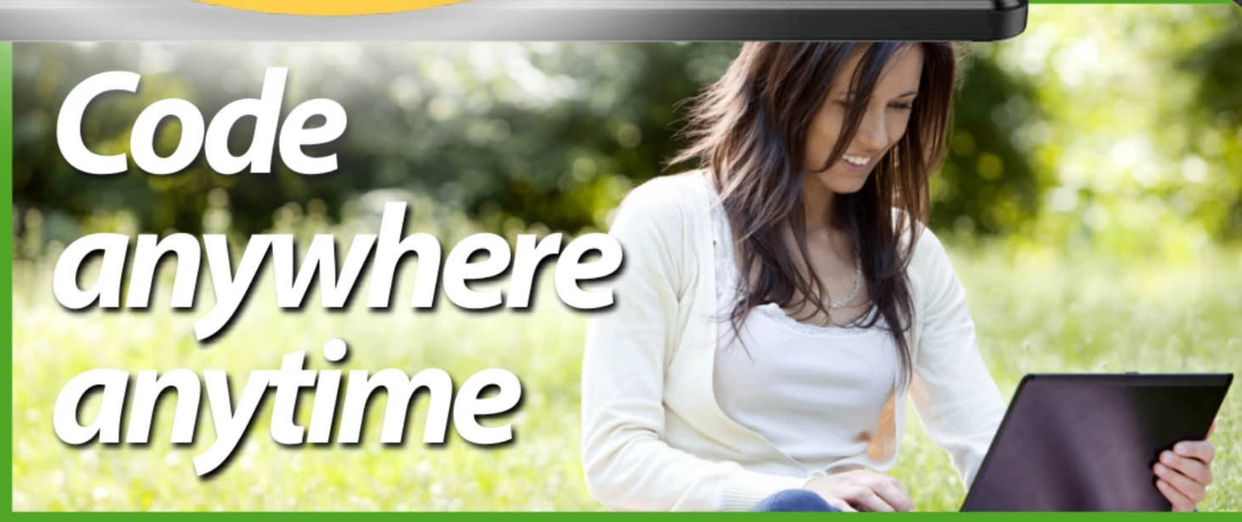
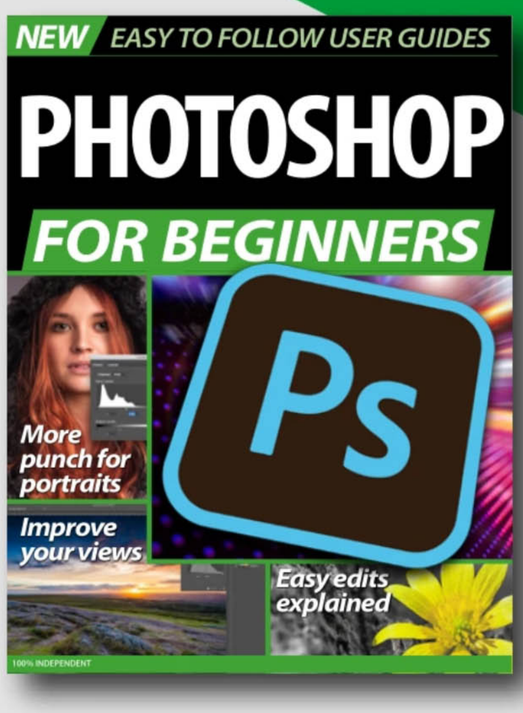
NEW EASY TO FOLLOW USER GUIDES

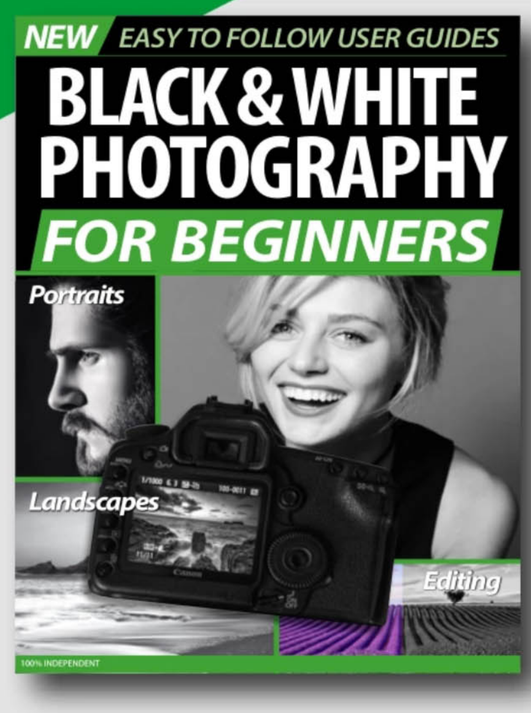
# PYTHON

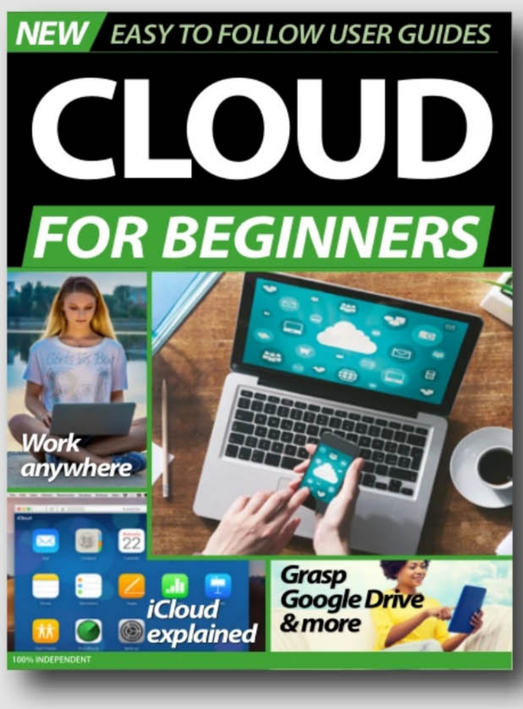
## FOR BEGINNERS

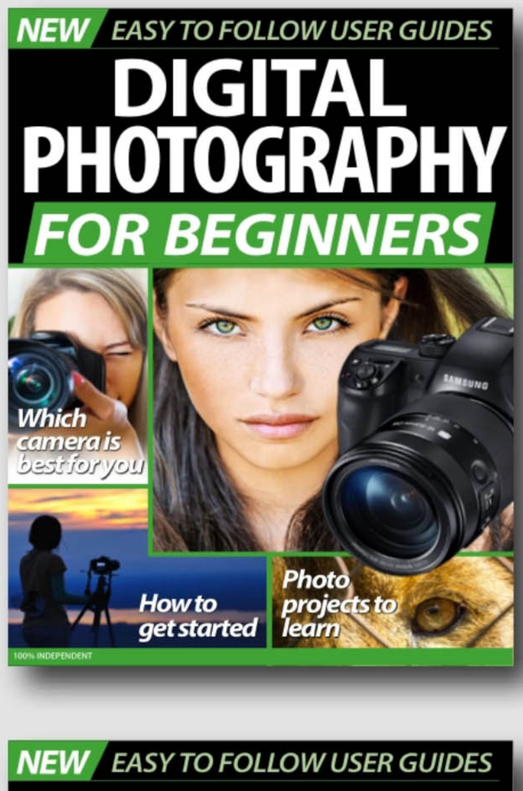


# Discover more of our guides...







# PYTHON

## FOR BEGINNERS

Starting something new can be daunting. Learning a skill or mastering a new piece of hardware is tough. Even tougher if you have no-one at hand to help. Conversely as the complexity of our consumer technology increases, the size of the requisite instruction manual decreases or in some cases it simply disappears. At numerous times in our lives we have all been "beginners", there is no shame in that fact and rightly so. How many times have you asked aloud, "What does this button do?". "Why doesn't that work?". "What do you mean it doesn't do that?". "HELP!". At the start of any new journey or adventure we are all beginners but fortunately for you we are here to stand beside you at every stage.

Over this extensive series of titles we will be looking in great depth at the latest consumer electronics, software, hobbies and trends out of the box! We will guide you step-by-step through using all aspects of the technology that you may have been previously apprehensive at attempting. Let our expert guide help you build your technology understanding and skills, taking you from a novice to a confident and experienced user.

Over the page our journey begins. We would wish you luck but we're sure with our support you won't need it.

# Contents

**BDM's Code Portal**
**60+ Python programs**
**21,500+ lines of code**
Master Python with the help of our fantastic Code Portal, featuring code for games, tools and more.
Visit: https://bdmpublications.com/code-portal, and log in to get access!

## 6 Working with Code

8 Common Coding Mistakes
10 Python Beginner's Mistakes
12 C++ Beginner's Mistakes
14 Where Next?

## 16 Say Hello to Python

18 Why Python?
20 Equipment You Will Need
22 Getting to Know Python
24 How to Set Up Python in Windows
26 How to Set Up Python in Linux

## 28 Getting Started with Python

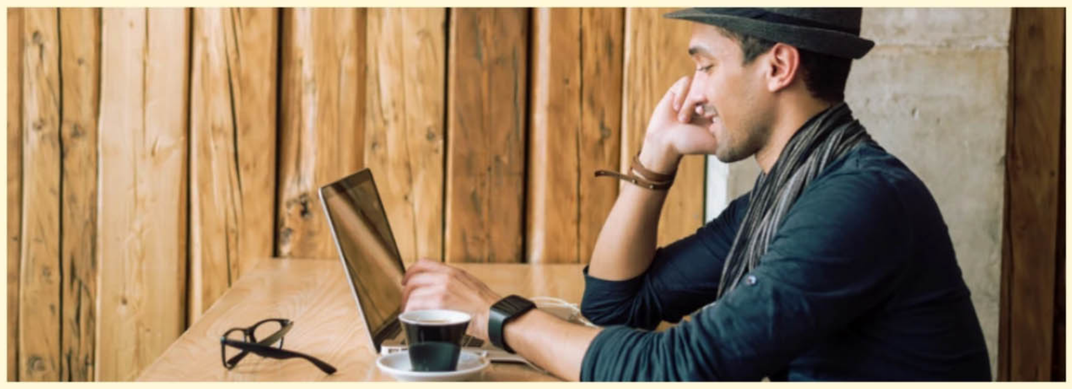
30 Starting Python for the First Time
32 Your First Code
34 Saving and Executing Your Code
36 Executing Code from the Command Line
38 Numbers and Expressions
40 Using Comments
42 Working with Variables
44 User Input
46 Creating Functions
48 Conditions and Loops
50 Python Modules

## 52 Working with Data

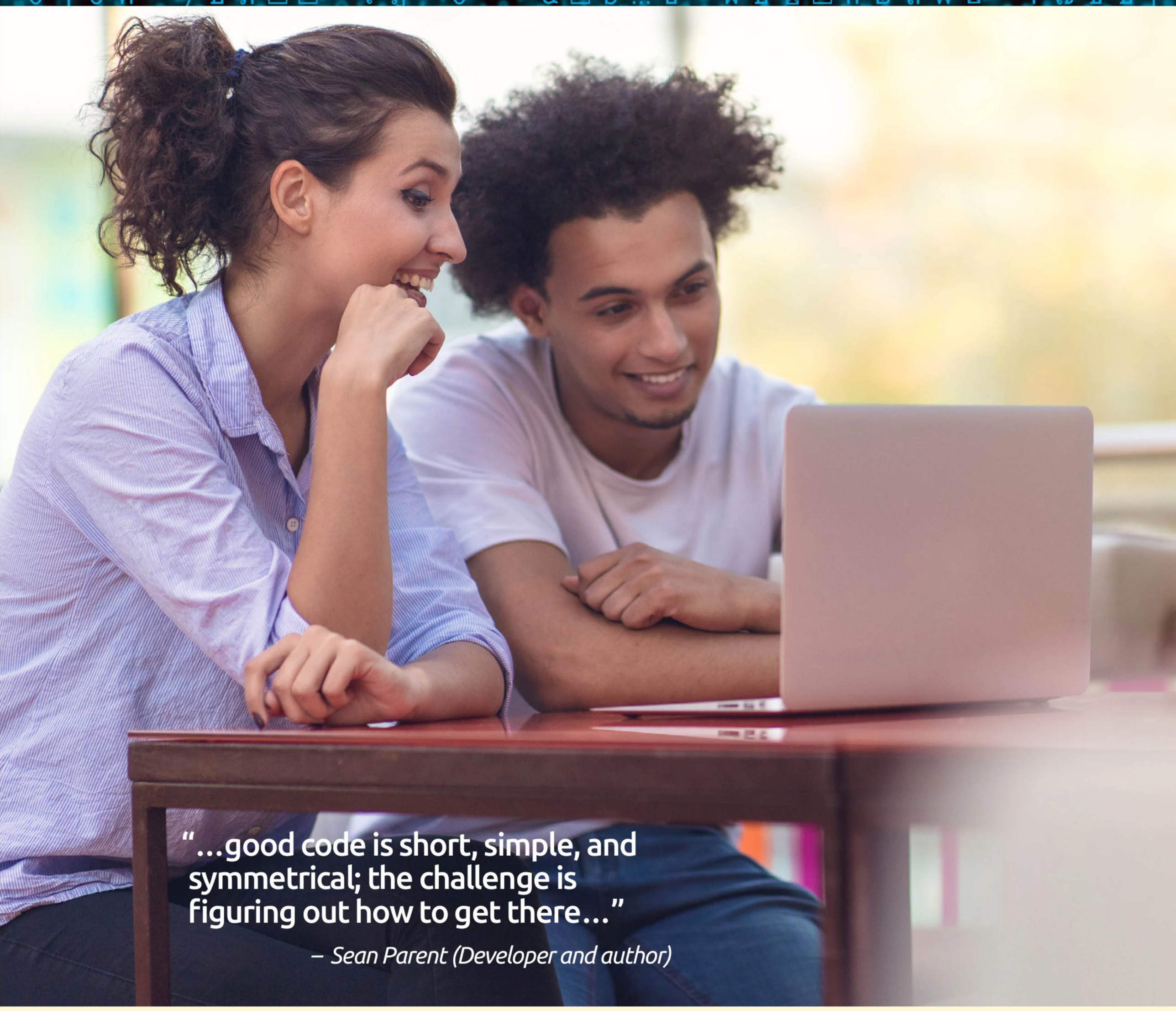
54 Lists
56 Tuples
58 Dictionaries
60 Splitting and Joining Strings
62 Formatting Strings
64 Date and Time
66 Opening Files
68 Writing to Files
70 Exceptions
72 Python Graphics

## 74 Using Modules

76 Calendar Module
78 OS Module
80 Random Module
82 Tkinter Module
84 Pygame Module
88 Basic Animation
90 Create Your Own Modules



Contents




**Code with Python For Beginners**
ISBN: 978-1-912847-11-2
**Published by:** Papercut Limited
**Digital distribution by:**
Readly AB, Zinio, Magzter, Cafeyn, PocketMags

© 2020 Papercut Limited All rights reserved. No part of this publication may be reproduced in any form, stored in a retrieval system or integrated into any other publication, database or commercial programs without the express written permission of the publisher. Under no circumstances should this publication and its contents be resold, loaned out or used in any form by way of trade without the publisher's written permission. While we pride ourselves on the quality of the information we provide, Papercut Limited reserves the right not to be held responsible for any mistakes or inaccuracies found within the text of this publication. Due to the nature of the tech industry, the publisher cannot guarantee that all apps and software will work on every version of device. It remains the purchaser's sole responsibility to determine the suitability of this book and its content for whatever purpose. Any app images reproduced on the front and back cover are solely for design purposes and are not representative of content. We advise all potential buyers to check listing prior to purchase for confirmation of actual content. All editorial opinion herein is that of the reviewer - as an individual - and is not representative of the publisher or any of its affiliates. Therefore the publisher holds no responsibility in regard to editorial opinion and content.
This is an independent publication and as such does not necessarily reflect the views or opinions of the producers of apps or products contained within. This publication is 100% unofficial. All copyrights, trademarks and registered trademarks for the respective companies are acknowledged. Relevant graphic imagery reproduced with courtesy of brands and products. Additional images contained within this publication are reproduced under licence from Shutterstock. Prices, international availability, ratings, titles and content are subject to change.

All information was correct at time of publication. Some content may have been previously published in other volumes or titles.

Papercut Limited
Registered in England & Wales No: 4308513

**“Most good programmers do programming not because they expect to get paid or get adulation by the public, but because it is fun to program.”**

*– Linus Torvalds (developer of the Linux kernel)*

# Working with Code

It's always worth looking at the common pitfalls of coding before you start hitting away at the keyboard; after all, bad habits are often formed at the start of a process. If you're aware of how to structure your code before you begin, then you will learn to avoid most of the mistakes that usually accompany poorly developed code.

This section starts your coding journey by examining those common mistakes, as well as investigating where your coding skills can take you as your experience grows.

# Common Coding Mistakes

When you start something new you're inevitably going to make mistakes, this is purely down to inexperience and those mistakes are great teachers in themselves. However, even experts make the occasional mishap. Thing is, to learn from them as best you can.

## X=MISTAKE, PRINT Y

There are many pitfalls for the programmer to be aware of, far too many to be listed here. Being able to recognise a mistake and fix it is when you start to move into more advanced territory, and become a better coder. Everyone makes mistakes, even coders with over thirty years' experience. Learning from these basic, common mistakes help build a better coding foundation.

## SMALL CHUNKS

It would be wonderful to be able to work like Neo from The Matrix movies. Simply ask, your operator loads it into your memory and you instantly know everything about the subject. Sadly though, we can't do that. The first major pitfall is someone trying to learn too much, too quickly. So take coding in small pieces and take your time.

## //COMMENTS

Use comments. It's a simple concept but commenting on your code saves so many problems when you next come to look over it. Inserting comment lines helps you quickly sift through the sections of code that are causing problems; also useful if you need to review an older piece of code.

```
52          orig += 2;
53          target += 2;
54          --n;
55        }
56  #endif
57    if (n == 0)
58      return;
59
60    //
61    // Loop unrolling. Here be dragons.
62    //
63
64    // (n & (~3)) is the greatest multiple of 4 r
65    // In the while loop ahead, orig will move ov
66    // increments (4 elements of 2 bytes).
67    // end marks our barrier for not falling outs
68    char const * const end = orig + 2 * (n & (~3)
69
70    // See if we're aligned for writting in 64 or
71  #if ACE_SIZEOF_LONG == 8 && \
72      !(defined(  amd64  ) || defined (  x86_64
```

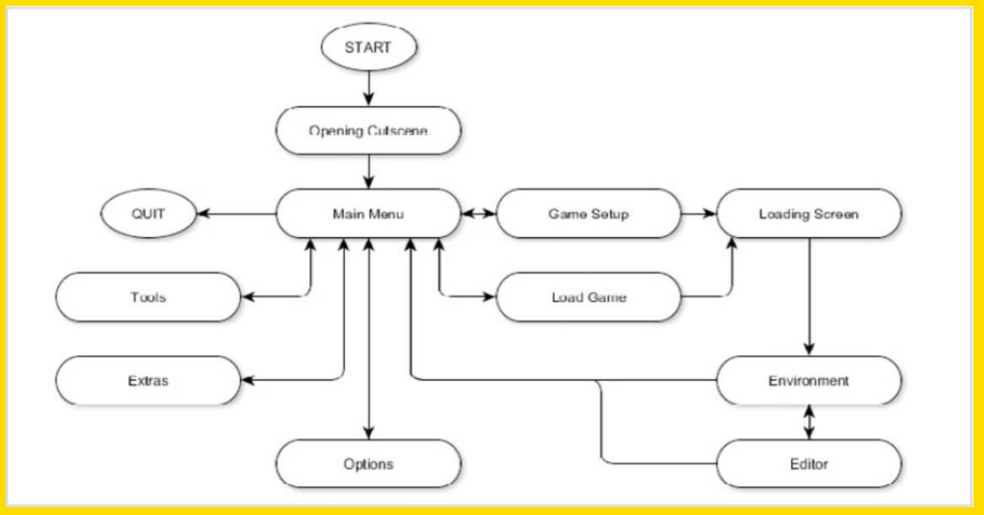
## EASY VARIABLES

Meaningful naming for variables is a must to eliminate common coding mistakes. Having letters of the alphabet is fine but what happens when the code states there's a problem with x variable. It's not too difficult to name variables lives, money, player1 and so on.

```
1  var points = 1023;
2  var lives = 3;
3  var totalTime = 45;
4  write("Points: "+points);
5  write("Lives: "+lives);
6  write("Total Time: "+totalTime+" secs");
7  write("--------------------------");
8  var totalScore = 0;
9  write("Your total Score is: "+totalScore);
```

## PLAN AHEAD

While it's great to wake up one morning and decide to code a classic text adventure, it's not always practical without a good plan. Small snippets of code can be written without too much thought and planning but longer and more in-depth code requires a good working plan to stick to and help iron out the bugs.



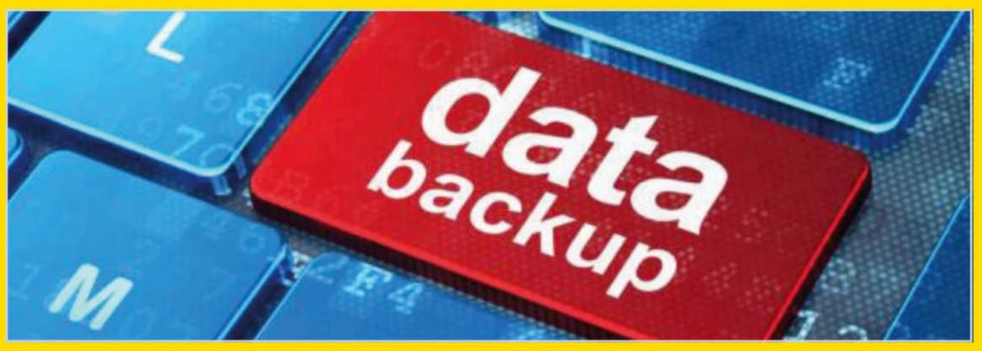
## USER ERROR

User input is often a paralysing mistake in code. For example, when the user is supposed to enter a number for their age and instead they enter it in letters. Often a user can enter so much into an input that it overflows some internal buffer, thus sending the code crashing. Watch those user inputs and clearly state what's needed from them.

```
Enter an integer number
aswdfdsf
You have entered wrong input
s
You have entered wrong input
!"£"!£!"
You have entered wrong input
sdfsdf213213123
You have entered wrong input
123234234234234234
You have entered wrong input
12
the number is: 12

Process returned 0 (0x0)   execution time : 21.495 s
Press any key to continue.
```

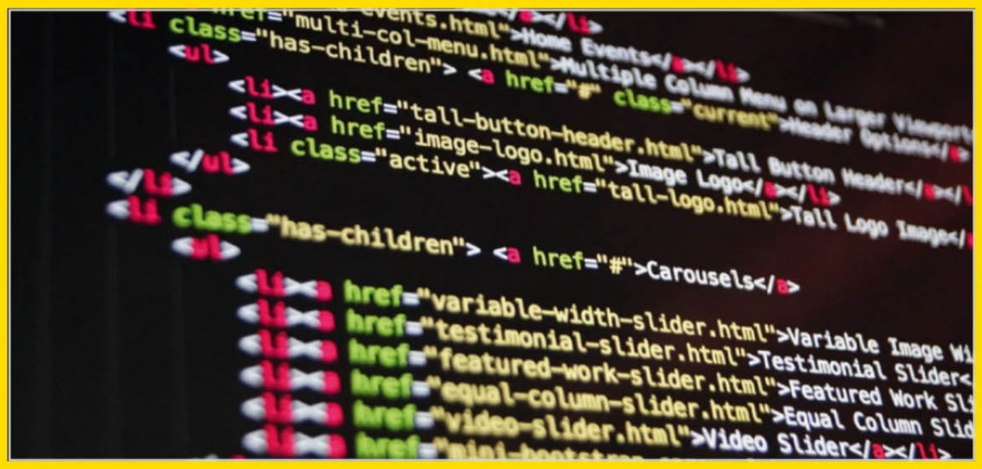
## BACKUPS

Always make a backup of your work, with a secondary backup for any changes you've made. Mistakes can be rectified if there's a good backup in place to revert to for those times when something goes wrong. It's much easier to start where you left off, rather than starting from the beginning again.




## RE-INVENTING WHEELS

You can easily spend days trying to fathom out a section of code to achieve a given result and it's frustrating and often time-wasting. While it's equally rewarding to solve the problem yourself, often the same code is out there on the Internet somewhere. Don't try and re-invent the wheel, look to see if some else has done it first.

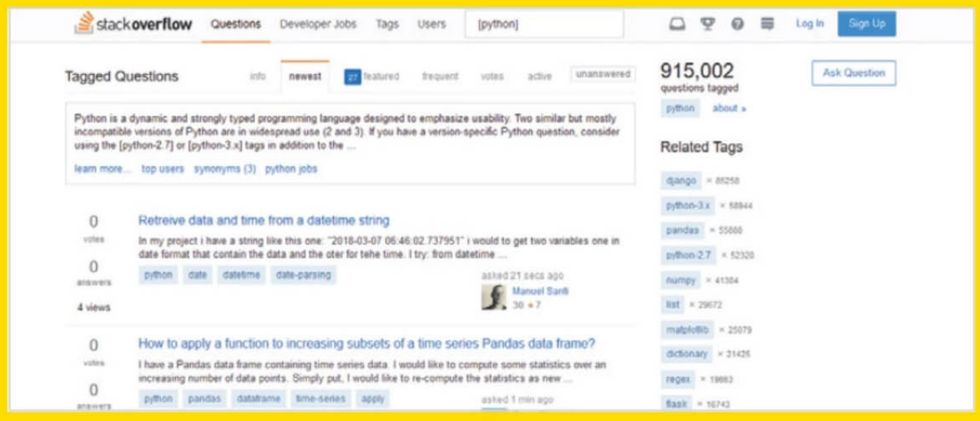


## SECURE DATA

If you're writing code to deal with usernames and passwords, or other such sensitive data, then ensure that the data isn't in cleartext. Learn how to create a function to encrypt sensitive data, prior to feeding into a routine that can transmit or store it where someone may be able to get to view it.

## HELP!

Asking for help is something most of us has struggled with in the past. Will the people we're asking laugh at us? Am I wasting everyone's time? It's a common mistake for someone to suffer in silence. However, as long as you ask the in the correct manner, obey any forum rules and be polite, then your question isn't silly.




## MATHS

If your code makes multiple calculations then you need to ensure that the maths behind it is sound. There are thousands of instances where programs have offered incorrect data based on poor mathematical coding, which can have disastrous effects depending on what the code is set to do. In short, double check your code equations.

```
set terminal x11
set output

rmax = 5
nmax = 100

complex (x, y) = x * {1, 0} + y * {0, 1}
mandel (x, y, z, n) = (abs (z)> rmax || n>= 100)? n: mandel (x, y, z * z + complex (x, y), n + 1)

set xrange [-0.5:0.5]
set yrange [-0.5:0.5]
set logscale z
set samples 200
set isosample 200
set pm3d map
set size square
a= #A#
b= #B#
splot mandel(-a/100,-b/100,complex(x,y),0) notitle
```

# Python Beginner's Mistakes

Python is a relatively easy language to get started in where there's plenty of room for the beginner to find their programming feet. However, as with any other programming language, it can be easy to make common mistakes that'll stop your code from running.

## DEF BEGINNER(MISTAKES=10)

**Here are ten common Python programming mistakes most beginners find themselves making. Being able to identify these mistakes will save you headaches in the future.**

## VERSIONS

To add to the confusion that most beginners already face when coming into programming, Python has two live versions of its language available to download and use. There is Python version 2.7.x and Python 3.6.x. The 3.6.x version is the most recent, and the one we'd recommend starting. But, version 2.7.x code doesn't always work with 3.6.x code and vice versa.

## INDENTS, TABS AND SPACES

Python uses precise indentations when displaying its code. The indents mean that the code in that section is a part of the previous statement, and not something linked with another part of the code. Use four spaces to create an indent, not the Tab key.

```
MOVESPEED = 11
MOVE = 1
SHOOT = 15

# set up counting
score = 0

# set up font
font = pygame.font.SysFont('calibri', 50)

def makeplayer():
    player = pygame.Rect(370, 635, 60, 25)
    return player

def makeinvaders(invaders):
    y = 0
    for i in invaders:
        x = 0
        for j in range(11):
            invader = pygame.Rect(75+x, 75+y, 50, 20)
            i.append(invader)
            x += 60
        y += 45
    return invaders

def makewalls(walls):
    wall1 = pygame.Rect(60, 520, 120, 30)
    wall2 = pygame.Rect(246, 520, 120, 30)
    wall3 = pygame.Rect(432, 520, 120, 30)
    wall4 = pygame.Rect(618, 520, 120, 30)
```

## THE INTERNET

Every programmer has and does at some point go on the Internet and copy some code to insert into their own routines. There's nothing wrong with using others' code, but you need to know how the code works and what it does before you go blindly running it on your own computer.

### Create/delete a .txt file in a python program

I have created a program to grab values from a text file. As you can see, depending on the value of the results, I have an if/else statement printing out the results of the scenario.

My problem is I want to set the code up so that the if statement creates a simple .txt file called data.txt to the C:\Python\Scripts directory.

In the event the opposite is true, I would like the else statement to delete this .txt file if it exists.

I'm a novice programmer and anything I've looked up or tried hasn't worked for me, so any help or assistance would be hugely appreciated.

```
import re

x = open("test.txt","r")
california = x.readlines(11)
dublin = x.readlines(125)

percentage_value = [float(re.findall('\d+\.\d+(?=\%)|\d+\.\d+(?=\s\%)', i[-1])[0]) for i in [ca

print(percentage_value)

if percentage_value[0] <= percentage_value[1]:
    print('Website is hosted in Dublin')
else:
```

## COMMENTING

Again we mention commenting. It's a hugely important factor in programming, even if you're the only one who is ever going to view the code, you need to add comments as to what's going on. Is this function where you lose a life? Write a comment and help you, or anyone else, see what's going on.

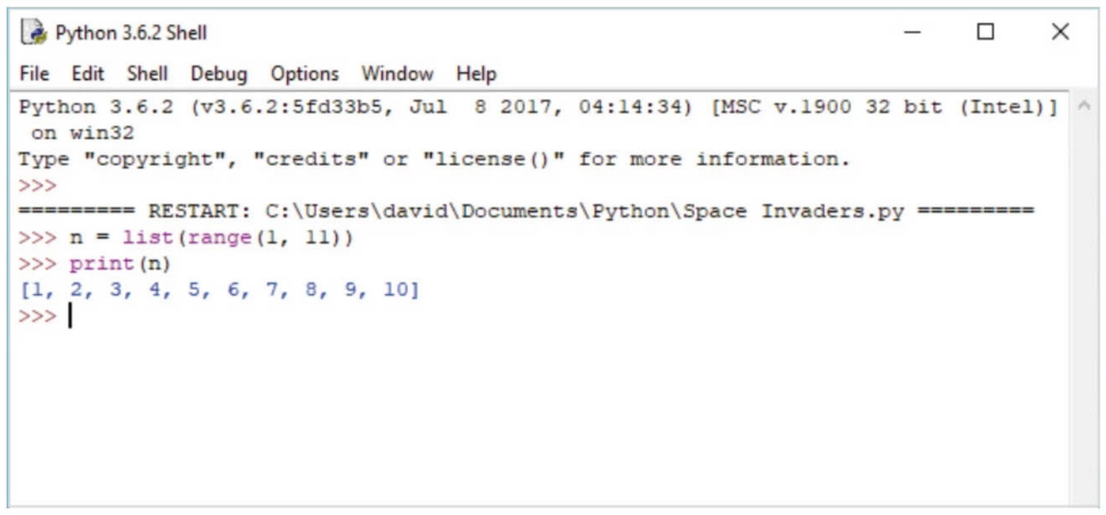
```
# set up pygame
pygame.init()
mainClock = pygame.time.Clock()

# set up the window
width = 800
height = 700
screen = pygame.display.set_mode((width, height), 0, 32)
pygame.display.set_caption('caption')

# set up movement variables
moveLeft = False
moveRight = False
moveUp = False
moveDown = False

# set up direction variables
DOWNLEFT = 1
DOWNRIGHT = 3
```

## COUNTING LOOPS

Remember that in Python a loop doesn't count the last number you specify in a range. So if you wanted the loop to count from 1 to 10, then you will need to use:

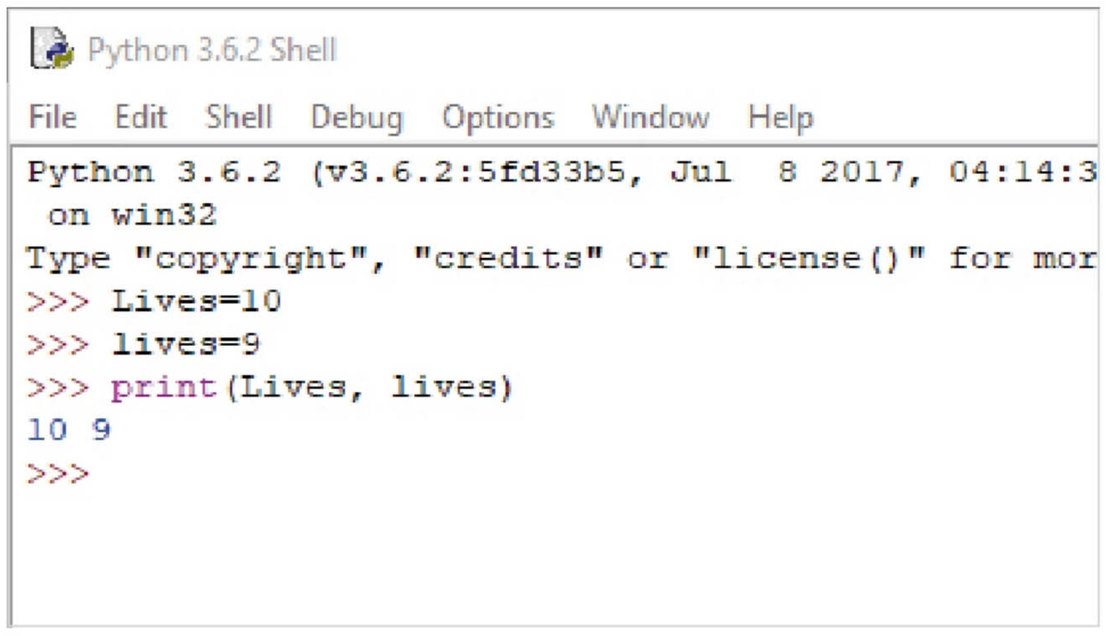
```
n = list(range(1, 11))
```

Which will return 1 to 10.




## CASE SENSITIVE

Python is a case sensitive programming language, so you will need to check any variables you assign. For example, `Lives=10` is a different variable to lives=10, calling the wrong variable in your code can have unexpected results.




## BRACKETS

Everyone forgets to include that extra bracket they should have added to the end of the statement. Python relies on the routine having an equal amount of closed brackets to open brackets, so any errors in your code could be due to you forgetting to count your brackets; including square brackets.

```
def print_game_status(self):
        print (board[len(self.missed_letters)])
        print ('Word: ' + self.hide_word())
        print ('Letters Missed: ',)
        for letter in self.missed_letters:
                print (letter,)
        print ()
        print ('Letters Guessed: ',)
        for letter in self.guessed_letters:
                print (letter,)
        print ()
```

## COLONS

It's common for beginners to forget to add a colon to the end of a structural statement, such as:

```
class Hangman:
def guess(self, letter):
```

And so on. The colon is what separates the code, and creates the indents to which the following code belongs to.

```
class Hangman:
        def __init__(self,word):
                self.word = word
                self.missed_letters = []
                self.guessed_letters = []

        def guess(self,letter):
                if letter in self.word and letter not in self.guessed_letters:
                        self.guessed_letters.append(letter)
                elif letter not in self.word and letter not in self.missed_letters:
                        self.missed_letters.append(letter)
                else:
                        return False
                return True

        def hangman_over(self):
                return self.hangman_won() or (len(self.missed_letters) == 6)

        def hangman_won(self):
                if '_' not in self.hide_word():
                        return True
                return False

        def hide_word(self):
                rtn = ''
                for letter in self.word:
                        if letter not in self.guessed_letters:
                                rtn += '_'
                        else:
                                rtn += letter
                return rtn
```

## OPERATORS

Using the wrong operator is also a common mistake to make. When you're performing a comparison between two values, for example, you need to use the equality operator (a double equals, ==). Using a single equal (=) is an assignment operator that places a value to a variable (such as, lives=10).

```
1 b = 5
2 c = 10
3 d = 10
4 b == c #false because 5 is not equal to 10
5 c == d #true because 10 is equal to 10
```

## OPERATING SYSTEMS

Writing code for multiple platforms is difficult, especially when you start to utilise the external commands of the operating system. For example, if your code calls for the screen to be cleared, then for Windows you would use `cls`. Whereas, for Linux you need to use `clear`. You need to solve this by capturing the error and issuing it with an alternative command.

```
# Code to detect error for using a different OS
run=1
while(run==1):
    try:
        os.system('clear')
    except OSError:
        os.system('cls')
    print('\n>>>>>>>>>>Python 3 File Manager<<<<<<<<<<\n')
```

# C++ Beginner's Mistakes

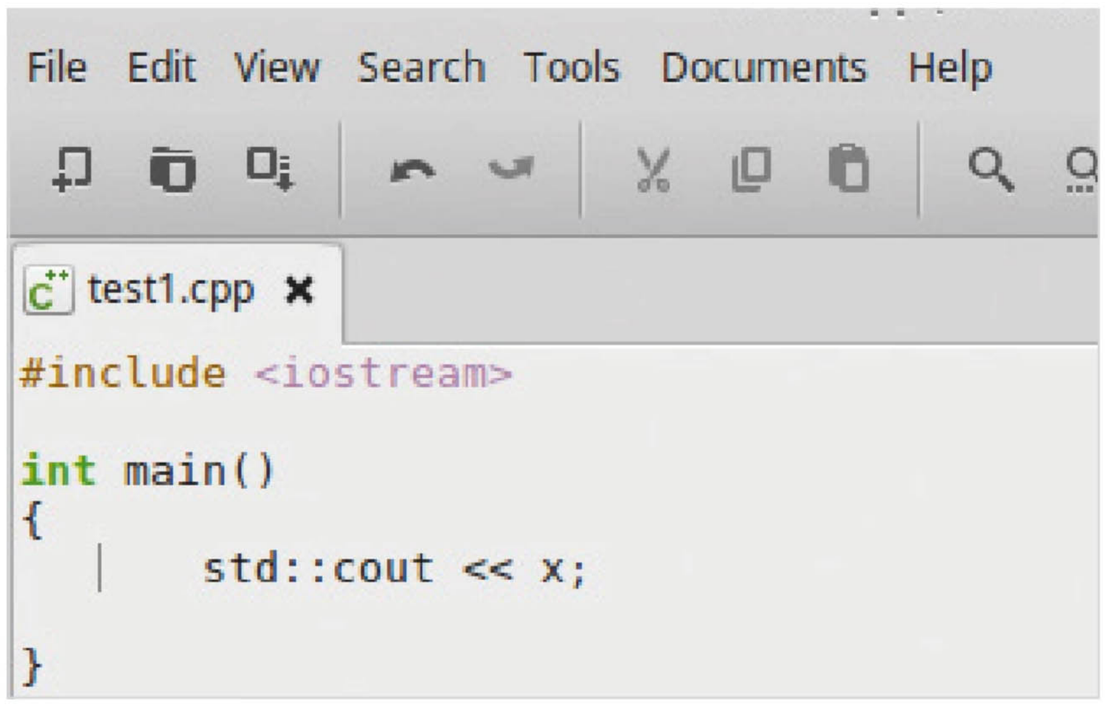
There are many pitfalls the C++ developer can encounter, especially as this is a more complex and often unforgiving language to master. Beginners need to take C++ a step at a time and digest what they've learned before moving on.

## VOID(C++, MISTAKES)

**Admittedly it's not just C++ beginners that make the kinds of errors we outline on these pages, even hardened coders are prone to the odd mishap here and there. Here are some common issues to try and avoid.**

## UNDECLARED IDENTIFIERS

A common C++ mistake, and to be honest a common mistake with most programming languages, is when you try and output a variable that doesn't exist. Displaying the value of x on-screen is fine but not if you haven't told the compiler what the value of x is to begin with.




## STD NAMESPACE

Referencing the Standard Library is common for beginners throughout their code, but if you miss the std:: element of a statement, your code errors out when compiling. You can combat this by adding:

```
using namespace std;
```

Under the #include part and simply using cout, cin and so on from then on.

```
#include <iostream>
using namespace std;

int main()
{
    int a, b, c, d;
    a=10;
    b=20;
    c=30;
    d=40;

    cout << a, b, c, d;

}
```

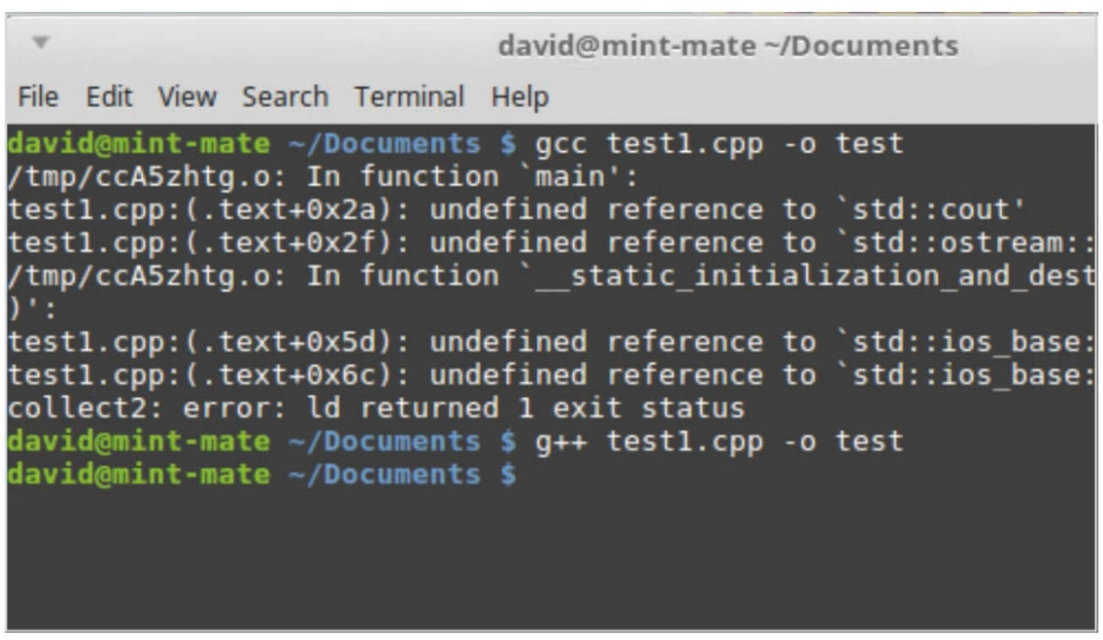
## SEMICOLONS

Remember that each line of a C++ program must end with a semicolon. If it doesn't then the compiler treats the line with the missing semicolon as the same line with the next semicolon on. This creates all manner of problems when trying to compile, so don't forget those semicolons.

```
#include <iostream>


int main()
{
    int a, b, c, d;
    a=10;
    b=20;
    c=30
    d=40;

    std::cout << a, b, c, d;

}
```

## GCC OR G++

If you're compiling in Linux then you will no doubt come across gcc and g++. In short, gcc is the Gnu Compiler Collection (or Gnu C Compiler as it used to be called) and g++ is the Gnu ++ (the C++ version) of the compiler. If you're compiling C++ then you need to use g++, as the incorrect compiler drivers will be used.



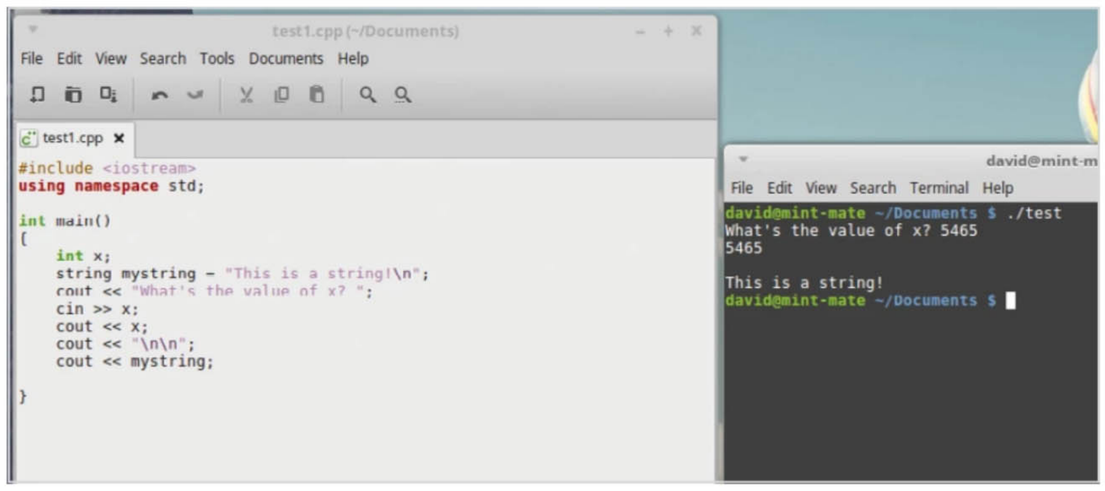
## COMMENTS (AGAIN)

Indeed the mistake of never making any comments on code is back once more. As we've previously bemoaned, the lack of readable identifiers throughout the code makes it very difficult to look back at how it worked, for both you and someone else. Use more comments.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    vector<double> v;

    double d;
    while(cin>>d) v.push_back(d);   // read elements
    if (!cin.eof()) {           // check if input failed
        cerr << "format error\n";
        return 1;                   // error return
    }

    cout << "read " << v.size() << " elements\n";

    reverse(v.begin(),v.end());
    cout << "elements in reverse order:\n";
    for (int i = 0; i<v.size(); ++i) cout << v[i] << '\n';

    return 0;                       // success return
}
```

## QUOTES

Missing quotes is a common mistake to make, for every level of user. Remember that quotes need to encase strings and anything that's going to be outputted to the screen or into a file, for example. Most compilers errors are due to missing quotes in the code.




## EXTRA SEMICOLONS

While it's necessary to have a semicolon at the end of every C++ line, there are some exceptions to the rule. Semicolons need to be at the end of every complete statement but some lines of code aren't complete statements. Such as:

`#include`
`if lines`
`switch lines`

If it sounds confusing don't worry, the compiler lets you know where you went wrong.

```
// Program to print positive number entered by the user
// If user enters negative number, it is skipped

#include <iostream>
using namespace std;

int main()
{
      int number;
      cout << "Enter an integer: ";
      cin >> number;

      // checks if the number is positive
      if ( number > 0)
      {
            cout << "You entered a positive integer: " << number << endl;
      }

      cout << "This statement is always executed.";
      return 0;

}
```

## TOO MANY BRACES

The braces, or curly brackets, are beginning and ending markers around blocks of code. So for every { you must have a }. Often it's easy to include or miss out one or the other facing brace when writing code; usually when writing in a text editor, as an IDE adds them for you.

```
#include <iostream>
using namespace std;

int main()
{
      int x;
      string mystring = "This is a string!\n";
      cout << "What's the value of x? ";
      cin >> x;
      cout << x;
      {
      cout << "\n\n";
      cout << mystring;

}
```

## INITIALISE VARIABLES

In C++ variables aren't initialised to zero by default. This means if you create a variable called x then, potentially, it is given a random number from 0 to 18,446,744,073,709,551,616, which can be difficult to include in an equation. When creating a variable, give it the value of zero to begin with: `x=0`.

```
#include <iostream>
using namespace std;

int main()
{
      int x;
      x=0;

      cout << x;


}
```

## A.OUT

A common mistake when compiling in Linux is forgetting to name your C++ code post compiling. When you compile from the Terminal, you enter:

```
g++ code.cpp
```

This compiles the code in the file code.cpp and create an a.out file that can be executed with ./a.out. However, if you already have code in a.out then it's overwritten. Use:

```
g++ code.cpp -o nameofprogram
```

```
david@mint-mate ~/Documents
File Edit View Search Terminal Help
david@mint-mate ~/Documents $ g++ test1.cpp
david@mint-mate ~/Documents $ ./a.out
0
david@mint-mate ~/Documents $ g++ test1.cpp -o printzero
david@mint-mate ~/Documents $ ./printzero
0
david@mint-mate ~/Documents $
```

# Where Next?

Coding, like most subjects, is a continual learning experience. You may not class yourself as a beginner anymore but you still need to test your code, learn new tricks and hacks to make it more efficient and even branch out and learn another programming language.

## #INCLUDE<KEEP ON LEARNING>

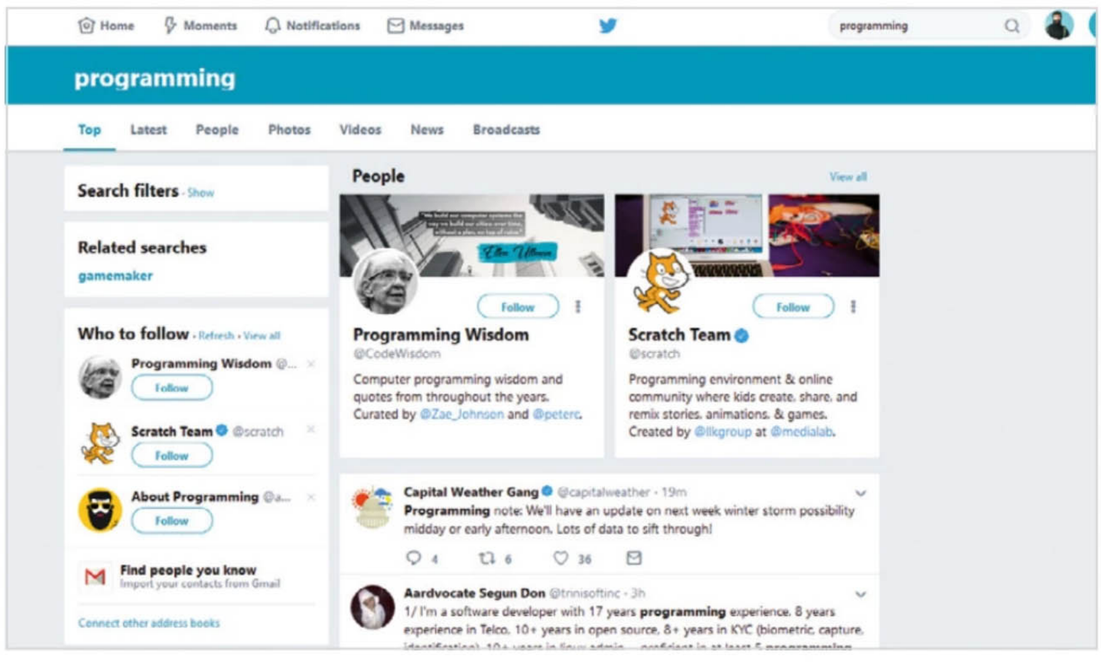
**What can you do to further your skills, learn new coding practises, experiment and present your code and even begin to help others using what you've experienced so far?**

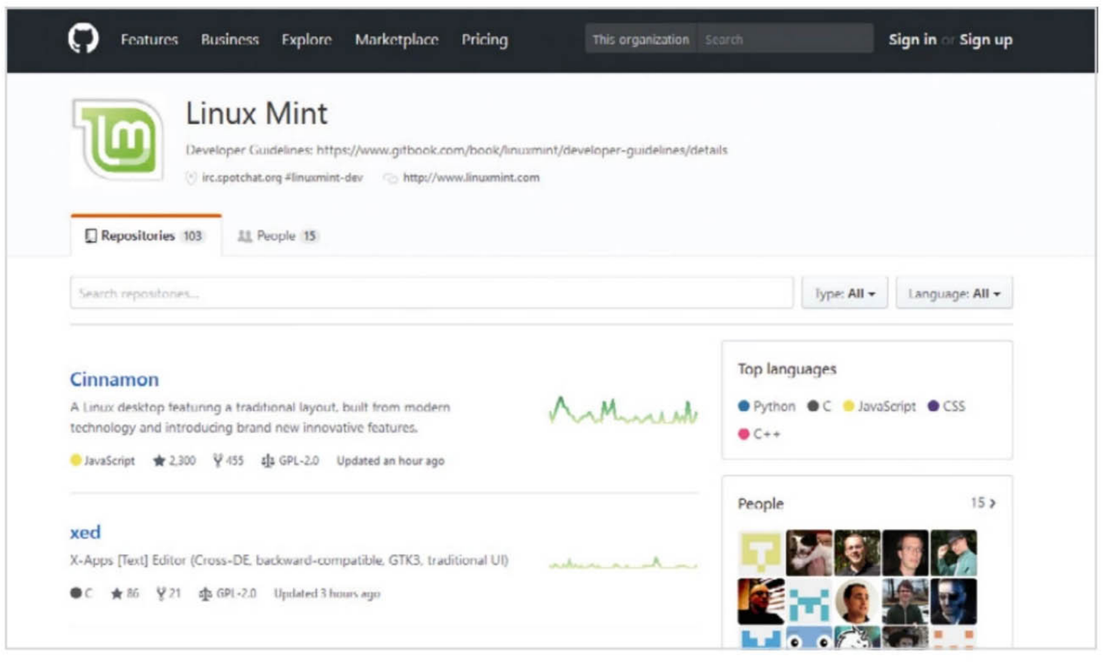
### TWITTER

Twitter isn't all trolls and antagonists, among the well-publicised vitriol are some genuine people who are more than willing to spread their coding knowledge. We recommend you find a few who you can relate to and follow them. Often they post great tips, hacks and fixes for common coding problems.

### OPEN PROJECTS

Look for open source projects that you like the sound of and offer to contribute to the code to keep it alive and up to date. There are millions of projects to choose from, so contact a few and see where they need help. It may only be a minor code update but it's a noble occupation for coders to get into.



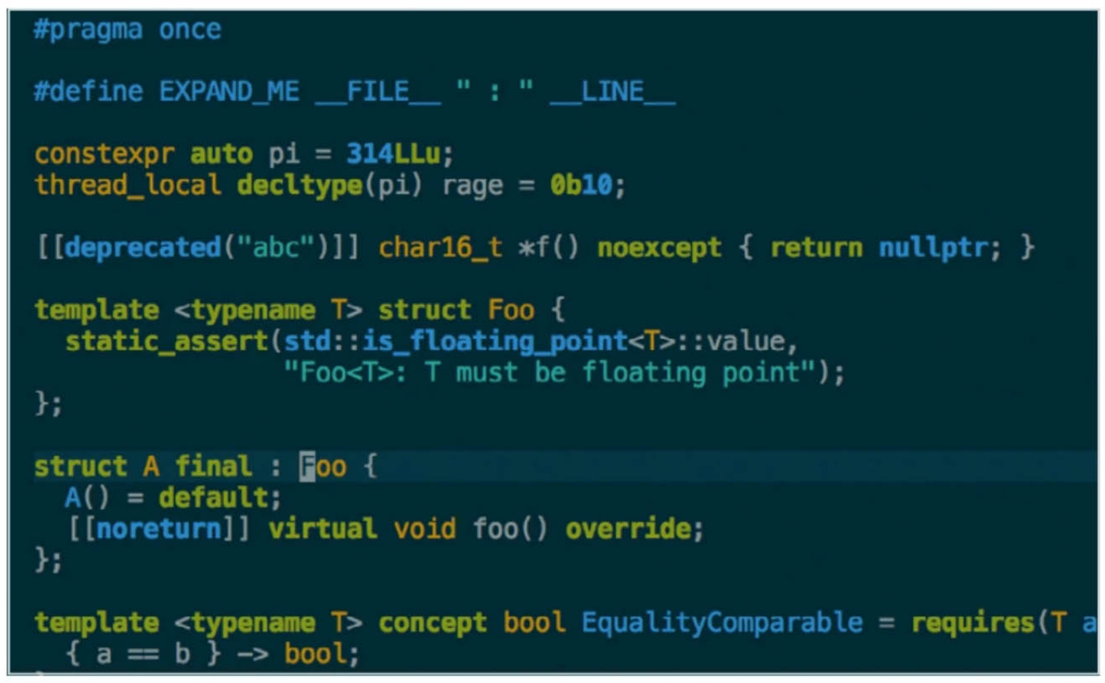




### KEEP CODING

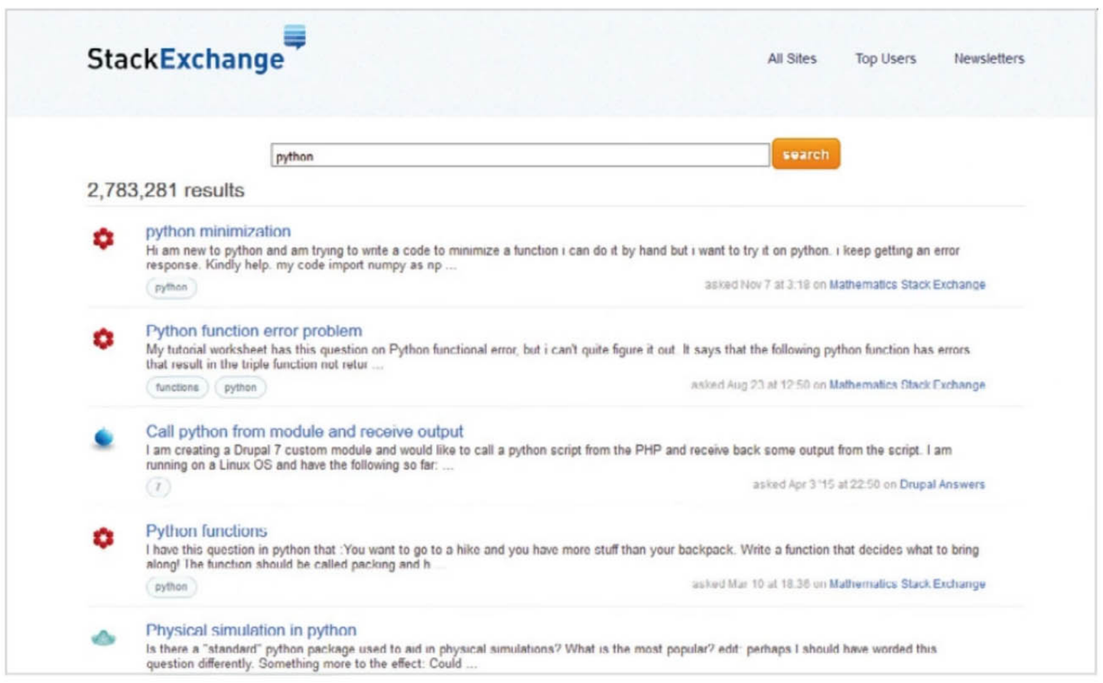
If you've mastered Python fairly well, then turn your attention to C++ or even C#. Still keep your Python skills going but learning a new coding language keeps the old brain ticking over nicely and give you a view into another community, and how they do things differently.

### SHARE SKILLS

Become more active on coding and development knowledge sites, such as StackExchange. If you have the skills to start and help others out, not only will you feel really good for doing so but you can also learn a lot yourself by interacting with other members.

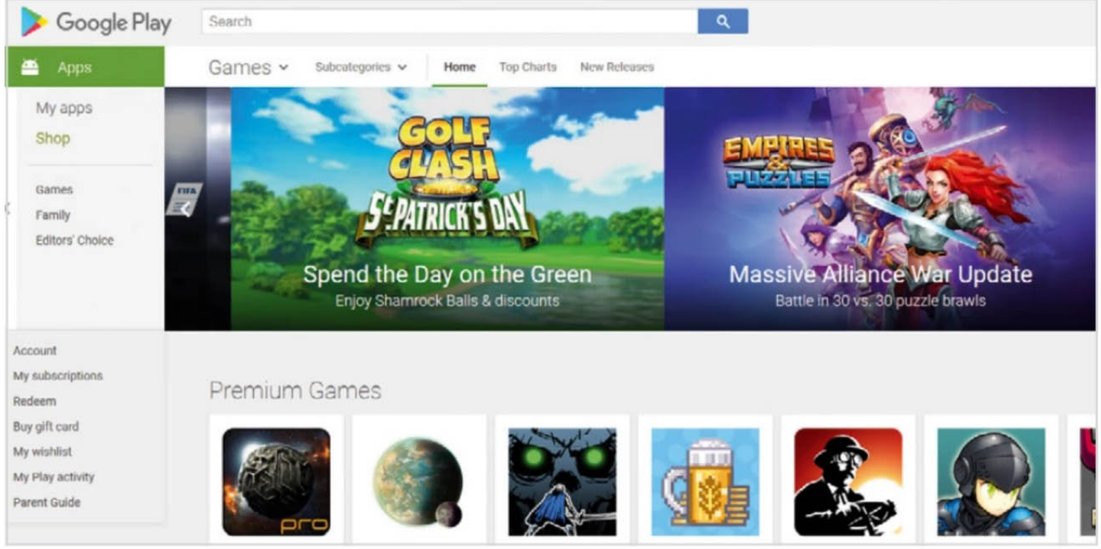


```
#pragma once

#define EXPAND_ME __FILE__ " : " __LINE__

constexpr auto pi = 314LLu;
thread_local decltype(pi) rage = 0b10;

[[deprecated("abc")]] char16_t *f() noexcept { return nullptr; }

template <typename T> struct Foo {
  static_assert(std::is_floating_point<T>::value,
                "Foo<T>: T must be floating point");
};

struct A final : Foo {
  A() = default;
  [[noreturn]] virtual void foo() override;
};

template <typename T> concept bool EqualityComparable = requires(T a
  { a == b } -> bool;
```



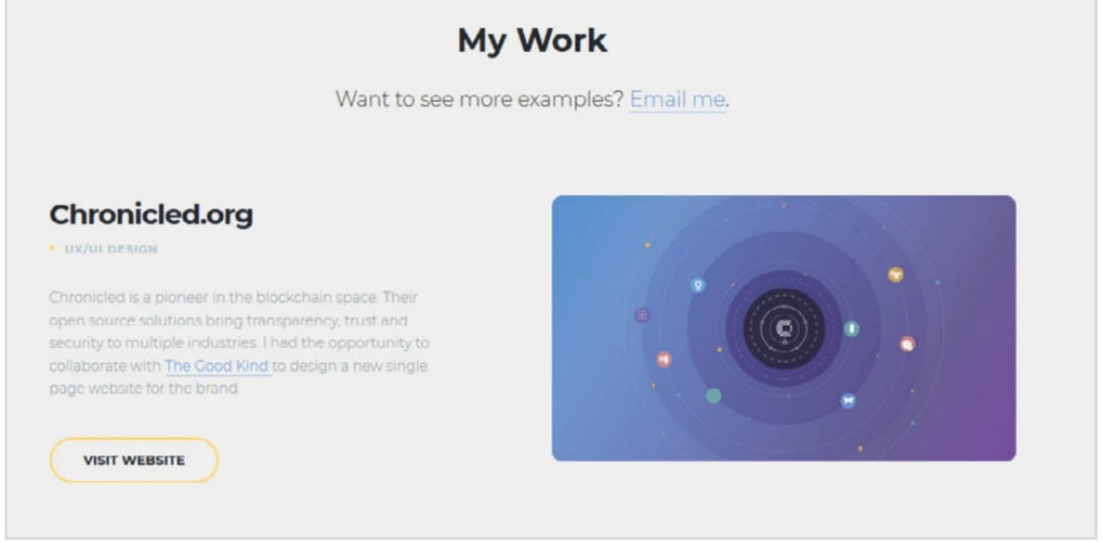
## GOING MOBILE

The mobile market is a great place to test your coding skills and present any games or apps you've created. If your app is good, then who knows, it could be the next great thing to appear on the app stores. It's a good learning experience nevertheless, and something worth considering.



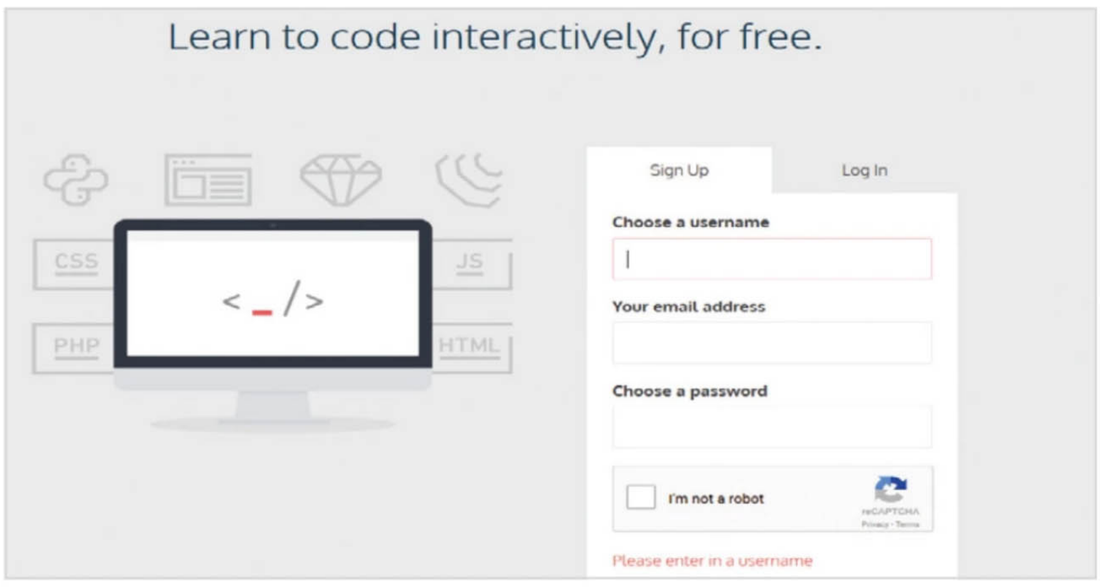

## PORTFOLIOS

If you've learned how to code with an eye for a developer job in the future, then it's worth starting to build up an online portfolio of code. Look at job postings and see what skills they require, then learn and code something with those skills and add it to the portfolio. When it comes to applying, include a link to the portfolio.



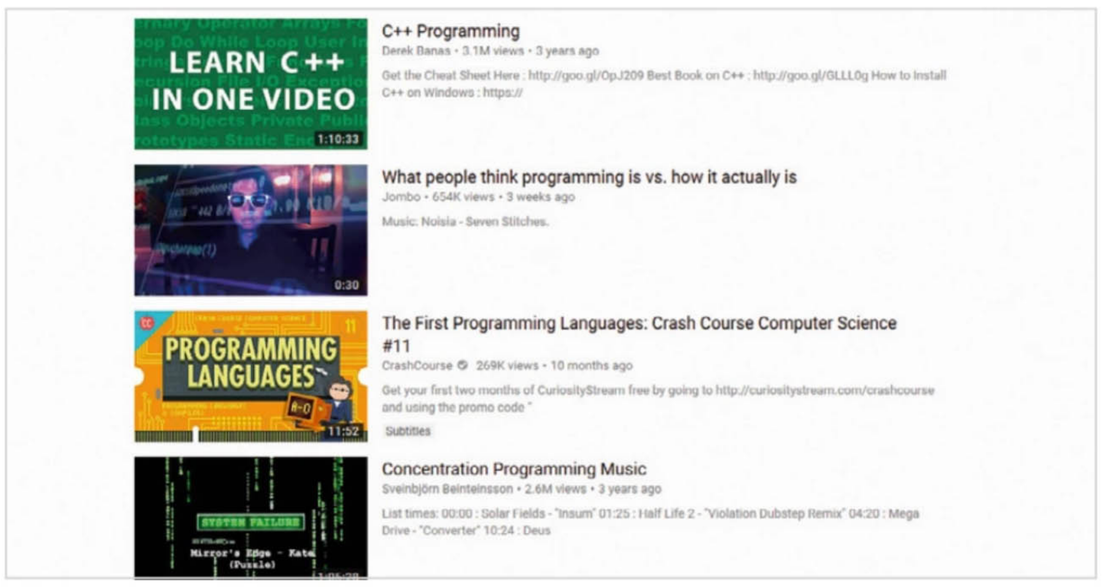

## ONLINE LEARNING

Online courses are good examples of where to take your coding skills next, even if you start from the beginner level again. Often, an online course follows a strict coding convention, so if you're self-taught then it might be worth seeing how other developers lay out their code, and what's considered acceptable.



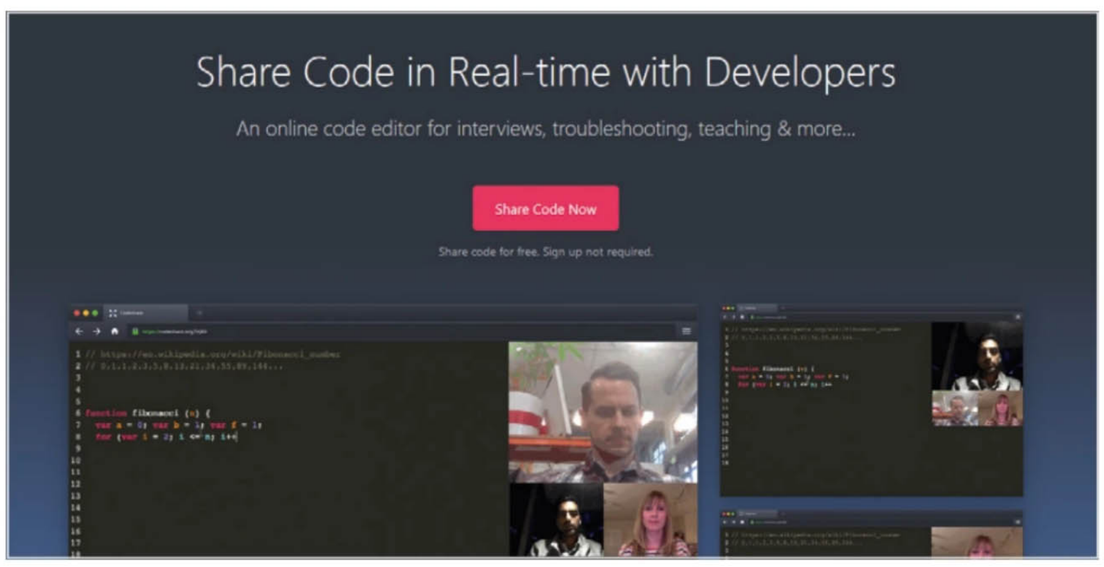

## TEACH CODE

Can you teach? If your coding skills are spot on, consider approaching a college or university to see if they have need for a programming language teacher, perhaps a part-time or evening course. If not teaching, then consider creating your own YouTube how to code channel.




## SHARE CODE

Get sharing, even if you think your code isn't very good. The criticism, advice and comments you receive back help you iron out any issues with your code, and you add them all to your checklist. Alternatively your code might be utterly amazing but you won't know unless you share it.




## HARDWARE PROJECTS

Contributing to hardware projects is a great resource for proving your code with others and learning from other contributors. Many of the developer boards have postings for coders to apply to for hardware projects, using unique code to get the most from the hardware that's being designed.

**“The best programs are written so that computing machines can perform them quickly, and so that human beings can understand them clearly.”**

*– Donald E. Knuth (Computer scientist, mathematician and author)*

# Say Hello to Python

There are many different programming languages available to learn and use. Some are terrifically complex and incredibly powerful, while some are extremely basic and used as minor utilities for the operating system. Python sits somewhere in the middle, combining ease of use with a generous helping of power, allowing the user to create projects ranging from minor utilities through to great games and performance-heavy computational tasks.

However, there's more to Python than simply being another programming language. It has a vibrant and lively community behind it that shares knowledge, code and project ideas, as well as bug fixes for future releases. It's thanks to this community that the language has grown and thrived, so now it's your turn to take the plunge and learn how to program in Python.

This section will get you started with Python and introduce you to this remarkable programming language. Before long, you will be able to code your own helpful systems tools, text adventures and even control a character as they move around the screen.

# Why Python?

There are many different programming languages available for the modern computer, and some still available for older 8 and 16-bit computers too. Some of these languages are designed for scientific work, others for mobile platforms and such. So why choose Python out of all the rest?

## PYTHON POWER

**Ever since the earliest home computers were available, enthusiasts, users and professionals have toiled away until the wee hours, slaving over an overheating heap of circuitry to create something akin to magic.**

These pioneers of programming carved their way into a new frontier, forging small routines that enabled the letter 'A' to scroll across the screen. It may not sound terribly exciting to a generation that's used to ultra high-definition graphics and open world, multi-player online gaming. However, forty-something years ago it was blindingly brilliant.

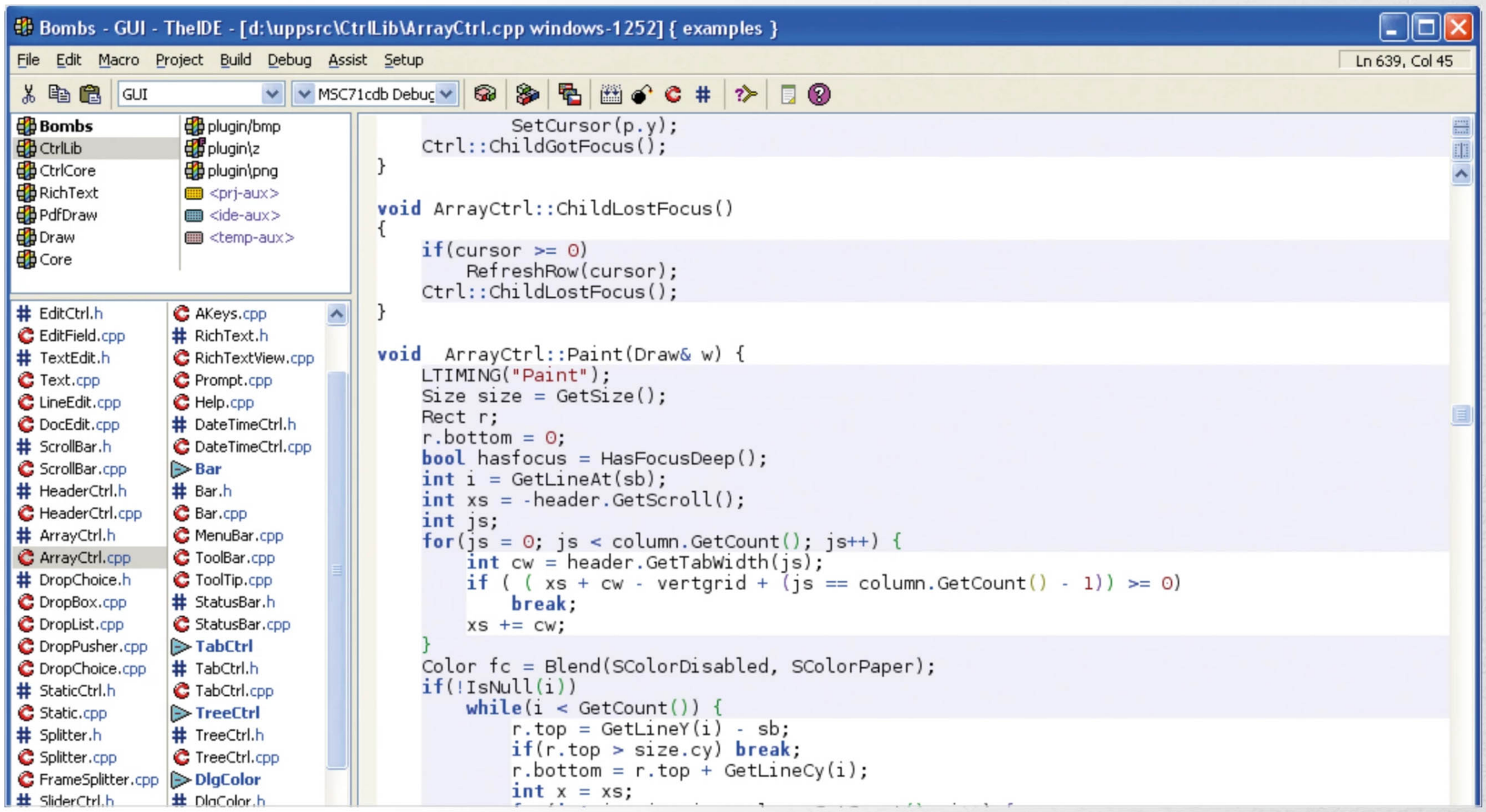
Naturally these bedroom coders helped form the foundations for every piece of digital technology we use today. Some went on to become chief developers for top software companies, whereas others pushed the available hardware to its limits and founded the billion pound gaming empire that continually amazes us.

Regardless of whether you use an Android device, iOS device, PC, Mac, Linux, Smart TV, games console, MP3 player, GPS device built-in to a car, set-top box or a thousand other connected and 'smart' appliances, behind them all is programming.

All those aforementioned digital devices need instructions to tell them what to do, and allow them to be interacted with. These instructions form the programming core of the device and that core can be built using a variety of programming languages.

The languages in use today differ depending on the situation, the platform, the device's use and how the device will interact with its environment or users. Operating systems, such as Windows, macOS and such are usually a combination of C++, C#, assembly and some form of visual-based language. Games generally use C++ whilst web pages can use a plethora of available languages such as HTML, Java, Python and so on.

More general-purpose programming is used to create programs, apps, software or whatever else you want to call them. They're widely used across all hardware platforms and suit virtually every conceivable application. Some operate faster than others and some



are easier to learn and use than others. Python is one such general-purpose language.

Python is what's known as a High-Level Language, in that it 'talks' to the hardware and operating system using a variety of arrays, variables, objects, arithmetic, subroutines, loops and countless more interactions. Whilst it's not as streamlined as a Low-Level Language, which can deal directly with memory addresses, call stacks and registers, its benefit is that it's universally accessible and easy to learn.

Python was created over twenty six years ago and has evolved to become an ideal beginner's language for learning how to program a computer. It's perfect for the hobbyist, enthusiast, student, teacher and those who simply need to create their own unique interaction between either themselves or a piece of external hardware and the computer itself.

Python is free to download, install and use and is available for Linux, Windows, macOS, MS-DOS, OS/2, BeOS, IBM i-series machines, and even RISC OS. It has been voted one of the top five programming languages in the world and is continually evolving ahead of the hardware and Internet development curve.

So to answer the question: why python? Simply put, it's free, easy to learn, exceptionally powerful, universally accepted, effective and a superb learning and educational tool.

