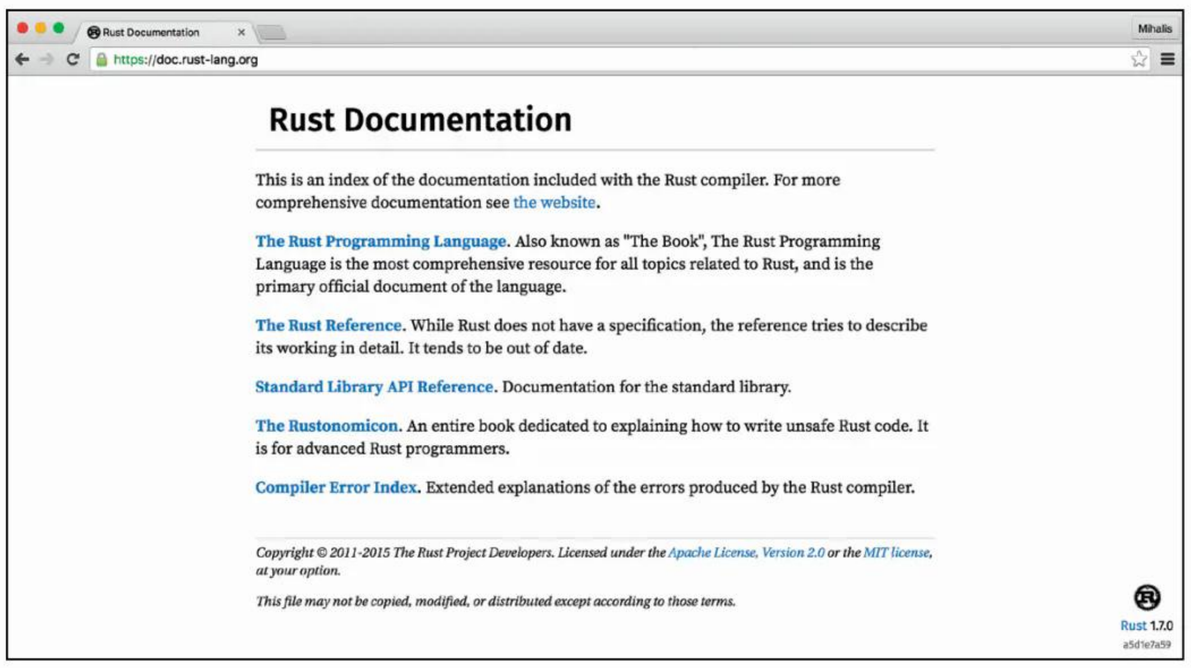
```
$ ./match_me
thread ‘<main>’ panicked at ‘Usage: name input output
string’, match_me.rs:13
$ ./match_me match_me.rs outputFile total_lines
Lines processed: 41
$ cat outputFile
        let mut total_lines = 0;
                total_lines = total_lines + 1;
        println!("Lines processed: {}”, total_lines);
```

Quick tip

Rust has the habit of generating lots of error messages and refusing to compile your code when error messages exist. This isn't necessarily bad because Rust code that compiles is highly likely not to have any silly bugs! The Rust motto is that 'If your program compiles, then it is highly unlikely to crash afterwards'. Not a catchy motto but quite wise.

Memory mapped file I/O

Memory mapped I/O is a pretty specialised technique that connects a file on disk with a buffer in memory. This is done in such a way that when you get bytes from the buffer, the matching bytes of the file are actually being read or written. »



› This is the main Rust documentation web page where you can find useful tutorials and references. However, nothing can replace practice!

» One way to use memory mapped file I/O in Rust is with the help of the mmap crate. The following Rust code, which is a part of the memoryMapped Cargo project, illustrates the use of mmap crate:

```
let mmap_opts = &[
    MapOption::MapNonStandardFlags(libc::MAP_SHARED),
    MapOption::MapReadable,
    MapOption::MapWritable,
    MapOption::MapFd(f.as_raw_fd()),
];
```

The libc crate is a Rust library for types and bindings to native C functions. The libc crate not only supports functions but also constants and type definitions. Its main purpose is allowing Rust developers to work with C code on all platforms supported by Rust by providing a straight binding to the actual system functions on the current platform.

You can find more information about the libc crate at **https://crates.io/crates/libc**. This will show you the definition of the MAP_SHARED constant (defined in **https://doc.rust-lang.org/libc/x86_64-unknown-linux-gnu/libc/constant.MAP_SHARED.html**), ie:

```
pub const MAP_SHARED: c_int = 0x0001
```

Unsafe blocks

As shown (see right) in the documentation page for the libc::chmod function for the linux-gnu platform, the c_char type is a binding to the C char type. Similarly, the c_int type is a binding for the C int type. Executing the memoryMapped project generates a new file and its name is given as a command-line argument:

```
$ cargo run newFile
$ ls -l newFile
-rw-r--r--  1 mtsouk  mtsouk  262145 Apr 11 09:30 newFile
$ head -1 newFile
Hello LXF from Mihalis Tsoukalos!
$ file newFile
newFile: ASCII text
```

It's impressive that the generated file is much bigger than the included text! A useful fact regarding memory-mapped file access is that the memory buffer that you create on your program doesn't occupy any actual memory space because it's directly associated with the disk file. The biggest advantage you get from using memory mapped file I/O is increased performance, especially when dealing with large files. However, the price you pay for better performance is having to write more lines of relatively complex code.

The last part of the **main.rs** file contains the following code which is an unsafe block and will, therefore, need some explanation:

```
unsafe
{
    ptr::copy(src_data.as_ptr(), data, src_data.len());
}
```

First of all, the sample code shows how to use the unsafe keyword and define an unsafe block. It's important to understand that the Rust compiler isn't always able to verify the safeness of your code so wrapping your code using the unsafe keyword tells the compiler not to worry about the code and that you, the developer, are responsible!

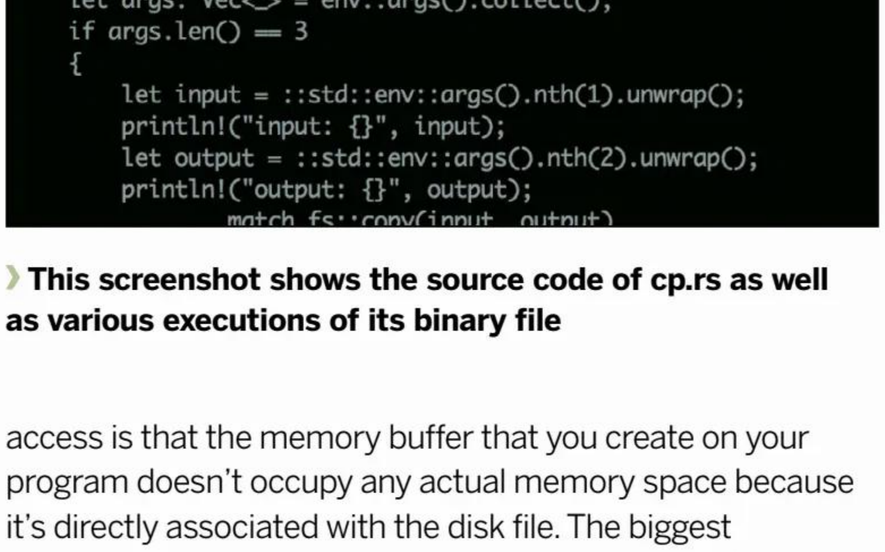
As a result, situations such as deadlocks, integer overflow, divide by zero and memory leaks aren't good in unsafe blocks of code. This type of code blocks should be used for accessing or updating static mutable variables, dereferencing raw pointers and, as in the example (above), calling unsafe functions. What unsafe functions usually do is trade safety for speed but they expect the programmer to be fully aware of what they are doing.

Module documentation

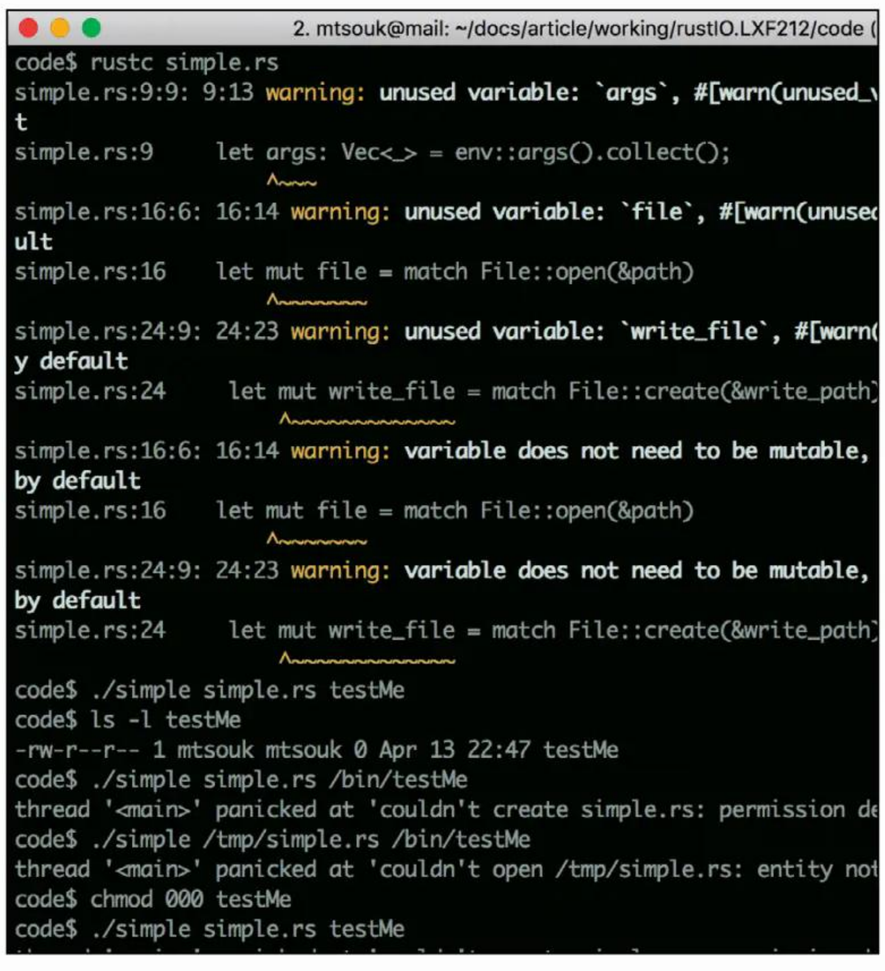
Now, we'll show you how to produce documentation for the memoryMapped project. You need the documentation where the Rust code is, eg to add documentation for **main.rs**, you should edit **main.rs**. Documentation information begins with either '///' or '//!' and is used by the *rustdoc* command-line utility. However, the Rust compiler ignores such information

Quick tip

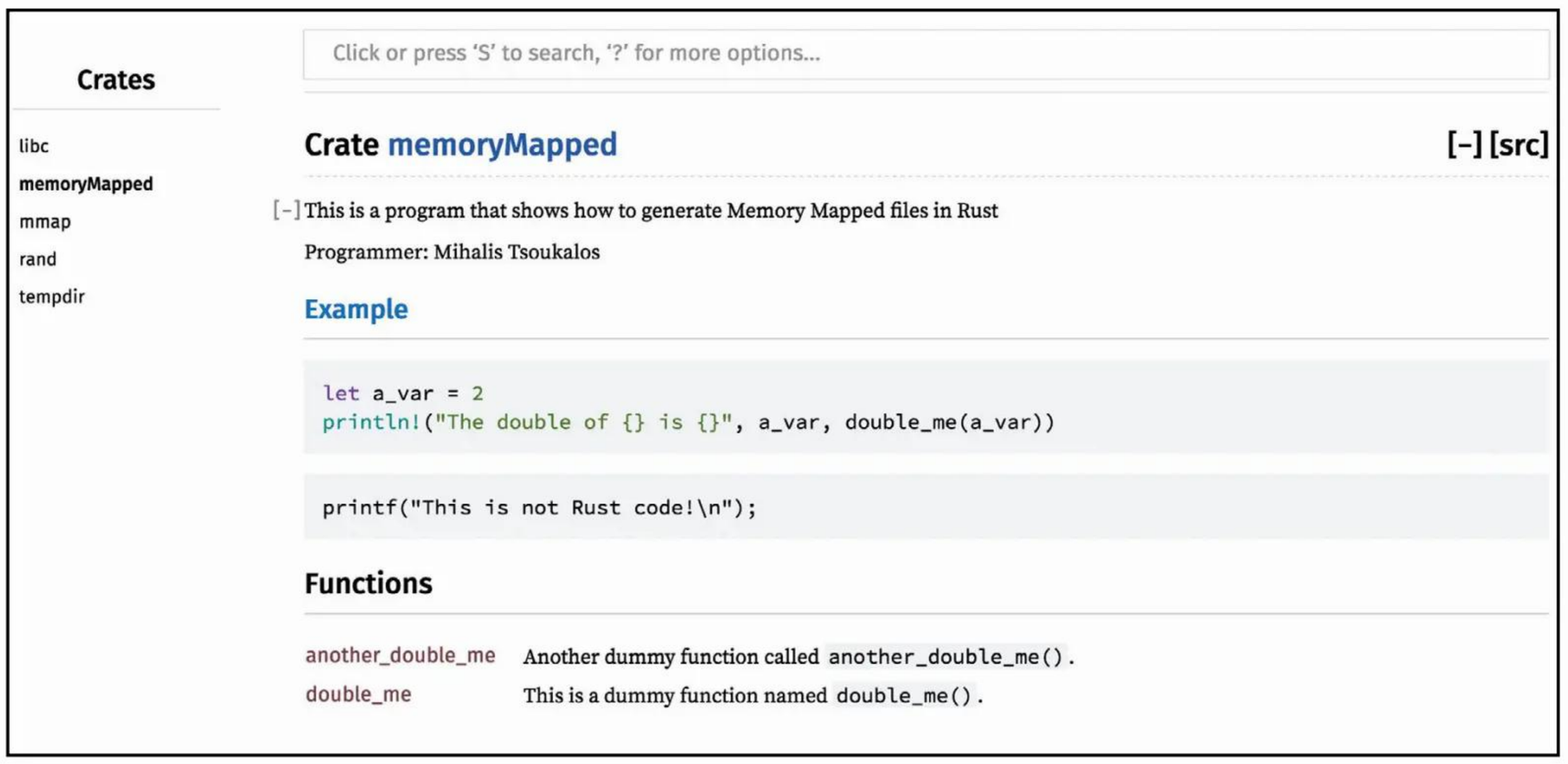
Rust has a steep learning curve However, you will find the Rust way of programming better, more secure and less annoying. The way Rust deals with variables allows it to avoid garbage collection and null errors, which will improve your code's quality.



› This screenshot shows the source code of cp.rs as well as various executions of its binary file



› Here we see simple.rs in action as well as various error messages that are produced due to UNIX file permission restrictions

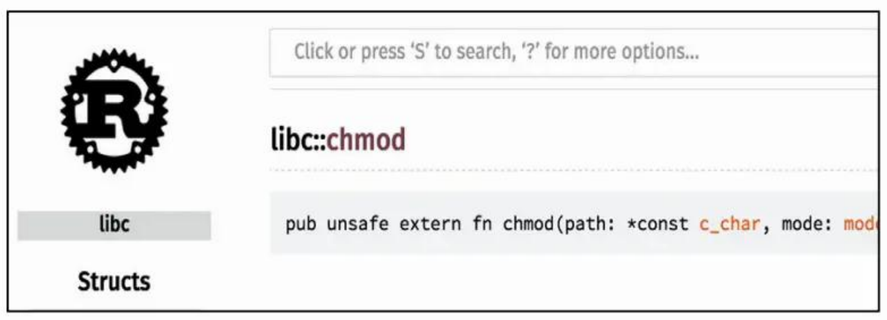


› **The *rustdoc* utility, as well as the cargo doc command, can generate HTML documentation for you Rust code as long as you put the required information inside of your Rust code**

because each line that begins with '//' is considered a comment by the compiler. The information that begins with '//!' applies to the entire code file whereas anything beginning with '///' is considered local information.
If you are using *Cargo* to build your project, then instead of using *rustdoc* to generate documentation you should execute `cargo doc`.

The output files will be generated inside the **./target/doc** directory inside your project directory. For the memoryMapped project, you should start with the **./target/doc/memoryMapped/index.html** file (which is pictured above).

There are four special headers: The example header – that's used in **main.rs** – must contain valid Rust code, the panics header, the safety header and the failures header.



› **This is the documentation for the libc::chmod function of the libc crate, which enables you to call C system functions on all platforms supprted by Rust**

In order to include Rust code in an example header, you should embed it in triple slashes. Should you wish to include other kinds of code, you should use the following notation:

```
//! ```c
//! printf("This is not Rust code!\n");
//! ```
```

It's important to use the connect notation because rustdoc won't be able to check the correctness of your Rust code and make sure that the code isn't outdated or incorrect! In case of an error in the format of the documentation information, the error messages that will be generated will be similar to the following example:

```
src/main.rs:36:8: 36:32 error: expected one of `.`, `;`, `}`, or an operator, found `/// Inline documentation`
src/main.rs:36        x * 2 /// Inline documentation
```

The following documentation code will print the double_me() string in a different font (as you can see pictured above):

```
/// This is a dummy function named `double_me()`.
```

The approach that Rust takes is actually quite useful because you write your documentation near your code. However, this assumes that you're willing to write documentation for your code! Rust generates local documentation information for all crates that are used by the project (as you can also see pictured above).

What you should remember from this section is that the documenting of Rust code begins with either a triple slash ('///'), a double slash ('//') or an exclamation mark. ■

More about pointers

In the previous tutorial, we covered how Rust supports four pointer types that can be categorised in two sets: references and raw pointers.

Raw pointers are defined as ***const T** and ***mut T**; you can find in-depth guidance on raw pointers at **http://doc.rust-lang.org/std/primitive.pointer.html**.

When you dereference a raw pointer you should use an unsafe block, because there's no guarantee that the pointer is pointing somewhere safe. If you don't, Rust will generate an error message, such as:

```
pointers.rs:6:34: 6:46 error: dereference of raw pointer requires unsafe function or block [E0133]
```

There are also mutable references that allow you to mutate the variable you are borrowing.

The following code shows that you need to include an extra block because you cannot use `another_var` while `yet_an_var` is in scope:

```
let mut another_var = 15;
{
   let yet_an_var = &mut another_var;
   *yet_an_var += 1;
}
```

Please note that you should use `mut` for both `another_var` and `yet_an_var` as you are going to change the value of `another_var`.

If you want to see this in action, the **pointers.rs** file contains the full example code.

Rust: Network programming

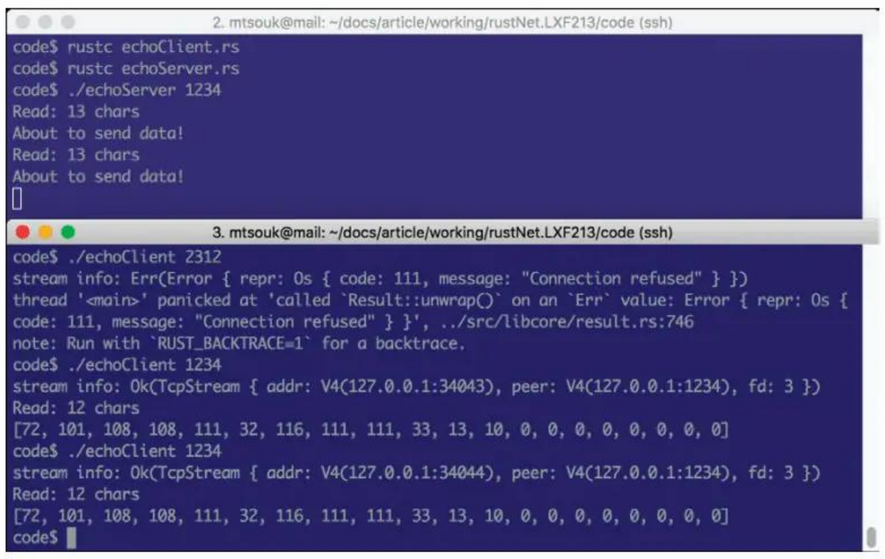
Our Rust series continues with a move to explain just how to start writing network applications with a basic TCP client and server

For this tutorial, we're covering network programming in Rust and it's not an easy subject, but Rust and its modules are here to make your life easier. Remember that if you have any problems using a Rust method, the best place to look for information is **https://doc.rust-lang.org**. A big chunk of the tutorial will be on programming a TCP client and a TCP server in Rust as well as a UDP server. After that, we'll cover using the HTTP protocol in Rust. Last, you will learn more about the various versions of Rust that are available at any given moment. Note: There's a small change since the last Rust tutorial—the latest version of Rust is now 1.8, but, without further ado, let's get started!

TCP/IP is a family of protocols that help the internet operate. The name, TCP/IP comes from the two most well-known protocols, TCP and IP. Every device that uses TCP/IP must have an IP address, which must be unique, at least, to its local network. It also needs a network mask that's related to the current network, one or more DNS servers and, optionally, the IP of a device that acts as the default gateway.

The main characteristic of TCP (Transmission Control Protocol) is that it's a reliable protocol, which means that it makes sure that a packet is delivered. If there's no proof for the delivery of a packet, the packet will be retransmitted.

On the other hand, IP (Internet Protocol) isn't a reliable protocol by nature. However, a programmer can program reliable applications that use IP by implementing their own error-checking code. UDP uses the underlying IP protocol to transport a message from one machine to another and provides the same unreliable, connectionless packet delivery as IP. It doesn't use acknowledgements to make sure that a message has arrived; doesn't put messages in order and doesn't provide feedback to control the rate at which information flows between machines. Thus, UDP network packets can be lost, duplicated or arrive out of order. Furthermore, packets can arrive faster than the recipient can process them. That's enough with the theory, it's now time to start developing TCP/IP clients and servers!



This screenshot shows the TCP server and the TCP client in full swing



Most Linux server processes, including web and mail servers, work using TCP/IP in order to accept network connections from all over the world.

TCP client and server

Let's see how to develop a TCP server and a TCP client in Rust. The client sends a message that's read by the server and the server sends a message to the client—despite the fact that different messages are being transferred between the server and the client, the whole operation looks like an echo server. The Rust code for the TCP client is:

```
use std::net::TcpStream;
use std::time::Duration;

use std::io::Write;
use std::io::Read;

fn main() {

        let one_second = Duration::new(1, 0);
    let port = ::std::env::args().nth(1).unwrap();
    let ip = "127.0.0.1";
    let my_bind = ip.to_string() + ":" + &port;
    let my_bind_final: &str = &my_bind;

    let mut buf: [u8; 20] = [0; 20];
    let stream = TcpStream::connect(my_bind_final);
        std::thread::sleep(one_second);
    println!("stream info: {:?}", stream);

        let mut my_stream = stream.unwrap();
  my_stream.write(b"Hello World\r\n").unwrap();

  let _ = match my_stream.read(&mut buf) {
    Err(e) => panic!("Error: {}", e),
    Ok(m) => {
      if m == 0 {
        println!("Failed to Read!");
      }
                                println!("Read: {} chars", m);
      m
    },
  };
```

```
        println!("{:?}", &buf);
}
```

The Rust code for the TCP server is bigger than the client:

```
use std::net::{TcpListener, TcpStream};
use std::thread;
use std::io::Read;
use std::io::Write;

fn handle_request(mut stream: TcpStream) {
   ...
}

fn main() {
   let port = ::std::env::args().nth(1).unwrap();
   let ip = "127.0.0.1";
   let my_bind = ip.to_string() + ":" + &port;
   let my_bind_final: &str = &my_bind;

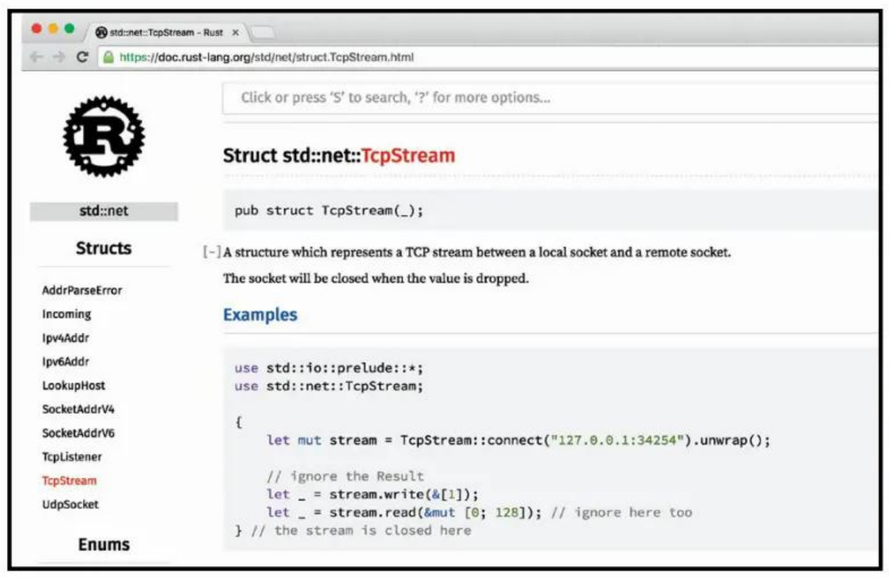
   let listener = TcpListener::bind(my_bind_final).unwrap();
   for stream in listener.incoming() {
      match stream {
         Err(e) => { println!("Failed: {}", e) }
         Ok(stream) => {
            thread::spawn(move || {
               handle_request(stream)
            });
         }
      }
   }
}
```

The **echoServer.rs** program accepts one command-line argument which is the port number the server is going to listen to. You can check the implementation of `handle_request()` to see how the TCP server interacts with a client.

We've shown an interaction between **echoClient.rs** and **echoServer.rs** using two terminals (pictured left). While the server process is always running, the client process exits as soon as it's done. Both programs follow the traditional approach of implementing their functionality without using any modules to simplify their tasks. If you have ever implemented an echo client or server in C, you will immediately understand what the Rust code does as well as the functions that are used. The single most important part of the code of a server application is where you define the port number and the IP address that the server will listen to:

```
let listener = TcpListener::bind(my_bind_final).unwrap();
```

Data is read using a **read()** call and a buffer to hold the data. Similarly, the process writes its data using a **write()** call. The Rust code of **echoClient.rs** is also pretty straightforward:



You can find more information about the various Rust functions and structures, on the Rust documentation website

it connects to the server using a **TcpStream::connect()** call, writes using a **write()** call and reads using a read() call.

If you try to run a second instance of **echoServer.rs** while another is running, you will get an error message because a TCP/IP port can only be controlled by a single process:

```
$ ./echoServer 1234
thread '<main>' panicked at 'called `Result::unwrap()` on an
`Err` value: Error { repr: Os { code: 48, message: "Address
already in use" } }', src/libcore/result.rs:746
note: Run with RUST_BACKTRACE=1 for a backtrace.
```

If you want to get more information about an error message, you can execute your binary file as `RUST_BACKTRACE=1 ./echoServer`. Of course, if you choose to use a different port number, you can run as many server processes as you want! If you try to execute **echoServer.rs** using a privileged port without having the necessary privileges, you will get a different error message:

```
$ ./echoServer 12
thread '<main>' panicked at 'called `Result::unwrap()` on an
`Err` value: Error { repr: Os { code: 13, message: "Permission
denied" } }', src/libcore/result.rs:746
```

Using UDP in Rust

Let's continue onwards, if you're going to learn how to develop a UDP server in Rust. The code for this is:

```
use std::str;
use std::thread;
use std::net::UdpSocket;

fn main() {

   let port = ::std::env::args().nth(1).unwrap();
   let ip = "0.0.0.0";
   let my_bind = ip.to_string() + ":" + &port;
   let my_bind_final: &str = &my_bind;
```



Two very good books on TCP/IP are *Internet working with TCP/IP Volume I* by Douglas E Comer and *TCP/IP Illustrated: The Protocols, Volume 1* by W Richard Stevens. The definitive guide to network programming in C is *UNIX Network Programming Volume 1, 2nd Edition* by W Richard Stevens.

Using netcat to test a client or a server

The *netcat* utility is the perfect tool for debugging TCP/IP applications, both clients and servers. The good thing is that netcat works with both UDP and TCP—telnet works with TCP connections only. The following *netcat* command makes *netcat* act as a UDP client to a server at localhost that uses port number 1234:

```
$ nc -v -u localhost 1234
```

It's worth clarifying that a server process can't know what kind of client is trying to communicate with it—the only two things that a server process knows are the IP address of the client and the port number the client is using to communicate. The following netcat command makes netcat act as a TCP client that connects to a server at localhost using port number 2345:

```
$ netcat localhost 2345
```

Combining any of the above commands with `-l` will make netcat act as a server process that waits for connections. Nevertheless, it is more rational to use nc as a client to test your servers because nc cannot easily mimic the actual functionality of a server process like a web server or a chat server.

```
let socket = match UdpSocket::bind(my_bind_final) {
    Ok(s) => s,
    Err(e) => panic!("Could not bind socket: {}", e)
};

let mut buf = [0; 2048];
loop {
    match socket.recv_from(&mut buf) {
        Ok((amt, src)) => {
            thread::spawn(move || {
                println!("Got {} bytes from {}.", amt, src);
                println!("{}", str::from_utf8(&buf).unwrap_or(""));
            });
        },
        Err(e) => {
            println!("Failed to receive data: {}", e);
        }
    }
  }
}
```

Note: The code of the server can be found in **UDPServer.rs**. The port number that will be used is given as a command-line argument to **UDPServer.rs**. (*You can see the UDP server in action, below*). The first terminal shows **UDPServer.rs** and the second terminal shows the *netcat* doing its thing, which we've used for testing the server. As soon as the UDP server receives data, it prints it on the screen along with some other useful information.

The most important part of **UDPServer.rs** is where you define the port the server is going to listen to:

```
UdpSocket::bind(my_bind_final)
```

If the `bind()` call fails, then you will have no UDP server! After having a valid socket, you can get data from it using the `recv_from()` function. The use of the 0.0.0.0 IP address for the server means that it should listen on every available network interface. This is used for illustrative reasons only. The general principle is that you should manually specify the IP for security reasons. The `recv_from()` function is used for reading the data from a UDP connection.

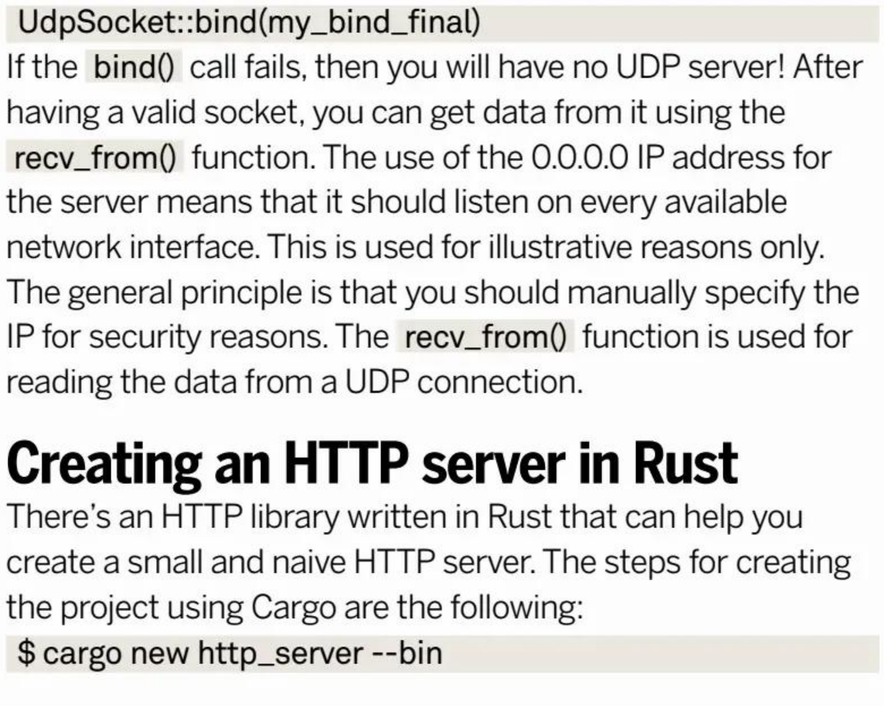
Quick tip

When creating TCP/IP server processes, remember that port numbers 0-1024 have restricted access and can only be used by the root user, which means you will need administrative privileges to use a port in that range. Running a process as root is a security risk and, therefore, should be avoided.

Creating an HTTP server in Rust

There's an HTTP library written in Rust that can help you create a small and naive HTTP server. The steps for creating the project using Cargo are the following:

```
$ cargo new http_server --bin
```



Here we see the UDP server in action with the help of the *netcat* utility that acts as the client

```
$ cd http_server/
$ vi Cargo.toml
$ vi src/main.rs
$ cargo build
```

The contents of **Cargo.toml**, where you specify the extra module that will be used is:

```
[package]
name = "http_server"
version = "0.1.0"
authors = ["mtsouk"]

[dependencies]
hyper = "*"
```

The contents of **./src/main.rs**, which is the file that contains the Rust code, is:

```
extern crate hyper;
use hyper::Server;
use hyper::server::Request;
use hyper::server::Response;

fn hello(_: Request, res: Response) {
   res.send(b"Hello World using Rust!").unwrap();
}

fn main() {
        let port = ::std::env::args().nth(1).unwrap();
        let ip_and_port = "127.0.0.1".to_string() + ":" +
&port;
        let ip_port: &str = &ip_and_port;

   Server::http(ip_port).unwrap().handle(hello).unwrap();
}
```

As you can see the `Server::http()` call defines the port number your web server is going to listen to—the port number itself is given as a command line argument that you need to process before calling the `Server::http()` function. You can see the Rust Web Server with the help of the *wget* utility.

Fetching a web page in Rust

The easiest way to fetch a web page using Rust is with the help of the `hyper::client` module. You will still need to create a new project using Cargo; the **Cargo.toml** file will be the same as before. The contents of **./src/main.rs** file are:

```
extern crate hyper;
use hyper::Client;

fn main() {
   let client = Client::new();
   let res = client.get("http://www.linuxformat.com").send().
unwrap();
   assert_eq!(res.status, hyper::Ok);
        println!("Status: {}", res.status);
        println!("{}", res.headers);

}
```

The web page you want to visit is specified with the `get()` function. Executing the *http_client* project generates the next output, like this:

```
$ cargo run
    Running `target/debug/http_client`
Status: 200 OK
```

An FTP client in Rust

There's a Rust module, called ftp, that can help you create FTP clients. However, it looks like there's currently a problem with the module because it fails to communicate with any ftp server:

```
$ cargo run test.talia.net
...
Trying to connect to: test.talia.net
thread '<main>' panicked at 'Invalid response',
src/main.rs:22
```

The Rust code used for connecting to an FTP server is the following:

```
extern crate ftp;

use std::str;
use std::io::Cursor;
use ftp::FTPStream;

fn main() {
    let ftpserver = ::std::env::args().nth(1).
unwrap();
    println!("Trying to connect to: {}", ftpserver);
    let mut ftp_stream = match
FTPStream::connect(ftpserver.to_string(), 21) {
            Ok(s) => s,
            Err(e) => panic!("{}", e)
    };
    match ftp_stream.login("anonymous",
"test@gmail.com") {
            Ok(_) => (),
            Err(e) => panic!("{}", e)
    }
}
```

Nevertheless, the ftp module looks to be a very promising indeed; if you are interested in developing an FTP client in Rust you should keep an eye on it. You can find more information about its development at **https://github.com/mattnenterprise/rust-ftp**.

```
Expires: Sun, 19 Nov 1978 05:00:00 GMT
Date: Tue, 17 May 2016 06:21:51 GMT
Last-Modified: Tue, 17 May 2016 06:21:51 GMT
Cache-Control: store, no-cache, must-revalidatepost-
check=0, pre-check=0
Connection: close
Content-Type: text/html; charset=utf-8
Transfer-Encoding: chunked
Set-Cookie: SESSc1e62b721508fb7ffebc7f81393bd318=1b
9e47dcafadb8b943a9dc34cc4f07ef; path=/; domain=.
linuxformat.com
Server: Apache/2.2.8 (Ubuntu) PHP/5.2.4-2ubuntu5.26 with
Suhosin-Patch mod_ssl/2.2.8 OpenSSL/0.9.8g
X-Powered-By: PHP/5.2.4-2ubuntu5.26
X-Clacks-Overhead: GNU Terry Pratchett
Vary: Accept-Encoding
```

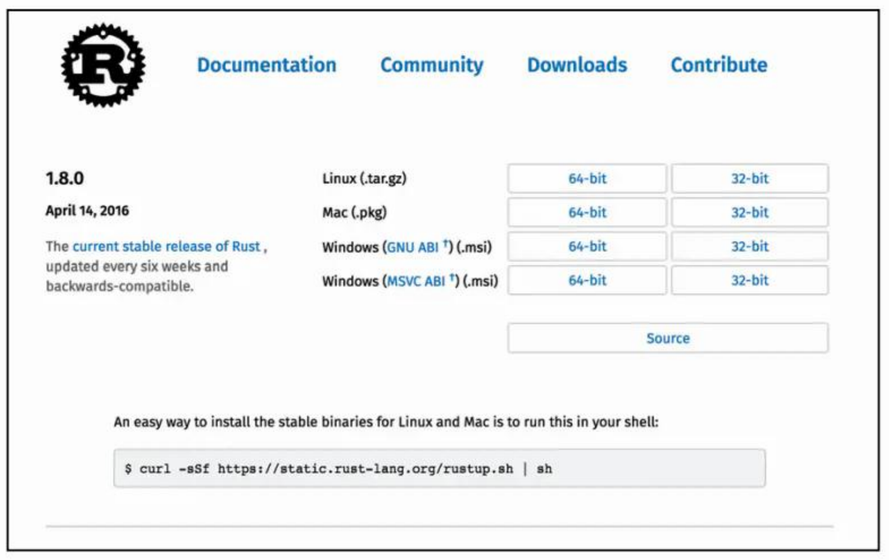
The Hyper package has many more capabilities than those we've covered. You can find more information about it at **https://github.com/hyperium/hyper**.

Beta and Nightly Rust versions

Apart from the current stable version, the team that develop Rust are always maintaining two additional versions: Beta and Nightly. The Beta version of Rust is a preview of the upcoming stable release. The Nightly version is the current development branch that contains many new features. The good thing is that most of the experimental features will make it to a stable Rust version. The bad thing is that the Nightly version will contain unstable features.

You can install the beta version of Rust as follows:

```
# curl -sSf https://static.rust-lang.org/rustup.sh | sh -s --
--channel=beta
rustup: gpg available. signatures will be verified
rustup: downloading manifest for 'beta'
rustup: downloading toolchain for 'beta'
########################################
########################################
100.0%
gpg: assuming signed data in `/root/.rustup/
dl/38ff98844877ad6e8d97/rust-beta-x86_64-unknown-
linux-gnu.tar.gz'
gpg: Signature made Thu 05 May 2016 08:55:35 AM EEST
using RSA key ID 7B3B09DC
gpg: Good signature from "Rust Language (Tag and Release
Signing Key) <rust-key@rust-lang.org>"
gpg: WARNING: This key is not certified with a trusted
signature!
```



This is the web page where you can find information about the current and the forthcoming versions of Rust

```
gpg:          There is no indication that the signature belongs
to the owner.
Primary key fingerprint: 108F 6620 5EAE B0AA A8DD
5E1C 85AB 96E6 FA1B E5FE
     Subkey fingerprint: C134 66B7 E169 A085 1886  3216
5CB4 A934 7B3B 09DC
rustup: installing toolchain for 'beta'
rustup: extracting installer
install: uninstalling component 'rustc'
install: uninstalling component 'rust-std-x86_64-unknown-
linux-gnu'
install: uninstalling component 'rust-docs'
install: uninstalling component 'cargo'
install: creating uninstall script at /usr/local/lib/rustlib/
uninstall.sh
install: installing component 'rustc'
install: installing component 'rust-std-x86_64-unknown-
linux-gnu'
install: installing component 'rust-docs'
install: installing component 'cargo'

  Rust is ready to roll.
# rustc --version
rustc 1.9.0-beta.2 (87cb73304 2016-05-04)
```

Similarly, you can install the Nightly build as follows:

```
# curl -sSf https://static.rust-lang.org/rustup.sh | sh -s --
--channel=nightly
...
# rustc --version
rustc 1.10.0-nightly (8492b6aa4 2016-05-14)
```

You can find more information about the Stable, Beta and Nightly Rust versions at **https://www.rust-lang.org/downloads.html** (which is also pictured above). In the next section we will be covering concurrent programming in Rust, so turn the page! ■

Rust: A guide to concurrency

It's time to explain the complex subject of concurrency, covering all that you need to know to start programming using threads in Rust

This final part will cover concurrent programming in Rust. Concurrency and threads are first class citizens in Rust and you should use concurrent techniques as a first resort when solving a problem. Bear in mind, however, that usually there's more than one way to solve a problem concurrently! So, when developing concurrent applications, think about their design and consider that the most important property of a concurrent application is making functions as autonomous and independent as possible so that they can be executed in parallel.

This tutorial uses a stable release of Rust, 1.9. You can check out the Rust website to find the latest stable version of the code, or a Nightly build, depending on your particular preferences. This tutorial will not use *Cargo* to create the presented programs because it's not necessary. As you already know, *Cargo* is more useful when you need to use external crates.

Concurrency is a complex subject that Rust tries to make easier than it usually is. The difficulty with concurrency is that 'A Shared Mutable State is the root of all problems' and to deal with that problem Rust takes a different approach: Instead of dealing with the 'Mutable' part, it deals with the 'shared' part. Put simply, the compiler will let you know about any errors in the 'shared' part, so if your Rust code compiles, you'll have no such errors and as a result, it isn't dangerous to have shared memory among threads in Rust!

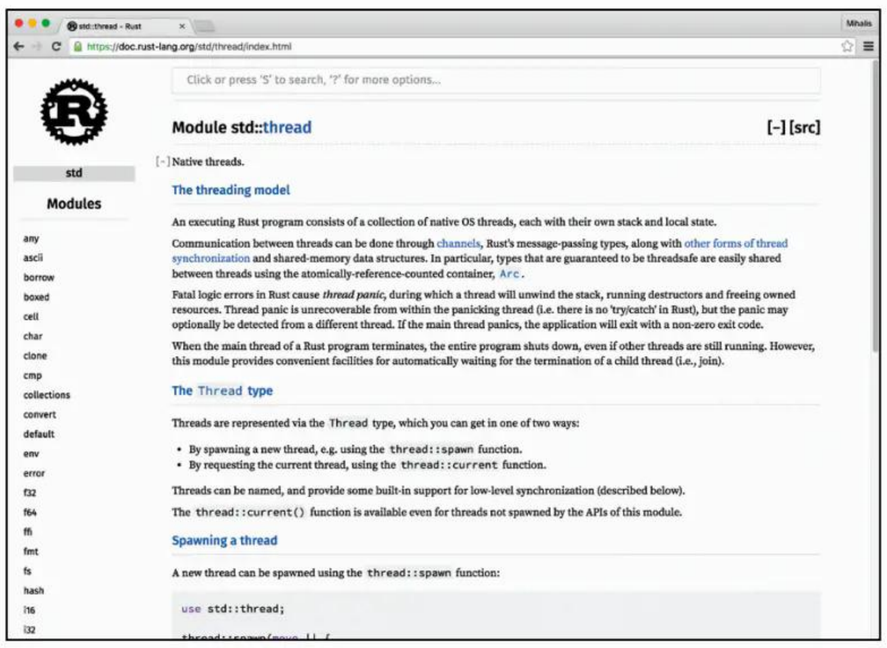
Concurrency in Rust

Look at the following Rust code:

```
&foo
&mut bar
```

The first line defines a reference to a variable named `foo` that's shared but immutable, whereas the second line defines a reference to a variable named bar that is mutable but not shared! Neither of these variables can generate a race condition when used in concurrent programming. The first variable because it's immutable, so it can only be read and the second variable because it's not shared. This approach is fundamentally different from the C approach. The question now is what happens when you need variables that are both shared and mutable? Keep reading to find the answer.

Rust starts a new thread using the **thread::spawn** which is part of the **std::thread** module. (The documentation page of the **std::thread** module is pictured, above.) The thread::spawn function returns a JoinHandle structure that's 'an owned permission to join on a thread'. In other words, JoinHandle provides a way to join the child thread. Most often, you will use the provided JoinHandle to call the **join()** method to make the **main()** function wait for the thread to finish.



› This is a small section of the extensive std::thread function man page. This is the standard Rust module that deals with thread manipulation

Instead of starting with something that works, it would be more interesting to begin with something that doesn't work and start appreciating the Rust compiler, which is aware of concurrency. The following code looks valid:

```
use std::thread;
fn main() {
    let my_array = [1, 2, 3, 4, 5];
    let join_handle = thread::spawn( || {
        println!("The array is {:?}", my_array);
    });
}
```

Unfortunately, trying to compile it will fail with the next error message:

```
error.rs:7:35: 9:3 error: closure may outlive the current function, but it borrows `my_array`, which is owned by the current function [E0373]
```

Compiler help

So Rust compiler tells you what the error is and it's that you are trying to borrow the `my_array` variable that is owned by another function. Unfortunately, this isn't allowed by the type system of Rust and the compiler is here to make sure that this will not happen!

The following code, saved as **error.rs**, will compile just fine:

```
use std::thread;
```

Quick tip

Concurrency is when multiple computations are being executed at the same time on a computer. Sometimes, they might need to interact with each other, which can be very tricky and needs special care.

Concurrency vs Parallelism

It is a common misconception that concurrency and parallelism are the same thing. As you will see, concurrency is actually better!

Parallelism is the simultaneous execution of multiple things whereas concurrency is a composition of independently executing things. In other words, concurrency is about structuring a thing in a way that it can be executed in parallel. So, the goal of concurrency is not parallelism but creating things with an independent and good structure. It is only when you manage to make two or more things that you build in a concurrent way to execute simultaneously that you have parallelism.

What is being executed is usually a function—this isn't compulsory but is very helpful because functions, when implemented correctly, allow for a somehow autonomous execution.

In concurrency, adding more concurrent entities actually makes things run faster and better—in other words, you improve the performance of a program by adding concurrent procedures to an existing design, so the parallelism comes from a better concurrent expression of the problem. It's the job of the developer to take concurrency into account when designing a solution to a problem in order to benefit from parallelism. Even if you cannot run your procedures in parallel, you'll still benefit from a valid concurrent design because a future system might be able to execute multiple things! Remember when CPUs had only one core?

So when doing concurrency, you should not think about parallelism but about breaking things into independent components that when composed solve the original problem.

```
fn main() {
   let my_array = [1, 2, 3, 4, 5];
   let join_handle = thread::spawn( move || {
      println!("The array is {:?}", my_array);
   });
   join_handle.join();
}
```

All the difference here is made by the move keyword that tells Rust that instead of just borrowing `my_array`, you want the closure – the function that's used as a free variable and is called by `thread::spawn` – to have complete access to it. As you will see in a while the `join()` call is really important despite the fact that it has nothing to do with the error message you saw earlier. Please note that `{:?}` prints the debug representation whereas `{}` prints the user representation of a variable when used in `println`. The general idea here is to try not to fight with the compiler; you will have much better results if you consider the Rust compiler as your friend!

The presented example will just display text coming from multiple threads on the screen. The Rust code of simple.rs is:

```
use std::thread;
use std::time::Duration;

fn main()
{
   for _ in 0..5 {
      thread::spawn( || {
         thread::sleep(Duration::from_millis(4000));
         println!("Hello from thread!");
      });
   }
}
```

A simple example

As you already know, you can create a new thread with the help of the std::thread module that contains the `thread::spawn` function. The reason for using the

Quick tip

Concurrent programming is important because it allows things to run in parallel in a way that increases the performance of a program. However, you must design your software with concurrency in mind – a very simple prerequisite is that functions must be autonomous and independent.

```
2. mtsouk@mail: ~/docs/article/working/rustConc.LXF214/code (ssh)
code$ rustc --version
rustc 1.9.0 (e4e8b6668 2016-05-18)
code$ rustc synch.rs
code$ ./synch
Hello from thread 6
Hello from thread 7
Hello from thread 8
Hello from thread 5
Hello from thread 9
Hello from thread 10
Hello from thread 4
Hello from thread 11
Hello from thread 12
Hello from thread 13
Hello from thread 14
Hello from thread 3
Hello from thread 2
Hello from thread 1
Hello from thread 0
code$ ./synch
Hello from thread 6
Hello from thread 7
Hello from thread 8
Hello from thread 5
Hello from thread 9
Hello from thread 10
Hello from thread 4
Hello from thread 11
Hello from thread 12
Hello from thread 13
Hello from thread 14
Hello from thread 3
Hello from thread 2
Hello from thread 1
Hello from thread 0
code$ ./synch
Hello from thread 6
Hello from thread 7
Hello from thread 5
Hello from thread 8
Hello from thread 9
Hello from thread 4
Hello from thread 10
Hello from thread 11
Hello from thread 12
Hello from thread 3
Hello from thread 13
Hello from thread 14
Hello from thread 2
Hello from thread 1
Hello from thread 0
code$
```

The output of synch.rs demonstrates that you cannot predict the order of execution of the threads

» `thread::sleep()` is for mimicking the processing time of a function because functions don't just print a message and exit. The delay introduced by `sleep()` in this case is four seconds. The for loop is used for creating five threads. The reason for using an underscore (_) instead of a real variable in the for loop is that you don't need to use any of the values in the 0..5 range. If you use a variable, Rust will let you know that it's not being used:

```
simple.rs:6:6: 6:7 warning: unused variable: `i`,
#[warn(unused_variables)] on by default
```

And the output of **simple.rs** is the following:

```
$ time ./simple
real    0m0.004s
user    0m0.000s
sys 0m0.000s
```

If you look closely at the Rust code, you should expect to get five messages on screen because five threads are created. However, **simple.rs** produces no output! As if this was not enough, **simple.rs** finishes in much less than four seconds. The root of the problem is simple: as soon as the main program exits all of its threads are automatically terminated so none of them has enough time to execute `println!()`. Therefore, the current implementation of **simple.rs** isn't that good. The next section will try to improve **simple.rs**.

Giving threads time to finish

The most difficult task when you are dealing with lots of threads is synchronising them and giving them time to finish their jobs. As you already know, the return value of `thread::spawn` is a JoinHandle structure that can be used for thread synchronisation. In this case, you'll use the JoinHandle structure to make the `main()` function synchronise with its threads—this will give the necessary time to the threads to print their messages and will let the main program know when all threads have finished their jobs. After that, the `main()` function can exit peacefully and with a clear conscience.

The following is the code of **synch.rs**, which is based on **simple.rs**:

```
use std::thread;
use std::time::Duration;

fn main()
{
    let mut my_threads = vec![];
    for i in 0..5 {
        let join_handle = thread::spawn( move || {
            thread::sleep(Duration::from_millis(4000));
            println!("Hello from thread {}", i);
        });
        my_threads.push(join_handle);
    }

    for join_handle in my_threads {
        let _ = join_handle.join();
    }
```

As you can see from the code, it was relatively easy to correct the problem. What you need to do is to collect the return values of each call to `thread::spawn`, store them into a vector, which is named `my_threads`, and use the `join()` function afterwards to give them time to do their jobs.

Executing **synch.rs** generates the expected output with a small surprise:

```
$ time ./synch
Hello from thread 3
Hello from thread 2
Hello from thread 0
Hello from thread 4
Hello from thread 1

real    0m4.005s
user    0m0.000s
sys     0m0.000s
```

The surprise here is the fact that threads don't get executed in the same order that they were created—this happens for a lot of reasons but it's mainly due to the way UNIX works so never make assumptions about the order of execution when dealing with threads! If you have more threads, the results are even more unpredictable (as you can see bottom of the previous page).

This section will present an example where all threads need to access the same variable, in this case the array you saw in **error.rs**; the rest of the code will be based on **synch.rs**.

The code of the new example, which is saved as **access.rs**, is the following:

```
use std::thread;
use std::sync::Arc;

static NTHREADS: i32 = 5;

fn main() {
        let my_array = Arc::new([-1, -2, -3, -4, -5]);

        let mut my_threads = vec![];

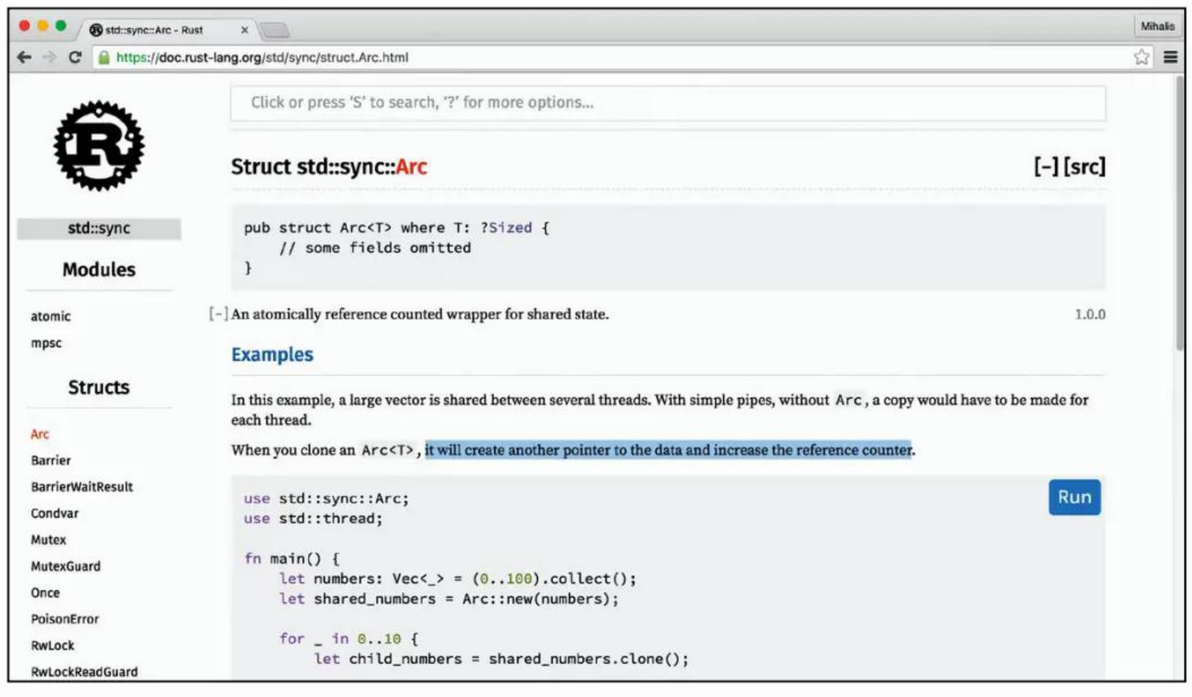
        for i in 0..NTHREADS as usize {
                let ar = my_array.clone();
                let join_handle = thread::spawn( move ||
{
                        println!("The {} element of the
array is {}", i, ar[i]);
                });
                my_threads.push(join_handle);
        }

        for join_handle in my_threads {
                let _ = join_handle.join();
```

Quick tip

A process is an execution environment with instructions, user-data and system-data parts, and other kinds of resources that are obtained during runtime. A quick and naïve way to differentiate a thread from a process is to consider a process the binary file and a thread a subset of a process.

› This is the documentation page of std::sync::Arc. This will automatically deal with the number of references of a variable



Unsafe blocks of code

If you use a mutable static global variable then you will need to put it in an unsafe block of code. You mainly use unsafe blocks of code for low level operations that you don't want to be handled by the strict type system of Rust. The following code, saved as **unsafe.rs**, illustrates the concept of the unsafe block:

```
static mut MONEY: i32 = 500;

fn main() {
    unsafe {
        MONEY += 10;
        println!("MONEY: {}", MONEY);
    }
}
```

Here, the static global mutable variable is `MONEY`. Being mutable means than its value can change. If you're using multiple threads, then one thread can read this variable while another one is changing its value which may result in unsafe memory—for this reason you have to use an unsafe block.

The good thing is that very little code needs to be unsafe.

Arc stands for Atomic Reference Counting and is an atomically reference counted wrapper for shared state. Put simply, when you use the `clone()` call for an `Arc<T>`, it will create another pointer to the data and increase the reference counter. This will happen each time the for loop gets executed. As you have many pointers pointing to a variable, multiple threads can read the single variable. You can learn more about `std::sync::Arc` at its documentation page (*see left*). Executing **access.rs** produces the following output:

```
$ ./access
The 0 element of the array is -1
The 1 element of the array is -2
The 4 element of the array is -5
The 2 element of the array is -3
The 3 element of the array is -4
```

However, you are still unable to change any of the values in `my_array`, even if you make it mutable because of the Rust type system:

```
access.rs:14:4: 14:21 error: cannot assign to immutable indexed content
access.rs: 14                                   ar[i] = ar[i] + 1;
```

About sharing

In this section will show you an easy way to solve this problem. We'll illustrate a technique that allows multiple threads to write to a shared variable using a Mutex. This is a mutually exclusive flag that allows a single thread to do something critical. In this case, it changes the value of a shared variable while the other threads are waiting for the same mutex to become available in order to get it and do their jobs. You can achieve mutation using either `RWLock<T>` or `Mutex<T>`. The former allows multiple readers and only one writer, whereas the latter allows only one reference to a shared resource. In both cases, only one thread can modify the shared resource.

The code of **share.rs**, which uses `Mutex<T>`, is:

```
use std::thread;
use std::sync::Arc;
use std::sync::Mutex;

static NTHREADS: i32 = 5;

fn main() {
        let my_array = Arc::new(Mutex::new([-1, -2, -3, -4, -5]));

        let mut my_threads = vec![];

        for i in 0..NTHREADS as usize {
                let mutex = my_array.clone();
                let join_handle = thread::spawn( move || {
                        let mut ar = mutex.lock().unwrap();
                        ar[i] = ar[i] + 2;
                        println!("The {} element of the array is {}", i, ar[i]);
                });
                my_threads.push(join_handle);
        }

        for join_handle in my_threads {
                let _ = join_handle.join();
        }
}
```

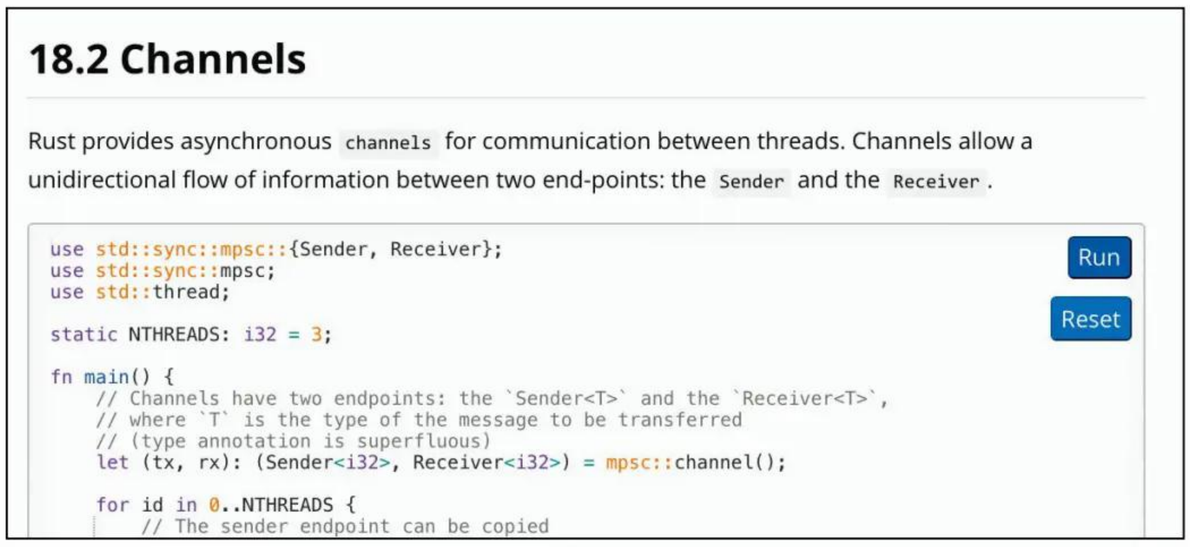
Most of the code is the same as before. The main difference is that you need to define `my_array` differently in order to make the Mutex available for use and that you introduce the `ar` variable inside the closure instead of outside. Additionally, the `ar` variable is defined using `let mut` which makes it mutable. The output of **share.rs** is:

```
$ ./share
The 0 element of the array is 1
The 1 element of the array is 0
The 3 element of the array is -2
The 2 element of the array is -1
The 4 element of the array is -3
```

As you can see, the value of each array item is increased by two using a single thread for processing each array item.

About channels

Finally we'll briefly mention another technique that allows you to create threads that communicate with each other. The new concept is called a Channel and is used by Rust to provide asynchronous communication between threads. (Above, shows a simple example as well as its output as found in **http://rustbyexample.com**.) You can find the Rust code of the example as **communicate.rs**. This is the last tutorial in the Rust series. The best thing you can do now is start writing real applications in Rust using what you have learnt from the tutorials. Good luck and have fun with Rust! ■



This illustrates the use of channels in Rust as presented in http://rustbyexample.com. Channels allow threads to talk to each other

Go: Master Google's new programming language

Get yourself familiar with the fundamentals of using Go

Go is an open source systems programming language that makes it easy to build reliable and efficient software. On a Debian Linux system you can install Go as follows:

```
# apt-get install golang
```

If you execute apt-cache search golang you will find that there exist a plethora of packages related to Go that you can also install. Find out the version of Go you are using with:

```
$ go version
go version go1.3.3 linux/amd64
```

As the latest stable Go version is currently 1.6, if you need to install a newer Go version on your Linux system, visit **https://golang.org/dl**. The Go programs we'll take you through here will be relatively easy, but upcoming Go tutorials will include more advanced code so stay tuned!

All About Go

Go tries to bring joy and fun back to programming and can be used for developing web applications and system software. Go supports procedural, concurrent and distributed programming as well as garbage collection. Despite the fact that Go has no direct support for object-oriented programming, Composition can help you mimic inheritance. The Go syntax will look familiar to C programmers, which is not a bad thing. However, Go does not have a preprocessor, as does C. Go tries to make its code clean, simple and therefore readable, in order to help the programmer work better and more efficiently, and keeps its concepts orthogonal, which means that concepts do not overlap.

The main advantage of Go is that its code is portable to many operating systems. The other advantage of Go is that it has a very active community with various conferences where you can directly talk to the developers of Go! The Go version of the "Hello World!" program is the following:

```
package main
import "fmt"
func main() {
  fmt.Println("Hello, world!")
}
```

Each file that contains Go code begins with a package declaration followed by the necessary import declarations. The package declaration shows the package that this file belongs to. Please note the absence of semicolons to declare the end of a Go statement. In order to run the hello.go file, you should execute the following command:

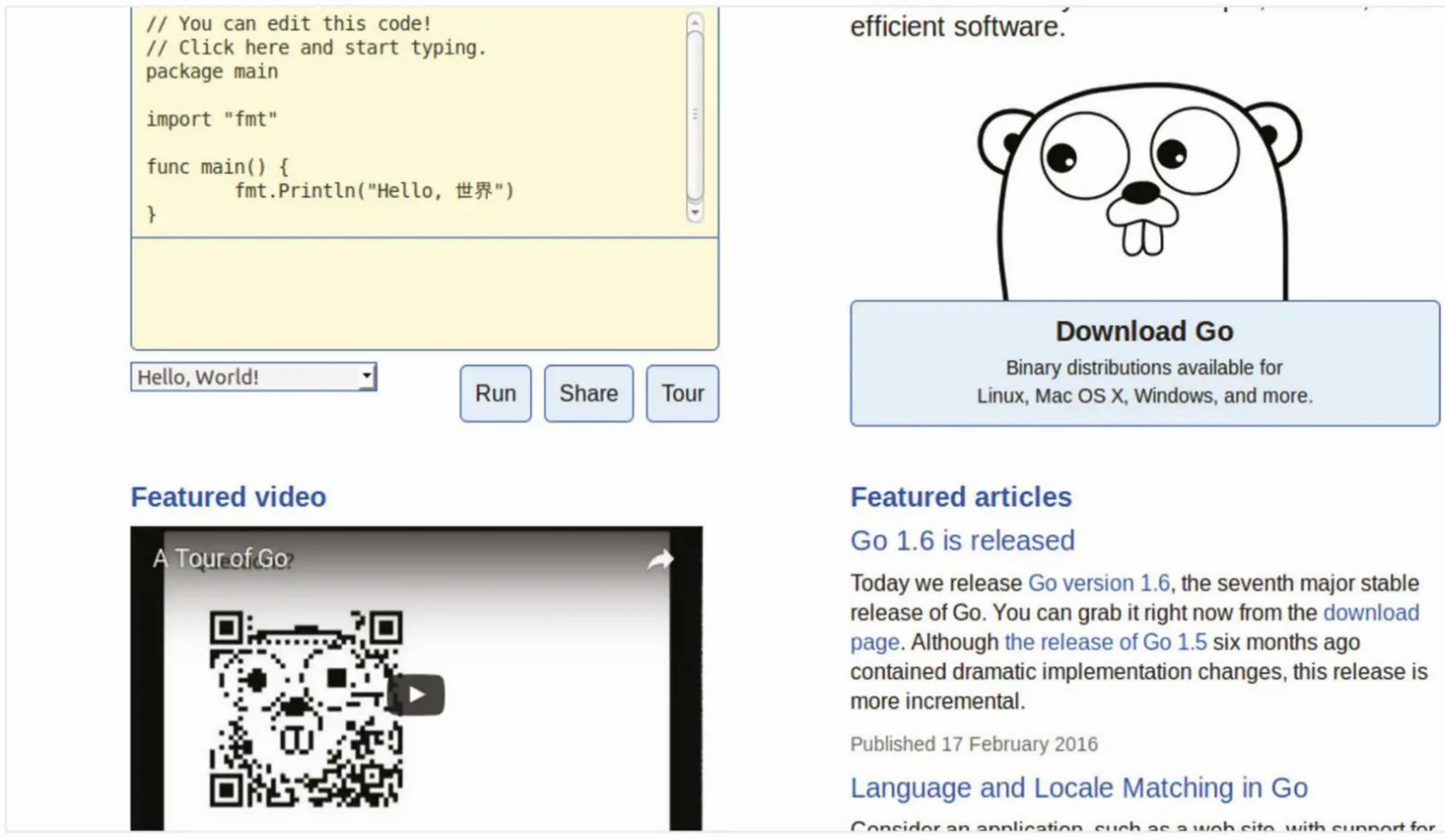
```
$ go run hello.go
Hello, world!
```

As Go is strict about how you format its code, putting the opening { on a separate line is going to generate the next error message:

```
# command-line-arguments
./hello.go:4: syntax error: unexpected semicolon
or newline before {
```

The official reason for this is that Go requires the use of semicolons as statement terminators in many contexts, and the compiler is able to automatically insert the required semicolons at the end of nonblank lines. Putting the openingbrace (it looks like this: {) on its own line will make Go

Go: Master Google's new programming language



› **Go is designed to be a readable language, making it very approachable for first-time coders to work with**

put a semicolon at the end of the previous line, which is what generates the error message. The good thing is that you no longer need to worry about brace placement as in other programming languages.