```java
//file: Invoke.java
import java.lang.reflect.*;

class Invoke {
  public static void main( String [] args ) {
    try {
      Class c = Class.forName( args[0] );
      Method m = c.getMethod( args[1], new Class
      [] { } );
      Object ret =  m.invoke( null, null );
      System.out.println(
          "Invoked static method: " + args[1]
          + " of class: " + args[0]
          + " with no args\nResults: " + ret );
    } catch ( ClassNotFoundException e ) {
      // Class.forName(  ) can't find the class
    } catch ( NoSuchMethodException e2 ) {
```

Java is a powerful language that's used in web pages, set-top boxes, TVs and even cars.

```
print(HANGMAN[0])
attempts = len(HANGMAN) - 1


while (attempts != 0 and "-" in word_guessed):
    print(("\nYou have {} attempts remaining").format(attempts))
    joined_word = "".join(word_guessed)
    print(joined_word)

    try:
        player_guess = str(input("\nPlease select a letter between A-Z" + "\n> ")).
    except: # check valid input
        print("That is not valid input. Please try again.")
        continue
    else:
        if not player_guess.isalpha(): # check the input is a letter. Also checks a
            print("That is not a letter. Please try again.")
            continue
        elif len(player_guess) > 1: # check the input is only one letter
            print("That is more than one letter. Please try again.")
            continue
        elif player_guess in guessed_letters: # check it letter hasn't been guessed
            print("You have already guessed that letter. Please try again.")
            continue
        else:
            pass

    guessed_letters.append(player_guess)

    for letter in range(len(chosen_word)):
        if player_guess == chosen_word[letter]:
            word_guessed[letter] = player_guess # replace all letters in the chosen

    if player_guess not in chosen_word:
```

Python is a more modern take on BASIC, it's easy to learn and makes for an ideal beginner's programming language.

```
40 LET py=15
70 FOR w=1 TO 10
71 CLS
75 LET by=INT (RND*28)
80 LET bx=0
90 FOR d=1 TO 20
100 PRINT AT px,py;" U "
110 PRINT AT bx,by;"o"
120 IF INKEY$="p" THEN LET py=p
y+1
130 IF INKEY$="o" THEN LET py=p
y-1                U
135 FOR n=1 TO 100: NEXT n
140 IF py<2 THEN LET py=2
150 IF py>27 THEN LET py=27
180 LET bx=bx+1
185 PRINT AT bx-1,by;" "
190 NEXT d
200 IF (by-1)=py THEN LET s=s+1
210 PRINT AT 10,10;"score=";s
220 FOR v=1 TO 1000: NEXT v
300 NEXT w

0 OK, 0:1
```

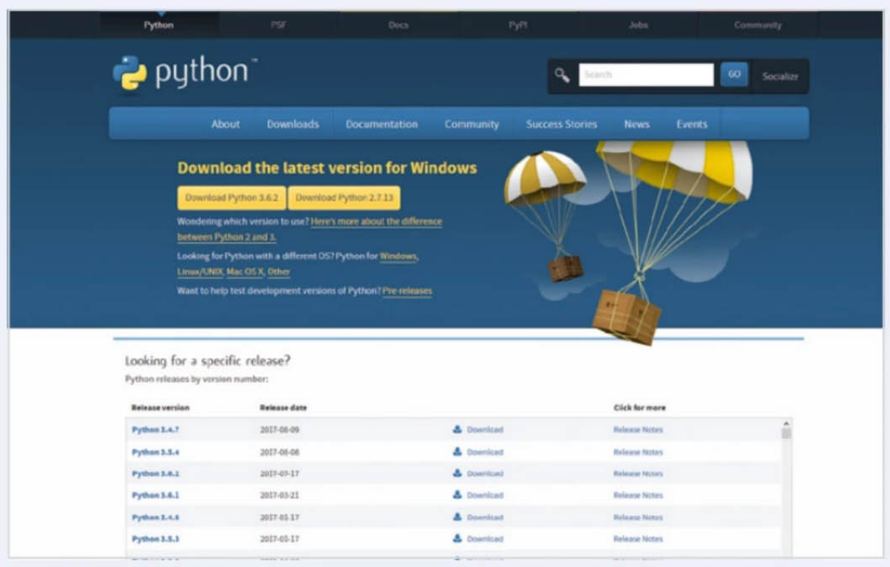
BASIC was once the starter language that early 8-bit home computer users learned.

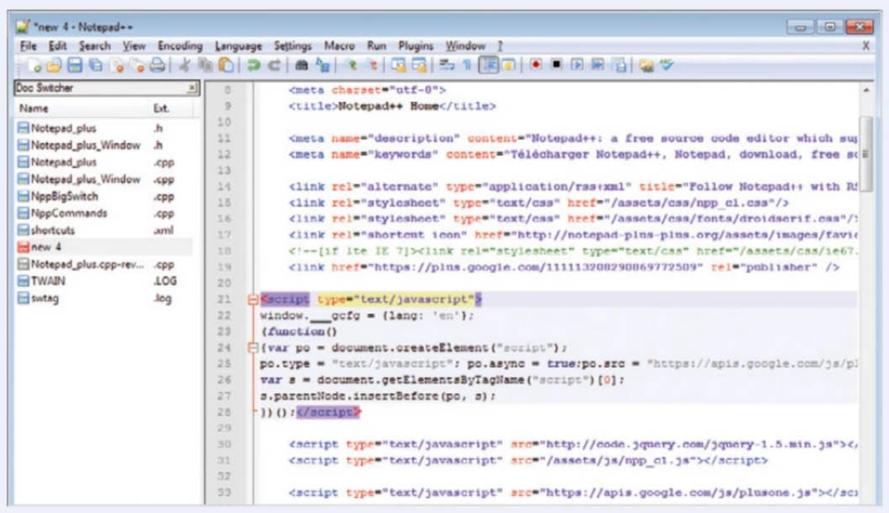
# Equipment You Will Need

You can learn Python with very little hardware or initial financial investment. You don't need an incredibly powerful computer and any software that's required is freely available.

## WHAT WE'RE USING

**Thankfully, Python is a multi-platform programming language available for Windows, macOS, Linux, Raspberry Pi and more. If you have one of those systems, then you can easily start using Python.**




### COMPUTER

Obviously you're going to need a computer in order to learn how to program in Python and to test your code. You can use Windows (from XP onward) on either a 32 or 64-bit processor, an Apple Mac or Linux installed PC.

### AN IDE

An IDE (Integrated Developer Environment) is used to enter and execute Python code. It enables you to inspect your program code and the values within the code, as well as offering advanced features. There are many different IDEs available, so find the one that works for you and gives the best results.

### PYTHON SOFTWARE

macOS and Linux already come with Python preinstalled as part of the operating system, as does the Raspberry Pi. However, you need to ensure that you're running the latest version of Python. Windows users need to download and install Python, which we'll cover shortly.

### TEXT EDITOR

Whilst a text editor is an ideal environment to enter code into, it's not an absolute necessity. You can enter and execute code directly from the IDLE but a text editor, such as Sublime Text or Notepad++, offers more advanced features and colour coding when entering code.

### INTERNET ACCESS

Python is an ever evolving environment and as such new versions often introduce new concepts or change existing commands and code structure to make it a more efficient language. Having access to the Internet will keep you up-to-date, help you out when you get stuck and give access to Python's immense number of modules.

### TIME AND PATIENCE

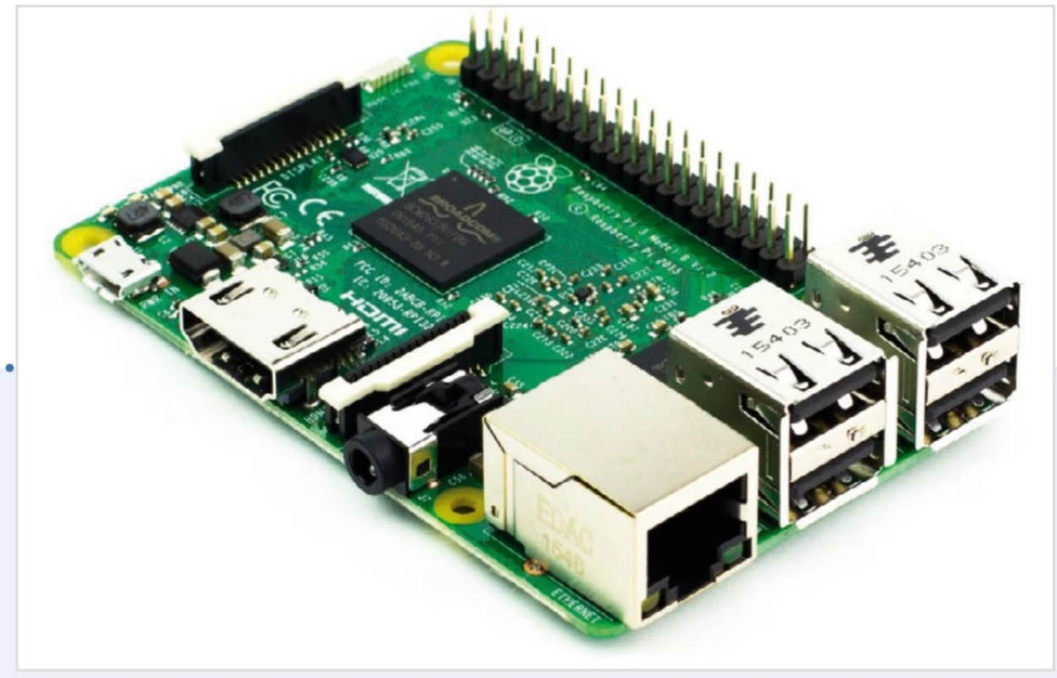
Despite what other books may lead you to believe, you won't become a programmer in 24-hours. Learning to code in Python takes time, and patience. You may become stuck at times and other times the code will flow like water. Understand you're learning something entirely new, and you will get there.

# THE RASPBERRY PI

**Why use a Raspberry Pi? The Raspberry Pi is a tiny computer that's very cheap to purchase but offers the user a fantastic learning platform. Its main operating system, Raspbian, comes preinstalled with the latest Python along with many modules and extras.**

## RASPBERRY PI

The Raspberry Pi 3 is the latest version, incorporating a more powerful CPU, more memory, Wi-Fi and Bluetooth support. You can pick up a Pi for around £32 or as a part of kit for £50+, depending on the kit you're interested in.

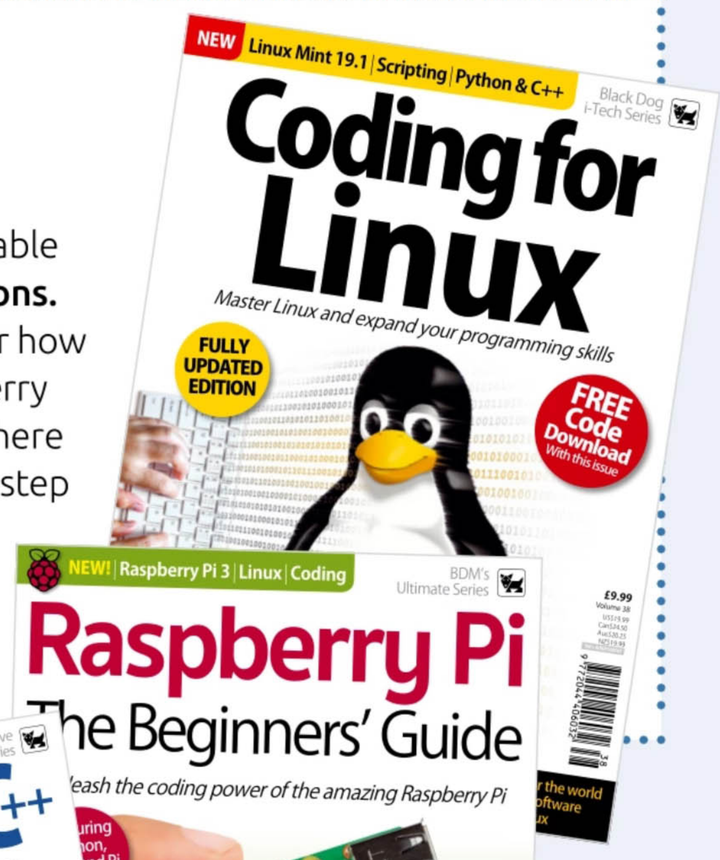


## FUZE PROJECT

The FUZE is a learning environment built on the latest model of the Raspberry Pi. You can purchase the workstations that come with an electronics kit and even a robot arm for you to build and program. You can find more information on the FUZE at **www.fuze.co.uk**.

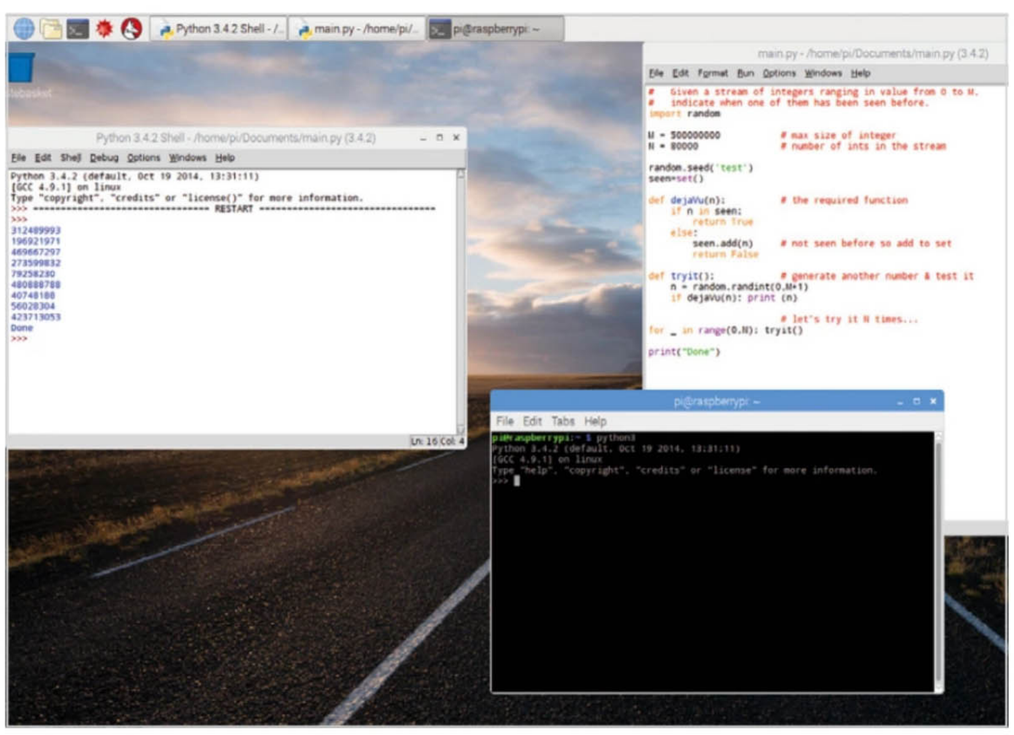
## BOOKS

We have several great Raspberry Pi titles available via **www.bdmpublications.com**. Our Pi books cover how to buy your first Raspberry Pi, set it up and use it; there are some great step-by-step project examples and guides to get the most from the Raspberry Pi too.



## RASPBIAN

The Raspberry Pi's main operating system is a Debian-based Linux distribution that comes with everything you need in a simple to use package. It's streamlined for the Pi and is an ideal platform for hardware and software projects, Python programming and even as a desktop computer.



# Getting to Know Python

Python is the greatest computer programming language ever created. It enables you to fully harness the power of a computer, in a language that's clean and easy to understand.

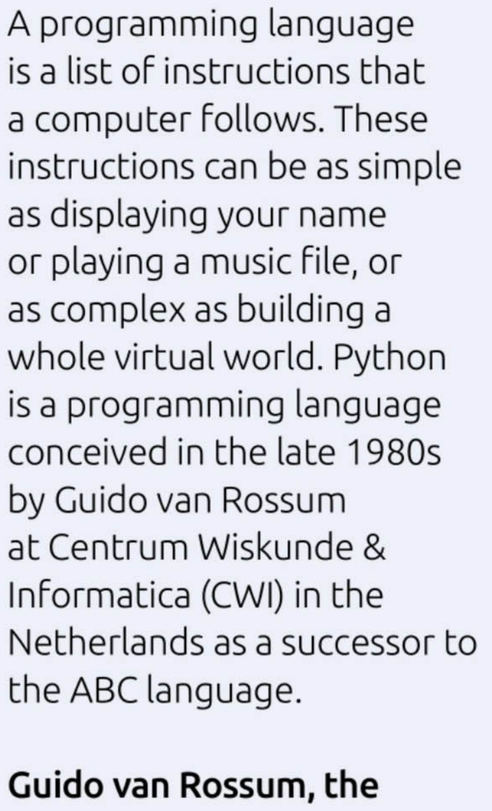
## WHAT IS PROGRAMMING?

**It helps to understand what a programming language is before you try to learn one, and Python is no different. Let's take a look at how Python came about and how it relates to other languages.**

### PYTHON

A programming language is a list of instructions that a computer follows. These instructions can be as simple as displaying your name or playing a music file, or as complex as building a whole virtual world. Python is a programming language conceived in the late 1980s by Guido van Rossum at Centrum Wiskunde & Informatica (CWI) in the Netherlands as a successor to the ABC language.

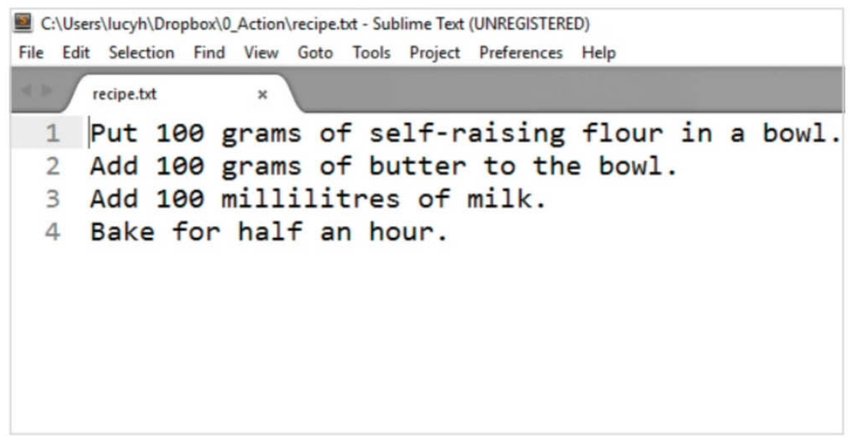
**Guido van Rossum, the father of Python.**



## PROGRAMMING RECIPES

Programs are like recipes for computers. A recipe to bake a cake could go like this:

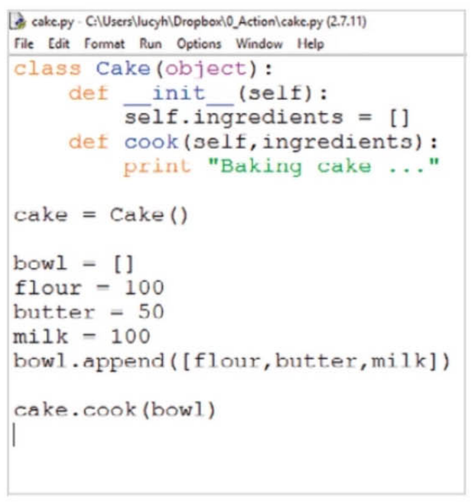
Put 100 grams of self-raising flour in a bowl.
Add 100 grams of butter to the bowl.
Add 100 millilitres of milk.
Bake for half an hour.




## CODE

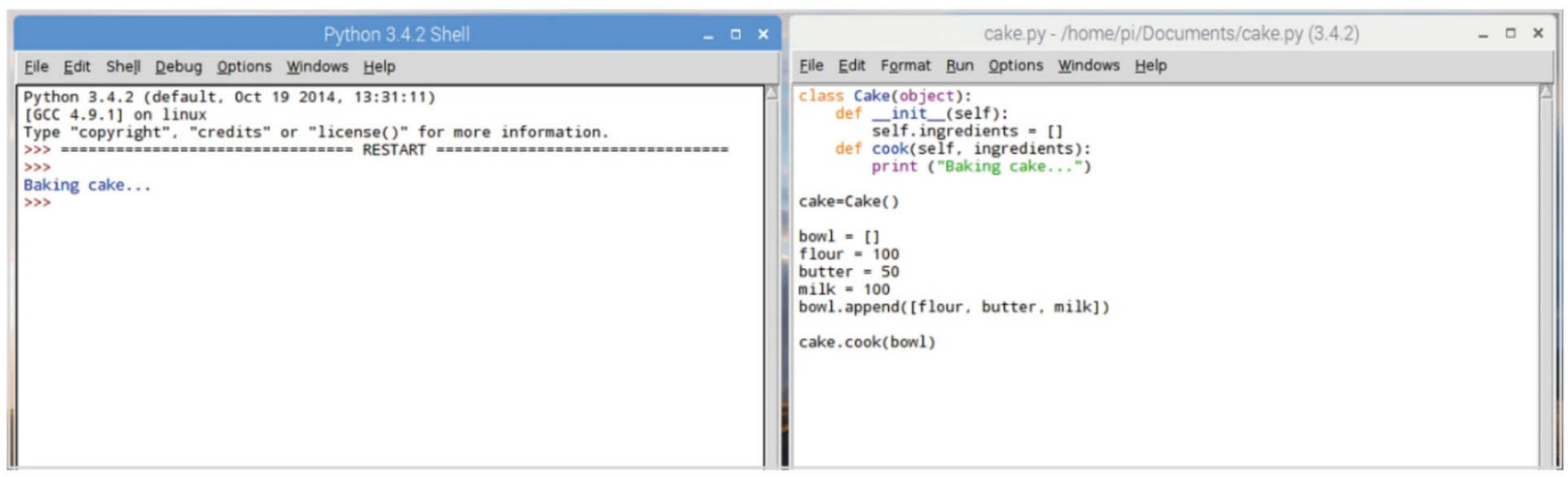
Just like a recipe, a program consists of instructions that you follow in order. A program that describes a cake might run like this:

```
bowl = []
flour = 100
butter = 50
milk = 100
bowl.append([flour,butter,milk])
cake.cook(bowl)
```



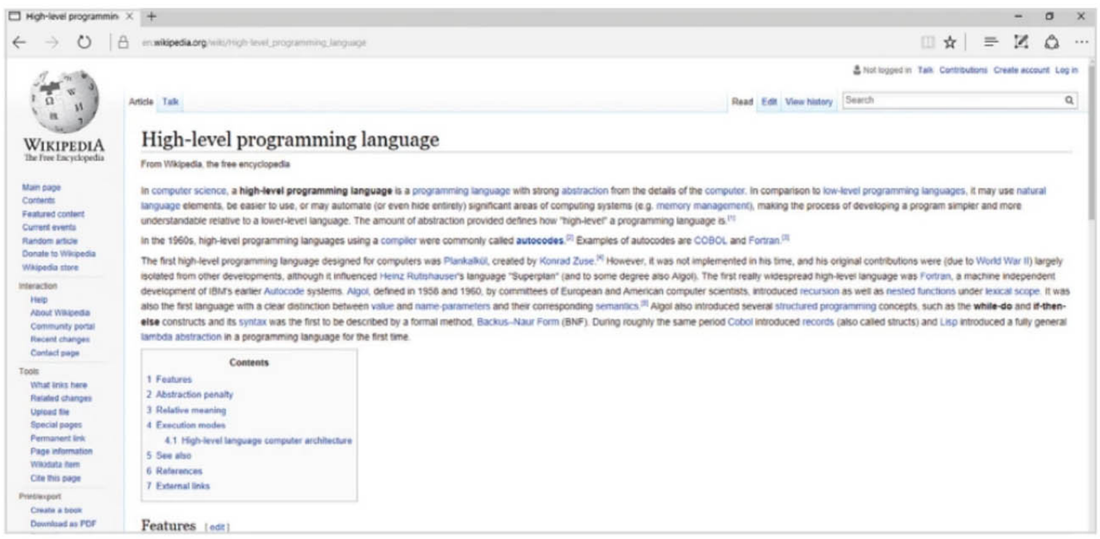

## PROGRAM COMMANDS

You might not understand some of the Python commands, like bowl.append and cake.cook(bowl). The first is a list, the second an object; we'll look at both in this book. The main thing to know is that it's easy to read commands in Python. Once you learn what the commands do, it's easy to figure out how a program works.



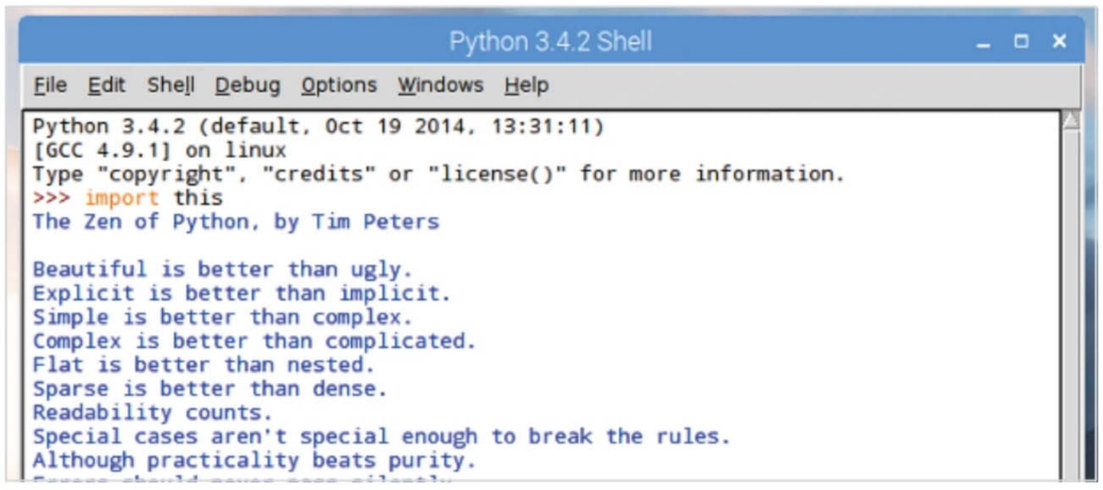
## HIGH-LEVEL LANGUAGES

Computer languages that are easy to read are known as “high-level”. This is because they fly high above the hardware (also referred to as “the metal”). Languages that “fly close to the metal,” like Assembly, are known as “low-level”. Low-level languages commands read a bit like this: `msg db ,0xa len equ $ - msg`.




## ZEN OF PYTHON

Python lets you access all the power of a computer in a language that humans can understand. Behind all this is an ethos called “The Zen of Python.” This is a collection of 20 software principles that influences the design of the language. Principles include “Beautiful is better than ugly” and “Simple is better than complex.” Type `import this` into Python and it will display all the principles.




## PYTHON 3 VS PYTHON 2

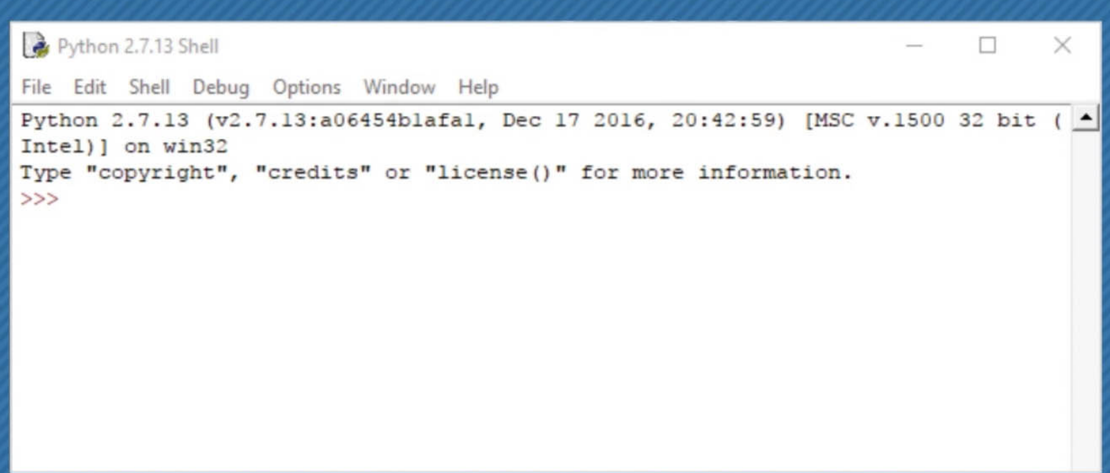
**In a typical computing scenario, Python is complicated somewhat by the existence of two active versions of the language: Python 2 and Python 3.**

### WORLD OF PYTHON

Python 3.7 is the newest release of the programming language. However, if you dig a little deeper into the Python site, and investigate Python code online, you will undoubtedly come across Python 2. Although you can run Python 3 and Python 2 alongside each other, it’s not recommended. Always opt for the latest stable release as posted by the Python website.

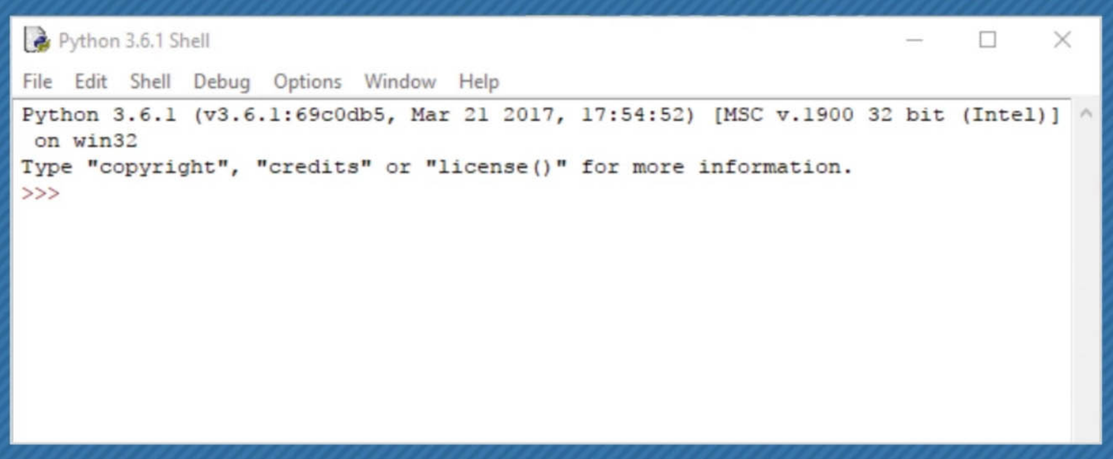
### PYTHON 2.X

So why two? Well, Python 2 was originally launched in 2000 and has since then adopted quite a large collection of modules, scripts, users, tutorials and so on. Over the years Python 2 has fast become one of the first go to programming languages for beginners and experts to code in, which makes it an extremely valuable resource.



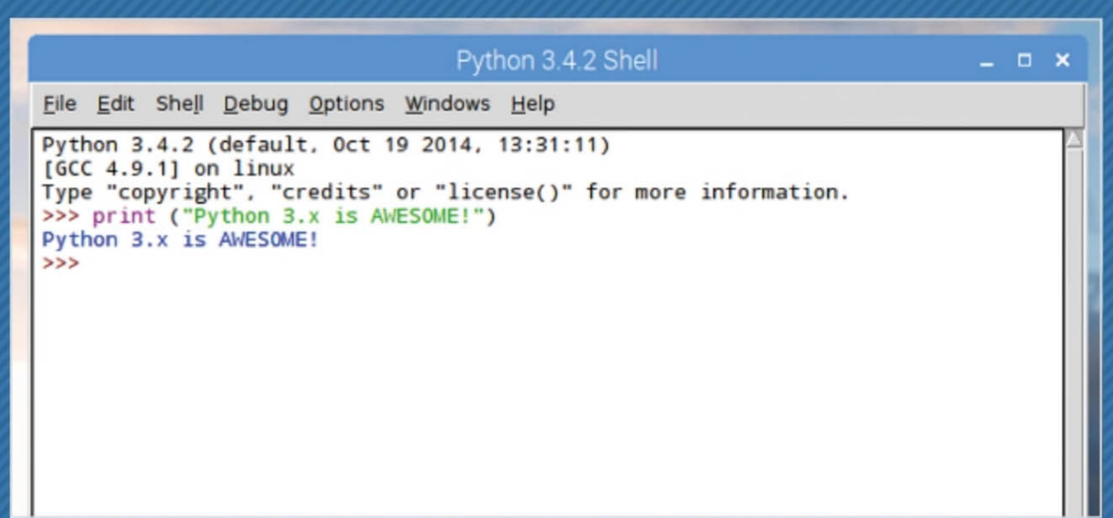

### PYTHON 3.X

In 2008 Python 3 arrived with several new and enhanced features. These features provide a more stable, effective and efficient programming environment but sadly, most (if not all) of these new features are not compatible with Python 2 scripts, modules and tutorials. Whilst not popular at first, Python 3 has since become the cutting edge of Python programming.




### 3.X WINS

Python 3’s growing popularity has meant that it’s now prudent to start learning to develop with the new features and begin to phase out the previous version. Many development companies, such as SpaceX and NASA use Python 3 for snippets of important code.



# How to Set Up Python in Windows

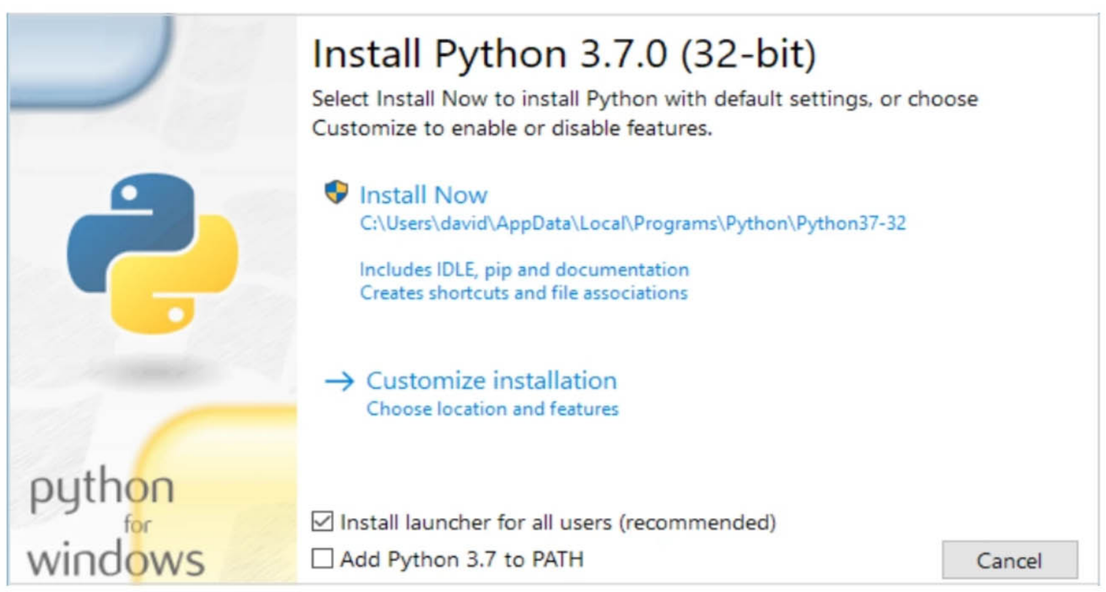
Windows users can easily install the latest version of Python via the main Python Downloads page. While most seasoned Python developers may shun Windows as the platform of choice for building their code, it's still an ideal starting point for beginners.

## INSTALLING PYTHON 3.X

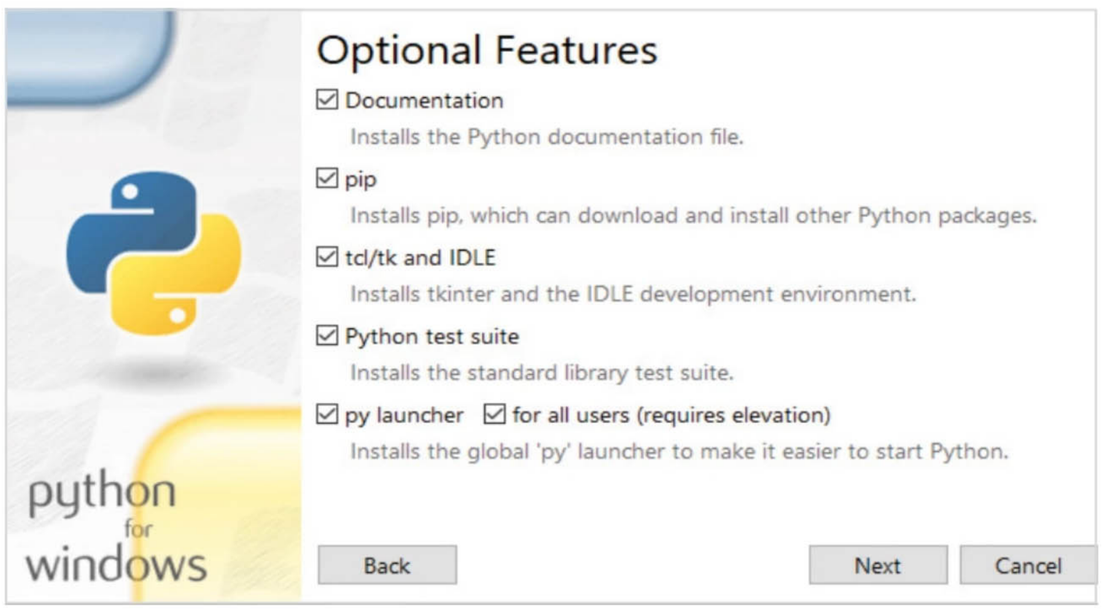
**Microsoft Windows doesn't come with Python preinstalled as standard, so you're going to have to install it yourself manually. Thankfully, it's an easy process to follow.**

**STEP 1** Start by opening your web browser to **www.python.org/downloads/**. Look for the button detailing the download link for Python 3.x. The latest version at the time of writing is 3.7.0 but as Python is frequently updated this may be a different version for you.

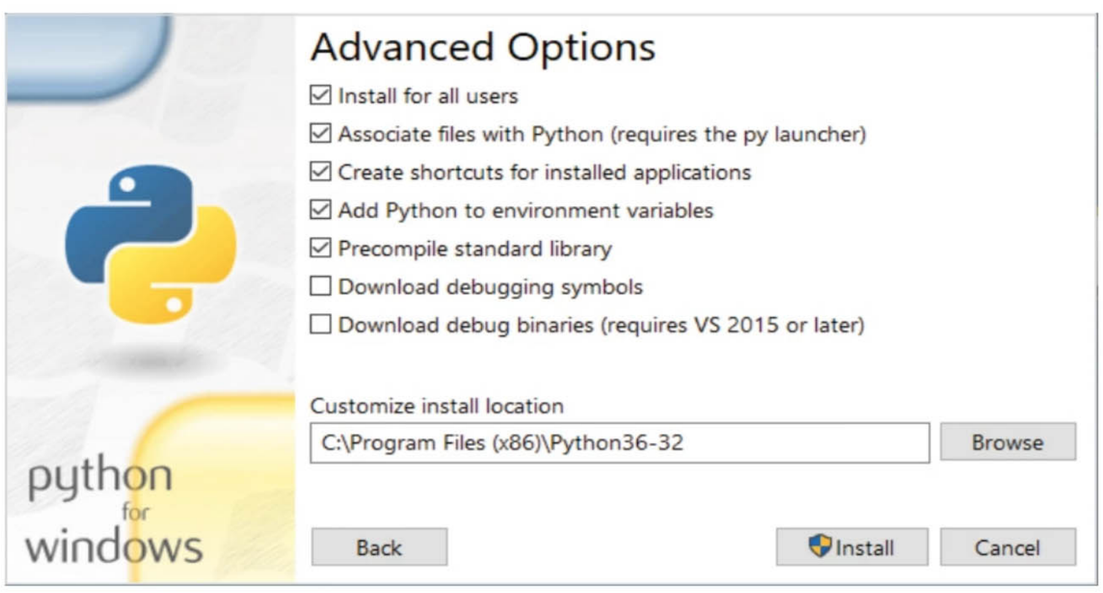
**STEP 2** Click the Download button for version 3.x, and save the file to your Downloads folder. When the file is downloaded, double-click the executable and the Python installation wizard will launch. From here you have two choices: Install Now and Customise Installation. We recommend opting for the Customise Installation link.



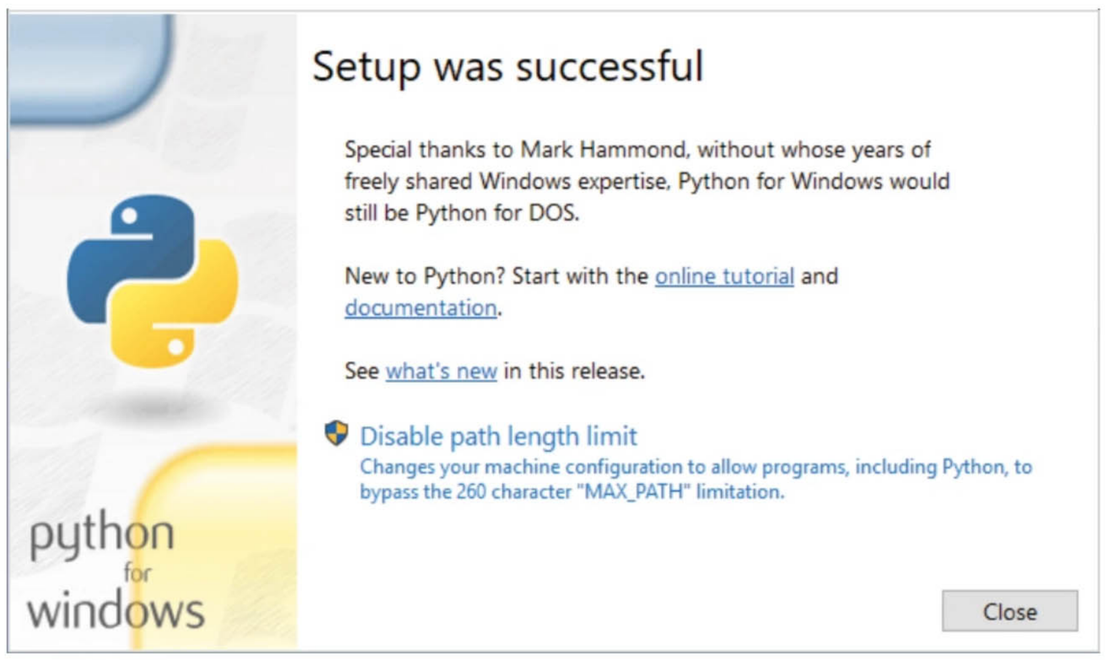

**STEP 3** Choosing the Customise option allows you to specify certain parameters, and whilst you may stay with the defaults, it's a good habit to adopt as sometimes (not with Python, thankfully) installers can include unwanted additional features. On the first screen available, ensure all boxes are ticked and click the Next button.



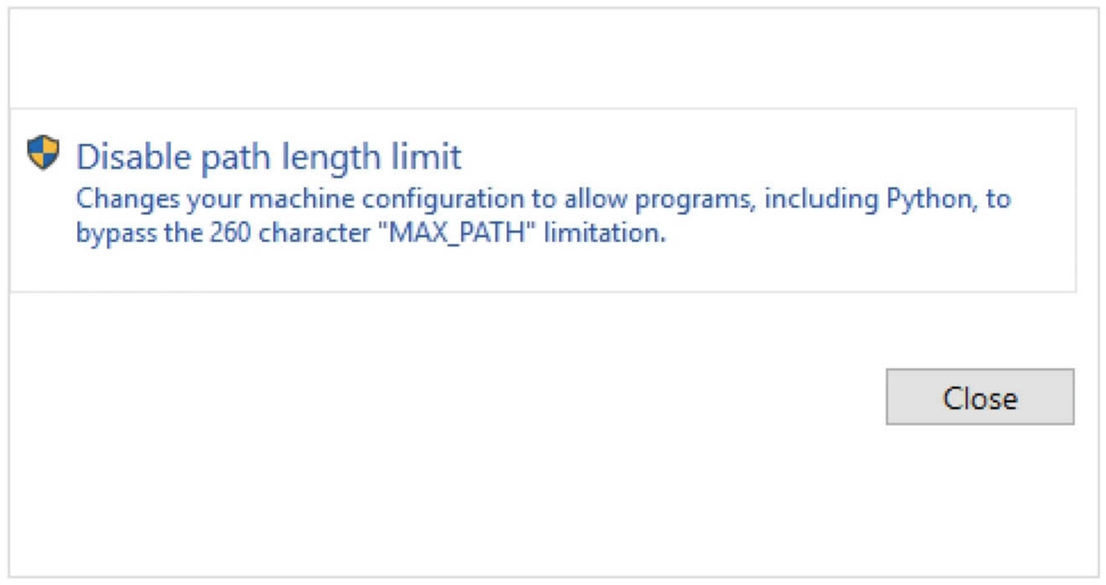

**STEP 4** The next page of options include some interesting additions to Python. Ensure the Associate file with Python, Create Shortcuts, Add Python to Environment Variables, Precompile Standard Library and Install for All Users options are ticked. These make using Python later much easier. Click Install when you're ready to continue.



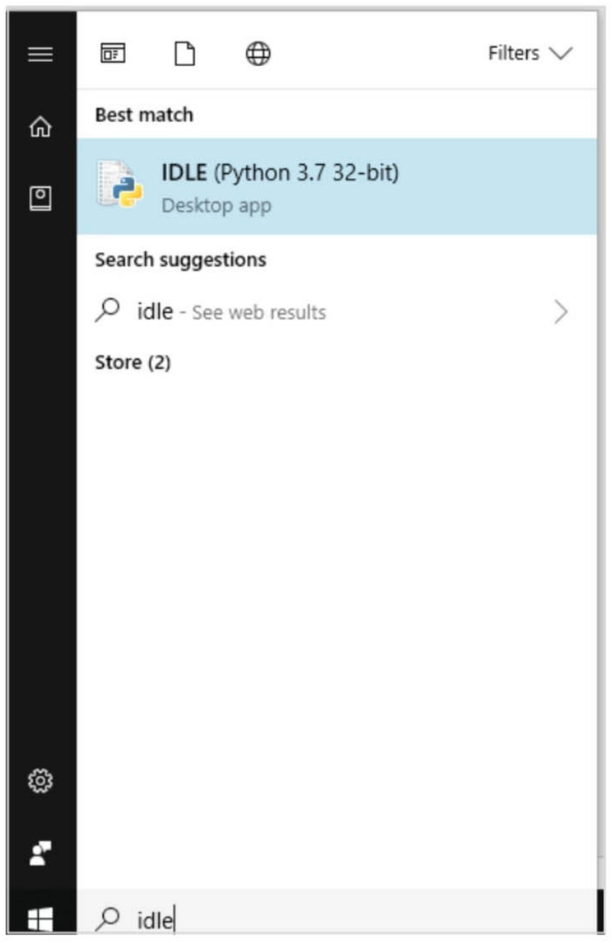
**STEP 5** You may need to confirm the installation with the Windows authentication notification. Simply click Yes and Python will begin to install. Once the installation is complete the final Python wizard page will allow you to view the latest release notes, and follow some online tutorials.



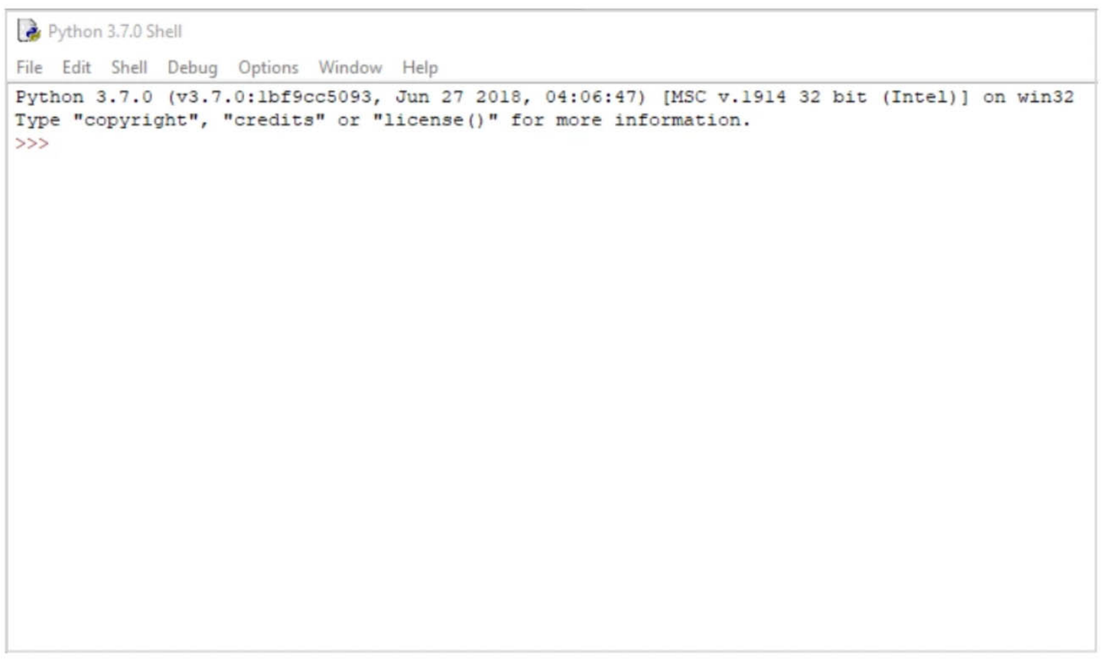

**STEP 6** Before you close the install wizard window, however, it's best to click on the link next to the shield detailed Disable Path Length Limit. This will allow Python to bypass the Windows 260 character limitation, enabling you to execute Python programs stored in deep folders arrangements. Again, click Yes to authenticate the process; then you can Close the installation window.



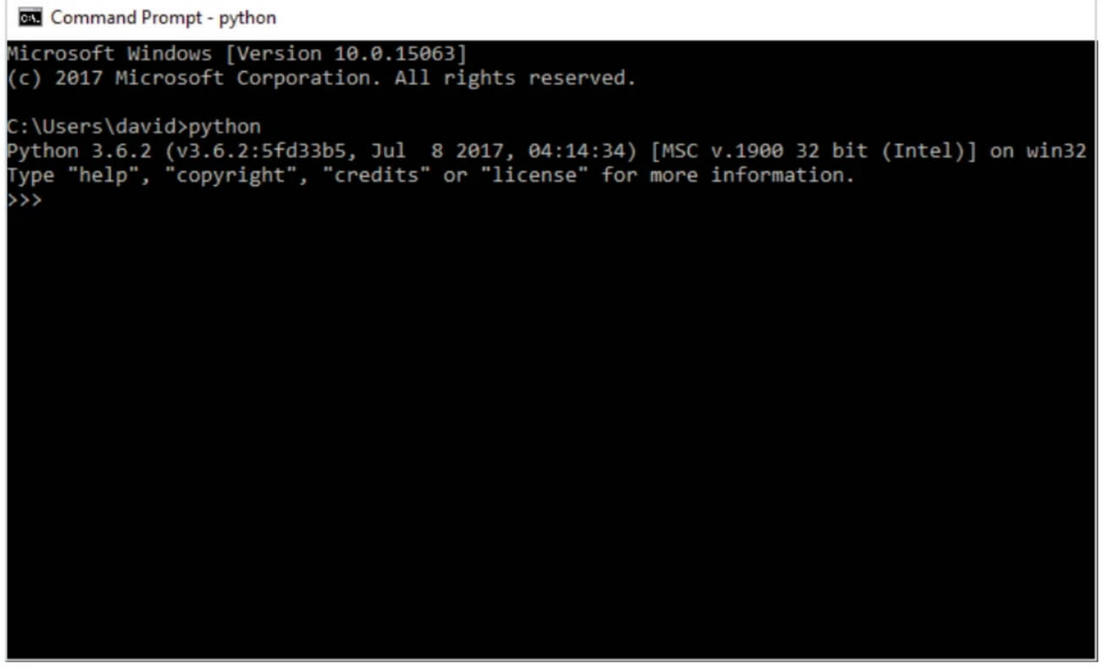

**STEP 7** Windows 10 users can now find the installed Python 3.x within the Start button Recently Added section. The first link, Python 3.7 (32-bit) will launch the command line version of Python when clicked (more on that in a moment). To open the IDLE, type IDLE into Windows start.



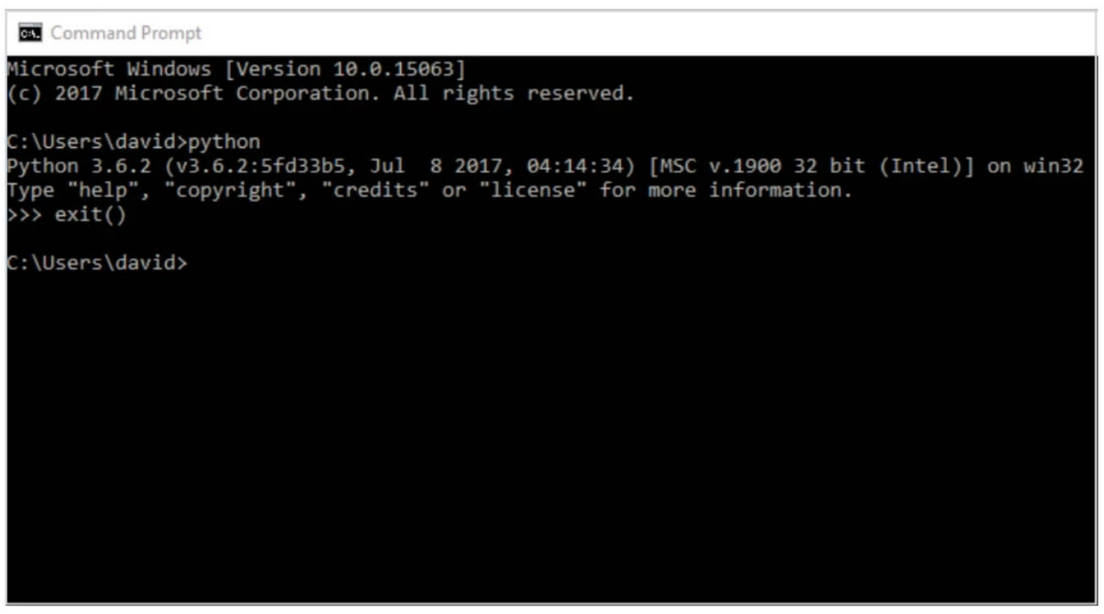

**STEP 8** Clicking on the IDLE (Python 3.7 32-bit) link will launch the Python Shell, where you can begin your Python programming journey. Don't worry if your version is newer as long as it's Python 3.x our code works inside your Python 3 interface.




**STEP 9** If you now click on the Windows Start button again, and this time type: `CMD`, you'll be presented with the Command Prompt link. Click it to get to the Windows command line environment. To enter Python within the command line, you need to type: `python` and press Enter.




**STEP 10** The command line version of Python works in much the same way as the Shell you opened in Step 8; note the three left-facing arrows (>>>). Whilst it's a perfectly fine environment, it's not too user-friendly, so leave the command line for now. Enter: `exit()` to leave and close the Command Prompt window.



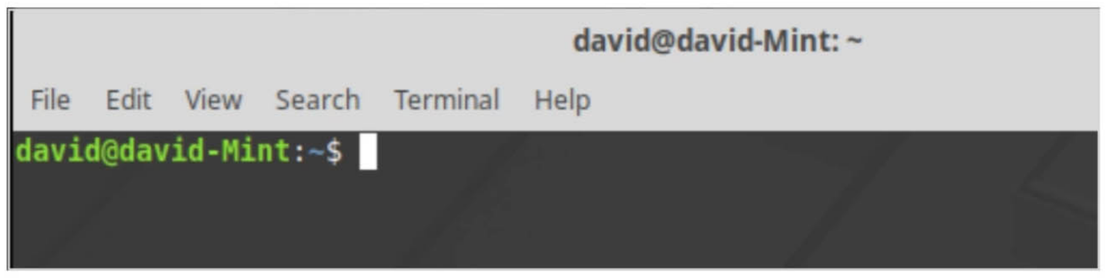
# How to Set Up Python in Linux

While the Raspberry Pi's operating system contains the latest, stable version of Python, other Linux distros don't come with Python 3 pre-installed. If you're not going down the Pi route, then here's how to check and install Python for Linux.

## PYTHON PENGUIN

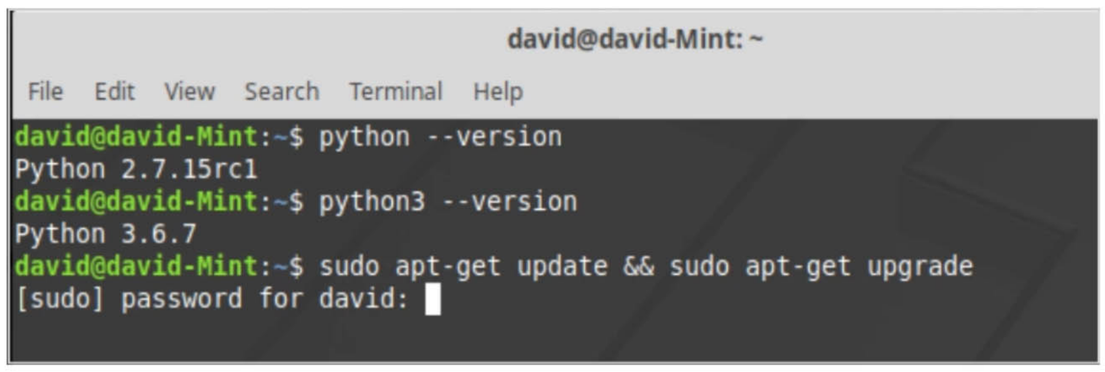
**Linux is such a versatile operating system that it's often difficult to nail down just one way of doing something. Different distributions go about installing software in different ways, but for this particular tutorial, we will stick to Linux Mint.**

**STEP 1** First you need to ascertain which version of Python is currently installed in your Linux system. To begin with, drop into a Terminal session from your distro's menu, or hit the Ctrl+Alt+T keys.

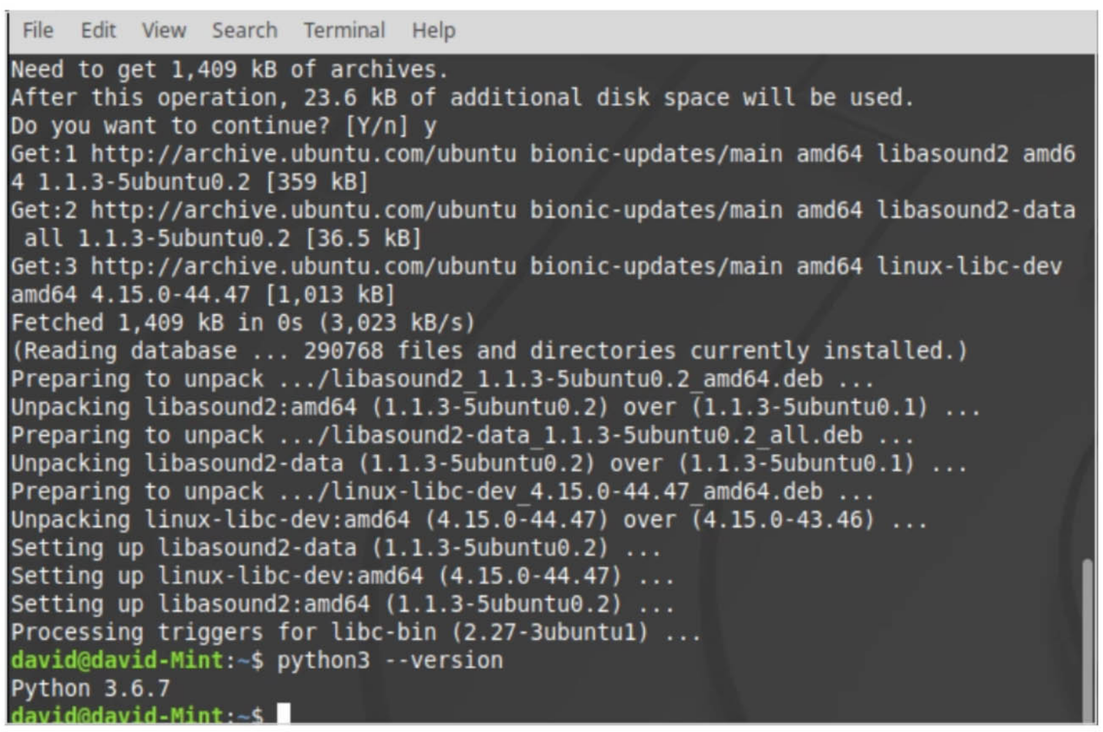



**STEP 2** Next enter: `python --version` into the Terminal screen. You should have the output relating to version 2.x of Python, in the display. By default, most Linux distros come with both Python 2 and 3, as there's plenty of code out there still available for Python 2. Now enter: `python3 --version`.

**STEP 3** In our case we have both Python 2 and 3 installed. As long as Python 3.x.x is installed, then the code in our tutorials will work. It's always worth checking to see if the distro has been updated with the latest versions, enter: `sudo apt-get update && sudo apt-get upgrade` to update the system.

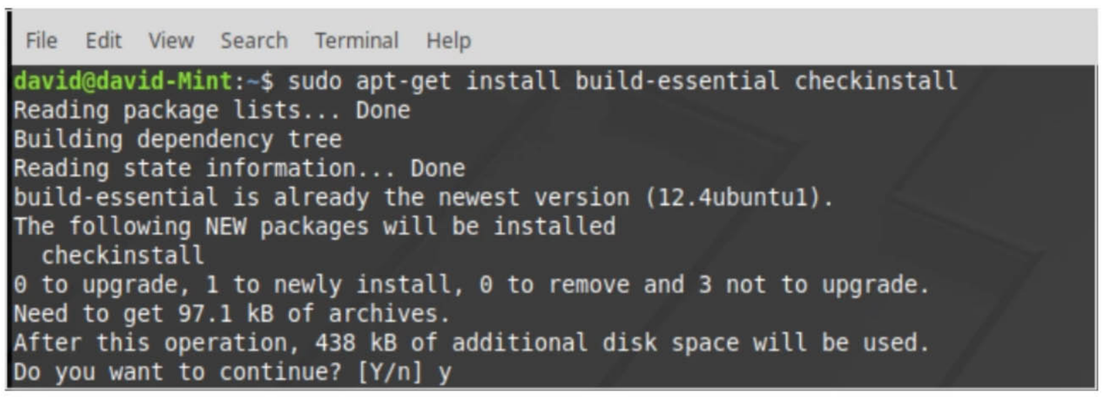



**STEP 4** Once the update and upgrade is complete, enter: `python3 --version` again to see if Python 3.x is updated or even installed. As long as you have Python 3.x, you're running the most recent major version. The numbers after the 3. indicate patches and further updates. Often they're unnecessary, but can contain vital new elements.

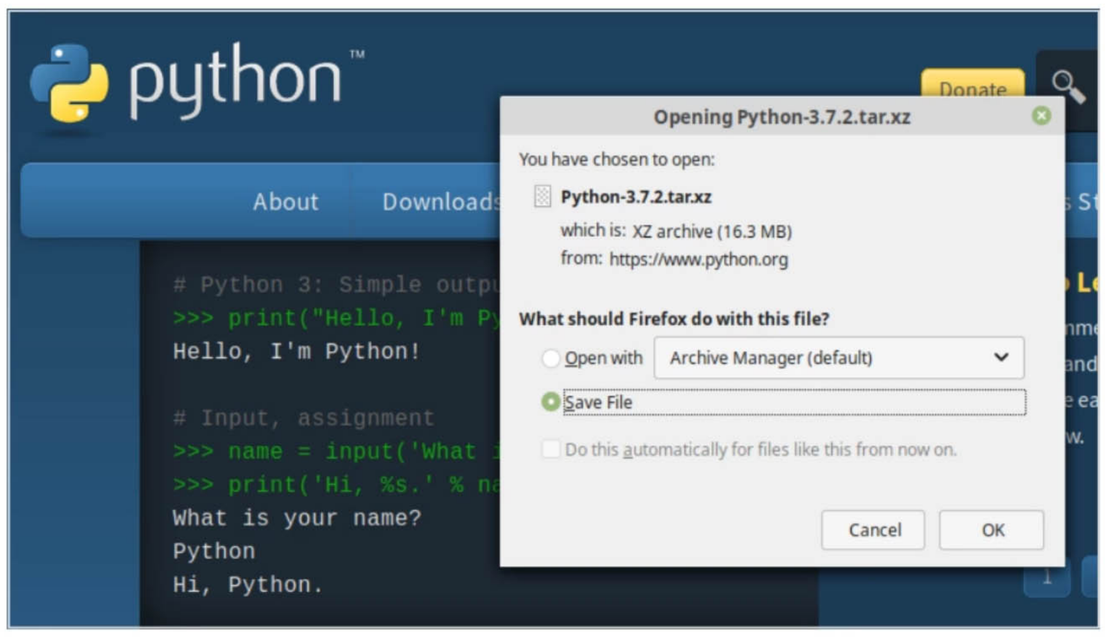



**STEP 5** However, if you want the latest, cutting edge version, you'll need to build Python from source. Start by entering these commands into the Terminal:

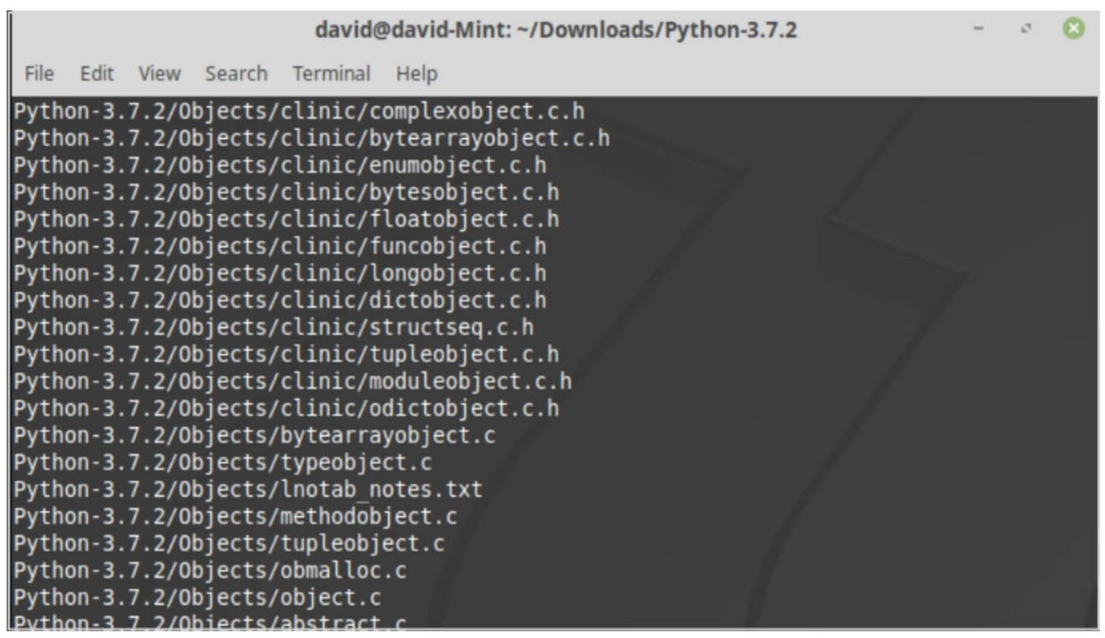
```
sudo apt-get install build-essential checkinstall
sudo apt-get install libreadline-gplv2-dev
libncursesw5-dev libssl-dev libsqlite3-dev tk-dev
libgdbm-dev libc6-dev libbz2-dev
```



**STEP 6** Open up your Linux web browser and go to the Python download page: **https://www.python.org/downloads**. Click on the Downloads followed by the button under the Python Source window. This opens a download dialogue box, choose a location and start the download process.



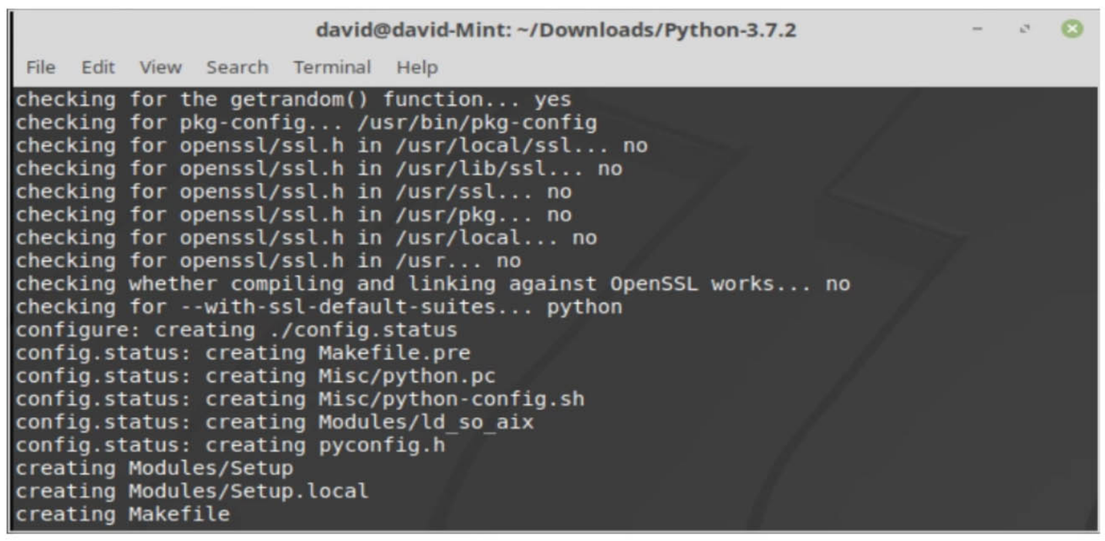

**STEP 7** In the Terminal, go to the Downloads folder by entering: `cd Downloads/`. Then unzip the contents of the downloaded Python source code using: `tar -xvf Python-3.Y.Y.tar.xz`. (replace the Y's with the version numbers you've downloaded). To access the newly unzipped folder, enter: `cd Python-3.Y.Y/`.




**STEP 8** Within the Python folder, enter:

```
./configure
sudo make altinstall
```

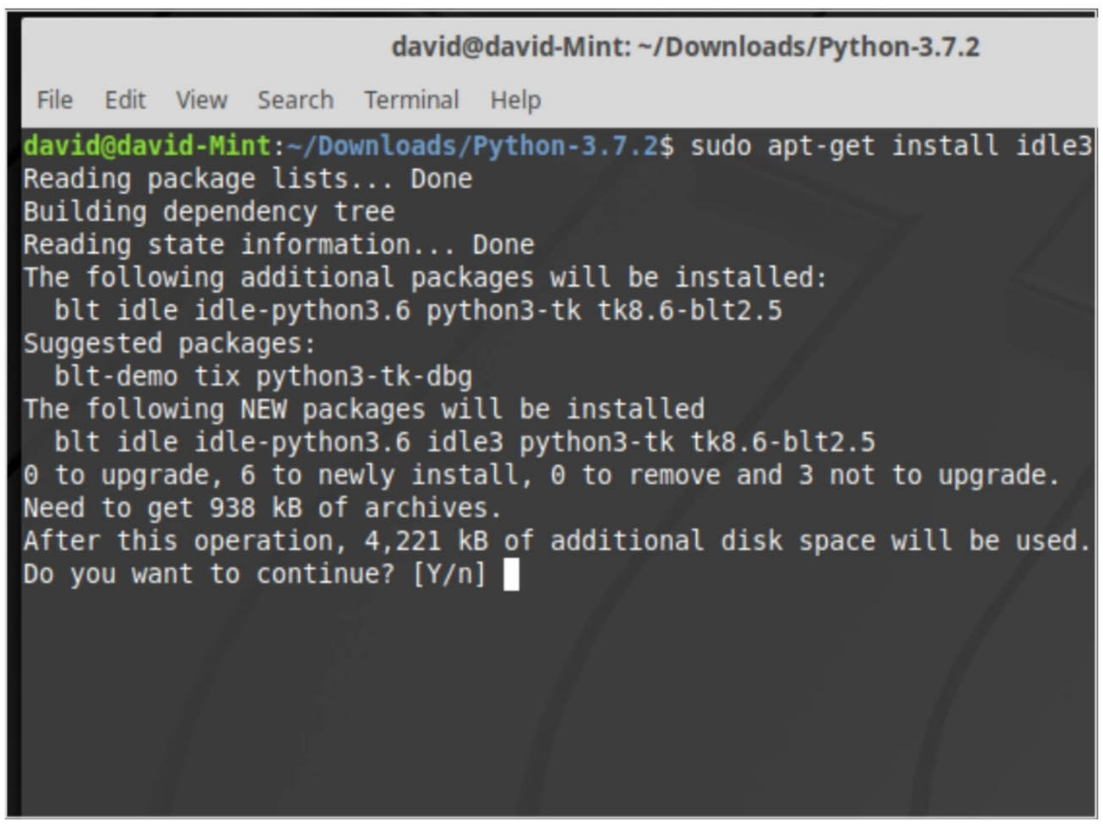
This could take a while, depending on the speed of your computer. Once finished, enter: `python3.7 --version` to check the latest installed version. You should now have Python 3.7 installed, alongside older Python 3.x.x and Python 2.




**STEP 9** For the GUI IDLE, you'll need to enter the following command into the Terminal:

```
sudo apt-get install idle3
```

The IDLE can then be started with the command: `idle3`. Note that IDLE runs a different version from the one you installed from source.

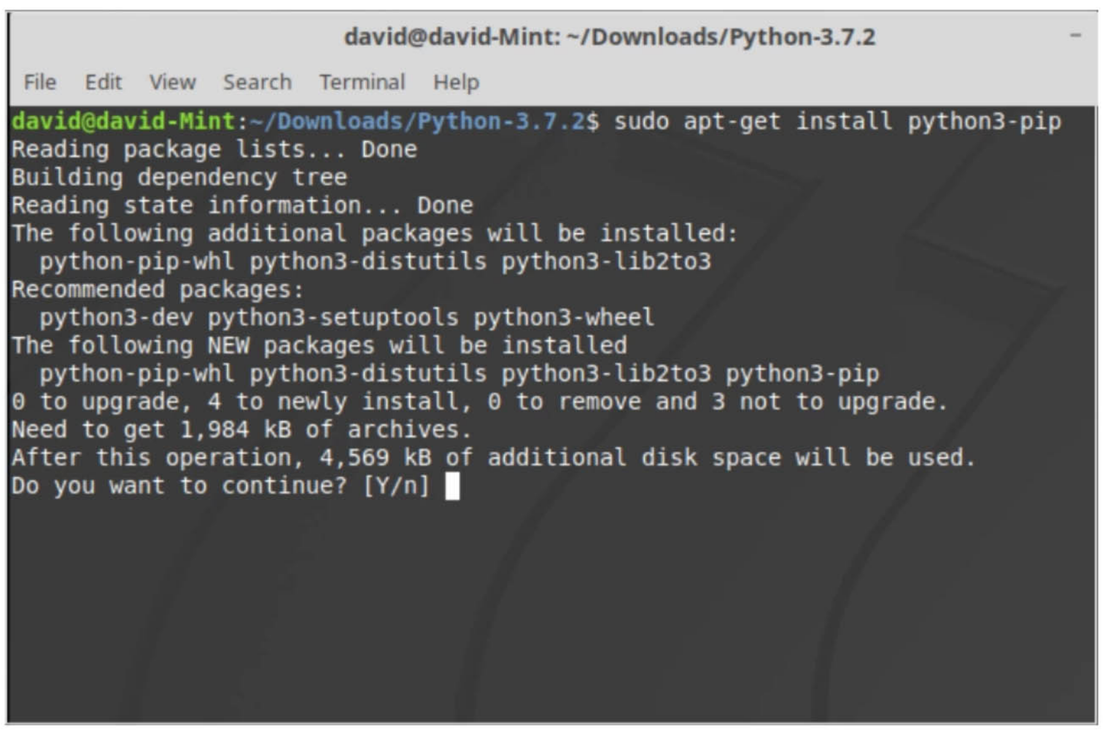



**STEP 10** You'll also need PIP (Pip Installs Packages), which is a tool to help you install more modules and extras. Enter: `sudo apt-get install python3-pip`

PIP is then installed, check for the latest update with:

```
pip3 install --upgrade pip
```

When complete, close the Terminal and Python 3.x will be available via the Programming section in your distro's menu.




## PYTHON ON MACOS

Installation of Python on macOS can be done in much the same way as the Windows installation. Simply go to the Python webpage, hover your mouse pointer over the Downloads link and select Mac OS X from the options. You will then be guided to the Python releases for Mac versions and the necessary installers for macOS 64-bit, for OS X 10.9 and later.

**"I have always wished for my computer to be as easy to use as my telephone; my wish has come true, because I can no longer figure out how to use my telephone."**

*– Bjarne Stroustrup (Developer and creator of C++)*

# Getting Started with Python

Learning how to code may seem a little daunting at first. Thankfully, the Python language has been designed with simplicity in mind. Like most things, you need to start slow, learn how to get a result and how to get what you want from the code.

In this section, we will cover the core concepts: saving and executing your code, variables, numbers and expressions, user input, conditions and loops.

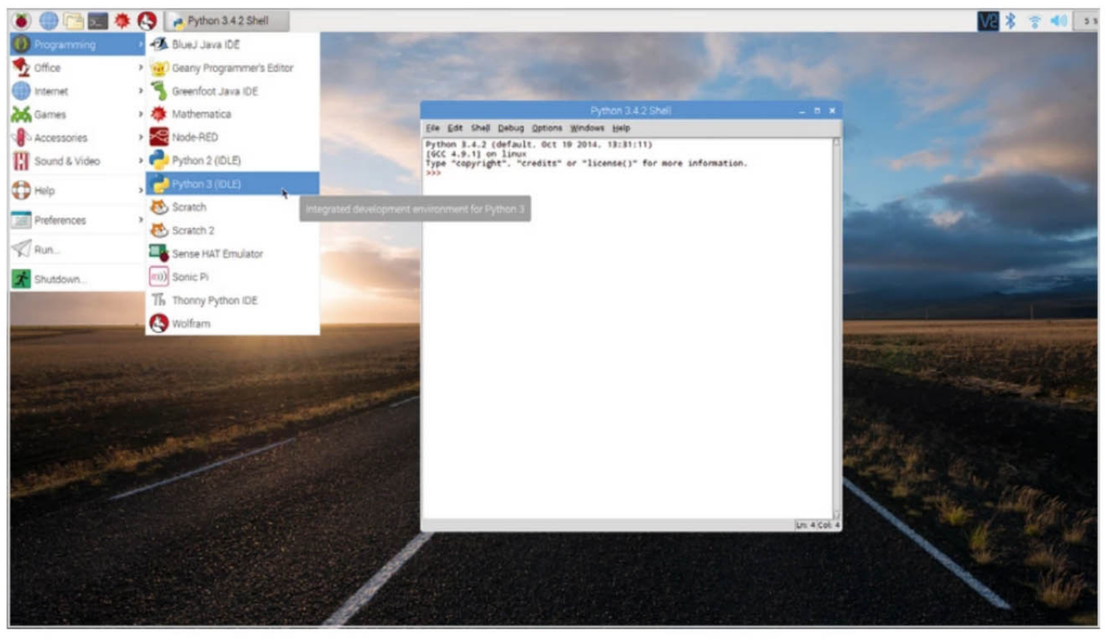
# Starting Python for the First Time

The Raspberry Pi offers one of the best all-round solutions for learning and coding on, in particular, Python. Raspbian, the Pi's recommended OS, come pre-installed with the latest stable version of Python 3, which makes it a superb coding platform.

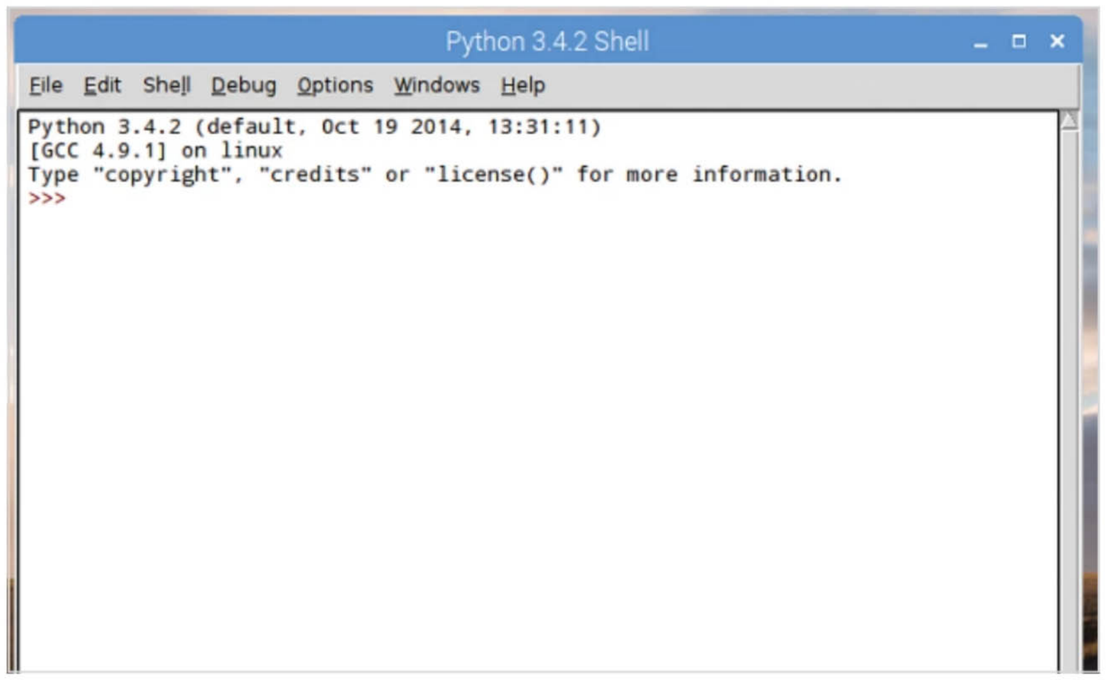
## STARTING PYTHON

**We won't go into the details of getting the Raspberry Pi up and running, there's plenty of material already available on that subject. However once you're ready, fire up your Pi and get ready for coding.**

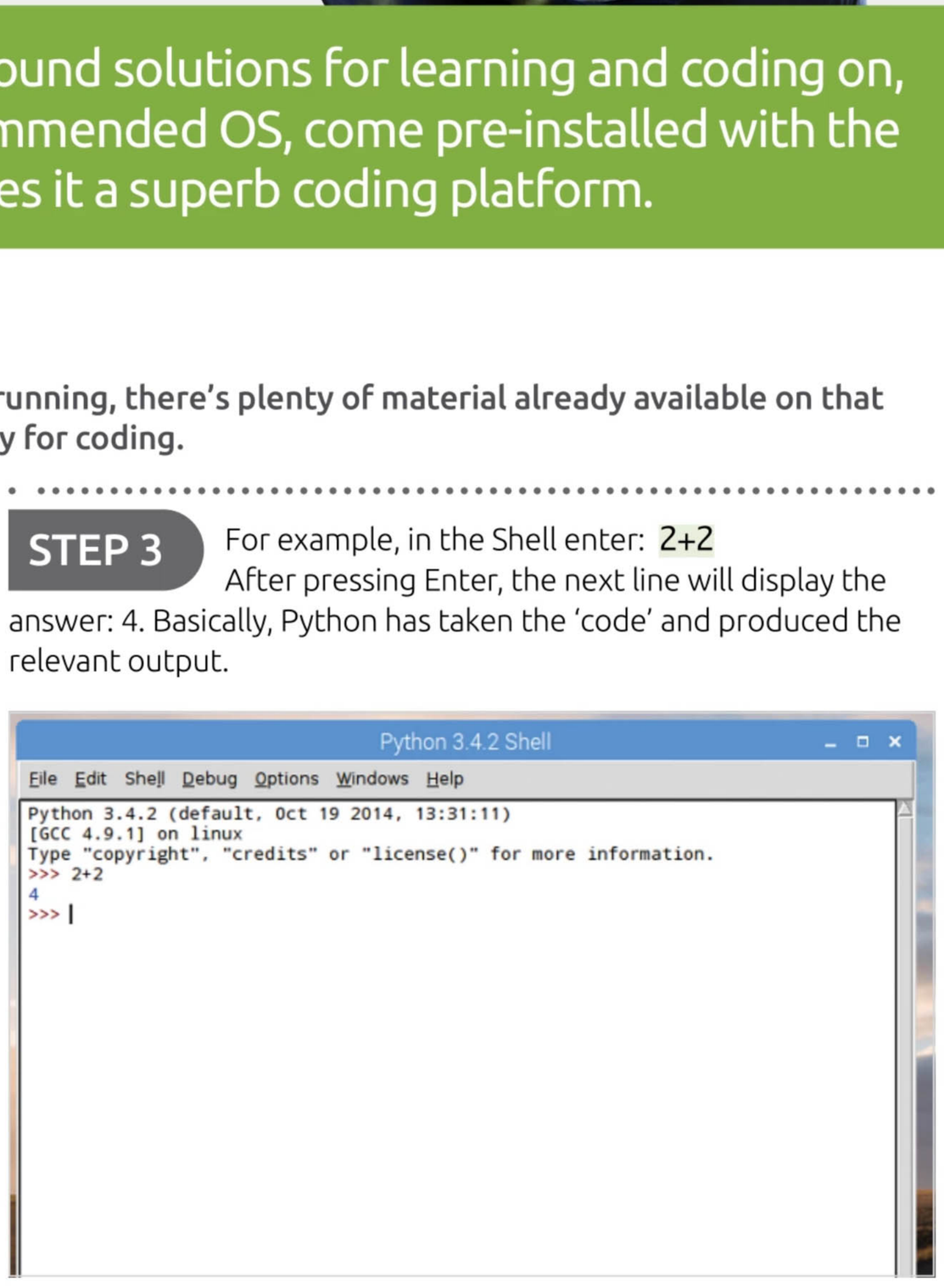
**STEP 1** With the Raspbian desktop loaded, click on the Menu button followed by Programming > Python 3 (IDLE). This will open the Python 3 Shell. Windows and Mac users can find the Python 3 IDLE Shell from within the Windows Start button menu and via Finder.



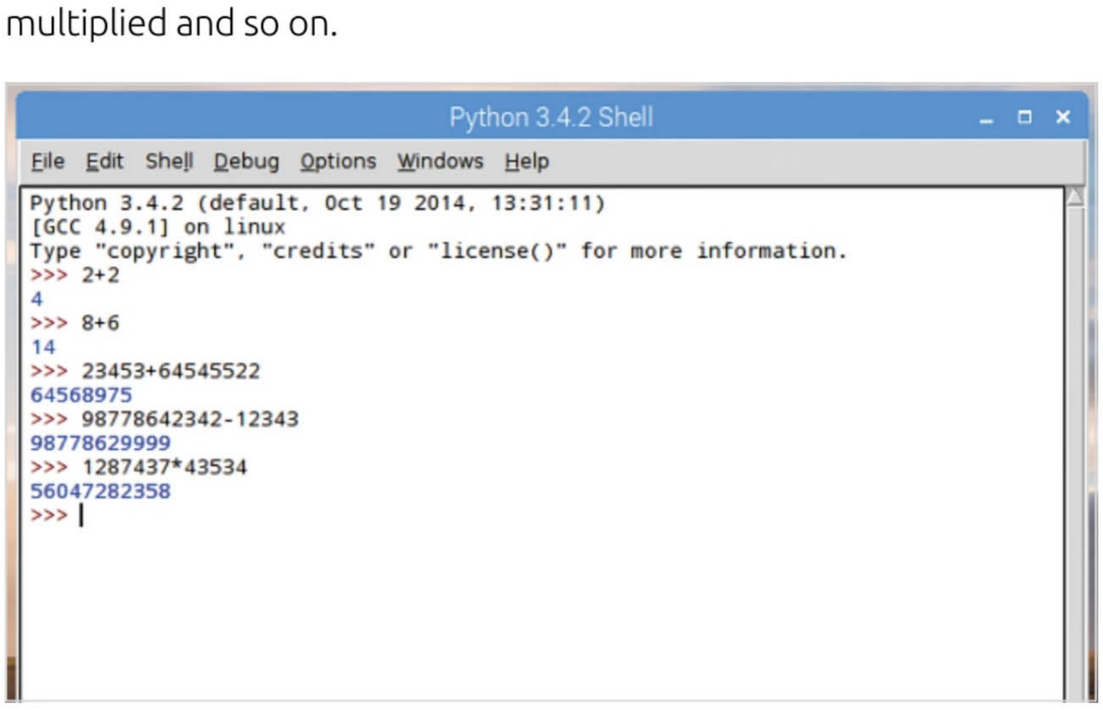

**STEP 2** The Shell is where you can enter code and see the responses and output of code you've programmed into Python. This is a kind of sandbox, where you're able to try out some simple code and processes.




**STEP 3** For example, in the Shell enter: `2+2` After pressing Enter, the next line will display the answer: 4. Basically, Python has taken the 'code' and produced the relevant output.




**STEP 4** The Python Shell acts very much like a calculator, since code is basically a series of mathematical interactions with the system. Integers, which are the infinite sequence of whole numbers can easily be added, subtracted, multiplied and so on.

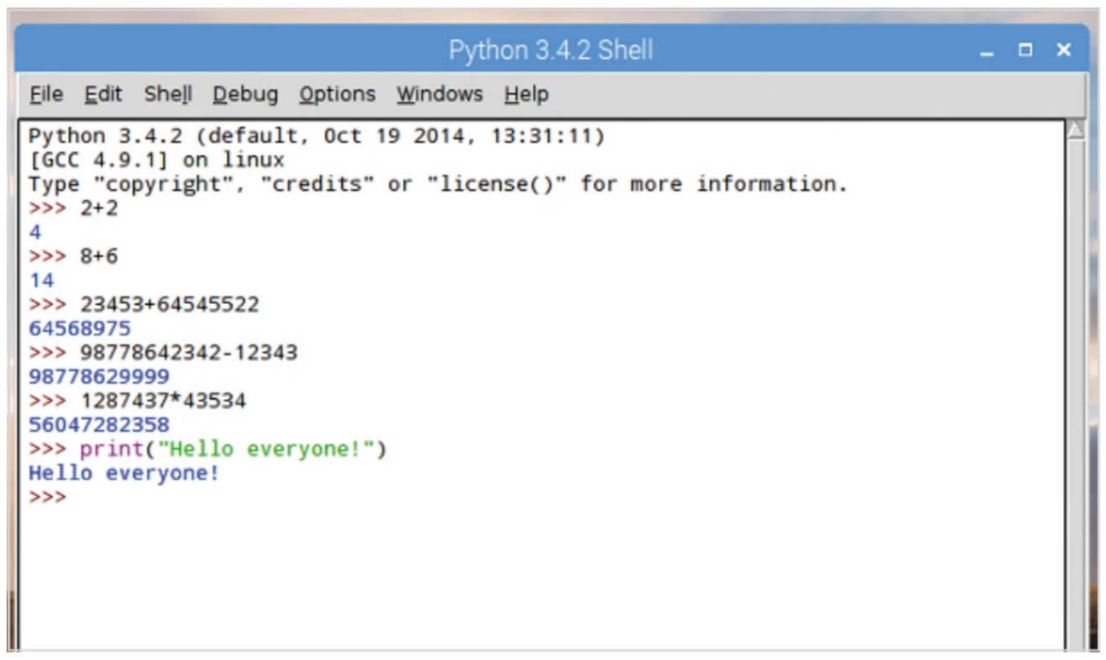


## STEP 5

While that's very interesting, it's not particularly exciting. Instead, try this:

```
print("Hello everyone!")
```

Just enter it into the IDLE as you did in the previous steps.




## STEP 6

This is a little more like it, since you've just produced your first bit of code. The Print command is fairly self-explanatory, it prints things. Python 3 requires the brackets as well as quote marks in order to output content to the screen, in this case the Hello everyone! bit.

## STEP 7

You may have noticed the colour coding within the Python IDLE. The colours represent different elements of Python code. They are:

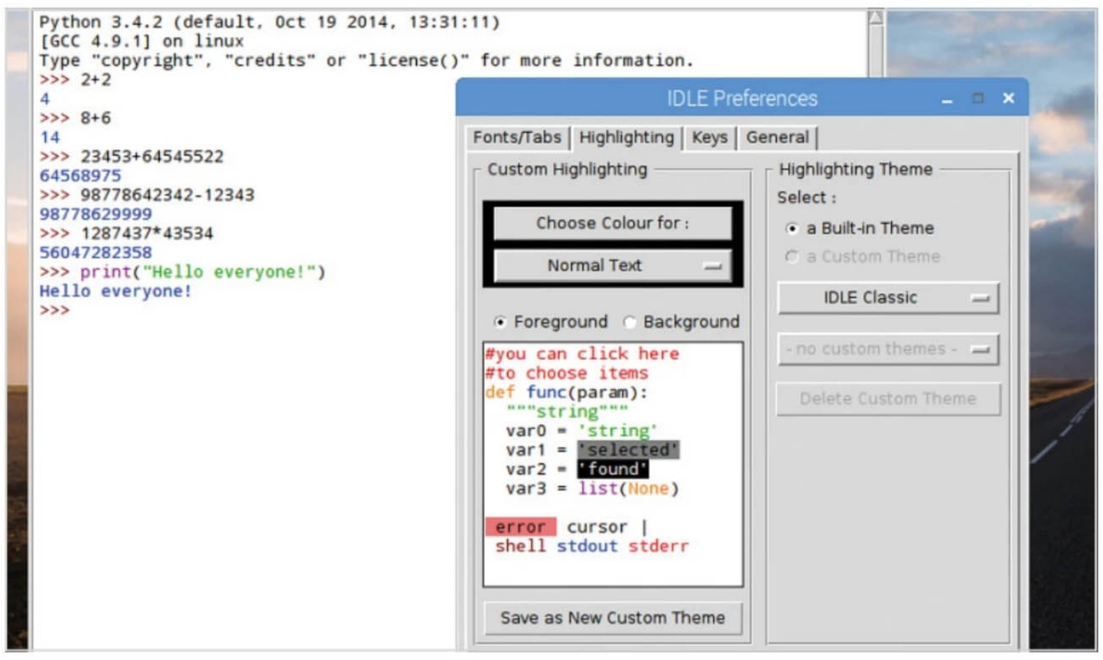
Black – Data and Variables
Green – Strings
Purple – Functions
Orange – Commands
Blue – User Functions
Dark Red – Comments
Light Red – Error Messages

**IDLE Colour Coding**

| Colour | Use for | Examples |
|---|---|---|
| Black | Data & variables | `23.6 area` |
| Green | Strings | `"Hello World"` |
| Purple | Functions | `len() print()` |
| Orange | Commands | `if for else` |
| Blue | User functions | `get_area()` |
| Dark red | Comments | `#Remember VAT` |
| Light red | Error messages | `SyntaxError:` |

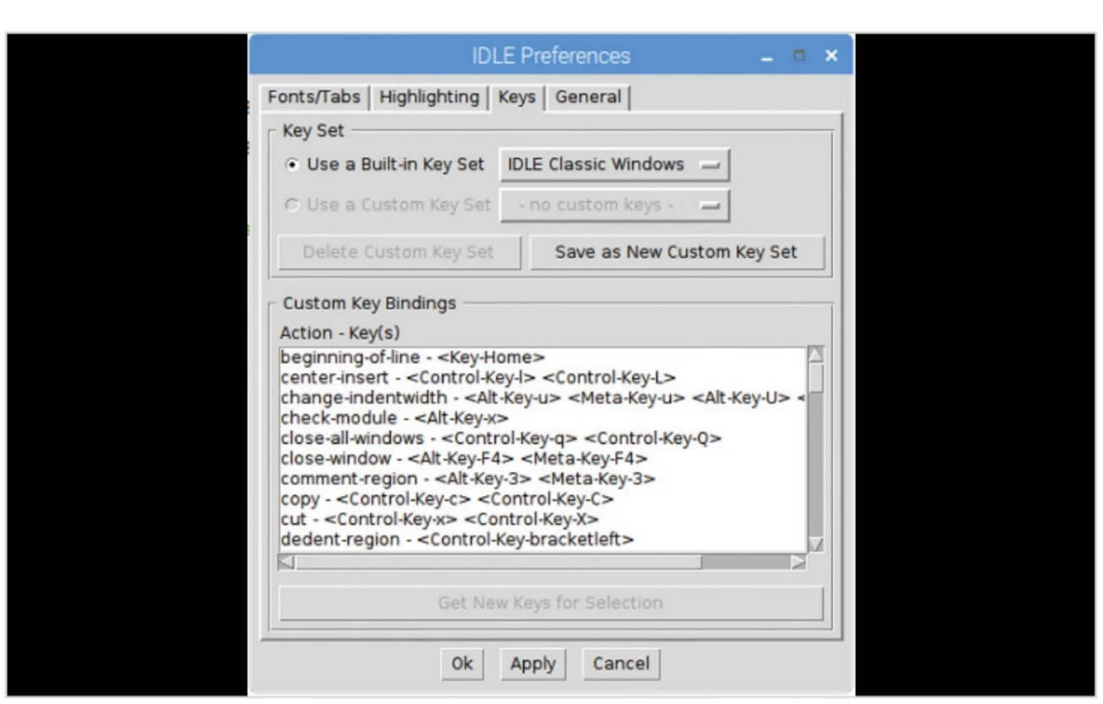
## STEP 8

The Python IDLE is a configurable environment. If you don't like the way the colours are represented, then you can always change them via Options > Configure IDLE and clicking on the Highlighting tab. However, we don't recommend that as you won't be seeing the same as our screenshots.



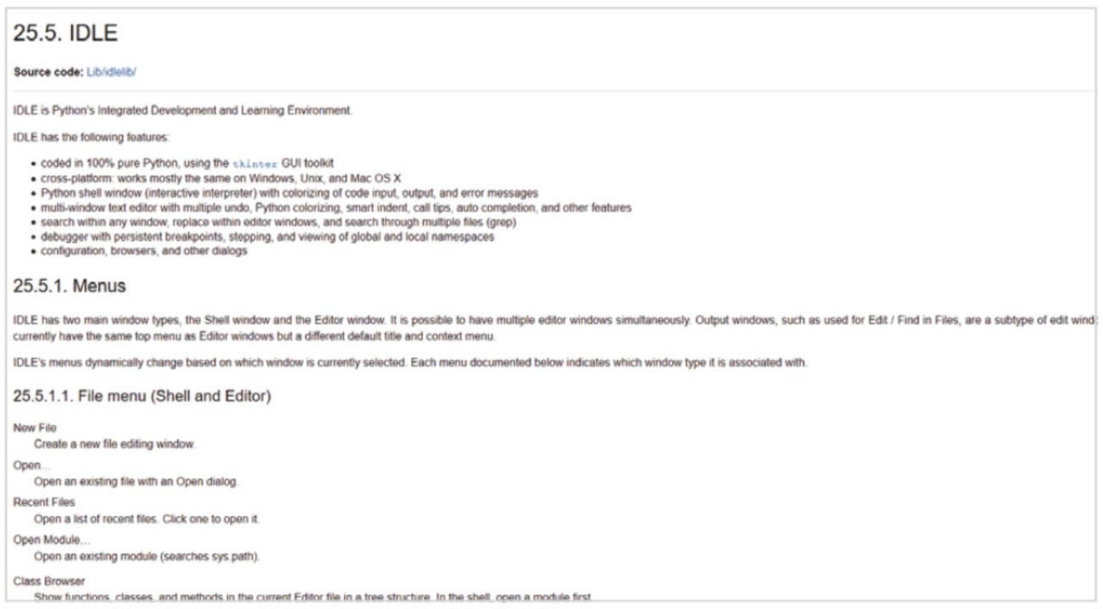

## STEP 9

Just like most programs available, regardless of the operating system, there are numerous shortcut keys available. We don't have room for them all here but within the Options > Configure IDLE and under the Keys tab, you can see a list of the current bindings.




## STEP 10

The Python IDLE is a power interface, and one that's actually been written in Python using one of the available GUI toolkits. If you want to know the many ins and outs for the Shell, we recommend you take a few moments to view **www.docs.python.org/3/library/idle.html**, which details many of the IDLE's features.



25.5. IDLE

**Source code:** Lib/idlelib/

IDLE is Python's Integrated Development and Learning Environment.

IDLE has the following features:

- coded in 100% pure Python, using the `tkinter` GUI toolkit
- cross-platform: works mostly the same on Windows, Unix, and Mac OS X
- Python shell window (interactive interpreter) with colorizing of code input, output, and error messages
- multi-window text editor with multiple undo, Python colorizing, smart indent, call tips, auto completion, and other features
- search within any window, replace within editor windows, and search through multiple files (grep)
- debugger with persistent breakpoints, stepping, and viewing of global and local namespaces
- configuration, browsers, and other dialogs

25.5.1. Menus

IDLE has two main window types, the Shell window and the Editor window. It is possible to have multiple editor windows simultaneously. Output windows, such as used for Edit / Find in Files, are a subtype of edit wind currently have the same top menu as Editor windows but a different default title and context menu.

IDLE's menus dynamically change based on which window is currently selected. Each menu documented below indicates which window type it is associated with.

25.5.1.1. File menu (Shell and Editor)

New File
Create a new file editing window.

Open...
Open an existing file with an Open dialog.

Recent Files
Open a list of recent files. Click one to open it.

Open Module...
Open an existing module (searches sys.path).

Class Browser
Show functions, classes, and methods in the current Editor file in a tree structure. In the shell, open a module first.

# Your First Code

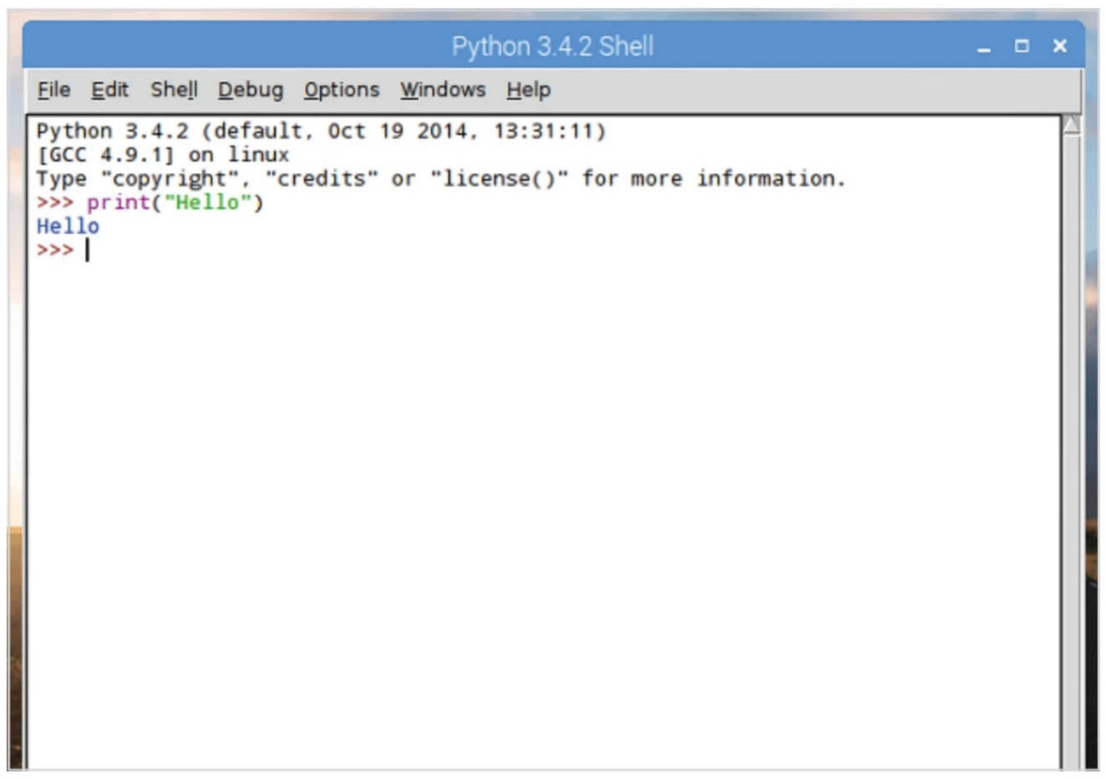
Essentially, you've already written your first piece of code with the 'print("Hello everyone!")' function from the previous tutorial. However, let's expand that and look at entering your code and playing around with some other Python examples.

## PLAYING WITH PYTHON

**With most languages, computer or human, it's all about remembering and applying the right words to the right situation. You're not born knowing these words, so you need to learn them.**

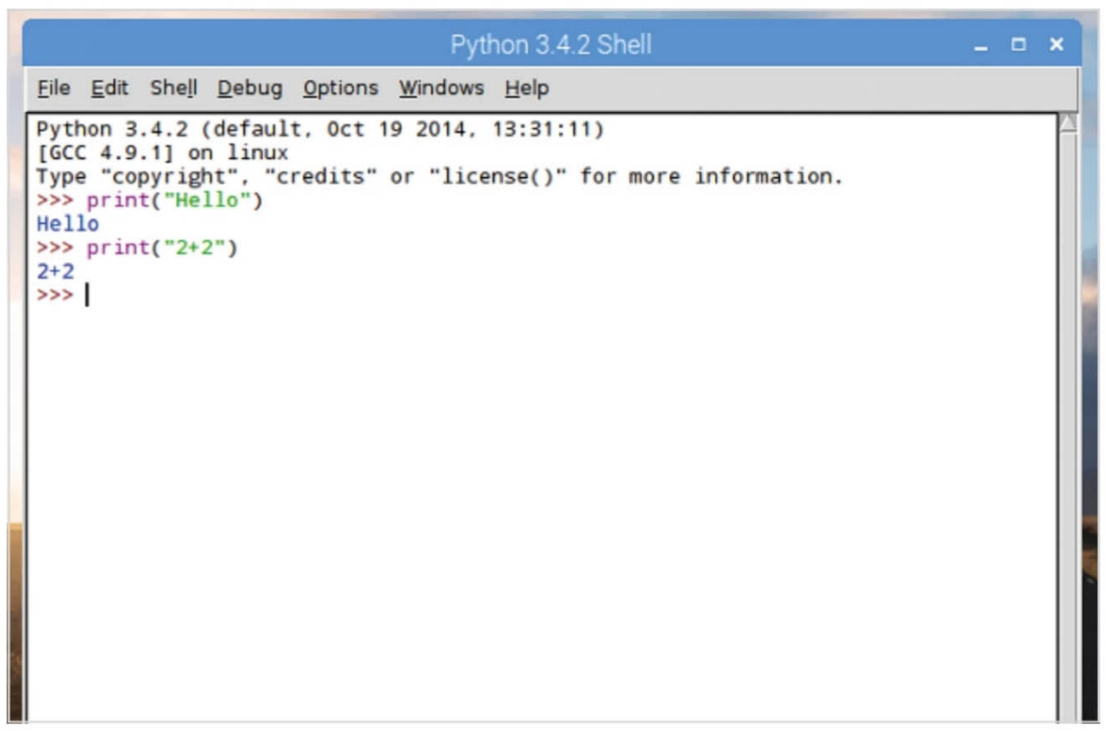
**STEP 1** If you've closed Python 3 IDLE, reopen it in whichever operating system version you prefer. In the Shell, enter the familiar following:

```
print("Hello")
```



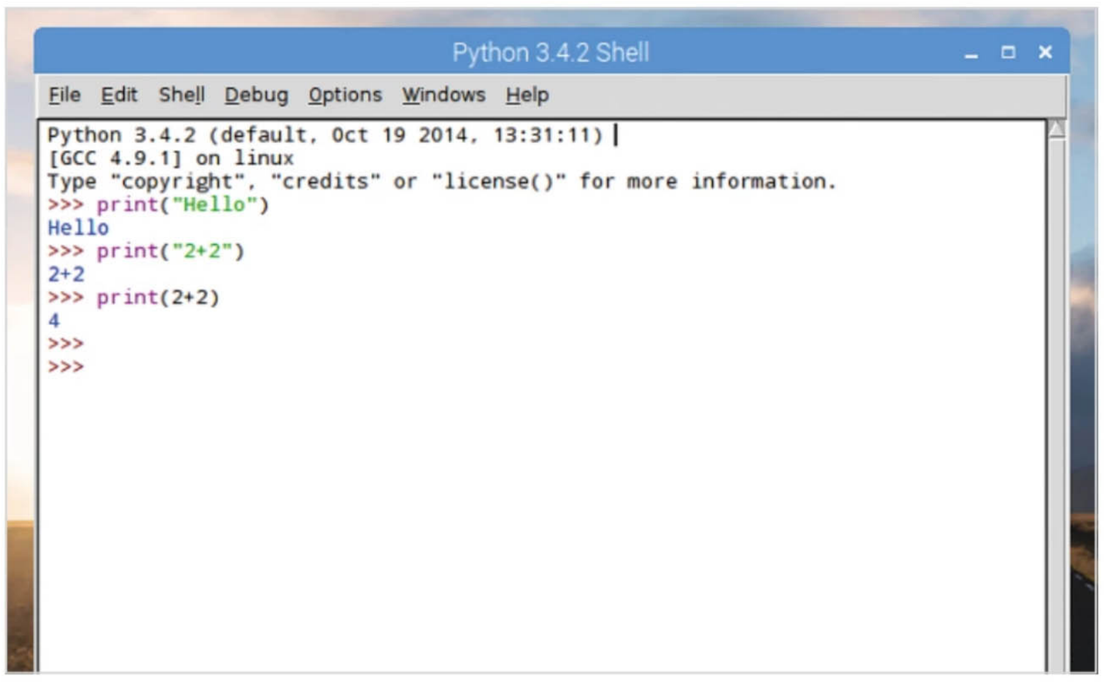

**STEP 2** Just as predicted, the word Hello appears in the Shell as blue text, indicating output from a string. It's fairly straightforward and doesn't require too much explanation. Now try:

```
print("2+2")
```



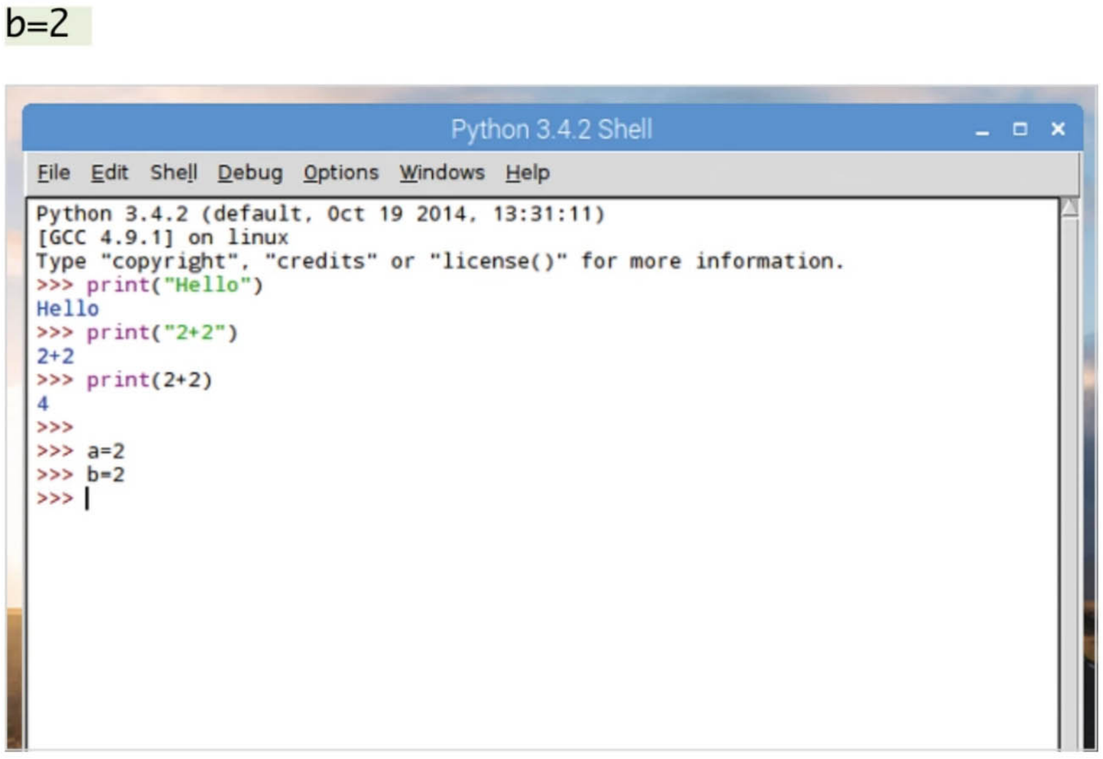

**STEP 3** You can see that instead of the number 4, the output is the 2+2 you asked to be printed to the screen. The quotation marks are defining what's being outputted to the IDLE Shell; to print the total of 2+2 you need to remove the quotes:

```
print(2+2)
```



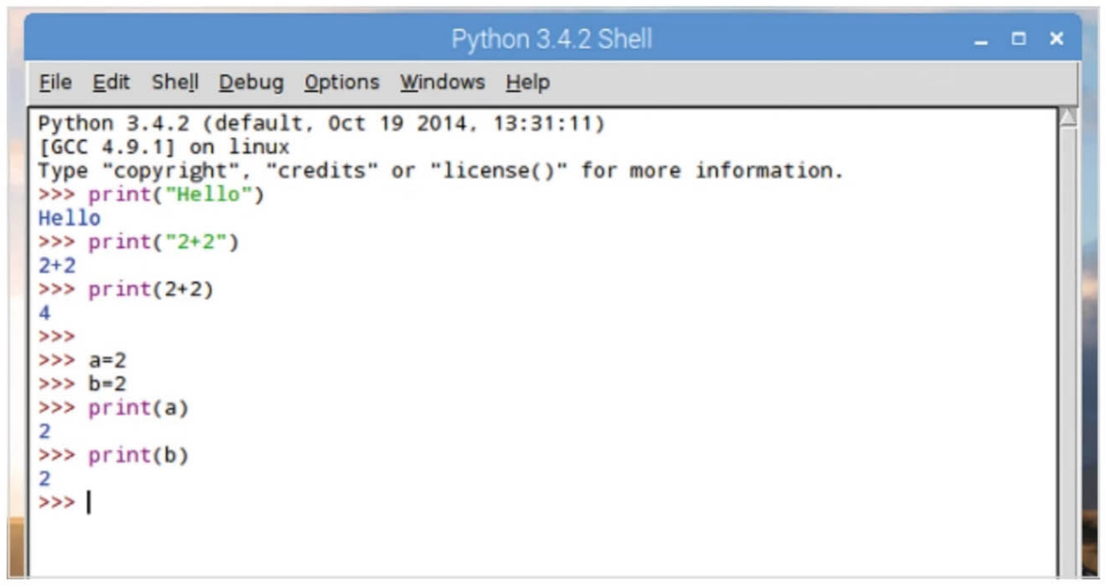

**STEP 4** You can continue as such, printing 2+2, 464+2343 and so on to the Shell. An easier way is to use a variable, which is something we will cover in more depth later. For now, enter:

```
a=2
b=2
```



**STEP 5** What you have done here is assign the letters a and b two values: 2 and 2. These are now variables, which can be called upon by Python to output, add, subtract, divide and so on for as long as their numbers stay the same. Try this:

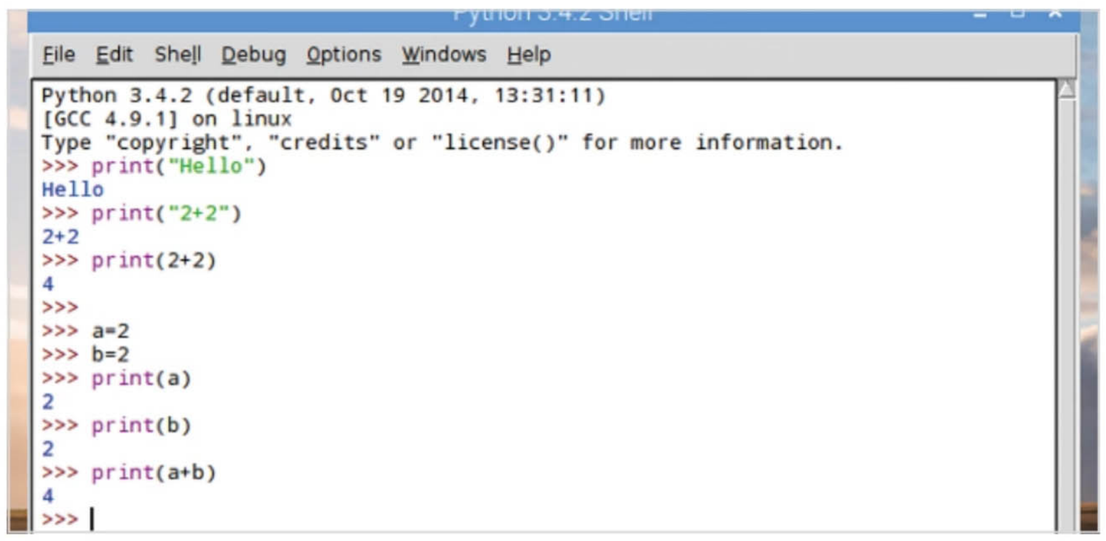
```
print(a)
print(b)
```




**STEP 6** The output of the last step displays the current values of both a and b individually, as you've asked them to be printed separately. If you want to add them up, you can use the following:

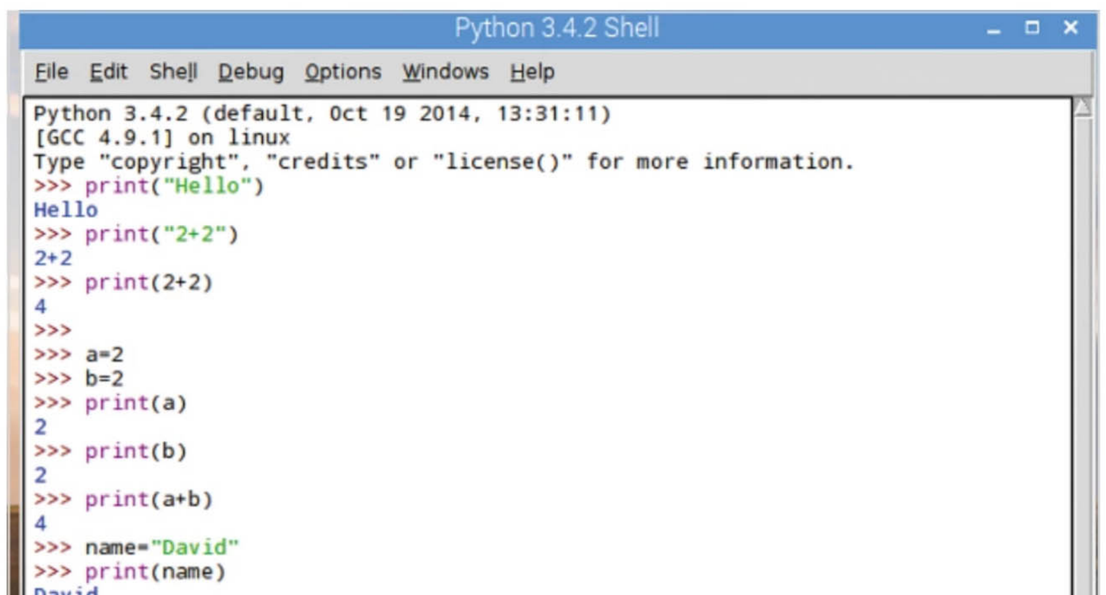
```
print(a+b)
```

This code simply takes the values of a and b, adds them together and outputs the result.




**STEP 7** You can play around with different kinds of variables and the Print function. For example, you could assign variables for someone's name:

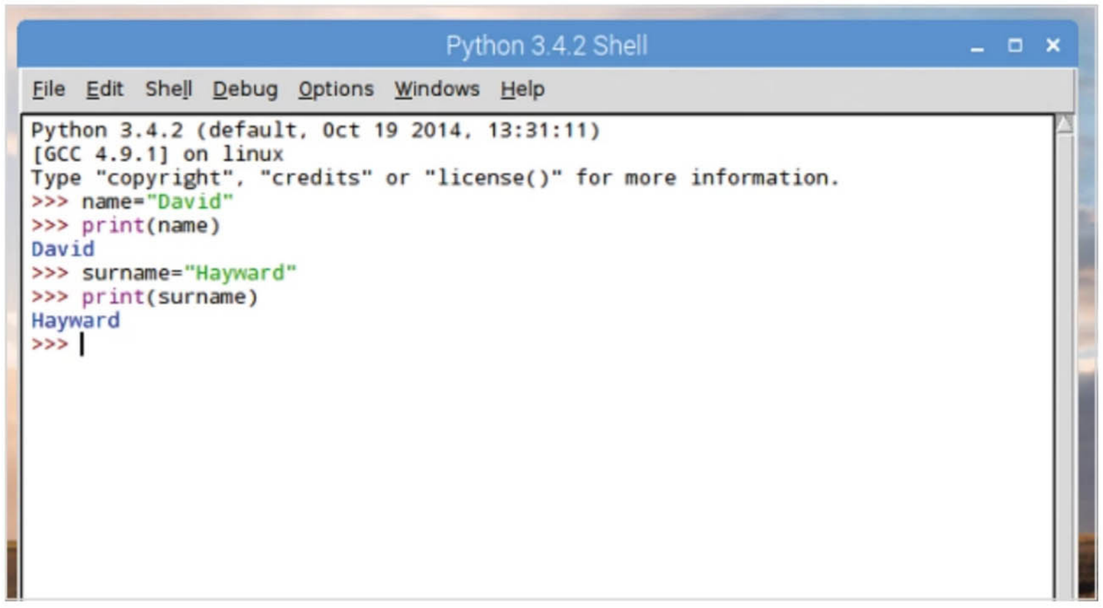
```
name=”David”
print(name)
```




**STEP 8** Now let's add a surname:

```
surname=”Hayward”
print(surname)
```

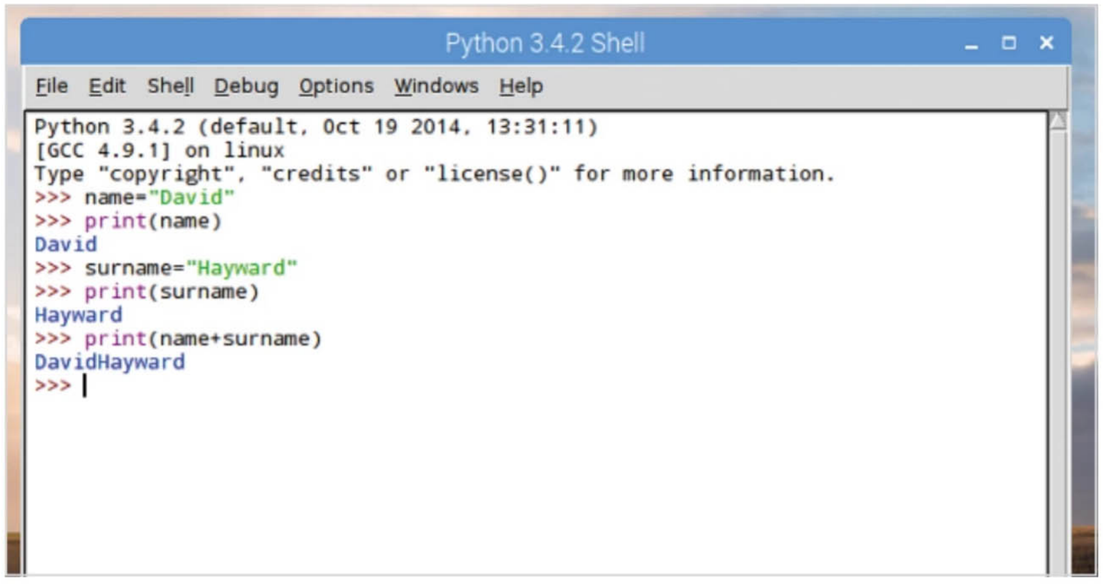
You now have two variables containing a first name and a surname and you can print them independently.




**STEP 9** If we were to apply the same routine as before, using the + symbol, the name wouldn't appear correctly in the output in the Shell. Try it:

```
print(name+surname)
```

You need a space between the two, defining them as two separate values and not something you mathematically play around with.



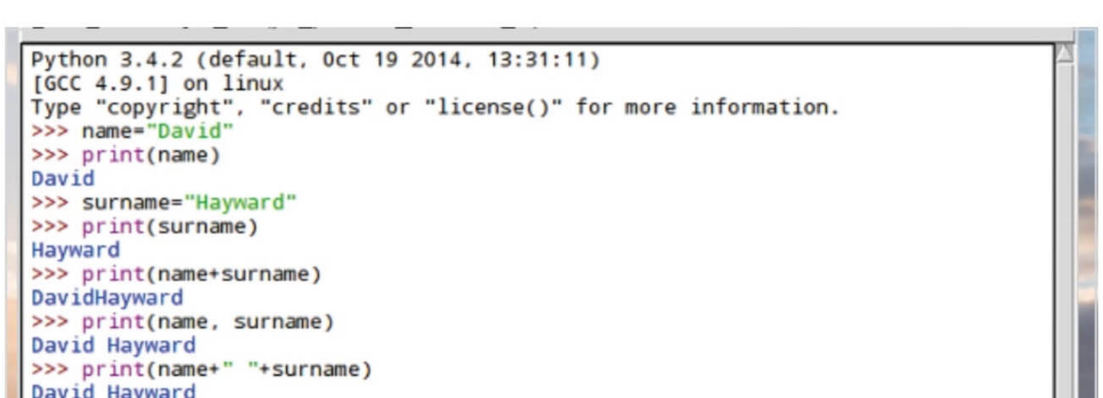

**STEP 10** In Python 3 you can separate the two variables with a space using a comma:

```
print(name, surname)
```

Alternatively, you can add the space ourselves:

```
print(name+” “+surname)
```

The use of the comma is much neater, as you can see. Congratulations, you've just taken your first steps into the wide world of Python.



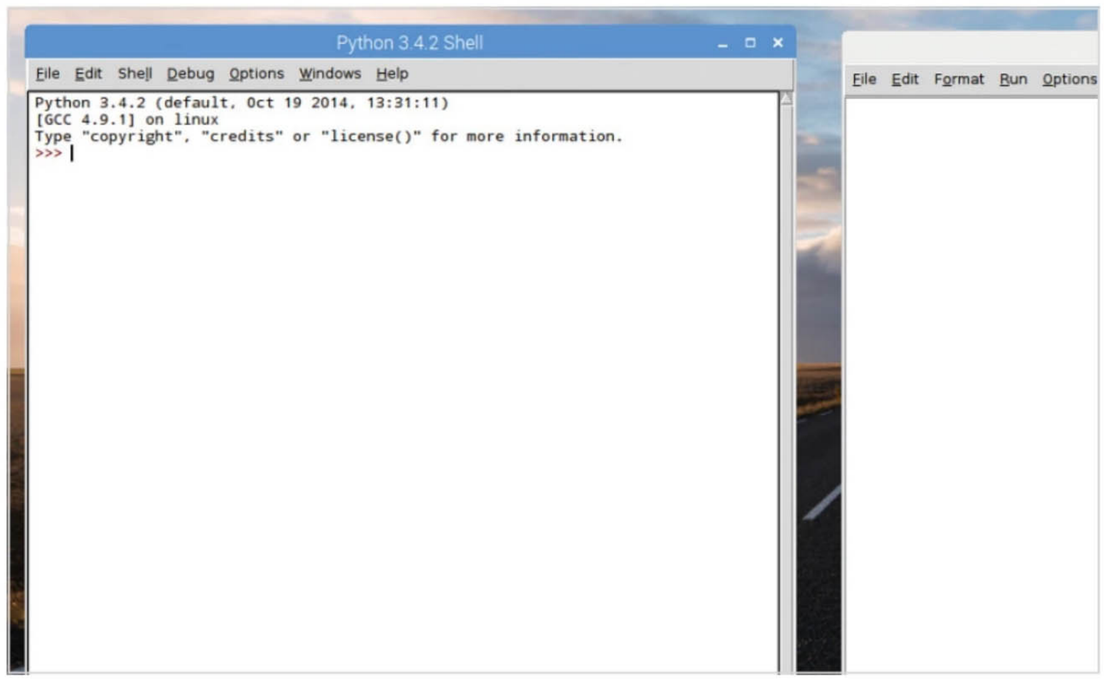
# Saving and Executing Your Code

While working in the IDLE Shell is perfectly fine for small code snippets, it's not designed for entering longer program listings. In this section you're going to be introduced to the IDLE Editor, where you will be working from now on.

## EDITING CODE

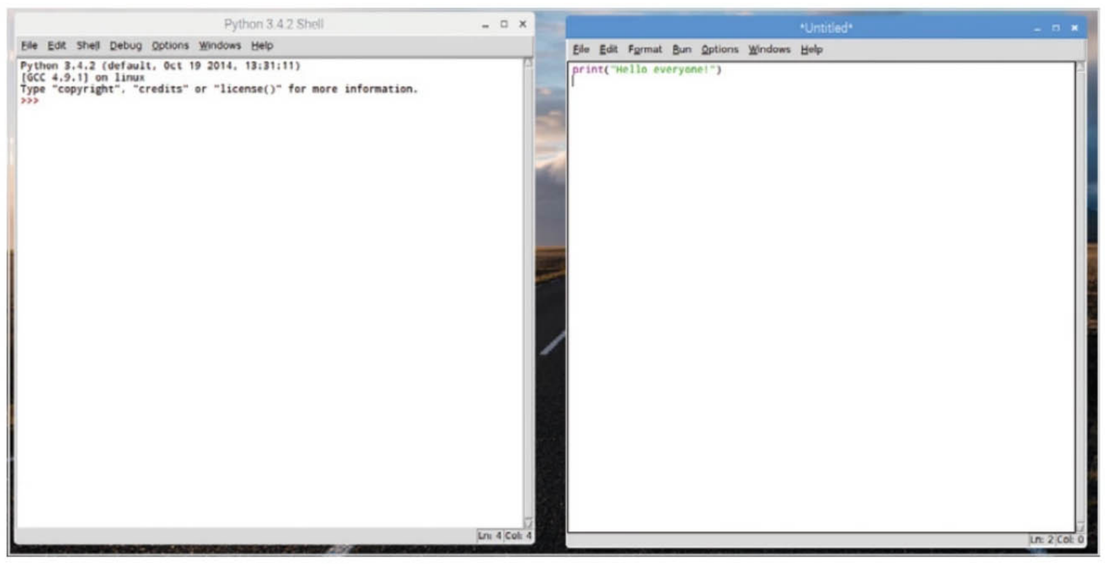
**You will eventually reach a point where you have to move on from inputting single lines of code into the Shell. Instead, the IDLE Editor will allow you to save and execute your Python code.**

**STEP 1** First, open the Python IDLE Shell and when it's up, click on File > New File. This will open a new window with Untitled as its name. This is the Python IDLE Editor and within it you can enter the code needed to create your future programs.

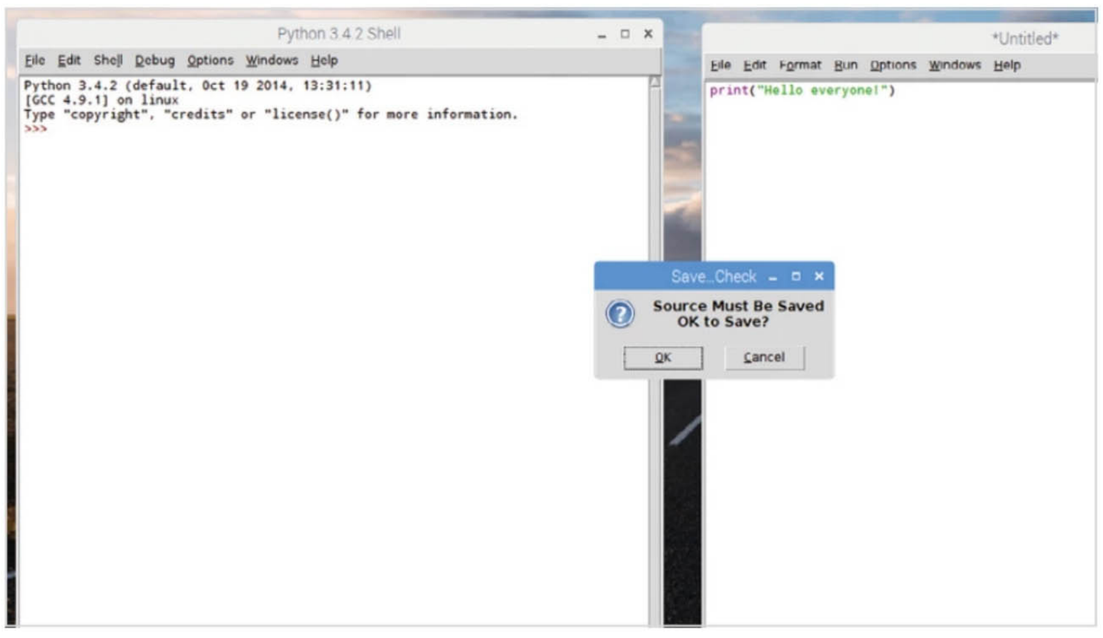



**STEP 2** The IDLE Editor is, for all intents and purposes, a simple text editor with Python features, colour coding and so on; much in the same vein as Sublime. You enter code as you would within the Shell, so taking an example from the previous tutorial, enter:

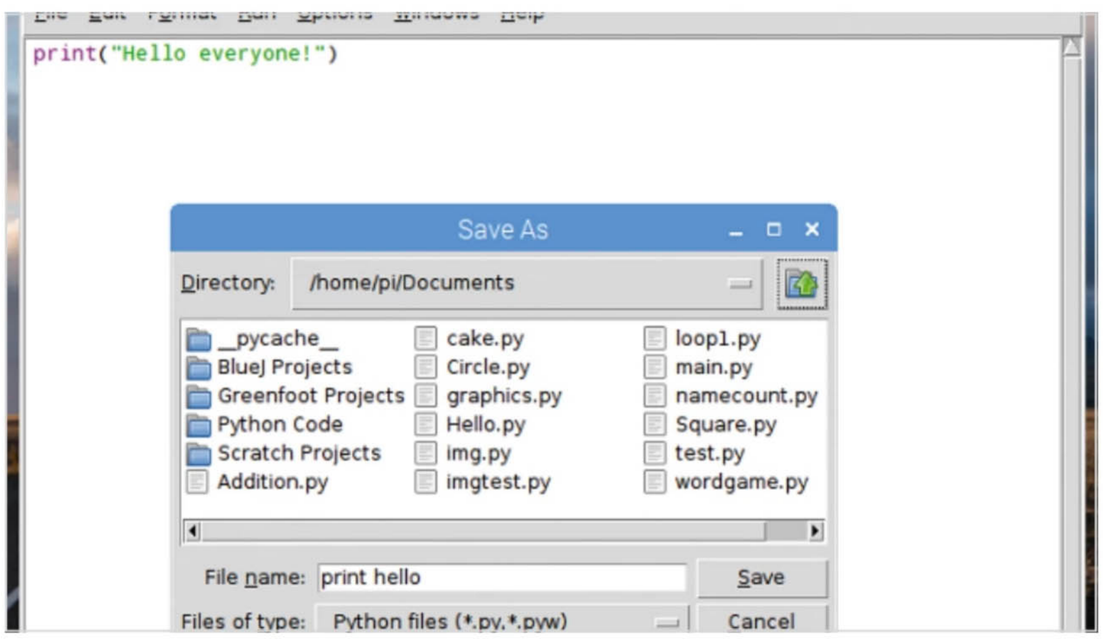
```
print("Hello everyone!")
```



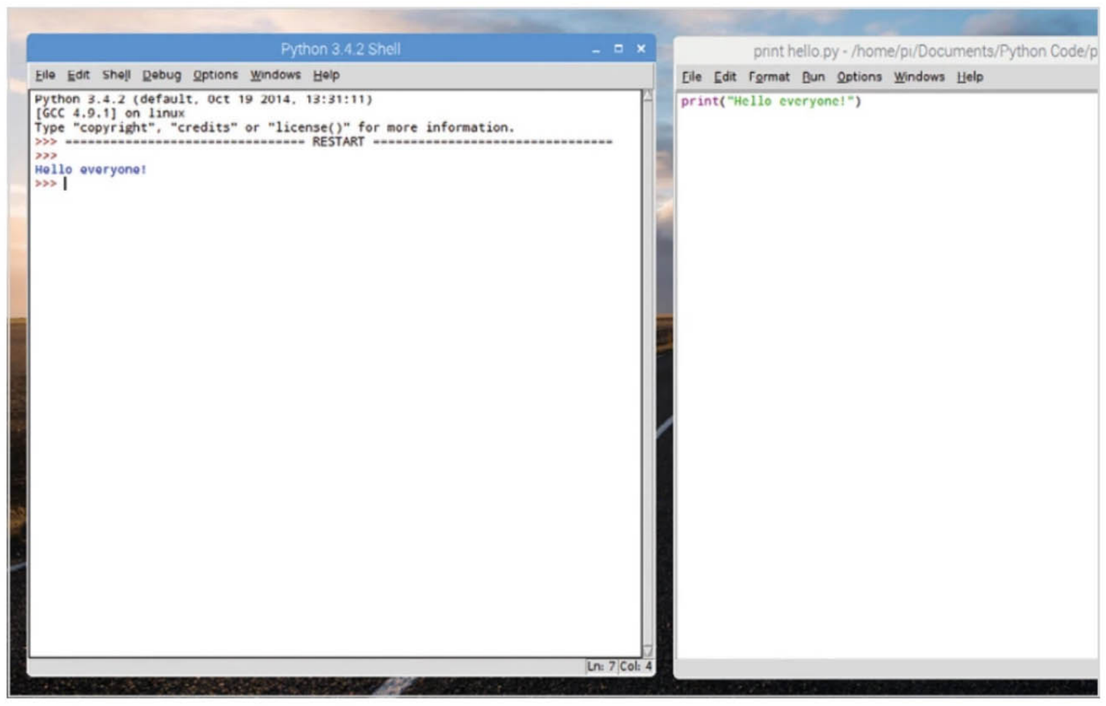

**STEP 3** You can see that the same colour coding is in place in the IDLE Editor as it is in the Shell, enabling you to better understand what's going on with your code. However, to execute the code you need to first save it. Press F5 and you get a Save…Check box open.



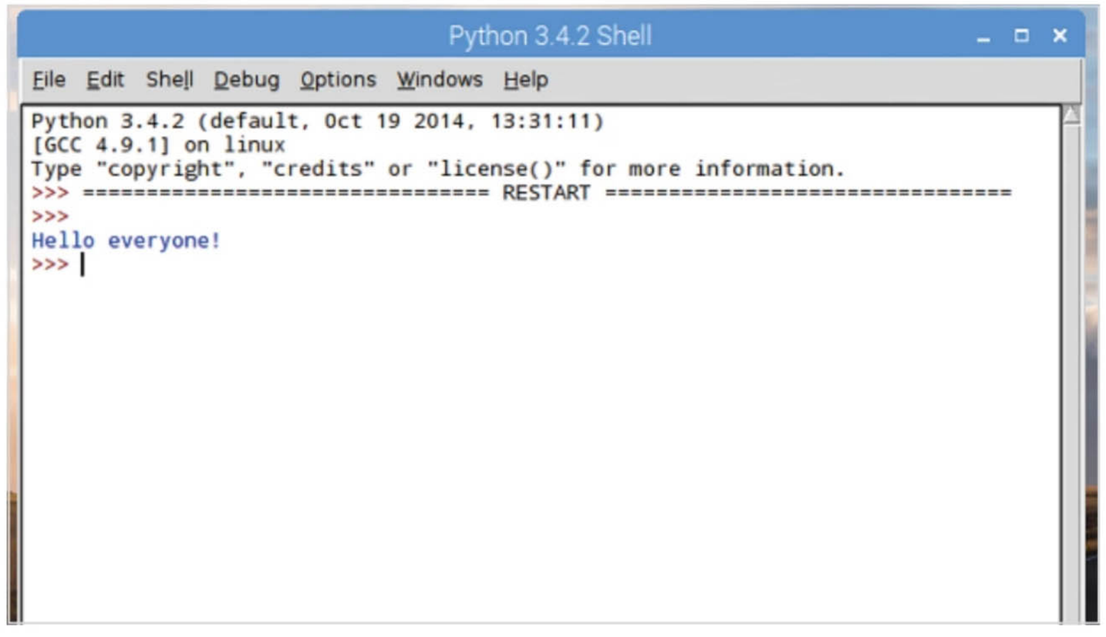

**STEP 4** Click on the OK button in the Save box and select a destination where you'll save all your Python code. The destination can be a dedicated folder called Python or you can just dump it wherever you like. Remember to keep a tidy drive though, to help you out in the future.



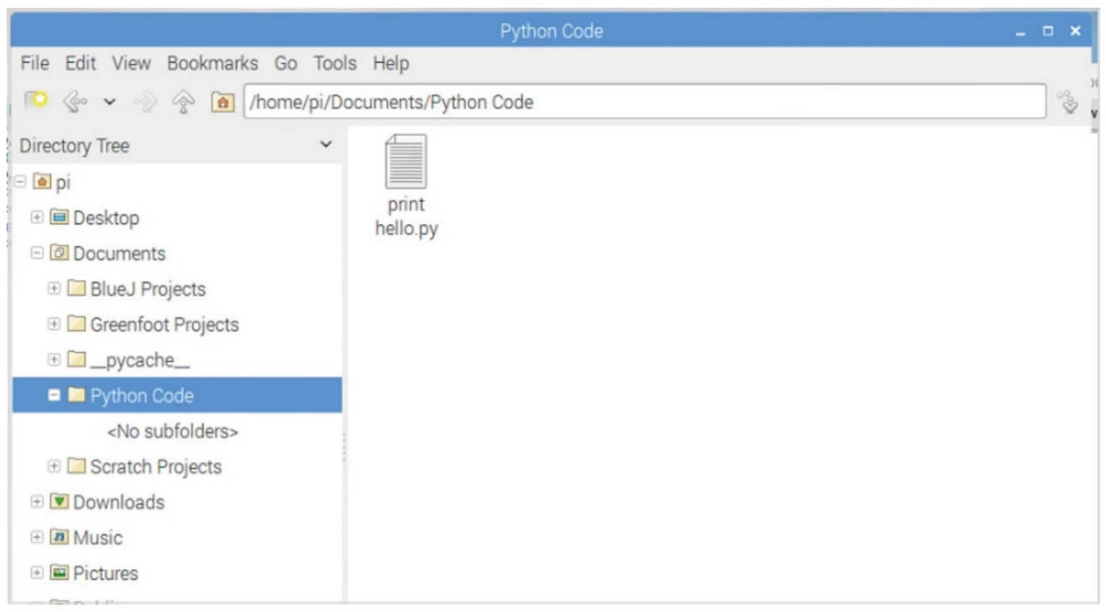
**STEP 5** Enter a name for your code, 'print hello' for example, and click on the Save button. Once the Python code is saved it's executed and the output will be detailed in the IDLE Shell. In this case, the words 'Hello everyone!'.




**STEP 6** This is how the vast majority of your Python code will be conducted. Enter it into the Editor, hit F5, save the code and look at the output in the Shell. Sometimes things will differ, depending on whether you've requested a separate window, but essentially that's the process. It's the process we will use throughout this book, unless otherwise stated.



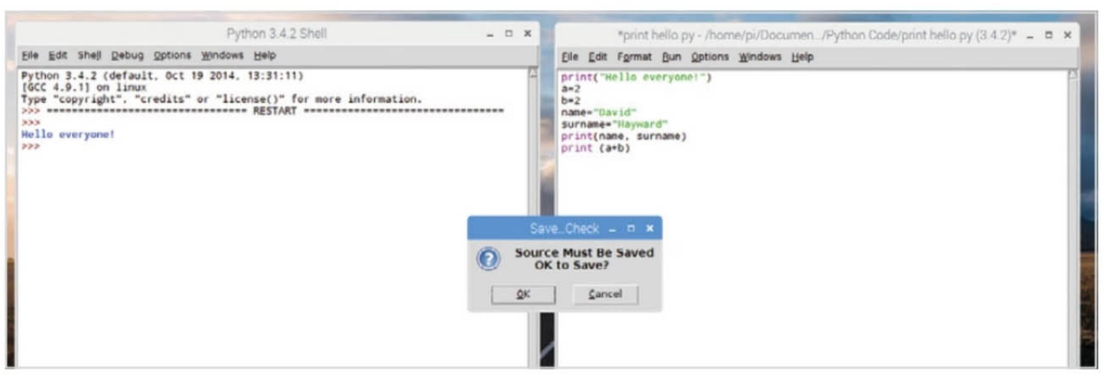

**STEP 7** If you open the file location of the saved Python code, you can see that it ends in a .py extension. This is the default Python file name. Any code you create will be whatever.py and any code downloaded from the many Internet Python resource sites will be .py. Just ensure that the code is written for Python 3.



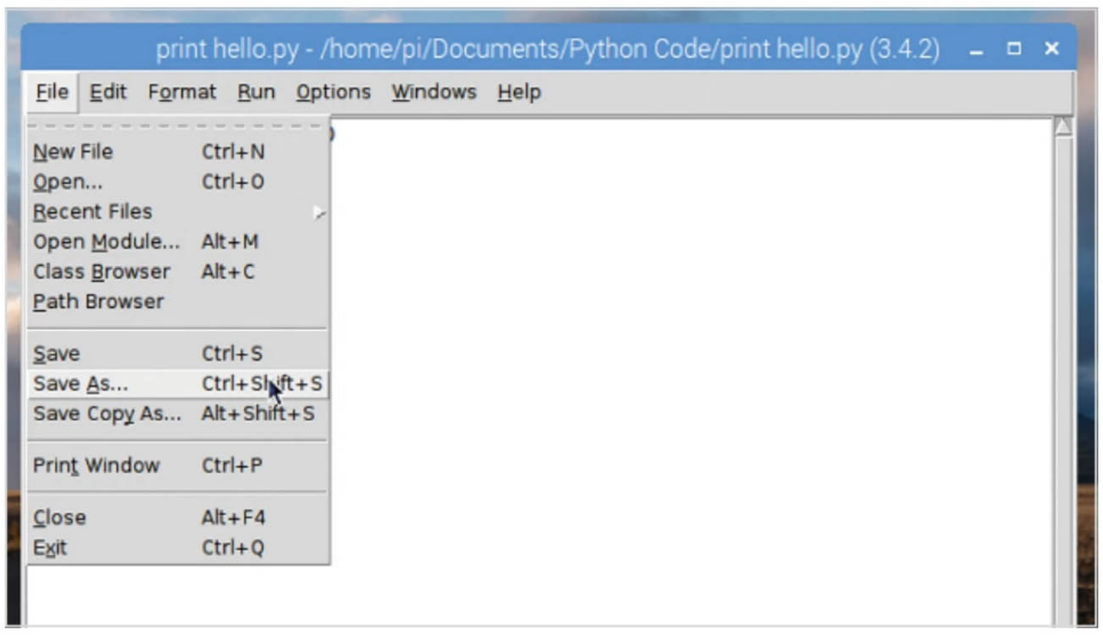

**STEP 8** Let's extend the code and enter a few examples from the previous tutorial:

```
a=2
b=2
name="David"
surname="Hayward"
print(name, surname)
print (a+b)
```

If you press F5 now you'll be asked to save the file, again, as it's been modified from before.



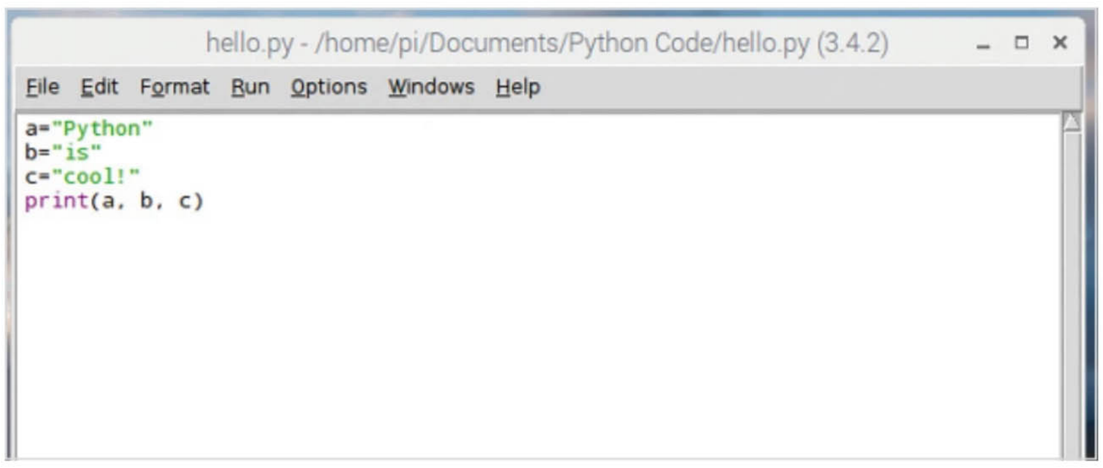

**STEP 9** If you click the OK button, the file will be overwritten with the new code entries, and executed, with the output in the Shell. It's not a problem with just these few lines but if you were to edit a larger file, overwriting can become an issue. Instead, use File > Save As from within the Editor to create a backup.




**STEP 10** Now create a new file. Close the Editor, and open a new instance (File > New File from the Shell). Enter the following and save it as hello.py:

```
a="Python"
b="is"
c="cool!"
print(a, b, c)
```

You will use this code in the next tutorial.



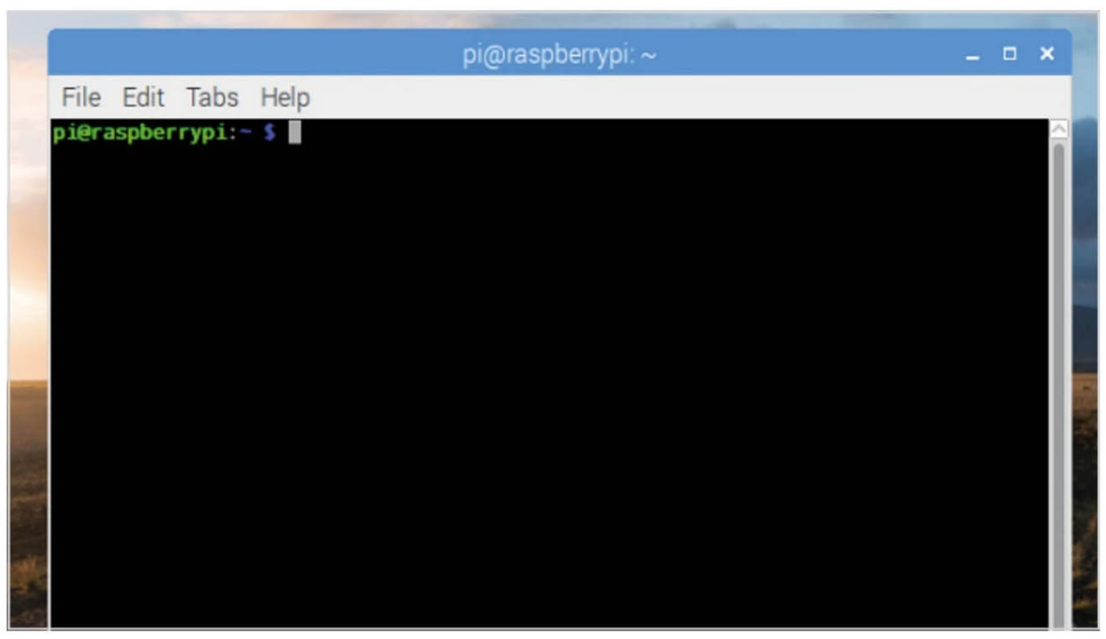
# Executing Code from the Command Line

Although we're working from the GUI IDLE throughout this book, it's worth taking a look at Python's command line handling. We already know there's a command line version of Python but it's also used to execute code.

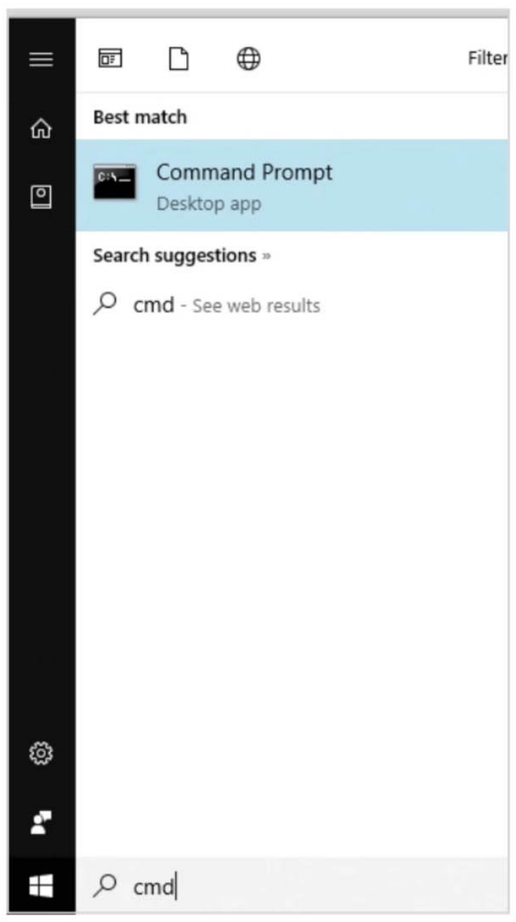
## COMMAND THE CODE

**Using the code we created in the previous tutorial, the one we named hello.py, let's see how you can run code that was made in the GUI at the command line level.**

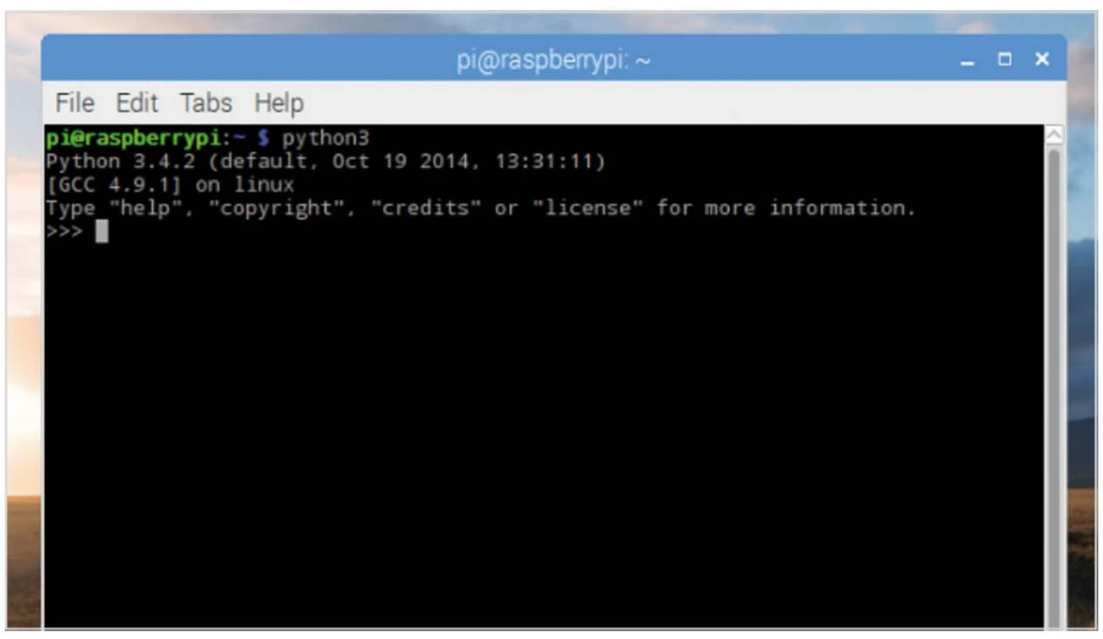
**STEP 1** Python, in Linux, comes with two possible ways of executing code via the command line. One of the ways is with Python 2, whilst the other uses the Python 3 libraries and so on. First though, drop into the command line or Terminal on your operating system.




**STEP 2** Just as before, we're using a Raspberry Pi: Windows users will need to click the Start button and search for CMD, then click the Command Line returned search; and macOS users can get access to their command line by clicking Go > Utilities > Terminal.



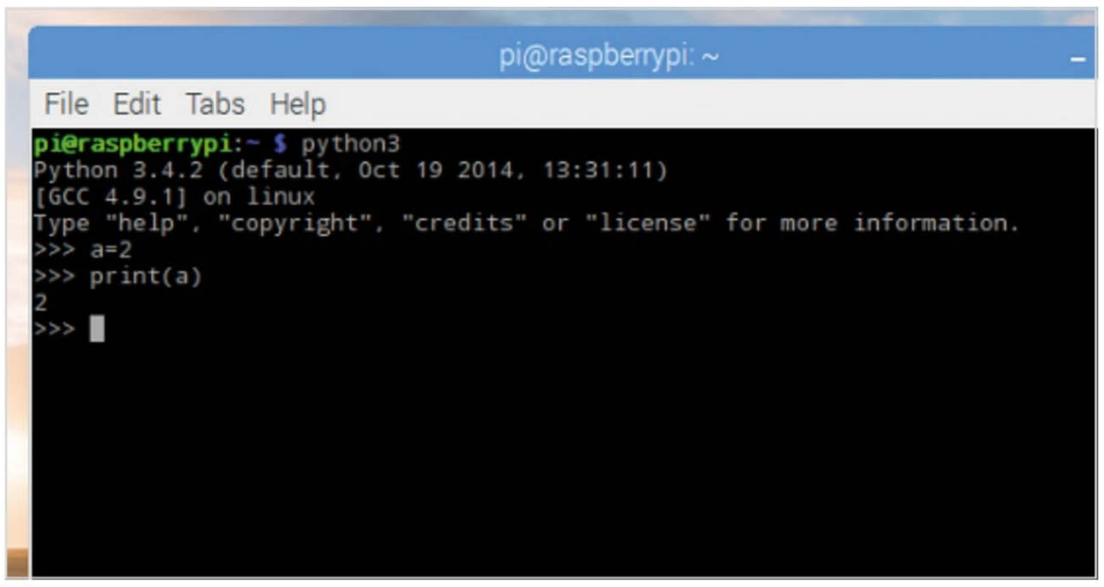

**STEP 3** Now you're at the command line we can start Python. For Python 3 you need to enter the command **python3** and press Enter. This will put you into the command line version of the Shell, with the familiar three right-facing arrows as the cursor (>>>).



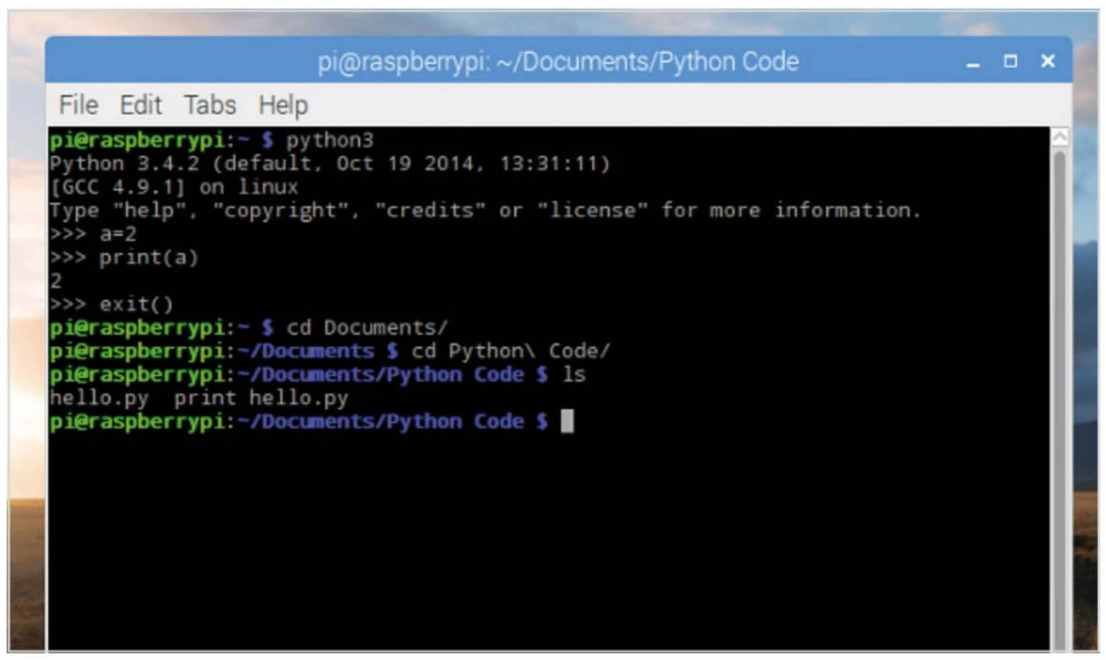

**STEP 4** From here you're able to enter the code you've looked at previously, such as:

```
a=2
print(a)
```

You can see that it works exactly the same.



**STEP 5** Now enter: **exit()** to leave the command line Python session and return you back to the command prompt. Enter the folder where you saved the code from the previous tutorial and list the available files within; hopefully you should see the hello.py file.

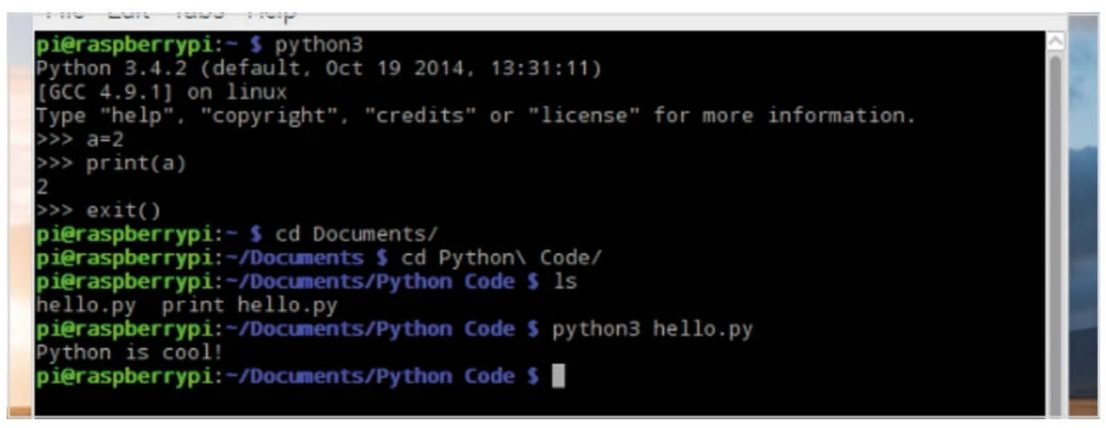



**STEP 6** From within the same folder as the code you're going to run, enter the following into the command line:

```
python3 hello.py
```

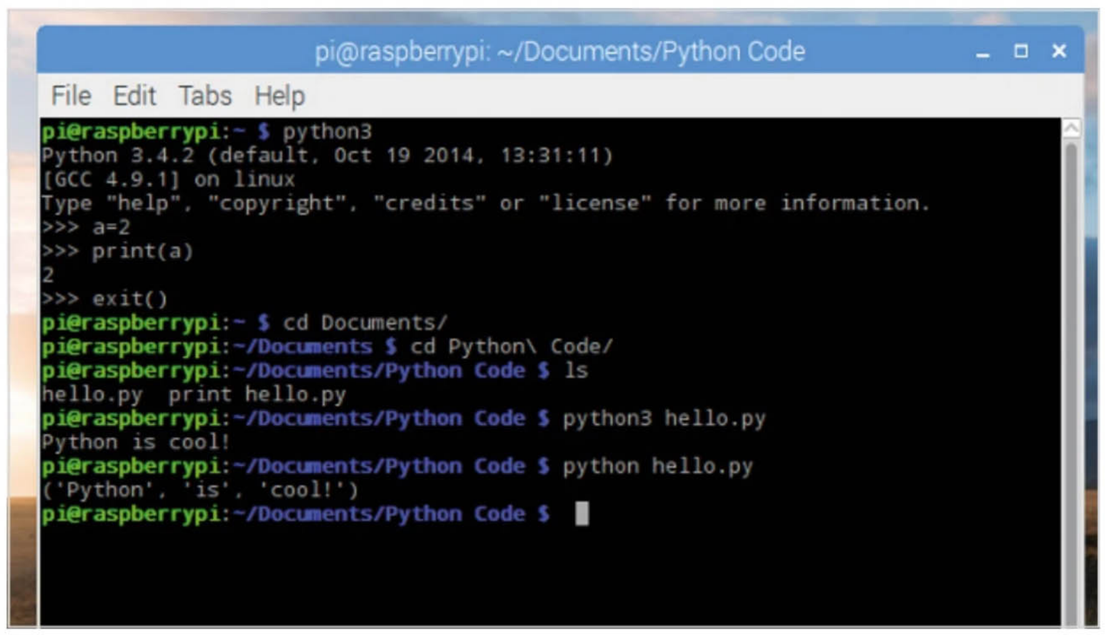
This will execute the code we created, which to remind you is:

```
a=”Python”
b=”is”
c=”cool!”
print(a, b, c)
```

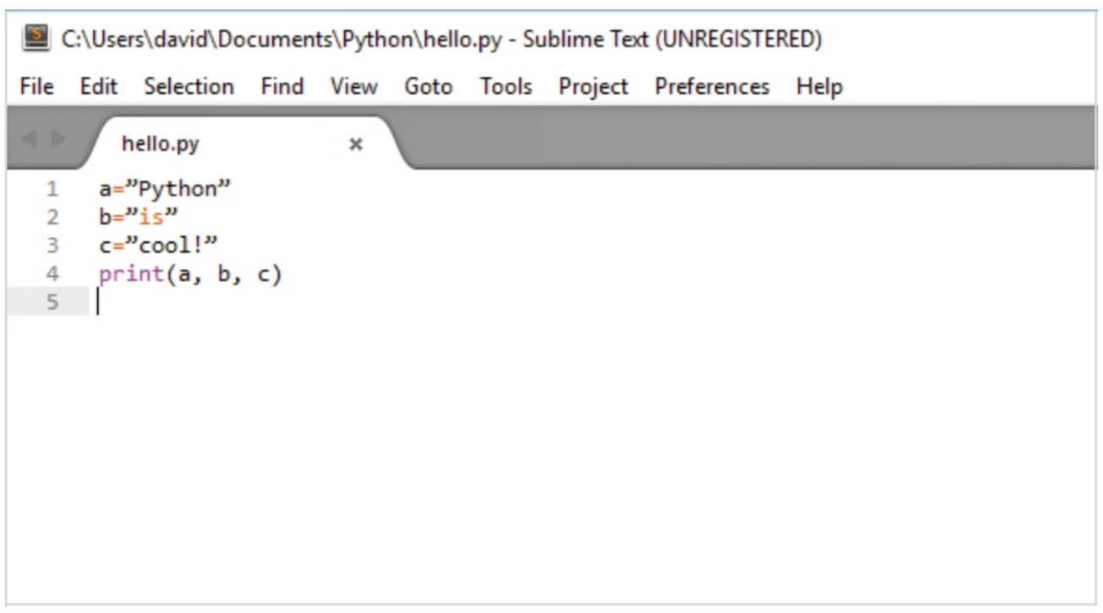



**STEP 7** Naturally, since this is Python 3 code, using the syntax and layout that's unique to Python 3, it only works when you use the python3 command. If you like, try the same with Python 2 by entering:

```
python hello.py
```

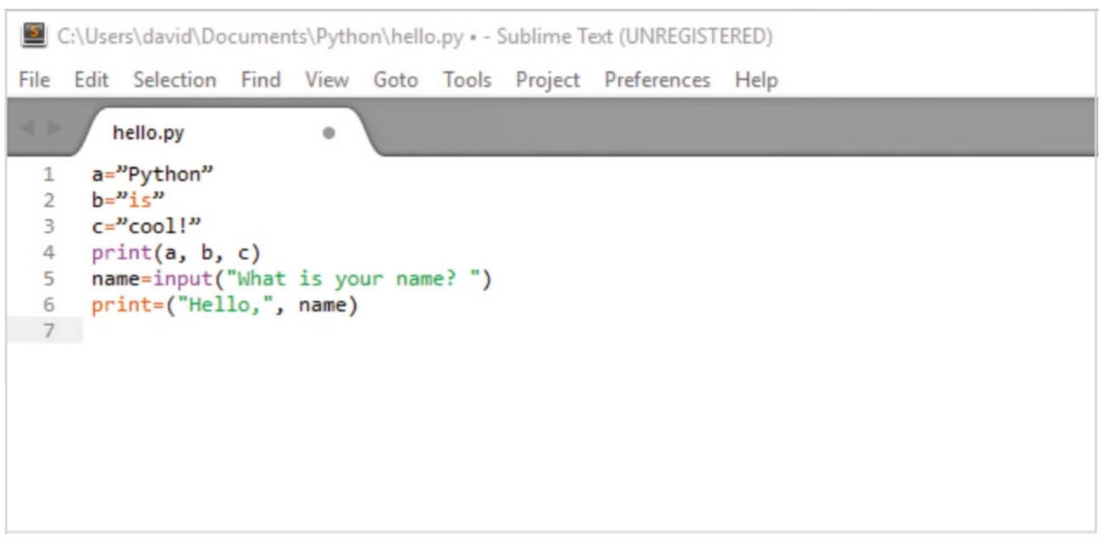



**STEP 8** The result of running Python 3 code from the Python 2 command line is quite obvious. Whilst it doesn't error out in any way, due to the differences between the way Python 3 handles the Print command over Python 2, the result isn't as we expected. Using Sublime for the moment, open the hello.py file.




**STEP 9** Since Sublime Text isn't available for the Raspberry Pi, you're going to temporarily leave the Pi for the moment and use Sublime as an example that you don't necessarily need to use the Python IDLE. With the hello.py file open, alter it to include the following:

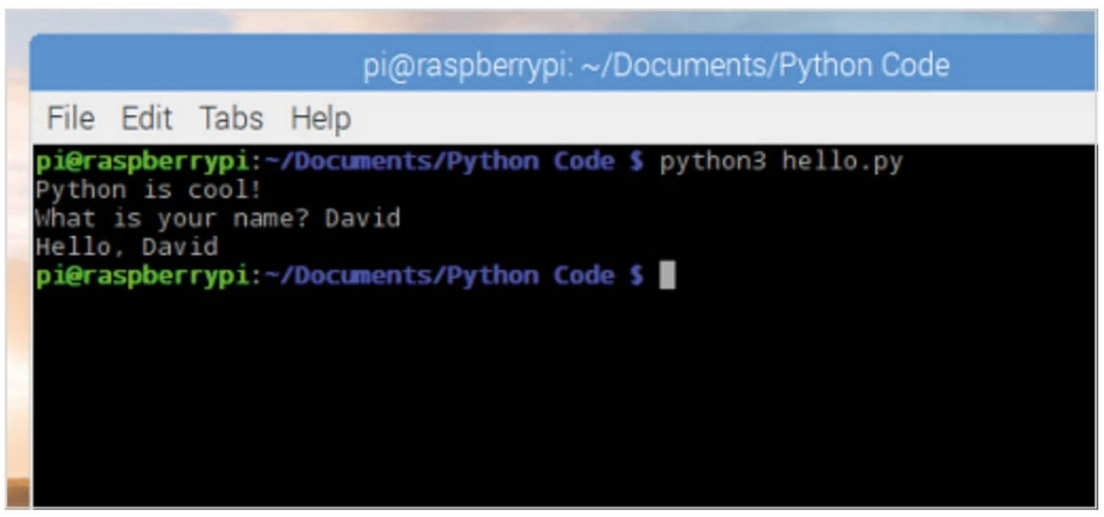
```
name=input(“What is your name? “)
print(“Hello,”, name)
```




**STEP 10** Save the hello.py file and drop back to the command line. Now execute the newly saved code with:

```
python3 hello.py
```

The result will be the original Python is cool! statement, together with the added input command asking you for your name, and displaying it in the command window.



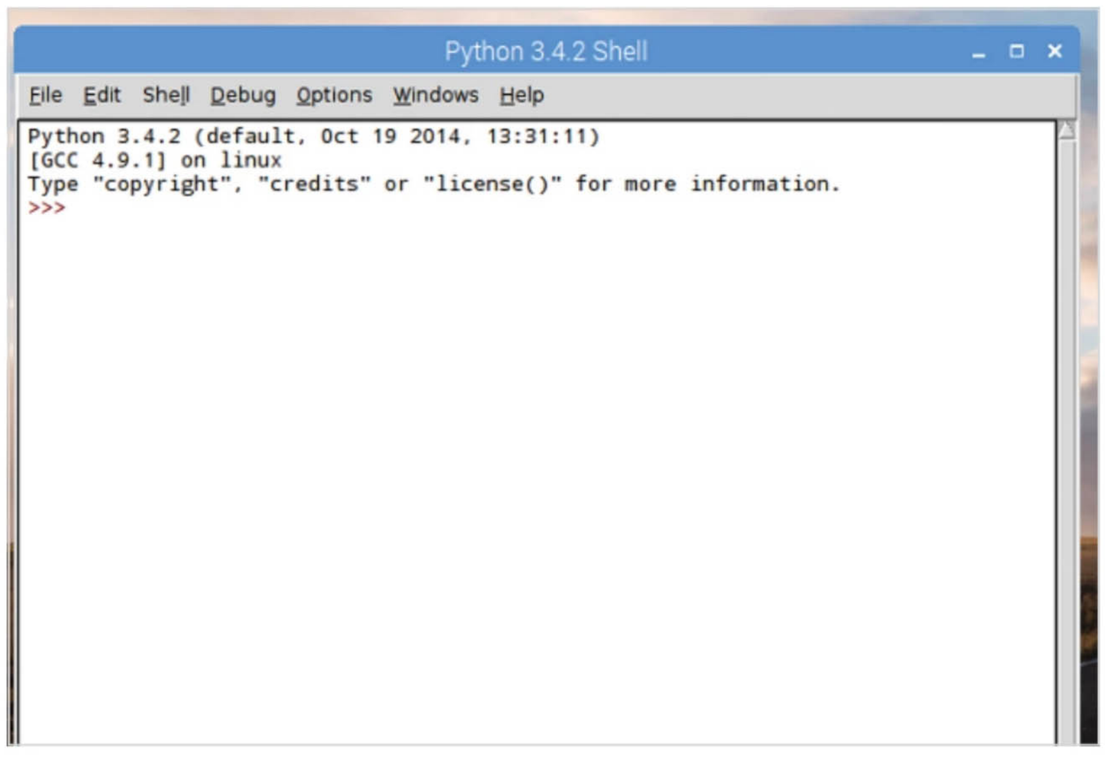
# Numbers and Expressions

We've seen some basic mathematical expressions with Python, simple addition and the like. Let's expand on that now and see just how powerful Python is as a calculator. You can work within the IDLE Shell or in the Editor, whichever you like.

## IT'S ALL MATHS, MAN

**You can get some really impressive results with the mathematical powers of Python; as with most, if not all, programming languages, maths is the driving force behind the code.**

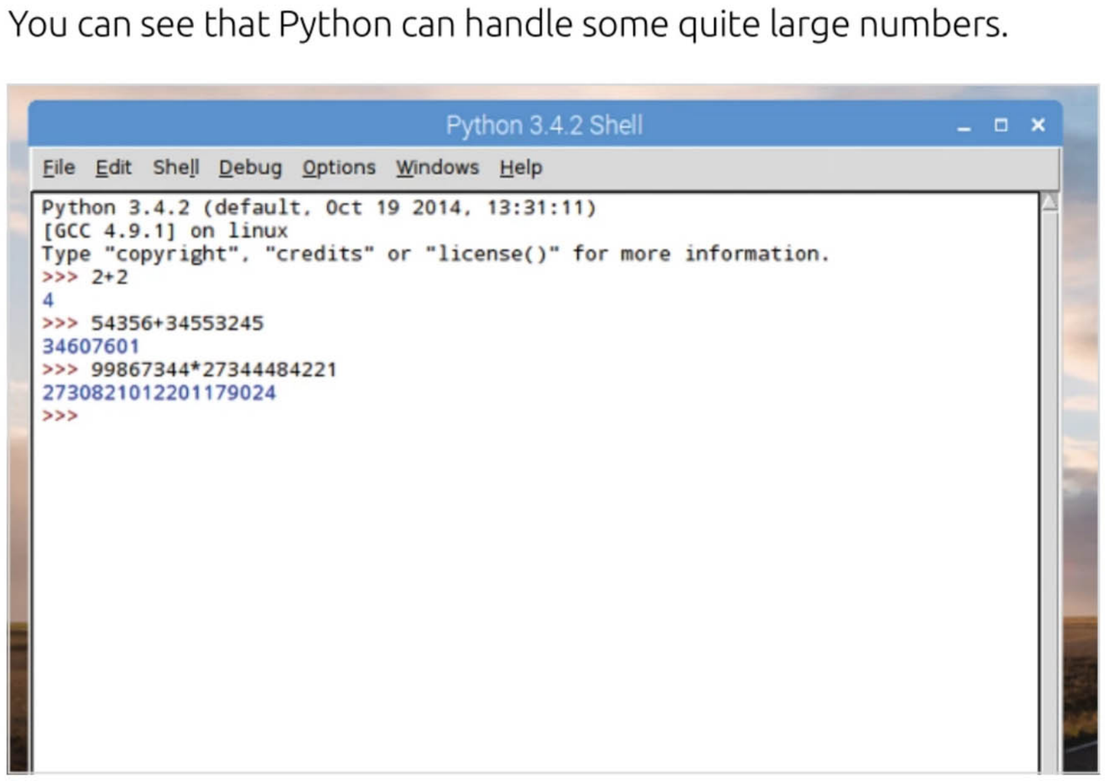
**STEP 1** Open up the GUI version of Python 3, as mentioned you can use either the Shell or the Editor. For the time being, you're going to use the Shell just to warm our maths muscle, which we believe is a small gland located at the back of the brain (or not).




**STEP 2** In the Shell enter the following:

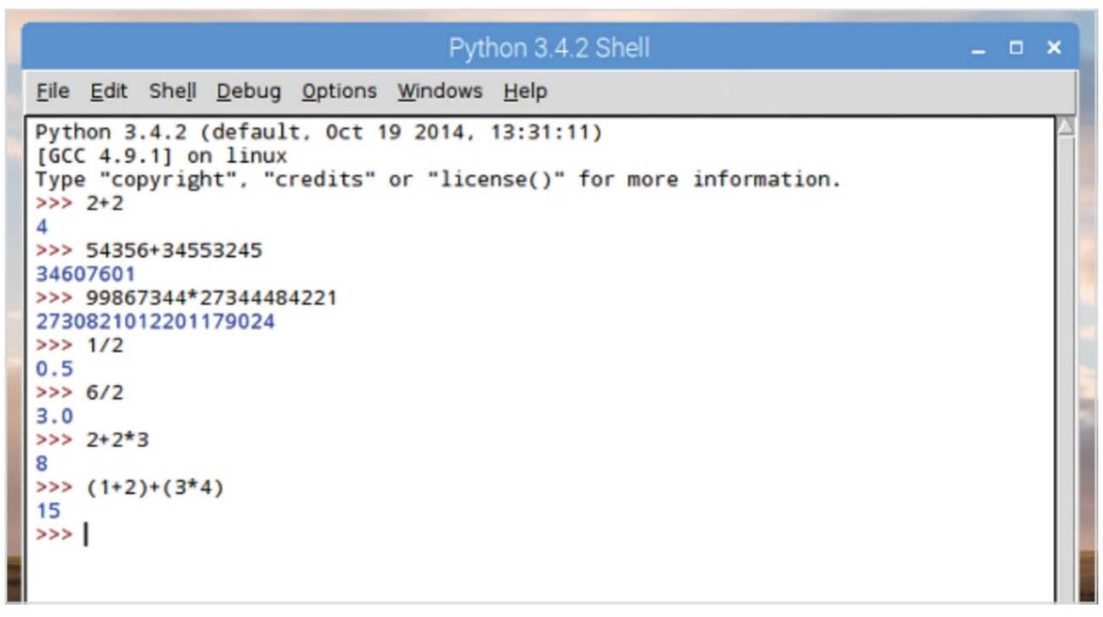
```
2+2
54356+34553245
99867344*27344484221
```

You can see that Python can handle some quite large numbers.




**STEP 3** You can use all the usual mathematical operations: divide, multiply, brackets and so on. Practise with a few, for example:

```
1/2
6/2
2+2*3
(1+2)+(3*4)
```

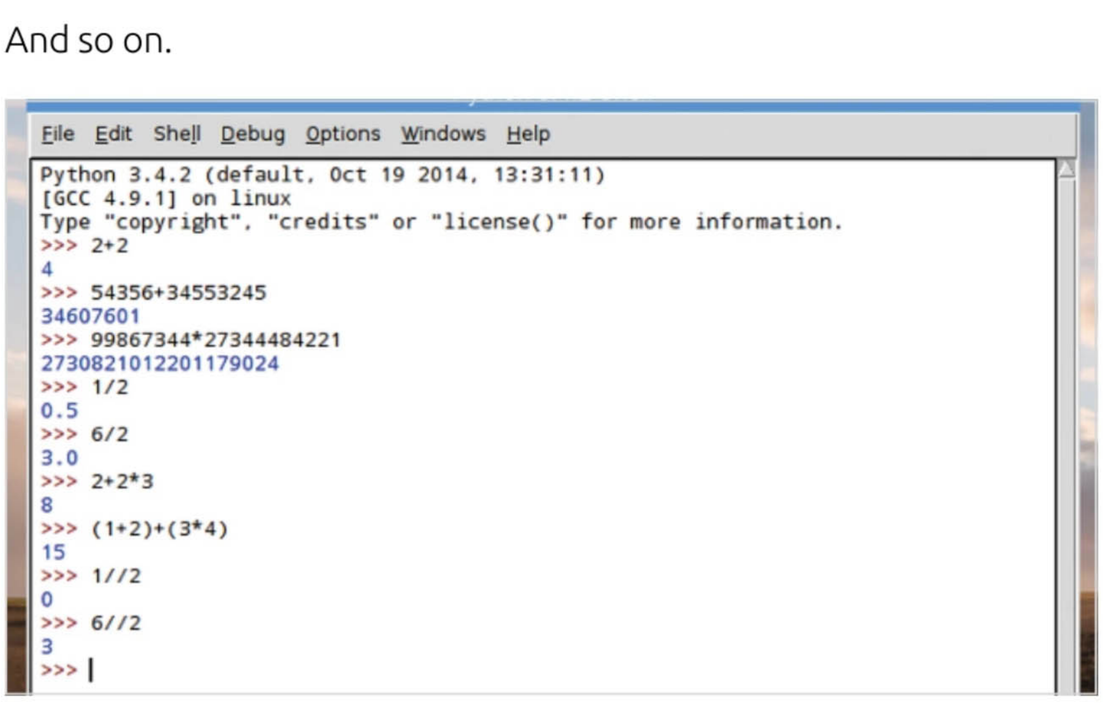



**STEP 4** You've no doubt noticed, division produces a decimal number. In Python these are called floats, or floating point arithmetic. However, if you need an integer as opposed to a decimal answer, then you can use a double slash:

```
1//2
6//2
```

And so on.



## STEP 5

You can also use an operation to see the remainder left over from division. For example:

```
10/3
```

Will display 3.333333333, which is of course 3.3-recurring. If you now enter:

```
10%3
```

This will display 1, which is the remainder left over from dividing 10 into 3.

```
2730821012201179024
>>> 1/2
0.5
>>> 6/2
3.0
>>> 2+2*3
8
>>> (1+2)+(3*4)
15
>>> 1//2
0
>>> 6//2
3
>>> 10/3
3.3333333333333335
>>> 10%3
1
```

## STEP 6

Next up we have the power operator, or exponentiation if you want to be technical. To work out the power of something you can use a double multiplication symbol or double-star on the keyboard:

```
2**3
10**10
```

Essentially, it's 2x2x2 but we're sure you already know the basics behind maths operators. This is how you would work it out in Python.

```
>>> 6/2
3.0
>>> 2+2*3
8
>>> (1+2)+(3*4)
15
>>> 1//2
0
>>> 6//2
3
>>> 10/3
3.3333333333333335
>>> 10%3
1
>>> 2**3
8
>>> 10**10
10000000000
>>>
```

## STEP 7

Numbers and expressions don't stop there. Python has numerous built-in functions to work out sets of numbers, absolute values, complex numbers and a host of mathematical expressions and Pythagorean tongue-twisters. For example, to convert a number to binary, use:

```
bin(3)
```

```
>>> 1/2
0.5
>>> 6/2
3.0
>>> 2+2*3
8
>>> (1+2)+(3*4)
15
>>> 1//2
0
>>> 6//2
3
>>> 10/3
3.3333333333333335
>>> 10%3
1
>>> 2**3
8
>>> 10**10
10000000000
>>> bin(3)
'0b11'
>>>
```

## STEP 8

This will be displayed as '0b11', converting the integer into binary and adding the prefix 0b to the front. If you want to remove the 0b prefix, then you can use:

```
format(3, 'b')
```

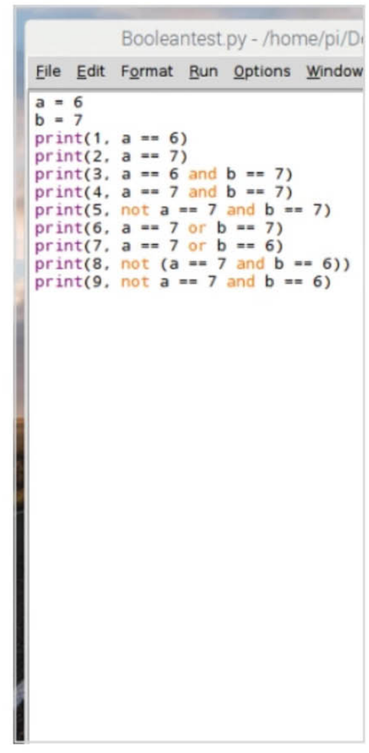
The Format command converts a value, the number 3, to a formatted representation as controlled by the format specification, the 'b' part.

```
>>> 2+2*3
8
>>> (1+2)+(3*4)
15
>>> 1//2
0
>>> 6//2
3
>>> 10/3
3.3333333333333335
>>> 10%3
1
>>> 2**3
8
>>> 10**10
10000000000
>>> bin(3)
'0b11'
>>> format(3,'b')
'11'
>>>
```

## STEP 9

A Boolean Expression is a logical statement that will either be true or false. We can use these to compare data and test to see if it's equal to, less than or greater than. Try this in a New File:

```
a = 6
b = 7
print(1, a == 6)
print(2, a == 7)
print(3, a == 6 and b == 7)
print(4, a == 7 and b == 7)
print(5, not a == 7 and b == 7)
print(6, a == 7 or b == 7)
print(7, a == 7 or b == 6)
print(8, not (a == 7 and b == 6))
print(9, not a == 7 and b == 6)
```




## STEP 10

Execute the code from Step 9, and you can see a series of True or False statements, depending on the result of the two defining values: 6 and 7. It's an extension of what you've looked at, and an important part of programming.

```
Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
1 True
2 False
3 True
4 False
5 True
6 True
7 False
8 True
9 False
>>>
```

# Using Comments

When writing your code, the flow, what each variable does, how the overall program will operate and so on is all inside your head. Another programmer could follow the code line by line but over time, it can become difficult to read.

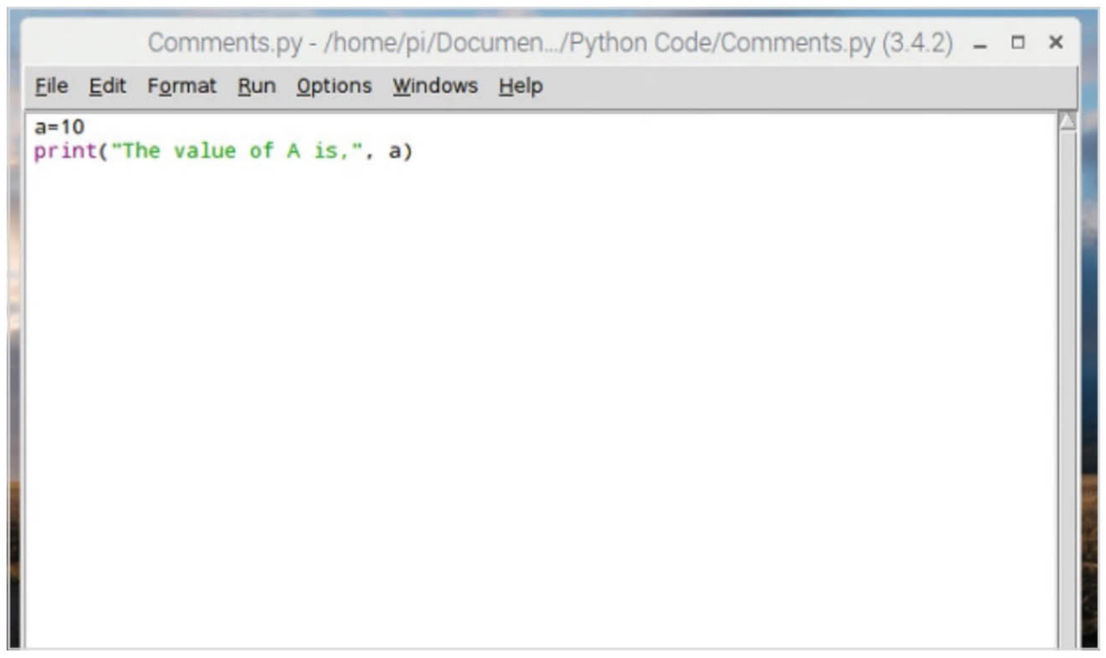
## #COMMENTS!

**Programmers use a method of keeping their code readable by commenting on certain sections. If a variable is used, the programmer comments on what it's supposed to do, for example. It's just good practise.**

**STEP 1** Start by creating a new instance of the IDLE Editor (File > New File) and create a simple variable and print command:

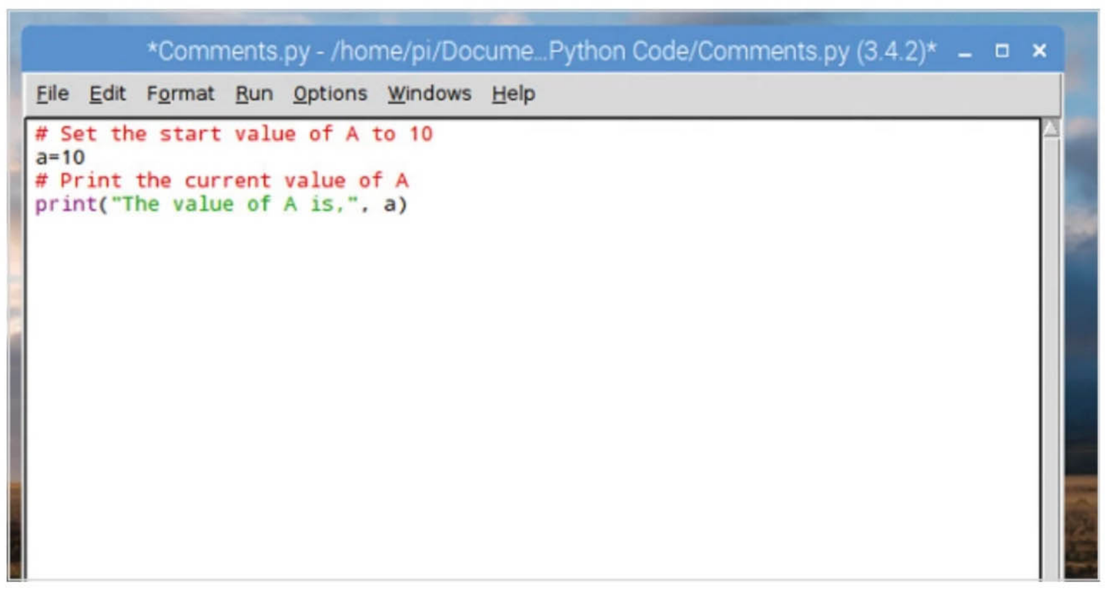
```
a=10
print("The value of A is,", a)
```

Save the file and execute the code.

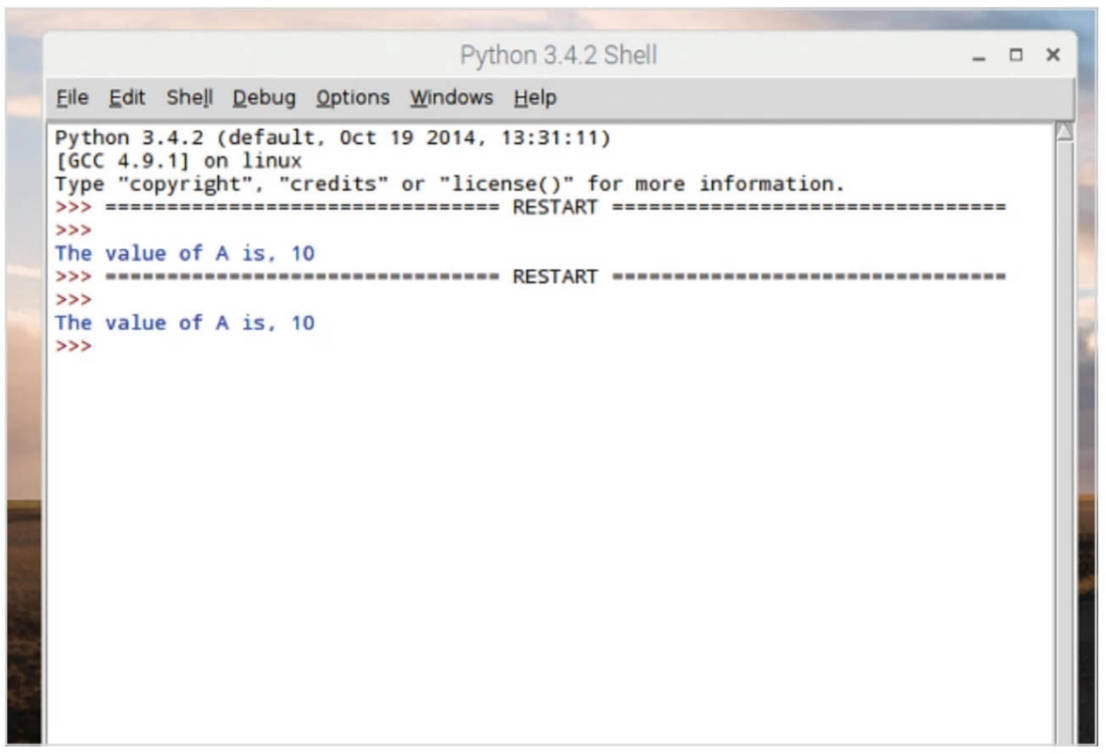



**STEP 2** Running the code will return the line: The value of A is, 10 into the IDLE Shell window, which is what we expected. Now, add some of the types of comments you'd normally see within code:

```
# Set the start value of A to 10
a=10
# Print the current value of A
print("The value of A is,", a)
```

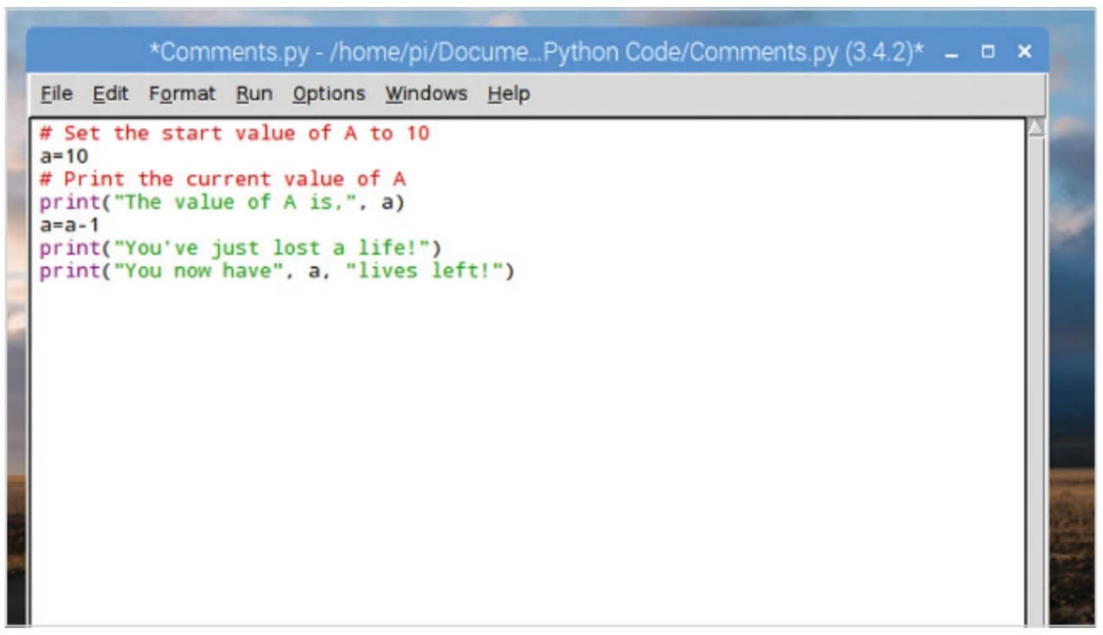



**STEP 3** Resave the code and execute it. You can see that the output in the IDLE Shell is still the same as before, despite the extra lines being added. Simply put, the hash symbol (#) denotes a line of text the programmer can insert to inform them, and others, of what's going on without the user being aware.

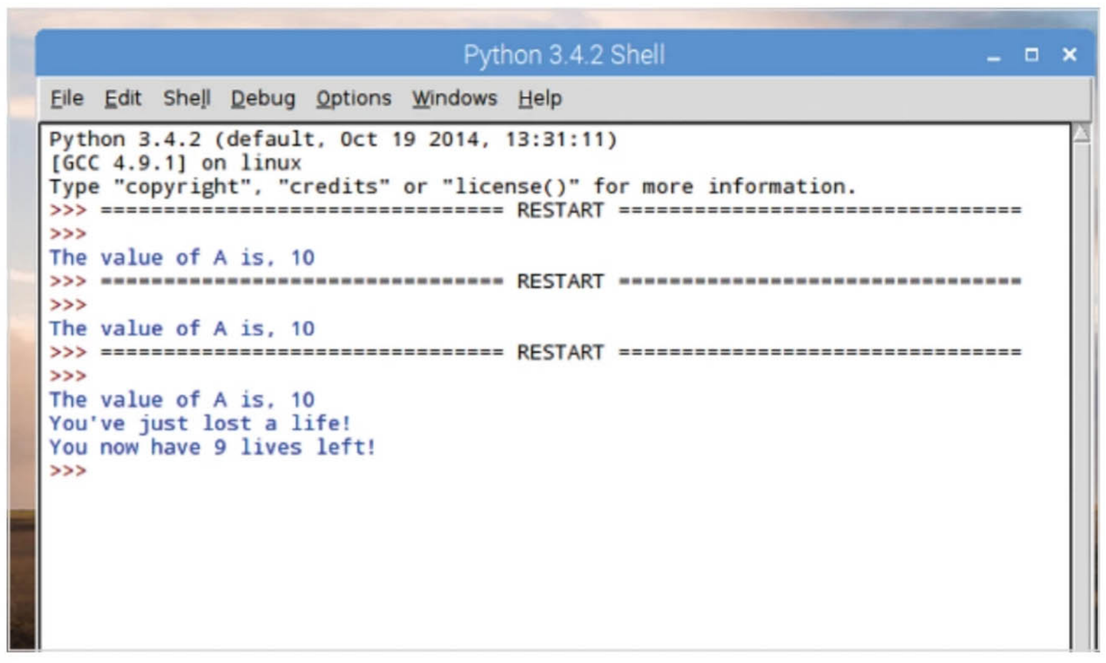



**STEP 4** Let's assume that the variable A that we've created is the number of lives in a game. Every time the player dies, the value is decreased by 1. The programmer could insert a routine along the lines of:

```
a=a-1
print("You've just lost a life!")
print("You now have", a, "lives left!")
```

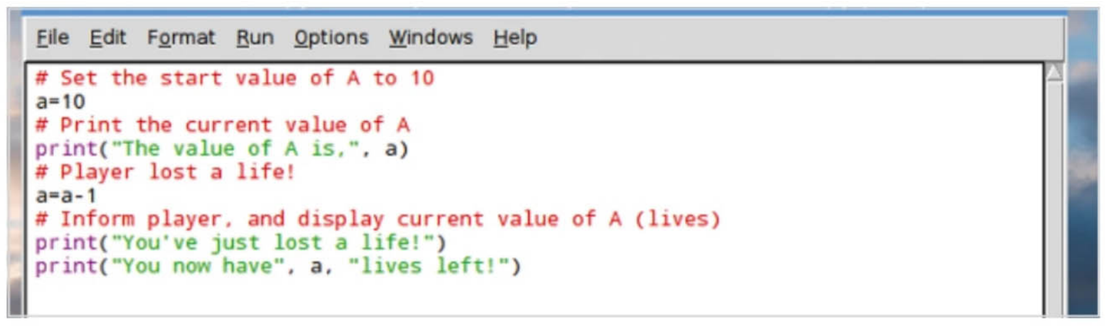


**STEP 5** Whilst we know that the variable A is lives, and that the player has just lost one, a casual viewer or someone checking the code may not know. Imagine for a moment that the code is twenty thousand lines long, instead of just our seven. You can see how handy comments are.



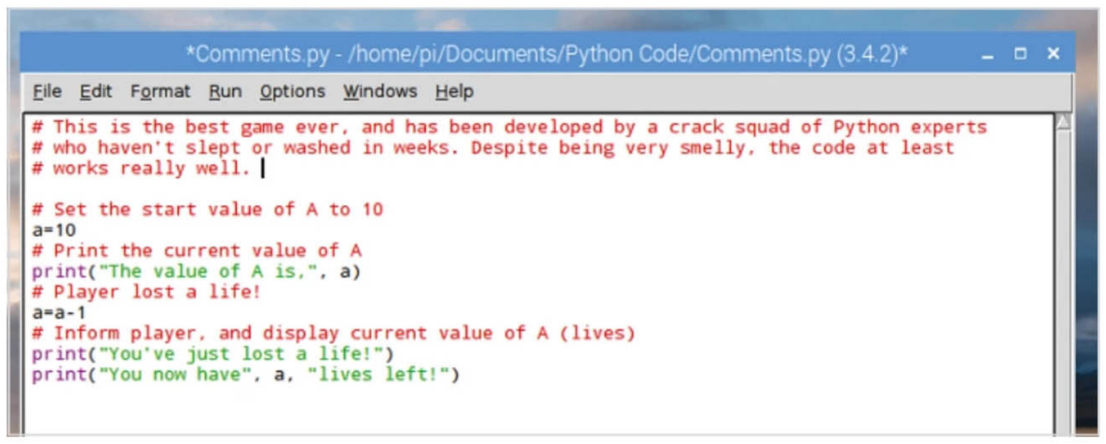

**STEP 6** Essentially, the new code together with comments could look like:

```
# Set the start value of A to 10
a=10
# Print the current value of A
print(“The value of A is,”, a)
# Player lost a life!
a=a-1
# Inform player, and display current value of A (lives)
print(“You’ve just lost a life!”)
print(“You now have”, a, “lives left!”)
```




**STEP 7** You can use comments in different ways. For example, Block Comments are a large section of text that details what’s going on in the code, such as telling the code reader what variables you’re planning on using:

```
# This is the best game ever, and has been
developed by a crack squad of Python experts
# who haven’t slept or washed in weeks. Despite
being very smelly, the code at least
# works really well.
```

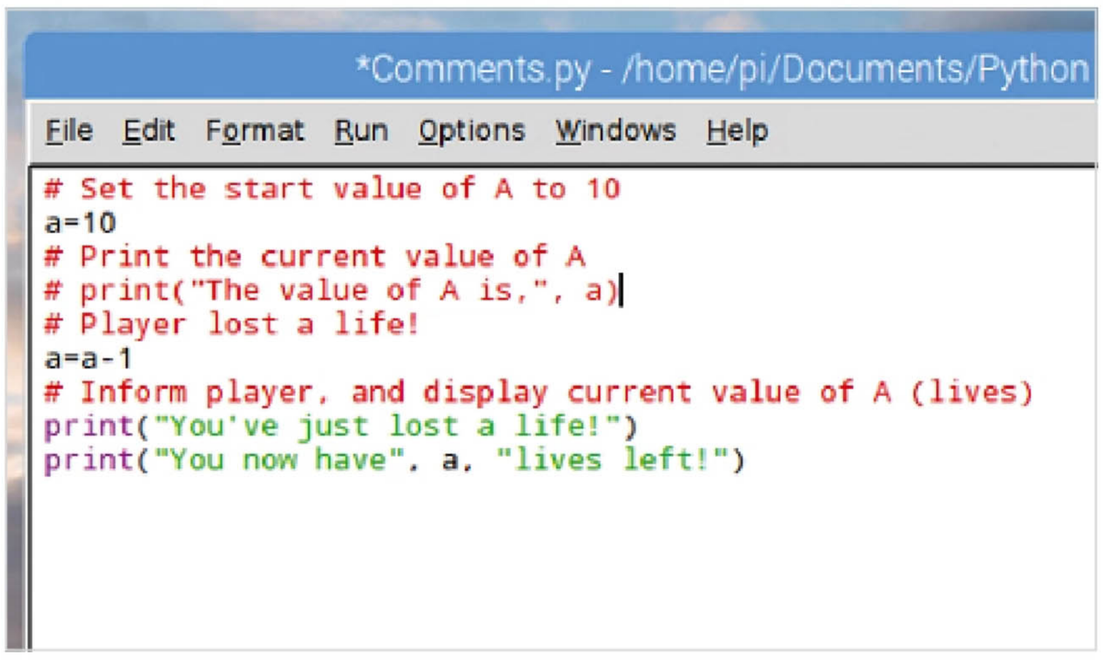



**STEP 8** Inline comments are comments that follow a section of code. Take our examples from above, instead of inserting the code on a separate line, we could use:

```
a=10 # Set the start value of A to 10
print(“The value of A is,”, a) # Print the current
value of A
a=a-1 # Player lost a life!
print(“You’ve just lost a life!”)
print(“You now have”, a, “lives left!”) # Inform
player, and display current value of A (lives)
```

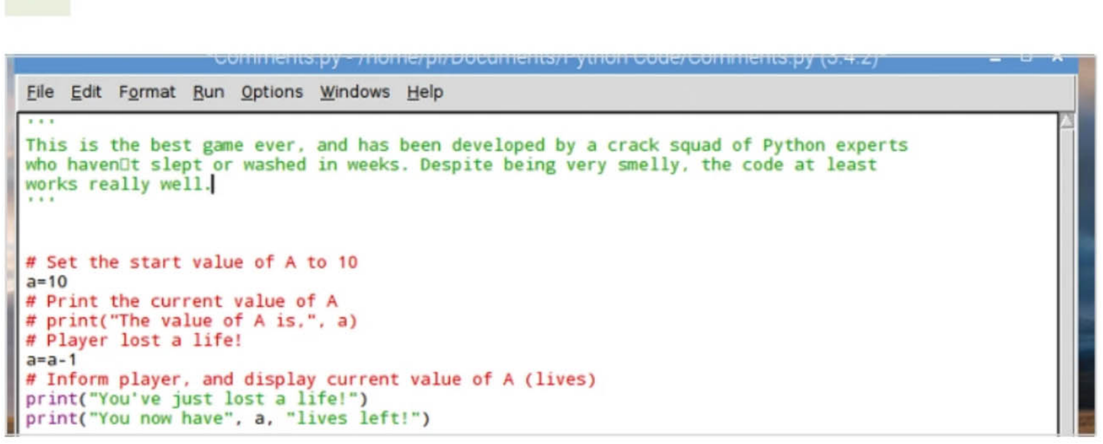
**STEP 9** The comment, the hash symbol, can also be used to comment out sections of code you don’t want to be executed in your program. For instance, if you wanted to remove the first print statement, you would use:

```
# print(“The value of A is,”, a)
```




**STEP 10** You also use three single quotes to comment out a Block Comment or multi-line section of comments. Place them before and after the areas you want to comment for them to work:

```
‘’’
This is the best game ever, and has been developed
by a crack squad of Python experts who haven’t
slept or washed in weeks. Despite being very
smelly, the code at least works really well.
’’’
```



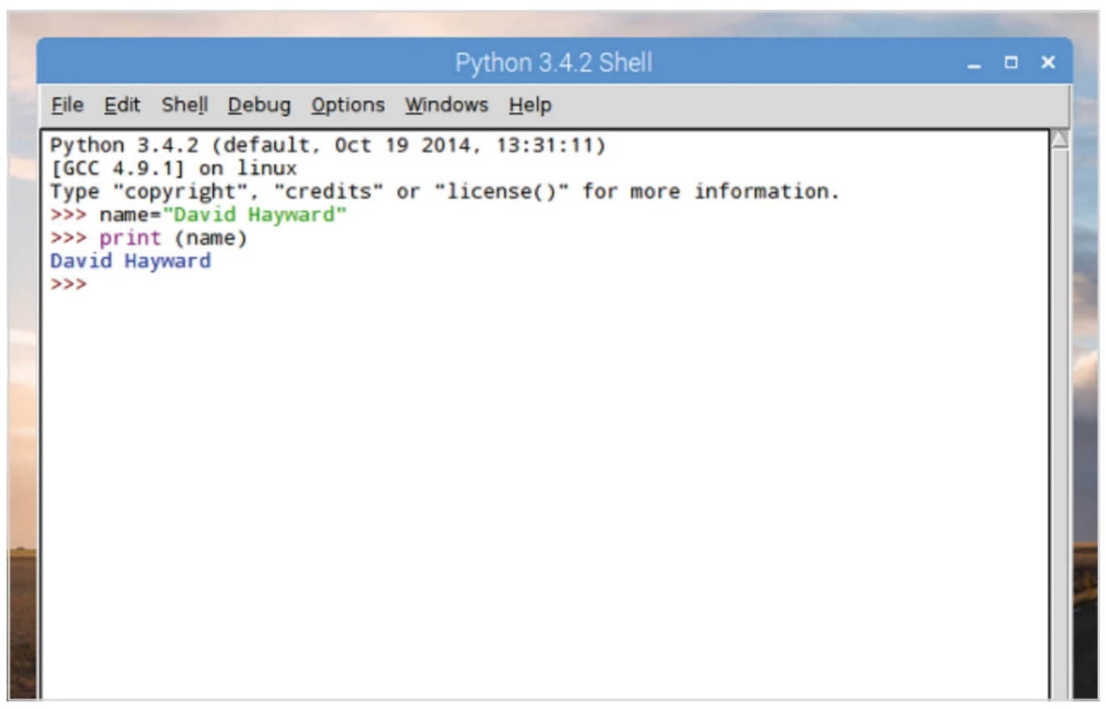
# Working with Variables

We've seen some examples of variables in our Python code already but it's always worth going through the way they operate and how Python creates and assigns certain values to a variable.

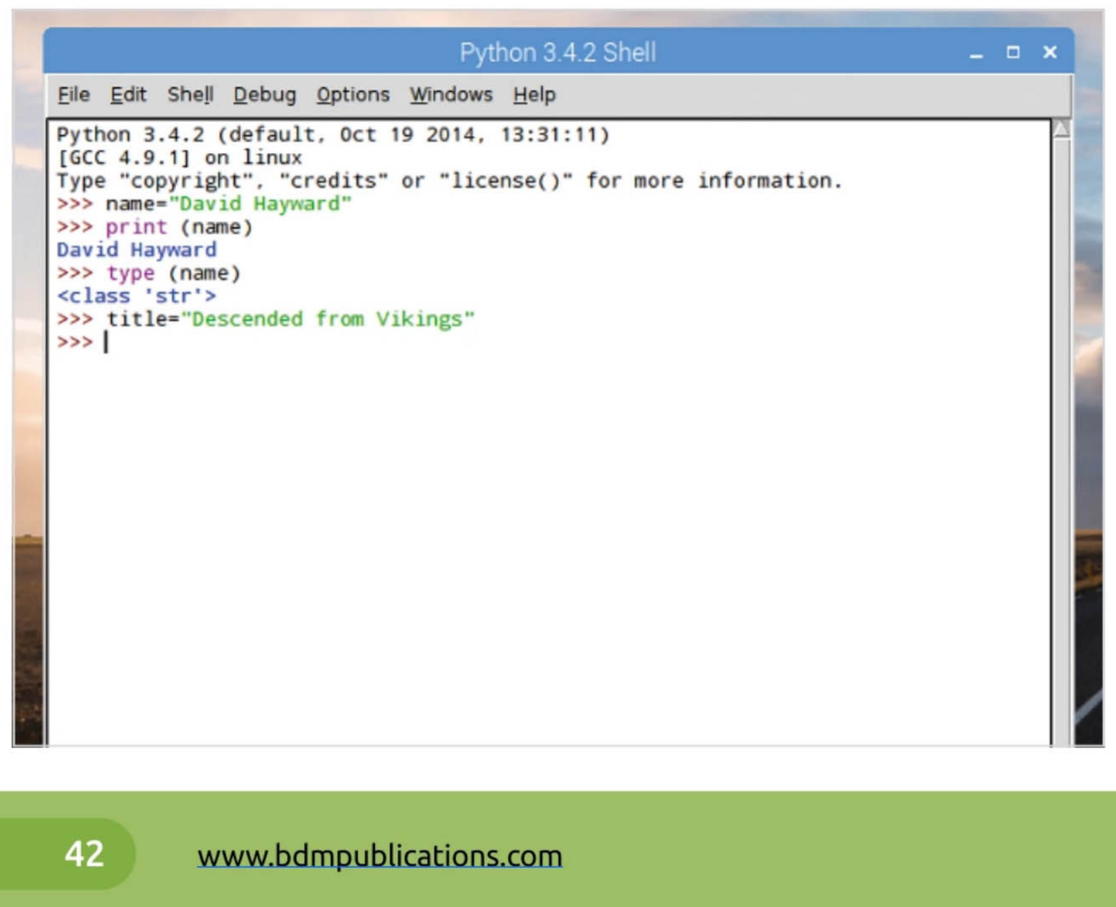
## VARIOUS VARIABLES

**You'll be working with the Python 3 IDLE Shell in this tutorial. If you haven't already, open Python 3 or close down the previous IDLE Shell to clear up any old code.**

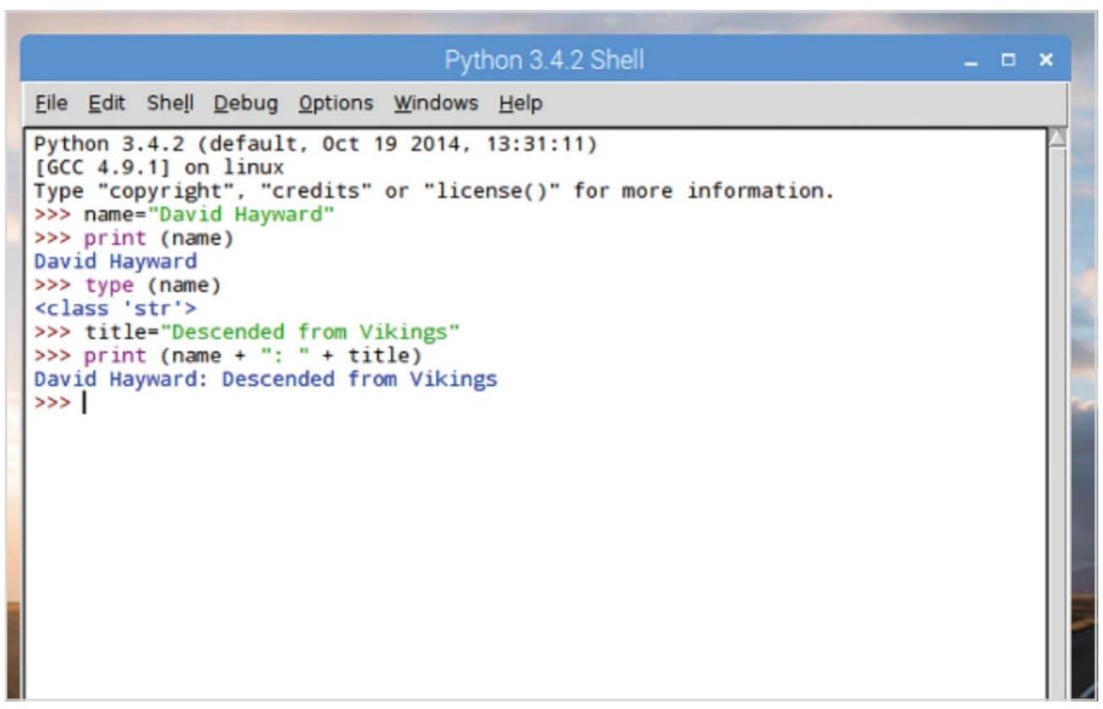
**STEP 1** In some programming languages you're required to use a dollar sign to denote a string, which is a variable made up of multiple characters, such as a name of a person. In Python this isn't necessary. For example, in the Shell enter: `name="David Hayward"` (or use your own name, unless you're also called David Hayward).




**STEP 2** You can check the type of variable in use by issuing the **type ()** command, placing the name of the variable inside the brackets. In our example, this would be: `type (name)`. Add a new string variable: `title="Descended from Vikings"`.




**STEP 3** You've seen previously that variables can be concatenated using the plus symbol between the variable names. In our example we can use: `print (name + ": " + title)`. The middle part between the quotations allows us to add a colon and a space, as variables are connected without spaces, so we need to add them manually.




**STEP 4** You can also combine variables within another variable. For example, to combine both name and title variables into a new variable we use:

```
character=name + ": " + title
```

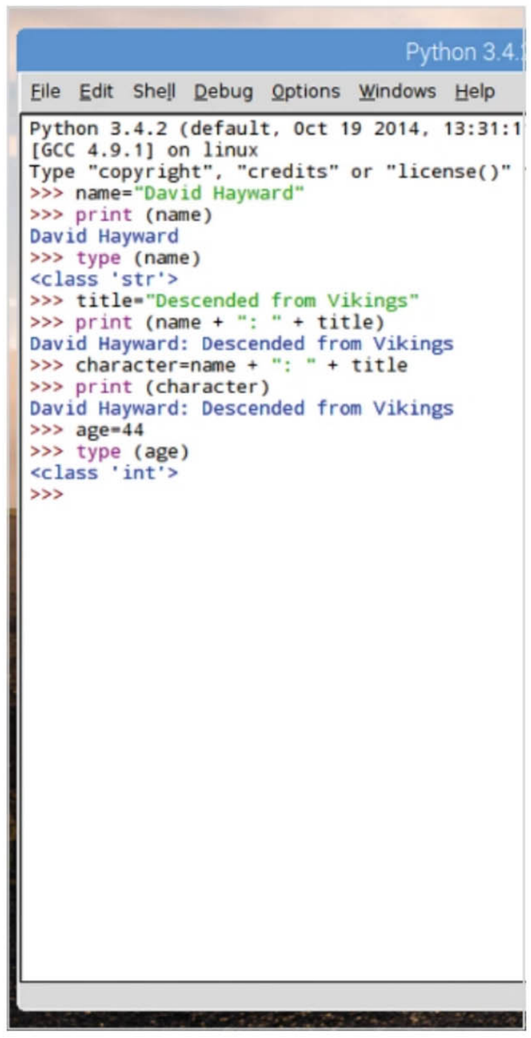
Then output the content of the new variable as:

```
print (character)
```

Numbers are stored as different variables:

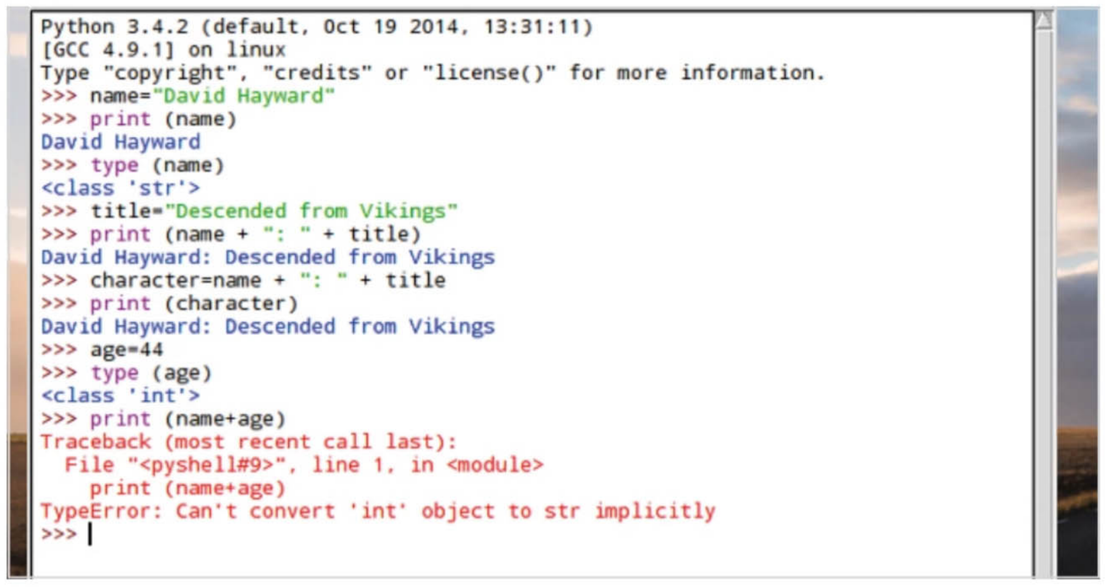
```
age=44
Type (age)
```

Which are integers, as we know.



**STEP 5** However, you can't combine both strings and integer type variables in the same command, as you would a set of similar variables. You need to either turn one into the other or vice versa. When you do try to combine both, you get an error message:

```
print (name + age)
```




**STEP 6** This is a process known as TypeCasting. The Python code is:

```
print (character + " is " + str(age) + " years old.")
```

or you can use:

```
print (character, "is", age, "years old.")
```

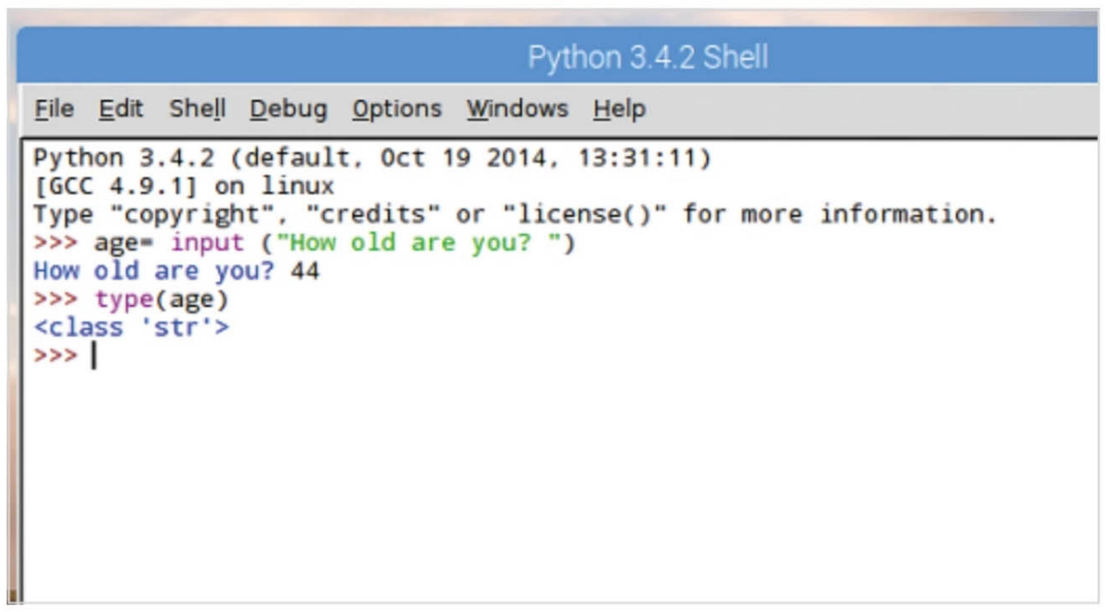
Notice again that in the last example, you don't need the spaces between the words in quotes as the commas treat each argument to print separately.