Figure 1, as you can see to the right, shows the output you get when you execute the go command line utility without any other parameter. If you import a package but you do not use any of its components, Go will complain and deny compiling or running the code! This happens for a reason: Go tries to reduce accidental programming mistakes so it forces you to be sure about what you are doing; as a result, Go can detect silly errors and make you correct them:

```
$ go run hello.go
# command-line-arguments
./hello.go:4: imported and not used: "flag"
```

If you want to generate an executable file from hello.go, you can use the build command:

```
$ go build hello.go
$ ls -l hello
 -rwxr-xr-x 1 mtsouk mtsouk 1823728 Mar 24 19:26
hello
```

The executable file, which is automatically named after the name of the source code file, is rather big because it is

```
mtsouk@mail:~$ go
Go is a tool for managing Go source code.

Usage:

        go command [arguments]

The commands are:

    build       compile packages and dependencies
    clean       remove object files
    env         print Go environment information
    fix         run go tool fix on packages
    fmt         run gofmt on package sources
    get         download and install packages and dependencies
    install     compile and install packages and dependencies
    list        list packages
    run         compile and run Go program
    test        test packages
    tool        run specified go tool
    version     print Go version
    vet         run go tool vet on packages

Use "go help [command]" for more information about a command.

Additional help topics:

    c           calling between Go and C
    filetype    file types
    gopath      GOPATH environment variable
    importpath  import path syntax
    packages    description of package lists
    testflag    description of testing flags
    testfunc    description of testing functions
```

Figure 1

The history of Go

Go started as an internal Google project back in 2007 to help Google handle big programs developed by big teams. The Go project started by Robert Griesemer, Rob Pike and Ken Thompson with systems programming in mind.

On November 2009, Go went publicly available under a liberal open source license. Its latest stable release (1.8) was announced in February 2017. You can find more about it at **https://golang.org/** and at **https://gophercon.com/** Incidentally, as Go is a common and generic word, Go is also called Golang, which is pretty handy when you want to Google something related to the Go programming language.

Figure 2

```
1  package main
2
3  import "fmt"
4
5  func myFibo(x int) int {
6
7      if x == 0 || x == 1 {
8          return x
9      }
10     return myFibo(x-1) + myFibo(x-2)
11 }
12
13 func main() {
14
15     for i := 20; i < 30; i++ {
16         fmt.Println(i, "->", myFibo(i))
17     }
18 }
19
20 $ go run simple.go
21 20 -> 6765
22 21 -> 10946
23 22 -> 17711
24 23 -> 28657
25 24 -> 46368
26 25 -> 75025
27 26 -> 121393
28 27 -> 196418
29 28 -> 317811
30 29 -> 514229
```

» statically built, which means that it is an autonomousprogram that requires no external libraries to run on another Linux system, which is also confirmed by the output of the file(1) command line utility:

```
$ file hello
hello: ELF 64-bit LSB executable, x86-64, version 1
(SYSV), statically linked, not stripped
```

Switch, If and For

The switch statement is very powerful in Go, as in most modern programming languages, and can save you from long chained if statements while making your code more readable. A Go switch statement does not need an operand – you can just list the cases – and does not require break statements to end a case block as in C.

However, you should have a default branch when the case expressions do not cover every possible value. Please note that if you use a switch expression, the type of the switch expression and the types of all case expressions should match; if they do not match you will get a compiler error as in the following example:

```
var x = 1
switch x {
case true:
 fmt.Println("True")
case false:
fmt.Println("False")
default:
 fmt.Println("Neither true nor false!")
 }
```

The compiler errors would be the following:

```
# command-line-arguments
./a.go:11: invalid case true in switch on x
(mismatched types bool and int)
 ./a.go:13: invalid case false in switch on x
(mismatched types bool and int) }
```

In other words, if the type of the variable of the switch statement is an integer, then the type of the expression of the cases cannot be a Boolean. The following switch statement, which is a tagless switch, determines the sign of an integer:

```
var x = 1
switch {
case x < 0:
 .fmt.Println("Negative integer")
case x > 0:
 .fmt.Println("Positive integer")
default:
fmt.Println("0!")
}
```

Please note that if the first line of the switch block is "switch x", you will get an error message during compilation as in the earlier example. An alternative way of expressing it is the following, which is syntactically correct because the evaluated expression can be either true or false:

```
var y = -10
switch y < 0 {
 case true:
   fmt.Println("Negative integer")
case false:
   fmt.Println("Positive integer")
 }
```

Also note that a case statement can have more than one value as in the next example:

```
 switch {
 case 1, 3, 5, 7, 9:
  fmt.Println("Odd!")
 case 0, 2, 4, 6, 8:
  fmt.Println("Even!")
 default:
  fmt.Println("What?")
 }
```

A switch statement can be used for determining the type of a variable, or even a function, as in the following code:

```
 var t interface{}
 t = "123"
 switch t := t.(type) {
 case bool:
  fmt.Println("Boolean:", t)
 case int:
  fmt.Println("Integer:", t)
 case float64:
  fmt.Println("Float:", t)
```

Is Go good for system admin tasks?

Go has many advantages that make it a rational candidate for writing scripts that perform system administration tasks. First, you should bear in mind that Go is used in Google for system administration tasks; however, you should also recognise that you should not try to replace one-line Perl and Python scripts with equivalent Go scripts as this would be inefficient. The advantage of Go for system administration tasks is that its code is easy to read; this also means that you can easily make changes to existing Go programs.

As Go uses static linking, the generated binaries can easily get transferred to other Linux machines without worrying about libraries, dependencies and different library versions. Furthermore, the standard Go distribution offers many packages that can help a system administrator. Finally, Go can easily create handy web interfaces that can be the frontend to Go scripts.

Figure 3

```
1  package main
2
3  import (
4      "fmt"
5      "strings"
6  )
7
8  func main() {
9      // Define a string
10     var aString string = "Linux User and Developer"
11     // Declare a constant string
12     const anotherString = "Mihalis Tsoukalos"
13
14     for i := 0; i < len(aString); i++ {
15         fmt.Printf("%x ", aString[i])
16     }
17     fmt.Print("\n")
18
19     // Use Binary Output
20     for i := 0; i < len(aString); i++ {
21         fmt.Printf("%b ", aString[i])
22     }
23     fmt.Print("\n")
24
25     // Use the strings package
26     fmt.Println(strings.Contains(aString, "Linux"))
27     fmt.Println(strings.Contains(aString, "linux"))
28
29     fmt.Println(strings.Count(anotherString, "a"))
30     fmt.Println(strings.Count(anotherString, "w"))
31
32     // A String array
33     var aStringArray = []string{"Linux", "User", "and Developer"}
34     fmt.Println(strings.Join(aStringArray, "_"))
35 }
36
37
38 $ go run myStrings.go
39 4c 69 6e 75 78 20 55 73 65 72 20 61 6e 64 20 44 65 76 65 6c 6f 70 65 72
40 1001100 1101001 1101110 1110101 1111000 100000 1010101 1110011 1100101 1110010 100000
41 1101110 1100100 100000 1000100 1100101 1110110 1100101 1101100 1101111 1110000 110010
42 true
43 false
44 2
45 0
46 Linux_User_and Developer
```

```
default:
   fmt.Println("Unknown type!")
 }
```

Forget about the **interface{}** keyword for now as it will be explained in a forthcoming tutorial. The output from the previous switch statement for various types of input will be as follows:

```
 Input: t = "123"
 Output: Unknown type! (because "123" is a string
which is handled by default case)
 Input: t = 123
 Output: Integer
 Input: 123.2
 Output: Float
```

Using the Go switch statement effectively will make you a better Go programmer! The if statement works as you would expect but without any parentheses to surround the conditions:

```
 if x <= 20 {
   fmt.Println("X is bigger than 20")
 } else {
   fmt.Println("X is smaller than or equal to 20")
 }
```

What you should remember is that the variables you declare – using short variable declarations – in the condition of an if statement are also visible inside any of the else blocks but not after the end of the if statement (this is not illustrated here).

The for loop in Go looks similar to the C version but without the parentheses. The following code defines a simple for loop:

```
 for i := 20; i < 30; i++ {
   // Do something
 }
```

However, apart from the normal syntax of the for loop, a for loop can also be used for iterating through a string as follows:

```
 var aString = "Linux User & Developer"
```

Figure 4

```
func Compare(a, b string) int
func Contains(s, substr string) bool
func ContainsAny(s, chars string) bool
func ContainsRune(s string, r rune) bool
func Count(s, sep string) int
func EqualFold(s, t string) bool
func Fields(s string) []string
func FieldsFunc(s string, f func(rune) bool) []string
func HasPrefix(s, prefix string) bool
func HasSuffix(s, suffix string) bool
func Index(s, sep string) int
func IndexAny(s, chars string) int
func IndexByte(s string, c byte) int
func IndexFunc(s string, f func(rune) bool) int
func IndexRune(s string, r rune) int
func Join(a []string, sep string) string
func LastIndex(s, sep string) int
func LastIndexAny(s, chars string) int
func LastIndexByte(s string, c byte) int
func LastIndexFunc(s string, f func(rune) bool) int
func Map(mapping func(rune) rune, s string) string
func Repeat(s string, count int) string
func Replace(s, old, new string, n int) string
func Split(s, sep string) []string
func SplitAfter(s, sep string) []string
func SplitAfterN(s, sep string, n int) []string
func SplitN(s, sep string, n int) []string
func Title(s string) string
func ToLower(s string) string
func ToLowerSpecial(_case unicode.SpecialCase, s string) string
func ToTitle(s string) string
func ToTitleSpecial(_case unicode.SpecialCase, s string) string
func ToUpper(s string) string
func ToUpperSpecial(_case unicode.SpecialCase, s string) string
func Trim(s string, cutset string) string
func TrimFunc(s string, f func(rune) bool) string
func TrimLeft(s string, cutset string) string
func TrimLeftFunc(s string, f func(rune) bool) string
func TrimPrefix(s, prefix string) string
func TrimRight(s string, cutset string) string
func TrimRightFunc(s string, f func(rune) bool) string
func TrimSpace(s string) string
func TrimSuffix(s, suffix string) string
type Reader
   func NewReader(s string) *Reader
   func (r *Reader) Len() int
   func (r *Reader) Read(b []byte) (n int, err error)
   func (r *Reader) ReadAt(b []byte, off int64) (n int, err error)
   func (r *Reader) ReadByte() (b byte, err error)
   func (r *Reader) ReadRune() (ch rune, size int, err error)
   func (r *Reader) Seek(offset int64, whence int) (int64, error)
   func (r *Reader) Size() int64
   func (r *Reader) UnreadByte() error
   func (r *Reader) UnreadRune() error
   func (r *Reader) WriteTo(w io.Writer) (n int64, err error)
type Replacer
   func NewReplacer(oldnew ...string) *Replacer
   func (r *Replacer) Replace(s string) string
   func (r *Replacer) WriteString(w io.Writer, s string) (n int, err error)
Bugs
```

```
for _, r := range aString {
   fmt.Println(string(r))
 }
```

A simple example

You already know how to run and create an executable file from Go code so it is time to write something useful in Go. The simple.go program illustrates how to calculate Fibonacci numbers in Go to help you get a better sense of Go code. The **myFibo()** function takes an integer an its input and returns aanother integer as its output. Then **myFibo()** is called from the **main()** function using a for loop. The definition of **myFibo()** function is the following: »

Do more

Figure 5

```
1  package main
2
3  import "fmt"
4
5  func main() {
6
7      // Read a number
8      fmt.Print("Give me a number: ")
9      var aNumber int
10     fmt.Scanln(&aNumber)
11     fmt.Println("Your number is:", aNumber)
12
13     // Read a string
14     fmt.Print("Enter your Name: ")
15     var name string
16     fmt.Scanln(&name)
17     fmt.Println("Your name is:", name)
18
19     // Print its type
20     fmt.Printf("The type of variable name is %T\n", name)
21
22     // Define the decimal points of the output
23     fmt.Printf("%.4f\n", 123.1231231)
24 }
```

```
func myFibo(x int) int {
  if x == 0 || x == 1 {
    return x
  }
  return myFibo(x-1) + myFibo(x-2)
}
```

You can also see an if statement with two conditions separated by **||**. Should you wish to import two or more packages on your Go code, you should use the following syntax instead of writing multiple import statements:

```
import (
  “flag”
  “fmt”
)
```

The aforementioned code imports two Go packages, fmt and flag, but you can use the same pattern to add even more Go packages. The source code of simple.go can be seen in Figure 2.

Basic data types

Go supports various types of integers and floating point numbers that differ in their size. Go also supports complex numbers natively. So, Go offers support four different sizes of signed and unsigned integers, named **int8**, **int16**, **int32**, **int64**, **uint8**, **uint16**, **uint32** and **uint64**, respectively. Additionally, there exist int and uint that are the most efficient signed and unsigned integers for your current platform. When in doubt, use int and uint. Go supports only two types of floating point numbers, **float32** and **float64**. The first one provides about six decimal digits of precision, whereas the second one gives you fifteen digits of precision.

Last, Go offers two types of complex numbers: **complex64** and **complex128**. The first one uses **float32** and the second one **float64** for the real and the imaginary parts of a complex number. As you saw, Go also supports the bool type as well as strings.

Figure 6

```
2. mtsouk@mail: ~ (bash)
iMac:code mtsouk$ gofmt
package main

import "fmt"

func main() {

// Read a number
fmt.Print("Give me a number: ")
var aNumber int
fmt.Scanln(&aNumber)
fmt.Println("Your number is:", aNumber)

// Read a string
fmt.Print("Enter your Name: ")
var name string
fmt.Scanln(&name)
fmt.Println("Your name is:", name)

// Print its type
fmt.Printf("The type of variable name is %T\n", name)

// Define the decimal points of the output
fmt.Printf("%.4f\n", 123.1231231)
}
package main

import "fmt"

func main() {

        // Read a number
        fmt.Print("Give me a number: ")
        var aNumber int
        fmt.Scanln(&aNumber)
        fmt.Println("Your number is:", aNumber)

        // Read a string
        fmt.Print("Enter your Name: ")
        var name string
        fmt.Scanln(&name)
        fmt.Println("Your name is:", name)
```

About strings

You can define a string in Go as follows:

```
var aString string = “Linux User and Developer”
```

You can define an array of strings with three elements as follows:

```
var aStringArray = []string{“Linux”, “User”, “and
Developer”}
```

As you already know, the **fmt.Println()** function is used for printing strings. The **myStrings.go** file shows various ways of dealing with strings in Go. You might the output of the following strange; this is mainly because Go uses Unicode characters everywhere:

```
for i := 0; i < len(aString); i++ {
  fmt.Printf(“%x “, aString[i])
}
```

The for loop uses the **len()** function, which returns the length of a string, accesses the string as an array and prints every individual byte of each character of the string.

The strange output is because each Unicode character needs more than one byte for its representation. The **%x** verb tells Go to print integers using base 16. Should you wish to get the output using binary numbers, you should replace **%x** with **%b**. Figure 3, as you can see on page 129, shows the Go code and the output of **myString.go**, which briefly showcases string manipulation in Go.

The string package provides many functions that can help you deal with strings. Figure 4 shows the available functions of the strings package as found in **https://golang.org/pkg/strings**.

Standard Go Packages

Go comes with a rich set of packages that implement many useful things. Apart from the fmt and io packages that you briefly saw in action, the standard Go library includes math, which provides mathematical functions and constants, net that is a powerful package that supports portable TCP and UDP connections, http which offers HTTP server and client implementations, url that allows you to easily parse URLs, the os package that is a portable interface to system operations and syscall that is an interface to low level system calls. Additionally, there is time, which supports time and date operations as well as crypto for cryptographic operations and compress that implements common compression algorithms. For the full list of packages please visit **https://golang.org/pkg/**. Go tries to make your life simpler by implementing many things for you – the main advantage of a Go package is that its code has been already tested for you.

Go: Master Google's new programming language

Figure 7

```
package main

import (
    "fmt"
    "io/ioutil"
    "os"
)

func main() {
    if len(os.Args) != 3 {
        fmt.Println("Please use two command line arguments!")
        os.Exit(-1)
    }

    in := os.Args[1]
    out := os.Args[2]

    input, err := ioutil.ReadFile(in)
    if err != nil {
        fmt.Println("Error reading the input!")
    }

    err = ioutil.WriteFile(out, input, 0644)
    if err != nil {
        fmt.Println("Error creating the destination file!")
    }
```

Figure 8

```
package main

import "fmt"

func main() {

    // Read a number
    fmt.Print("Give me a number: ")
    var aNumber int
    fmt.Scanln(&aNumber)
    fmt.Println("Your number is:", aNumber)

    // Read a string
    fmt.Print("Enter your Name: ")
    var name string
    fmt.Scanln(&name)
    fmt.Println("Your name is:", name)

    // Print its type
    fmt.Printf("The type of variable name is %T\n", name)

    // Define the decimal points of the output
    fmt.Printf("%.4f\n", 123.1231)
}
```

The standard fmt package

You will now learn how to do some formatting using the fmt package. The fmt package offers similar functionality to the **printf()** and **scanf()** **C** functions. Figure 5 shows the contents of the **usingFMT.go** file. Executing it produces the following output:

```
$ go run usingFMT.go
Give me a number: 12
Your number is: 12
Enter your Name: Mihalis
Your name is: Mihalis
The type of variable name is string
123.1231
```

The **fmt.Print** function is analogous to fmt.Println without automatically printing a newline character. The **fmt.Scanln()** function helps you get user input. Go calls **%T** used in **fmt.Printf()** a verb. The **%.4f** verb tells Go to print only four decimal points from a floating point number.

Go offers a very handy documentation system, supported by the **godoc** command line utility that can help you find information about provided functions. In order to learn more about **fmt.Println()**, you should type the following command:

```
$ godoc fmt Println
```

To briefly touch on the goftm utility, it comes with the Go distribution and formats Go programs. If you provide it with a filename, it works with the file; if you give it a directory to process, it will operate on all .go files in that directory and all of its subdirectories, recursively. Figure 6 shows a simple use of gofmt that gets its input from the standard input.

File I/O in Go

This section will present a simple Go example that performs file copying. You can see the full source code of **cp.go** in Figure 7, above and to the left, which is pretty compact, considering that it can copy any type of file. There exist many ways to implement file copying in Go; the method that we have presented to you here reads the entire input file and copies it all at once to the output file. The following code, found in cp.go, shows the error handling approach of Go:

```
if err != nil {
  fmt.Println("Error reading the input!")
}
```

Even EOF (end of file), which is defined in the io package, is an error which can be handled as follows:

```
if error == io.EOF {
  // Do something
}
```

It is the job of the programmer to examine the return value of a function and determine whether there was an error or not. Another interesting thing in **cp.go** is the use of **:=** for defining the filenames for the input and destination files. This is part of the short variable declaration, which is a handy way of declaring and initialising local variables.

The reason for calling it 'short' is because you do not need to declare the type of the variable as it is being determined by the type of the variable on the right side of the expression. A forthcoming tutorial will talk solely about File I/O in Go, as there is much to discuss in this subject, so keep an eye out for that in the coming issues.

Regular expressions and pattern matching

The regexp package helps you with regular expressions. The **regExp.go** file contains various examples – you can see the source of **regExp.go** in Figure 8 above. The **regexp.Compile()** function reads a regular expression. If the parsing of the regular expressing is successful, it returns a Regexp object that you can use to match against text as you can see in **regExp.go.**

We will also be taking you through more key information about regular expressions in Go in the next couple of tutorials as they play a key role in modern programming. ■

Declaration Types

Go supports four types of declarations: var, const, type and func. As you already know, the func keyword is used for defining a function. The const keyword is used for defining constants whereas the var keyword is used for creating variables of a specific type attached to a specific name following by its initial value. You can omit the type or the initial value from a var declaration but not both. The purpose of the type keyword is defining new variable types based on existing ones. The following statement defines a new variable type named natural, which is based on the existing int64 type:

```
type natural int64
```

You usually put type statements after the import section. The following statement defines a variable of type natural and a variable of type **float64**:

```
var anInt natural = 100
var myBalance float64 = 100.32
```

The following statement defines a constant named PI using two ways:

```
const PI = 3.14159
const PI float64 = 3.14159
```

Although both statements are equivalent, it is considered a better practice to specify the type of a variable.

Go: Learn data types

Get to grips with the composite data types of Go

Go is an open source systems programming language that makes it easy to build reliable and efficient software. Part of its greatness comes from the fact that Go supports many data types and allows you to build new ones easily. The data types that do not belong to the basic data types are called composite data types.

Combining basic data types together generates composite data types. A composite data type can also contain another composite data type. Although composite data types look pretty simple, it is important to master them before continuing with more advanced Go topics because at the end of the day, every program has to deal with variables, data and data structures – all three depend on data types!

This tutorial will cover arrays, slices, maps, pointers and structures. It will also illustrate how to convert an array into a map. Go also supports channel variables; a channel is a special kind of data type used in concurrent programming. You will learn more about channel variables when we talk about concurrent programming in Go.

Arrays

Arrays are a very popular data type due to their simplicity and speed. Almost all programs use arrays for storing data of the same type. An array can be defined as follows:

```
myArray := [4]int{1, 2, 4, 4}
```

The previous notation also declares the number of elements in the array. The indices of **myArray** go from 0 up to 3. Therefore, the first element is **myArray[0]** and the last one is **myArray[3]**. You can also define an array without specifying the number of its elements:

```
otherArray := [...]int{2, 4, 5, 6, 7, 8}
```

You can find the length of an array using the len() function. This is pretty useful when you want to access the elements of an array using a for loop:

```
length := len(myArray)
for i := 0; i < length; i++ {
  fmt.Printf("%d ", myArray[i])
}
```

There is another way to access all elements, and that is by using iteration:

```
for _, number := range otherArray {
  fmt.Printf("%d ", number)
}
```

The convenient thing with iteration is that you do not need to calculate the length of the array because the compiler does the work for you.An array with two dimensions can be defined as follows:

```
twoD := [3][3]int{{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}
```

Last, an array with three dimensions is defined as follows:

```
threeD := [2][2][2]int{{{1, 2}, {3, 4}}, {{5, 6}, {7, 8}}}
```

You can access individual array elements from **twoD** and **threeD** as usual:

```
twoD[1][2] = 13
threeD[0][1][1] = 14
```

Iterating over an array with two dimensions is tricky because you have to take into account its dimensions:

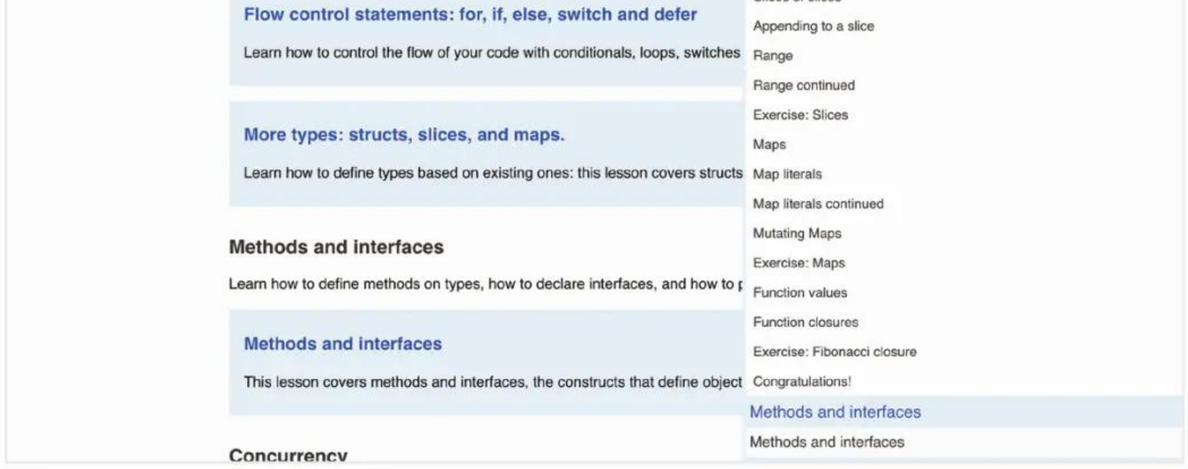
```
for _, number := range twoD {
  for _, other := range number {
    fmt.Printf("%d ", other)
  }
}
```

Figure 1 shows the source code of **array.go** that includes Go code related to arrays and produces the following output:

```
$ go run array.go.
1 2 4 4
2 4 5 6 7 8
1 2 3 4 5 13 7 8 9
```

```
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func main() {
8
9      // An array
10     myArray := [4]int{1, 2, 4, 4}
11
12     length := len(myArray)
13     for i := 0; i < length; i++ {
14         fmt.Printf("%d ", myArray[i])
15     }
16     fmt.Printf("\n")
17
18     // Another array
19     otherArray := [...]int{2, 4, 5, 6, 7, 8}
20     for _, number := range otherArray {
21         fmt.Printf("%d ", number)
22     }
23     fmt.Printf("\n")
24
25     // Printing a two dimensional array
26     twoD := [3][3]int{{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}
27     twoD[1][2] = 13
28
29     for _, number := range twoD {
30         for _, other := range number {
31             fmt.Printf("%d ", other)
32         }
33     }
34     fmt.Printf("\n")
35
36     // Printing a three dimensional array
37     threeD := [2][2][2]int{{{1, 2}, {3, 4}}, {{5, 6}, {7, 8}}}
38     threeD[0][1][1] = 14
39
40     for _, number := range threeD {
41         fmt.Printf("%d ", number)
```

Figure 1




```
[[1 2] [3 14]] [[5 6] [7 8]]
```

Please bear in mind that arrays in Go have a fixed size that cannot be changed afterwards. So, if you want to include an additional element to an existing array with no empty space, you will have to create a bigger array and copy all elements from the old array to the new one.

Also, it is useful to know that when you pass an array to a function, you actually pass a copy of the array; therefore, any changes you make to an array inside a function will be lost when the function finishes. Additionally, passing a large array to a function can be pretty slow. The solution to all these problems is to use slices instead of arrays.

Network sockets

Although arrays are very useful, they can only hold variables of the same type. When you need to group various types of variables, you can use a structure. The various elements of a structure are called fields. A structure with three fields can be defined as follows:

```
type point struct {
  x int
  y int
  label string
}
```

However, as you are actually declaring a new type, you will have to create a new variable after the definition of the structure, which can happen in two ways:

```
p1 := point{23, 12, "A Point"}
p2 := point{}
```

The first method not only creates a new point variable but it also gives values to the three fields of the structure.

On the other hand, p2 is an empty point variable. You can access individual fields of a structure using the next method:

```
p2.Label = "Another Point"
```

You can iterate over the fields of a structure as follows:

```
p1 := point{23, 12, "A Point"}
s1 := reflect.ValueOf(&p1).Elem()
typeOfT := s1.Type()
for i := 0; i < s1.NumField(); i++ {
  f := s1.Field(i)
  fmt.Printf("%d: %s %s = %v\n", i, typeOfT.Field(i).
Name, f.Type(), f.Interface())
}
```

› **The source code of arrays.go shows how to deal with arrays in Go**

» **"Passing a large array to a function can be pretty slow. The solution to all these problems is to use slices instead of arrays"**

This image presents the Go code of structures, where you can understand how structures are defined and used in Go

Figure 2

```
package main

import (
    "fmt"
    "reflect"
)

func main() {

    type point struct {
        X     int
        Y     int
        Label string
    }
    p1 := point{23, 12, "A Point"}
    p2 := point{}
    p2.Label = "Another Point"

    s1 := reflect.ValueOf(&p1).Elem()
    s2 := reflect.ValueOf(&p2).Elem()
    fmt.Println("S2= ", s2)

    typeOfT := s1.Type()
    fmt.Println("P1=", p1)
    fmt.Println("P2=", p2)

    for i := 0; i < s1.NumField(); i++ {
        f := s1.Field(i)

        fmt.Printf("%d: %s %s = %v\n", i, typeOfT.Field(i).Name, f.Type(), f.Int
    }

    fmt.Println("P1=", p1)
    fmt.Println("P2=", p2)

    // Array of structures
    // manyPoints := [4]point{}
}
```

This is a complex technique that can show you some of the advanced capabilities of Go. Please keep in mind that if any of the fields of the structure begins with a lowercase letter, you will get an error message when you are trying to call **reflect.Value.Interface()**:

```
epanic: reflect.Value.Interface: cannot return value
obtained from unexported field or method
```

This happens because lowercase fields do not get exported and therefore cannot be used by the **reflect.Value.Interface()** method. You can see the Go code of structures.go in Figure 2 above; its output is the following:

```
$ go run structures.go
S2=  {0 0 Another Point}
P1= {23 12 A Point}
P2= {0 0 Another Point}
0: X int = 23
1: Y int = 12
2: Label string = A Point
P1= {23 12 A Point}
P2= {0 0 Another Point}
```

Creating an array with point structures is possible and can actually be very convenient when you need to declare many point structures:

```
manyPoints := [4]point{}
```

Slices

Slices cannot be found in many programming languages yet they are a smart and handy concept. A slice, which is similar in nature to an array, allows you to overcome the shortcomings of arrays. You can define a slice as follows:

```
aSlice := []int{2, 4, 5, 6, 7, 8}
```

Note that you are writing **[]** to define a slice instead of **[...]** that was used for defining an array. Alternatively, you can define a slice with room for three elements of type int as follows:

```
anotherSlice := make([]int, 3)
```

The **make()** function initialises all elements of a slice; you can verify this by printing the elements of a slice after the **make()** call.

As slices are passed by reference to functions, any modifications you make to a slice inside a function will not be lost after the function ends. You can add one or more elements to a slice with the help of the **append()** function:

```
aSlice = append(aSlice, -100)
```

Slices have a capacity and a length, which are not always the same. The length of a slice is the same as the length of an array with the same number of elements and can be found using the **len()** function.

The capacity of a slice is the current room that has been allocated for this particular slice and can be found with the **cap()** function. As slices are dynamic in size, if a slice runs out of room, Go doubles its current length to make room for more elements.

You can see the source code of **slices.go**, which illustrates slices, in Figure 3, which is on the right-hand page. Executing **slices.go** generates the following output:

```
$ go run slices.go
Before changeOne
2 4 5 6 7 8
After changeOne
2 4 5 -2 7 8
Before. Cap: 6, length: 6
After. Cap: 12, length: 7
0 0 0
```

As you can see, at the time of its definition, the capacity and the length of a slice are exactly the same – after adding another element to the slice using **append()**, its length goes from 6 to 7, but the slices capacity doubles and goes from 6 to 12.

The main advantage you get from doubling the capacity of a slice is better performance, as you do not have to allocate more space all the time.

Please note that passing an array as input to a function that expects a slice will generate an error message and your code will not compile:

```
slices.go:33: cannot use x (type [6]int) as type []
int in argument to printSlice
```

Anonymous functions

The reason for mentioning anonymous functions in this tutorial is because an anonymous function is a special kind of variable. Anonymous functions can be defined inline without the need for a name and are usually used for implementing functions that require a small amount of code. In Go, a function can return an anonymous function or take an anonymous function as one of its arguments.

A slice of an existing array

If you have an existing array, you can create a slice from it:

```
myArray := [4]int{1, 2, 3, 4}
aSlice := myArray[1:4]
```

Subtracting the left index from the right index of the slice reveals the number of elements the new slice will have. The relevant example code can be found in **sliceArray.go**; executing it produces the following output:

```
$ go run sliceArray.go
1 2 3 4
2 3 4
1 3 4 5
```

Be aware, any change to the **aSlice** slice will affect the elements of **myArray**.

Copying a slice

The **copy()** function copies the elements from its second argument, which is an existing slice, into the first argument, which must also exist in advance. The following code showcases the use of the **copy()** function that can be very tricky to use:

```
aSlice := []int{1, 2, 3}
aSliceNew := []int{-1, -2, -3, 4, 5}
anotherSliceNew := []int{-1, -2}
copy(aSliceNew, aSlice)
copy(anotherSliceNew, aSlice)
```

The contents of the three slices after the execution of the two **copy()** calls will be the following:

```
aSlice: 1 2 3
aSliceNew: 1 2 3 4 5
anotherSliceNew: 1 2
```

This demonstrates the fact that if the destination slice is smaller than the source slice, the **copy()** function will only copy the elements that fit into the destination slice. Similarly, if the destination slice is bigger than the source, the **copy()** function will copy all elements from the source slice into the destination slice and leave the remaining elements of the destination slice intact!

Maps

The Map data type in Go is equivalent to the well-known hash table. Other programming languages use the word dictionary to describe a hash table. Maps can use any arbitrary type as an index whereas arrays and slices can only use an integer as an index. The following code defines a new map variable using the **make()** function:

```
aMap := make(map[string]int)
```

The previous command defines an empty map that connects a string with an integer. You can add two elements to **aMap** as follows:

```
aMap[“Mon”] = 0
aMap[“Tue”] = 1
```

Similarly, you can print the value that is associated with a specific key as follows:

```
fmt.Printf(“Sunday is the %dth day of the week.\n”,
aMap[“Sun”])
```

Figure 3

```go
package main

import (
    "fmt"
)

func changeOne(x []int) {
    x[3] = -2
}

func printSlice(x []int) {

    for _, number := range x {
        fmt.Printf("%d ", number)
    }
    fmt.Printf("\n")
}

func main() {

    // This is a slice
    aSlice := []int{2, 4, 5, 6, 7, 8}
    fmt.Printf("Before changeOne\n")
    printSlice(aSlice)
    changeOne(aSlice)
    fmt.Printf("After changeOne\n")
    printSlice(aSlice)

    // Before
    fmt.Printf("Before. Cap: %d, length: %d\n", cap(aSlice), len(aSlice))
    // Add an element to it
    aSlice = append(aSlice, -100)
    // After
    fmt.Printf("After. Cap: %d, length: %d\n", cap(aSlice), len(aSlice))
    // This is another slice
    anotherSlice := make([]int, 3)
    printSlice(anotherSlice)

    // This will not work
    // otherArray := [...]int{2, 4, 5, 6, 7, 8}
    // printSlice(otherArray)
}
```

This is the Go code of slices.go where you can see how to work with the slices function

You can check whether a specific key can be found on an existing map as follows:

```
_, ok := aMap[“Tuesday”]
if ok {
        fmt.Printf(“The Tuesday key exists!\n”)
} else {
        fmt.Printf(“The Tuesday key does not
exist!\n”)
}
```

Structures

As the Go programming language does not support classes using a dedicated keyword such as class or object, you will need to get creative if you find the need to use a class. The best solution is to create a structure instead.

Composition, instead of inheritance, is what Go prefers. The following code briefly shows an example of good composition:

```
type Point struct {
  X    int
  Y    int
Label string
}
type colorPoint struct {
Point // Composition!
Color []string
}
```

› The maps.go file shows how to create and use maps in Go

Figure 4

```
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func main() {
8
9      aMap := make(map[string]int)
10     aMap["Mon"] = 0
11     aMap["Tue"] = 1
12     aMap["Wed"] = 2
13     aMap["Thu"] = 3
14     aMap["Fri"] = 4
15     aMap["Sat"] = 5
16     aMap["Sun"] = 6
17
18     fmt.Printf("Sunday is the %dth day of the week.\n", aMap["Sun"])
19
20     // Check if a key exists on a Map
21     _, ok := aMap["Tuesday"]
22     if ok {
23         fmt.Printf("The Tuesday key exists!\n")
24     } else {
25         fmt.Printf("The Tuesday key does not exist!\n")
26     }
27
28     count := 0
29     // Visit all Map elements
30     for key, _ := range aMap {
31         count++
32         fmt.Printf("%s ", key)
33     }
34     fmt.Printf("\n")
35     fmt.Printf("The aMap has %d elements\n", count)
36
37     count = 0
38     delete(aMap, "Fri")
39     for _, _ = range aMap {
40         count++
41     }
42
```

Figure 5

```
package main

import (
    "fmt"
    "strconv"
)

func main() {
    myArray := [4]int{1, 2, 4, -4}
    aMap := make(map[string]int)

    length := len(myArray)
    for i := 0; i < length; i++ {
        fmt.Printf("%s ", strconv.Itoa(i))
        aMap[strconv.Itoa(i)] = myArray[i]
    }
    fmt.Printf("\n")

    for key, value := range aMap {
        fmt.Printf("%s: %d\n", key, value)
    }

}
```

› The source code of convert.go helps you understand how you can convert an array into a map

» You can iterate over all keys and elements of a map as follows:

```
count := 0
for key, _ := range aMap {
  count++
  fmt.Printf("%s ", key)
}
```

An underscore is used for the parts of the map that you are not going to use. If you wanted to use both the key and the value of a map, the first line of the for loop should have been the following:

```
for key, value := range aMap {
```

Should you want just to count the number of elements on a map, the first line of the for loop should have been as follows:

```
for _, _ = range aMap {
```

Please note that you do not need to use a semicolon in this case as no new variables are declared. Go is strict, so the following code will not compile:

```
count := 0
for key, value := range aMap {
  count++
}
```

The error messages you will get are the following:

```
maps.go:36: key declared and not used
maps.go:36: value declared and not used
```

In other words, you should only declare the values you are going to use. The words key and value are not mandatory – you can use any variable names you want. Figure 4 shows the source code of **maps.go** that produces the following output:

```
$ go run maps.go
Sunday is the 6th day of the week.
The Tuesday key does not exist!
Tue Wed Thu Fri Sat Sun Mon
The aMap has 7 elements
The aMap has 6 elements
```

Define maps

Before you go any further, it is important to remember that the keys are returned in a random order, which might be different from the order you inserted them on the map – this can be seen in the third line of the presented output. If you want to erase a key, you should use the **delete()** function as follows:

```
delete(aMap, "Fri")
```

If the "Fri" key does not exist, Go will not complain. However, if you want to make sure that the key you are going to delete exists, you can use the Go code that checks whether a specific key can be found on a map.

Last, the following command defines a map without the use of the **make()** function:

```
anotherMap := map[string]int{
  "One":  1,
  "Two":  2,
  "Three":  3,
  "Four":  4,
}
```

This code can be suitable when you know the elements of a map in advance.

Quick tip

Apart from defining new types in Go, you can also create methods on these types – this looks like object-oriented programming. Interfaces are abstract types that define a set of functions that need to be implemented so that a type can be considered an instance of the interface. When this happens, we say that the type satisfies this interface.

Figure 6

```
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func main() {
8
9      type point struct {
10         X     int
11         Y     int
12         Label string
13     }
14     aStruct := point{23, 12, "A Point"}
15     aPointer := &aStruct
16     aPointer.X = -1
17     fmt.Printf("The X field is %d.\n", aStruct.X)
18
19     i := 10
20     anotherPointer := &i
21     *anotherPointer = -10
22     fmt.Printf("The value of i is %d.\n", i)
23
24     fmt.Println("The memory address of anotherPointer is", anotherPointer
25
26 }
```

Figure 7

```
1  package main
2
3  import (
4      "io/ioutil"
5  )
6
7  func main() {
8      aByteSlice := []byte("Hello World!\n")
9      ioutil.WriteFile("aByteSlice", aByteSlice, 0644)
10
11 }
```

Converting an array into a map

This section will present a practical example that will show you how to convert an existing array into a map. As you can imagine, such conversions are not always possible; you usually go from a simpler data type to a complex one and not the other way around. Executing **convert.go** produces the following output:

```
$ go run convert.go
0 1 2 3
1: 2
2: 4
3: -4
0: 1
```

You can see the source code of **convert.go** in Figure 5.

Pointers in Go

Go has support for pointers, which are memory addresses that offer speed in exchange for difficult-to-debug code. However, pointers can be pretty useful from time to time. You can create a pointer named aPointer to a structure named aStruct as follows:

```
aPointer := &aStruct
```

Figure 6 shows the source code of **pointers.go**; its output is the following:

```
$ go run pointers.go
The X field is -1.
The value of i is -10.
```

The memory address of **anotherPointer** is 0xc82000a2b0.

Figure 8

```
1  package main
2
3  import (
4      "fmt"
5      "os"
6  )
7
8  func main() {
9
10     filename := os.Args[1]
11     _, err := os.Open(filename)
12     if err != nil {
13         fmt.Println(err)
14         os.Exit(1)
15     } else {
16         fmt.Println("File is open!")
17     }
18 }
```

› The Go code in pointers.go shows how to define and use pointers in Go

› The code in errorHandle.go illustrates the Go approach in error handling

Byte slices

As you can guess by now, slices are first-class citizens in Go. There is another special kind of slice, which is called byte slice and is mainly used for reading and writing operations. The code in **byteSlice.go** creates a new file and writes to it using a byte slice variable.

You can see the source code of **byteSlice.go** in Figure 7 above. Executing byteSlice.go will produce output similar to the following:

```
$ go run byteSlice.go
$ cat aByteSlice
Hello World!
```

You will learn more about byte slices in the next tutorial that will talk about File I/O in Go, over the page. ■

Go error handling

As we all know, errors happen all the time, and more often than not when you are least expecting them. Therefore, you must be prepared in advance, and your code must always be ready to deal with errors. The **maps.go** file includes error-handling code for checking whether a specific key existed on the **aMap** map. Named **errorHandle.go**, it will use file I/O to demonstrate some of the error handling capabilities of Go because lots of things can go wrong when dealing with files and file permissions.
You can see the Go code of **errorHandle.go** in Figure 8 – executing it with various inputs produces the following kind of output:

```
type Point struct {
    $ go build errorHandle.go
$ ./errorHandle convert.go
File is open!
$ chmod 000 convert.go
$ ./errorHandle convert.go
open convert.go: permission
denied
$ ./errorHandle convert1.go
open convert1.go: no such file or
directory
```

Go: Explore functions

Learn how you can develop and use Go functions

Although functions look like a simple concept, Go makes them more powerful and efficient by introducing additional and unique features. Strictly speaking, a Function is an autonomous entity in a computer program that can have input, output, both of them or none of them. Function declarations in Go start with the func keyword. The single most popular Go function is **main()** that is declared as follows:

```
func main() {
// various Go commands
}
```

About functions

Functions are an important element of every programming language. They allow you to break a big program into smaller, manageable parts. Functions need to be as independent from each other as possible. This means that a function must do one job and only one job. If you find yourself writing functions that do multiple things, you might consider writing multiple functions instead!

Another rule of thumb is that if a function is more than 20-30 lines of Go code, you should consider breaking it into smaller functions. The last unofficial rule about function length is that you must be able to see the entire code of a function on your screen at once. These are practical rules, but you should not follow them to the letter all the time – there are always exceptions to a rule.

A side effect of having small functions is that smaller functions can be optimised more clearly because you can easily find out where the bottleneck is.

The last element of a good function is a descriptive function name; therefore a function that implements integer division should not be called **multiplyInts()**!

If you find yourself using the same functions all the time, you might consider putting them into a Go module. This has two advantages: first, you can use your functions more easily and second, in case your functions offer something truly amazing, other people can use them as well because it is easier to distribute packages.

The easy part

Just like almost every programming language Go supports functions with arguments and return values. As you can understand from the **main() function**, a Go function can have no input return no output – as you are going to see, this is rarely the case! The following Go code defines a function that takes one integer variable as input and returns one integer variable as output:

```
func oneArgument(x int) int {
  return x + x
}
```

Note that the variable type follows the name of the variable in the input arguments. As you can imagine, you do not have to declare a variable name for the output. If you have a function with two or more arguments of the same type, you can write it as follows:

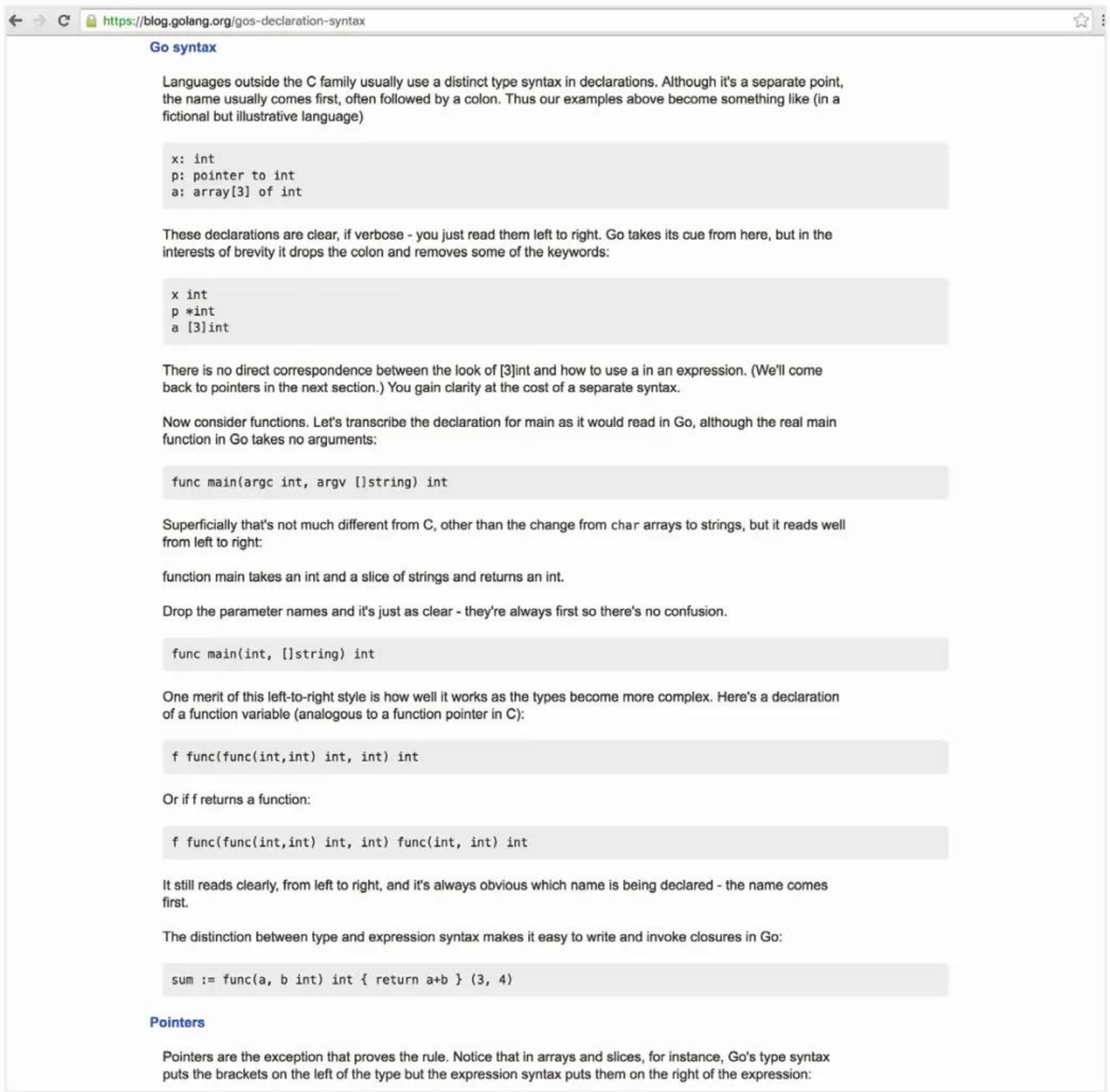
```
func sameType(x, y int) {
  fmt.Println(x + y)
}
```

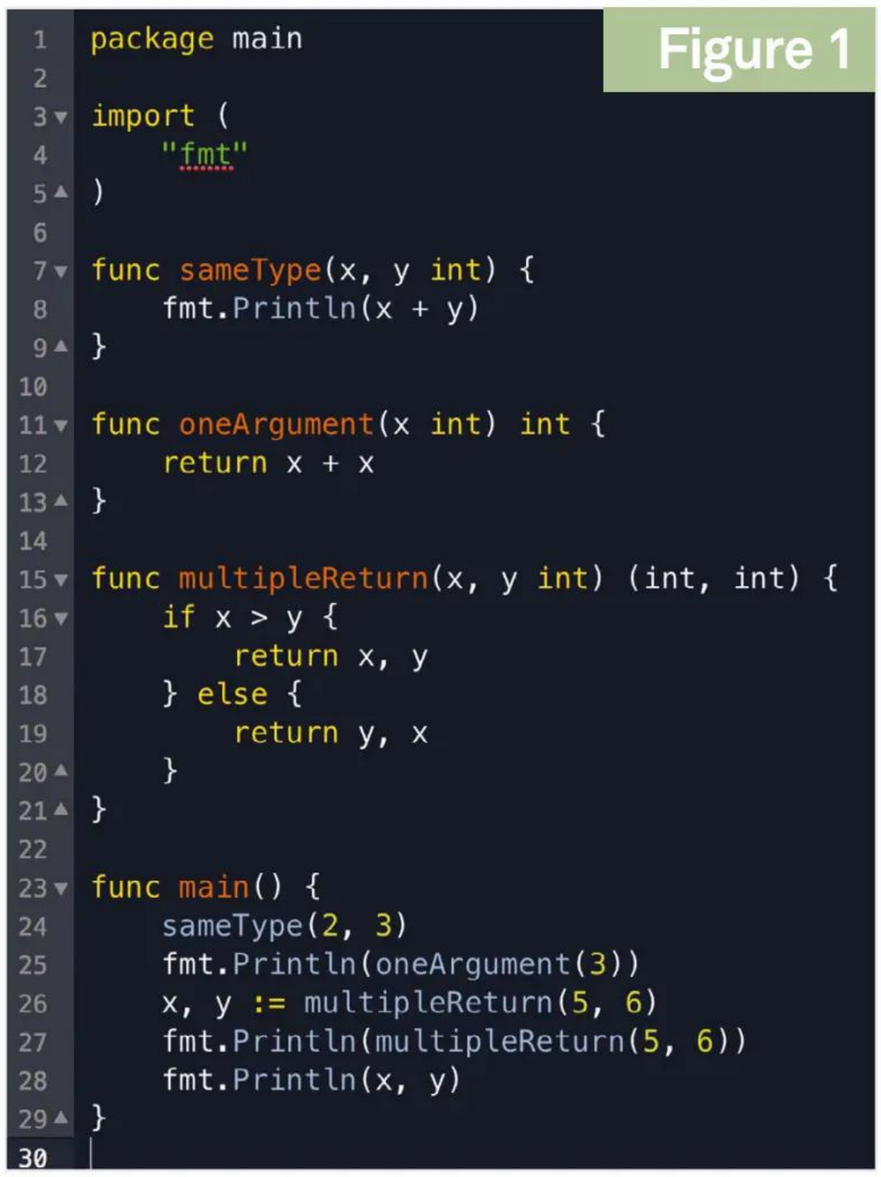
Go has support for functions that return multiple values, so you do not have to create a separate structure to carry the return values and makes your code easier to read:

```
func multipleReturn(x, y int) (int, int) {
  if x > y {
    return x, y
  } else {
    return y, x
  }
}
```

What the **multipleReturn()** function does is sort two integers from the bigger to the smaller one. You can get its output as follows:

Go: Explore functions





This is the source code of simple.go that introduces you to Go functions using simple examples

```
x, y := multipleReturn(5, 6)
```

The same approach can be used for swapping two values without a temporary variable. You can find all aforementioned examples in **simple.go** that is also shown in Figure 1, pictured in the top-right.

Naming return values

It is now time to learn the whole truth about the return values of a Go function: you can name the return values of a Go function! The following code presents a version of **multipleReturn()** that uses named return values:

```
func minMax(x, y int) (min, max int) {
  if x > y {
    min = y
    max = x
  } else {
    min = x
    max = y
  }
  return min, max
}
```

The following variation of **minMax()** will also work:

```
func namedMinMax(x, y int) (min, max int) {
  // The same commands as in minMax()
  return
}
```

The reason the previous statement works is because when you have a return statement without any arguments, Go automatically returns the named return values!

The whole idea is handy because you just have to put a "return" statement in your code and Go will return the appropriate variables! However, named return statements can introduce nasty bugs and damage the readability of your code so you should only use them in very short functions. Named functions return their values in the order they were declared in the definition of the function.

The Go Profiler

A Profiler is a program that allows you to analyse the behaviour of your code and find problems or logical errors. In order to enable profiling, you will need to add some extra code in your Go programs; however, we will not deal with the extra code in this tutorial. The related Go code can be found in **profileMe.go**. Profiling **profileMe.go** requires the following steps and generates an additional file that holds the data:

```
$ go build profileMe.go
$ ls -l profileMe
-rwxr-xr-x 1 mtsouk mtsouk
1829776 May 11 17:48 profileMe
$ ./profileMe -cpuprofile=profileMe
prof
```

```
$ ls -l profileMe.prof
-rw-r--r-- 1 mtsouk mtsouk 64 May
11 18:08 profileMe.prof
$ file profileMe.prof
profileMe.prof: data
$ go tool pprof profileMe profileMe
prof
Welcome to pprof!  For help, type
'help'.
(pprof)
```

The last command allows you to enter the environment of the Go profiler. You will learn more about profiling Go applications in a forthcoming tutorial that will talk about testing Go code.

Go supports named return values. When you use named return values, Go also supports a return statement without any arguments!

Figure 2

```
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func minMax(x, y int) (min, max int) {
8      if x > y {
9          min = y
10         max = x
11     } else {
12         min = x
13         max = y
14     }
15     return min, max
16 }
17
18 func namedMinMax(x, y int) (min, max int) {
19     if x > y {
20         min = y
21         max = x
22     } else {
23         min = x
24         max = y
25     }
26     return
27 }
28
29 func main() {
30
31     fmt.Println(minMax(15, 6))
32     fmt.Println(namedMinMax(15, 6))
33     min, max := namedMinMax(12, -1)
34     fmt.Println(min, max)
35 }
```

The use of named return values does not prohibit you from getting each individual return value as you did previously with **multipleReturn()**. You can find all aforementioned examples in named.go that is also shown in Figure 2. Run **named.go** and try to guess the results!

Anonymous functions

Anonymous functions are simply functions without a name. You can use anonymous functions either as parameters to other functions or as the return values of other functions. If you need a way to keep an anonymous function, you can use a variable. You can declare an anonymous function and attach it to a variable as follows:

```
double := func(s int) int {
  return s + s
}
```

Then you can use the double variable to call the anonymous function:

```
fmt.Println("The double of", y, "is", double(y))
```

The tricky thing here is that if you change the value of the double variable and point it to a different anonymous function, the double variable will have a different meaning. This can generate nasty bugs and reduce the readability of your program.

However, anonymous functions need not always to be attached to variables. In the following code, an anonymous function is declared and used in line:

```
for i := 0; i < 5; i++ {
  myDouble := func(s int) int {
    return s + s
  }(i)
  }(i)
  fmt.Println("The double of", i, "is", myDouble)
}
```

The (i) at the end of the anonymous function is very important because "i" is the parameter of the anonymous function which means that **myDouble** just holds the return value of the anonymous function.

The anonymous function inside the for loop cannot be used elsewhere in the program! That is why such anonymous functions are usually small in size and have a local usage.

You can see the Go code of anonymous.go in Figure 3. Executing **anonymous.go** generates the following output:

```
$ go run anonymous.go
The double of 0 is 0
The double of 1 is 2
The double of 2 is 4
The double of 3 is 6
The double of 4 is 8
The double of 5 is 10
```

The next section shows more examples of anonymous functions combined with defer.

About defer

The defer keyword defers the execution of a function until the surrounding function returns. A very common use of defer is when performing file I/O:

```
r, err := os.Open(aFilename)
if err !=nil {
  panic(err)
}
defer r.Close()
// Process aFilename using r
```

In this case, you declare that you want to close the file assigned to the r variable when the surrounding **functionends**; in the meantime, you can keep using **aFilename**. In other words, you cannot accidentally close **aFilename** and try to use it afterward, which will crash your

Anonymous functions

Bugs are like the Spanish Inquisition, no one knows where to expect them! We recommend always checking functions for bugs. Functions may increase the size of smaller programs. If the developer decides to use pointer variables to functions, they must be extra careful not to introduce nasty bugs. If your program uses many functions, you might find yourself dealing more with the functions than dealing with the actual program, so always stay focused. Apart from these minor disadvantages, your programs will only benefit from the use of functions.

program. This technique also puts the **Close()** function close to the **Open()** function so you cannot accidentally leave **aFilename** open after the function exits.

However, defer can also be used in other cases. You can see the Go code of **myDefer.go** in Figure 4 – **myDefer.go** generates the next output, which is quite interesting:

```
$ go run myDefer.go
3 2 1 0
4 4 4 4
3 2 1 0
```

The first output verifies that deferred functions are executed in LIFO order after the surrounding function returns! As the for loop in **a()** only defers a single function call that uses a given value of the "i" variable, it is logical that all numbers are printed in the reverse order.

The defer call in the **b()** function is extremely tricky. The defer call is evaluated when the for loop ends. However, when the for loop ends, the value of the "i" variable is 4. As the anonymous function is called four times because of the for loop and the value of "i" is 4 when the for loop ends, the number 4 is printed on your screen four times! In other words, you have four function calls that use the last value of a variable because this is what is passed to the function.

As far as the **c()** function call is concerned, its output is the same as the **a()** function because the function to defer takes an argument that keeps its value – as expected the numbers are printed in reverse order because of the LIFO order that deferred functions are executed. Using defer can be tricky, so try to guess their output before executing the code if you attempt to write your own examples.

Functions and errors

Go has a clever way of dealing with errors with the help of pattern matching. Functions can take advantage of this capability and help you deal with errors that might happen inside a function outside of the function. Imagine the next function declaration:

```
func division(x, y int) (int, error) {
  if y == 0 {
    return 0, errors.New("Cannot divide by
zero!")
  }
  return x / y, nil
}
```

The division function returns an integer value and a variable of type error. In other words, a Go function can pass an error message to the main program. The main program will decide how to react to the error message based on the existing policy. The **errors.New()** function is used for converting a string to an error variable. You can see the Go code of **errors.go** in Figure 5. This produces output similar to the following:

```
$ go run errors.go
The result is 1
2016/05/12 08:30:13 Cannot divide by zero!
exit status 1
```

> **"Apart from these minor disadvantages, your programs will only benefit from the use of functions"**

Using pointer variables in functions

Go has support for pointers, which are memory addresses that offer speed in exchange for difficult-to-debug code. The following defines a function that takes a pointer as an argument:

```
func withPointer(x *int) {
  *x = *x * *x
}
```

Figure 3

```
package main

import (
    "fmt"
)

func main() {

    y := 5
    double := func(s int) int {
        return s + s
    }

    for i := 0; i < 5; i++ {
        myDouble := func(s int) int {
            return s + s
        }(i)
        fmt.Println("The double of", i, "is", myDouble)
    }

    fmt.Println("The double of", y, "is", double(y))
}
```

› **The code in anonymous.go illustrates how anonymous functions are declared and used in Go**

Figure 4

```
package main

import (
    "fmt"
)

func a() {
    for i := 0; i < 4; i++ {
        defer fmt.Print(i, " ")
    }
}

func b() {
    for i := 0; i < 4; i++ {
        defer func() { fmt.Print(i, " ") }()
    }
}

func c() {
    for i := 0; i < 4; i++ {
        defer func(n int) { fmt.Print(n, " ") }(i)
    }
}

func main() {
    a()
    fmt.Println()
    b()
    fmt.Println()
    c()
    fmt.Println()
}
```

› **The presented Go code illustrates the way defer works, which is relatively tricky sometimes**

Figure 5

```
package main

import (
    "errors"
    "fmt"
    "log"
)

func division(x, y int) (int, error) {
    if y == 0 {
        return 0, errors.New("Cannot divide by zero!")
    }
    return x / y, nil
}

func main() {

    result, err := division(2, 2)

    if err != nil {
        log.Fatal(err)
    }
    fmt.Println("The result is", result)

    result2, err := division(2, 0)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println("The result is", result2)
}
```

This code demonstrates how error messages are passed from a function to the main program

Figure 6

```
package main

import (
    "fmt"
)

func withPointer(x *int) {
    *x = *x * *x
}

type complex struct {
    x, y int
}

func newComplex(x, y int) *complex {
    return &complex{x, y}
}

func main() {
    x := 4
    withPointer(&x)
    fmt.Println(x)

    w := newComplex(4, 5)
    fmt.Println(*w)
}
```

The code in pointers.go shows how a function can get a pointer as input and how a function can return a pointer

As you can see, you do not need to return the new value – the new value is stored to the argument of the function because it is passed using a pointer!The next code defines a function that returns a pointer to a structure named complex:

```
func newComplex(x, y int) *complex {
  return &complex{x, y}
}
```

The full Go code, which is named **p2Functions.go**, can be seen in Figure 6. Executing **p2Functions.go** creates the following output:

```
$ go run p2Functions.go
16
{4 5}
```

Despite their usefulness, pointers are the origin of many problems and bugs, so only use them when necessary!

Interfaces

Interfaces are an advanced Go feature that will be explained here as gently as possible, though you may want to do some more research in order to fully understand their place here. Still, the basic information that you need to be aware of is that although Go has no direct support for objects and hierarchies, interfaces can be used for mimicking object-oriented programming. An interface is two things: a set of methods and a type. Interfaces are used for defining the behaviour of an object. Suppose that you have the following Go code that defines a new type named complex:

```
type complex struct {
  X int
  Y int
}
```

The complex type is a structure with two integers fields. You can now create an interface that you want to be used by the complex type. The most important part of interface.go is the following because this where the interface is defined:

```
type complexParts interface {
  real() int
  imaginary() int
}
```

It is important to notice that the interface does not state any other specific types apart from the interface itself. On the other hand, the two functions of the interface should state their return values.

The definition of the **complexParts** interface says that if you want to confront to the **complexParts** interface, you will have to implement two functions named **real()** and **imaginary()**. The main advantage of it is that if you have another type that contains a complex number in some form, you can make it confront to the complexParts interface by implementing the **real()** and **imaginary()** functions.

Advantages of functions

The biggest advantage of functions is that you do not have to write the same code many times in your programs; this improves the maintainability of your code. A side effect of maintainability is that you only have to look at one place for bugs. By contrast, if you optimise the code of a function, all optimisations are automatically propagated to the rest of the program. Additionally, functions improve the readability of a program.

Figure 7

```
type complexParts interface {
    real() int
    imaginary() int
}

type complex struct {
    X int
    Y int
}

func (s complex) real() int {
    return s.X
}

func (s complex) imaginary() int {
    return s.Y
}

func findComplexParts(a complexParts) {
    fmt.Println("R:", a.real())
    fmt.Println("I:", a.imaginary())
}

type myInt int

func (s myInt) real() int {
    return int(s)
}

func (s myInt) imaginary() int {
    return 0
}

func main() {

    x := complex{X: 1, Y: 2}
    fmt.Println(x)
    findComplexParts(x)

    y := myInt(10)
    findComplexParts(y)
}
```

› The Go code of interface.go illustrates the use of interfaces in Go

The following code illustrates this by treating integer numbers as complex numbers with an imaginary part of 0:

```
type myInt int
func (s myInt) real() int {
  return int(s)
}
func (s myInt) imaginary() int {
  return 0
}
```

From now on, you can use the f**indComplexParts()** function for both complex and **myInt** types! In Object Oriented terminology, this means that both complex and **myInt** types are **complexParts** objects. But none of them is just a **complexParts** object. View the source code of **interface.go** in Figure 7, stage left. Executing **interface.go** will generate:

```
{1 2}
R: 1
I: 2
R: 10
I: 0
```

The more you use interfaces the more you will fully understand their usefulness and their true importance. ■

Figure 8

```
package main

import (
    "fmt"
)

func main() {
    x := 0

    if x == 0 {
        fmt.Println("x equals 0!")
        return
    } else {
        fmt.Println("x is not equal 0!")
        return
    }

    fmt.Println("Reach me if you can!")

}
```

› Go offers a tool that can help you detect unreachable code, which is a sign of a design error or a bug in your code

The "go vet" command

Sometimes, your code has logical errors that are hard to find because they are still valid Go code. One of these logical errors is the presence of unreachable code, which is source that can never be executed because there is no path to it. Go offers a command (known as 'go vet') that can help you detect such errors. The following example Go code, included in **unreachable.go**, contains unreachable code:

```
if x == 0 {
fmt.Println("x equals 0!")
return
} else {
  fmt.Println("x is not equal 0!")
  return
}
fmt.Println("Reach me if you can!")
```

Running "go vet" against it will produce the following output:

```
$ go vet unreachable.go
unreachable.go:18: unreachable code
exit status 1
```

You can see the pretty simplistic Go code of **unreachable.go** in Figure 8. As you can guess, the go vet command is more suitable for larger source files.

CODING ACADEMY

CODING ACADEMY

Reference

Stumped by a coding concept? You've come to the right place

140 **Get to grips with Python lists**
It's one thing after another, isn't it?

142 **Understanding functions and objects**
All about things and what things do

144 **Adapt and evolve with conditionals**
Because change is inevitable

146 **Variable scope of various variables**
There's a variety of variables, and they vary

148 **Recursion: round and round we go**
Repetition is no bad thing

150 **Super sorting algorithms**
Get everything orderly, and quick

152 **Hidden secrets of numbers**
Integers thoroughly demystified

154 **Using loops and using loops**
And using loops and using loops

156 **The magic of compilers**
How they translate your code

158 **Avoid common coding mistakes**
Get it right the first time

Reference

Get to grips with Python lists

Let's start by exploring some of the fundamental ideas behind programming logic, beginning with the concept of lists

It's difficult to know where to start when it comes to explaining the basic programming concepts. When someone begins coding, there's an overwhelming number of different ideas to understand, absorb and eventually turn into solutions. And that's on top of the syntax and peculiarities of your chosen language.

One of the best approaches is to just start writing code, either by following an example project, such as the ones we provide in this book, or by piecing together examples from documentation. Both methods work, and they're the only real way to get to grips with the problems and complications of any particular concept.