```
>>> print (name + age)
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    print (name + age)
TypeError: Can't convert 'int' object to str implicitly
>>> print (character + " is " + str(age) + " years old.")
David Hayward: Descended from Vikings is 44 years old.
>>> print (character, "is", age, "years old.")
David Hayward: Descended from Vikings is 44 years old.
>>>
>>>
```

**STEP 7** Another example of TypeCasting is when you ask for input from the user, such as a name. for example, enter:

```
age= input ("How old are you? ")
```

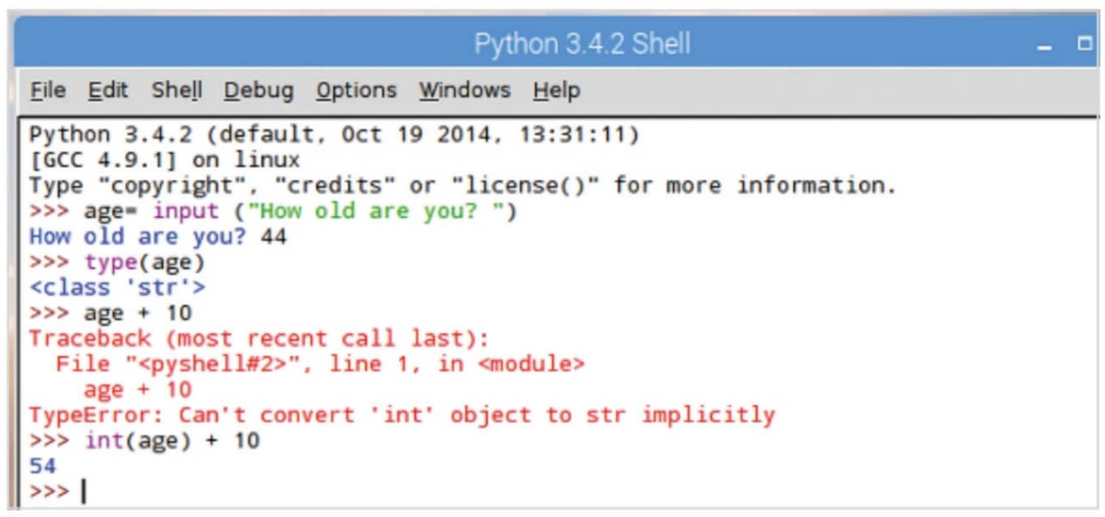
All data stored from the Input command is stored as a string variable.




**STEP 8** This presents a bit of a problem when you want to work with a number that's been inputted by the user, as age + 10 won't work due to being a string variable and an integer. Instead, you need to enter:

```
int(age) + 10
```

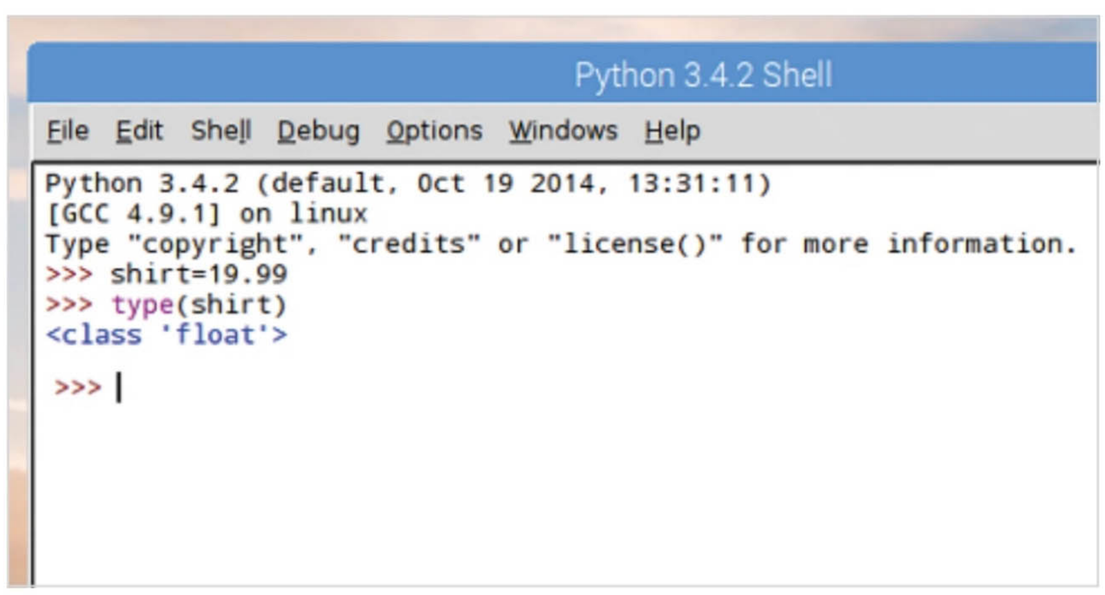
This will TypeCast the age string into an integer that can be worked with.



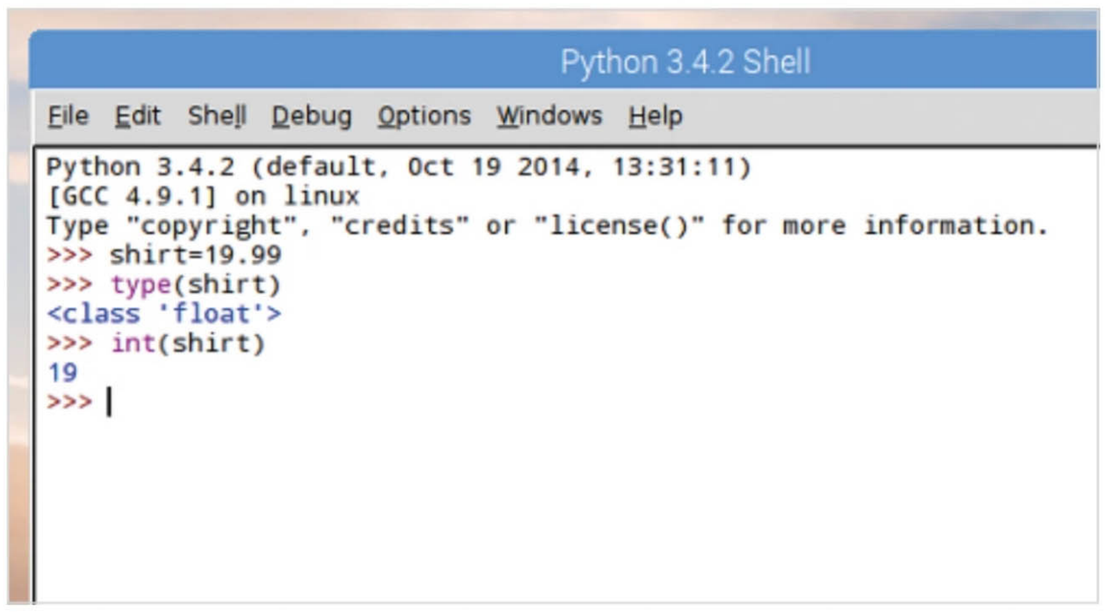

**STEP 9** The use of TypeCasting is also important when dealing with floating point arithmetic; remember: numbers that have a decimal point in them. For example, enter:

```
shirt=19.99
```

Now enter `type(shirt)` and you'll see that Python has allocated the number as a 'float', because the value contains a decimal point.




**STEP 10** When combining integers and floats Python usually converts the integer to a float, but should the reverse ever be applied it's worth remembering that Python doesn't return the exact value. When converting a float to an integer, Python will always round down to the nearest integer, called truncating; in our case instead of 19.99 it becomes 19.



# User Input

We've seen some basic user interaction with the code from a few of the examples earlier, so now would be a good time to focus solely on how you would get information from the user then store and present it.

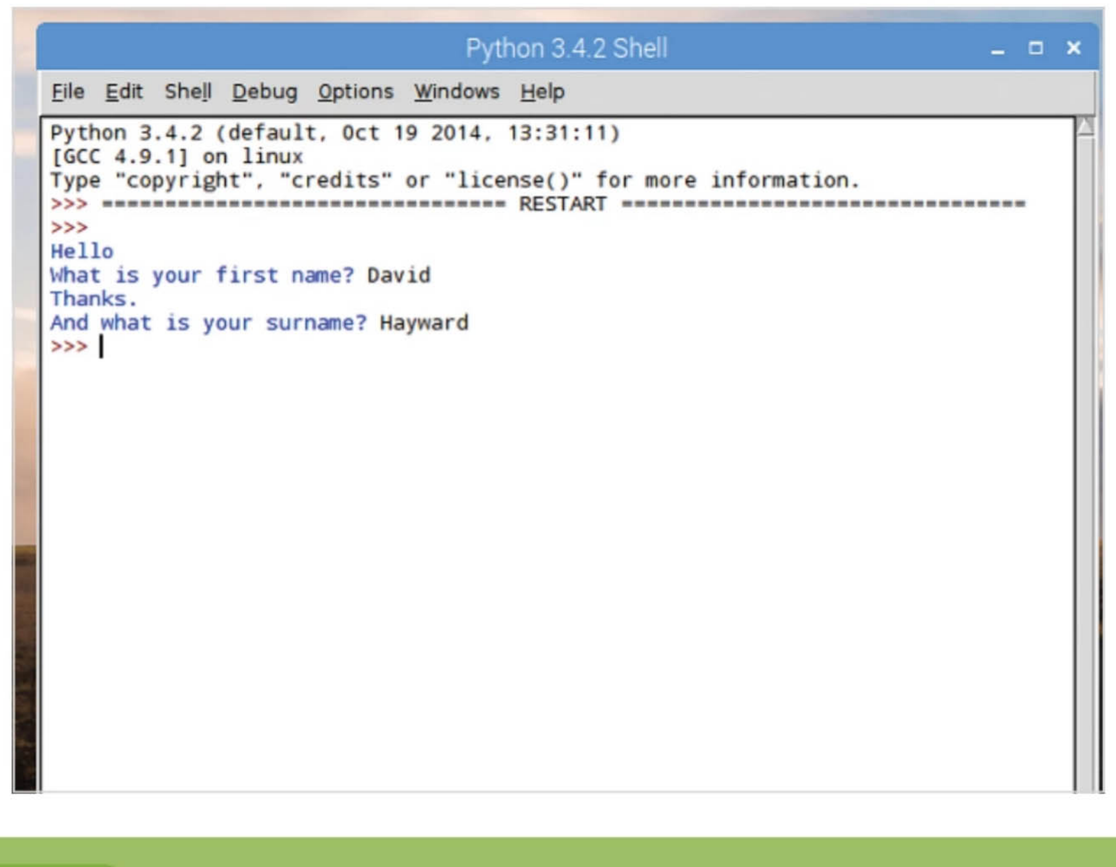
## USER FRIENDLY

**The type of input you want from the user will depend greatly on the type of program you're coding. For example, a game may ask for a character's name, whereas a database can ask for personal details.**

**STEP 1** If it's not already, open the Python 3 IDLE Shell, and start a New File in the Editor. Let's begin with something really simple, enter:

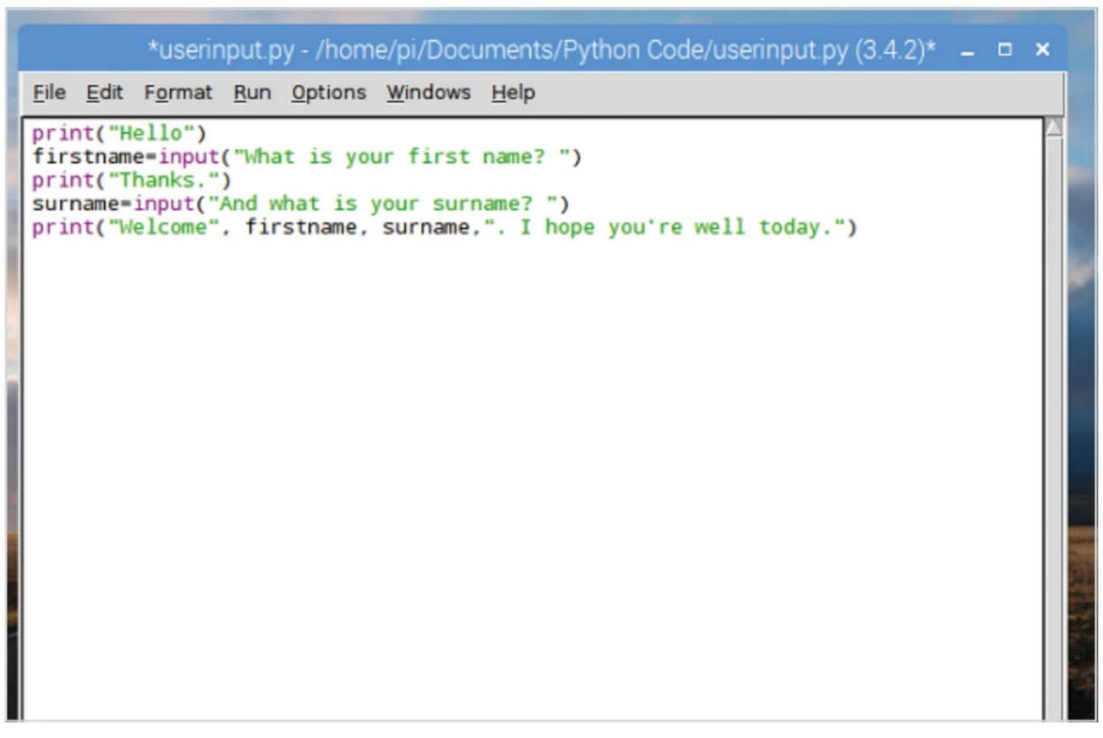
```
print("Hello")
firstname=input("What is your first name? ")
print("Thanks.")
surname=input("And what is your surname? ")
```

**STEP 2** Save and execute the code, and as you already no doubt suspected, in the IDLE Shell the program will ask for your first name, storing it as the variable firstname, followed by your surname; also stored in its own variable (surname).



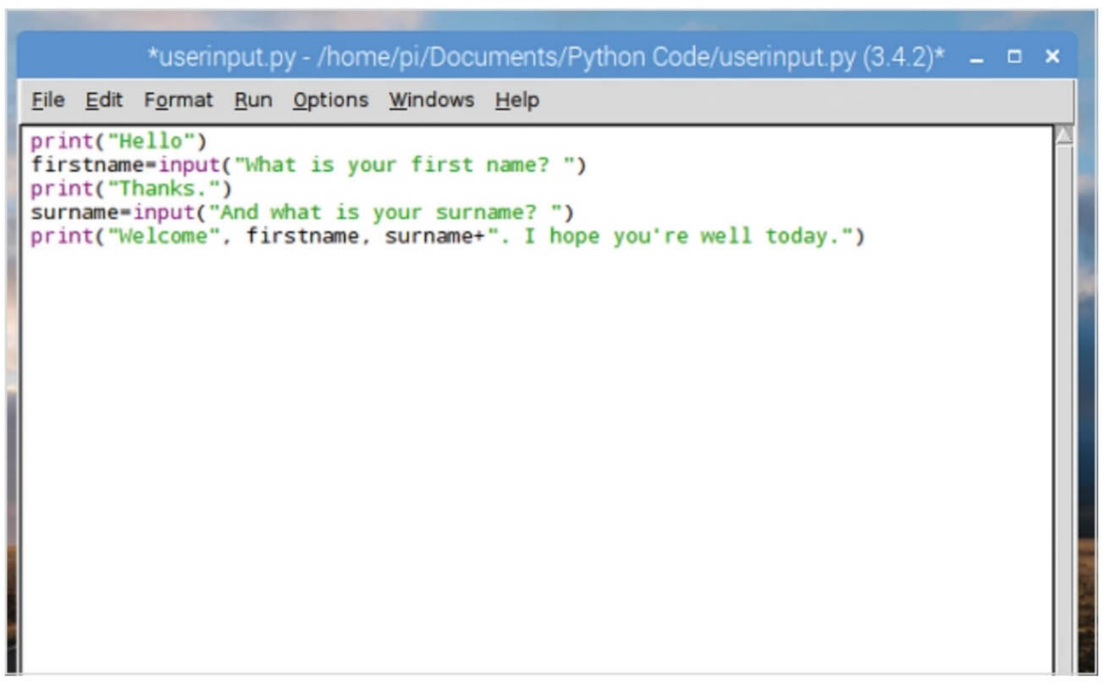

**STEP 3** Now that we have the user's name stored in a couple of variables we can call them up whenever we want:

```
print("Welcome", firstname, surname, ". I hope you're well today.")
```



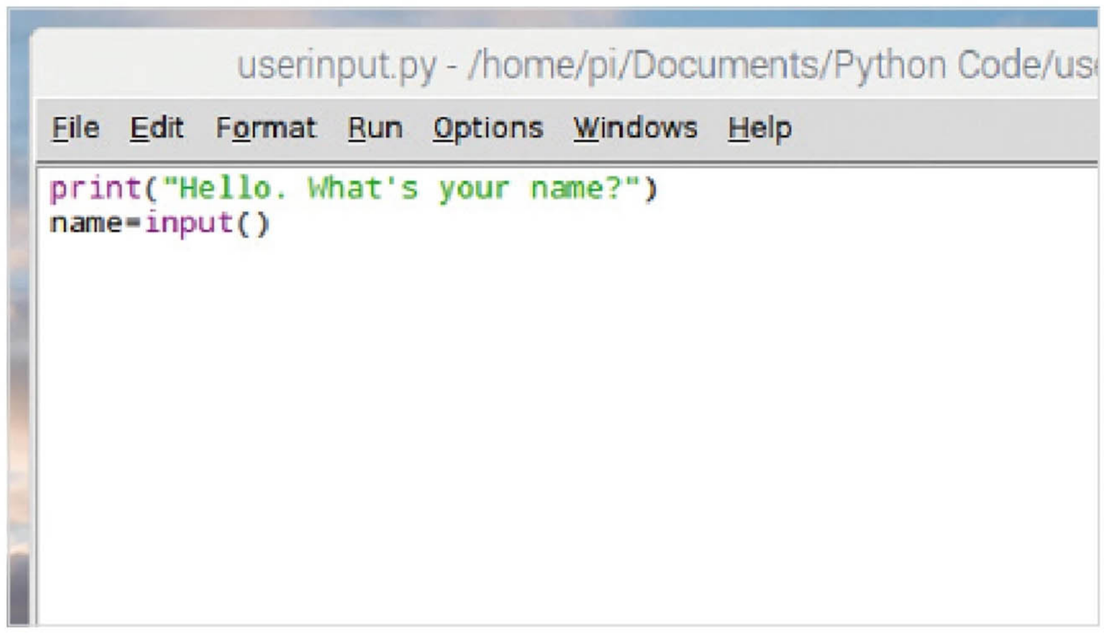

**STEP 4** Run the code and you can see a slight issue, the full stop after the surname follows a blank space. To eliminate that we can add a plus sign instead of the comma in the code:

```
print("Welcome", firstname, surname+". I hope you're well today.")
```



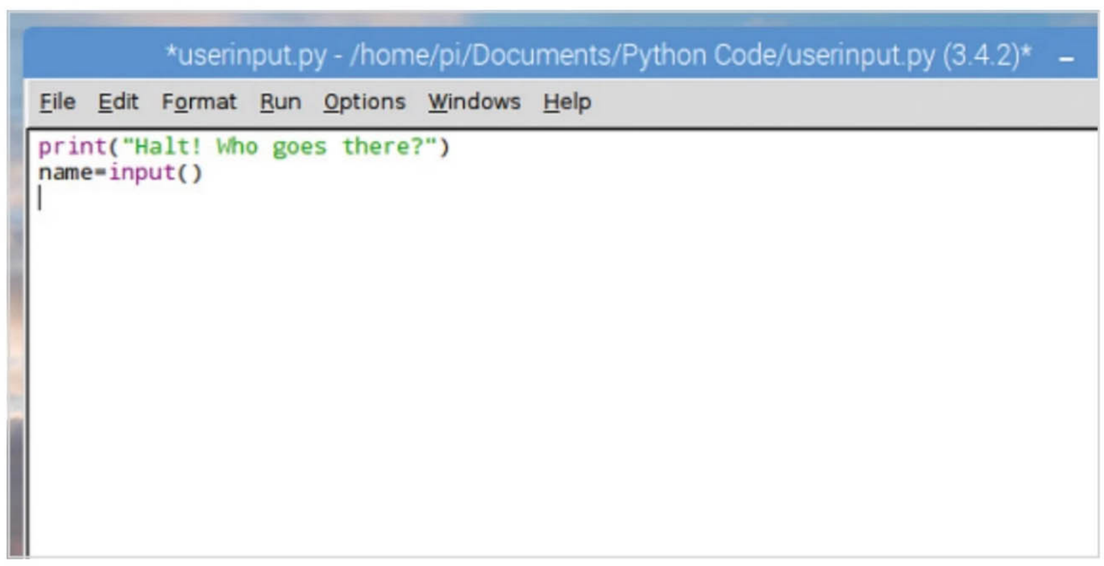
**STEP 5** You don't always have to include quoted text within the input command. For example, you can ask the user their name, and have the input in the line below:

```
print("Hello. What's your name?")
name=input()
```



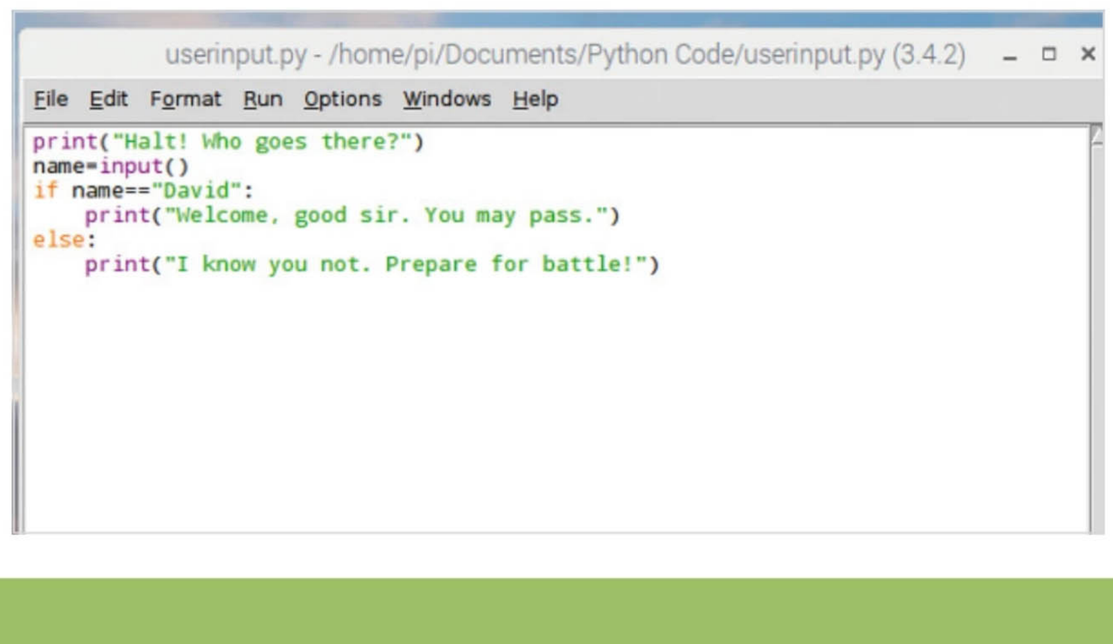

**STEP 6** The code from the previous step is often regarded as being a little neater than having a lengthy amount of text in the input command, but it's not a rule that's set in stone, so do as you like in these situations. Expanding on the code, try this:

```
print("Halt! Who goes there?")
name=input()
```

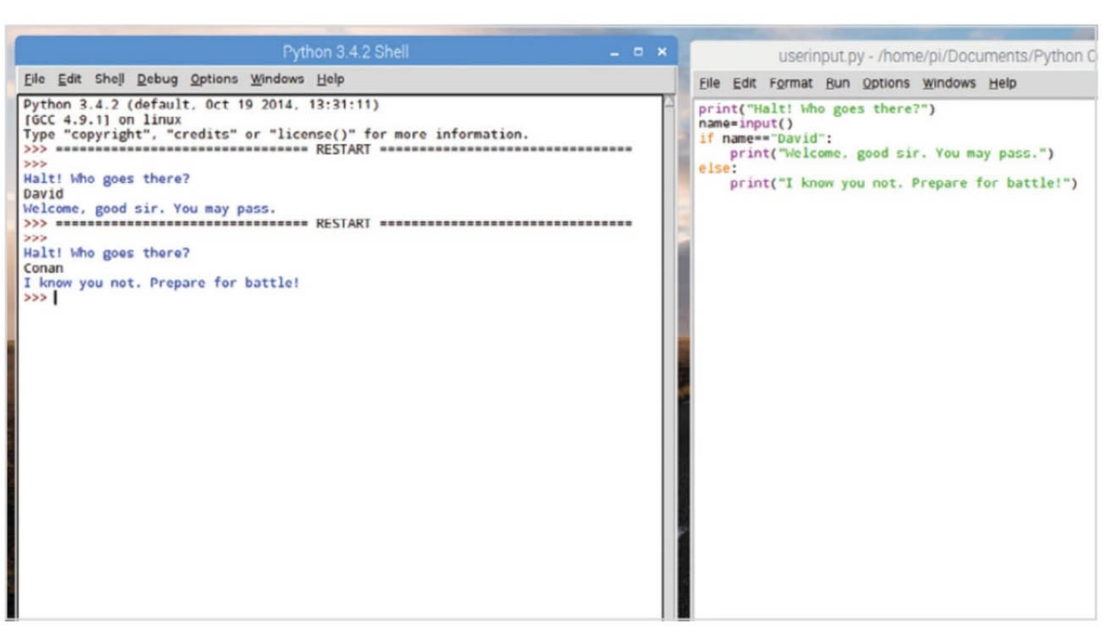



**STEP 7** It's a good start to a text adventure game, perhaps? Now you can expand on it and use the raw input from the user to flesh out the game a little:

```
if name=="David":
  print("Welcome, good sir. You may pass.")
else:
  print("I know you not. Prepare for battle!")
```

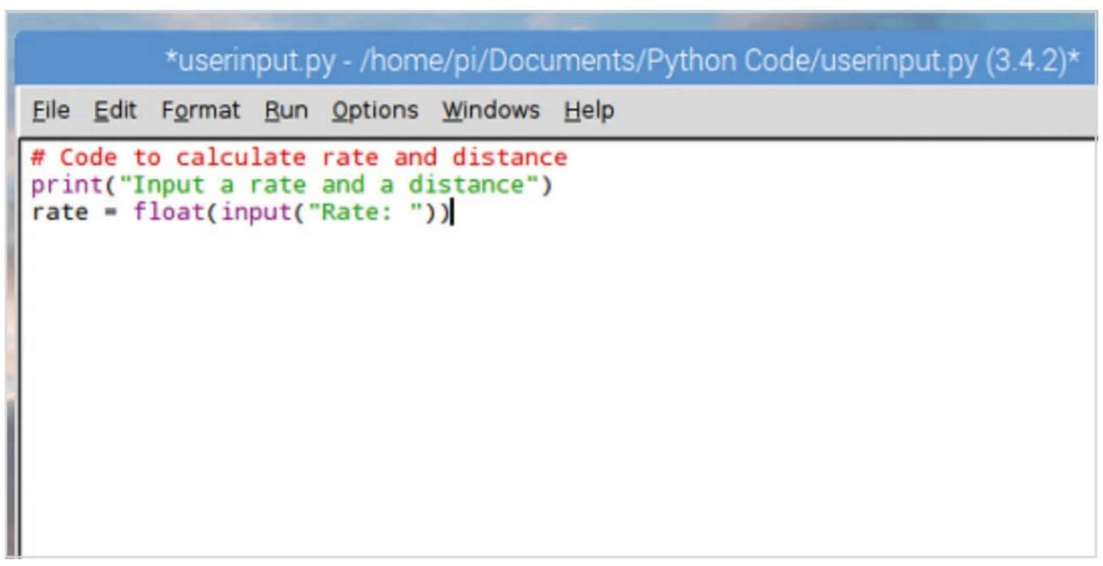



**STEP 8** What you've created here is a condition, which we will cover soon. In short, we're using the input from the user and measuring it against a condition. So, if the user enters David as their name, the guard will allow them to pass unhindered. Else, if they enter a name other than David, the guard challenges them to a fight.




**STEP 9** Just as you learned previously, any input from a user is automatically a string, so you need to apply a TypeCast in order to turn it into something else. This creates some interesting additions to the input command. For example:

```
# Code to calculate rate and distance
print("Input a rate and a distance")
rate = float(input("Rate: "))
```

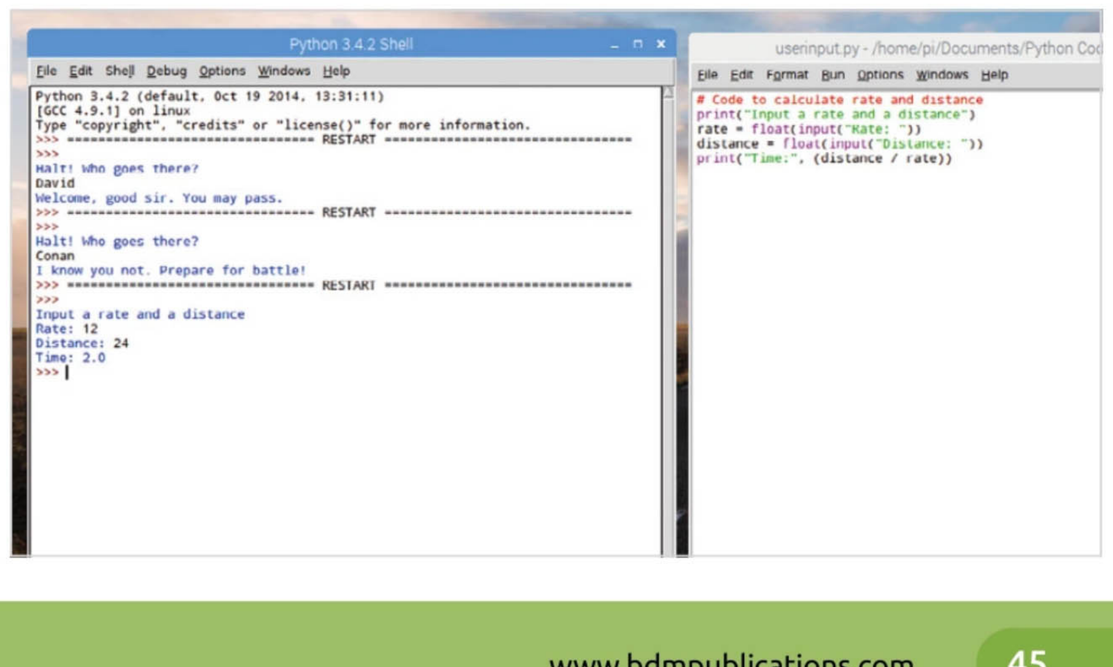



**STEP 10** To finalise the rate and distance code, we can add:

```
distance = float(input("Distance: "))
print("Time:", (distance / rate))
```

Save and execute the code and enter some numbers. Using the float(input element, we've told Python that anything entered is a floating point number rather than a string.



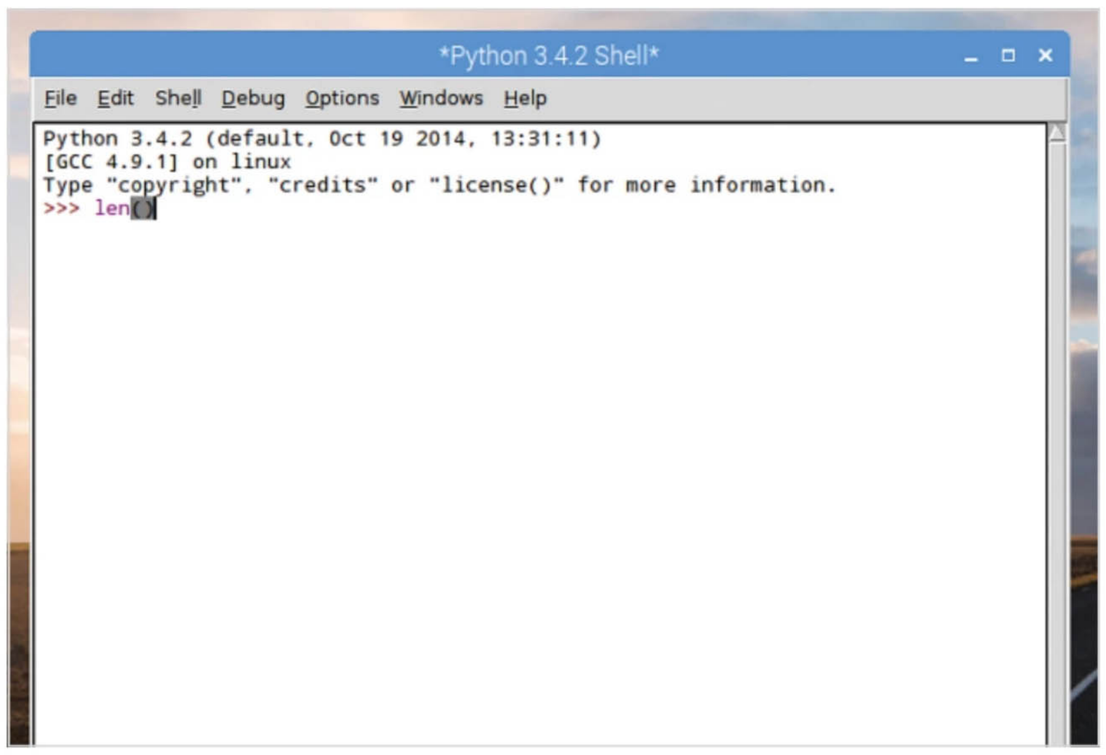
# Creating Functions

Now that you've mastered the use of variables and user input, the next step is to tackle functions. You've already used a few functions, such as the print command but Python enables you to define your own functions.

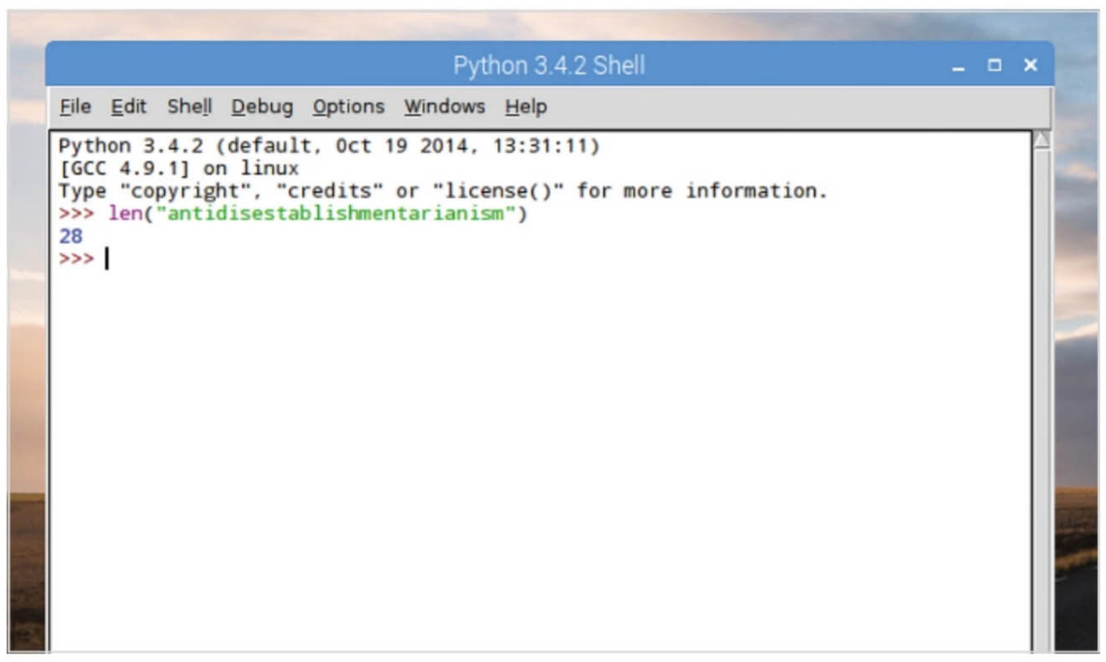
## FUNKY FUNCTIONS

**A function is a command that you enter into Python to do something. It's a little piece of self-contained code that takes data, works on it and then returns the result.**

**STEP 1** It's not just data that a function works on. They can do all manner of useful things in Python, such as sort data, change items from one format to another and check the length or type of items. Basically, a function is a short word that's followed by brackets. For example, **len()**, **list()** or **type()**.



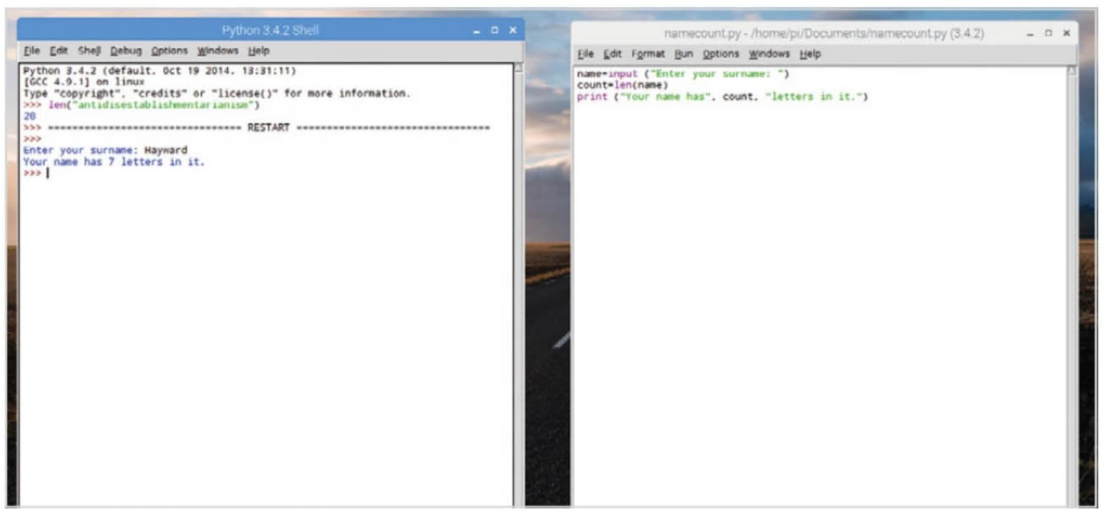

**STEP 2** A function takes data, usually a variable, works on it depending on what the function is programmed to do and returns the end value. The data being worked on goes inside the brackets, so if you wanted to know how many letters are in the word antidisestablishmentarianism, then you'd enter: `len("antidisestablishmentarianism")` and the number 28 would return.



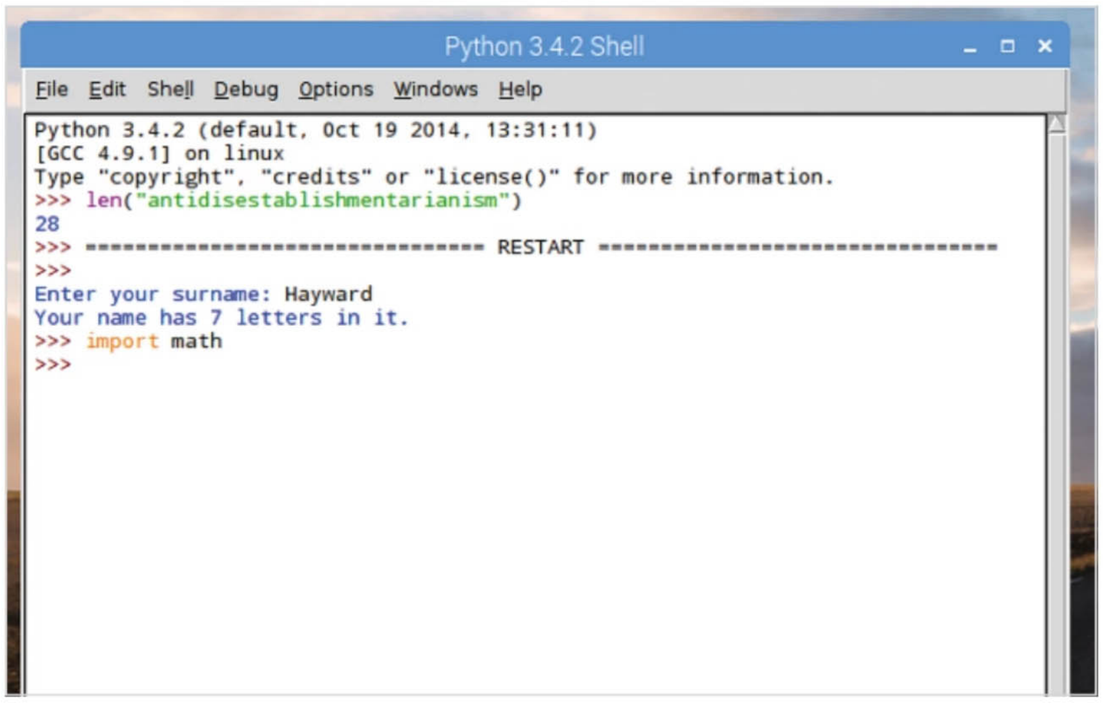

**STEP 3** You can pass variables through functions in much the same manner. Let's assume you want the number of letters in a person's surname, you could use the following code (enter the text editor for this example):

```
name=input ("Enter your surname: ")
count=len(name)
print ("Your surname has", count, "letters in
it.")
```

Press F5 and save the code to execute it.




**STEP 4** Python has tens of functions built into it, far too many to get into in the limited space available here. However, to view the list of built-in functions available to Python 3, navigate to **www.docs.python.org/3/library/functions.html**. These are the predefined functions, but since users have created many more, they're not the only ones available.



**STEP 5** Additional functions can be added to Python through modules. Python has a vast range of modules available that can cover numerous programming duties. They add functions and can be imported as and when required. For example, to use advanced mathematics functions enter:

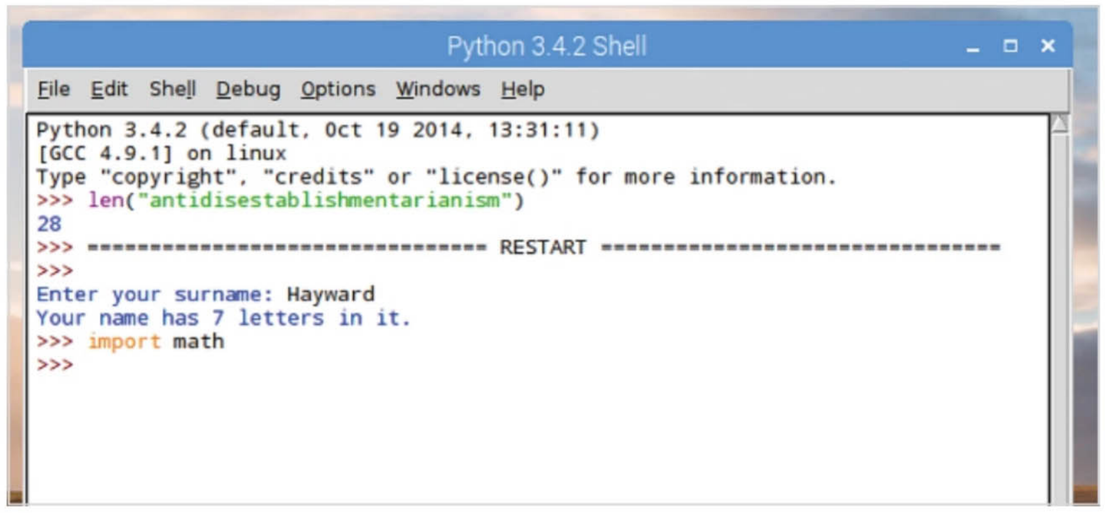
```
import math
```

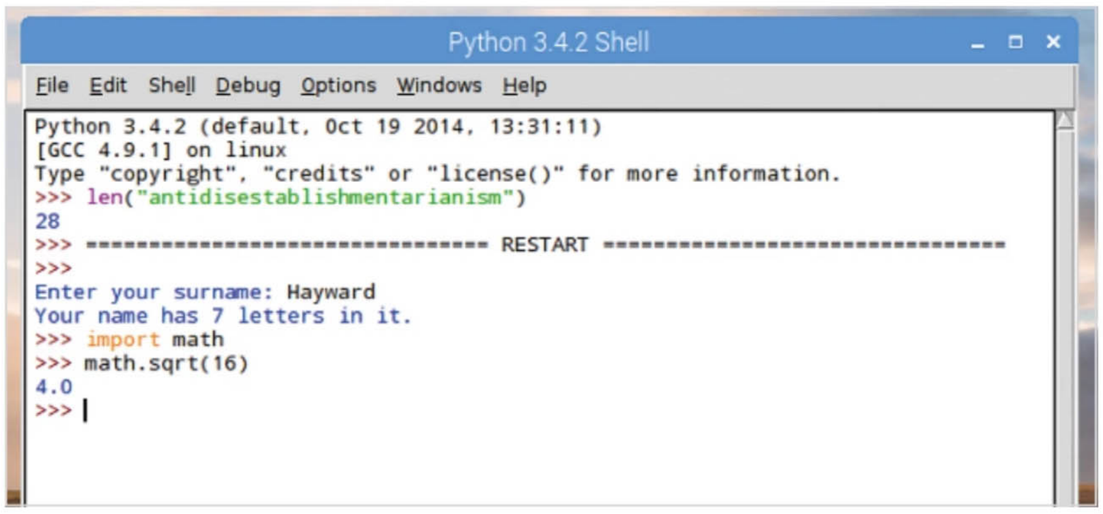
Once entered, you have access to all the Math module functions.

**STEP 6** To use a function from a module enter the name of the module followed by a full stop, then the name of the function. For instance, using the Math module, since you've just imported it into Python, you can utilise the square root function. To do so, enter:

```
math.sqrt(16)
```

You can see that the code is presented as module.function(data).



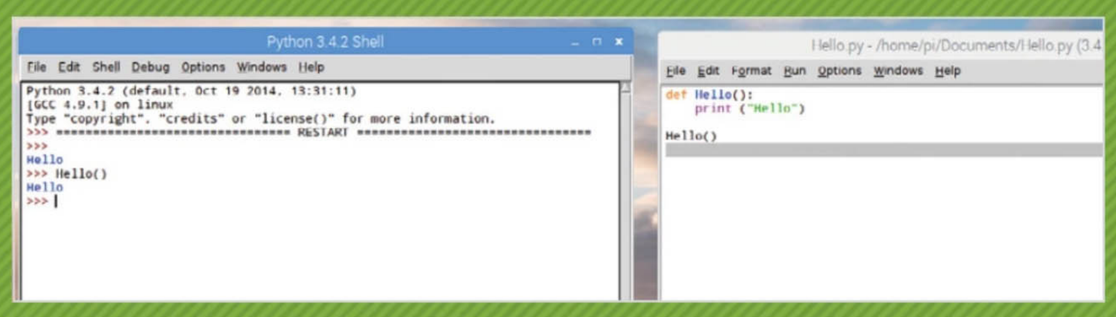

## FORGING FUNCTIONS

**There are many different functions you can import created by other Python programmers and you will undoubtedly come across some excellent examples in the future; you can also create your own with the def command.**

**STEP 1** Choose File > New File to enter the editor, let's create a function called Hello, that greets a user. Enter:

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

Hello()
```

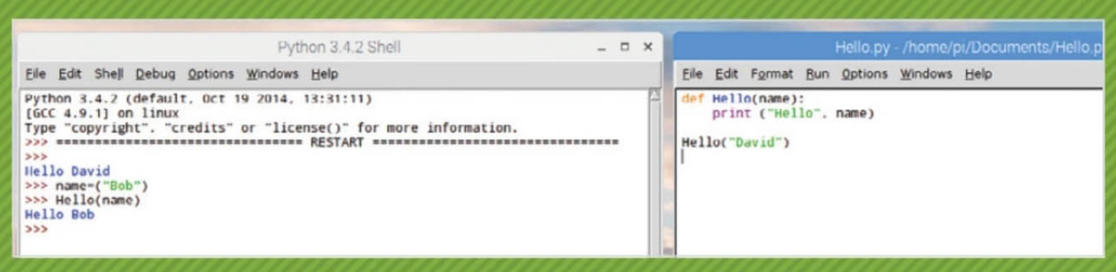
Press F5 to save and run the script. You can see Hello in the Shell, type in Hello() and it returns the new function.




**STEP 2** Let's now expand the function to accept a variable, the user's name for example. Edit your script to read:

```
def Hello(name):
    print ("Hello", name)

Hello("David")
```

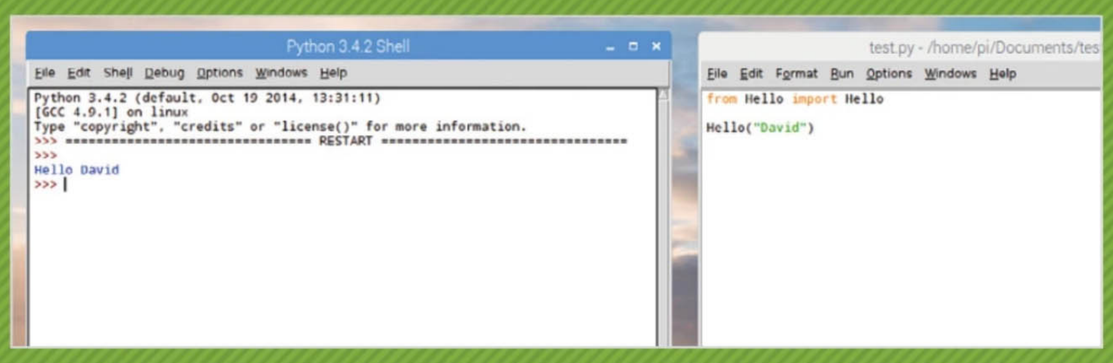
This will now accept the variable name, otherwise it prints Hello David. In the Shell, enter: name=("Bob"), then: Hello(name). Your function can now pass variables through it.




**STEP 3** To modify it further, delete the Hello("David") line, the last line in the script and press Ctrl+S to save the new script. Close the Editor and create a new file (File > New File). Enter the following:

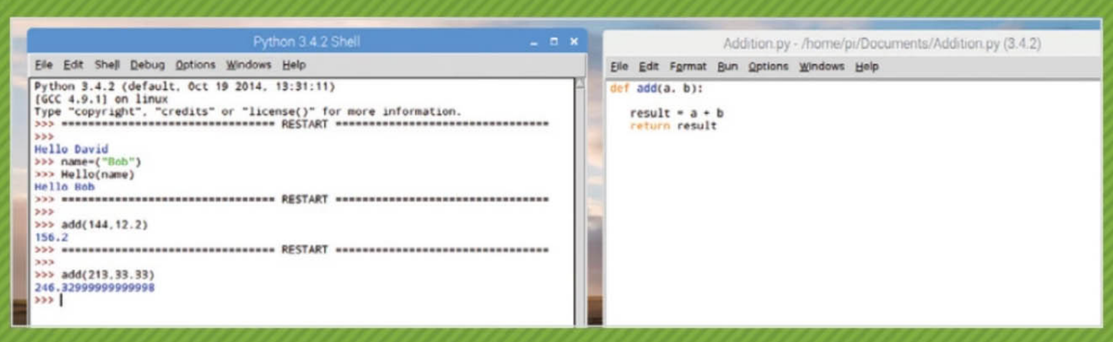
```
from Hello import Hello

Hello("David")
```

Press F5 to save and execute the code.




**STEP 4** What you've just done is import the Hello function from the saved Hello.py program and then used it to say hello to David. This is how modules and functions work: you import the module then use the function. Try this one, and modify it for extra credit:

```
def add(a, b):
    result = a + b
    return result
```



# Conditions and Loops

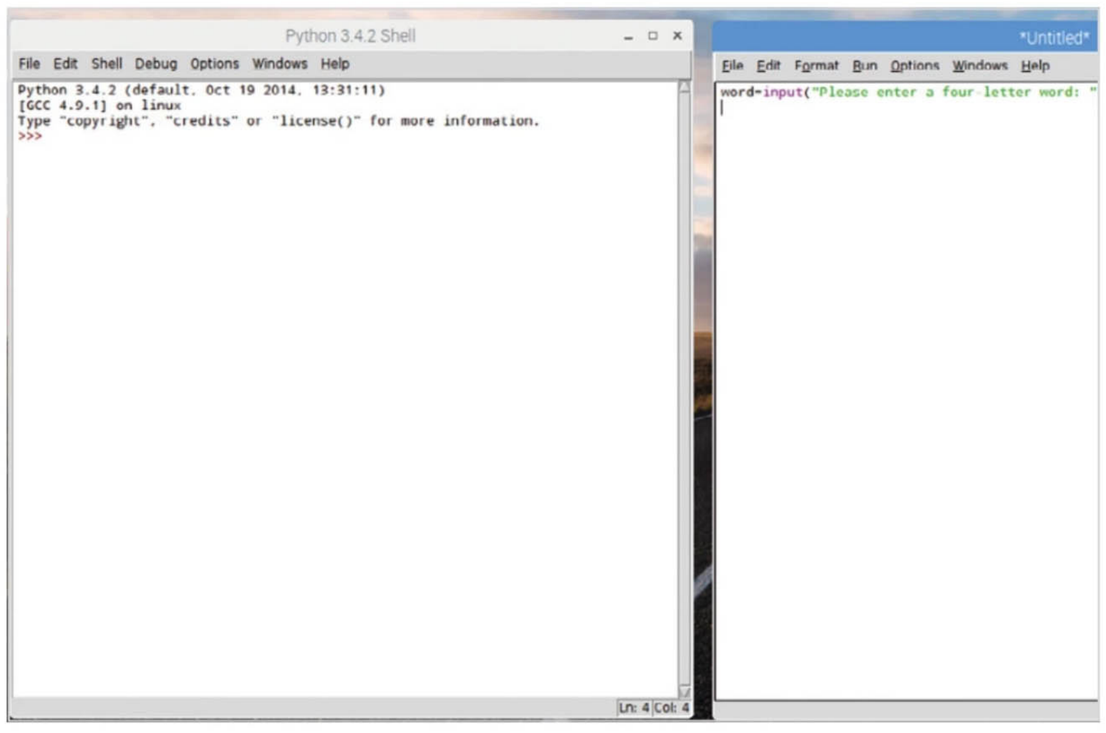
Conditions and loops are what makes a program interesting; they can be simple or rather complex. How you use them depends greatly on what the program is trying to achieve; they could be the number of lives left in a game or just displaying a countdown.

## TRUE CONDITIONS

**Keeping conditions simple to begin with makes learning to program a more enjoyable experience. Let's start then by checking if something is TRUE, then doing something else if it isn't.**

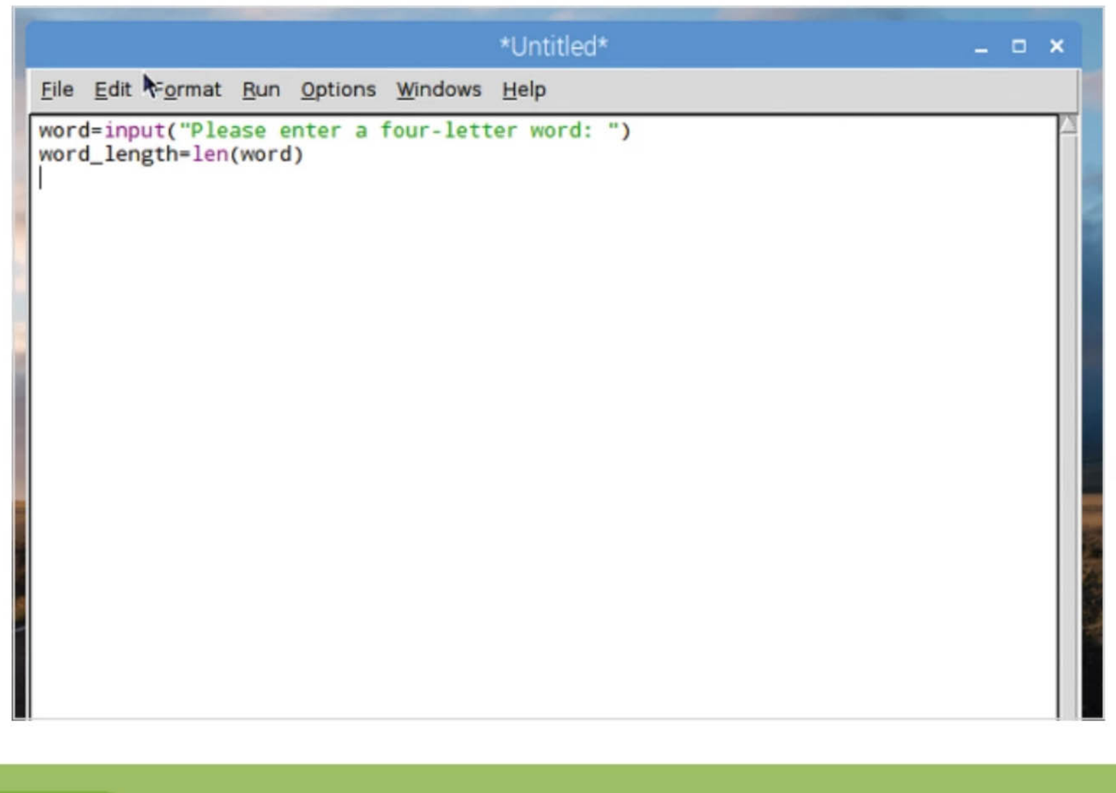
**STEP 1** Let's create a new Python program that will ask the user to input a word, then check it to see if it's a four-letter word or not. Start with File > New File, and begin with the input variable:

```
word=input("Please enter a four-letter word: ")
```




**STEP 2** Now we can create a new variable, then use the len function and pass the word variable through it to get the total number of letters the user has just entered:

```
word=input("Please enter a four-letter word: ")
word_length=len(word)
```




**STEP 3** Now you can use an if statement to check if the word_length variable is equal to four and print a friendly conformation if it applies to the rule:

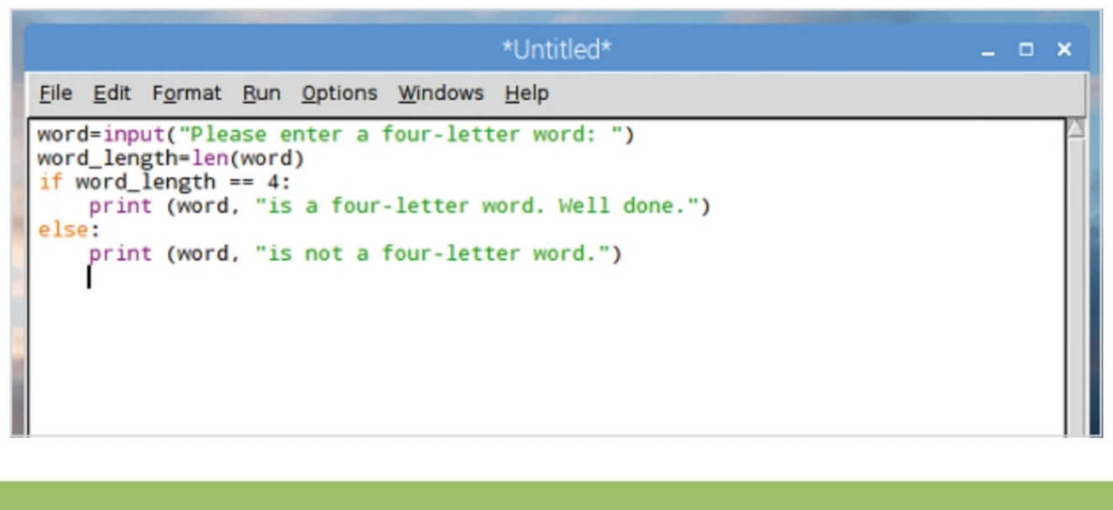
```
word=input("Please enter a four-letter word: ")
word_length=len(word)
if word_length == 4:
   print (word, "is a four-letter word. Well done.")
```

The double equal sign (==) means check if something is equal to something else.

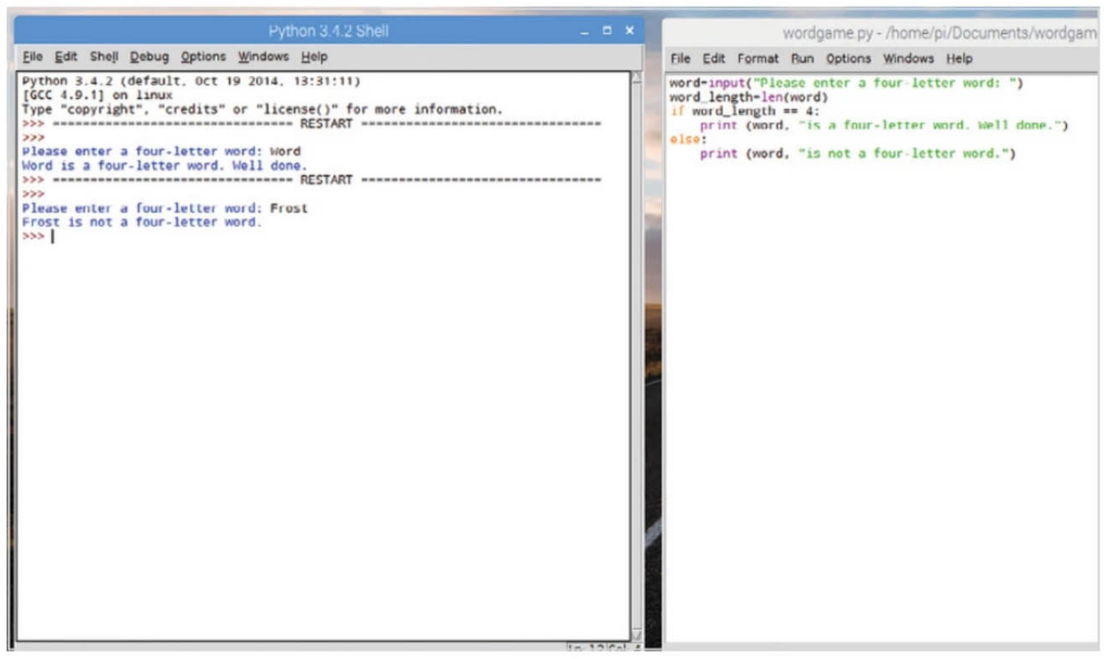



**STEP 4** The colon at the end of IF tells Python that if this statement is true do everything after the colon that's indented. Next, move the cursor back to the beginning of the Editor:

```
word=input("Please enter a four-letter word: ")
word_length=len(word)
if word_length == 4:
   print (word, "is a four-letter word. Well done.")
else:
   print (word, "is not a four-letter word.")
```

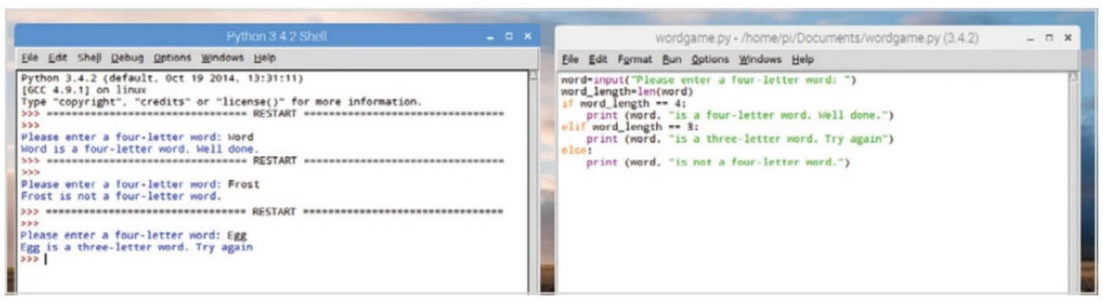


**STEP 5** Press F5 and save the code to execute it. Enter a four-letter word in the Shell to begin with, you should have the returned message that it's the word is four letters. Now press F5 again and rerun the program but this time enter a five-letter word. The Shell will display that it's not a four-letter word.




**STEP 6** Now expand the code to include another conditions. Eventually, it could become quite complex. We've added a condition for three-letter words:

```
word=input("Please enter a four-letter word: ")
word_length=len(word)
if word_length == 4:
   print (word, "is a four-letter word. Well
done.")
elif word_length == 3:
   print (word, "is a three-letter word. Try again.")
else:
   print (word, "is not a four-letter word.")
```

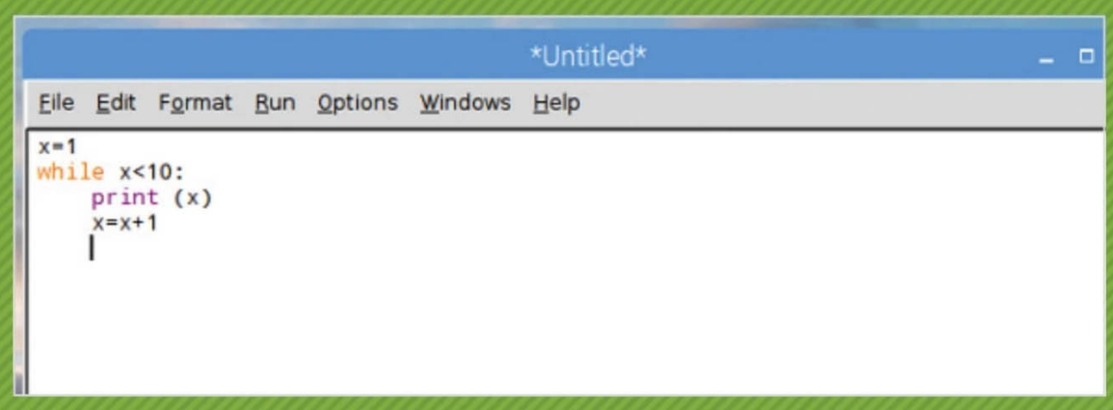



## LOOPS

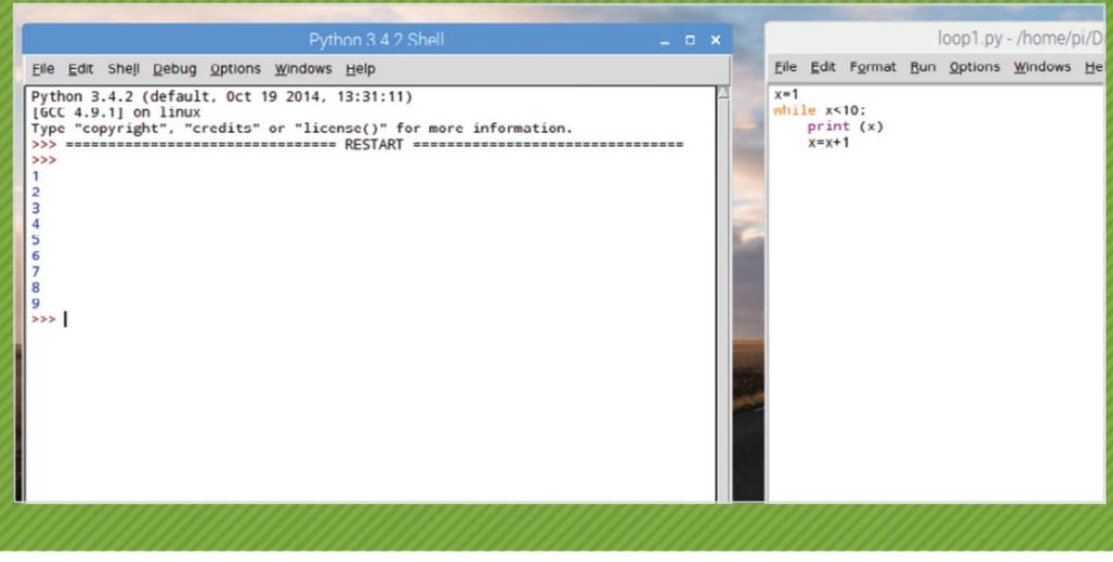
**A loop looks quite similar to a condition but they are somewhat different in their operation. A loop will run through the same block of code a number of times, usually with the support of a condition.**

**STEP 1** Let's start with a simple While statement. Like IF, this will check to see if something is TRUE, then run the indented code:

```
x = 1
while x < 10:
   print (x)
   x = x + 1
```

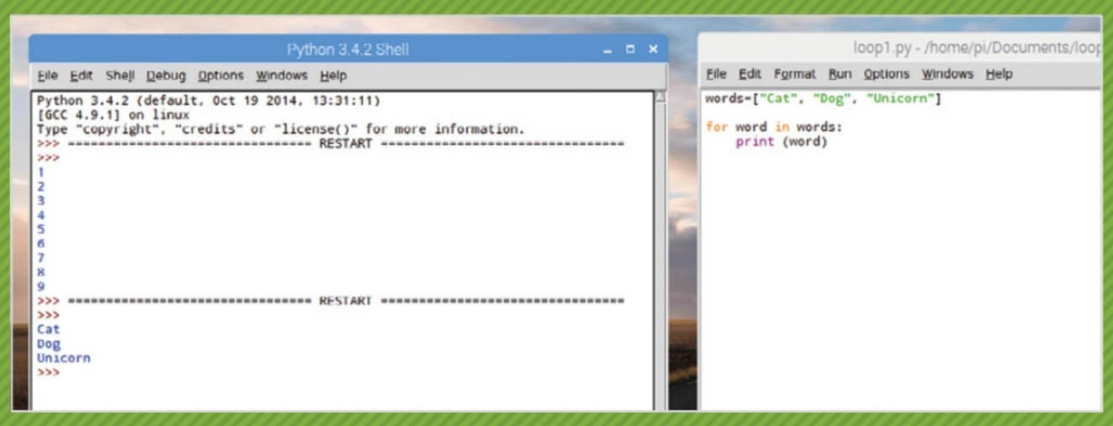



**STEP 2** The difference between if and while is when while gets to the end of the indented code, it goes back and checks the statement is still true. In our example x is less than 10. With each loop it prints the current value of x, then adds one to that value. When x does eventually equal 10 it stops.