We'd like to make this challenge slightly easier by looking at basic approaches that apply to the vast majority of circumstances and languages. And our first target is lists.

You don't get far with any moderately complex task without a list, whether that's a holiday to the Seychelles or a trip to the supermarket. And list logic in code is just like in real life. A list is a convenient place to store loosely-related snippets of information. Programmers like to call loosely-related snippets of information a data structure, and a list is just another example. With a shopping list, for instance, the relationship might be something as simple as food – 'milk', 'bread' and 'baked beans'. If you were writing an application to remind you what's on your shopping list, it would make much more sense to put these values together. In Python, for example, you could create a list of those values with the following line:

```
>>> shoppinglist = ["milk", "bread", "baked beans"]
```

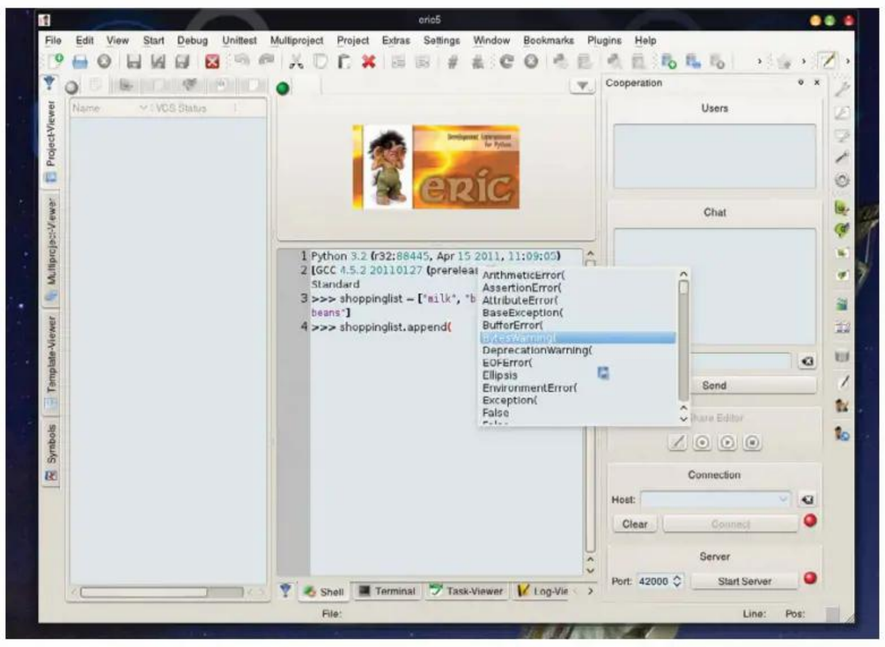
This is technically a list of lists, because each value is itself a list – a string of characters. Some languages like to make that distinction, but most prefer the convenience.

Each of these items has been added to the freshly created **shoppinglist** list. But lists are dynamic, and one of the first things you often want to do is add new values.

This is where lists start to become interesting, because the method for adding and removing values can help to define the list's function. For example, it's faster for the CPU to add a new item to the end of a list because all the application has to do is link the last element to the new one.

If you insert an item into the middle of a list, the process first splits the links between the two items on either side of the insertion point, then links both the preceding item and the following one to the new item.

› If you're learning Python, the tab completion in the Eric programming environment is a great helper



Stacks and queues

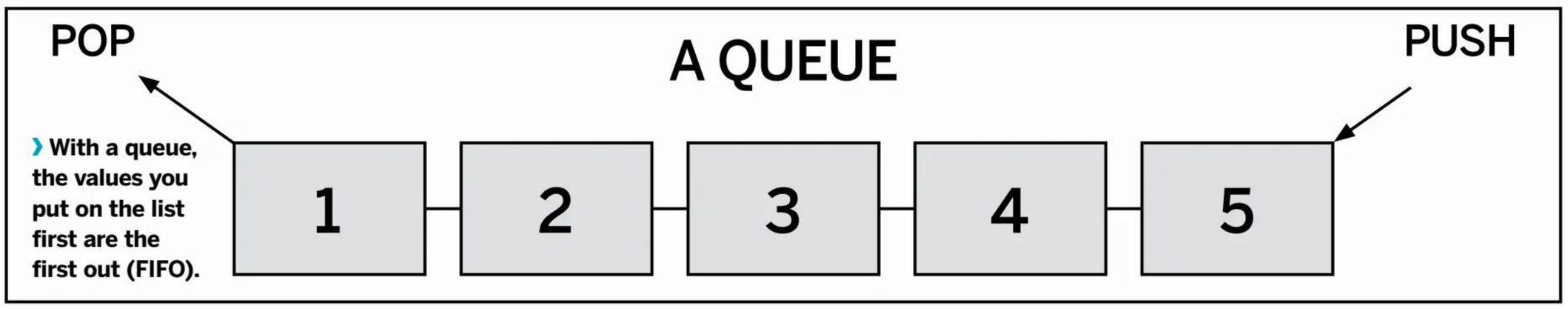
The most important characteristic for a list is that the chain of data it contains is in a specific order. It might not be important what that order is, but it's the order that differentiates a list from a random array of values, such as a chunk of memory.

It's for this reason that some of the earliest lists are stacks, with values either pushed on to the end or pulled off, rather than lists with values inserted and removed from the middle, which are more processor intensive.

Processing speed means that stacks aren't a necessity any more, but they're still useful for temporarily holding values, for example before retrieving them in reverse order like a stack of cards. The list could be your terminal command history, or a browser's 'Back' button.

In Python, you can execute a function called **append** to the same effect:

```
>>> shoppinglist.append("soup")
```



› With a queue, the values you put on the list first are the first out (FIFO).

This will add **soup** to our shopping list. When we've added it to our shop, it can be removed easily with **pop**:

```
>>> shoppinglist.pop()
```

If you're running these commands from the interpreter, you'll see the contents of the last value in the list as it's popped off the stack, in our case the word **soup**. If you've got processing limitations in mind, you might wonder why there isn't an option to pop the first value off the stack, too.

This would require the same amount of relinking as popping off a value from the end. It's perfectly possible, and turns a list into something called a 'queue' – the values you put on the list first are first out, known as FIFO. This is in contrast to a stack, which is LIFO.

There are many ways to use a queue, but the most obvious is to preserve the order of incoming data. As we explained, the only difference between a queue and a stack is where the **pop** value comes from, which is why Python doesn't have any special commands for accessing the first value, it just uses an argument for the **pop** command. The following will remove the first item in the list, and should output 'milk' from the interpreter:

```
>>> shoppinglist.pop(0)
```

Modern lists

Lists are no longer limited by processor speed to stacks and queues, and most modern languages and frameworks provide a vast number of handy functions for dealing with them. These offer a big advantage over the original primitive operations, and can turn lists into super-flexible data types.

Python is particularly good at this, and includes plenty of built-in functions for manipulating lists without having to write your own code. You can output a value from any point in the list by treating it as an array. Typing **shoppinglist[1]**, for instance, will return the second value in the list. As with nearly all things code-wise, **0** is the real first element in a list and array, making **1** the second. And if you want to output the entire list, you typically need to construct a simple loop that shuffles through these values from beginning to end.

Python is an exception, because simply typing **shoppinglist** will output the list's contents, but for other languages, you might have to construct something like:

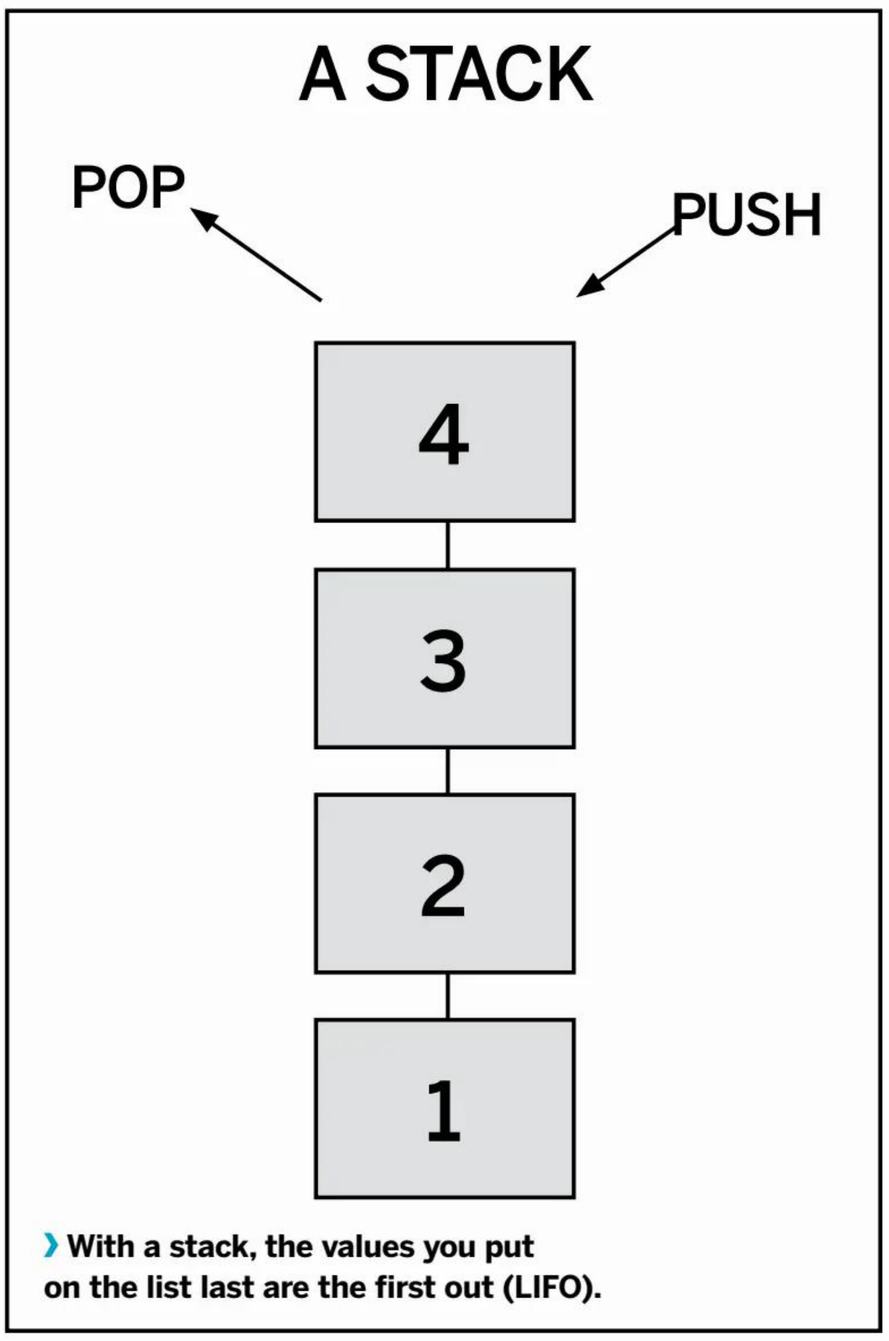
```
>>> for x in range(0,len(shoppinglist)):
...     shoppinglist[x]
...     print
```

The **for** loop here gives a value to **x** that steps from **0** to **len(shoppinglist)**, which is a method to return the length of a list. Each location in the list is output as we loop through it in the order they were added (FIFO).

You can change the order in lots of ways, but Python's **sort** method is tough to beat:

```
>>> shoppinglist
['baked beans', 'bread', 'milk', 'soup']
>>> shoppinglist.append("apples")
>>> shoppinglist
['baked beans', 'bread', 'milk', 'soup', 'apples']
>>> shoppinglist.sort()
>>> shoppinglist
['apples', 'baked beans', 'bread', 'milk', 'soup']
```

If you can follow the logic of that simple piece of code, then it's safe to assume you've grasped the logic of lists. There's really nothing more to it, and now we hope you're wondering what all the fuss is about. The main problem is that it can get complicated quickly, and because lists are shorthand for so many different data types, it can be difficult working out what might be happening in any one chunk of code. This is especially true of Python – because it has so many convenience functions, unless you know what they do, you're unlikely to grasp the logic. But at their heart, they're just a chain of values, and that's the best place to start. ■



› **With a stack, the values you put on the list last are the first out (LIFO).**

You might wonder why we have three **>** symbols preceding the snippets of code on these pages, as well as in our more comprehensive Python tutorials.

These symbols represent the cursor of Python's interactive interpreter, and they help to distinguish between something you might want to build into a script and something you can try immediately by typing in to the interpreter – although they are, in fact, interchangeable.

Launching the interpreter is as simple as typing **python** from the command line without any further arguments. It's at this point you'll see the cursor symbols, and you can now start typing lines of Python code.

This is a brilliant way of learning how the language works, as well as experimenting with syntax and the concepts you're working with. Errors in a single line are far easier to spot than those in a 20-line script that's being run for the first time, and you can also get a really good feel for how Python, and programming in general, handles logic and data. If you execute a function that returns a value, for example, such as **len(list)** to return the length of a list, the value is output to your terminal immediately, rather than needing any explanation in a script or assigning to another variable. You can quit the interpreter by pressing [Ctrl]+[D] together, but you will obviously lose the current state of your code.



› **If you're learning to program, the Python interpreter can teach you a great deal, regardless of your chosen language**

Quick tip

We're using Python for our examples because it's a great language for beginners. But every concept we discuss works with other languages, too – it's just a matter of finding the correct syntax for the functionality.

Understanding functions & objects

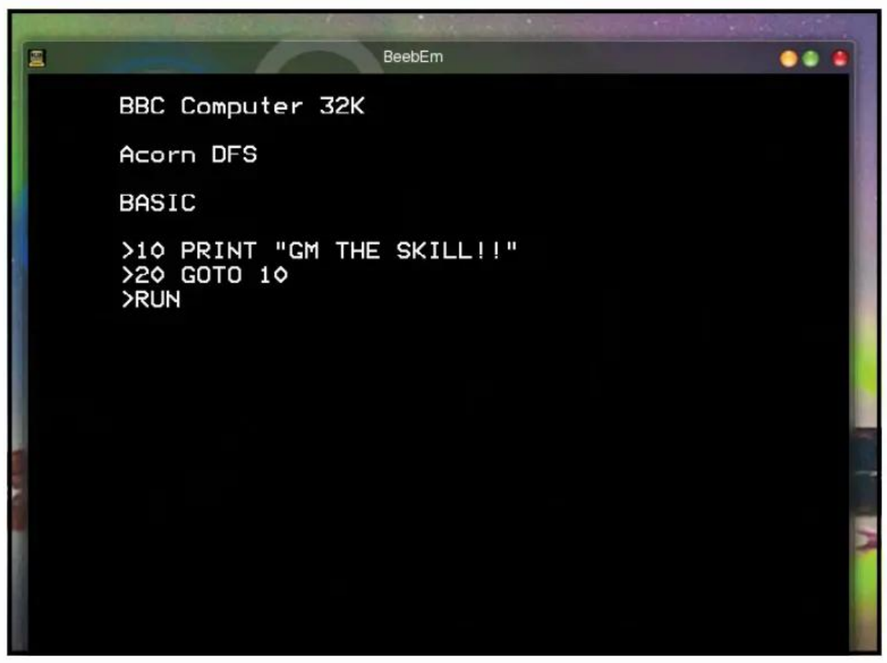
After introducing the concept of lists, it's time for us to tackle the fundamental building blocks behind any application

We've already covered a concept called a list. Lists have been around since people started to write one line of code after another because, in essence, they simply store one value after another.

But what must have come soon after lists in the mists of language creation is the idea of a function, because a function is a kind of list for code logic. It's not built at runtime, but while you're constructing your project, and much like their mathematical counterparts, functions are independent blocks of code that are designed to be reused.

Functions are fundamental for all kinds of reasons. They allow the programmer to break a complex task into smaller chunks of code, for instance, and they allow you to write a single piece of code that can be reused within a project. You can then easily modify this code to make it more efficient, to add functionality or to fix an error. And when it's one piece of code that's being used in lots of places, fixing it once is far easier than trawling through your entire project and fixing your same broken logic many times.

Functions also make program flow more logical and easier to read, and you can keep your favourite functions and use them again and again. When you break out of single-file projects, functions become the best way of grouping functionality and splitting your project into different source files, naturally breaking complex ideas into a group of far easier ones. If you then work with a team of programmers, they can work on functions and files without breaking the operation of the application, and it's this idea that led to the creation of objects. These are a special kind of function that bundle both the code logic and the data the logic needs to work into completely encapsulated blocks. But let's start at the beginning. If your first programming project was from about 30 years ago, and written on a home computer, then



› BBC BASIC was the first programming language many of us came into contact with

there's a good chance it was written in BASIC – a language embedded within the ROMs of many of the home computers of the early eighties.

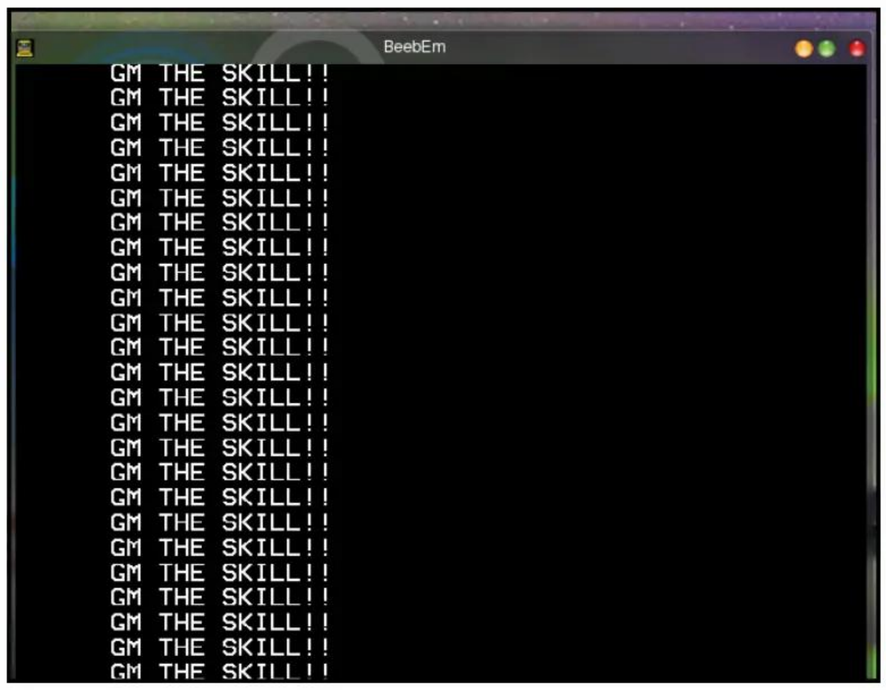
BASIC code was mostly sequential, with many systems even using line numbers, and there's a good chance that the following might have been your first program:

```
10 PRINT "Hello world!"
```

This led to one of the most primitive hacks of the day. You had to find a computer shop displaying a selection of machines. You'd then sneak in and quickly type the above line followed by **20 GOTO 10**, **RUN** and a carriage return. This would create an infinite loop that printed 'Hello World' for ever – or for the 10 minutes it took for the store staff to notice and wearily turn it off and on again. The cooler kids would change the text to say something rude and add colour to the output. Either way, it's a basic example of a function if you can execute it from another part of your project.

BASIC programming without functions can be a little like writing assembly language code. If you want to repeat a series of lines, you need to remember where you are in the code, jump to the new code, and jump back after execution. BASIC provided some of this functionality with procedures, but functions give that chunk of code a label, which can then be called from any other point in your project. Functions can also return a value, such as the sum of a group of numbers, and accept arguments. That could be a list of numbers to be added, for example. But you'd have needed to be a wealthy Acorn Archimedes owner to get those facilities out of BASIC.

With Python, you can try these ideas without getting too retro. From the interpreter, for example (just type **python** on the command line and add our code), type the following:



› BASIC wasn't always put to the best uses

```
>>> def helloworld():
...     print (“Hello World”)
...     return
...
```

As ever with Python, you need to be very careful about formatting. After creating a new function called **helloworld()** with the **def** keyword, the interpreter precedes each new line with **...**. Any lines of code you now add will be applied to the function, and you'll need to press [Tab] at the beginning of each to indent the code and make sure Python understands this. Other languages aren't so picky. After the **return** statement, which signals the end of the function, press [Return] to create a blank line. You've now created a function, and you can execute it by typing **helloworld()** into the interpreter. You should see the output you'd expect from a function with a name like **helloworld()**, and you've just jumped forward 30 years.

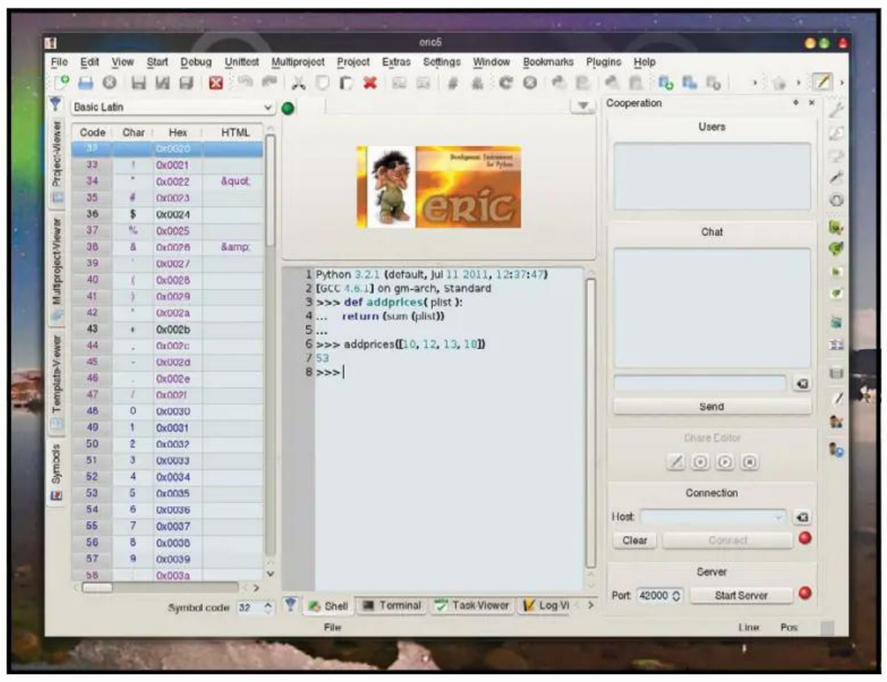
Passing arguments

There's a problem with our new function: it's static. You can't change anything. This can be solved by opening the function up to external influence, and you can do that by enabling it to pass and process arguments. If you alter the function definition to **def helloworld(str)**, for instance, Python will expect to find a string passed to the function whenever it's called. You can process this string within the function by changing **print (“Hello World”)** to **print str**, but you should obviously be doing anything other than simply printing out what's passed to the function. It can now be executed by typing **helloworld(“something to output”)** and you'll see the contents of the quotes output to the screen.

To show how easy it is to create something functional, we'll now build on the idea of lists from the previous tutorial. Instead of a shopping list, we'll create a list of prices and then write a function for adding them together and returning the total value. Here's the code:

```
>>> def addprices( plist ):
...     return (sum (plist))
...
>>> pricelist = [1.49, 2.30, 4.99, 6.50, 0.99]
>>> addprices(pricelist)
16.27
```

Python is very forgiving when it comes to passing types to functions, which is why we didn't need to define exactly what **plist** contained. Most other languages will want to know exactly what kind of types an argument may contain before it's able to create the function, and that's why you often get definitions before the program logic, and why types are often declared in header files because these are read before the main source code file.



› **An IDE, such as *Eric*, can make working with functions and objects easier**

Working out what your language requires and where is more difficult than working out how to use functions. And that's why they can sometimes appear intimidating. But Python is perfect for beginners because it doesn't have any convoluted requirements – apart from its weird tab formatting. All we've done in our code is say there's going to be some data passed as an argument to this function, and we'd like Python to call this **plist**. It could even have had the same name as the original list but that might look confusing.

The only slight problem with this solution is that you'll get an error if you try to pass anything other than a list of numbers to the function. This is something that should be checked by the function, unless you can be certain where those numbers come from. But it does mean you can send a list of values in exactly the same way you define a list:

```
>>> addprices([10, 12, 13, 18])
53
```

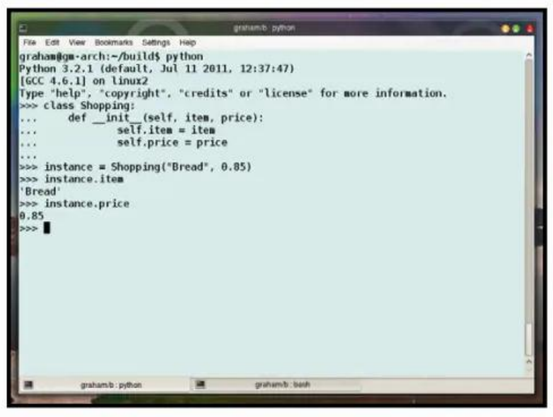
The biggest challenge comes from splitting larger problems into a group of smaller ones. For a programmer, this task becomes second nature, but for a beginner, it's an intimidating prospect. The best way is to first write your application, regardless of functions or programming finesse. Often, it's only when you're attempting to write your first solution that better ones will occur to you, and you'll find yourself splitting up ideas as the best way to tackle a specific issue within your project. The best example is a GUI, where it's common sense to give each menu item its own function, which can be launched from either the icons in the toolbar or by selecting the menu item. Many developers consider their first working copy to be a draft of the final code, and go to the trouble of completely rewriting the code after proving the idea works. But don't worry about that yet. ■

The art of objects

An object bundles functions and the type of data they accept together, so that there's less ambiguity on how a function should be used, and better separation from the main flow and side processes. Dealing with objects can quickly get complicated, partly because there are so many different concepts, language-specific features and implementations, and partly because it requires the programmer to think about problems differently. This simple example creates a class that contains both the **item** and the **price** from our previous shopping list:

```
>>> class Shopping:
...     def __init__(self, item, price):
...         self.item = item
...         self.price = price
...
>>> instance = Shopping(“Bread”, 0.85)
>>> instance.item
‘Bread’
>>> instance.price
0.85
```

A class is a bit like the specification for an object, whereas the term **object** refers to when specification is used within your code. In our example, the class is the structure of **Shopping** while the object is **instance**. The function called **__init__** is run when the object is created, and we pass the two values – one for the item's name and the other for its price. You should be able to see how this class could be quickly expanded to include other functions (or methods), and how the world of classes and objects can transform your coding projects for the better.



› **You can still create classes and functions from within the interpreter, but it begins to make less sense**

Quick tip

When you're using other packages, only the functions are exposed. You don't need to know how they work, just what input they're expecting and what output they return. That's functional programming in a nutshell.

Reference

Adapt and evolve with conditionals

Any non-trivial program needs to make decisions based on circumstances. Get your head around coding's ifs and buts

Murray Walker, the great Formula 1 commentator, used to say "IF is F1 spelled backwards." Ifs, buts and maybes play a vital role in motor racing, as they do in computer programming. If you're writing a program that simply processes and churns out a bunch of numbers, without any user interaction, then you might be able to get away without any kind of conditional statements. But most of the time, you'll be asking questions in your code: if the user has pressed the [Y] key, then continue. If not, stop. Or if the variable PRICE is bigger than 500, halt the transaction. And so forth.

A condition is just a question, as in everyday life: if the kettle has boiled, turn off the gas. (We don't use newfangled electricity around here.) Or if all the pages have gone to the printers, go to the pub. Here's an example in Python code:

```
x = 5

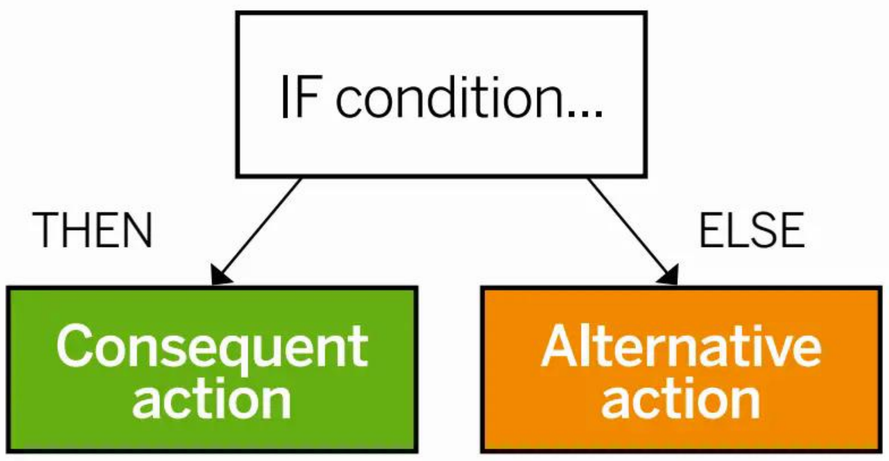
if x == 10:
    print "X is ten"
else:
    print "X is NOT ten"

print "Program finished"
```

Here, we create a new variable (storage place for a number) called **X**, and store the number **5** in it. We then use an **if** statement – a conditional – to make a decision. If **X** contains **10**, we print an affirmative message, and if not (the **else** statement), we print a different message. Note the double-equals in the **if** line: it's very important, and we'll come on to that in a moment.

Here, we're just executing single **print** commands for the **if** and **else** sections, but you can put more lines of code in there, providing they have the indents, Python style:

```
if x == 10:
    print "X is ten"
    somefunction()
else:
    anotherfunction()
```



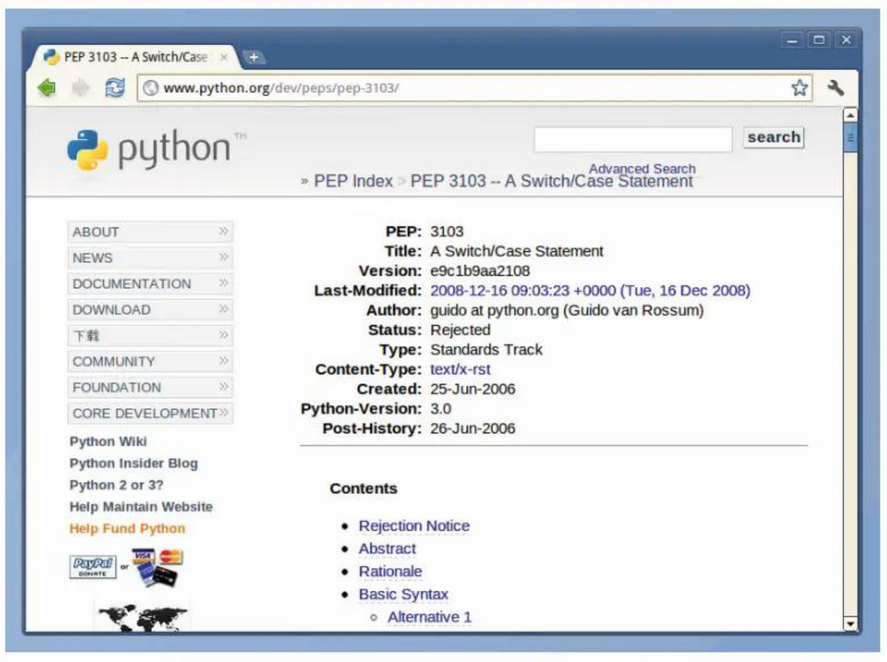
At its core, a conditional statement is something like this

In this case, if **X** contains **10** we print the message as before, but then call the **somefunction** routine elsewhere in the code. That could be a big function that calls other functions and so forth, thereby turning this into a long branch in the code. There are alternatives to the double-equals we've used:

- **if x > 10** If X is greater than 10
- **if x < 10** If X is less than 10
- **if x >= 10** If X is greater than or equal to 10
- **if x <= 10** If X is less than or equal to 10
- **if x != 10** If X is NOT equal to 10

These comparison operators are standard across most programming languages. You can often perform arithmetic inside the conditional statement, too:

```
if x + 7 == 10:
    print "Well, X must be 3"
```



Python doesn't have switch/case. Why? Read the explanation at www.python.org/dev/peps/pep-3103/

Comparing function results

While many **if** statements contain mathematical tests such as above, you can call a function inside an **if** statement and perform an action depending on the number it sends back. Look at the following Python code:

```
def truefunc():
    return 1

def falsefunc():
    return 0

print "Execution begins here..."

if truefunc():
    print "Yay"
```

```
if falsefunc():
    print "Nay"
```

The first four lines of code define functions (subroutines) that aren't executed immediately, but are reserved for later use. They're really simple functions: the first sends back the number 1, the second zero. Program execution begins at the **print** line, and then we have our first **if** statement. Instead of doing a comparison, we call a function here, and act on the result.

If the **if** statement sees the number **1**, it goes ahead with executing the indented code, if not, it skips past it. So when you run this, you'll see **Yay** but not **Nay**, because **if** here only does its work if it receives the number **1** back from the function it calls. In Python and many other languages, you can replace **1** and **0** with **True** and **False** for code clarity, so you could replace the functions at the start with:

```
def truefunc():
    return True

def falsefunc():
    return False
```

and the program will operate in the same way.