**STEP 3** The For loop is another example. For is used to loop over a range of data, usually a list stored as variables inside square brackets. For example:

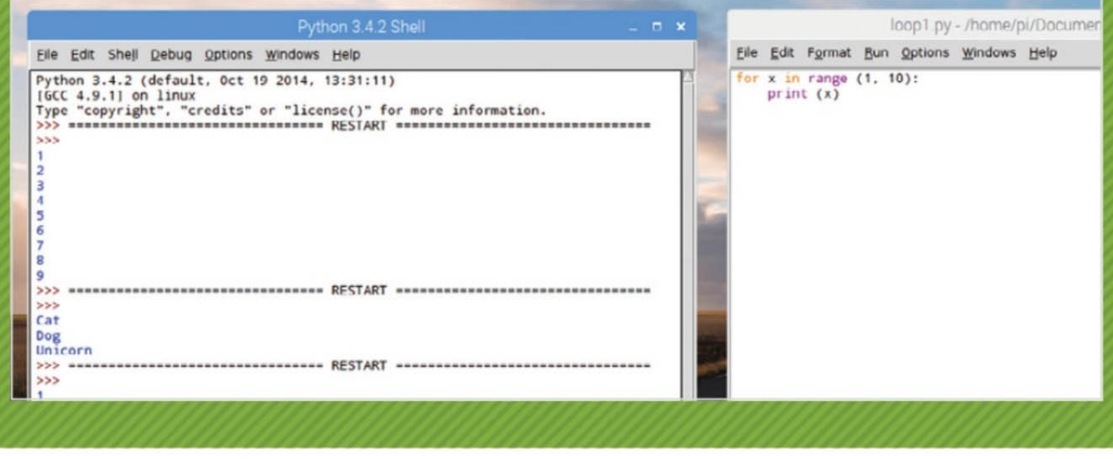
```
words=["Cat", "Dog", "Unicorn"]

for word in words:
   print (word)
```




**STEP 4** The For loop can also be used in the countdown example by using the range function:

```
for x in range (1, 10):
  print (x)
```

The x=x+1 part isn't needed here because the range function creates a list between the first and last numbers used.



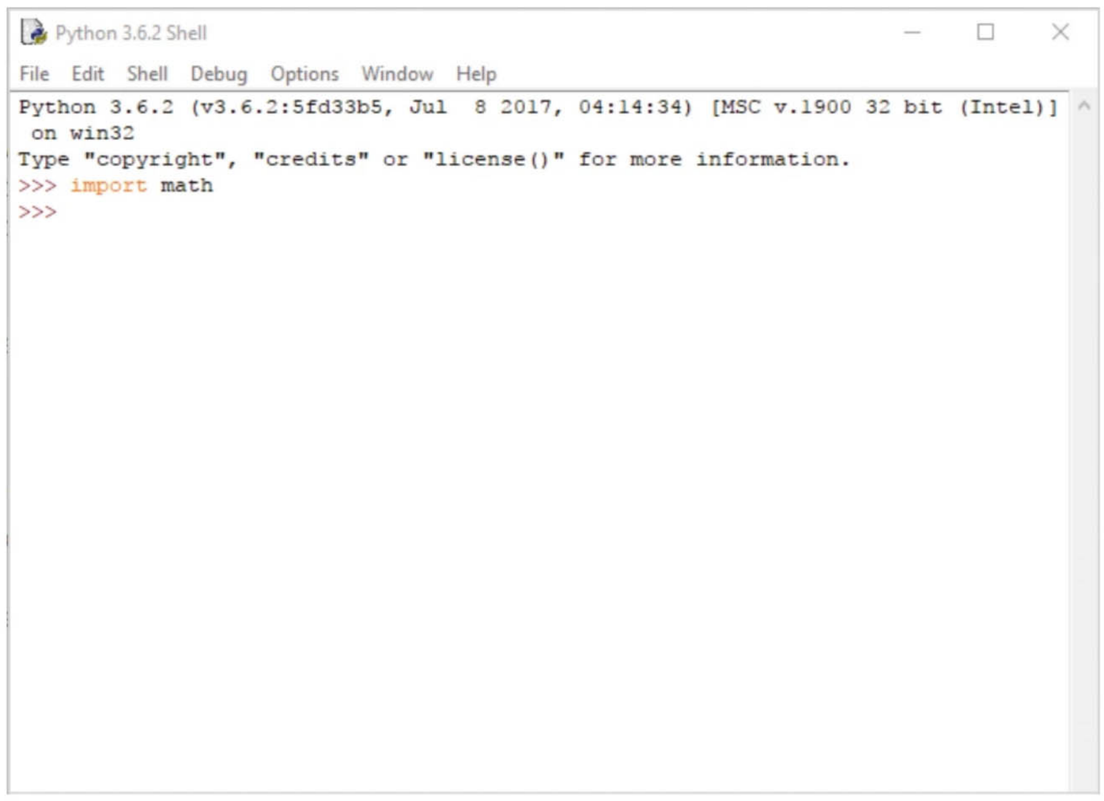
# Python Modules

We've mentioned modules previously, (the Math module) but as modules are such a large part of getting the most from Python, it's worth dedicating a little more time to them. In this instance we're using the Windows version of Python 3.

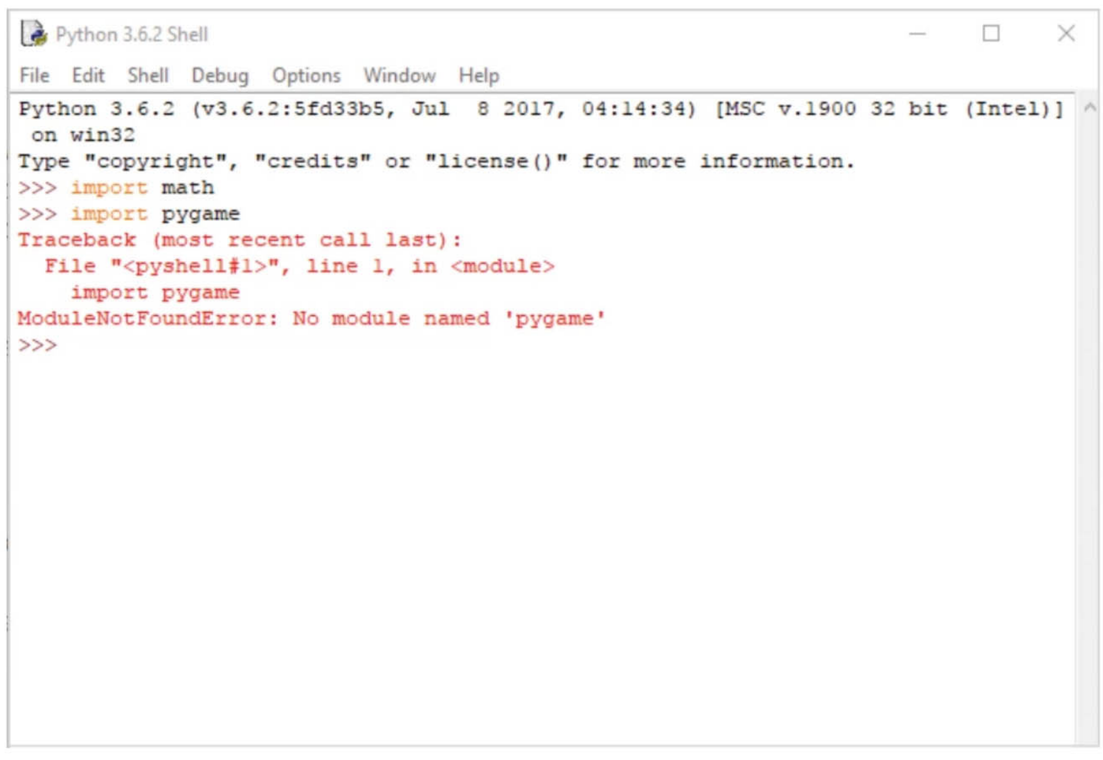
## MASTERING MODULES

**Think of modules as an extension that's imported into your Python code to enhance and extend its capabilities. There are countless modules available and as we've seen, you can even make your own.**

**STEP 1** Although good, the built-in functions within Python are limited. The use of modules, however, allows us to make more sophisticated programs. As you are aware, modules are Python scripts that are imported, such as import math.

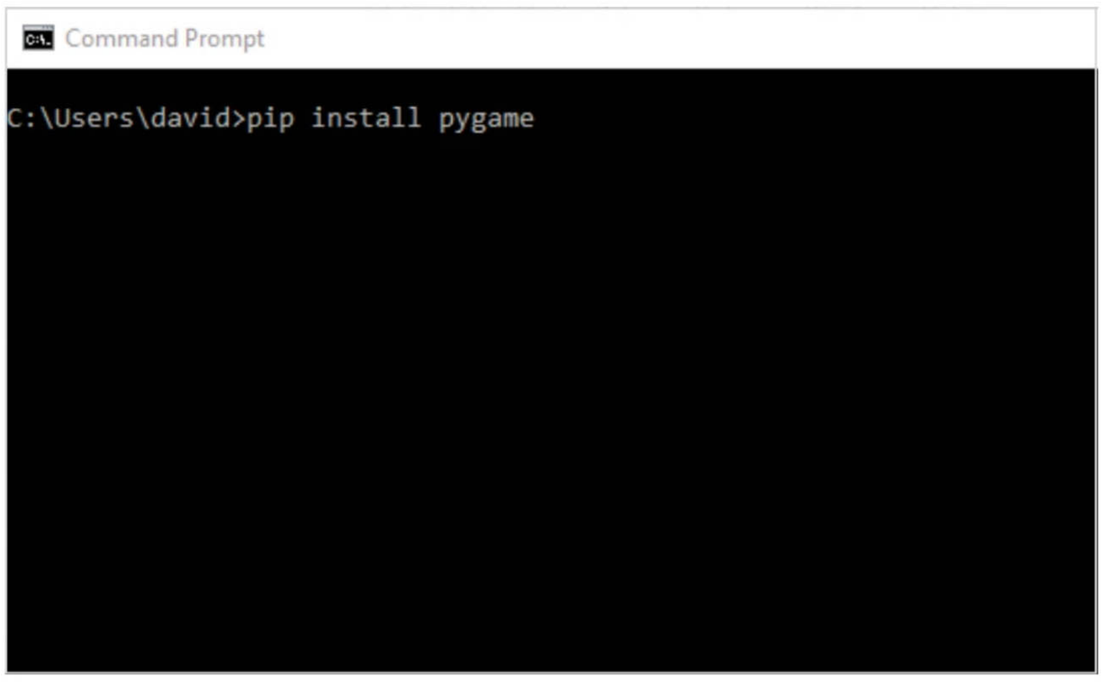



**STEP 2** Some modules, especially on the Raspberry Pi, are included by default, the Math module being a prime example. Sadly, other modules aren't always available. A good example on non-Pi platforms is the Pygame module, which contains many functions to help create games. Try: **import pygame**.

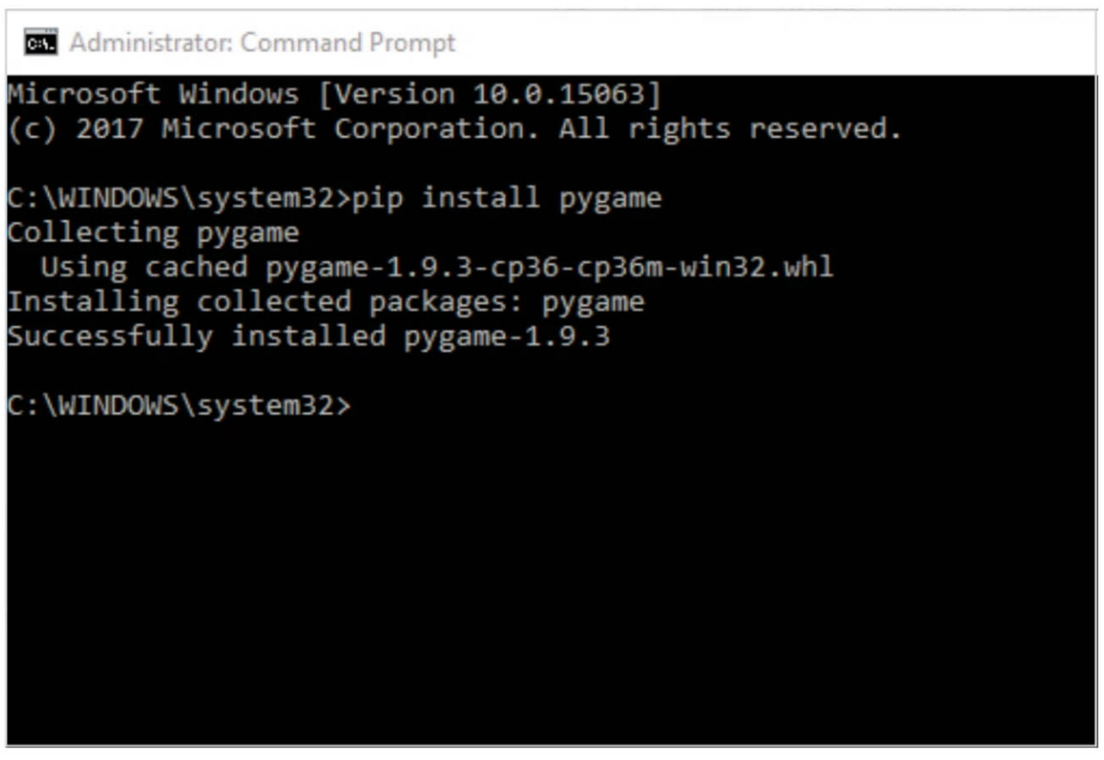



**STEP 3** The result is an error in the IDLE Shell, as the Pygame module isn't recognised or installed in Python. To install a module we can use PIP (Pip Installs Packages). Close down the IDLE Shell and drop into a command prompt or Terminal session. At an elevated admin command prompt, enter:

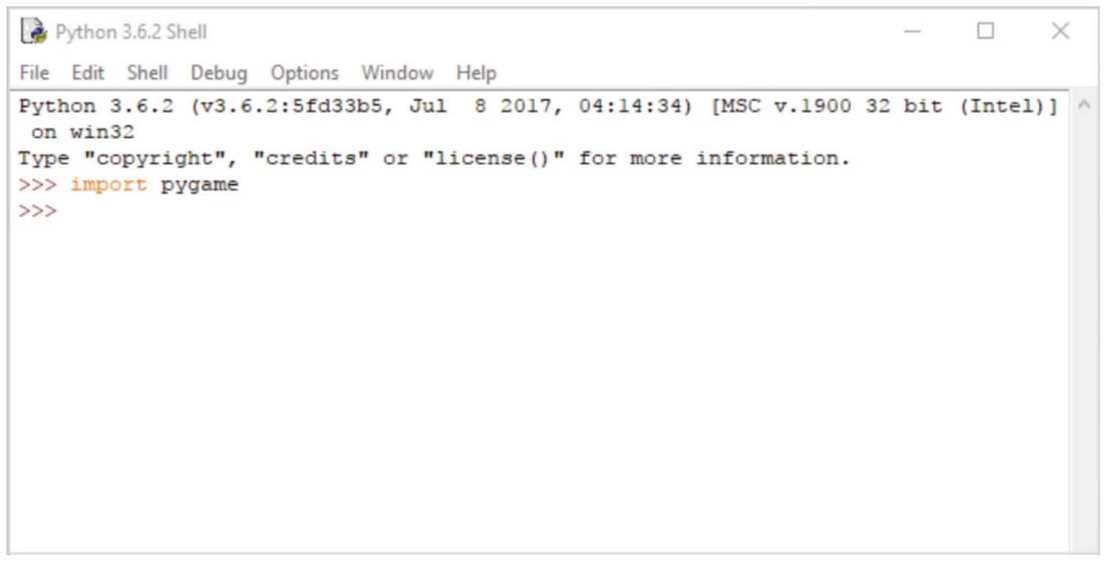
```
pip install pygame
```




**STEP 4** The PIP installation requires an elevated status due it installing components at different locations. Windows users can search for CMD via the Start button and right-click the result then click Run as Administrator. Linux and Mac users can use the Sudo command, with sudo pip install package.



**STEP 5** Close the command prompt or Terminal and relaunch the IDLE Shell. When you now enter: `import pygame`, the module will be imported into the code without any problems. You'll find that most code downloaded or copied from the Internet will contain a module, mainstream of unique, these are usually the source of errors in execution due to them being missing.




**STEP 6** The modules contain the extra code needed to achieve a certain result within your own code, as we've previously experimented with. For example:

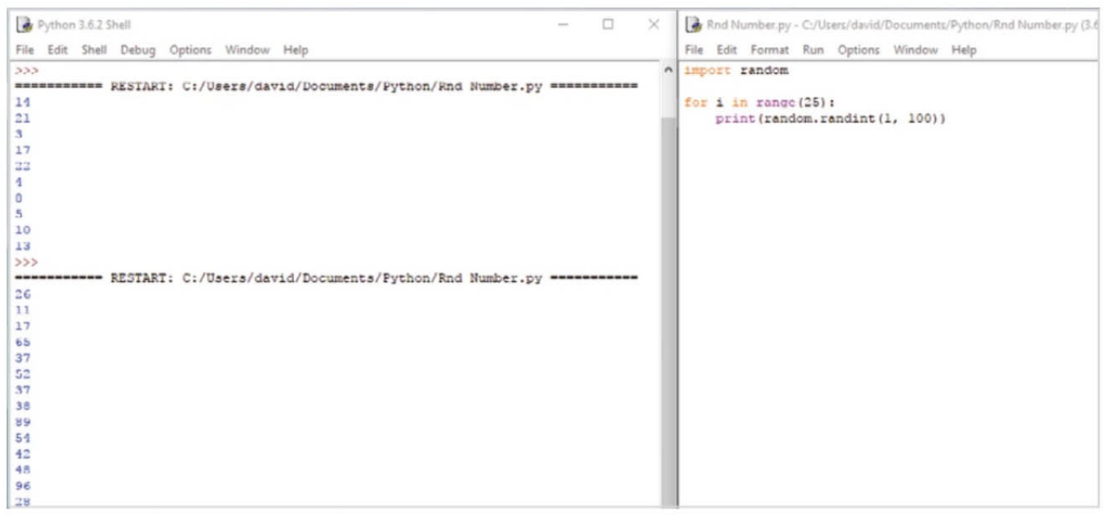
```
import random
```

Brings in the code from the Random Number Generator module. You can then use this module to create something like:

```
for i in range(10):
  print(random.randint(1, 25))
```

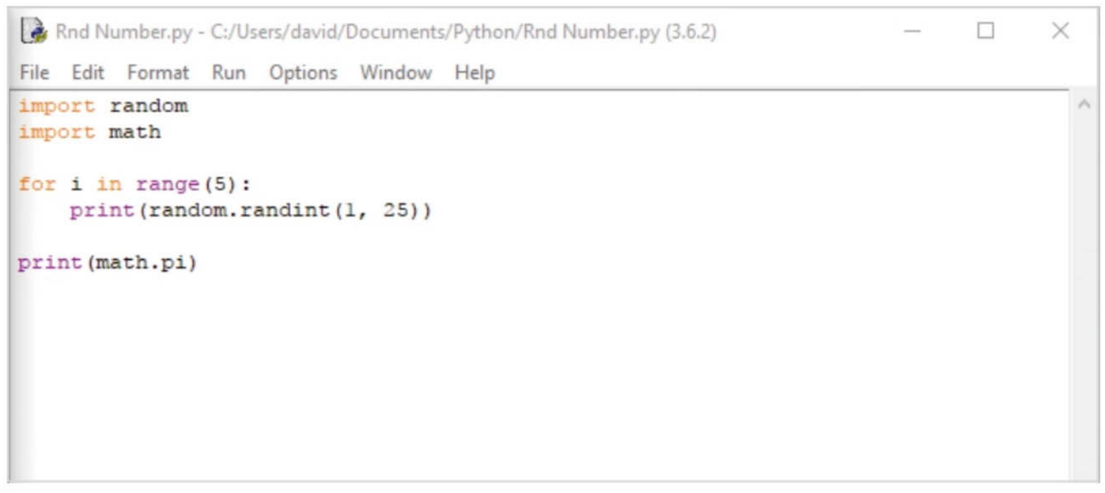
**STEP 7** This code, when saved and executed, will display ten random numbers from 1 to 25. You can play around with the code to display more or less, and from a great or lesser range. For example:

```
import random

for i in range(25):
  print(random.randint(1, 100))
```



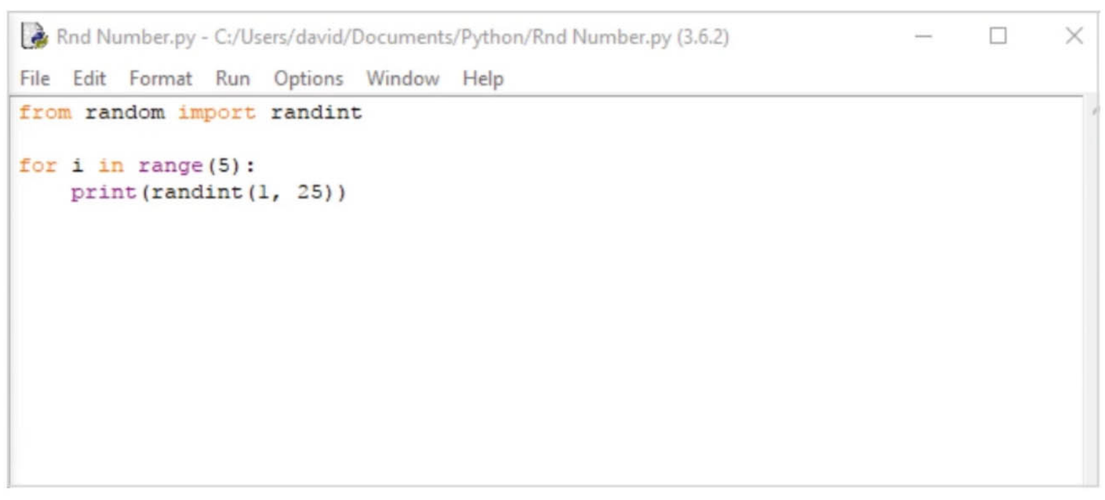

**STEP 8** Multiple modules can be imported within your code. To extend our example, use:

```
import random
import math

for I in range(5):
  print(random.randint(1, 25))

print(math.pi)
```




**STEP 9** The result is a string of random numbers followed by the value of Pi as pulled from the Math module using the print(math.pi) function. You can also pull in certain functions from a module by using the from and import commands, such as:

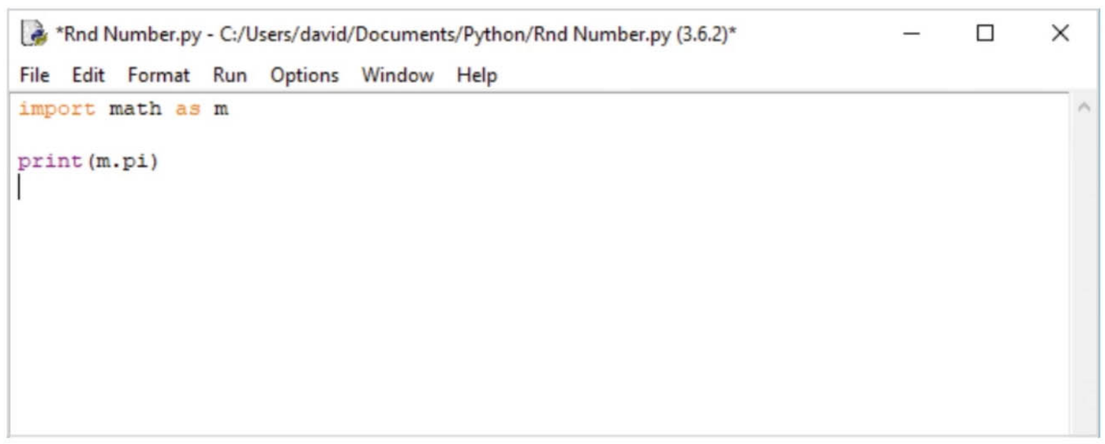
```
from random import randint

for i in range(5):
  print(randint(1, 25))
```




**STEP 10** This helps create a more streamlined approach to programming. You can also use import module*, which will import everything defined within the named module. However, it's often regarded as a waste of resources but it works nonetheless. Finally, modules can be imported as aliases:

```
import math as m

print(m.pi)
```

Of course, adding comments helps to tell others what's going on.



"It's OK to figure out murder mysteries, but you shouldn't need to figure out code. You should be able to read it."

– *Steve McConnell (Software Engineer and Author)*

# Working with Data

Data is everything. With it, you can display, control, add, remove, create and manipulate Python to your every demand. Over these coming pages we look at how you can create lists, tuples, dictionaries and multi-dimensional lists then how you can use them to forge exciting and useful programs. It's little wonder that data is now valued more than gold or oil.

You will also learn how you can use the date and time functions, write to files in your system and even create graphical user interfaces that will take your coding skills to new levels and into new project ideas.

# Lists

Lists are one of the most common types of data structures you will comes across in Python. A list is simply a collection of items, or data if you prefer, that can be accessed as a whole, or individually if wanted.

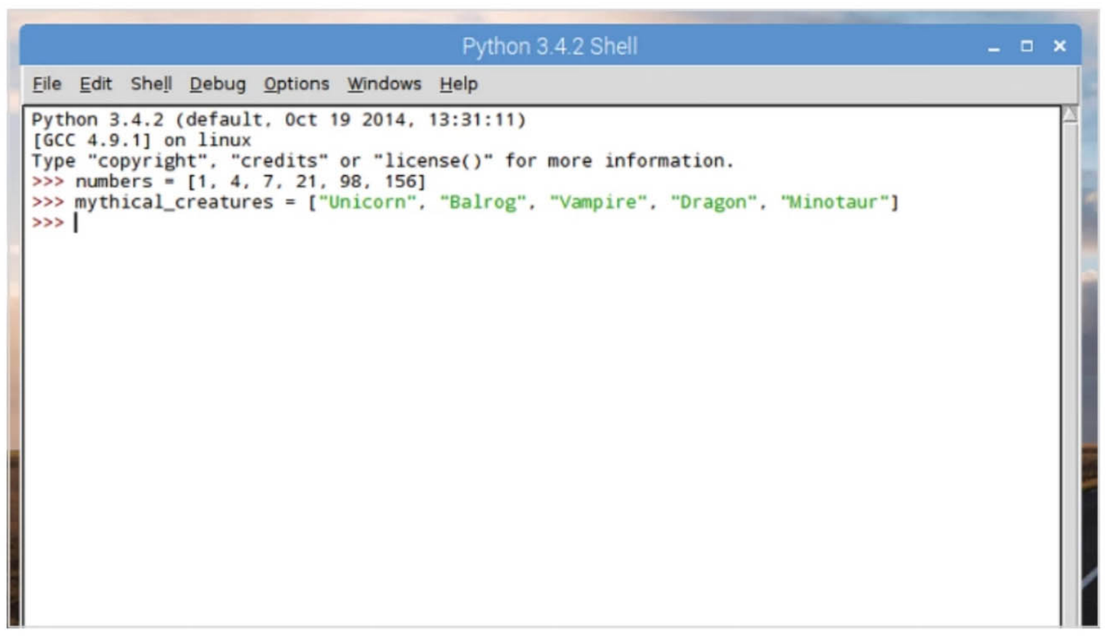
## WORKING WITH LISTS

**Lists are extremely handy in Python. A list can be strings, integers and also variables. You can even include functions in lists, and lists within lists.**

### STEP 1

A list is a sequence of data values called items. You create the name of your list followed by an equals sign, then square brackets and the items separated by commas; note that strings use quotes:

```
numbers = [1, 4, 7, 21, 98, 156]
mythical_creatures – [“Unicorn”, “Balrog”,
“Vampire”, “Dragon”, “Minotaur”]
```




### STEP 2

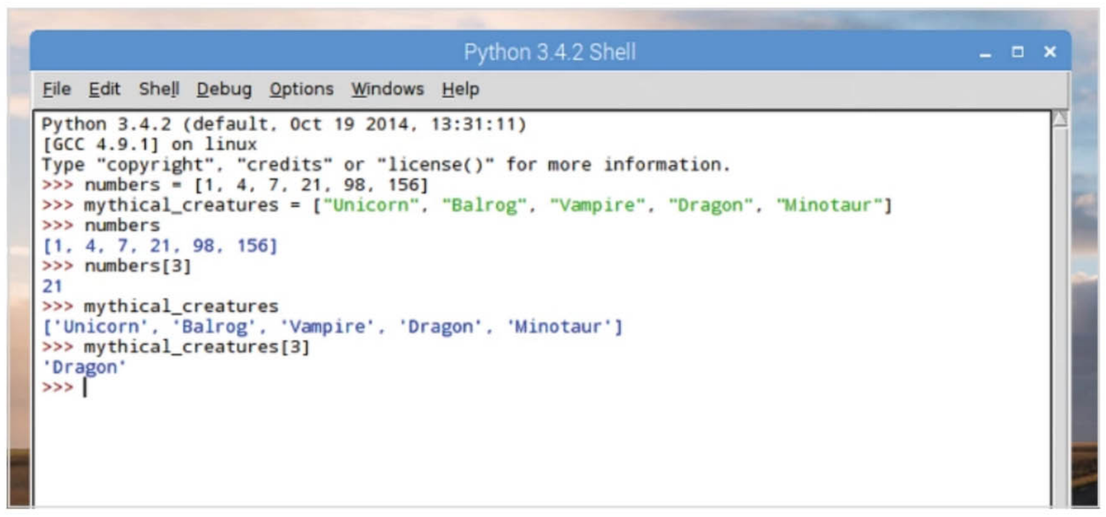
Once you’ve defined your list you can call each by referencing its name, followed by a number. Lists start the first item entry as 0, followed by 1, 2, 3 and so on.
For example:

```
numbers
```

To call up the entire contents of the list.

```
numbers[3]
```

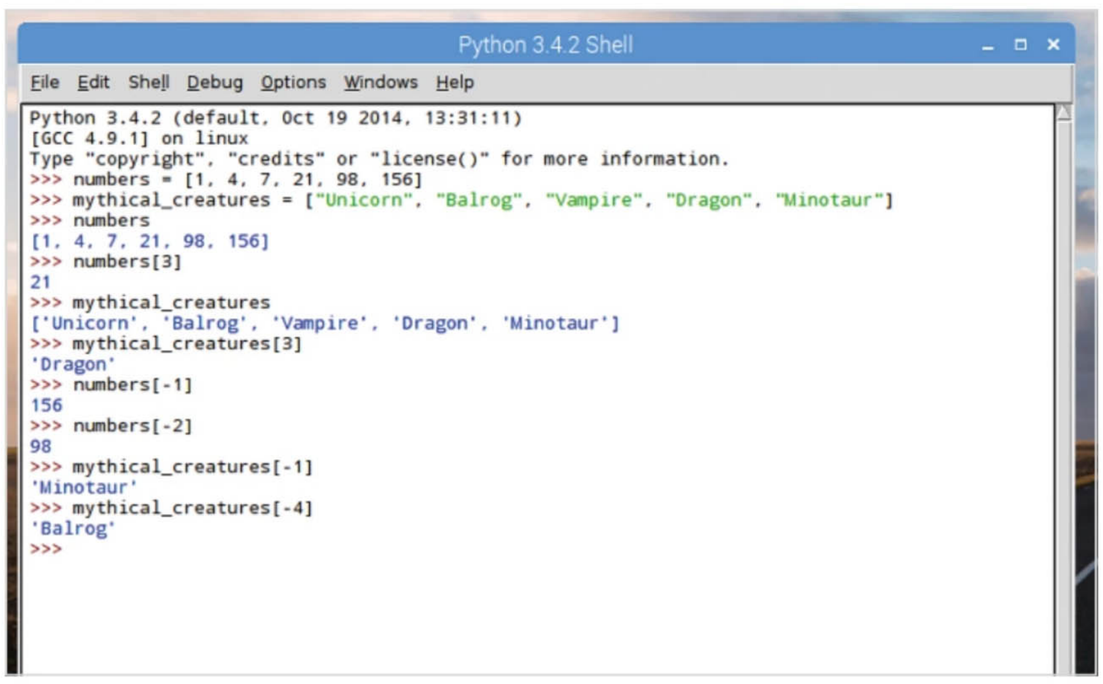
To call the third from zero item in the list (21 in this case).




### STEP 3

You can also access, or index, the last item in a list by using the minus sign before the item number [-1], or the second to last item with [-2] and so on. Trying to reference an item that isn’t in the list, such as [10] will return an error:

```
numbers[-1]
mythical_creatures[-4]
```




### STEP 4

Slicing is similar to indexing but you can retrieve multiple items in a list by separating item numbers with a colon. For example:

```
numbers[1:3]
```

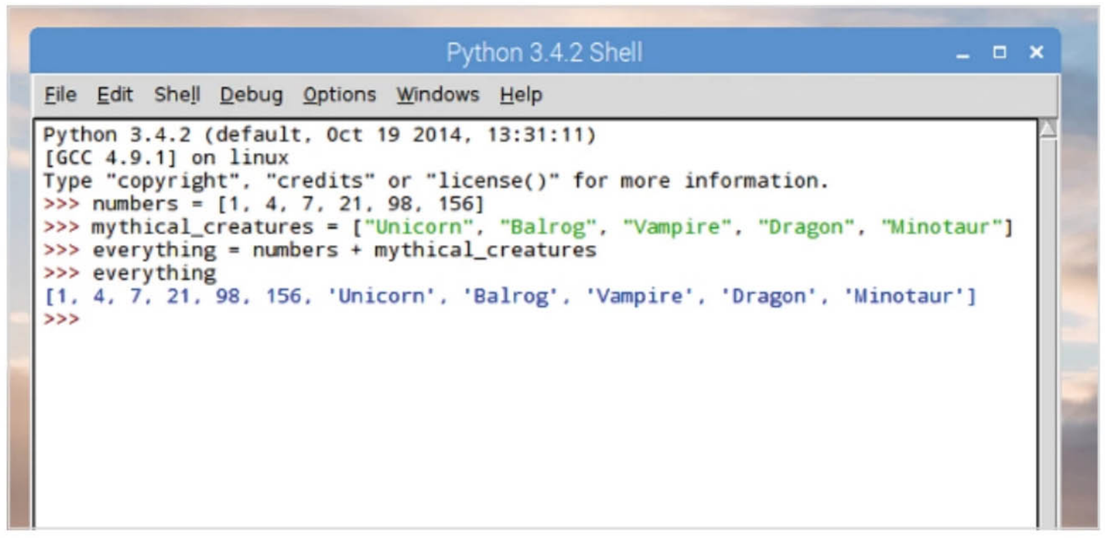
Will output the 4 and 7, being item numbers 1 and 2. Note that the returned values don’t include the second index position (as you would numbers[1:3] to return 4, 7 and 21).

**STEP 5** You can update items within an existing list, remove items and even join lists together. For example, to join two lists you can use:

```
everything = numbers + mythical_creatures
```

Then view the combined list with:

```
everything
```




**STEP 6** Items can be added to a list by entering:

```
numbers=numbers+[201]
```

Or for strings:

```
mythical_creatres=mythical_creatures+[“Griffin”]
```

Or by using the append function:

```
mythical_creatures.append(“Nessie”)
numbers.append(278)
```

**STEP 7** Removal of items can be done in two ways. The first is by the item number:

```
del numbers[7]
```

Alternatively, by item name:

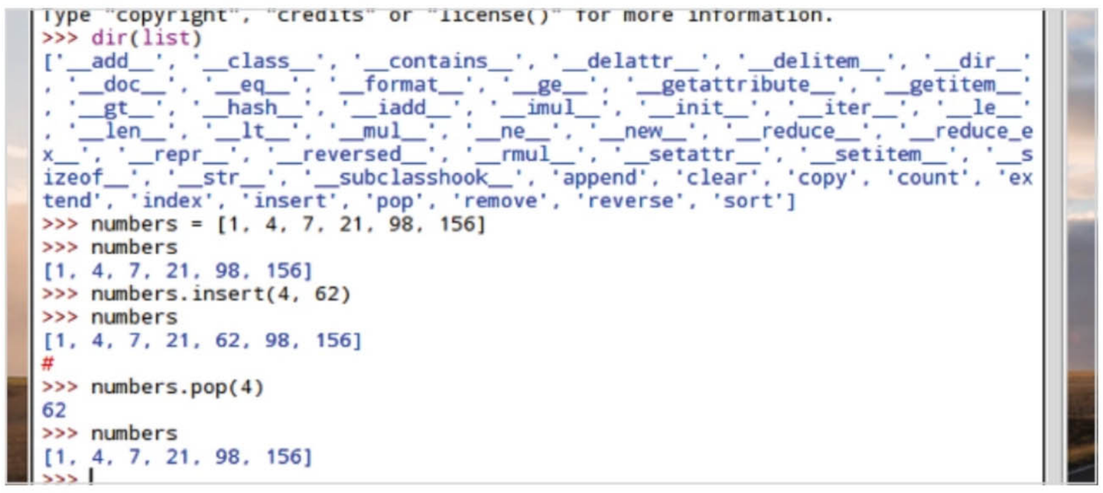
```
mythical_creatures.remove(“Nessie”)
```

**STEP 8** You can view what can be done with lists by entering dir(list) into the Shell. The output is the available functions, for example, insert and pop are used to add and remove items at certain positions. To insert the number 62 at item index 4:

```
numbers.insert(4, 62)
```

To remove it:

```
numbers.pop(4)
```

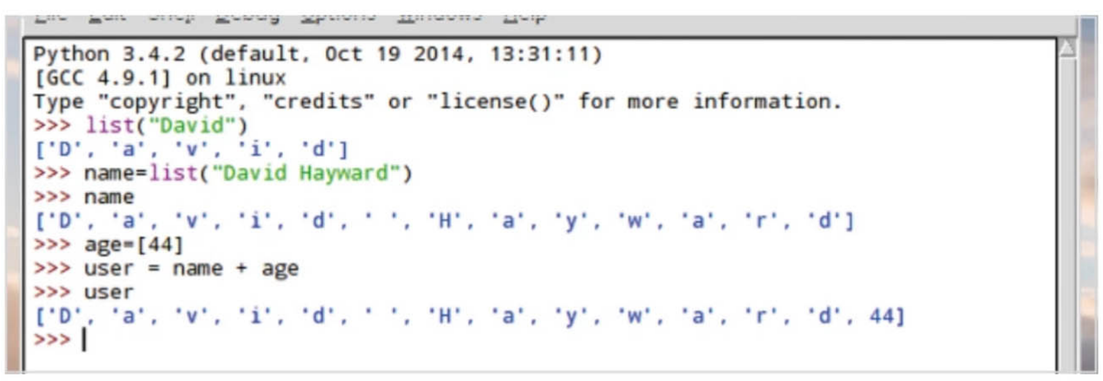



**STEP 9** You also use the list function to break a string down into its components. For example:

```
list(“David”)
```

Breaks the name David into ‘D’, ‘a’, ‘v’, ‘i’, ‘d’. This can then be passed to a new list:

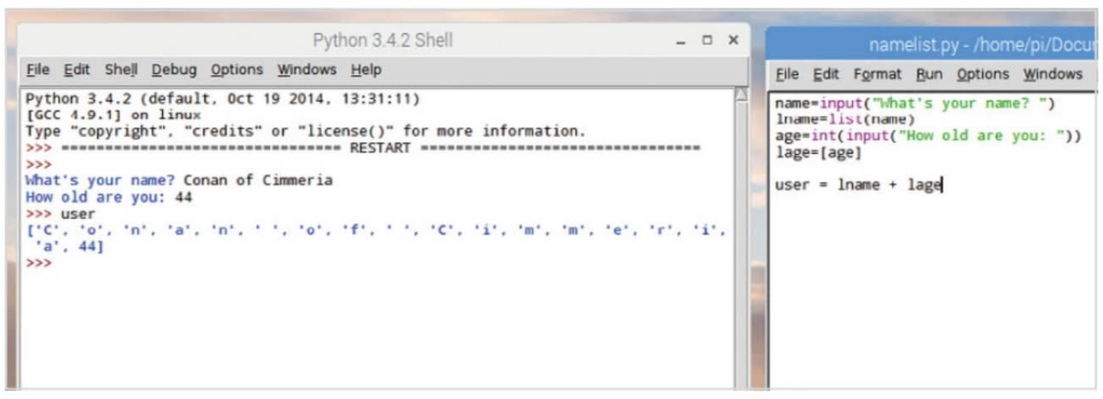
```
name=list(“David Hayward”)
name
age=[44]
user = name + age
user
```




**STEP 10** Based on that, you can create a program to store someone’s name and age as a list:

```
name=input(“What’s your name? “)
lname=list(name)
age=int(input(“How old are you: “))
lage=[age]

user = lname + lage
```

The combined name and age list is called user, which can be called by entering user into the Shell. Experiment and see what you can do.



# Tuples

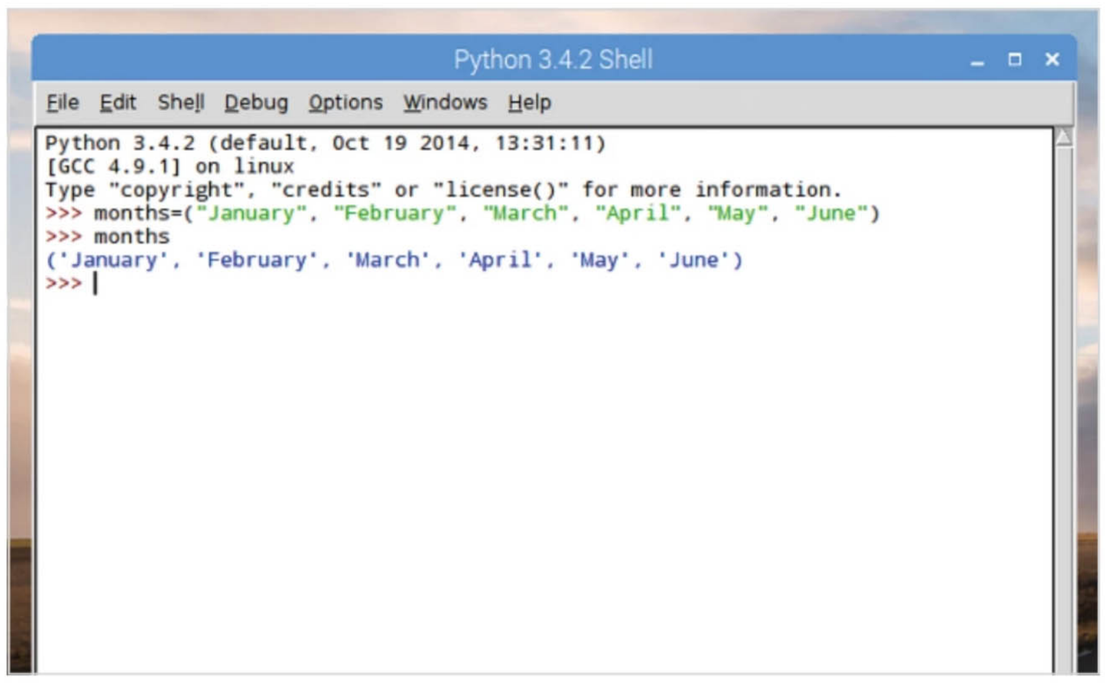
Tuples are very much identical to lists. However, where lists can be updated, deleted or changed in some way, a tuple remains a constant. This is called immutable and they're perfect for storing fixed data items.

## THE IMMUTABLE TUPLE

**Reasons for having tuples vary depending on what the program is intended to do. Normally, a tuple is reserved for something special but they're also used for example, in an adventure game, where non-playing character names are stored.**

**STEP 1** A tuple is created the same way as a list but in this instance you use curved brackets instead of square brackets. For example:

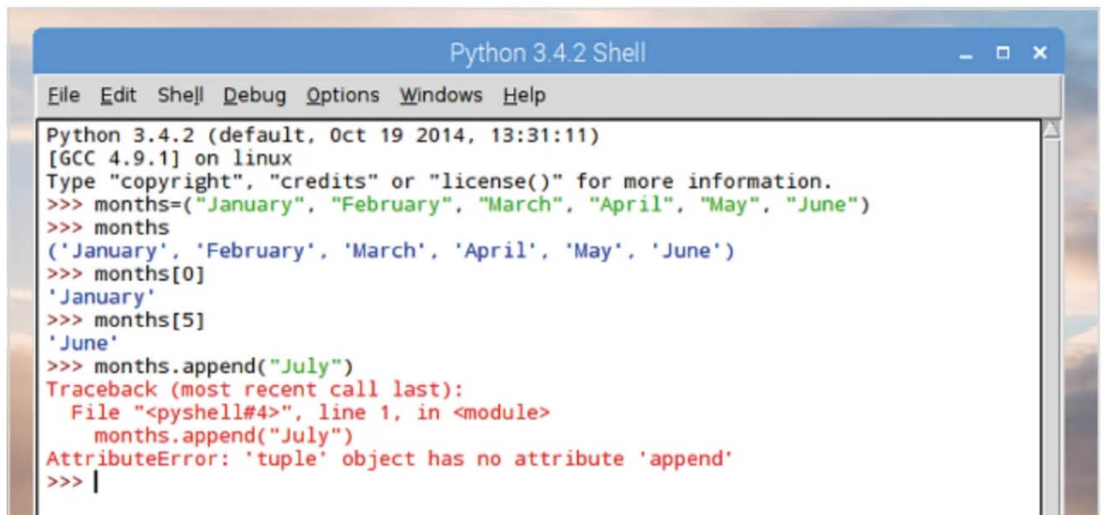
```
months=("January", "February", "March", "April",
"May", "June")
months
```




**STEP 2** Just as with lists, the items within a named tuple can be indexed according to their position in the data range, i.e.:

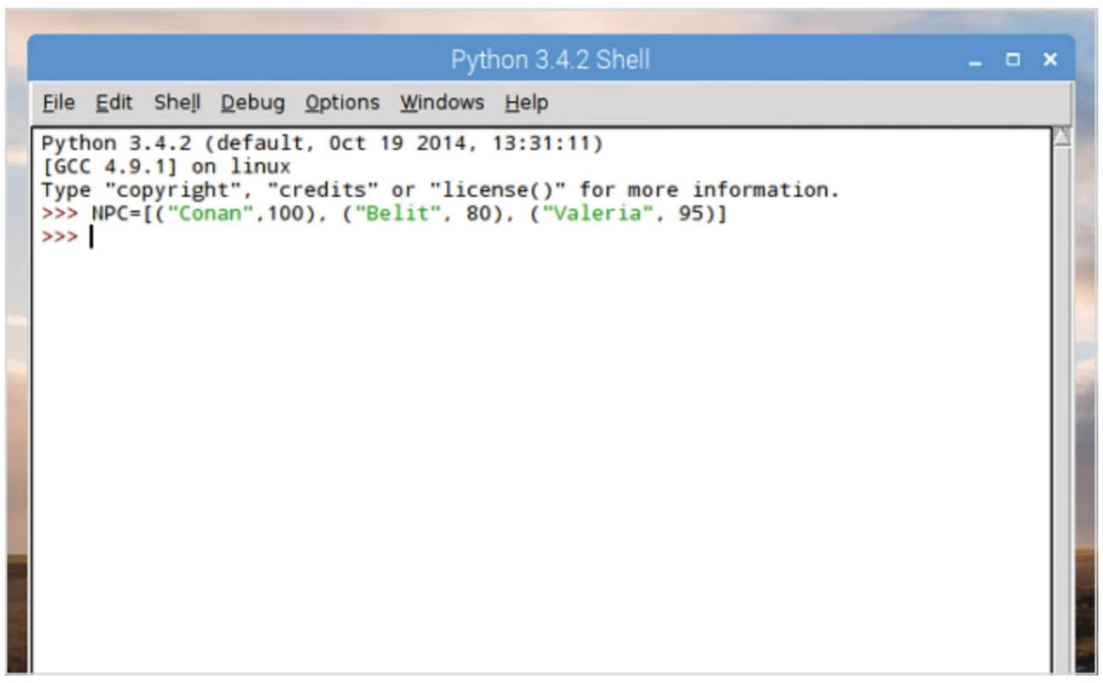
```
months[0]
months[5]
```

However, any attempt at deleting or adding to the tuple will result in an error in the Shell.




**STEP 3** You can create grouped tuples into lists that contain multiple sets of data. For instance, here is a tuple called NPC (Non-Playable Characters) containing the character name and their combat rating for an adventure game:

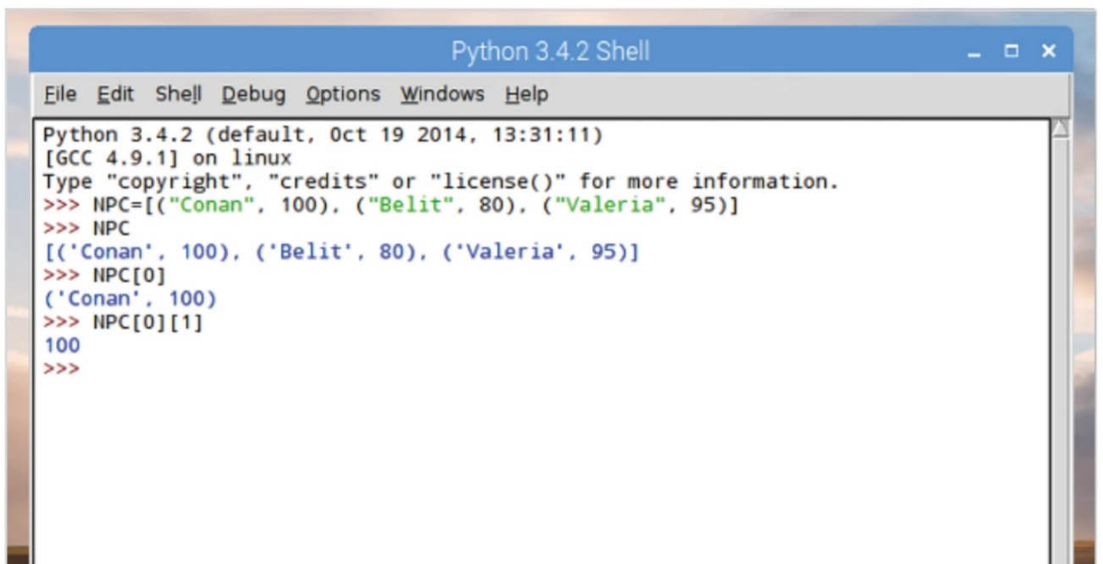
```
NPC=[("Conan", 100), ("Belit", 80), ("Valeria",
95)]
```




**STEP 4** Each of these data items can be accessed as a whole by entering NPC into the Shell; or they can be indexed according to their position NPC[0]. You can also index the individual tuples within the NPC list:

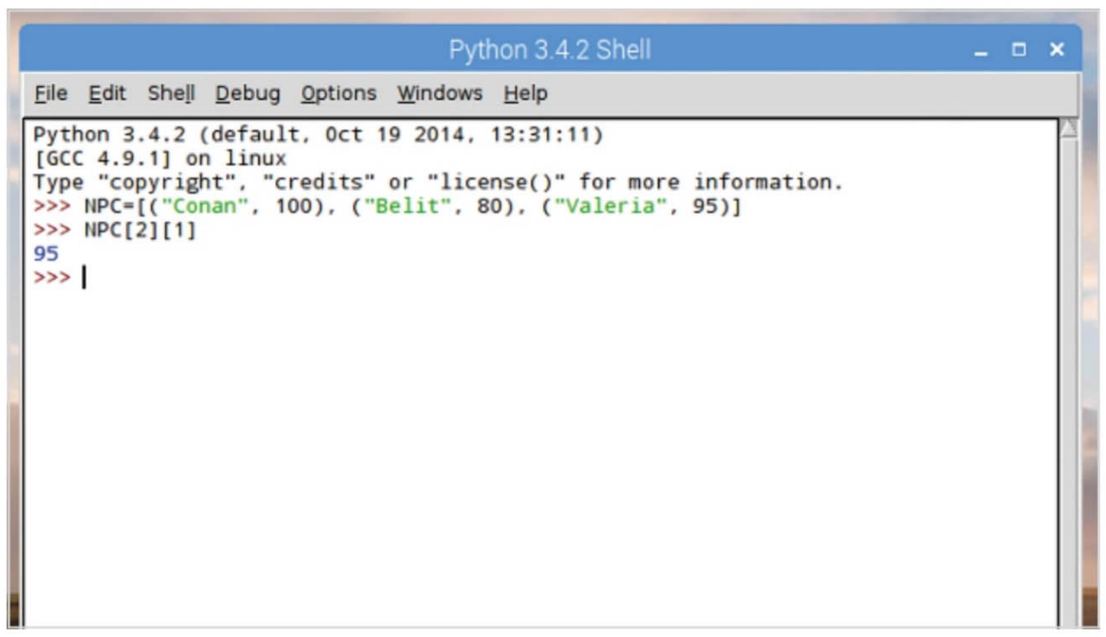
```
NPC[0][1]
```

Will display 100.



**STEP 5** It's worth noting that when referencing multiple tuples within a list, the indexing is slightly different from the norm. You would expect the 95 combat rating of the character Valeria to be NPC[4][5], but it's not. It's actually:

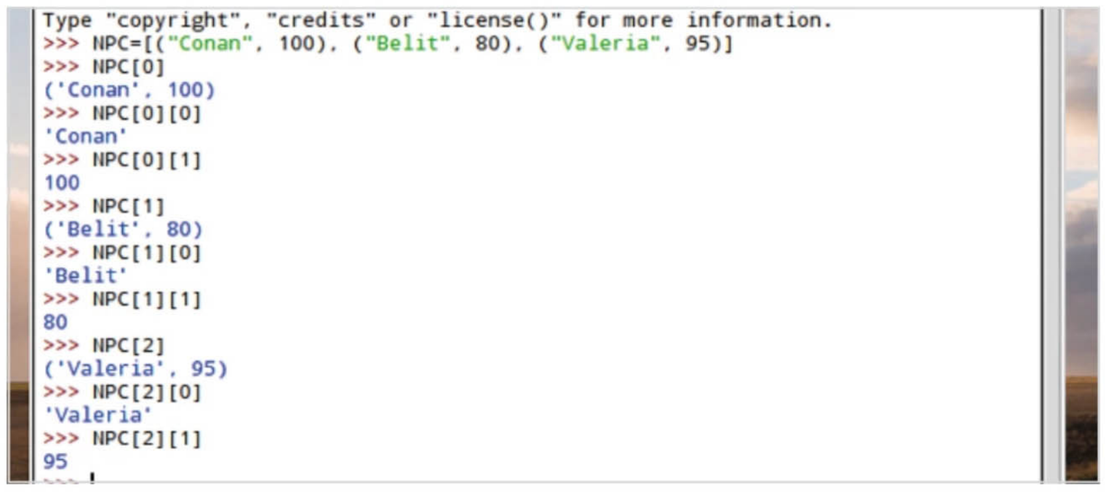
```
NPC[2][1]
```




**STEP 6** This means of course that the indexing follows thus:

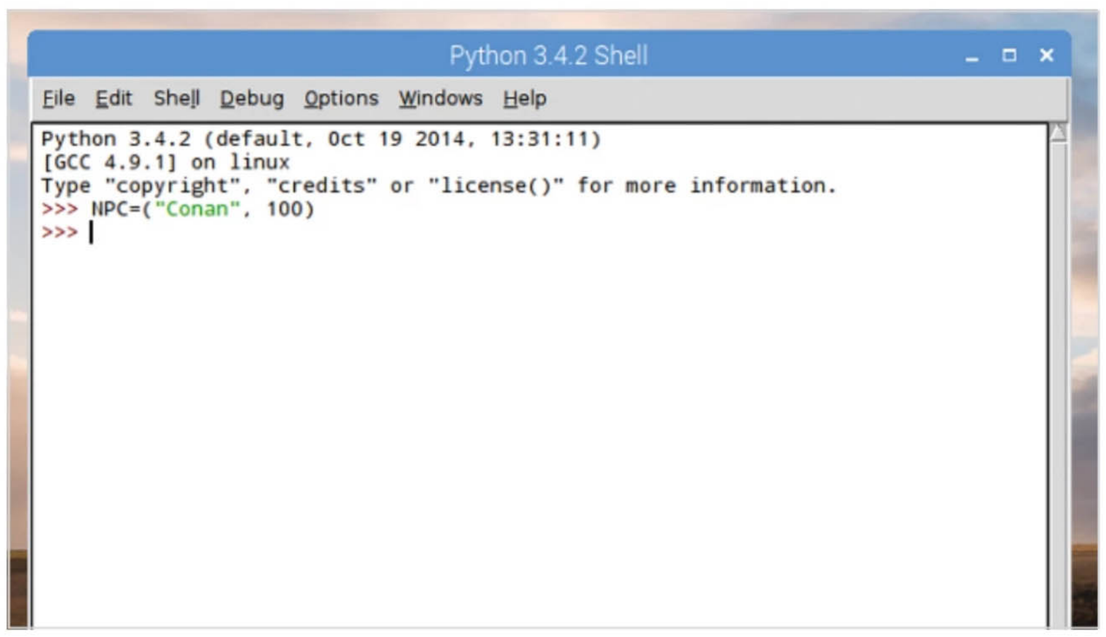
```
0                1, 1
0, 0             2
0, 1             2, 0
1                2,1
1, 0
```

Which as you can imagine, gets a little confusing when you've got a lot of tuple data to deal with.




**STEP 7** Tuples though utilise a feature called unpacking, where the data items stored within a tuple are assigned variables. First create the tuple with two items (name and combat rating):

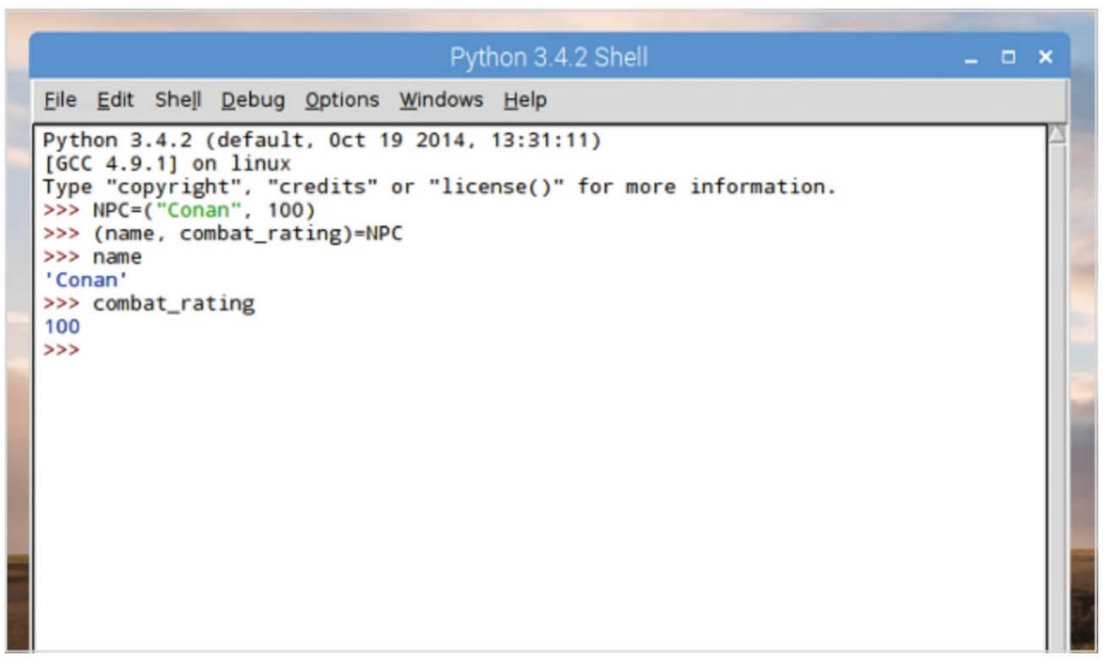
```
NPC=("Conan", 100)
```




**STEP 8** Now unpack the tuple into two corresponding variables:

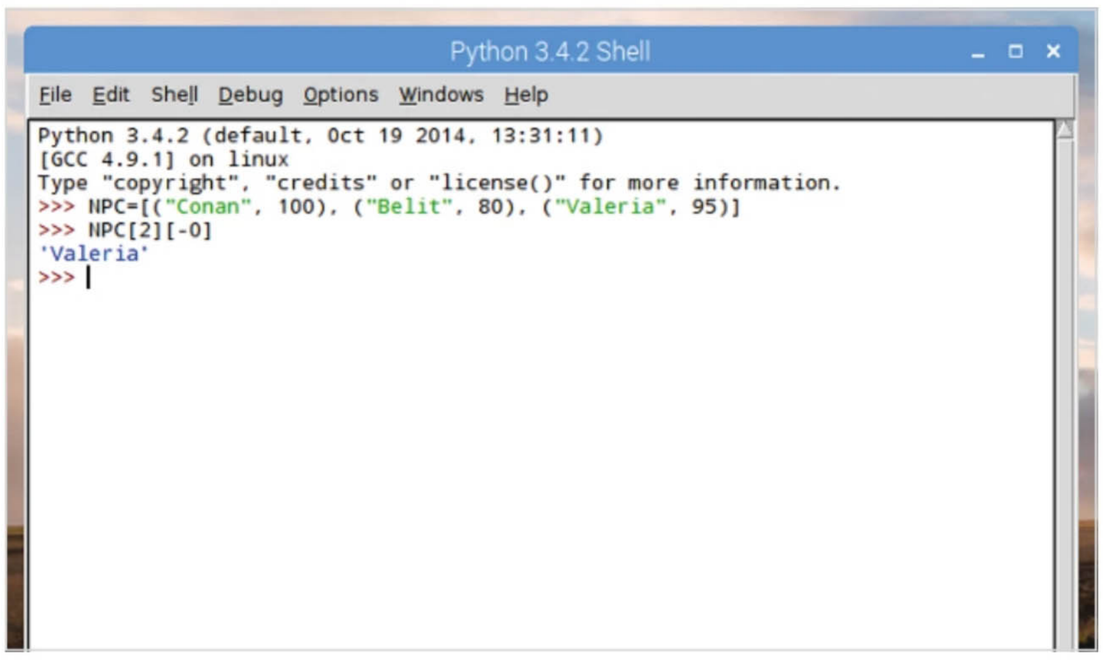
```
(name, combat_rating)=NPC
```

You can now check the values by entering name and combat_rating.




**STEP 9** Remember, as with lists, you can also index tuples using negative numbers which count backwards from the end of the data list. For our example, using the tuple with multiple data items, you would reference the Valeria character with:

```
NPC[2][-0]
```

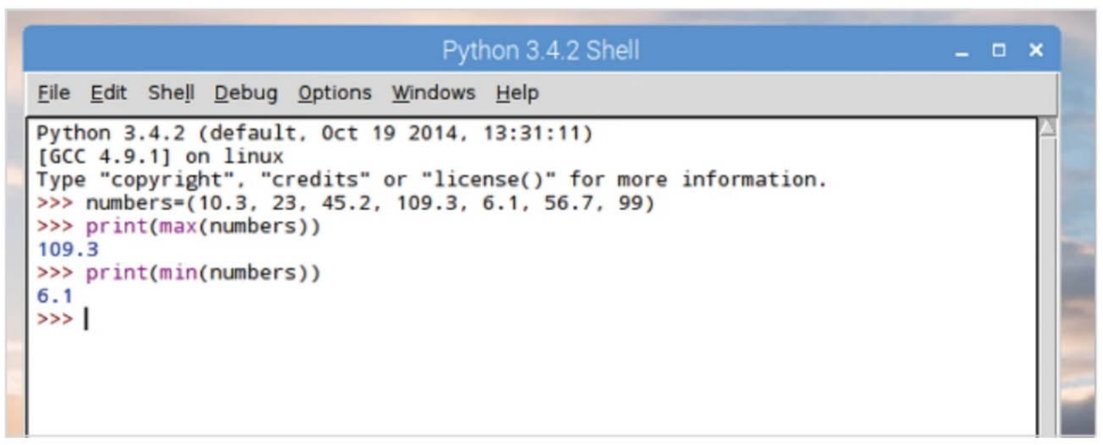



**STEP 10** You can use the max and min functions to find the highest and lowest values of a tuple composed of numbers. For example:

```
numbers=(10.3, 23, 45.2, 109.3, 6.1, 56.7, 99)
```

The numbers can be integers and floats. To output the highest and lowest, use:

```
print(max(numbers))
print(min(numbers))
```



# Dictionaries

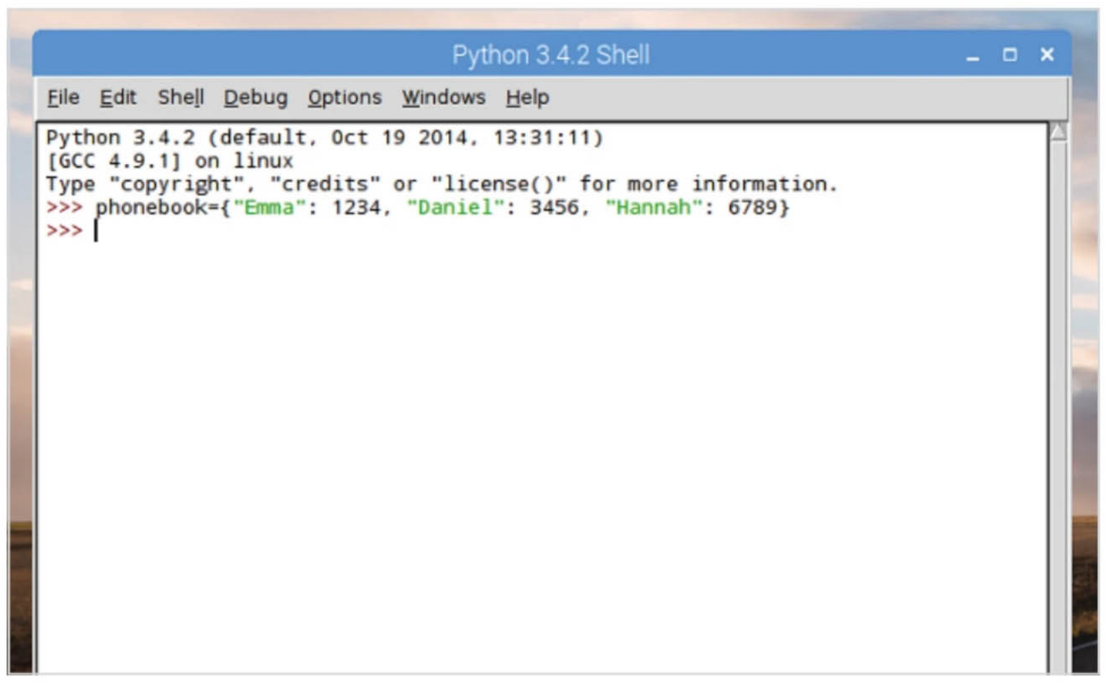
Lists are extremely useful but dictionaries in Python are by far the more technical way of dealing with data items. They can be tricky to get to grips with at first but you'll soon be able to apply them to your own code.

## KEY PAIRS

**A dictionary is like a list but instead each data item comes as a pair, these are known as Key and Value. The Key part must be unique and can either be a number or string whereas the Value can be any data item you like.**

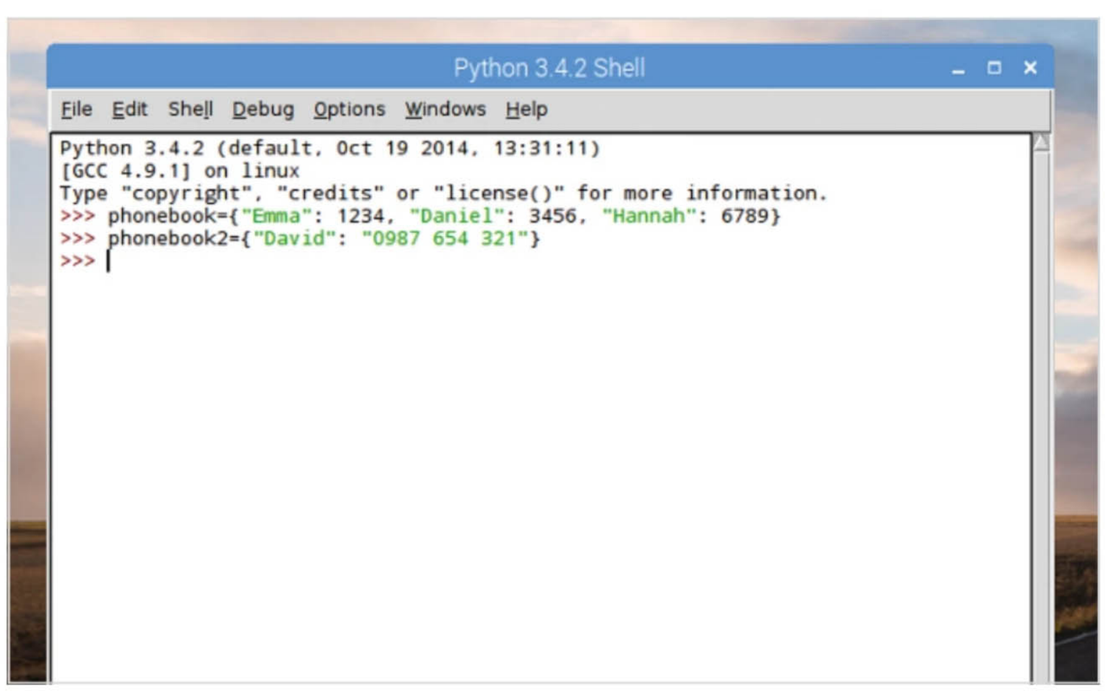
**STEP 1** Let's say you want to create a phonebook in Python. You would create the dictionary name and enter the data in curly brackets, separating the key and value by a colon `Key:Value`. For example:

```
phonebook={"Emma": 1234, "Daniel": 3456, "Hannah": 6789}
```

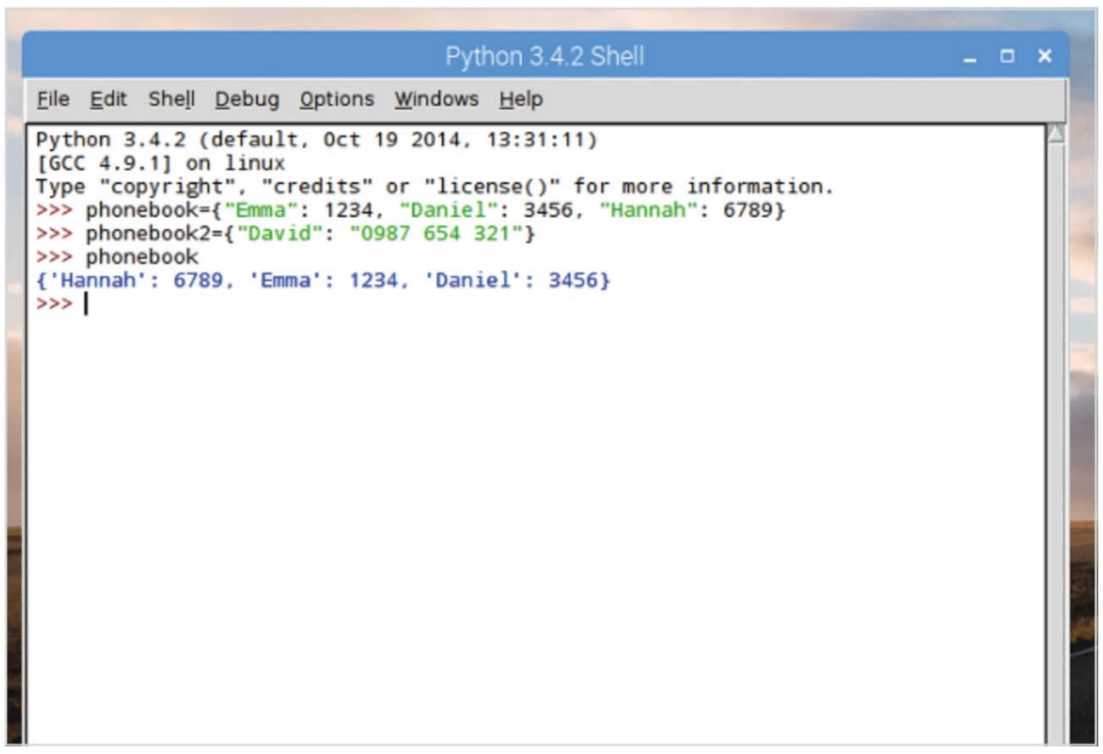



**STEP 2** Just as with most lists, tuples and so on, strings need be enclosed in quotes (single or double), whilst integers can be left open. Remember that the value can be either a string or an integer, you just need to enclose the relevant one in quotes:

```
phonebook2={"David": "0987 654 321"}
```

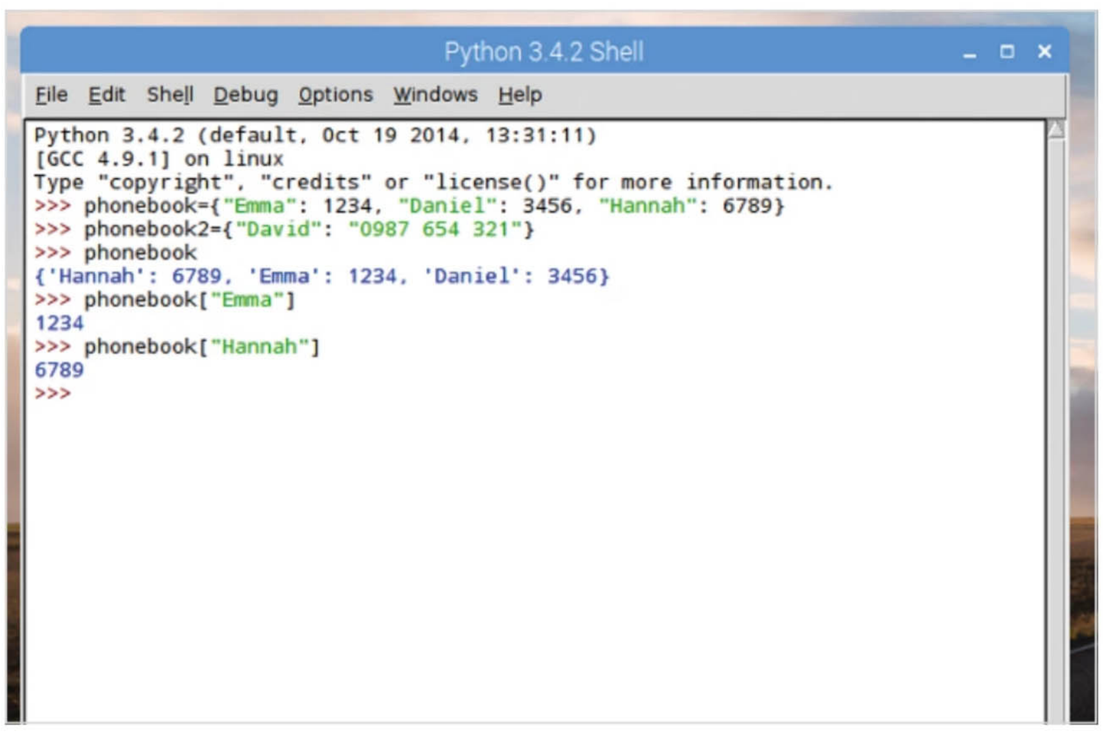



**STEP 3** As with lists and tuples, you can check the contents of a dictionary by giving the dictionary a name: phonebook, in this example. This will display the data items you've entered in a similar fashion to a list, which you're no doubt familiar with by now.