Clearing up code

In more complicated programs, a long stream of **ifs** and **elses** can get ugly. For instance, consider this C program:

```
#include <stdio.h>

int main()
{
    int x = 2;

    if (x == 1)
        puts("One");
    else if (x == 2)
        puts("Two");
    else if (x == 3)
        puts("Three");
}
```

C, and some other languages, include a **switch** statement which simplifies all these checks. The lines beginning with **if** here can be replaced with:

```
    switch(x) {
        case 1: puts("One"); break;
        case 2: puts("Two"); break;
        case 3: puts("Three"); break;
    }
```

This is neater and easier to read. Note that the **break** instructions are essential here – they tell the compiler to end the switch operation after the instruction(s) following **case** have been executed. Otherwise, it will execute everything following the first matching case.

Another way that some programmers in C-like languages shorten **if** statements is by using ternary statements. Consider the following code:

```
if (x > y)
    result = 1;
else
    result = 2;
```

This can be shortened to:

```
result = x > y ? 1 : 2;
```

Here, if **X** is bigger than **Y**, result becomes **1**; otherwise it's **2**. Note that this doesn't make the resulting code magically smaller – as powerful optimising compilers do all sorts of tricks – but it can make your code more compact.

So, that's conditionals covered. Although we've focused on Python and C in this guide, the principles are applicable to nigh-on every language. And it all boils down to machine code at the end of the day – the CPU can compare one of its number storage places (registers) with another number, and jump to a different place in the code depending on the result. Remember to thank your CPU for all the hard work it does. ■

Assignment vs comparison

In some languages, especially C, you have to be very careful with comparisons. For instance, look at this bit of C code – try to guess what it does, and if you like, type it into a file called **foo.c**, run **gcc foo.c** to compile it and then **./a.out** to execute it.

```
#include <stdio.h>

int main()
{
    int x = 1;

    if (x = 5)
        puts("X is five!");
}
```

If you run this program, you might be surprised to see the **X is five** message appear in your terminal window. Shurley shome mishtake? We clearly set the value of **X** to be **1**! Well actually, we did, but in the **if** line we then performed an assignment, not a comparison. That's what the single equals sign does.

We're not saying "if X equals 5", but rather, "put 5 in X, and if that succeeds, execute the code in the curly brackets". Ouch. If you recompile this code with **gcc -Wall foo.c** (to show all warnings), you'll see that **GCC** mentions **assignment used as truth value**. This is an indication that you might be doing something wrong.

The solution is to change the **if** line to **if (x == 5)** instead. Now the program runs properly. It's a small consideration, but if you've just written a few hundred lines of code and your program isn't behaving, this could be the root cause. It's caught us out many times...

Big ifs and small ifs

In C, and other languages that share its syntax, you don't need to use curly brackets when using an **if** statement followed by a single instruction:

```
if (a == 1)
    puts("Hello");
```

But only one instruction will be executed. If you do something like this:

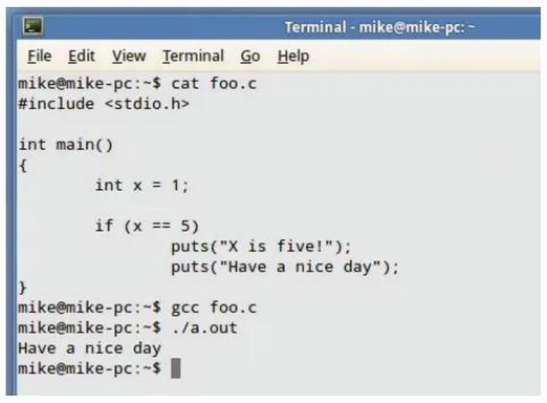
```
int x = 1;

if (x == 5)
    puts("X is 5");
    puts("Have a nice day");
```

When you execute it, it won't print the first message, but it will print the second. That's because only the first is tied to the **if** statement. To attach both instructions to the **if**, put them in curly braces like the following:

```
if (x == 5) {
    puts("X is 5");
    puts("Have a nice day");
}
```

Contrast this with Python, where the indentation automatically shows which code belongs to which statement. C doesn't care about indentation – you have to explicitly show what you want with curly brackets.



› Here we see how indentation isn't good enough for C – we need to use curly braces to bundle code.

Reference

Variable scope of various variables

Caution! That variable you're accessing might not be what you think it is. Why? Because variables are variable. Have we said 'variable' enough now?

Once upon a time, programs were just big lists of instructions, and these instructions could do anything with variables and the contents of memory. This was fine for simple, low-level programs but as time went on and people starting making bigger and more complicated programs, the idea that one part of the program could do anything with memory became less appealing.

For instance, you might have a program that controls a robotic arm, and stores the state of the arm in a variable called **X**. Your program calls lots of different subroutines, some of which may have been programmed by other people. How do you know that those routines aren't messing about with the RAM where **X** is stored?

The results could be disastrous, especially when the arm starts punching people instead of stroking puppies as originally envisaged. This is where the concept of variable scope comes into play. In programming, scope defines how a variable is visible to the rest of the program. In most high-level languages, we can control the scope of a variable – that is, we can choose whether it should be visible to the current chunk of code, to the current source code file, or absolutely everywhere in all of the program's files.

This level of control is absolutely fundamental to good program design, so that multiple programmers can work together on a project, implementing their own routines without accidentally trampling over one another's data. It's good for modularisation and keeping data safe and secure, and therefore it's a good thing to get right in the early days of your programming journey. Let's start with a bit of Python code. Try this:

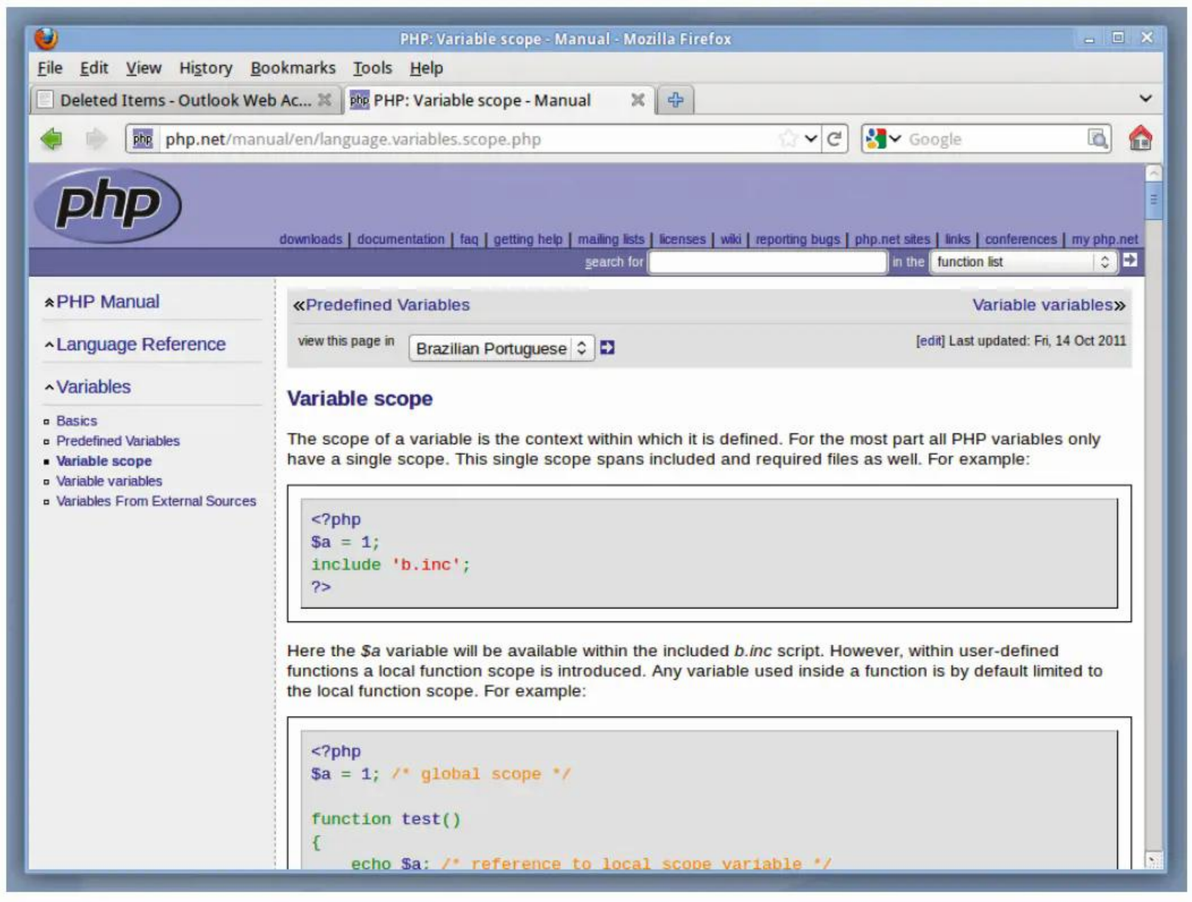
```
def myfunc():
    x = 10
    print x


x = 1
print x
myfunc()
print x
```

If you're totally new to Python, the **def** bit and the following two lines of code are a function which we call later. Program execution begins with the **x = 1** line. So, we create a variable called **x** and assign it the number **1**. We print it out to confirm that. We then call a function which sets **x** to **10**, and prints it. As control jumps back to the main chunk of our code, we print **x** again... and it's back to **1**. How did that happen? Didn't we just set **x** to **10** in the function?

Well, yes, but the function had its own copy of the variable. It didn't do anything with the **x** variable that was declared outside of it. The function lives happily on its own, and doesn't want to interfere with data it doesn't know about. This is essential to keeping code maintainable – imagine if all variables were accessible everywhere, and you were writing a routine to be dropped inside a 50,000-line software project. You'd be totally terrified of choosing variable names that might be in use elsewhere, and accidentally changing someone else's data.

› Most well-documented languages have explanations of variable scope, so once you've got the basics from this article, you can explore specific implementations



Change of routine

So most programming languages provide this level of protection. Still, there are legitimate reasons why you might want to make a variable accessible to other routines, and in Python you can do this by inserting the following line into the start of the **myfunc()** routine:

```
global x
```

This makes all the difference. Previously, the version of **x** we were using in **myfunc()** was a local variable – that is, it affects only the code located inside the function. By adding the **global** command, we access **x** as a global variable – that is, it's the same one as we used in the main body of the code. So if we run this Python code now, we see that **x** is set to **1** at the start, but then the **myfunc()** routine grabs that **x** as a global variable and sets it to **10**, and it stays that way back in the main code.

How variable scope is handled varies from language to language, so let's look at another implementation here, this time in C. Whereas Python is pretty flexible about using

variables that you haven't defined previously, C is a little more strict. Consider this:

```
#include <stdio.h>

void myfunc();

int main()
{
    int x = 10;
    printf("x is %d\n", x);
    myfunc();
    printf("x is %d\n", x);
}

void myfunc()
{
    int x = 20;
    printf("x is %d\n", x);
}
```

Here, because we're explicitly creating variables with **int**, it's somewhat clearer that **myfunc()** has its own version of the variable. But how do you go about making the variable global in this particular instance?

The trick is to put the variable declaration outside of the functions. Move **int x = 10;** from inside **main()** to after the **void myfunc();** line at the top, and now the variable has become global – that is, it's accessible by all functions in the file.

However, that's not the only job you need to do. Inside the **myfunc()** routine, we still have a line that says **int x = 20;**. The **int** here is still creating its own version of the variable, so take it away so that the line just reads **x = 20;**. Run the program, and you'll see the intended result, that **x** is global and can be modified everywhere.

Note that while putting the **x** declaration outside of functions makes it global to the current source code file, it doesn't make it global absolutely everywhere in all of the source code. If you've got a big project with multiple C files, you'll have to use the **extern** keyword. So if you have **int x = 10;** in **foo.c** and you want to use that in **bar.c**, you'd need the line **extern int x;** in the latter file.

All of this leads to one final question: when should you use global variables, and when local? Ultimately, that's a matter of personal choice for your particular program, but the general rule is: use global variables only when necessary. Try to keep data isolated inside functions, and only use a global variable when you have no other option.

There are usually ways to access the local variables from one function in another (for example, pointers in C and references in C++), but those features are specific to individual languages, so that's something to look up in your own time. Pointers are a tricky business! ■

Most local variables are automatic – that is, they are only created in RAM when program execution reaches the point of their initialisation. Once the routine containing the local variable has finished, that variable will no longer be useful, because outside routines can't access it. So, the compiler will typically reclaim memory used by that variable, making room for other bits and bobs.

There is, however, a way to stop that, and that's by declaring a static variable. This is still a local variable that can be manipulated only by code inside the current function, but it retains its state every time the function is called. For instance, look at the following C code:

```
#include <stdio.h>

void myfunc();

int main()
{
    myfunc();
    myfunc();
    myfunc();
}

void myfunc()
{
    int x = 0;
    x = x + 1;
    printf("x is %d\n", x);
}
```

Here, we call **myfunc()** three times, which creates a variable **x** containing **zero**, adds **1** to it and prints it out. You've no doubt already guessed that running this produces **1** with each call of **myfunc()**, because **x** is being created each time. But what happens if you change the declaration in **myfunc()** to **static int x = 0;**? Well, the output becomes this:

```
x is 1
x is 2
x is 3
```

The **static** keyword here tells the compiler that it should preserve the state of the variable between calls. So on the very first call it sets up **x** as **zero**, but in future calls it doesn't initialise the variable as new again, but retrieves its state from before. The compiler doesn't ditch it at the end of the function, as it does with automatic variables. So, using static variables is a great way to get some of the benefits of global variables (retaining its state throughout execution) without exposing it to the rest of the program.

By default, most C compilers don't set newly-created automatic variables to zero. This is especially true in the case of automatic variables, because they might be created somewhere in RAM that was used previously by other data.

So if you have a program like this:

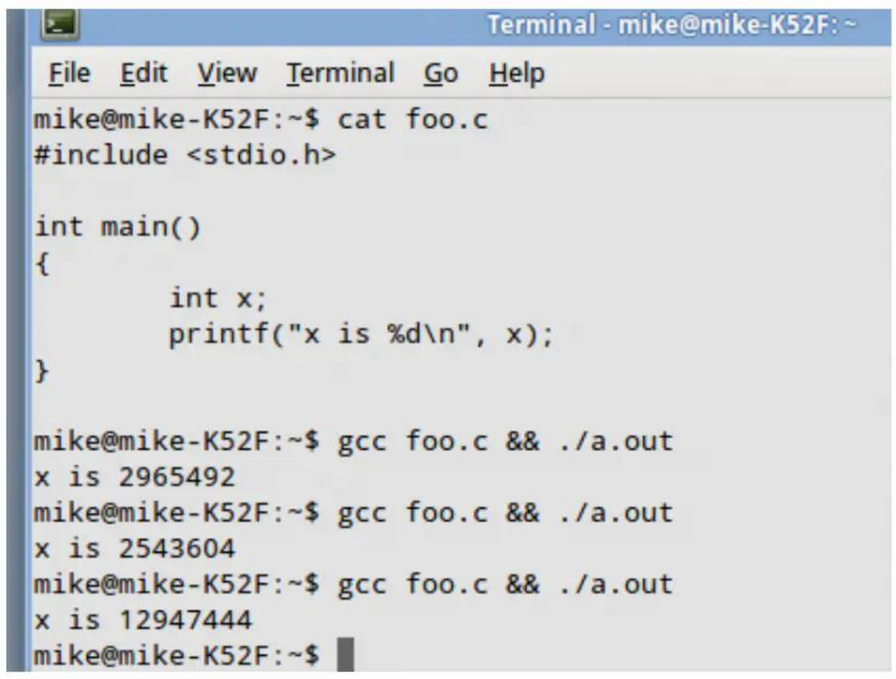
```
#include <stdio.h>

int main()
{
    int x;
    printf("x is %d\n", x);
}
```

then compile and run it, and then compile and run it again, the number will probably be different each time you do. The moral of the story is: don't blindly use variables that haven't been initialised with some kind of value!

Now, why don't compilers just set the variable to zero? That would make everything a lot safer, right? Well, yes, but it's extra work for the CPU and adds a performance hit. The compiler allocates space for variables in RAM that may have been used for other data before, so there's no guarantee that the RAM is zeroed-out.

Setting that chunk of RAM to zero requires an extra CPU instruction, and *GCC* tries to be pretty efficient. You can tell *GCC* to warn about the use of uninitialised variables with the **-Wall** flag.



› If you try to use automatic variables that haven't been given any value, prepare for some horrors

Reference

Recursion: round and round we go

Jump down the ultimate rabbit hole of programming ideas, but be careful – you may well land on your own head

Recursion is one of those concepts that sounds far more intimidating than it really is. Honestly. To make use of it, you don't need to grasp complicated mathematical ideas; or start thinking in four dimensions; or invade one level after another of someone's dreams to implant an idea into their subconscious; or even understand the meaning of recursive acronyms, such as GNU (GNU's not UNIX).

For the most part, many uses of recursion are well documented and explored. You can then just copy and paste them into your code without necessarily understanding how they work, or why they do what they do. But if you do this, you're missing out, because recursion is cool. It's one of the best ways of letting the CPU do all the hard work while you write a few lines of code; and as a result, it can save you from bending your brain around what might be a more complex problem without recursion.

This is because recursion is simply a way of solving a more complex problem by repeating a simple function. The trick is that this function repeats from within itself. Rather than being run manually from another function, it calls another instance of itself from within its own code. It's the programming equivalent of pointing two mirrors together to create a seemingly infinite corridor, or connecting to the same virtual desktop from within a virtual desktop.

X factorial

In programming, one of the classic, and easiest, examples of recursion is a function to give a number's factorial. The factorial of a positive integer (we have to say this because the function would change otherwise) is the product achieved by multiplying the whole numbers it contains against one another. The factorial of 4, for example, is **4x3x2x1**, which equals 24. If we wanted to approach this problem in the same way we've been doing with previous tutorials – a strategy known as iterative programming – we'd maybe write a function that multiplied two values within a **for** loop which was counting down through the factorial input value. In pseudo code, this might look something like the following:

```
function factorial(c)
        for i =c, i >= 1, i--
                total = total *i
        print total
```

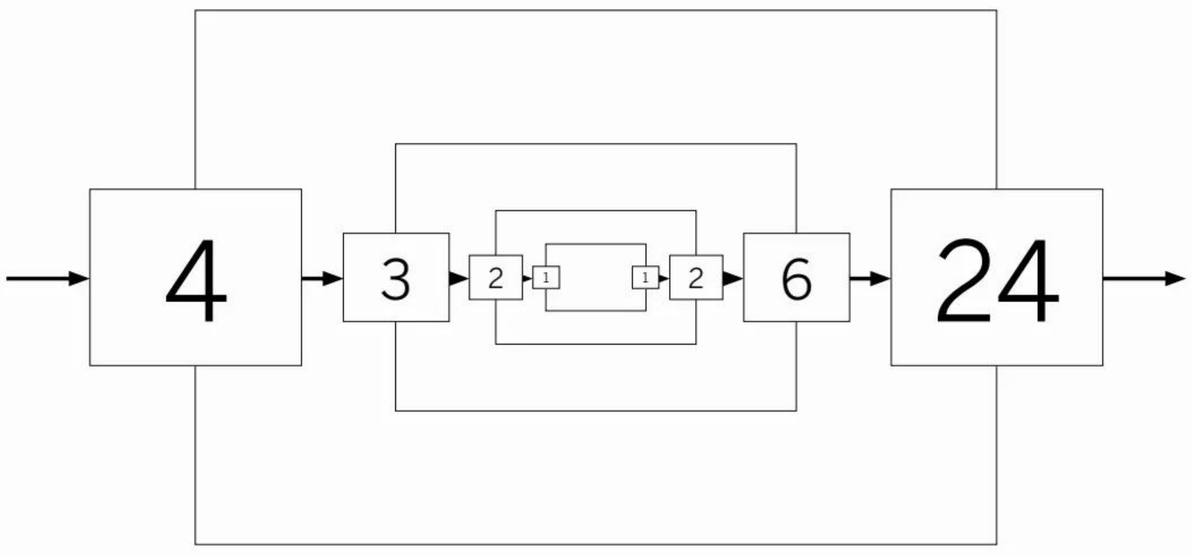
Pseudo code like this is used to explain a concept and to create a template to show the logic of a program, but the syntax isn't from any one specific language. It needs to be detailed and clear enough for the reader to understand the logic, without being weighed down by the peculiarities or eccentricities of one implementation. We've used a more C-like syntax for the **for** loop, for instance, because we think this is easier to read than the **range** keyword we'd need to use if we wrote the same code in Python.

If you're working on a problem and don't know how to approach it, writing a pseudo code sketch of any potential solution is a great way of communicating your ideas or proving your concept works. In the above snippet, we create a **for** loop that uses a variable called **i** (that's a common name for variables within a **for** loop) to step through a range of values. We've used the C-syntax to say this value needs to hold a number more than (**>**) or equal (**=**) to the value **1**, and that it should be increased by one with every iteration of the loop. In other words, the value of **i** steps down from **c**, the input value, to the value **1**. Each iteration of the multiplication is controlled by the code, which is why this is an iterative approach.

You might already see from the previous piece of code that there's a slightly more efficient way of achieving the exact same result. And that way uses recursion. Instead of controlling the function from a **for** loop, we're going to ask the function to control itself knowing that it has been called from within itself. That's confusing to read, but it's a concept much clearer in code. Here's the pseudo code for an updated factorial function that uses recursion instead:

```
function factorial(c)
        if c > 1
                return  c * factorial(c-1)
        else
                return 1
```

All that's happening here is that we've replaced the old factorial function with one that calls itself and expects a value in return. Don't forget that as well as passing values to a function, the function itself can return a value, usually using the **return** keyword, so that the code that called the function



› **To calculate the factorial of a number, we can use a function that calls itself and returns the correct answer. That's recursion!**

can use the value. You can see that while it's harder to understand the logic of how the final value is calculated, there's a lot less code than with the iterative approach, making this solution more efficient and less prone to errors. This is true of the majority of recursive solutions.

As long as the incoming value is greater than **1**, **factorial** calls itself with a new value that's one less than the one it was called with, and multiplies this by the value that's returned. If you call **factorial(4)**, for example, **factorial()** will be executed from within itself with the values **3**, **2** and then **1**. When it hits **1**, the return values start to trickle back because **factorial()** is no longer being called recursively. **1** is returned and multiplied by **2**, which itself returns **(2x1)** to the previous call, which returns **(3x2x1)** which, in turn, returns **(4x3x2x1)**. If you want to see this as Python code so you can try it out, type the following into the Python interpreter:

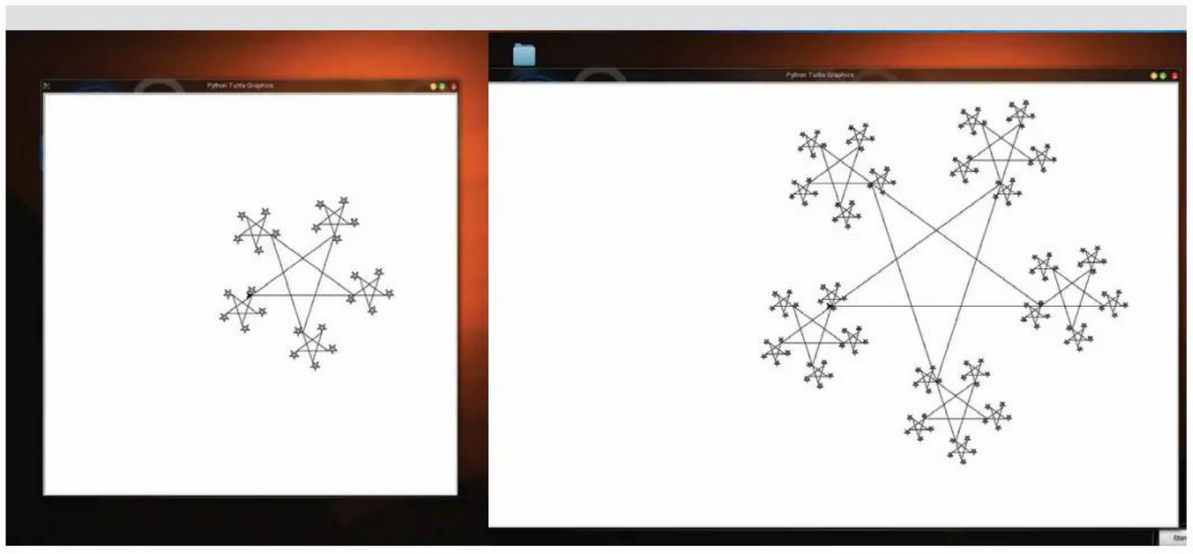
```
>>> def factorial(c):
...  if c > 1:
...   return c * factorial(c-1)
... else:
...   return 1
...
>>> factorial(4)
24
>>> factorial(5)
120
>>>
```

The last two lines in that example are simply executing the **factorial()** function with the value we want to calculate. When the function finishes and returns the final calculation, the Python interpreter prints this out without us needing to do anything else, which is why you see the results under this line. Congratulations – you've solved the problem using recursion! You should now be able to see how recursion can be used in all sorts of places that require an indeterminate amount of repetition, such as many sorting algorithms, other mathematical solutions and filesystem searches. Even if you can't think of a solution yourself, if you have an idea that something can be approached using recursion, then a simple Google search will reveal whether it can and what the code, or pseudo code, should look like.

Generating fractals

But if you want to visualise recursion in action, which is one of the quickest ways to understand the concept, there's one particularly effective set of functions that can be used to illustrate both its advantages and its complexities, and these are fractals. Fractal algorithms use recursion to add infinite levels of detail to what's a much simpler algorithm. Some of the most famous were documented by the French mathematician Benoît B Mandelbrot in his book, *The Fractal Geometry of Nature*, but there are many easier algorithms that can be created with just a few lines of code, which is what we're going to do using Python.

Before we start, though, you will need to make sure you have two dependencies installed. The first is Python's Turtle graphics module, but it should be part of any default Python installation. We use this to draw and display lines with as little code as possible. The Turtle module itself needs you to install **tk-8.5** for its drawing routines, which means if your distribution doesn't add this automatically, you'll have to install it yourself. Using the Turtle graphics module means we can also see exactly what our script is doing as it's doing it, because the cursor moves slowly along the path as our fractal is being drawn.



› We generate this fractal with just 16 lines of code. But its complexity is potentially infinite

The fractal we're going to create is called a star fractal. This is a five-pointed star, with another star drawn on to the end of each of its limbs, and a further one on each of those limbs, and so on, and on. But to start with, we're going to create a function to draw a star, then expand this to draw another star at the correct point. Drawing with Turtle graphics is perfect for fractals because it doesn't draw lines with co-ordinates, but instead uses the relative position of a cursor and an angle to draw the path with the cursor as it moves forward.

```
import turtle as trtl
def fractal(length=100):
    if length < 10:
        return
    trtl.fd(length)
    trtl.left(144)
    trtl.fd(length)
    trtl.left(144)
    trtl.fd(length)
    trtl.left(144)
    trtl.fd(length)
    trtl.left(144)
    trtl.fd(length)
    trtl.left(144)
    return
```

If you run the above piece of code, either from the interpreter, by typing **fractal(200)**, or by adding that command to the last line in a script, you'll see the Turtle cursor first move forward before turning left 144 degrees and drawing another line. It does this four more times to complete the star. To turn this into a fractal, we want to execute the same function from within the function itself at the end of each limb, and this can be easily achieved by adding **fractal(length*0.3)** after every **trtl.fd(length)** line.

This will call the fractal function again, only this time with a length value around a third smaller than the original. This star, too, will launch other iterations, but the reason why this doesn't go on for ever is because we first check whether the length of each line is going to be greater than 10. If it isn't, then we return from the function, and stars will stop being indefinitely drawn. As with the factorial function, this means that all the other part-completed stars can draw their respective limbs until the main star itself has finished and the fractal is complete. With just a few lines of code, you end up with what could be an infinitely complex image. And that's the never ending beauty of recursion. ■

Quick tip

If you want your Python application to pause before quitting, or doing anything else, add **from time import sleep** to the top of your script and use the **sleep** (seconds) command within your code. This is very useful if you want to see the final Turtle graphic rendering before the window closes.

Reference

Super sorting algorithms

Let's take our coding further and introduce you to the unruly world of algorithms, efficiency and sorted data

Python is a great language. You're happily writing a little program, and you've got a huge list that you need to put in order. What do you do? Easy, just call the **sort()** method on the list, and you're away. It works amazingly quickly, and it doesn't even care how big the list is – it could be millions of elements long and still take virtually the same amount of time to return the sorted list.

This is really handy, but have you ever stopped to wonder how on earth it works? It's a fascinating question, and in the next two articles we're going to introduce you to some of the techniques that are used to solve this kind of problem. Along the way, you'll also get a gentle introduction to algorithms and how to think about their performance.

So, where do we start? Sorting a list with a computer, like many programming problems, seems incredibly abstract and difficult when you first begin. If you start off trying to think about how to sort a Python list with a million items in it, you won't get anywhere very quickly.

Sorting two cards

But what if you step back, away from the computer and the list of a million items? Imagine, for example, that you have two hearts cards from a deck. How would you put these two cards in numerical order?

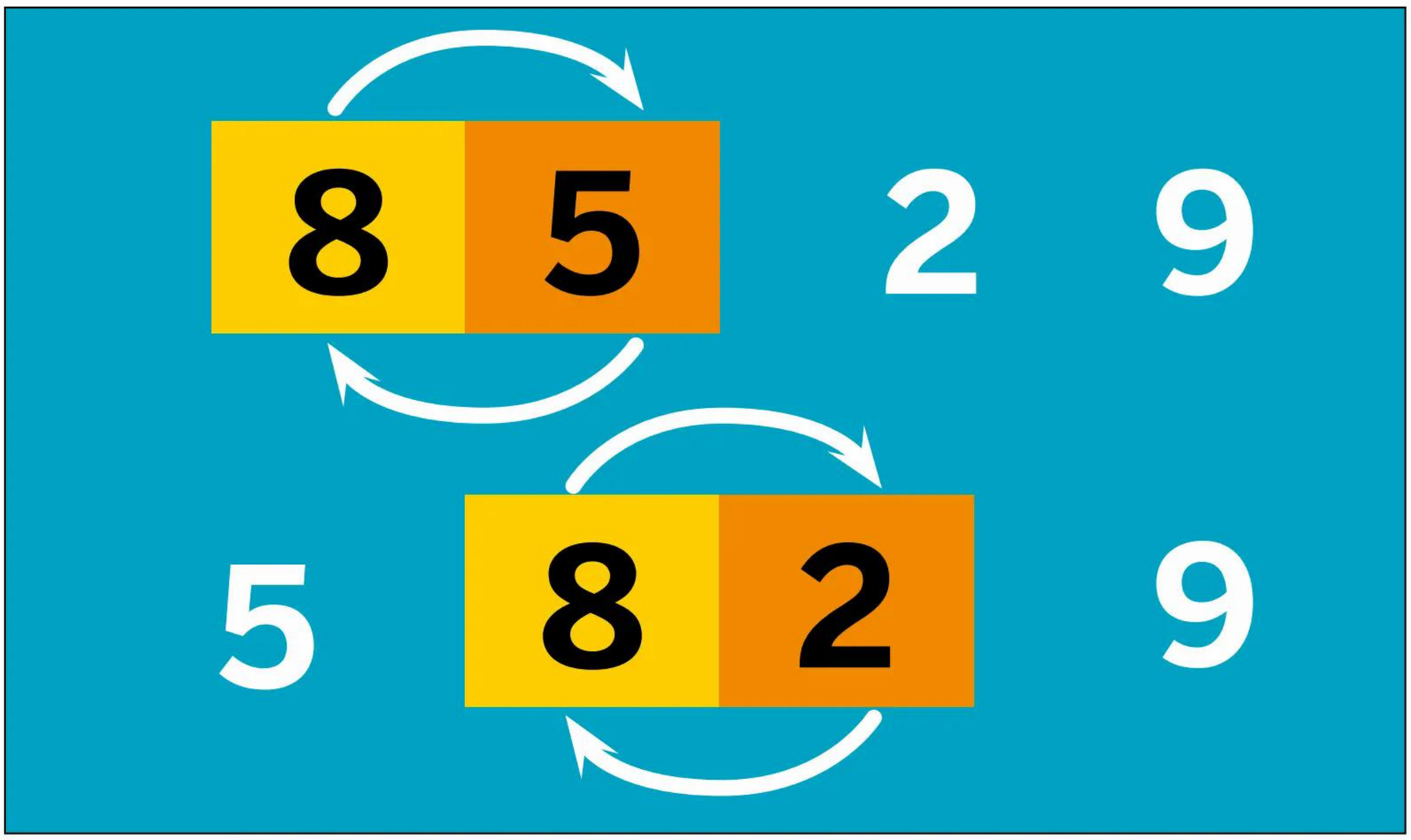
Well, it's pretty simple: you'd look at them, and if the card on the left was worth more than the card on the right, you'd switch them around; if the card on the right was worth more, you'd leave them as they were. What would that look like in Python?

```
cards = [8,2]
if card[0] > card[1]:
   card[0] = card[1]
   card[1] = card[0]
print cards
```

That seems pretty straightforward. There's a list of cards, with the values **8** and **2**. We then check to see whether the first card is more valuable than the second, and if it is, copy each card to the correct location in the list.

Run that code and see what happens. You should find that you don't get **[2,8]**, but **[2,2]**. That's obviously not what we want, so what happened?

Because there's no operator in Python that enables us to switch the position of two elements, we did what seems most natural to us – we used the assignment operator to copy the value of the two cards. The problem is that after the first **copy** statement, we ended up with both cards having the same value. When we tried to do the second assignment, we just copied two identical cards.