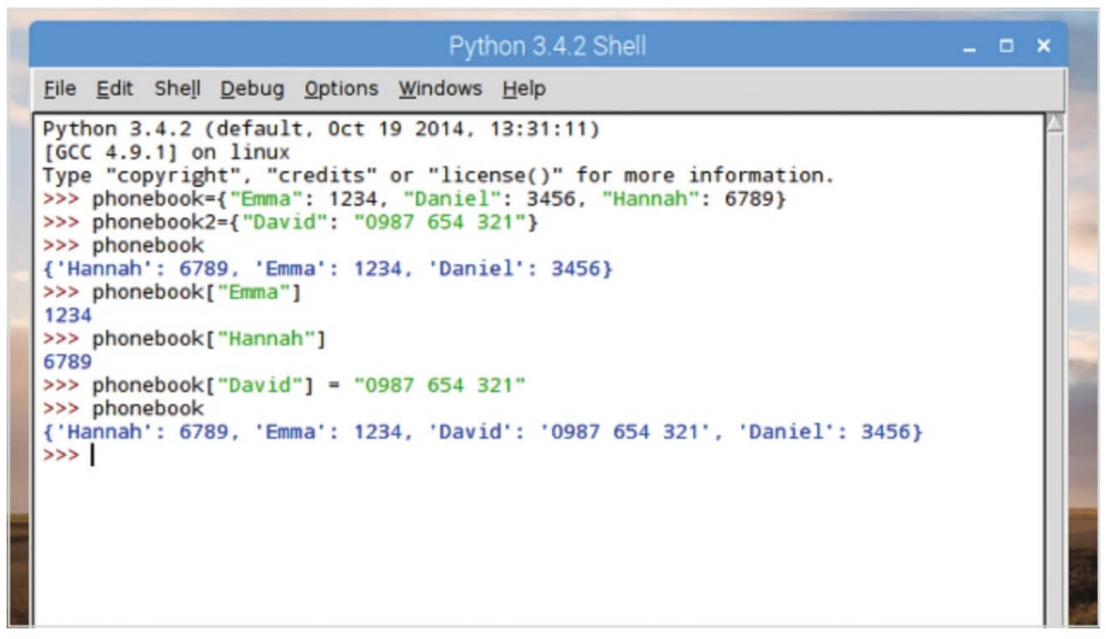

**STEP 4** The benefit of using a dictionary is that you can enter the key to index the value. Using the phonebook example from the previous steps, you can enter:

```
phonebook["Emma"]
phonebook["Hannah"]
```



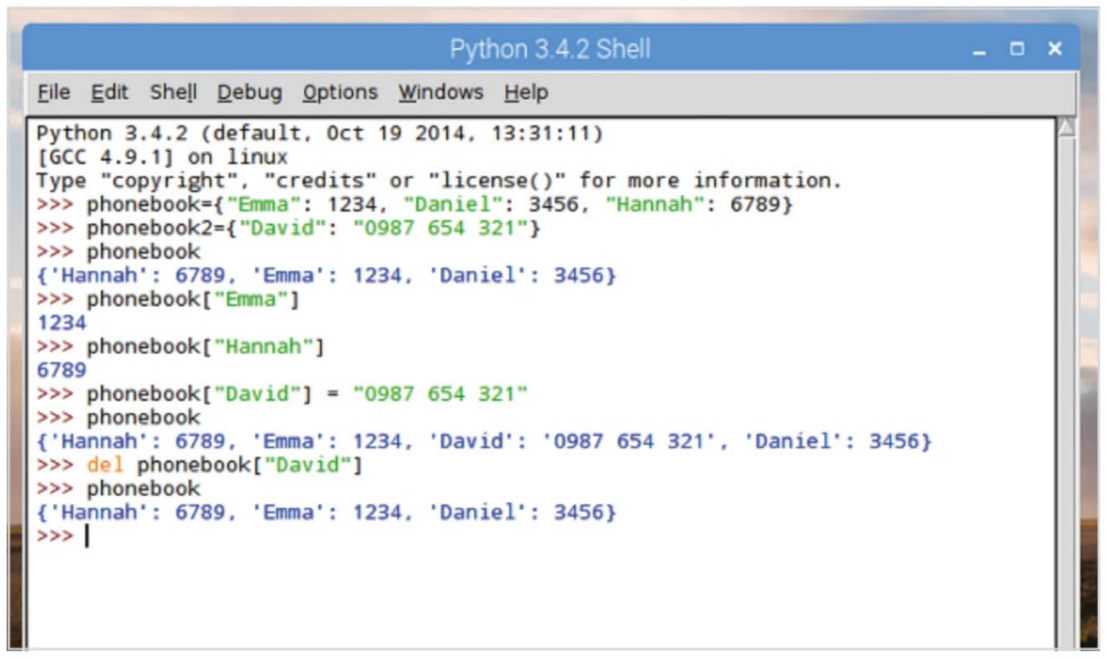
**STEP 5** Adding to a dictionary is easy too. You can include a new data item entry by adding the new key and value items like:

```
phonebook[“David”] = “0987 654 321”
phonebook
```



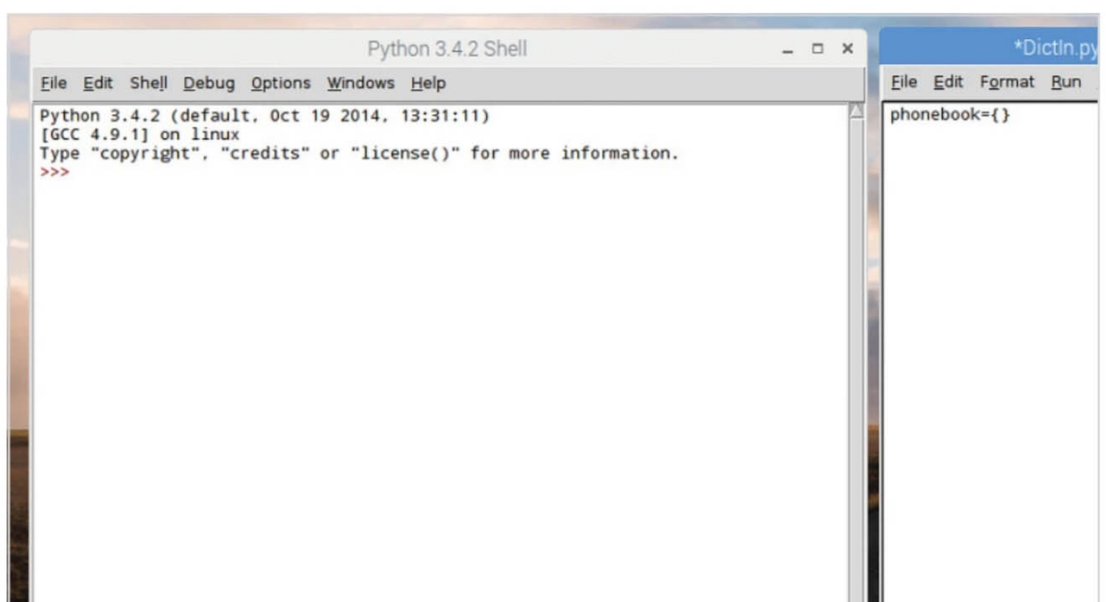

**STEP 6** You can also remove items from a dictionary by issuing the del command followed by the item’s key; the value will be removed as well, since both work as a pair of data items:

```
del phonebook[“David”]
```




**STEP 7** Taking this a step further, how about creating a piece of code that will ask the user for the dictionary key and value items? Create a new Editor instance and start by coding in a new, blank dictionary:

```
phonebook={}
```




**STEP 8** Next, you need to define the user inputs and variables: one for the person’s name, the other for their phone number (let’s keep it simple to avoid lengthy Python code):

```
name=input(“Enter name: “)
number=int(input(“Enter phone number: “))
```

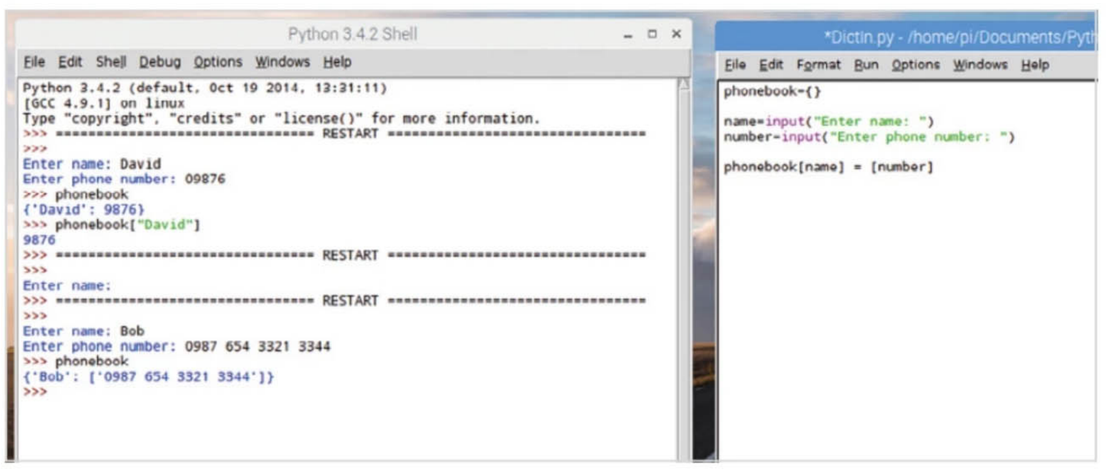
**STEP 9** Note we’ve kept the number as an integer instead of a string, even though the value can be both an integer or a string. Now you need to add the user’s inputted variables to the newly created blank dictionary. Using the same process as in Step 5, you can enter:

```
phonebook[name] = number
```

**STEP 10** Now when you save and execute the code, Python will ask for a name and a number. It will then insert those entries into the phonebook dictionary, which you can test by entering into the Shell:

```
phonebook
phonebook[“David”]
```

If the number needs to contain spaces you need to make it a string, so remove the int part of the input.



# Splitting and Joining Strings

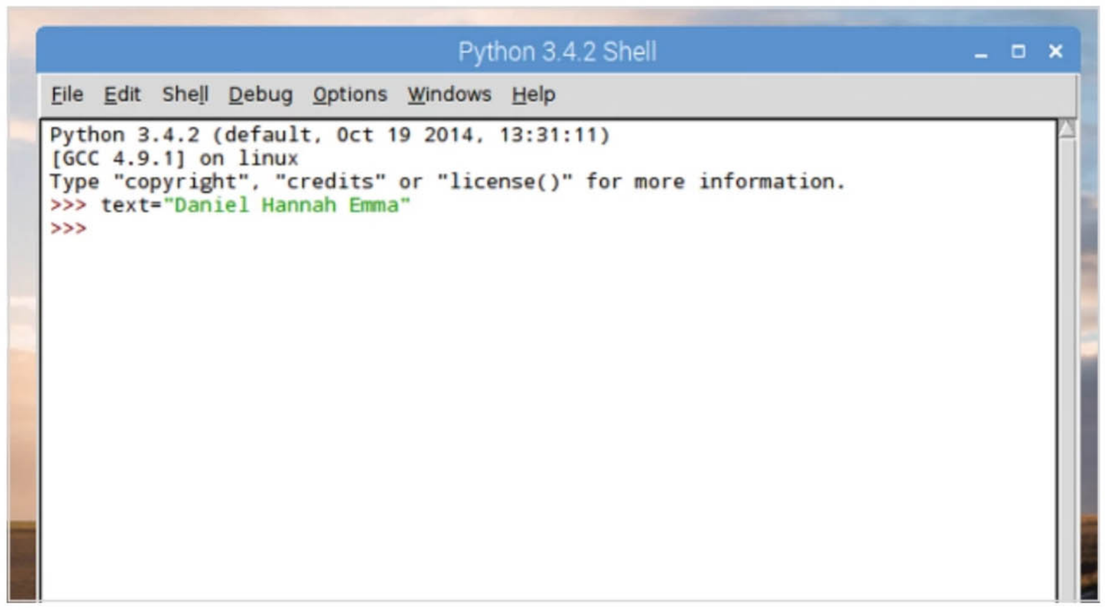
When dealing with data in Python, especially from a user's input, you will undoubtedly come across long sets of strings. A useful skill to learn in Python programming is being able to split those long strings for better readability.

## STRING THEORIES

**You've already looked at some list functions, using .insert, .remove, and .pop but there are also functions that can be applied to strings.**

**STEP 1** The main tool in the string function arsenal is .split(). With it you're able to split apart a string of data, based on the argument within the brackets. For example, here's a string with three items, each separated by a space:

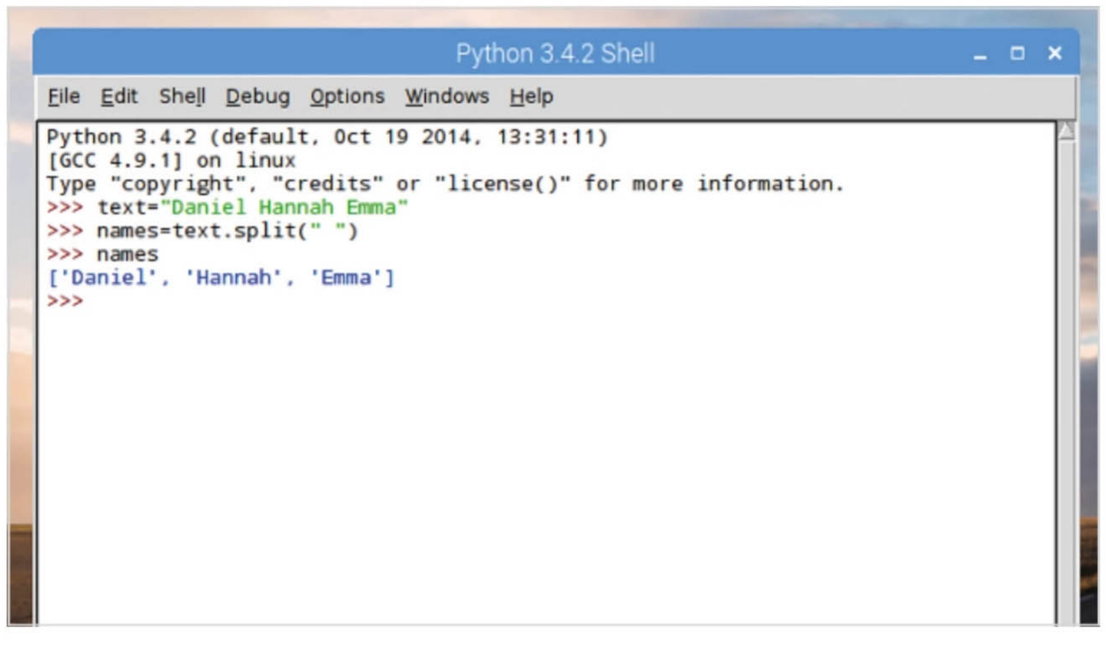
```
text=”Daniel Hannah Emma”
```




**STEP 2** Now let's turn the string into a list and split the content accordingly:

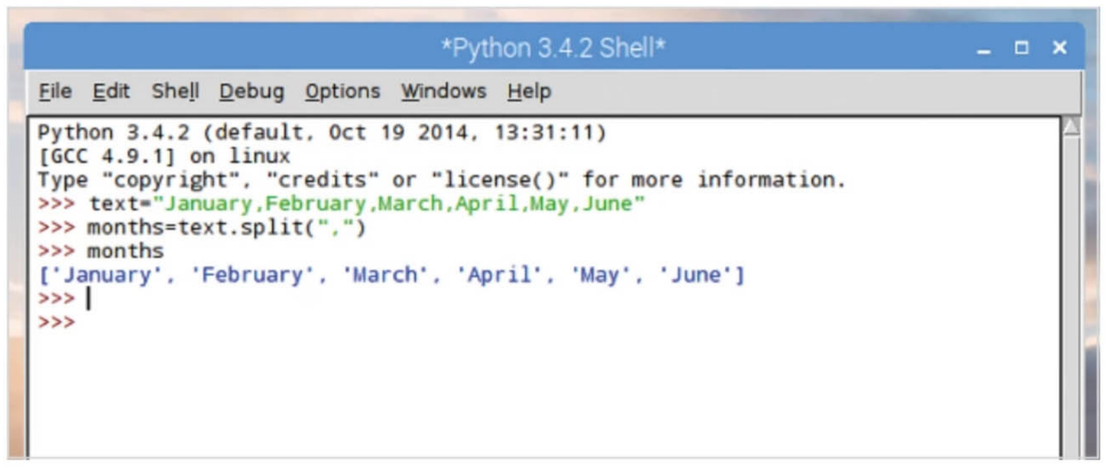
```
names=text.split(“ “)
```

Then enter the name of the new list, names, to see the three items.




**STEP 3** Note that the text.split part has the brackets, quotes, then a space followed by closing quotes and brackets. The space is the separator, indicating that each list item entry is separated by a space. Likewise, CSV (Comma Separated Value) content has a comma, so you'd use:

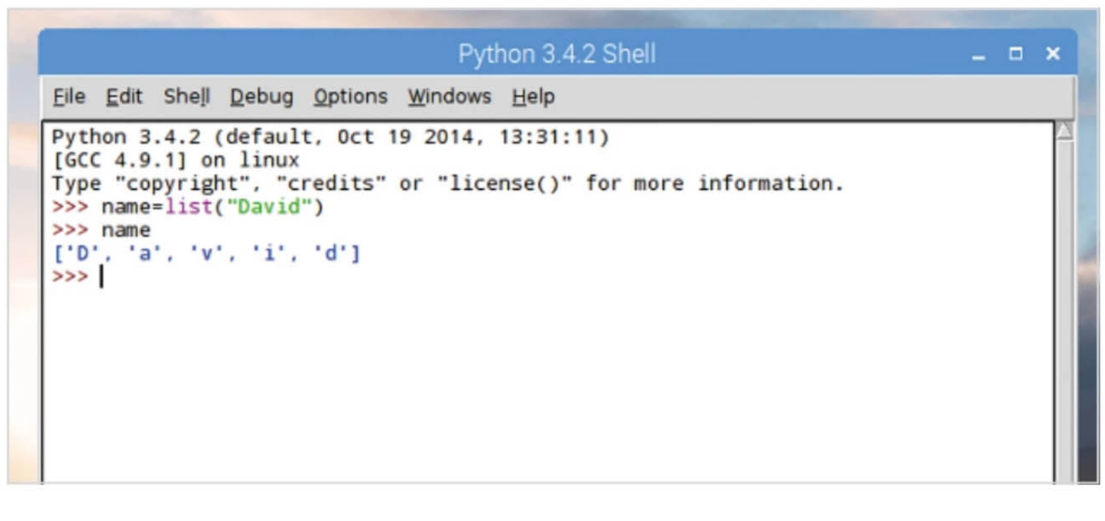
```
text=”January,February,March,April,May,June”
months=text.split(“,”)
months
```




**STEP 4** You've previously seen how you can split a string into individual letters as a list, using a name:

```
name=list(“David”)
name
```

The returned value is 'D', 'a', 'v', 'i', 'd'. Whilst it may seem a little useless under ordinary circumstances, it could be handy for creating a spelling game for example.

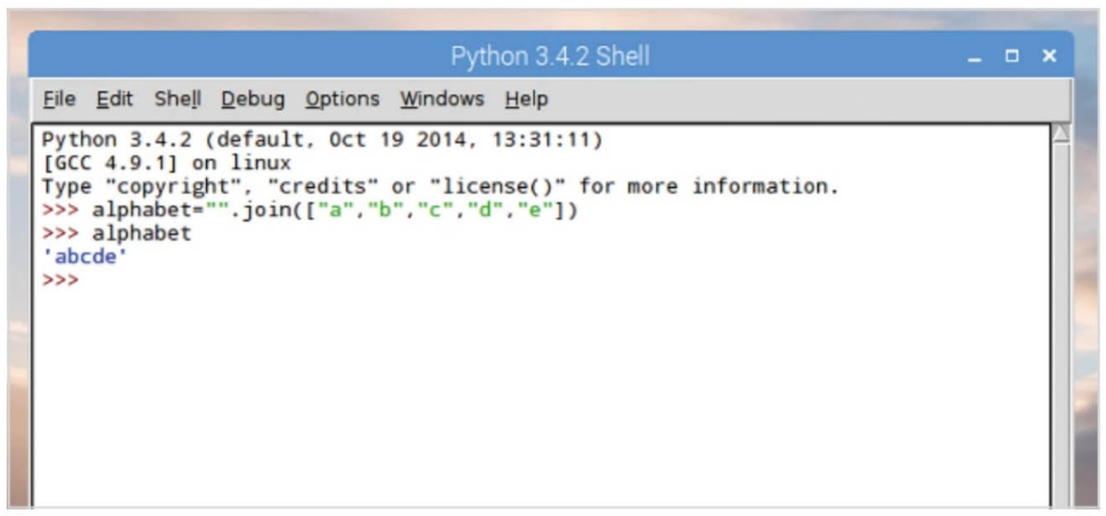


## STEP 5

The opposite of the .split function is .join, where you will have separate items in a string and can join them all together to form a word or just a combination of items, depending on the program you're writing. For instance:

```
alphabet="".join(["a","b","c","d","e"])
alphabet
```

This will display 'abcde' in the Shell.

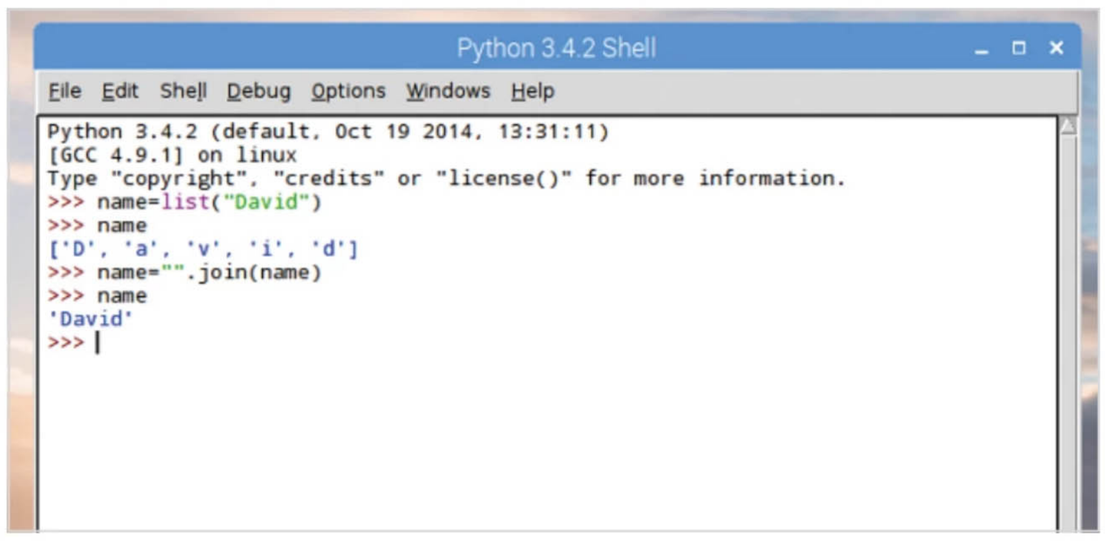



## STEP 6

You can therefore apply .join to the separated name you made in Step 4, combining the letters again to form the name:

```
name="".join(name)
name
```

We've joined the string back together, and retained the list called name, passing it through the .join function.

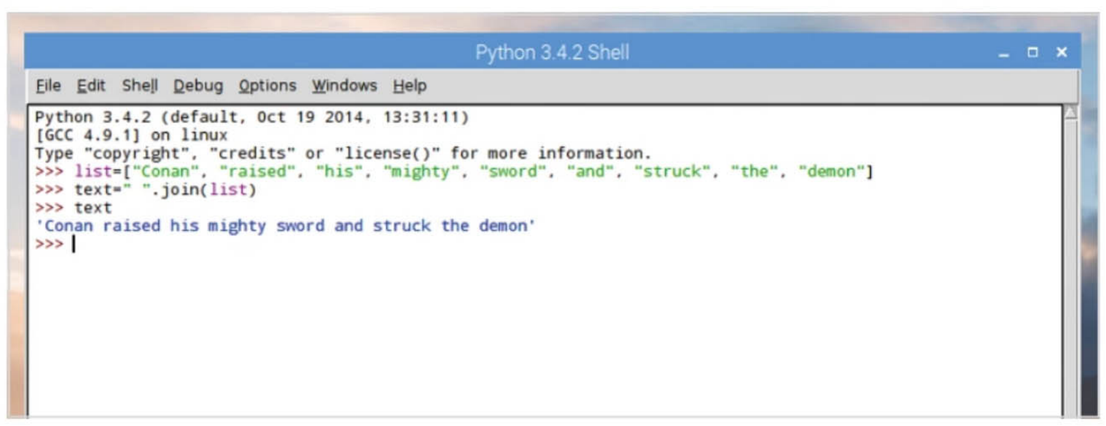



## STEP 7

A good example of using the .join function is when you have a list of words you want to combine into a sentence:

```
list=["Conan", "raised", "his", "mighty", "sword",
"and", "struck", "the", "demon"]
text=" ".join(list)
text
```

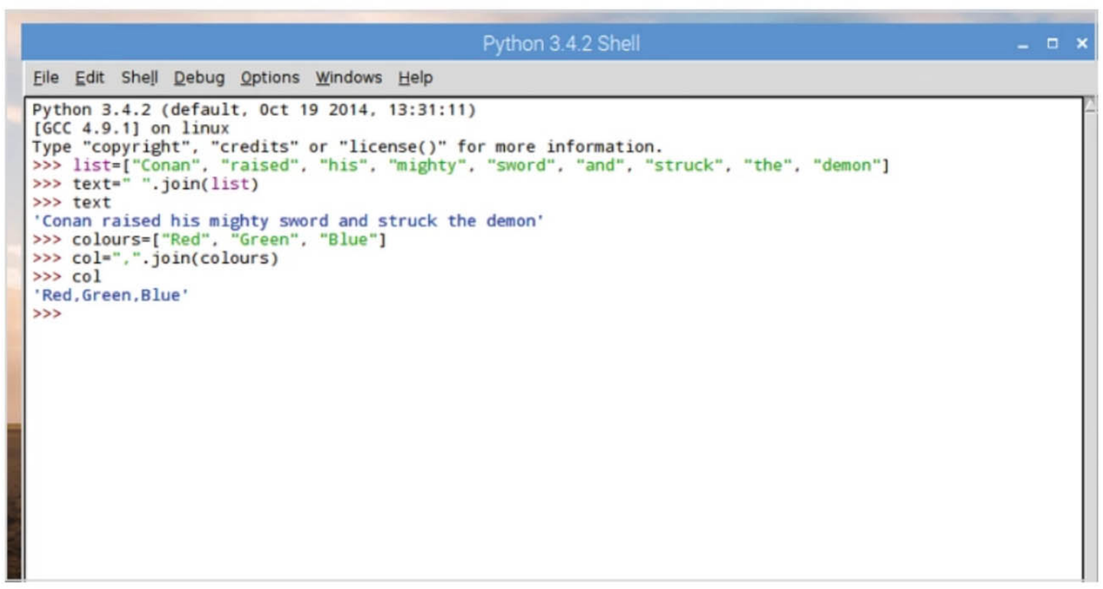
Note the space between the quotes before the .join function (where there were no quotes in Step 6's .join).




## STEP 8

As with the .split function, the separator doesn't have to be a space, it can also be a comma, a full stop, a hyphen or whatever you like:

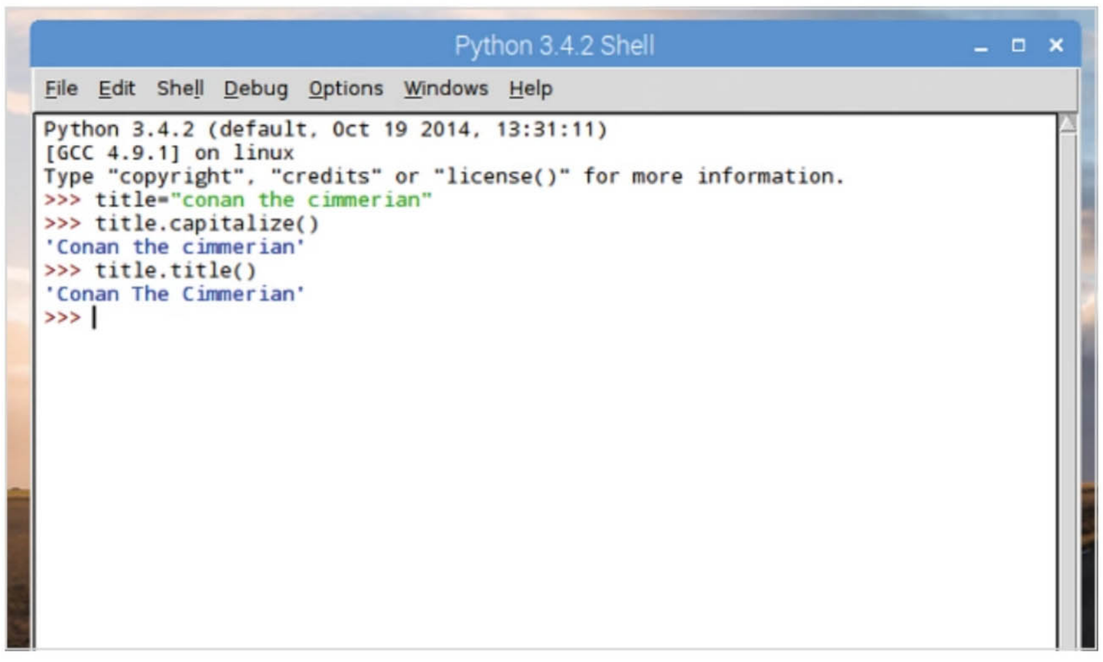
```
colours=["Red", "Green", "Blue"]
col=",".join(colours)
col
```




## STEP 9

There's some interesting functions you apply to a string, such as .capitalize and .title. For example:

```
title="conan the cimmerian"
title.capitalize()
title.title()
```

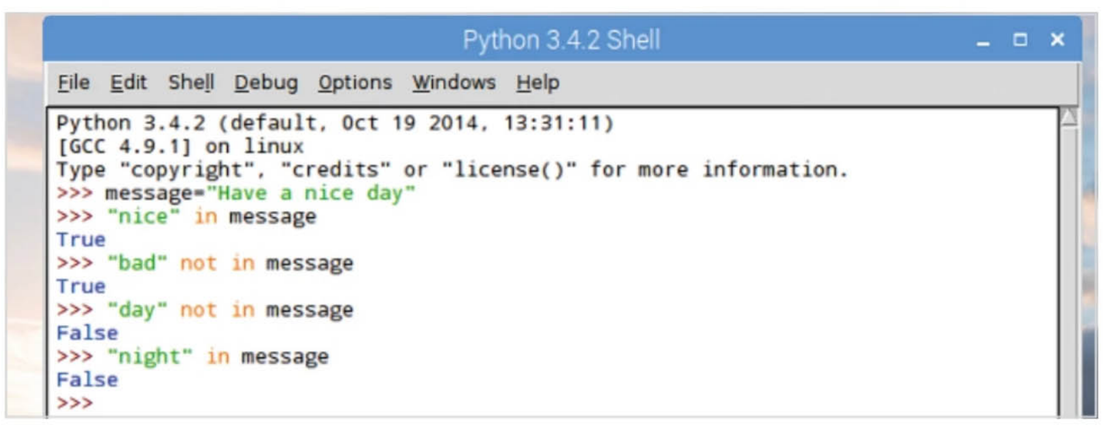



## STEP 10

You can also use logic operators on strings, with the 'in' and 'not in' functions. These enable you to check if a string contains (or does not contain) a sequence of characters:

```
message="Have a nice day"
"nice" in message

"bad" not in message
"day" not in message
"night" in message
```



# Formatting Strings

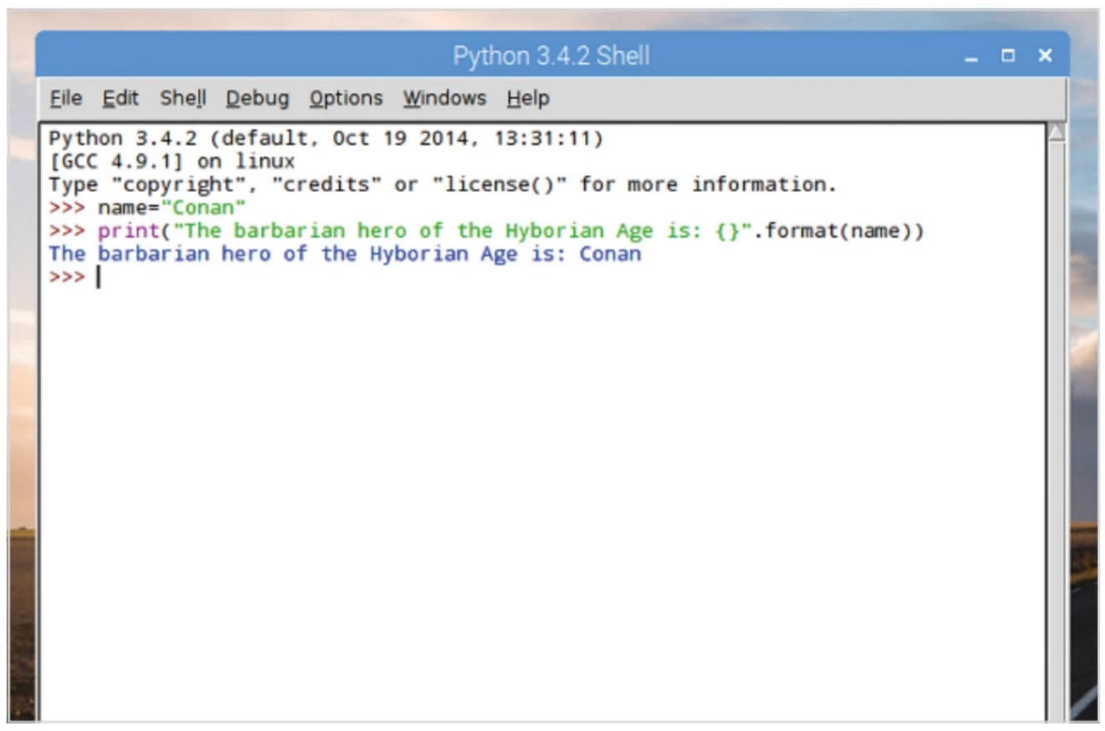
When you work with data, creating lists, dictionaries and objects you may often want to print out the results. Merging strings with data is easy especially with Python 3, as earlier versions of Python tended to complicate matters.

## STRING FORMATTING

**Since Python 3, string formatting has become a much neater process, using the .format function combined with curly brackets. It's a more logical and better formed approach than previous versions.**

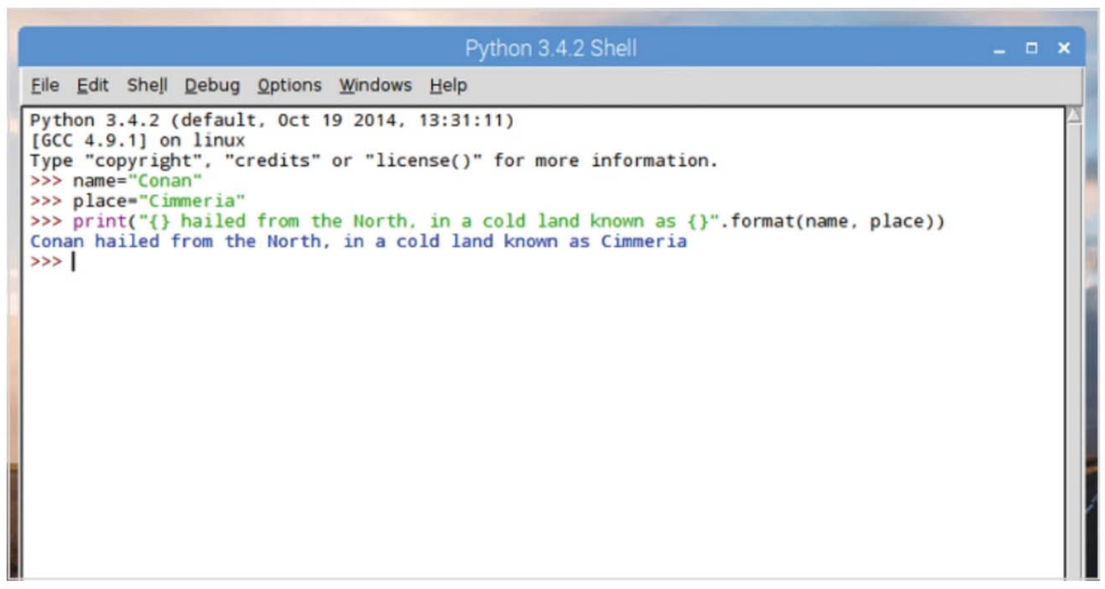
**STEP 1** The basic formatting in Python is to call each variable into the string using the curly brackets:

```
name="Conan"
print("The barbarian hero of the Hyborian Age is:
{}".format(name))
```



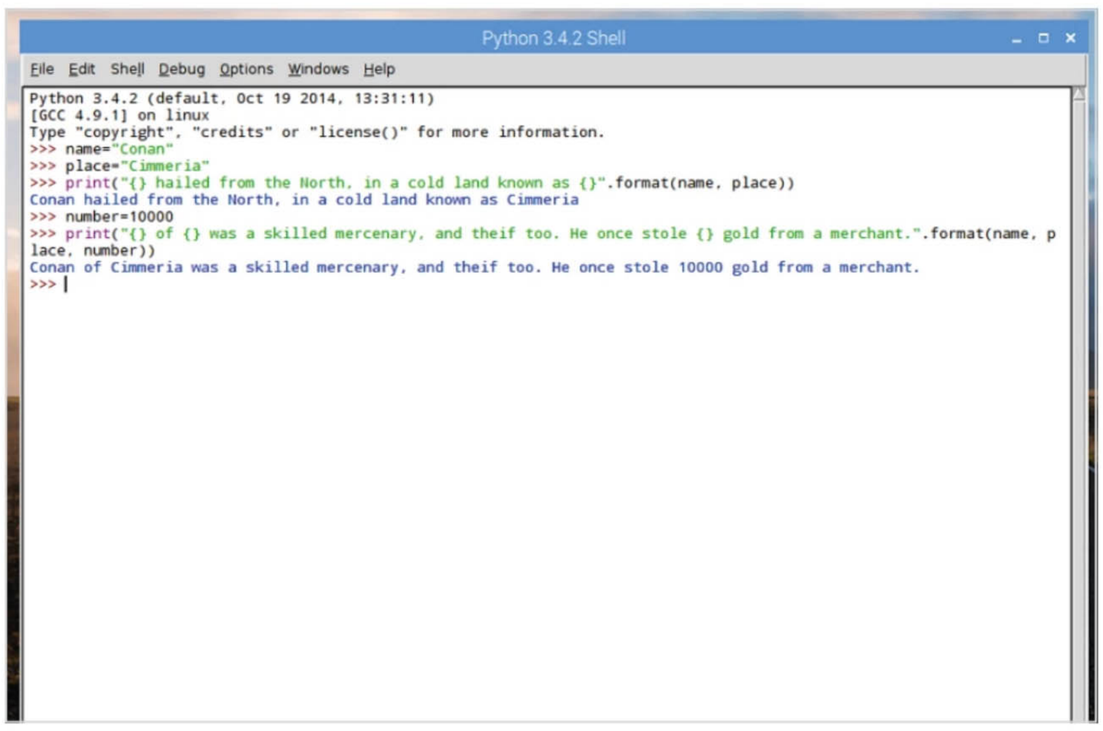

**STEP 2** Remember to close the print function with two sets of brackets, as you've encased the variable in one, and the print function in another. You can include multiple cases of string formatting in a single print function:

```
name="Conan"
place="Cimmeria"
print("{} hailed from the North, in a cold land
known as {}".format(name, place))
```

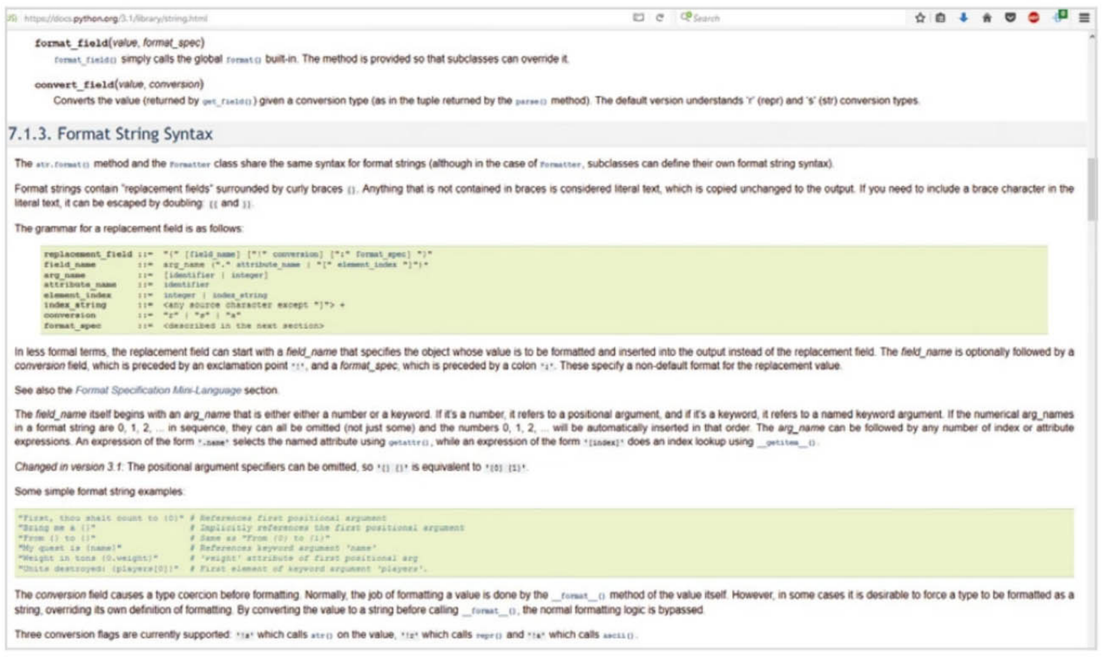



**STEP 3** You can of course also include integers into the mix:

```
number=10000
print("{} of {} was a skilled mercenary,
and thief too. He once stole {} gold from a
merchant.".format(name, place, number))
```




**STEP 4** There are many different ways to apply string formatting, some are quite simple, as we've shown you here; others can be significantly more complex. It all depends on what you want from your program. A good place to reference frequently regarding string formatting is the Python Docs webpage, found at **www.docs.python.org/3.1/library/string.html**. Here, you will find tons of help.



**format_field**(*value*, *format_spec*)

format_field() simply calls the global format() built-in. The method is provided so that subclasses can override it.

**convert_field**(*value*, *conversion*)

Converts the value (returned by get_field()) given a conversion type (as in the tuple returned by the parse() method). The default version understands 'r' (repr) and 's' (str) conversion types.

### 7.1.3. Format String Syntax

The str.format() method and the Formatter class share the same syntax for format strings (although in the case of Formatter, subclasses can define their own format string syntax).

Format strings contain "replacement fields" surrounded by curly braces {}. Anything that is not contained in braces is considered literal text, which is copied unchanged to the output. If you need to include a brace character in the literal text, it can be escaped by doubling: {{ and }}.

The grammar for a replacement field is as follows:

```
replacement_field ::=  "{" [field_name] ["!" conversion] [":" format_spec] "}"
field_name        ::=  arg_name ("." attribute_name | "[" element_index "]")*
arg_name          ::=  [identifier | integer]
attribute_name    ::=  identifier
element_index     ::=  integer | index_string
index_string      ::=  <any source character except "]"> +
conversion        ::=  "r" | "s" | "a"
format_spec       ::=  <described in the next section>
```

In less formal terms, the replacement field can start with a *field_name* that specifies the object whose value is to be formatted and inserted into the output instead of the replacement field. The *field_name* is optionally followed by a *conversion* field, which is preceded by an exclamation point '!', and a *format_spec*, which is preceded by a colon ':'. These specify a non-default format for the replacement value.

See also the *Format Specification Mini-Language* section.

The *field_name* itself begins with an *arg_name* that is either either a number or a keyword. If it's a number, it refers to a positional argument, and if it's a keyword, it refers to a named keyword argument. If the numerical arg_names in a format string are 0, 1, 2, ... in sequence, they can all be omitted (not just some) and the numbers 0, 1, 2, ... will be automatically inserted in that order. The *arg_name* can be followed by any number of index or attribute expressions. An expression of the form '.name' selects the named attribute using getattr(), while an expression of the form '[index]' does an index lookup using __getitem__().

*Changed in version 3.1*: The positional argument specifiers can be omitted, so '{} {}' is equivalent to '{0} {1}'.

Some simple format string examples:

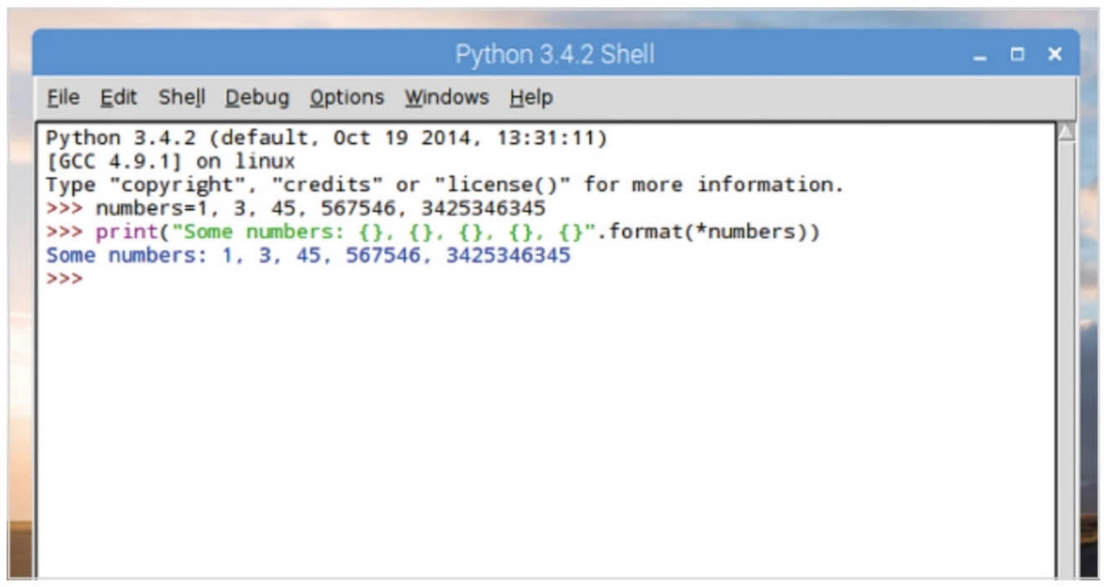
```
"First, thou shalt count to {0}" # References first positional argument
"Bring me a {}"                  # Implicitly references the first positional argument
"From {} to {}"                  # Same as "From {0} to {1}"
"My quest is {name}"             # References keyword argument 'name'
"Weight in tons {0.weight}"      # 'weight' attribute of first positional arg
"Units destroyed: {players[0]}"  # First element of keyword argument 'players'.
```

The *conversion* field causes a type coercion before formatting. Normally, the job of formatting a value is done by the __format__() method of the value itself. However, in some cases it is desirable to force a type to be formatted as a string, overriding its own definition of formatting. By converting the value to a string before calling __format__(), the normal formatting logic is bypassed.

Three conversion flags are currently supported: '!s' which calls str() on the value, '!r' which calls repr() and '!a' which calls ascii().

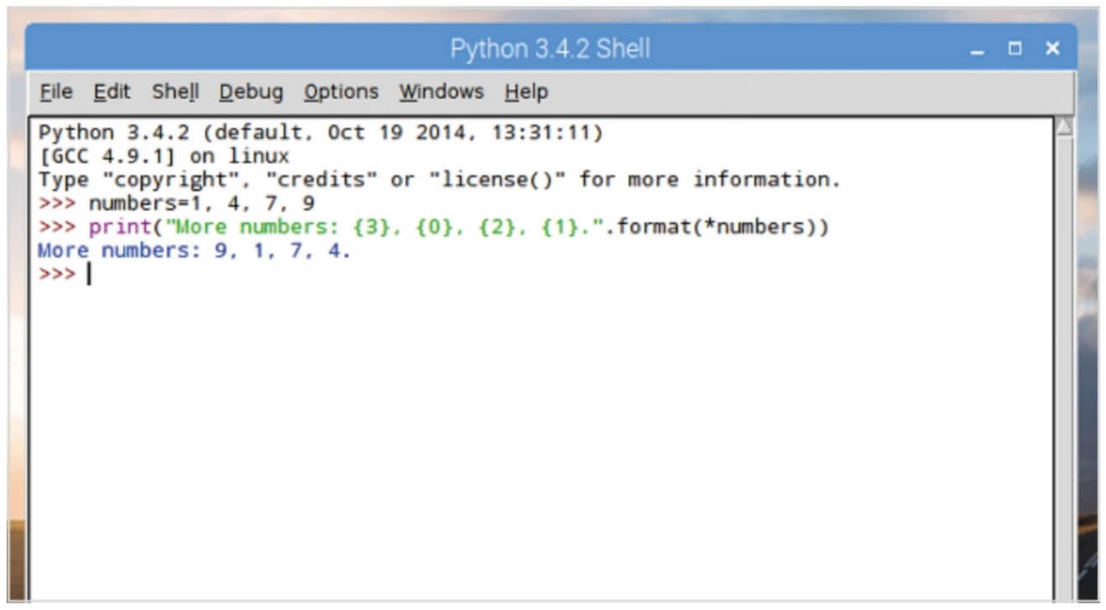
**STEP 5** Interestingly you can reference a list using the string formatting function. You need to place an asterisk in front of the list name:

```
numbers=1, 3, 45, 567546, 3425346345
print("Some numbers: {}, {}, {}, {}, {}".
format(*numbers))
```



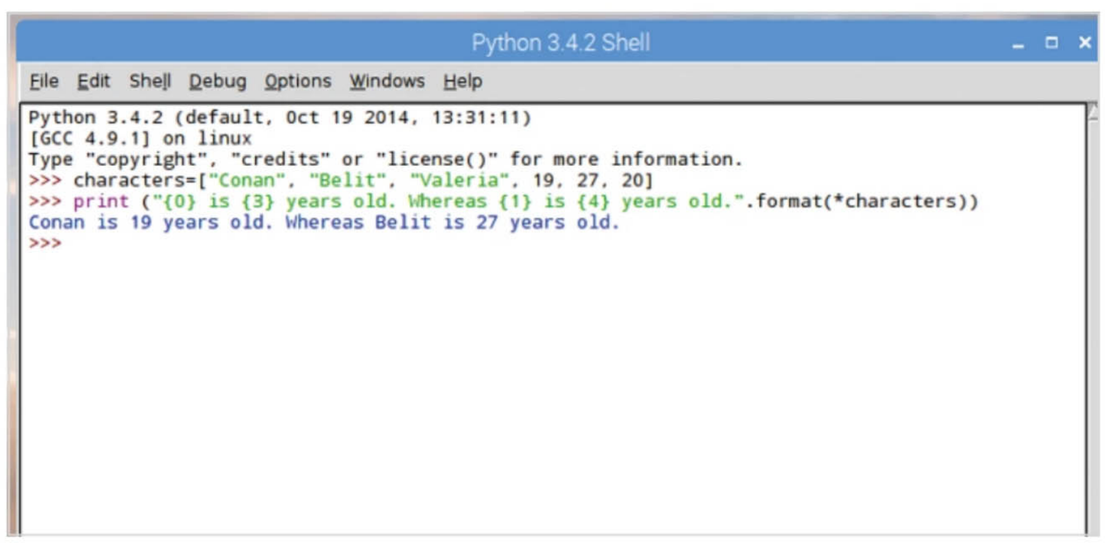

**STEP 6** With indexing in lists, the same applies to calling a list using string formatting. You can index each item according to its position (from 0 to however many are present):

```
numbers=1, 4, 7, 9
print("More numbers: {3}, {0}, {2},
{1}.".format(*numbers))
```



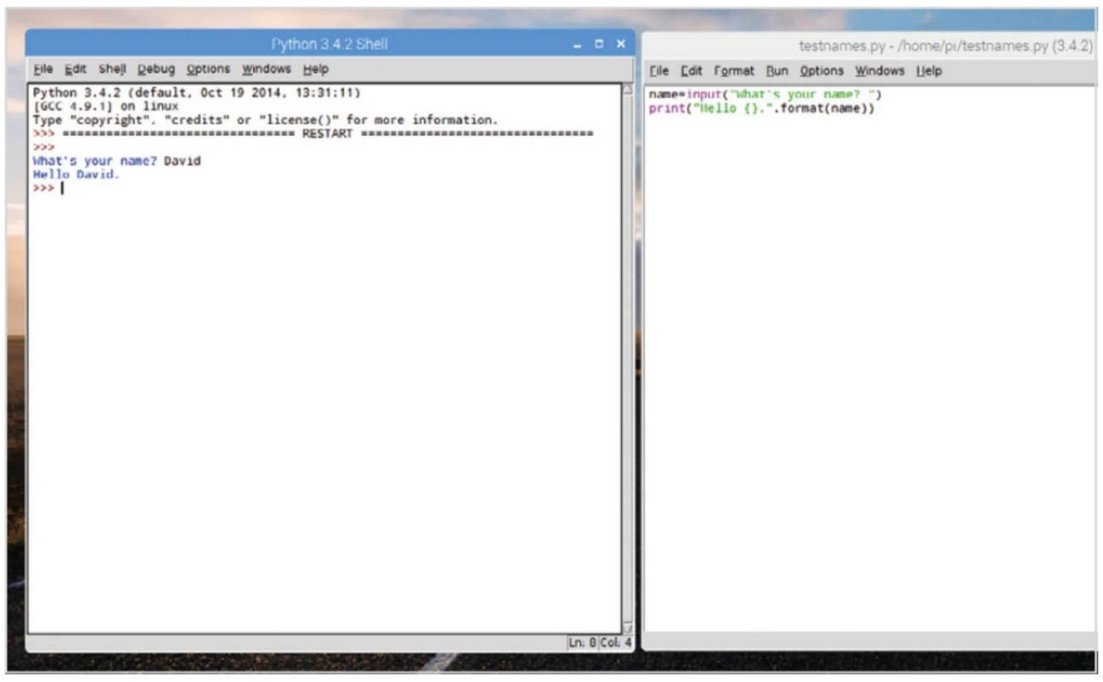

**STEP 7** And as you probably suspect, you can mix strings and integers in a single list to be called in the .format function:

```
characters=["Conan", "Belit", "Valeria", 19, 27,
20]
print ("{0} is {3} years old. Whereas {1} is {4}
years old.".format(*characters))
```



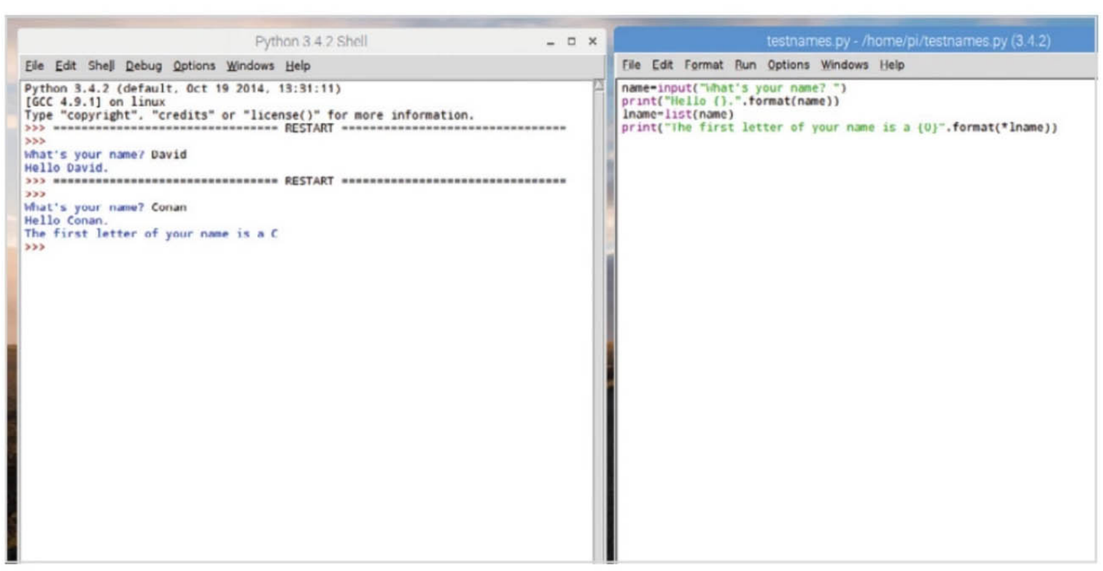

**STEP 8** You can also print out the content of a user's input in the same fashion:

```
name=input("What's your name? ")
print("Hello {}.".format(name)
```




**STEP 9** You can extend this simple code example to display the first letter in a person's entered name:

```
name=input("What's your name? ")
print("Hello {}.".format(name))
lname=list(name)
print("The first letter of your name is a {0}".
format(*lname))
```

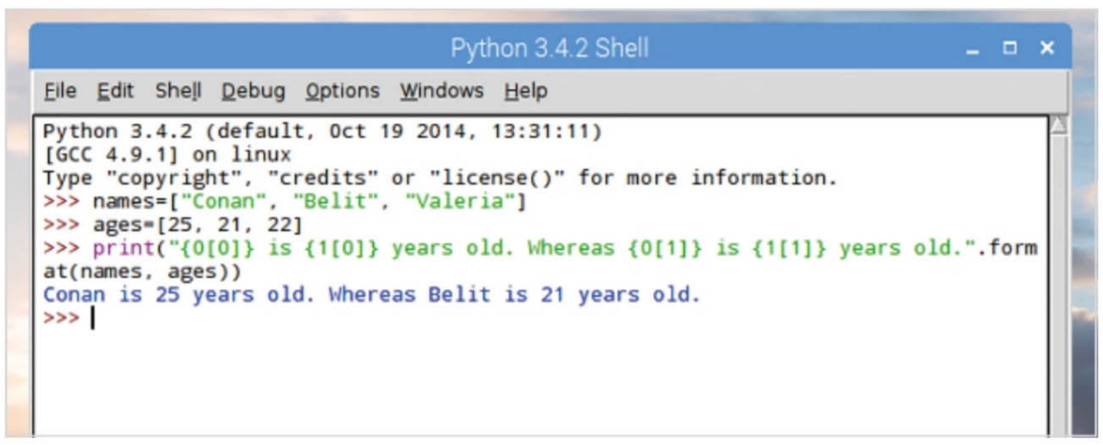



**STEP 10** You can also call upon a pair of lists and reference them individually within the same print function. Looking back the code from Step 7, you can alter it with:

```
names=["Conan", "Belit", "Valeria"]
ages=[25, 21, 22]
```

Creating two lists. Now you can call each list, and individual items:

```
print("{0[0]} is {1[0]} years old. Whereas {0[1]}
is {1[1]} years old.".format(names, ages))
```



# Date and Time

When working with data it's often handy to have access to the time. For example, you may want to time-stamp an entry or see at what time a user logged into the system and for how long. Luckily acquiring the date and time is easy, thanks to the Time module.

## TIME LORDS

**The Time module contains functions that help you retrieve the current system time, reads the date from strings, formats the time and date and much more.**

**STEP 1** First you need to import the Time module. It's one that's built-in to Python 3 so you shouldn't need to drop into a command prompt and pip install it. Once it's imported, you can call the current time and date with a simple command:

```
import time
time.asctime()
```

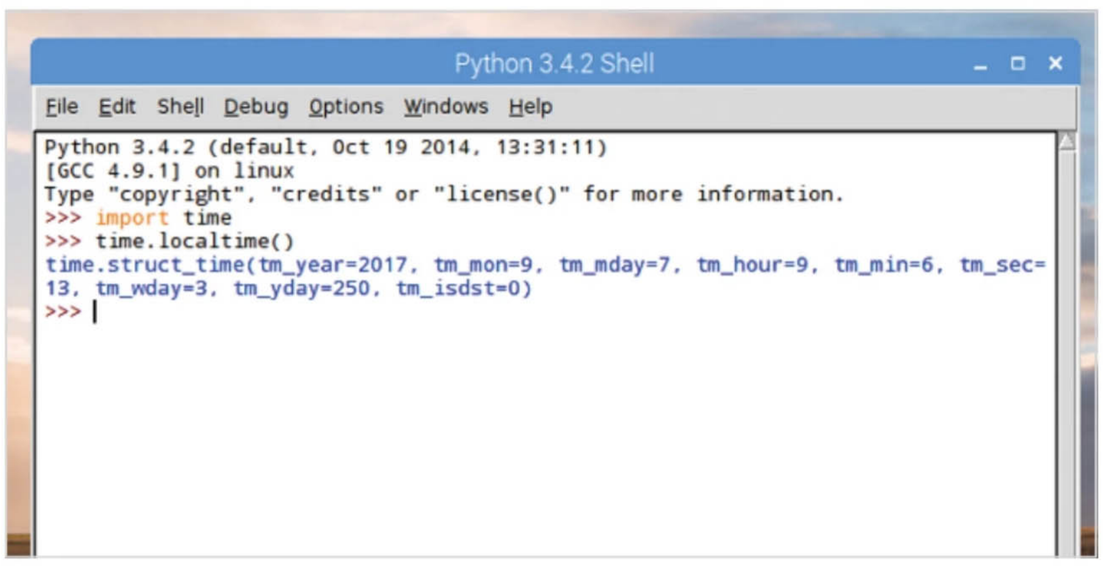
**STEP 2** The time function is split into nine tuples, these are divided up into indexed items, as with any other tuple, and shown in the screen shot below.

| Index | Field | Values |
|---|---|---|
| 0 | 4-digit year | 2016 |
| 1 | Month | 1 to 12 |
| 2 | Day | 1 to 31 |
| 3 | Hour | 0 to 23 |
| 4 | Minute | 0 to 59 |
| 5 | Second | 0 to 61 (60 or 61 are leap-seconds) |
| 6 | Day of Week | 0 to 6 (0 is Monday) |
| 7 | Day of year | 1 to 366 (Julian day) |
| 8 | Daylight savings | -1, 0, 1, -1 means library determines DST |

**STEP 3** You can see the structure of how time is presented by entering:

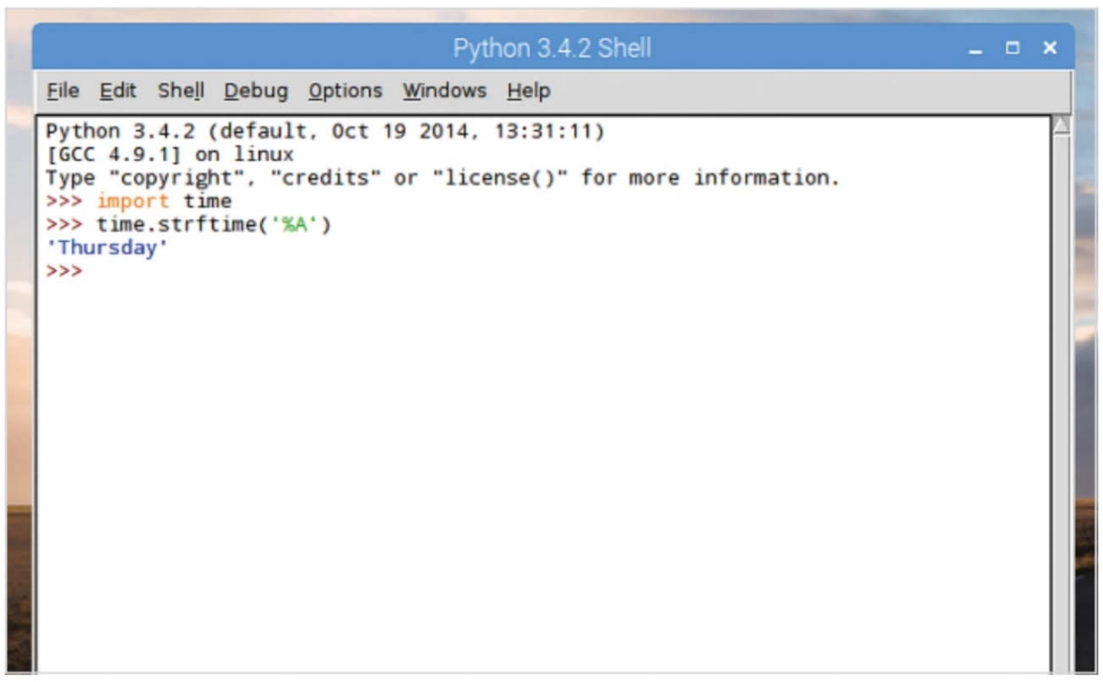
```
time.local.time()
```

The output is displayed as such: `'time.struct_time(tm_year=2017, tm_mon=9, tm_mday=7, tm_hour=9, tm_min=6, tm_sec=13, tm_wday=3, tm_yday=250, tm_isdst=0)'`; obviously dependent on your current time as opposed to the time this book was written.



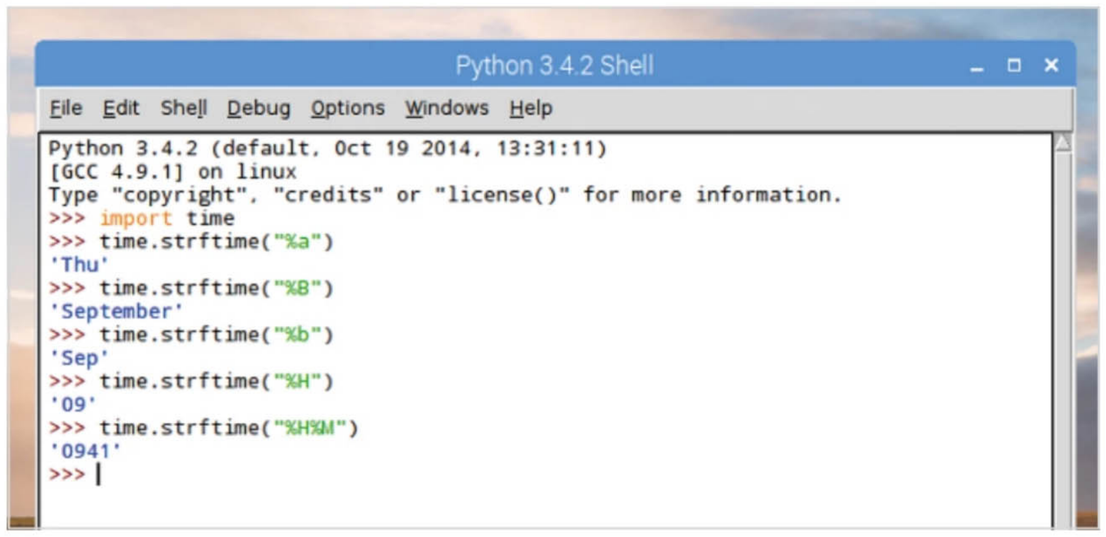

**STEP 4** There are numerous functions built into the Time module. One of the most common of these is .strftime(). With it, you're able to present a wide range of arguments as it converts the time tuple into a string. For example, to display the current day of the week you can use:

```
time.strftime('%A')
```



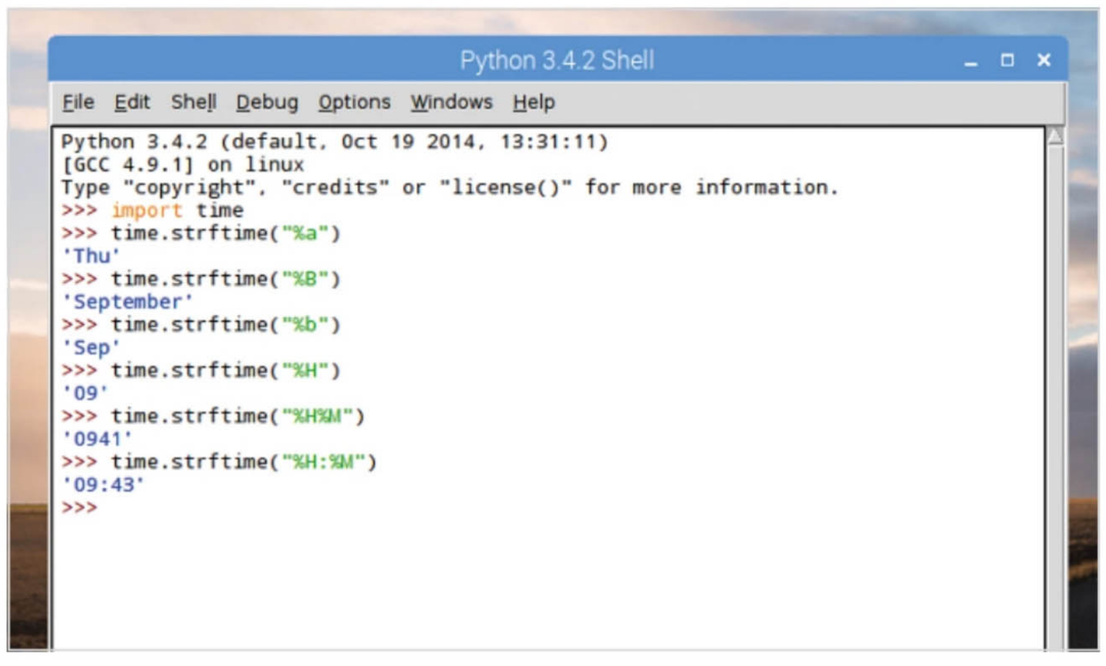
**STEP 5** This naturally means you can incorporate various functions into your own code, such as:

```
time.strftime("%a")
time.strftime("%B")
time.strftime("%b")
time.strftime("%H")
time.strftime("%H%M")
```




**STEP 6** Note the last two entries, with %H and %H%M, as you can see these are the hours and minutes and as the last entry indicates, entering them as %H%M doesn't display the time correctly in the Shell. You can easily rectify this with:

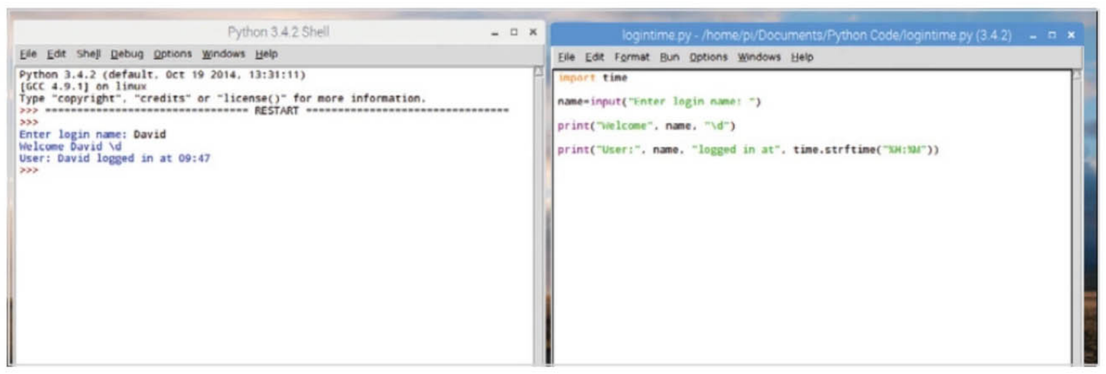
```
time.strftime("%H:%M")
```




**STEP 7** This means you're going to be able to display either the current time or the time when something occurred, such as a user entering their name. Try this code in the Editor:

```
import time
name=input("Enter login name: ")
print("Welcome", name, "\d")
print("User:", name, "logged in at", time.
strftime("%H:%M"))
```

Try to extend it further to include day, month, year and so on.