› Bubble sort works by comparing adjacent elements in the list. As it does so, the largest element 'bubbles' its way to the end of the list

The way to get around this is to store a copy of the first card's value outside of the list, and before we do any copying. Take a look at this code:

```
cards = [8,2]
card0 = cards[0]
if card[0] > card[1]:
   card[0] = card[2]
   card[1] = card0
print cards
```

If you run that, you should get the correct answer. Because we stored a copy of the first card's value when we overwrote it with the first assignment statement, we could use the copy to set the second card to the correct value.

In the end, this is not quite how we would do things in real life, but it's pretty close.

Bubble sort

OK, so sorting a list two elements long isn't particularly impressive. But we can extend the same technique to sort a list that has an infinite number of elements – this is known as bubble sort. The idea is that we loop over the entire list, comparing (and swapping, if necessary) each set of adjacent elements in turn. The element with the greatest value will always end up on the right-hand side of the comparison, so will always be compared in the next pair of elements. Eventually, it will end up at the very end of the list, where it belongs.

Notice that, if we were to loop over the list only once, we'd get only the largest element in the correct position. The way to get around this is to keep looping over the list; on each loop, we'll encounter another largest element that always ends up on the right-hand side of each comparison – that is, until it reaches the largest element from the previous iteration. At this point, it won't move any further, because it's now in the correct position.

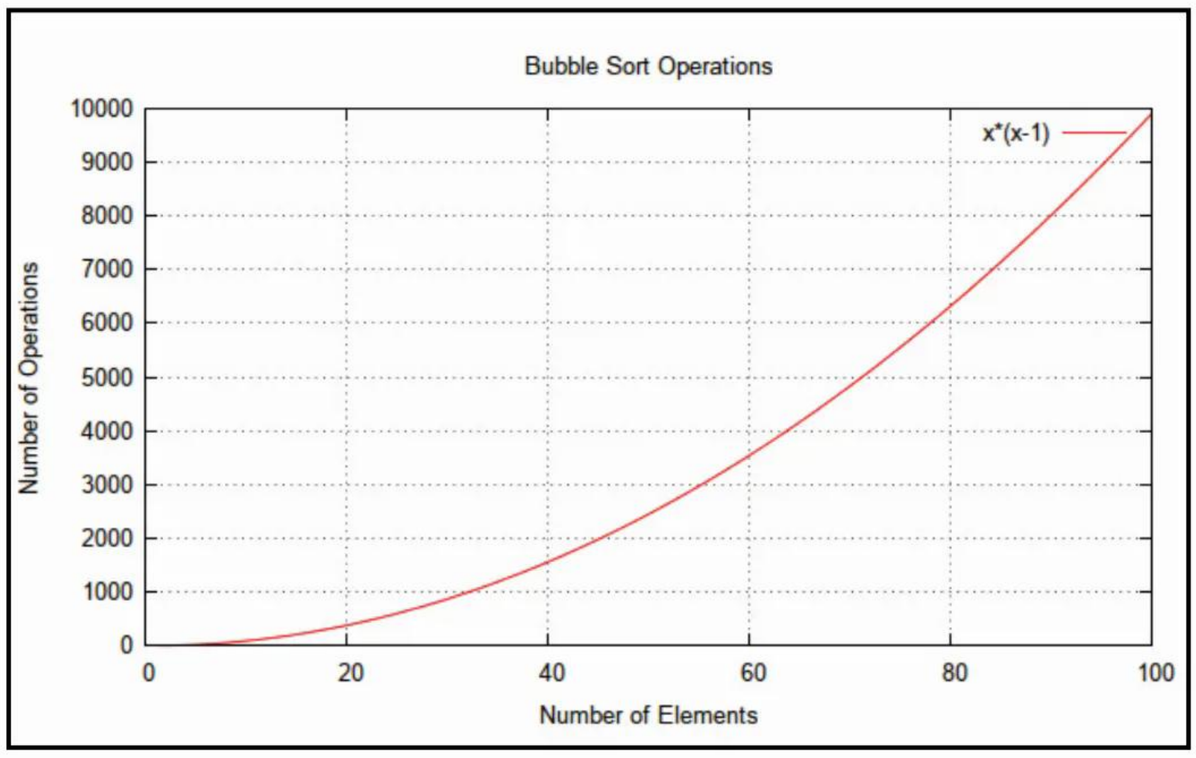
We'll know the list is sorted when we run a loop in which nothing gets swapped. This code shows how a bubble sort might be implemented in Python:

```
cards = [8, 3, 7, 4, 2, 1]
swapped = True
while swapped: #sort until swapped is False
   swapped = False #assume nothing is swapped
   for i in range(len(cards) - 1): #loop entire list
      cur = cards[i]
      j = i + 1
      if cards[i] > cards[j]:
         cards[i] = cards[j]
         cards[j] = cur
         swapped = True #reset swapped to True if anything is
swapped
print cards
```

Speed matters

It's pretty clever, right? You can now use this program to sort any size list and it will get the job done. However, if you try to use it on a large list, you'll find that it's painfully slow – even on a fast computer.

To understand why this is, let's think a little about how much work our bubble sort algorithm has to do. When we had just two cards, bubble sort had to do two comparisons – once to get the cards in the correct place, and then again to check that there's nothing else to sort. In total, it had to perform just two operations.



When we add a third card, assuming that they're in reverse order, bubble sort has to do the most work possible; it has do two comparisons on each loop, and three loops. In total – that's six operations.

What about if we add a fourth? It will take three comparisons on each loop, and four loops. That's 12 operations this time. Five cards? That's four comparisons and five loops: 20 operations. There's a definite pattern here. The work done by bubble sort seems to conform to the formula:

num. operations = num. loops x num. comparisons

And we can see that the number of loops is always equal to the number of elements in the list, and the number of comparisons is always equal to one less than the number of loops. This means that, if we have **n** elements, the amount of work done by bubble sort is:

num. operations = n x (n - 1) = n(n - 1) = n^2 - n

Big O

What this means is that the amount of work bubble sort does increases in polynomial time for every item added to the list. That is to say the increase in the amount of work becomes more and more rapid for every element added to the list. This is why it's so slow for large lists – a 10-element list may take only 90 operations, but a 1,000-element list will take 999,000 operations, and it will simply get worse for every card from there on.

In the general case given above, the **n^2** term is always going to be far larger than the **n** term, and hence will always have a much larger influence on how well the algorithm performs. Because of this, when discussing the performance of an algorithm, in computer science it's only ever the largest term that's considered.

The terminology used for this is 'big O' notation, and we say that bubble sort is a **O(n^2)** algorithm. Bubble sort is actually considered to be one of the worst-performing sorting algorithms, and it's definitely not how Python does it. That said, don't discard it entirely. It's easy to implement, and works alright for small lists on fast computers. There may be times when you just want to get the job done, and bubble sort makes the most sense here.

If you fancy looking at sorting algorithms that are much faster and is actually much closer to the way Python works, you could also try using quick sort (aka **qsort**) , however be aware that it is much harder to implement. ■

› **Bubble sort is an algorithm with O(n^2) performance – that means it takes increasingly longer for every element added**

Reference

Hidden secrets of numbers

Allow us to channel the living spirit of Dan Brown, and uncover the hidden meaning behind 1, 2 and -1b1010101

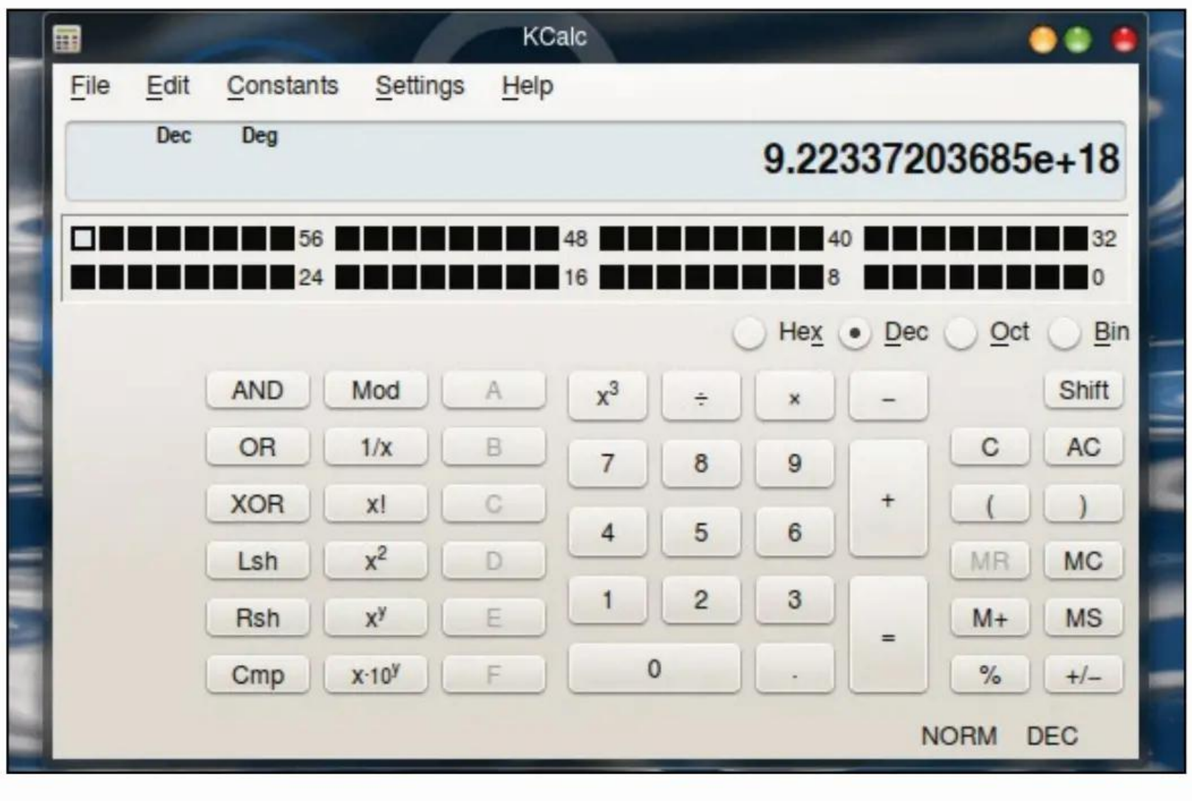
You wouldn't think that numbers would be a relevant topic in the 21st century. After all, if there's one thing computers can do well, it's count them. Billions of times a second. But for better or worse, having some knowledge of how computers deal with numbers, and the systems used to process them, will help you to write better programs. And it's not all for low-level stuff like assembler either. Number systems are just as important for high-level languages. If you write a PHP script for the web, for example, small changes in the way you process numbers within the script can have a huge impact on the performance of your servers. Your code might be run thousands of times a second, making any small discrepancy in how those numbers are calculated or stored have an impact on performance.

"The position of each bit within a binary value represents a number that's double the bit to the right"

Numbers are important because they engage the CPU at its lowest level, right down to hardware and design. Typically, a single operation of the CPU can deal only with a value held within one of its internal registers. And that value can be as large only as the number of bits the CPU supports. These days, that means either 32 or 64 bits. But what's a bit? It's literally a switch that's turned on or off – a single bit of information, and this is normally represented by a **1** for **on** and a **0** for **off**. Computers use a numeral system called binary to store these on/off values within their registers and memory. The position of each bit within a binary value represents a number that's double the bit to the right of it, giving a sequence of bits the ability to represent every value between zero and double the value of the most significant bit minus 1 (or 2^8**-1**). That's a difficult couple of sentences to digest, so here are some examples:

The positions within an 8-bit binary number have the following values: **1**, **2**, **4**, **8**, **16**, **32**, **64** and **128**. But rather than assign them in that order, the most significant bit – the one holding the largest value – is usually on the left. So, the binary number **10**, or **00000010**, is equivalent to **2**. If we move that bit to the left so that the **1** occupies the third slot along, the total value doubles to **4**, and so on. If we enable more than one bit at a time, then all the values are added together, so that **00000011** represents **3** (**1+2**) and **10000011** represents **131** (**128+2+1**). If we enable every bit in an 8-bit binary number, we get a maximum value of **255**.

› **If you want to see binary registers update in real time, KCalc has a handy display mode for up to 64 bits of data**



2's complement

Consider the following question: what does **10101010** represent in 2's complement? You might think, after the last couple of paragraphs, that the binary value represented here is **170** (**2+8+32+128**). But you'd be wrong. The clue is the reference to 2's complement. This is, in fact, a scheme for representing negative values in binary, and it changes the meaning of the bits and what they represent. It does this by reversing the configuration of bits, so that **1** becomes **0**, a process known as flipping, and adding **1** (there is a single exception to this).

So, the negative of **00000010** is first **11111101**, then with **1** added **11111110**, which is the 2's complement representation of **-2**. Using this logic with the above question, we first subtract **1** from **10101010** to get **10101001** and flip the bits to **01010110**, which gives a decimal value of **86**, so the answer to the question is **-86**. The reason this method was chosen rather than swapping the most significant bit is because most binary arithmetic can still be performed on these negative values in exactly the same way they're performed on positive values. Adding **1** to **10101010** (**-86**), say, yields **10101011**, which is **-85** – the correct value.

While we're in the binary zone, it's worth mentioning a couple of other numeral systems sometimes used when programming. The first is octal, a base 8 system that uses the numerals 0-7 to hold 8 values. **7+1**, for example, is **10** in octal,

just as **9+1=10** in decimal. It takes exactly three binary bits to represent a value in octal, which is why it's often used when you play with binary values. Similarly, hexadecimal, which is base 16, uses the characters 0-9 and a-f to represent decimal values 0-15, and that's the equivalent of four bits of data. So, the number **6c** in hex is **0110** (**6**) and **1100** (**12 = c**) in binary, which as an 8-bit value equates to **108**. Hex is a more readable equivalent to binary, which is why it's often used for memory and binary editors, and for storing raw binary data within your program.

You can avoid using these numeral systems in your code and your projects won't suffer. But understanding a little about what they mean and how they relate to one another can help you when looking at other projects, and how the underlying systems that take your code and make it run work. If you want to play with numeral systems, you can do so using Python's interpreter, and this is a great way of getting familiar with how the numbers work. From version 2.6 onwards, for example, you can enter a binary value using the **0b** prefix – that's a zero followed by the **b**. Typing a binary number will output the decimal value:

```
>>> 0b01010110
86
```

You can convert values to binary using the **bin** function:

```
>>> bin(86)
'0b1010110'
>>> bin(-85)
'-0b1010101'
```

As you can see with the last example, this also works with negative integers, returning a 2's complement binary. If you want to input a 2's complement binary, you need to prefix the data with **-0b**. You can do similar things with octal by replacing the **0b** with **0c** and with hexadecimal by using **0x**, and both of these numeral types have supporting functions for converting integers:

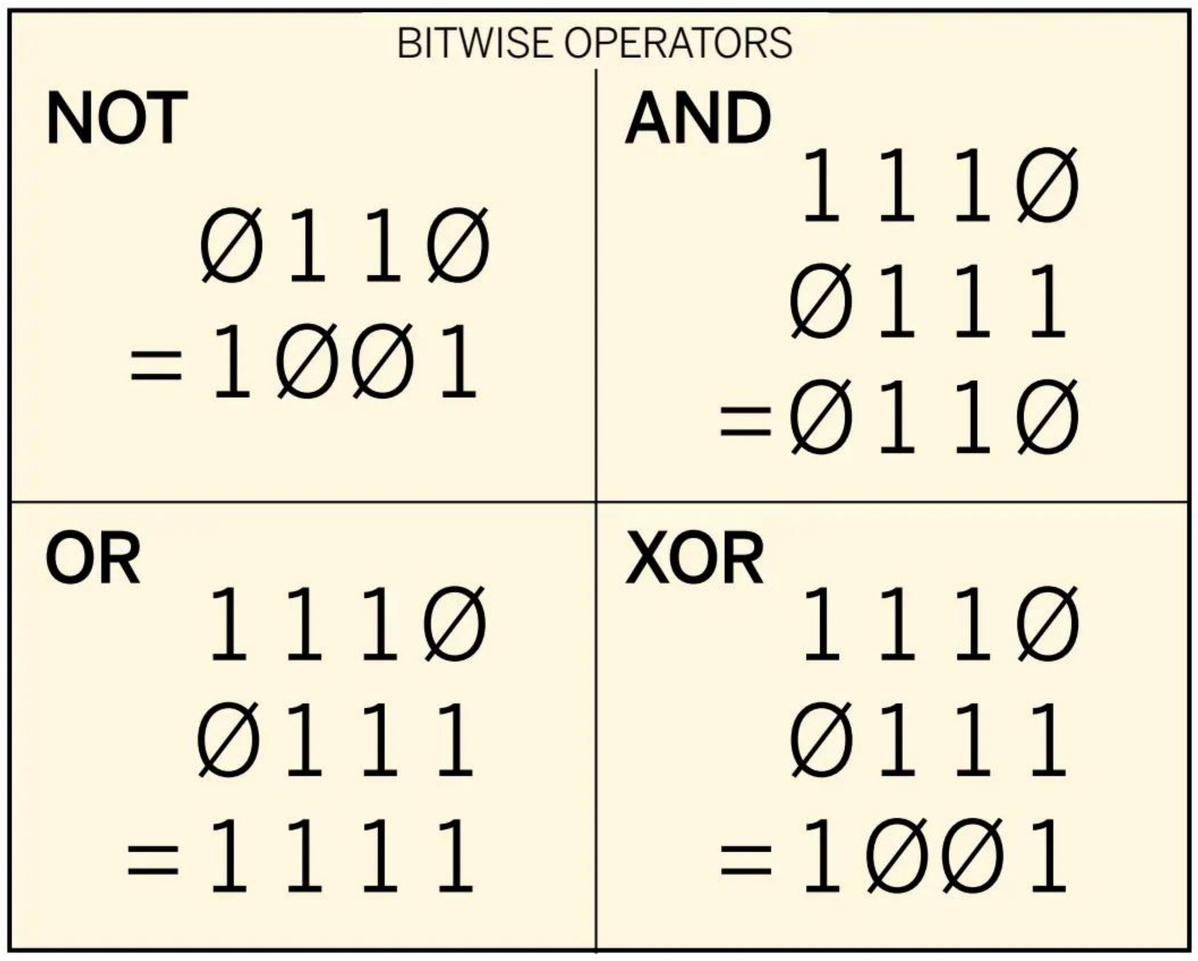
```
>>> hex(85)
'0x55'
>>> oct(85)
'0o125'
```

Data types

In many programming languages, you need to specify the type of a variable before you can use it. This can cause confusion for the beginner, because they wonder how they're supposed to know what they need to use before they write the code. But, in reality, you write the declaration for each variable as you come to use them.

The compiler or interpreter needs to know about type because it wants to assign only enough memory and processing to handle that number, and no more. If you know your variable is never going to have a value greater than 255 (the value held by an 8-bit value), there's no need for the compiler to ensure more than this amount of storage is available. An 8-bit value has become known as a byte. Although the number of bits in a byte used to vary, it has settled on 8, because this is typically used to store a single character. A byte's worth of data is still used to store characters to this day, which you can view through ASCII codes with any reasonable binary editor.

But computer hardware has moved on, through the 16/32-bit era of the Atari ST and Amiga, to the pure 32-bit PCs of the last decade and to today, where most machines have a processor capable of handling 64-bit instructions. For that reason, like early bytes, there's no standard storage size for many data types. It always depends on your computer's hardware and its internal representation of a value as bits. The best example is an ordinary integer, which is most often defined as **int** in programming languages such as C and C++. The largest number an **int** can hold depends on the architecture of your CPU, and this limit is hidden within something called an **include** file for C and C++, and an equivalent for other languages. On most machines, you'll find that an unsigned **int** can hold a maximum value of **4294967295**, and if you paste this value into a binary calculator, you'll see this requires 32 bits of storage – probably the most common architecture in use. Even 64-bit machines can properly interpret 32-bit values.



› **Binary and bitwise operators bring logic to your programs**

This is why an **int** is usually the most generic variable type; it can store large numbers and can be addressed efficiently by most recent x86 CPUs. If you need larger, there's **long int** and if you need smaller, there's **short int**. You generally don't need to worry about any of this in languages such as Python, unless you want to, because the designers have decided to forgo the advantages of specifying the size of a variable for the simplicity it gives to programming. This is an example of something known as Duck typing, and it's another reason why Python is such a good beginner's language.

One set of types we've neglected to cover is those that include a floating point, such as **0.5** or **3.14159**. Rather than worry about the size of a float, it's the precision that's important. But unless you're dealing with small fractions, it's usually enough just to create a variable of type float to deal with these numbers.

The biggest problems when dealing with floating point numbers, and why programmers try to stick to integers, are introduced by rounding a value up or down from a more precise number, such as when rounding the value of pi to 3. While small rounding errors aren't likely to cause problems, they do when many of these are combined. This problem, and its dire consequences, was brilliantly illustrated by Richard Pryor's character in *Superman III* when he siphoned off the rounding errors in employees' payslips into his own account, and amassed a large fortune. ■

Quick tip

Understanding how bits and bytes hold together is essential if you want to read or write to binary formats, or create a utility that communicates with external hardware.

Reference

Using loops and using loops

If we have explained how to use loops adequately, move to the next page. Otherwise, try reading this again

After losing ourselves in the surprisingly complex domain of using numbers in our code, we're returning to a simpler concept. It's the idea of a loop, and how programmers use them to solve problems. A loop, in the programming sense, is a chunk of the same code that's designed by the programmer to be run over and over again. It's different from a function or a method, because while they're both designed to be re-run over and over again too, they're self-contained islands of logic.

With a method or a function, the programmer needs to know only what input is needed and what output to expect. Everything else should be handled by the function. A loop is a much more primitive construction, and without loops almost any programming task would become impossibly tedious. This is because they encapsulate what computers do well: repetition and iteration. Without loops, programmers are forced to write everything in longhand, and write code that can't accommodate unknown values or sizes. It's the difference between sowing a field by hand or by using Jethro Tull's horse-drawn seed drill.

> **"Without loops, almost any programming task would become impossibly tedious"**

Loops are so integral to code, we've already used several of them in our examples in previous tutorials in this bookazine. They're used as counters, as ways to step through data, for when you need to wait for a condition, and for filling arrays and files. But we've not spent any time discussing the various kinds of loops you can implement, or how you might attempt to solve a linear problem by adding a loop in your code. In many ways, they're related to the idea of recursion. But where a loop or function in recursion calls another instance of itself, usually to build a more complex solution, a loop on its own is used mostly to solve a simple calculation, or for waiting for a specific condition. Which is why, perhaps, the most common loop combines a conditional statement within a loop's structure – and that's the **for** keyword. The **for** loop is one of those peculiar parts of any language, because it requires a specific syntax that doesn't feel that logical. In the case of Python, a typical **for** loop looks like the following:

```
>>> for i in range (5):
...     print (i)
...
0
1
2
3
4
```

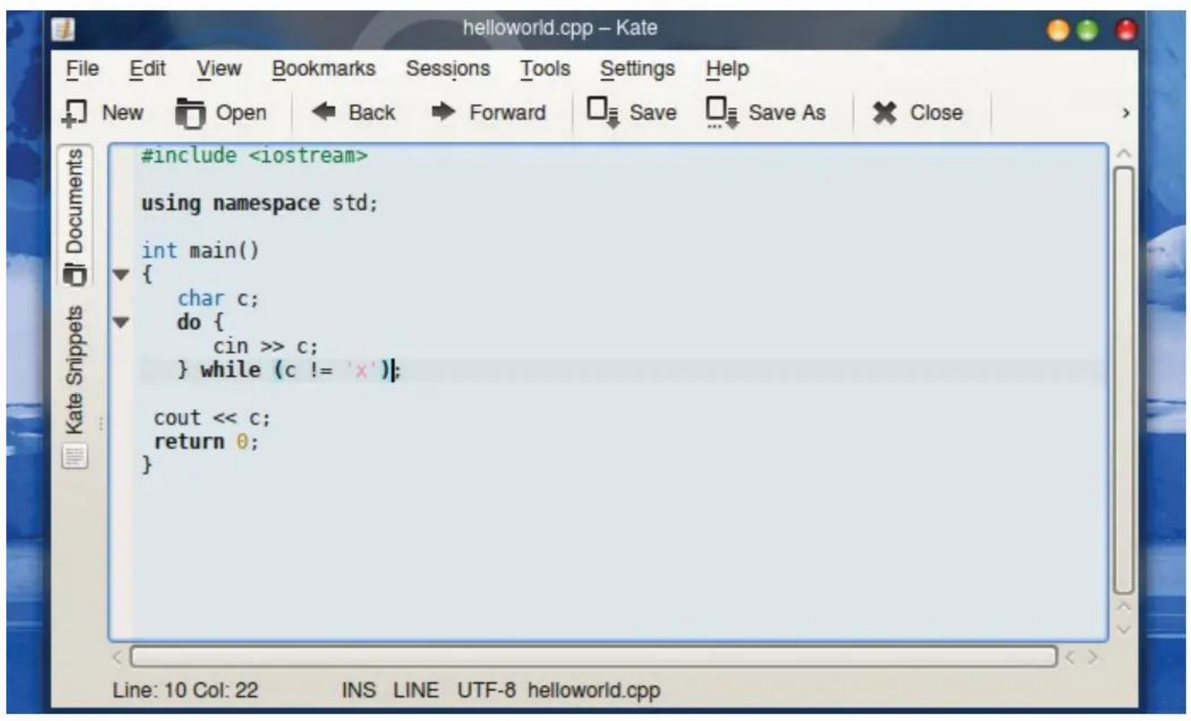
You can see in this snippet that the initial **for** statement requires a variable, which for some reason is nearly always called **i** in examples. Ours is no exception. This is followed in Python by the phrase **in range (5)**. **Range** is a special keyword in Python that we'll revisit in a couple of paragraphs. On the following line, we print the value of **i**. This line needs to be tabbed or spaced in because it's going to be executed as part of the for loop (as denoted by the **:**).

But without prior experience, you can't really guess at the output from the syntax of the **for** loop, and it's the same with other languages. This is because **for** is slightly different from most loops, because it includes the implicit definition of a variable. You just have to accept that whatever syntax they do use does the equivalent of defining a variable and creating an acceptable set of circumstances for the execution of your program to leave the loop. In our previous example, the variable **i** iterates between 0 and 4, leaving the loop when it gets to 5 (a total of five steps when you include zero). We print the value of **i** within the loop so you can see what's happening. If you write the same thing in C or C++, the syntax looks slightly different, but it gives you a better insight into what's happening:

```
for ( int i=0 ; i < 5 ; i++ ){
  cout << i << endl;
}
```

When compiled, linked and run (this is the big difference

› If you want to build your own C++ application, add the source code to a text file and build it with the command 'g++ helloworld.cpp -o helloworld'. Just run './helloworld' to see the results



between compiled languages such as C/C++ and interpreted languages such as Python and JavaScript), the output is identical to our Python example from earlier. In the C++ code, the variable **i** is declared as an integer (**int**) and incremented (**i++**) one value at a time for as long as it remains less than 5 (**<5**). This is exactly what the Python loop is doing, and the **cout** line is simply printing out the value of **i** during each iteration. You should be able to see from this that both the value of **i** and the value checked for by the condition can be changed, so you could get the value of **i** to count down instead by changing the loop to:

```
for ( int i=5 ; i > 0 ; i--){
  cout << i << endl;
}
```

To do the same in Python, we need to take a closer look at the **range** keyword used in the first example. **Range** is a function in Python that takes a start point, a stop point and a step size, much the same as the inputs for a C++ **for** loop. It's also much more versatile, as you don't need to restrict yourself to numbers – **range** can use indices of a sequence. If you put on one value in the range, as we did in the original loop, the function counts from zero up to the value **-1**. However, anything else will create a different effect. The advantage of farming it out to a function is that you're no longer restricted to using it in just the **for** loop, and you'll find many programmers use **range** for convenience throughout their code. If you wanted to count down in Python, for example, you could modify the **for** loop, as follows:

```
for i in range(5, 0, -1):
   print (i)
```

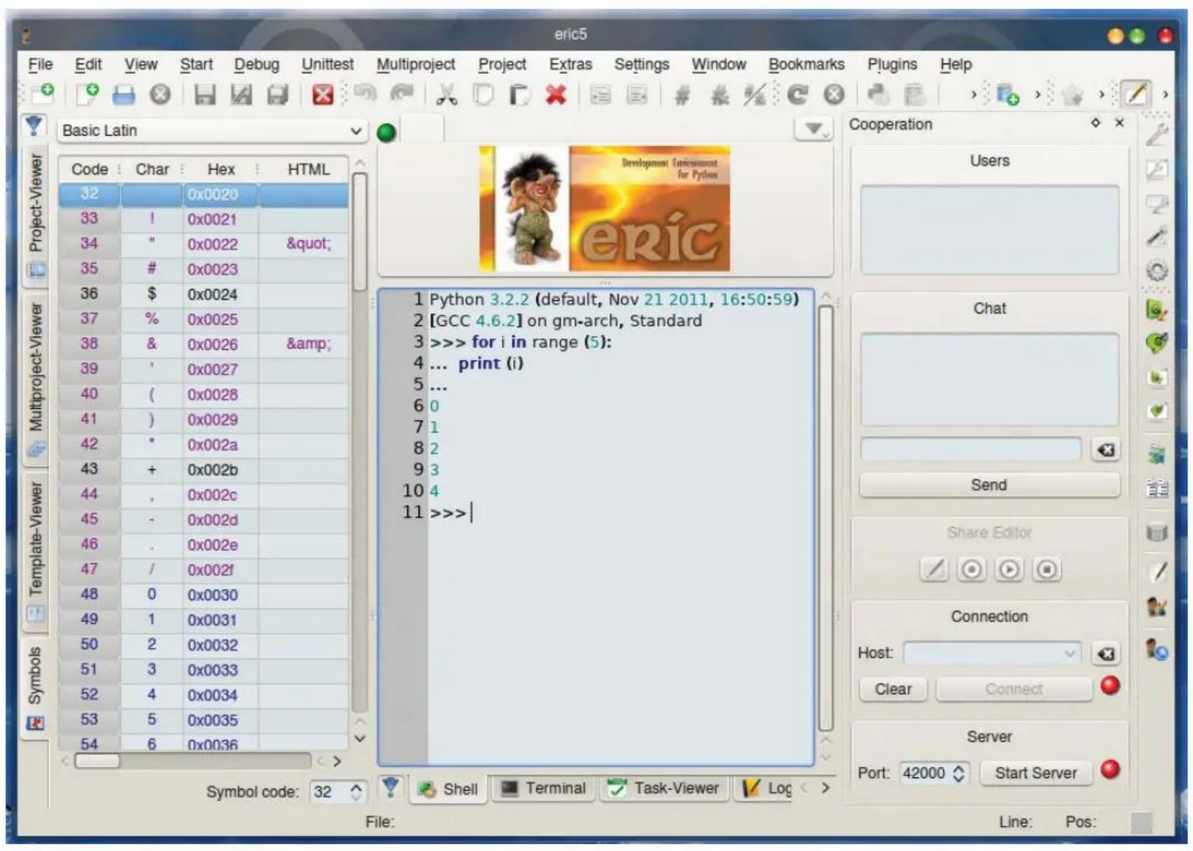
Infinite loop

If you don't need the iteration of a **for** loop, the best alternative is a **while** loop. This includes only the conditional check, and you're free to make that check positive in any way you choose. You could easily emulate the behaviour of a **for** loop, for example, by making the **while** loop wait for a variable to reach a certain value. Then, within your block of code, make sure your calculations attain that value at some point. In Python, we could recreate the previous **for** loop using **while**, like this:

```
>>> i = 5
>>> while i > 0:
... i -= 1
... print (i)
...
4
3
2
1
0
```

It's simpler this way, and by using **while** you can make the conditional checks for leaving a loop as complex as you need, rather than adhering to those required by **for**. Another trick in other languages, but not Python, is to place the conditional checking at the end of the block of code. This is normally called a **do...while** loop, because the **do** is used to start the section and the **while** is used to close the section.

Most importantly, the **while** statement at the end of the loop is checking the condition, such as 'do the washing up while there are still some dirty plates'. This can be helpful in certain circumstances, such as when you want the loop to take some input and then only test against that input at the end of the test. In C++, this function would look like this:



› Python is still a great language to learn with because it lets you experiment and see results as soon as you've typed your code

```
char c;
do {
  cin >> c;
} while (c != 'x');
```

This example is oversimplified, and isn't useful in itself, but it illustrates one example where **do...while** works well, and that's taking input. If you want to check the state of some input, whether it's from a file or from someone typing on the keyboard, you need to have first grabbed the input. You could do this in a normal **while** loop by asking for the input before you enter the loop, but it's slightly more efficient and elegant if you use a **do...while** loop to encapsulate the entire function. This is all the above loop is doing in C++. It grabs some input from the keyboard using **cin**, and waits for that input to consist only of the **x** character, after which it quits the loop. Another common use for **while** is to create what's known as an infinite loop – simply a loop that never satisfies its exit condition. This might sound like insanity if you ever want the processor to skip over the remainder of your code, but there are certain times, and languages, where the use of an infinite loop is the best way of building your application. One example is if you want to build a tool to monitor the state of a server or some other service.

You also find infinite loops watching hardware inputs or controlling the rendering within a game engine. Using an infinite loop means you can be sure your application is running, and hasn't exited under some other condition, and using **while** is probably the easiest way of creating one. **while (true) {}** is all that's needed. However, many languages also provide an escape from loops like this, and that's using something called the **break** command. **break** can be used to escape from an infinite loop, as well as leaving other kinds of loops early if you need to. You might want to escape from a **for** loop quickly, for example if you have detected that the user wants to quit the application. Using **break**, the execution of your code continues after the loop section, and you'd normally use this section of code to tidy up files, processes and memory before an exit. Of course, there's a good case for arguing against infinite loops, and building into your code the ability to escape on a specific condition, but this can take more effort and might not be as efficient, especially for smaller applications. ■

Quick tip

Nearly all languages count zero as a value, such as the first position in an array. This seems illogical at first, because we're used to thinking of zero as nothing, but it's just something you have to get used to.

Reference

The magic of compilers

A compiler turns human-readable code into machine code, but there's a difference between compiled and interpreted languages

Computers aren't actually very smart. Sure, they can do a million things in the space of a second, and help us to transmit videos of cats around the internet, but they don't understand human languages. Ultimately, everything in a computer boils down to binary – ones and zeros – because that's all that a CPU understands. For us programmers, this makes things a bit complicated. We can't directly tell the CPU in English that we want to print a message on the screen, wait for a key to be pressed and so forth; we have to somehow tell the machine this information in binary. But then, imagine if your programs looked like this:

```
10010010 11100011
01011011 10001101
...
```

That would be a nightmare, wouldn't it? Finding mistakes would be nigh-on impossible, and if you gave your code to someone else for modification... Well, the entire concept of free/open source software falls apart. So, to fix this problem, we use compiled and interpreted languages (more on the latter later). In a compiled language, human-readable source code is processed, pulled apart, jumbled up and eventually converted into binary code that the CPU can understand. The program that does this is called – naturally enough – a compiler, and it's one of the most important components of an operating system.

> **"In a compiled language, source code is processed, pulled apart and converted into binary code"**

Compilers exist for many different programming languages, but here we're going to focus on C, because it's the language used for many important projects, such as the Linux kernel. By far the most common compiler on Linux is **GCC**, from the GNU project, and that is what we are going to use here.

Dissecting a program

First up, consider this simple C program:

```
#include <stdio.h>
int main()
{
        puts("Hello, world!");
}
```

This simply prints the message "Hello world!" to the screen. If you've never seen C before, you might be a bit puzzled by some of the text, so here's a brief explanation: the **#include** line says that we want to use standard input and output routines (**stdio**). Then **main** says that this is the main part of our program (that is, where execution should begin), and the **puts** command means "put a string".

Enter this text into a file called **foo.c** and then run the following commands:

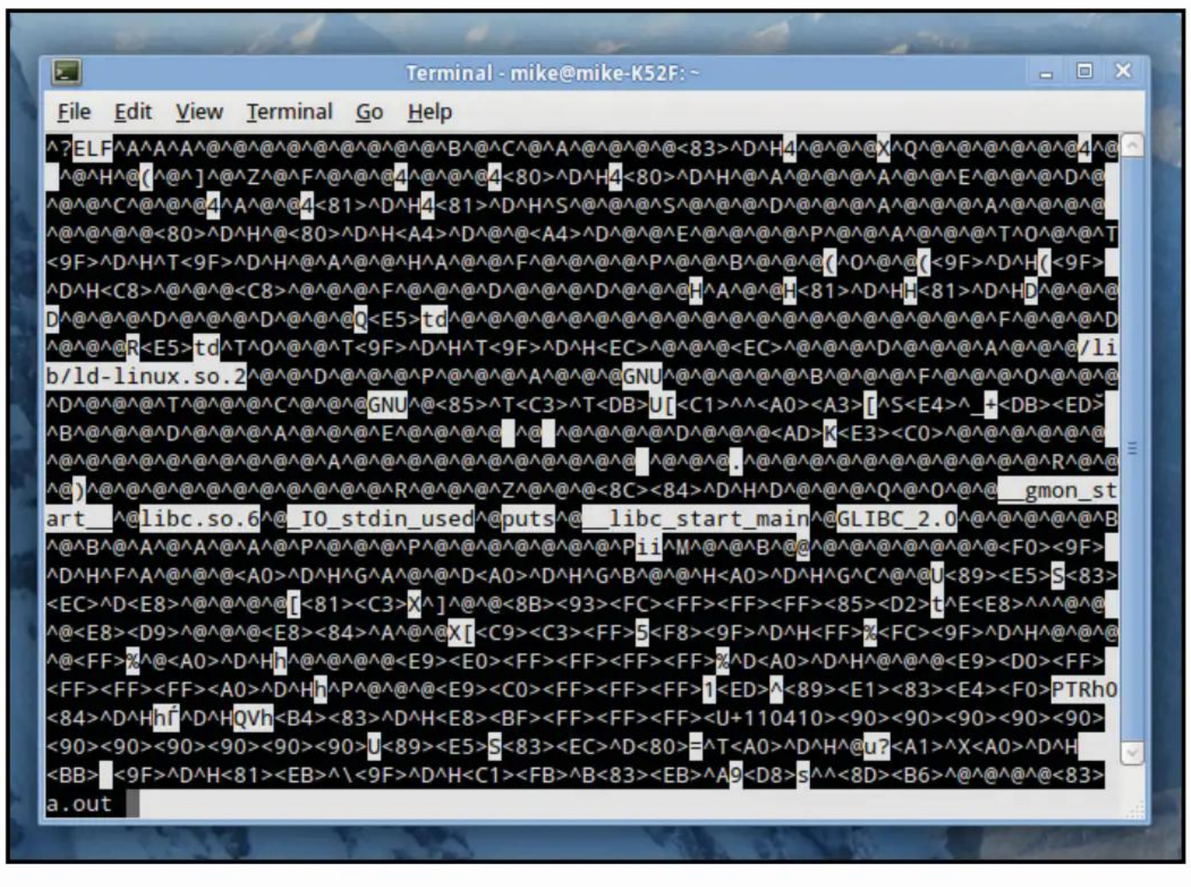
```
gcc foo.c
./a.out
```

The first command uses **GCC** to compile **foo.c** from the source code into a binary executable file called **a.out**, and the second line runs it. You can get more information about the resulting file by running **file a.out** – you'll see output similar to the following:

```
a.out: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.15, not stripped
```

If you try to examine the file yourself, using **less a.out** for instance, then you'll just see gobbledygook, as in the screenshot shown on the left. This is because we mere mortals can't read binary, and so **less** is trying to convert it into plain text characters.

This final, resulting file isn't intended for human consumption, but we can peek into some of the stages in the compilation process. First of all, enter this command:



› **Try to read a binary executable file in a text editor, and you'll just see a load of messed-up characters like this**

```
gcc -E foo.c > foo.p
```

With the **-E** switch, we tell **GCC** that we don't want a binary executable file just yet; instead, we want only to see what happens after the first stage of processing. Open **foo.p** in a text editor and you'll see that it's much longer than the original program, because the contents of the **stdio.h** file have been included in it. It's only right at the bottom that you can see the **main** function and our **puts** instruction.

In larger, more complicated programs, the compiler does much more work in this 'pre-processor' stage. It expands macros, handles **#define** statements and places additional information into the file for debugging purposes. This file is then complete, with all the information that **GCC** needs to start converting it into machine code.

Assembly

But wait! There's an intermediate step between the C code and binary, and it's called assembly language. Assembly is a human-readable way of representing the instructions that the CPU executes. It's much lower-level than typical languages such as C and Python, where you can simply say "print a string for me". In assembly, you have to move memory around into the video buffer – for instance, telling the CPU exactly which instructions it should execute. So, now enter this:

```
gcc -S foo.c
```

This produces a file called **foo.s** that contains the assembly language version of our program. In other words, it's a list of CPU instructions written in a slightly more readable form. Most of it will look like complete gibberish if you have never touched assembly before, but these are the two most important lines:

```
movl   $.LC0, (%esp)
call   puts
```

Essentially, this places the location of our "Hello world!" text string into a register (a little like a variable), and then calls the **puts** (**put string**) function from the C library. These two lines are exact CPU instructions – as programmers, there is no way that we can break them down into smaller pieces. Given that modern CPUs execute billions of instructions per second, it can be quite humbling to see single instructions laid bare like this.

In the normal process of compilation (that is, without the **-E** or **-S** switches), **GCC** now runs an assembler to convert those human-readable instructions into their binary equivalents. It then adds extra information to the resulting file (**a.out**) to make it a valid Linux executable, and we have a program that's ready to run.

Jump back!

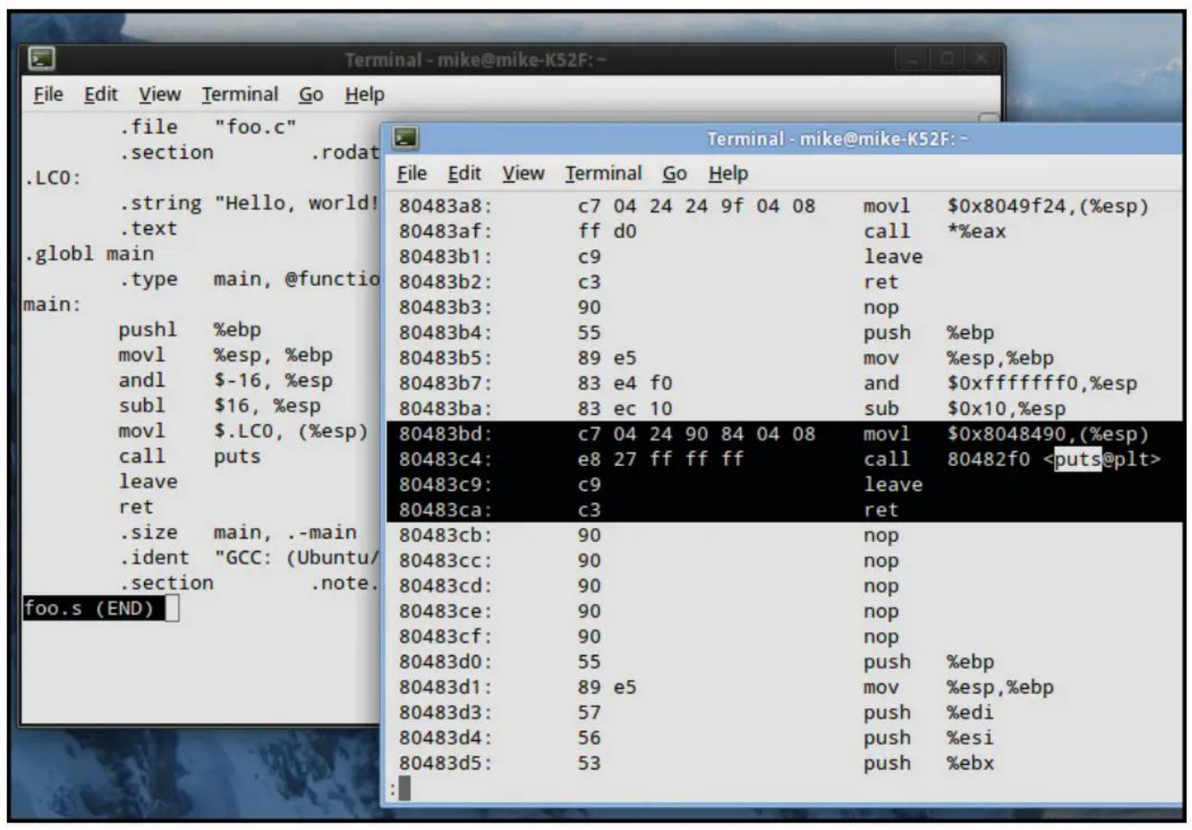
We can take the **a.out** file and go one step backwards in the process, to the assembly language stage, using the **objdump** command, like so:

```
objdump -d a.out > list
```

Have a look at the **list** file now – it's a disassembly of the binary file, and therefore full of assembly language instructions. Most of them are unrelated to our program, and are used to tell the Linux program loader some useful things and set up the environment. But tucked away in there, you'll find these lines:

```
movl $0x8048490,(%esp)
call 80482f0 <puts@plt>
```

Those are the exact same instructions as the ones we saw a moment ago, expressed in a slightly different way. So, that's the process of how human-readable source code is converted into CPU-understandable instructions. The inner workings of



› On the left, the output from gcc -S, and on the right a disassembled binary. We've highlighted the areas showing identical instructions

a compiler are deep and complex, and many would argue that writing a fully-featured compiler for a complicated language such as C++ is an even harder job than writing an operating system. Think of all the work a compiler has to do in terms of managing variables, parsing complicated instructions, passing control around to different functions, and so on.

Don't think for a second that we've completely covered the compiler here. Shelves full of whopping great books have been written about the science of compilers – it's not a subject for the faint-hearted! But now you know the fundamentals, and you will certainly look at your code in a different way from this point on. ■

What are interpreted languages?

There are two main types of programming language: compiled and interpreted. As we've seen in the former, a program called a compiler takes an entire source code file and converts it into CPU binary instructions in one fell swoop. In contrast, an interpreter reads individual lines from a source code file and acts on them one at a time, essentially interpreting each instruction at run time. For instance, take this BASIC program:

```
10 LET A = 1
20 PRINT "Hello world"
30 LET A = A + 1
40 IF A < 6 THEN GOTO 20
50 END
```

It's pretty clear what this does – it prints a message five times on the screen. Now, how would a compiler look at it? Because a compiler processes the source code file as a whole, it sees the bigger picture, knowing in advance that the loop is going to be executed five times. Consequently, the compiler can make various optimisations, such as replacing the whole bunch with five **PRINT** lines if that makes execution quicker. Also, the compiler can spot errors that might come up and warn the programmer about them.

An interpreter, on the other hand, is a much simpler program that simply steps through the program line-by-line. It doesn't know what's coming up, and just dutifully executes things as and when it sees them. This offers some advantages, in that you don't have to wait for a compilation process to finish before testing your work; and many interpreters allow you to pause them and make changes in the middle of execution.

Ultimately, most programmers prefer compiled languages over interpreted ones because of the performance benefits. Having your whole work condensed down into CPU-readable binary makes for better performance than a background program parsing each line step by step during execution. Interpreted languages can, however, be useful where speed isn't crucially important, for example the Bash scripting language. Precision and individual execution rule here.

Avoid common coding mistakes

Bug reports are useful, but you don't really want to cause too many. Here's what to avoid and how to avoid it

It doesn't matter how much care you put into writing your code. Even if you've had four cups of coffee and triple-check every line you write, sooner or later you are going to make a mistake. It might be as simple as a typo – a missing bracket or the wrong number, or it could be as complex as broken logic, memory problems or just inefficient code. Either way, the results will always be the same – at some point, your program won't do what you wanted it to. This might mean it crashes and dumps the user back to the command line. But it could also mean a subtle rounding error in your tax returns that prompts the Inland Revenue to send you a tax bill for millions of pounds, forcing you to sell your home and declare yourself bankrupt.

Finding the problems

How quickly your mistakes are detected and rectified is dependent on how complex the problem is, and your skills in the delicate art of troubleshooting. For instance, even though our examples of code from previous tutorials stretch to no more than 10 lines, you've probably needed to debug them as you've transferred them from these pages to the Python interpreter. When your applications grow more complex than just a few lines or functions, you can spend more time hunting down problems than you do coding. Which is why before you worry about debugging, you should follow a few simple rules while writing your code.

The first is that, while you can't always plan what you're going to write or how you're going to solve a specific problem, you should always go back and clean up whatever code you end up with. This is because it's likely you'll have used now-redundant variables and bolted on functionality into illogical places. Going back and cleaning up these areas makes the code easier to maintain and easier to understand. And making your project as easy to understand as possible becomes important as it starts to grow, and you seldom revisit these old bits of code.

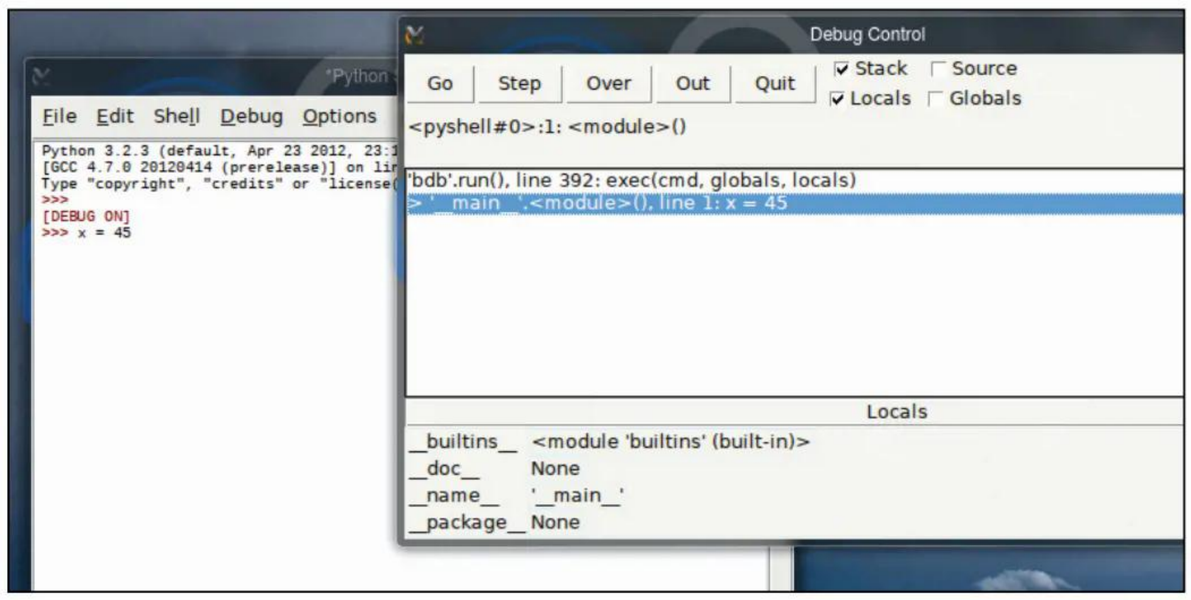
Whenever you write a decent chunk of functionality, the second thing you should do is add a few comments to describe what it does and how it does it. Comments are simple text descriptions about what your code is doing, usually including any inputs and expected output. They're not interpreted by the language or the compiler – they don't affect how your code works, they are there purely to help other developers and users understand what a piece of code does. But, more importantly, they are there to remind you of what your own code does.

This might sound strange, but no matter how clear your insight might have been when you wrote it, give it a few days, weeks or months, and it may as well have been written by someone else for all the sense it now makes. And as a programmer, one of the most frustrating things you have to do is solve a difficult problem twice – once when you create the code, and again when you want to modify it but don't understand how it works. A line or two of simple description can save you days of trying to work out what a function calculates and how it works, or may even obviate the need for you to understand anything about what a piece of code does, as you need to know only the inputs and outputs.

The importance of documentation

This is exactly how external libraries and APIs work. When you install *Qt*, for instance, you're not expected to understand how a specific function works. You need only to study the documentation of the interface and how to use it within the context of your own code. Everything a programmer needs to know should be included in the documentation. If you want to use *Qt*'s excellent sorting algorithms, for example, you don't have to know how it manages to be so efficient, you need to know only what to send to the function and how to get the results back.

You should model your own comments on the same idea, both because it makes documentation easier, and because self-contained code functionality is easier to test and forget about. But we don't mean you need to write a book. Keep your words as brief as they need to be – sometimes that might mean a single line. How you add comments to code is dependent on the language you're using. In Python, for example, comments are usually demarcated by the **#** symbol in the first column of a line. Everything that comes after this symbol will be ignored by the interpreter, and if you're using an editor with syntax highlighting, the comment will also be coloured differently to make it more obvious. The more detail



› The IDLE Python IDE has a debug mode that can show how your variables change over time

you put into a comment the better, but don't write an essay. Adding comments to code can be tedious when you just want to get on with programming, so make them as brief as you can without stopping your flow. If necessary, you can go back and flesh out your thoughts when you don't feel like writing code (usually the day before a public release). When you start to code, you'll introduce many errors without realising it. To begin with, for example, you won't know what is and isn't a keyword – a word used by your chosen language to do something important. Each language is different, but Python's list of keywords is quite manageable, and includes common language words such as **and**, **if**, **else**, **import**, **class** and **break**, as well as less obvious words such as **yield**, **lambda**, **raise** and **assert**. This is why it's often a good idea to create your own variable names out of composite parts, rather than go with real words. If you're using an IDE, there's a good chance that its syntax highlighting will stop you from using a protected keyword.

Undeclared values

A related problem that doesn't affect Python is using undeclared values. This happens in C or C++, for instance, if you use a variable without first saying what type it's going to be, such as **int x** to declare **x** an integer. It's only after doing this you can use the variable in your own code. This is the big difference between compiled languages and interpreted ones. However, in both languages, you can't assume a default value for an uninitialised variable. Typing **print (x)** in Python, for instance, will result in an error, but not if you precede the line with **x = 1**. This is because the interpreter knows the type of a variable only after you've assigned it a value. C/C+ can be even more random, not necessarily generating an error, but the value held in an uninitialised variable is unpredictable until you've assigned it a value.

Typos are also common, especially in conditional statements, where they can go undetected because they are syntactically correct. Watch out for using a single equals sign to check for equality, for example – although Python is pretty good at catching these problems. Another type of problem Python is good at avoiding is inaccurate indenting. This is where conditions and functions use code hierarchy to split the code into parts. Python enforces this by breaking execution if you get it wrong, but other languages try to make sense of code hierarchy, and sometimes a misplaced bracket is all that's needed to create unpredictable results. However, this can make Python trickier to learn. Initially, if you don't know about its strict tabbed requirements, or that it needs a colon at the end of compound statement headers, the errors created don't make sense. You also need to be careful about case sensitivity, especially with keywords and your own variable names.

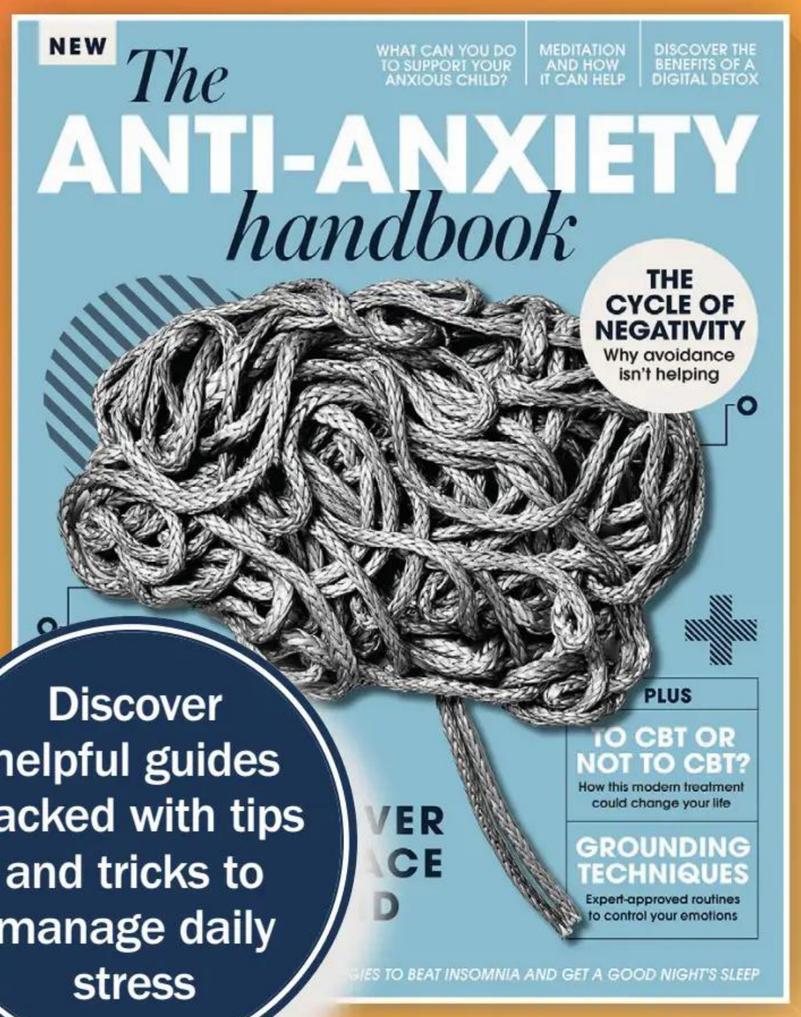
When you've got something that works, you need to test it – not just with the kind of values your application might expect, but with anything that can be input. Your code should fail gracefully, rather than randomly. And when you've got something ready to release, give it to other people to test. They'll have a different approach, and will be happier to break your code in ways you couldn't imagine. Only then will your code be ready for the wild frontier of the internet, and you'd better wear your flameproof jacket for that release. ■

```
graham : python
File  Edit  View  Bookmarks  Settings  Help
[962][graham.gm-arch: /home/graham]$ python
Python 3.2.3 (default, Apr 23 2012, 23:14:44)
[GCC 4.7.0 20120414 (prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> while (i):
...     print (i)
...     i = 0
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'i' is not defined
>>> i = 1
>>> while (i)
  File "<stdin>", line 1
    while (i)
           ^
SyntaxError: invalid syntax
```

› You have to be careful in Python that the colons and indentation are in the correct place, or your script won't run. But this does stop a lot of runtime errors

Different languages mark comments differently, and there seems to be little consensus on what a comment should look like. However, there are a couple of rules. Most languages offer both inline and block comments, for example. Inline are usually for a single line, or a comment after a piece of code on the same line, and they're initiated by using a couple of characters before the comment. Block comments are used to wrap pieces of text (or code you don't want interpreted/compiled), and usually have different start and end characters.

Bash	**#** A hash is used for comments in many scripting languages. When **#** is followed by a **!** it becomes a shebang **#** and is used to tell the system which interpreter to use, for example: **#!/usr/bin/bash**
BASIC	**REM** For many of us, this is the first comment syntax we learn
C	**/*** This kind of comment in C can be used to make a block of text span many lines ***/**
C++	**//** Whereas this kind of comment is used after the **//** code or for just a single line
HTML	**<!--** Though not a programming language, we've included this because you're likely to have already seen the syntax, and therefore comments, in action **-->**
Java	**/**** Similar to C, because it can span lines, but with an extra ***** at the beginning ***/**
Perl	**= heading Overview** As well as the hash, in Perl you can also use something called Plain Old Documentation. It has a specific format, but it does force you to explain your code more thoroughly **=cut**
Python	**'''** As well as the hash, Python users can denote blocks of comments using a source code literal called a docstring, which is a convoluted way of saying 'enclose your text in blocks of triple quotes', like this **'''**

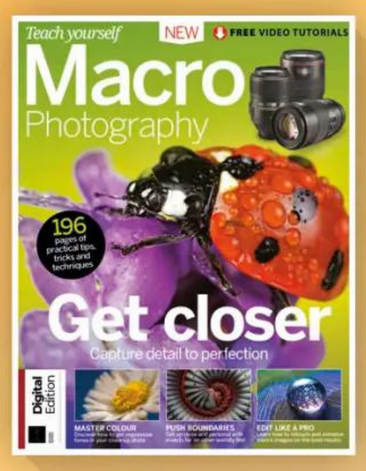


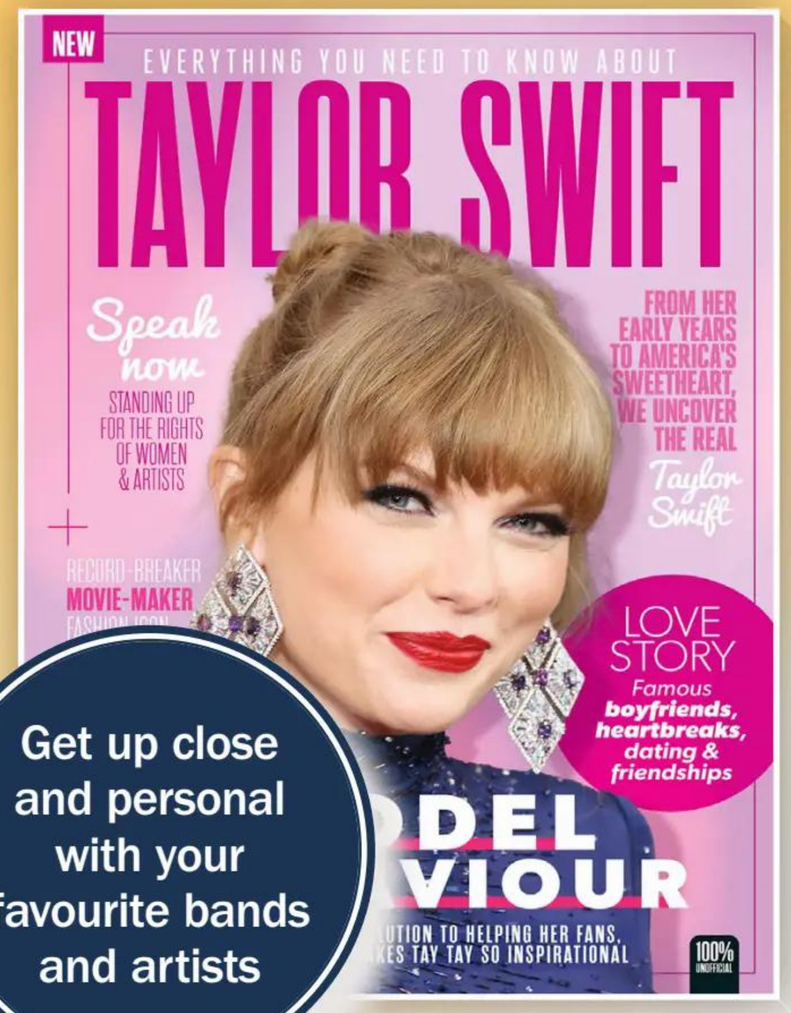
Discover helpful guides packed with tips and tricks to manage daily stress











Get up close and personal with your favourite bands and artists











Find inspiration and impress your dinner guests with recipes for every occasion











Get great savings when you buy direct from us

1000s of great titles, many not available anywhere else

World-wide delivery and super-safe ordering















DISCOVER OUR GREAT BOOKAZINES

From history and music to gaming and cooking, pick up a book that will take your hobby to the next level

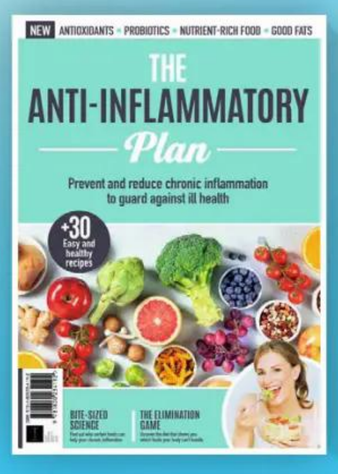














Embrace relaxation and get creative with a range of fun colouring books



Follow us on Instagram @futurebookazines

www.magazinesdirect.com

FUTURE

Magazines, back issues & bookazines.

SUBSCRIBE & SAVE UP TO 61%

Delivered direct to your door
or straight to your device



Choose from over 80 magazines and make great savings off the store price!

Binders, books and back issues also available

Simply visit www.magazinesdirect.com

No hidden costs · Shipping included in all prices · We deliver to over 100 countries · Secure online payment

FUTURE

magazinesdirect.com
Official Magazine Subscription Store

CODING ACADEMY 2025

164 PACKED PAGES! TAKE YOUR CODING SKILLS TO THE NEXT LEVEL

PYTHON Learn to code with practical, easy examples

PROJECTS Discover new tricks and functions with fun guides

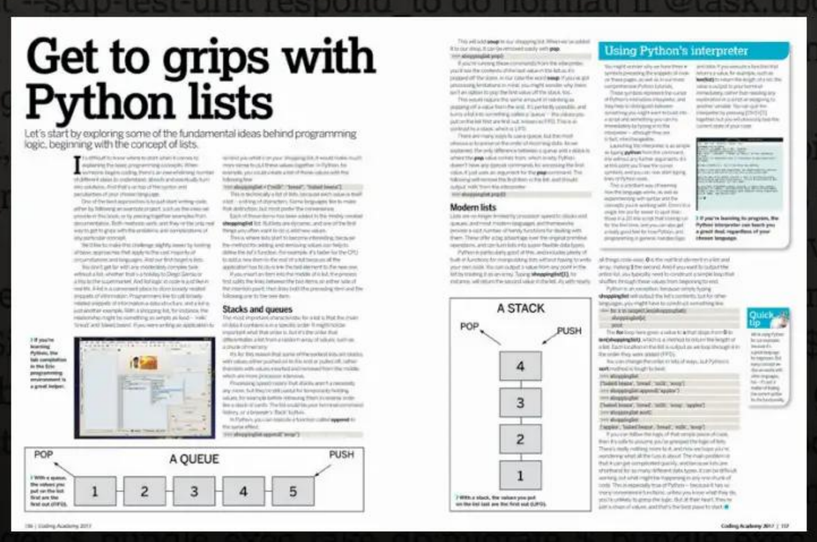
DATABASES Master MongoDB, MariaDB, NoSQL and more

RUST Get started with the new secure language



Learn coding essentials and dive straight into your first real-world program

Unleash your creativity with fun projects for games, graphing and GUI design



Includes a full programming reference with refreshers on every key coding concept



FULLY UPDATED EDITION

BOOKAZINE

9021

9000