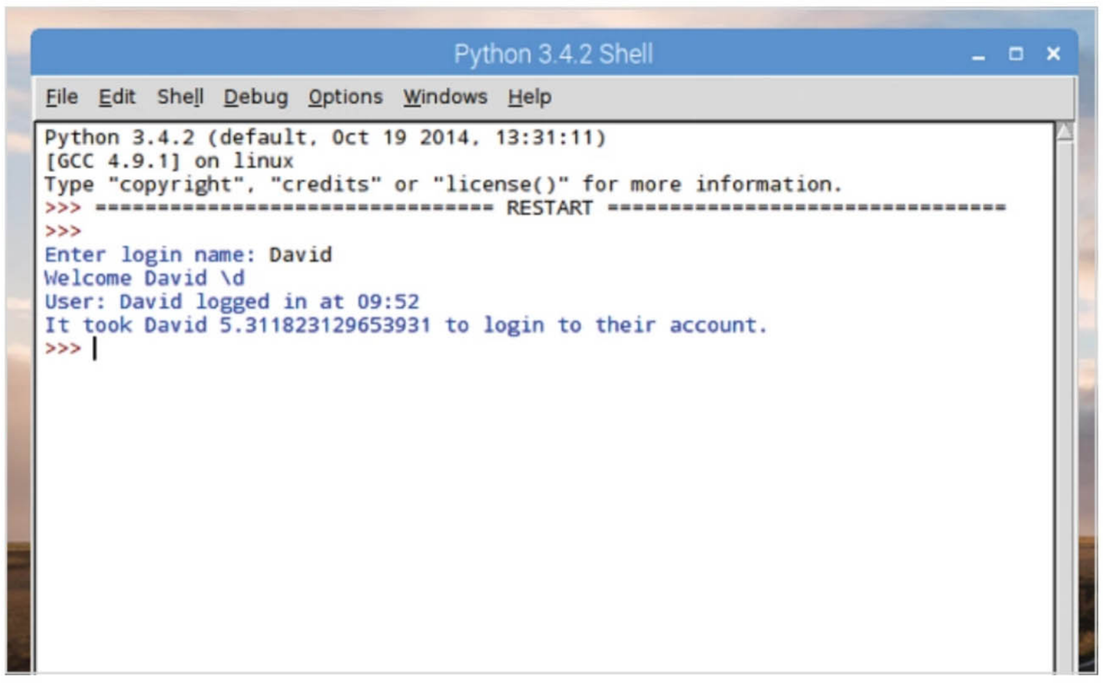

**STEP 8** You saw at the end of the previous section, in the code to calculate Pi to however many decimal places the users wanted, you can time a particular event in Python. Take the code from above and alter it slightly by including:

```
start_time=time.time()
```

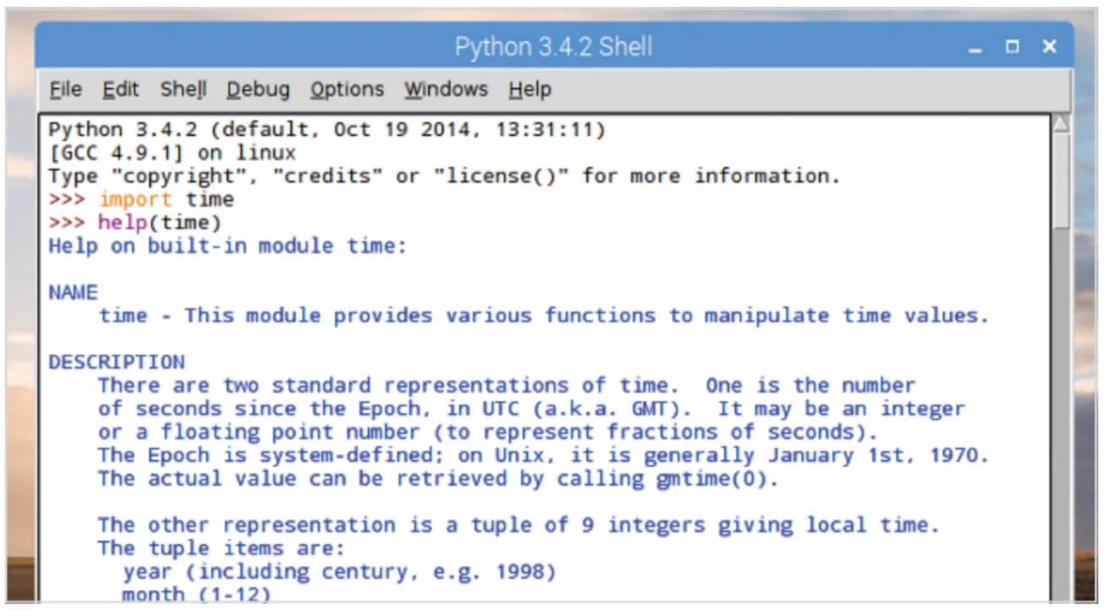
Then there's:

```
endtime=time.time()-start_time
```

**STEP 9** The output will look similar to the screenshot below. The timer function needs to be either side of the input statement, as that's when the variable name is being created, depending on how long the user took to log in. The length of time is then displayed on the last line of the code as the `endtime` variable.




**STEP 10** There's a lot that can be done with the Time module; some of it is quite complex too, such as displaying the number of seconds since January 1st 1970. If you want to drill down further into the Time module, then in the Shell enter: `help(time)` to display the current Python version help file for the Time module.



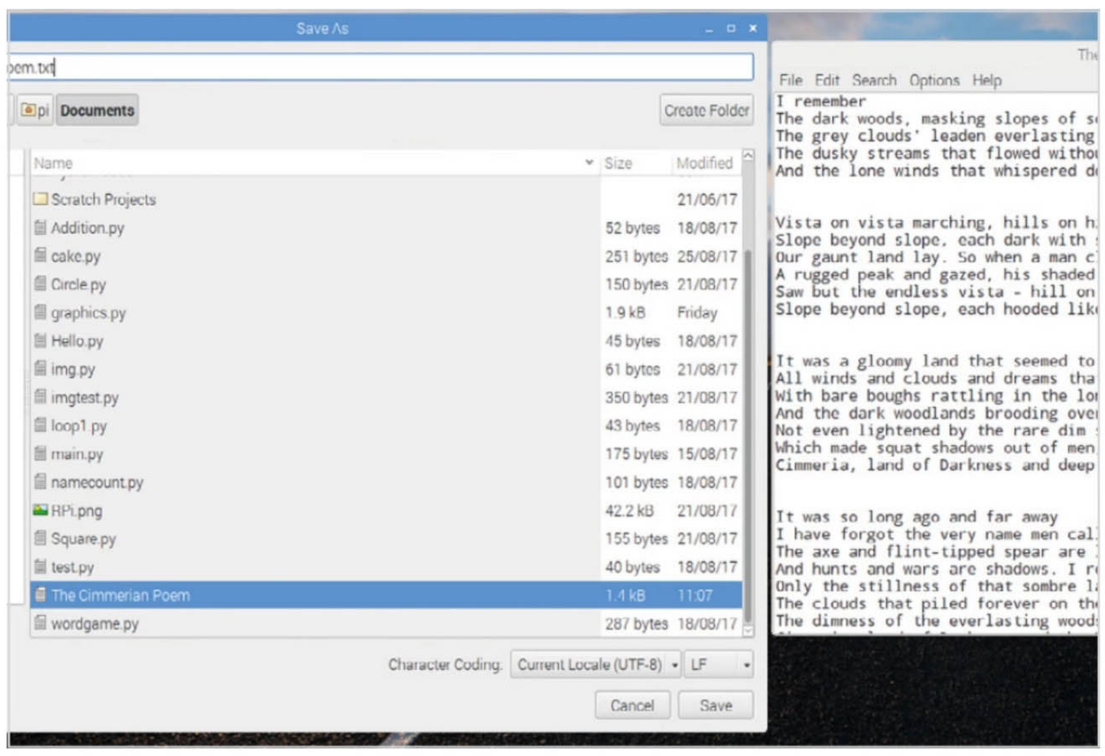
# Opening Files

In Python you can read text and binary files in your programs. You can also write to file, which is something we will look at next. Reading and writing to files enables you to output and store data from your programs.

## OPEN, READ AND WRITE

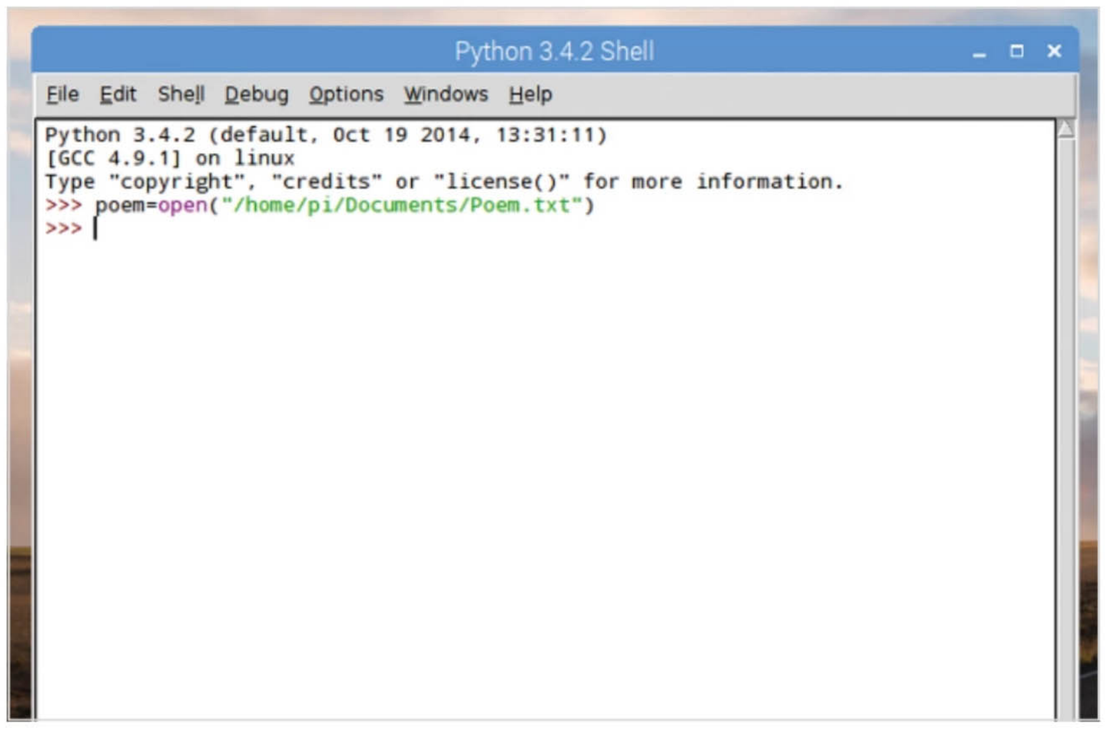
**In Python you create a file object, similar to creating a variable, only pass in the file using the open() function. Files are usually categorised as text or binary.**

**STEP 1** Start by entering some text into your system's text editor. The text editor is best, not a word processor, as word processors include background formatting and other elements. In our example, we have the poem The Cimmerian, by Robert E Howard. You need to save the file as poem.txt.




**STEP 2** You use the open() function to pass the file into a variable as an object. You can name the file object anything you like, but you will need to tell Python the name and location of the text file you're opening:

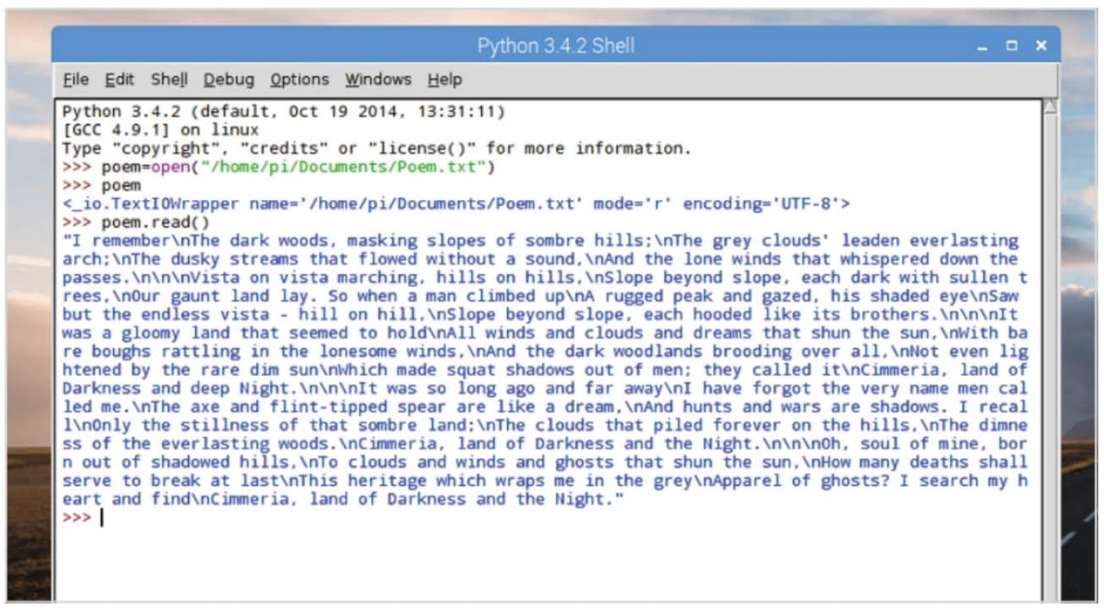
```
poem=open("/home/pi/Documents/Poem.txt")
```




**STEP 3** If you now enter poem into the Shell, you will get some information regarding the text file you've just asked to be opened. You can now use the poem variable to read the contents of the file:

```
poem.read()
```

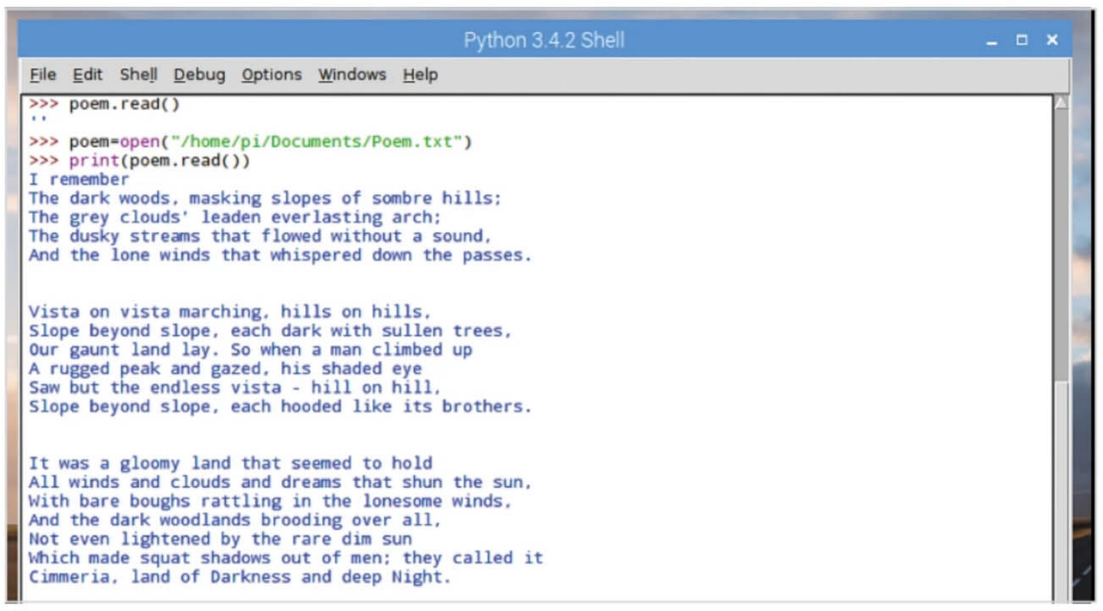
Note than a /n entry in the text represents a new line, as you used previously.




**STEP 4** If you enter poem.read() a second time you will notice that the text has been removed from the file. You will need to enter: `poem=open("/home/pi/Documents/Poem.txt")` again to recreate the file. This time, however, enter:

```
print(poem.read())
```

This time, the /n entries are removed in favour of new lines and readable text.



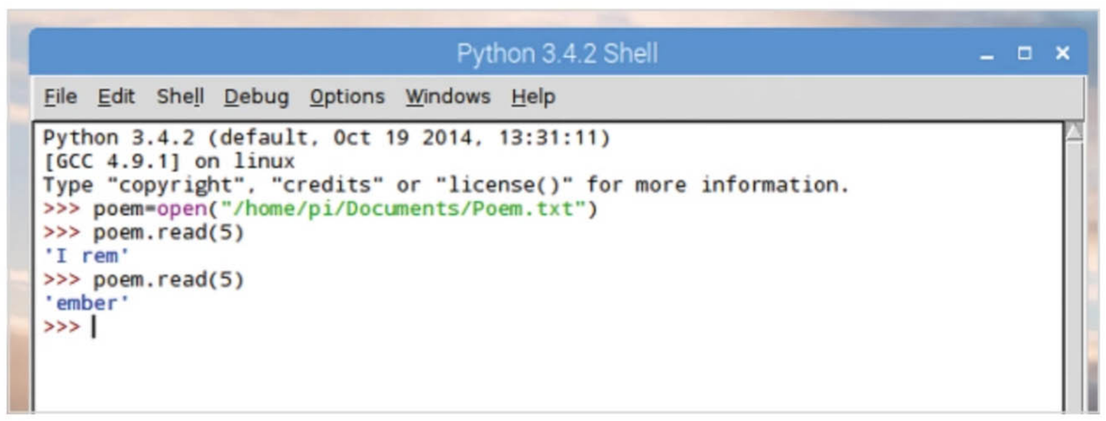
**STEP 5** Just as with lists, tuples, dictionaries and so on, you're able to index individual characters of the text. For example:

```
poem.read(5)
```

Displays the first five characters, whilst again entering:

```
poem.read(5)
```

Will display the next five. Entering (1) will display one character at a time.



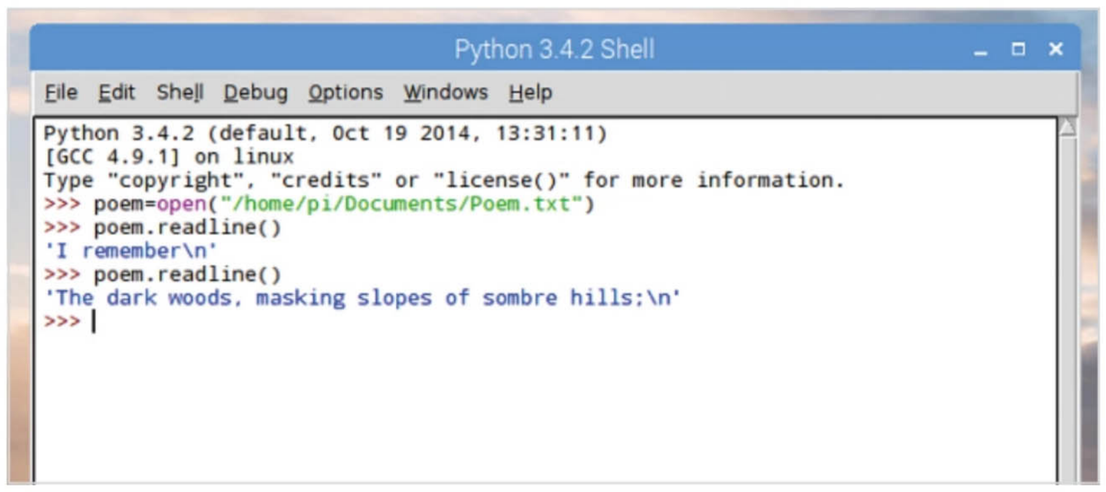

**STEP 6** Similarly, you can display one line of text at a time by using the readline() function. For example:

```
poem=open("/home/pi/Documents/Poem.txt")
poem.readline()
```

Will display the first line of the text with:

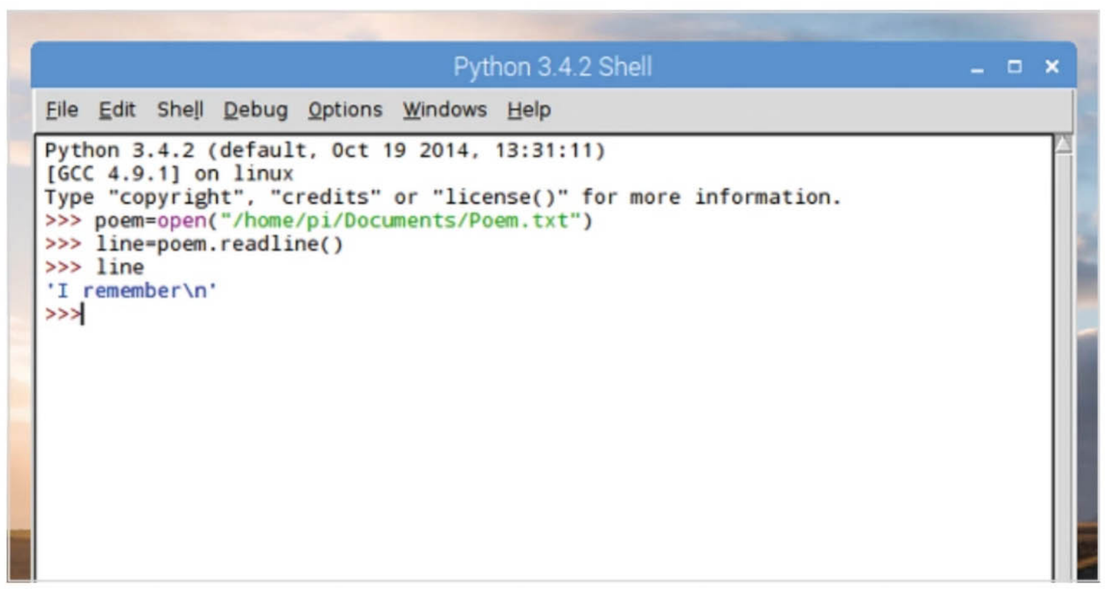
```
poem.readline()
```

Displaying the next line of text once more.




**STEP 7** You may have guessed that you can pass the readline() function into a variable, thus allowing you to call it again when needed:

```
poem=open("/home/pi/Documents/Poem.txt")
line=poem.readline()
line
```




**STEP 8** Extending this further, you can use readlines() to grab all the lines of the text and store them as multiple lists. These can then be stored as a variable:

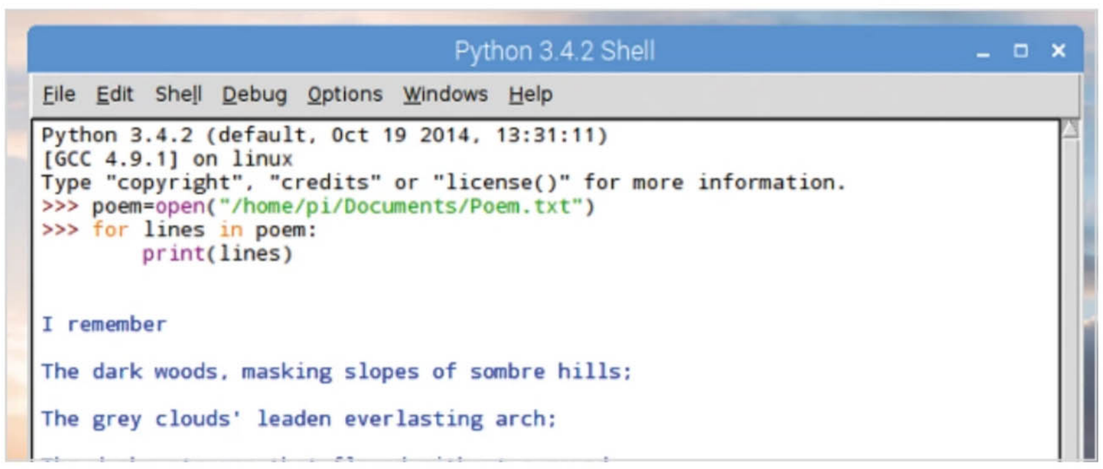
```
poem=open("/home/pi/Documents/Poem.txt")
lines=poem.readlines()
lines[0]
lines[1]
lines[2]
```

**STEP 9** You can also use the for statement to read the lines of text back to us:

```
for lines in lines:
   print(lines)
```

Since this is Python, there are other ways to produce the same output:

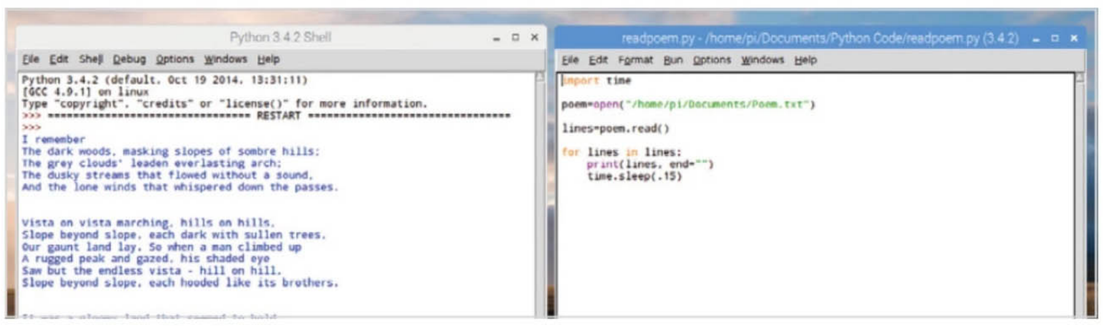
```
poem=open("/home/pi/Documents/Poem.txt")
for lines in poem:
   print(lines)
```




**STEP 10** Let's imagine that you want to print the text one character at a time, like an old dot matrix printer would. You can use the Time module mixed with what you've looked at here. Try this:

```
import time
poem=open("/home/pi/Documents/Poem.txt")
lines=poem.read()
for lines in lines:
  print(lines, end="")
  time.sleep(.15)
```

The output is fun to view, and easily incorporated into your own code.



# Writing to Files

The ability to read external files within Python is certainly handy but writing to a file is better still. Using the write() function, you're able to output the results of a program to a file, that you can then read() back into Python.

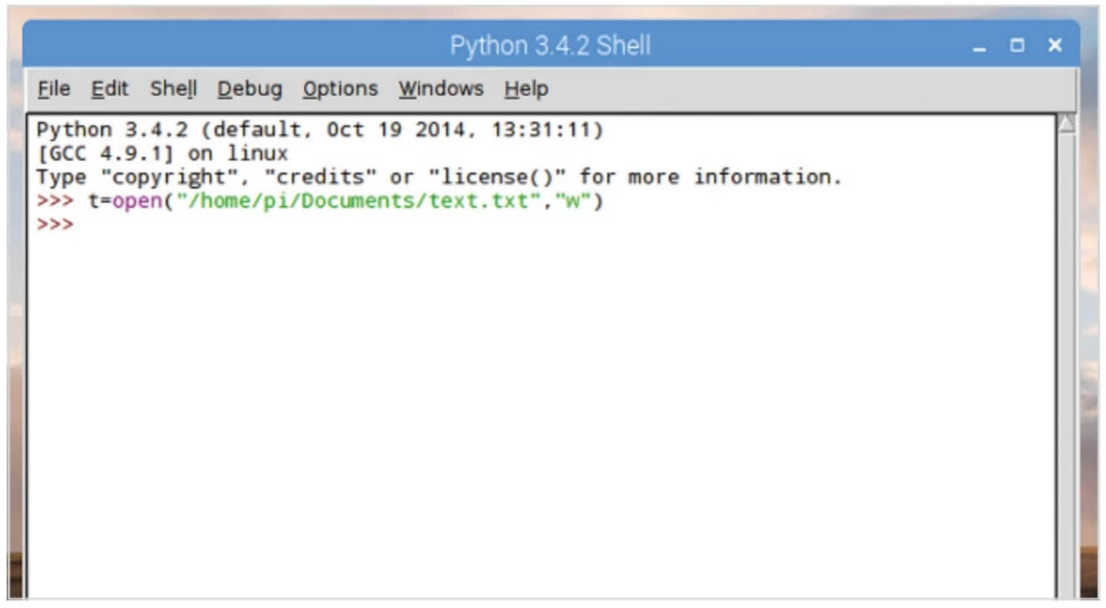
## WRITE AND CLOSE

**The write() function is slightly more complex than read(). Along with the filename you must also include an access mode which determines whether the file in question is in read or write mode.**

**STEP 1** Start by opening IDLE and enter the following:

```
t=open("/home/pi/Documents/text.txt","w")
```

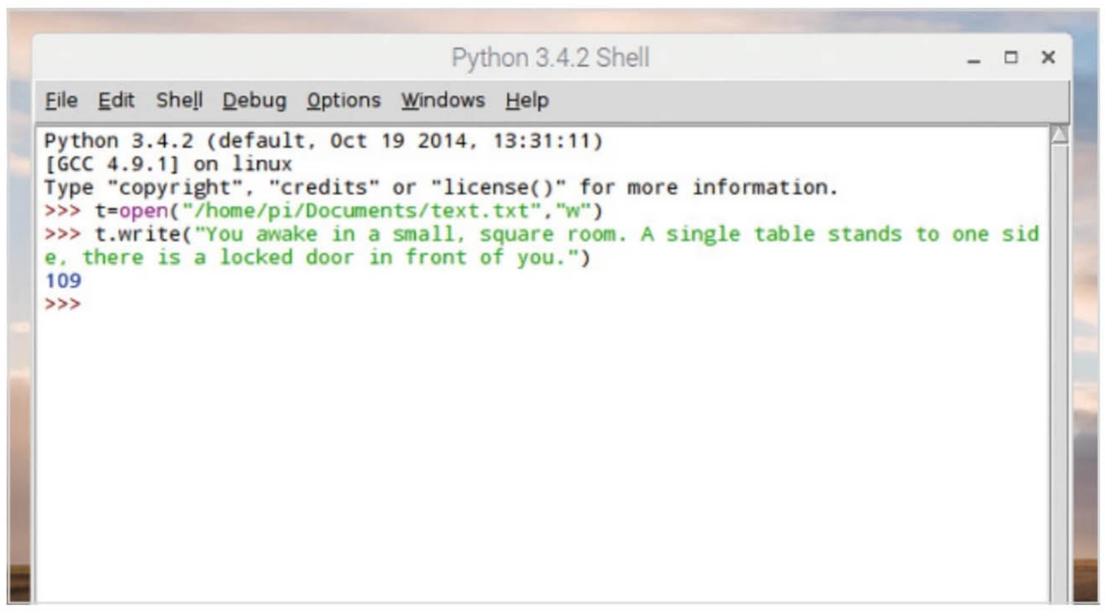
Change the destination from /home/pi/Documents to your own system. This code will create a text file called text.txt in write mode using the variable 't'. If there's no file of that name in the location, it will create one. If one already exits, it will overwrite it, so be careful.




**STEP 2** You can now write to the text file using the write() function. This works opposite to read(), writing lines instead of reading them. Try this:

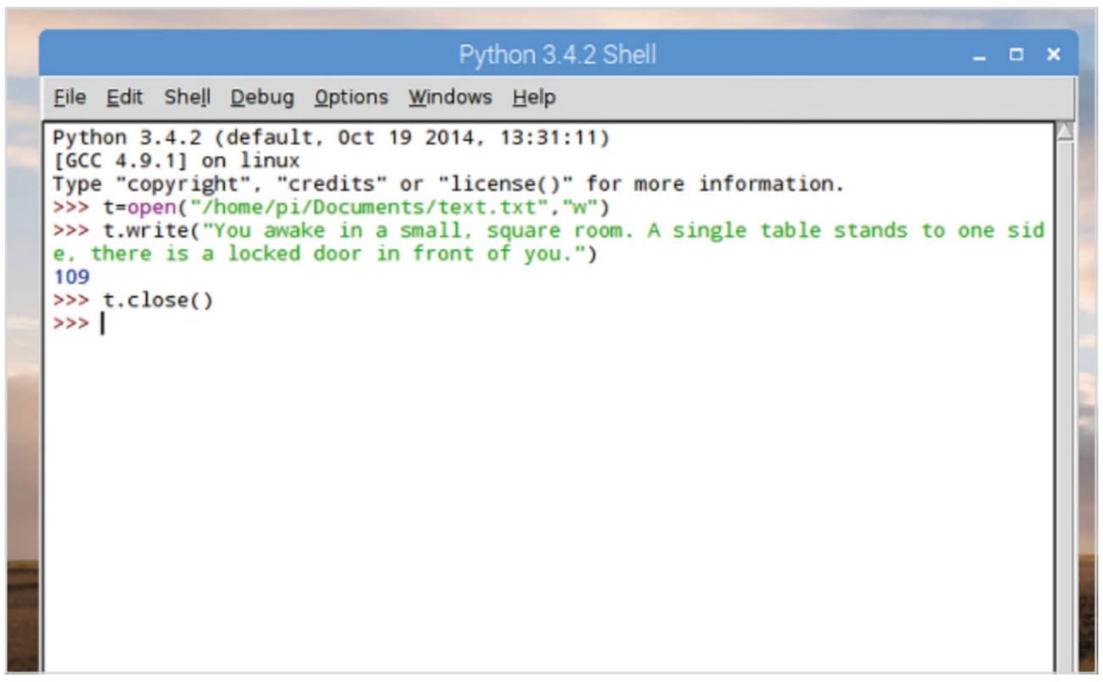
```
t.write("You awake in a small, square room. A single table stands to one side, there is a locked door in front of you.")
```

Note the 109. It's the number of characters you've entered.

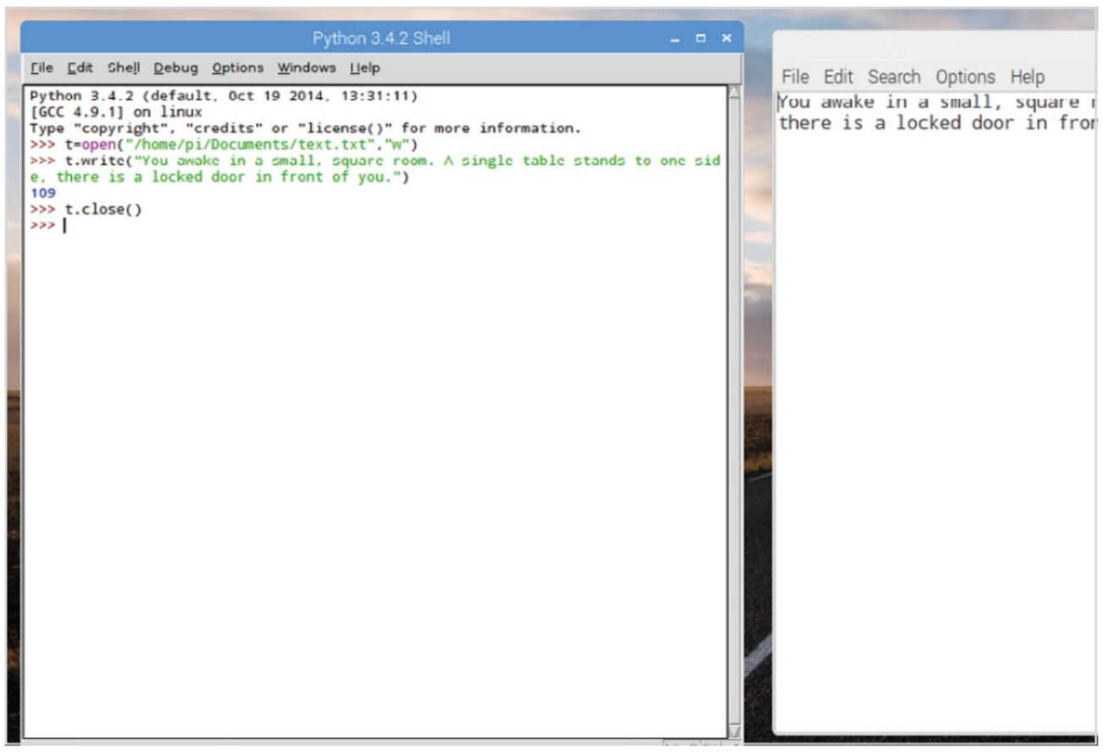



**STEP 3** However, the actual text file is still blank (you can check by opening it up). This is because you've written the line of text to the file object but not committed it to the file itself. Part of the write() function is that you need to commit the changes to the file; you can do this by entering:

```
t.close()
```

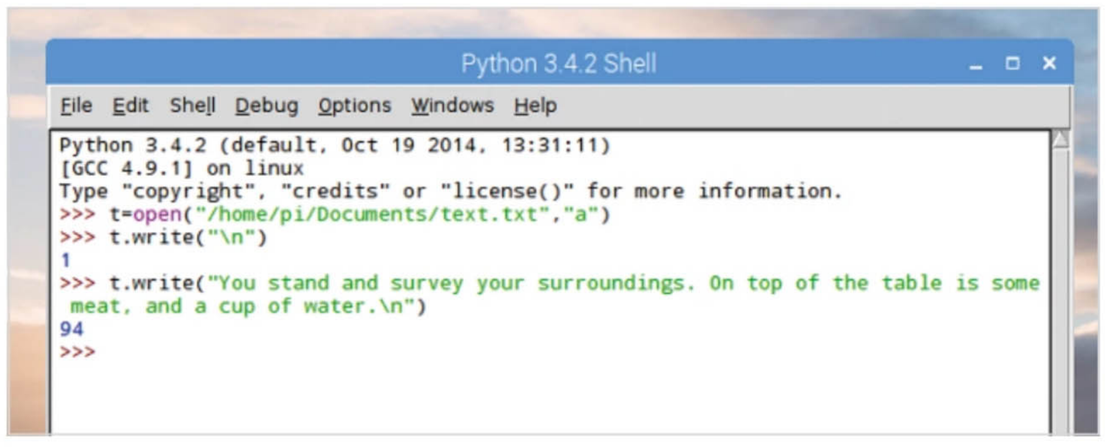



**STEP 4** If you now open the text file with a text editor, you can see that the line you created has been written to the file. This gives us the foundation for some interesting possibilities: perhaps the creation of your own log file or even the beginning of an adventure game.



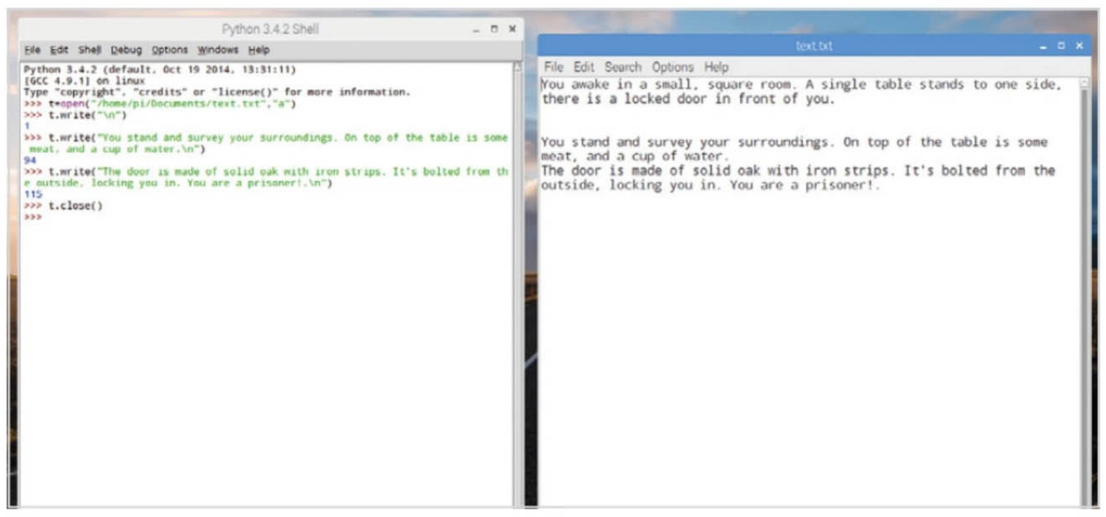
**STEP 5** To expand this code, you can reopen the file using 'a', for access or append mode. This will add any text at the end of the original line instead of wiping the file and creating a new one. For example:

```
t=open("/home/pi/Documents/text.txt","a")
t.write("\n")
t.write(" You stand and survey your surroundings.
On top of the table is some meat, and a cup of
water.\n")
```

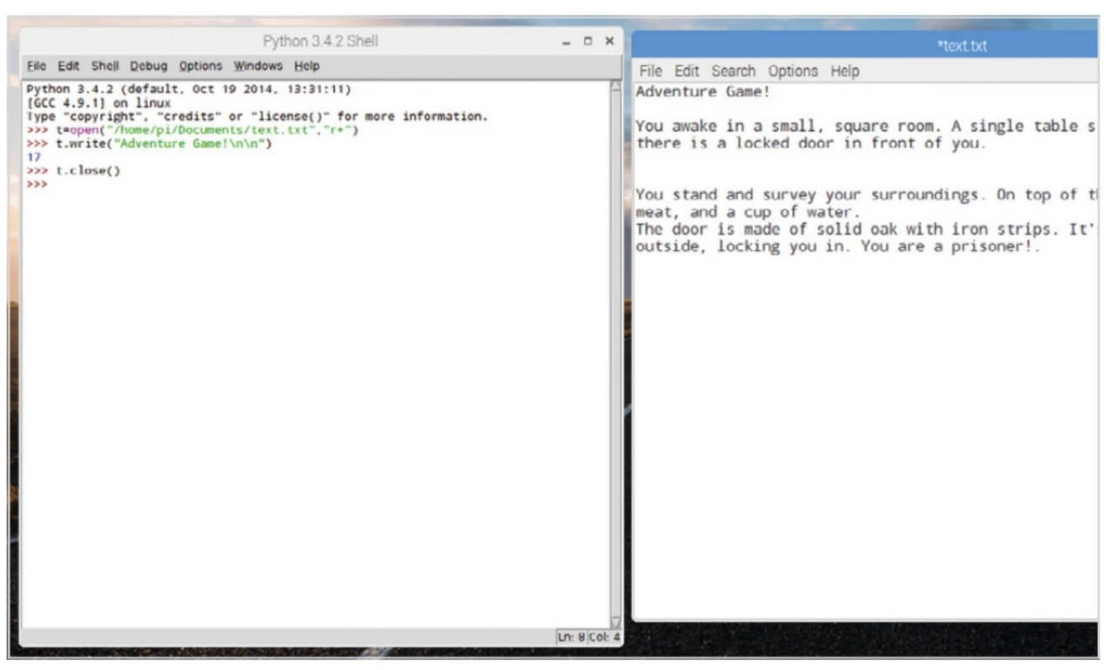



**STEP 6** You can keep extending the text line by line, ending each with a new line (\n). When you're done, finish the code with t.close() and open the file in a text editor to see the results:

```
t.write("The door is made of solid oak with iron
strips. It's bolted from the outside, locking you
in. You are a prisoner!.\n")
t.close()
```

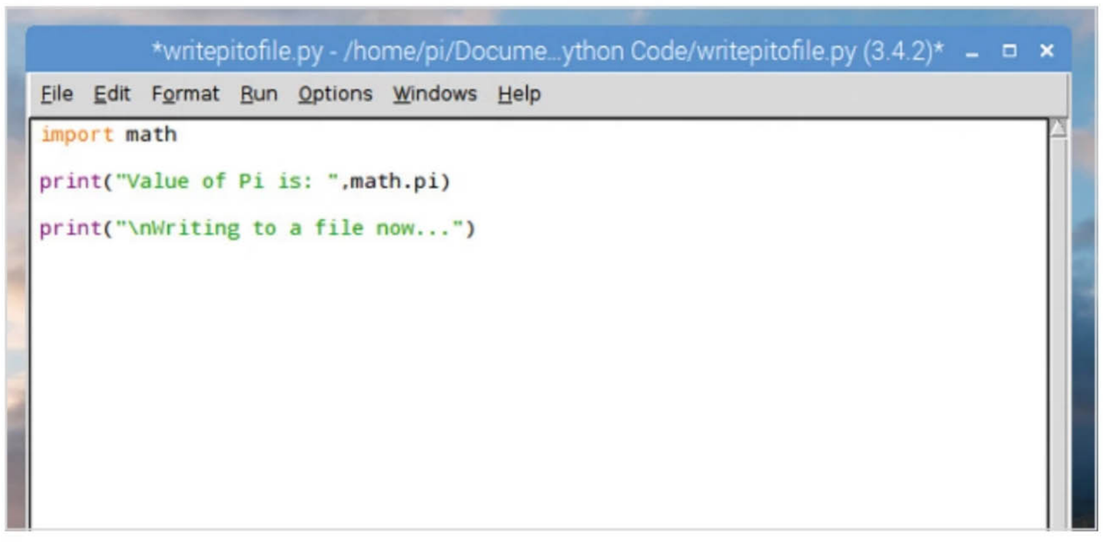



**STEP 7** There are various types of file access to consider using the open() function. Each depends on how the file is accessed and even the position of the cursor. For example, r+ opens a file in read and write and places the cursor at the start of the file.




**STEP 8** You can pass variables to a file that you've created in Python. Perhaps you want the value of Pi to be written to a file. You can call Pi from the Math module, create a new file and pass the output of Pi into the new file:

```
import math
print("Value of Pi is: ",math.pi)
print("\nWriting to a file now…")
```



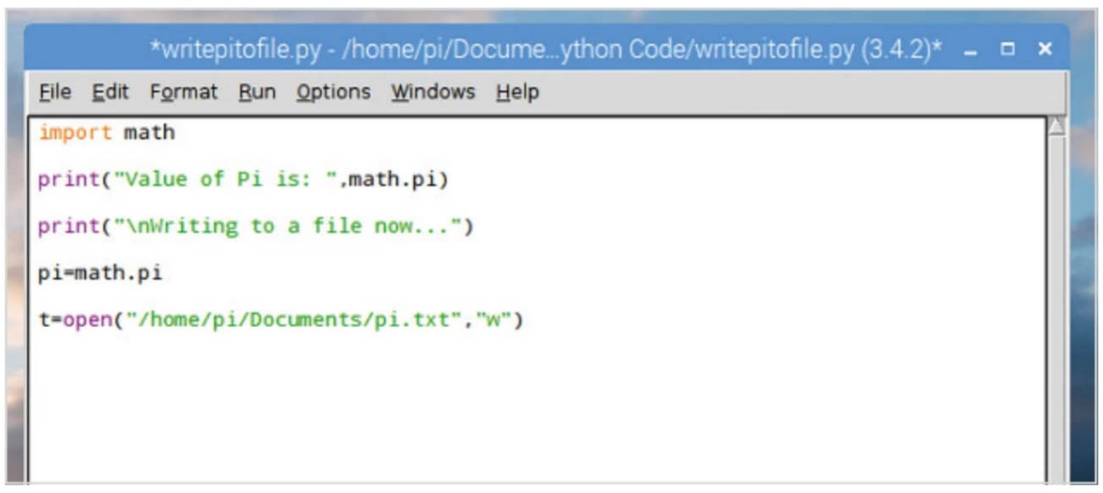

**STEP 9** Now let's create a variable called pi and assign it the value of Pi:

```
pi=math.pi
```

You also need to create a new file in which to write Pi to:

```
t=open("/home/pi/Documents/pi.txt","w")
```

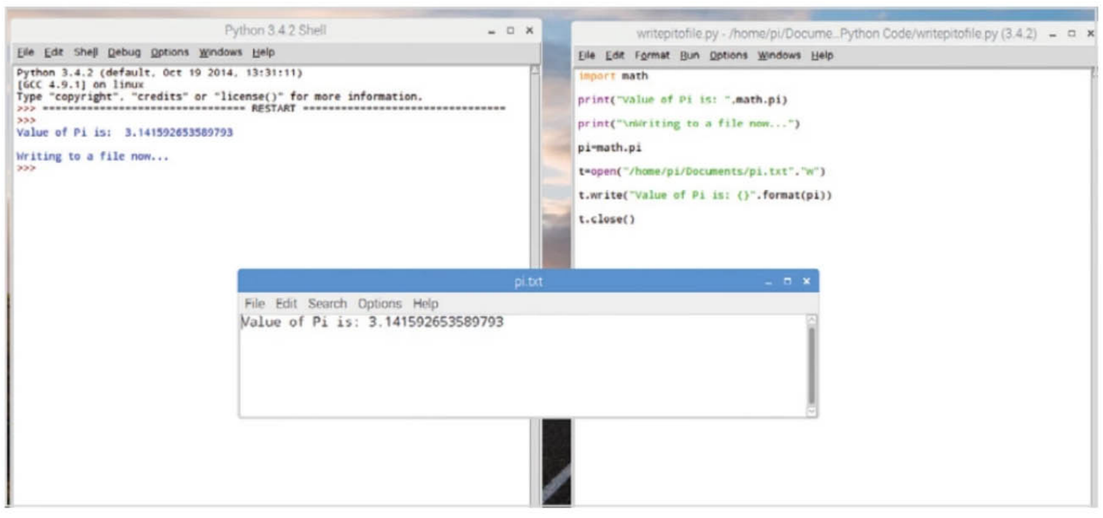
Remember to change your file location to your own particular system setup.




**STEP 10** To finish, you can use string formatting to call the variable and write it to the file, then commit the changes and close the file:

```
t.write("Value of Pi is: {}".format(pi))
t.close()
```

You can see from the results that you're able to pass any variable to a file.



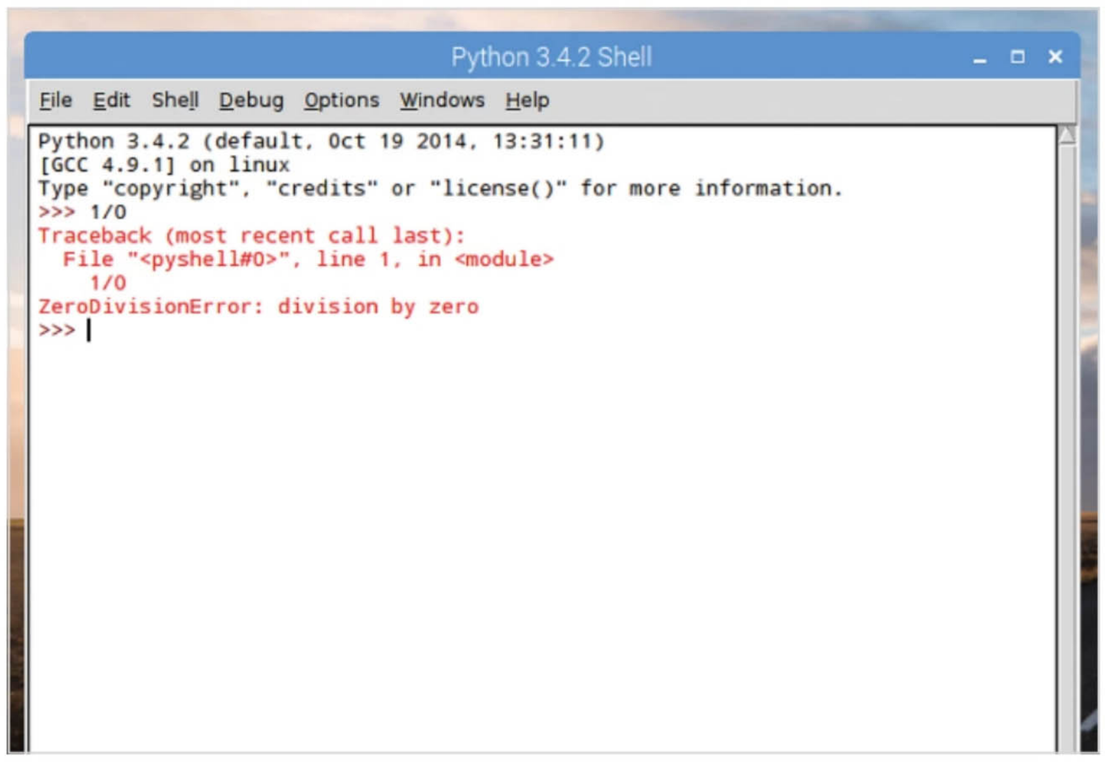
# Exceptions

When coding, you'll naturally come across some issues that are out of your control. Let's assume you ask a user to divide two numbers and they try to divide by zero. This will create an error and break your code.

## EXCEPTIONAL OBJECTS

**Rather than stop the flow of your code, Python includes exception objects which handle unexpected errors in the code. You can combat errors by creating conditions where exceptions may occur.**

**STEP 1** You can create an exception error by simply trying to divide a number by zero. This will report back with the ZeroDivisionError: Division by zero message, as seen in the screenshot. The ZeroDivisionError part is the exception class, of which there are many.

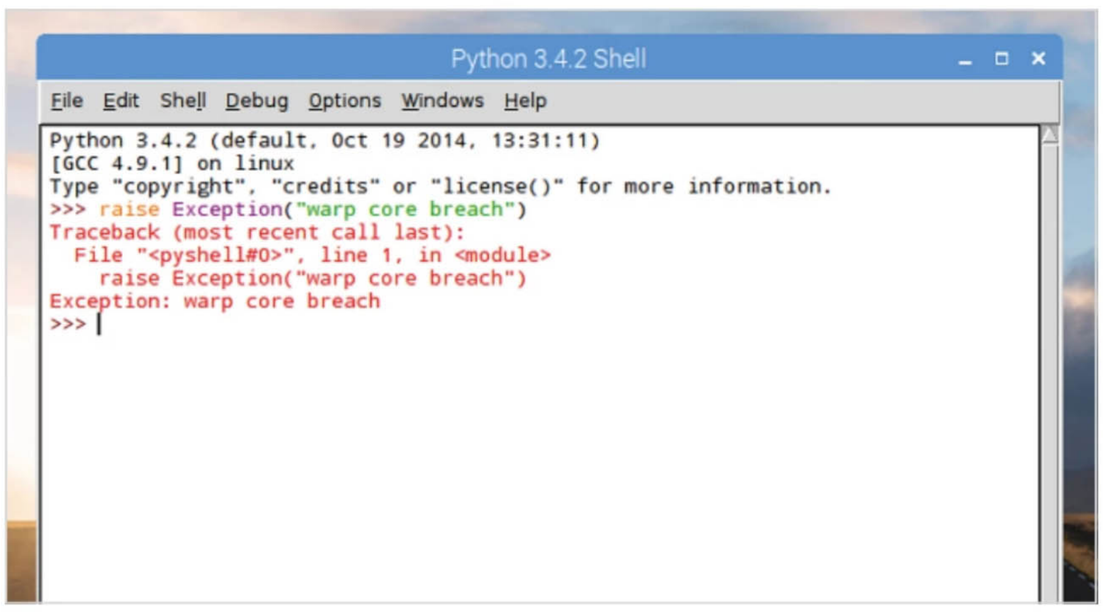



**STEP 2** Most exceptions are raised automatically when Python comes across something that's inherently wrong with the code. However, you can create your own exceptions that are designed to contain the potential error and react to it, as opposed to letting the code fail.

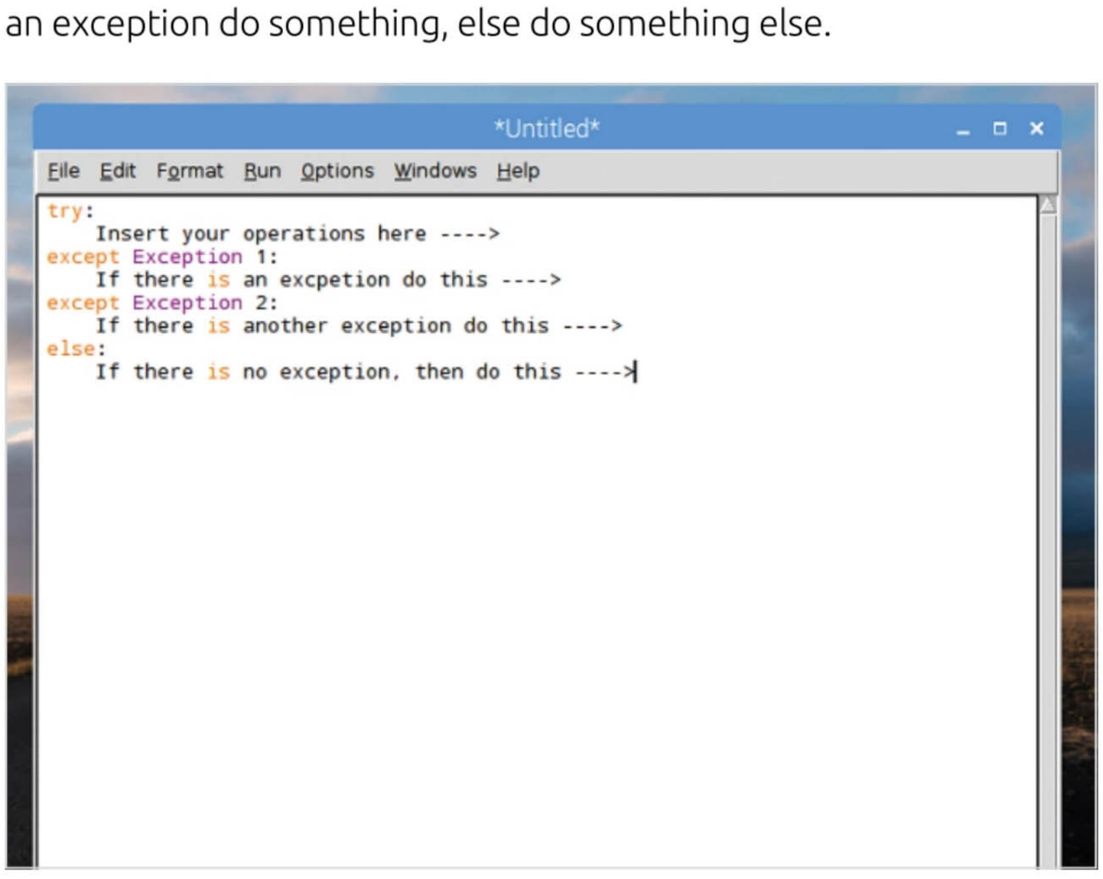
```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      |    +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      |    +-- RecursionError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
           +-- ResourceWarning
```

**STEP 3** You can use the functions raise exception to create our own error handling code within Python. Let's assume your code has you warping around the cosmos, too much however results in a warp core breach. To stop the game from exiting due to the warp core going supernova, you can create a custom exception:

```
raise Exception("warp core breach")
```

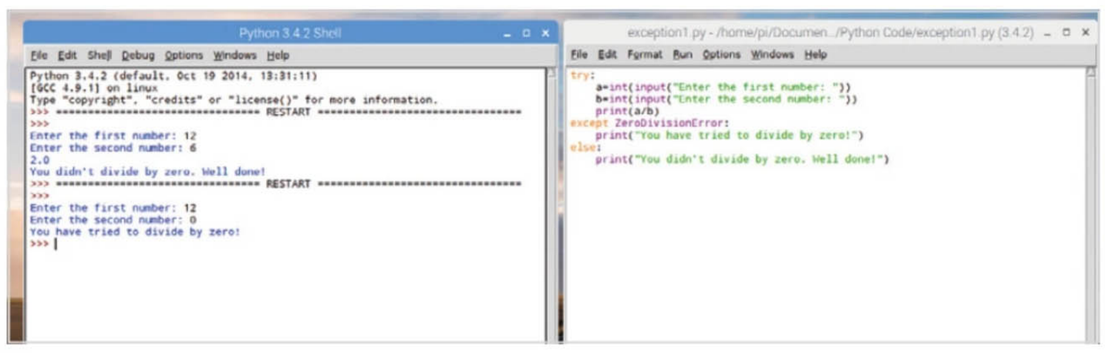



**STEP 4** To trap any errors in the code you can encase the potential error within a try: block. This block consists of try, except, else, where the code is held within try:, then if there's an exception do something, else do something else.



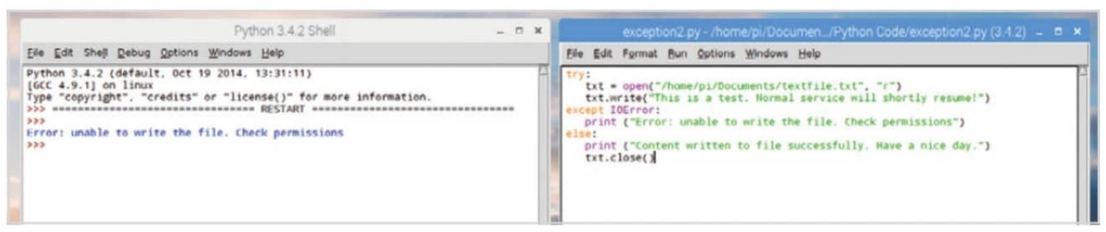
**STEP 5** For example, use the divide by zero error. You can create an exception where the code can handle the error without Python quitting due to the problem:

```
try:
  a=int(input("Enter the first number: "))
  b=int(input("Enter the second number: "))
  print(a/b)
except ZeroDivisionError:
  print("You have tried to divide by zero!")
else:
  print("You didn't divide by zero. Well done!")
```

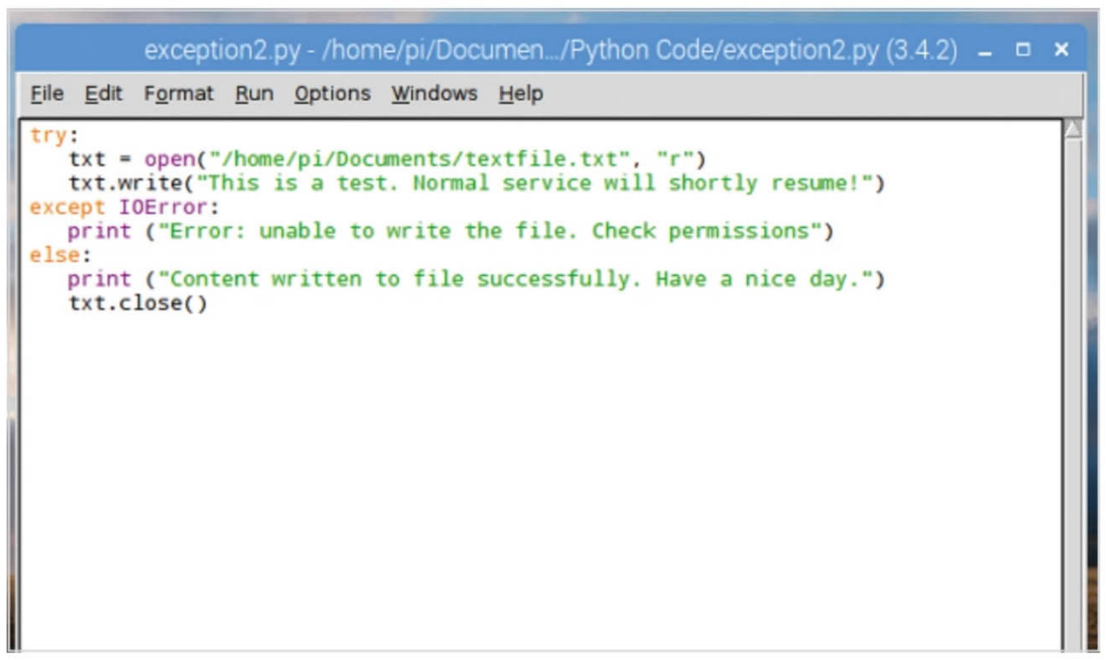



**STEP 6** You can use exceptions to handle a variety of useful tasks. Using an example from our previous tutorials, let's assume you want to open a file and write to it:

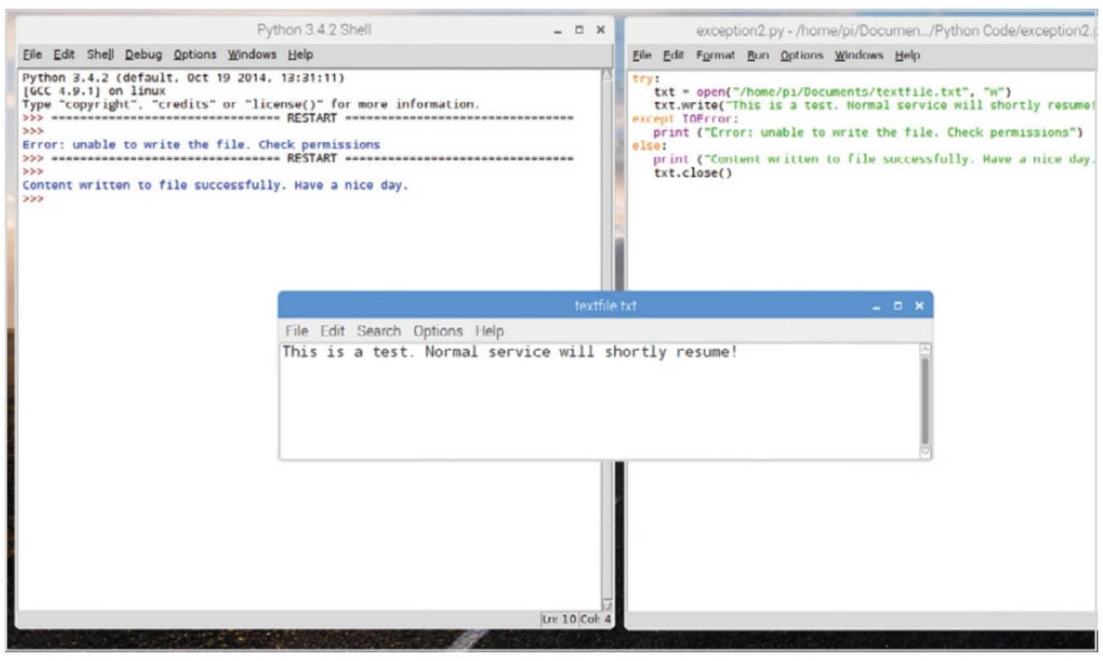
```
try:
  txt = open("/home/pi/Documents/textfile.txt", "r")
  txt.write("This is a test. Normal service will
shortly resume!")
except IOError:
  print ("Error: unable to write the file. Check
permissions")
else:
  print ("Content written to file successfully. Have
a nice day.")
  txt.close()
```




**STEP 7** Obviously this won't work due to the file textfile.txt being opened as read only (the "r" part). So in this case rather than Python telling you that you're doing something wrong, you've created an exception using the IOError class informing the user that the permissions are incorrect.

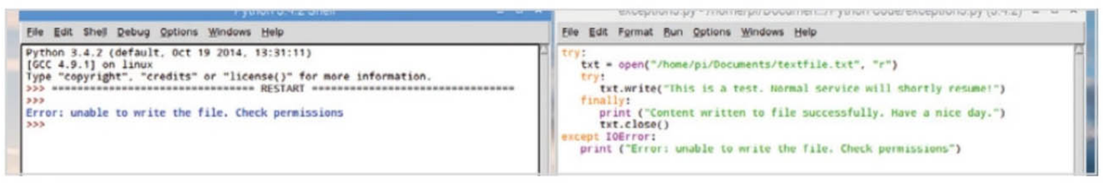



**STEP 8** Naturally, you can quickly fix the issue by changing the "r" read only instance with a "w" for write. This, as you already know, will create the file and write the content then commit the changes to the file. The end result will report a different set of circumstances, in this case, a successful execution of the code.

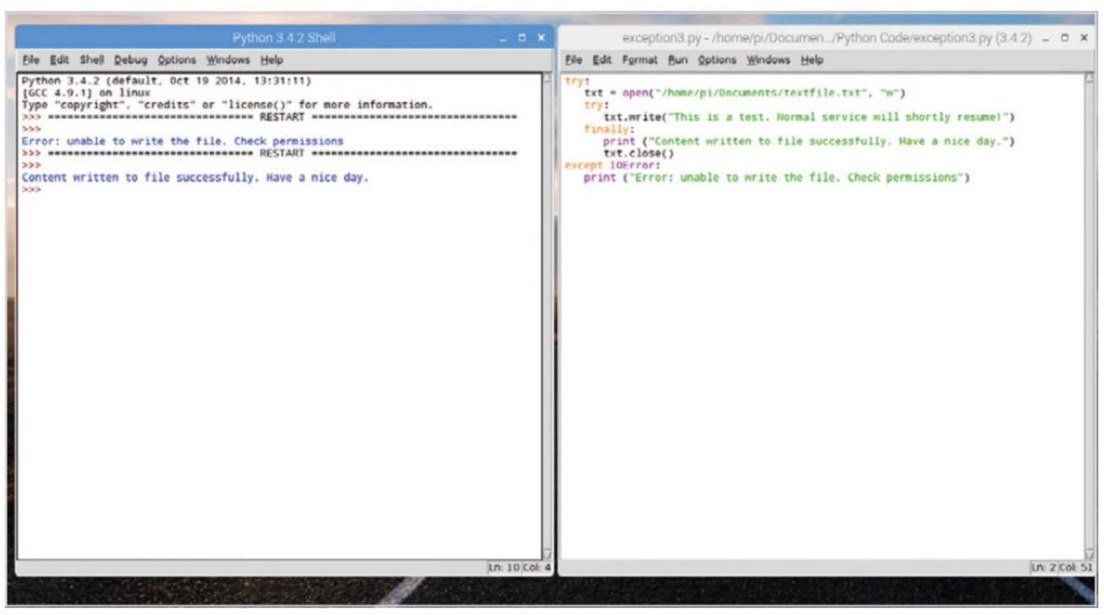



**STEP 9** You can also use a finally: block, which works in a similar fashion but you can't use else with it. To use our example from Step 6:

```
try:
  txt = open("/home/pi/Documents/textfile.txt", "r")
  try:
    txt.write("This is a test. Normal service will
shortly resume!")
  finally:
    print ("Content written to file successfully.
Have a nice day.")
    txt.close()
except IOError:
  print ("Error: unable to write the file. Check
permissions")
```




**STEP 10** As before an error will occur as you've used the "r" read-only permission. If you change it to a "w", then the code will execute without the error being displayed in the IDLE Shell. Needless to say, it can be a tricky getting the exception code right the first time. Practise though, and you will get the hang of it.



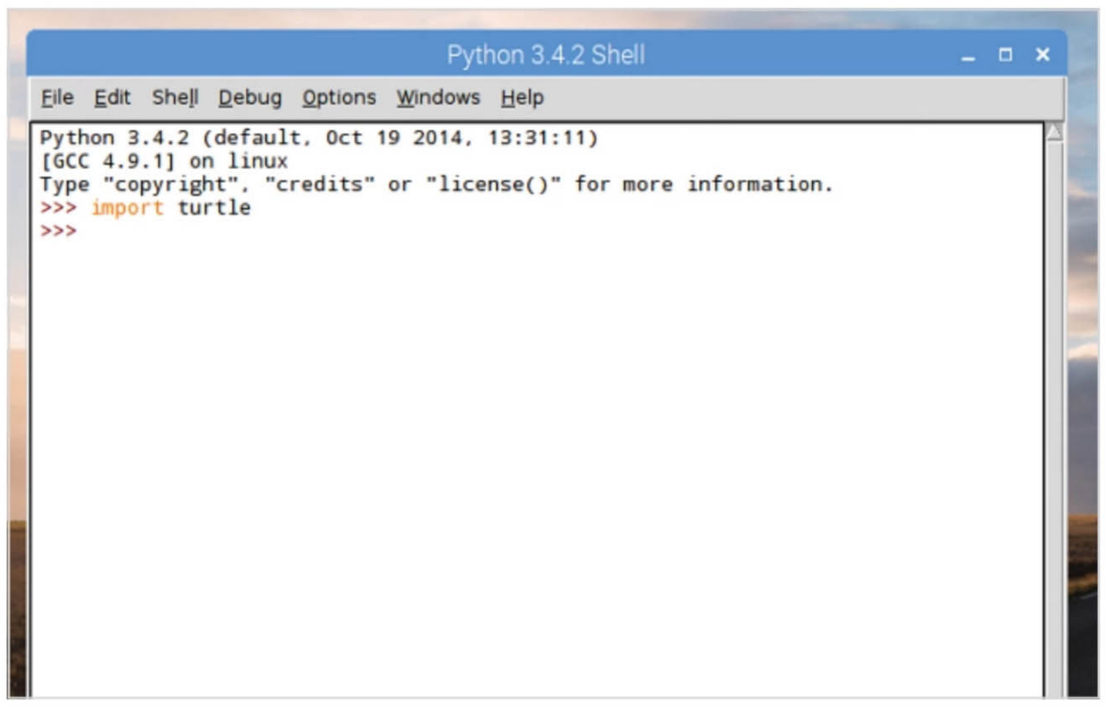
# Python Graphics

While dealing with text on the screen, either as a game or in a program, is great, there will come a time when a bit of graphical representation wouldn't go amiss. Python 3 has numerous ways in which to include graphics and they're surprisingly powerful too.

## GOING GRAPHICAL

**You can draw simple graphics, lines, squares and so on, or you can use one of the many Python modules available, to bring out some spectacular effects.**

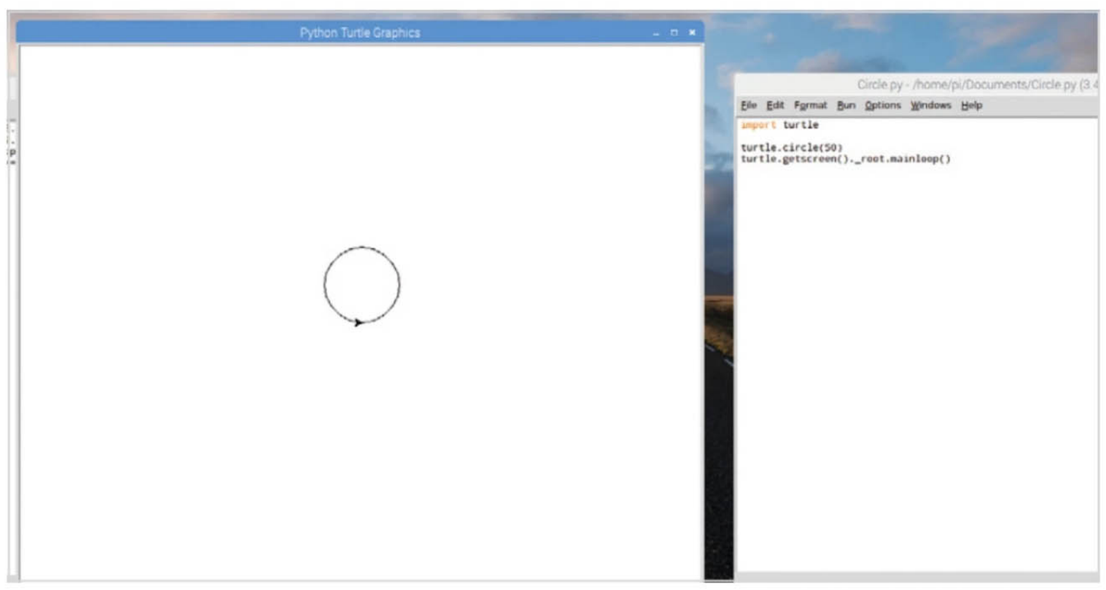
**STEP 1** One of the best graphical modules to begin learning Python graphics is Turtle. The Turtle module is, as the name suggests, based on the turtle robots used in many schools, that can be programmed to draw something on a large piece of paper on the floor. The Turtle module can be imported with: `import turtle`.



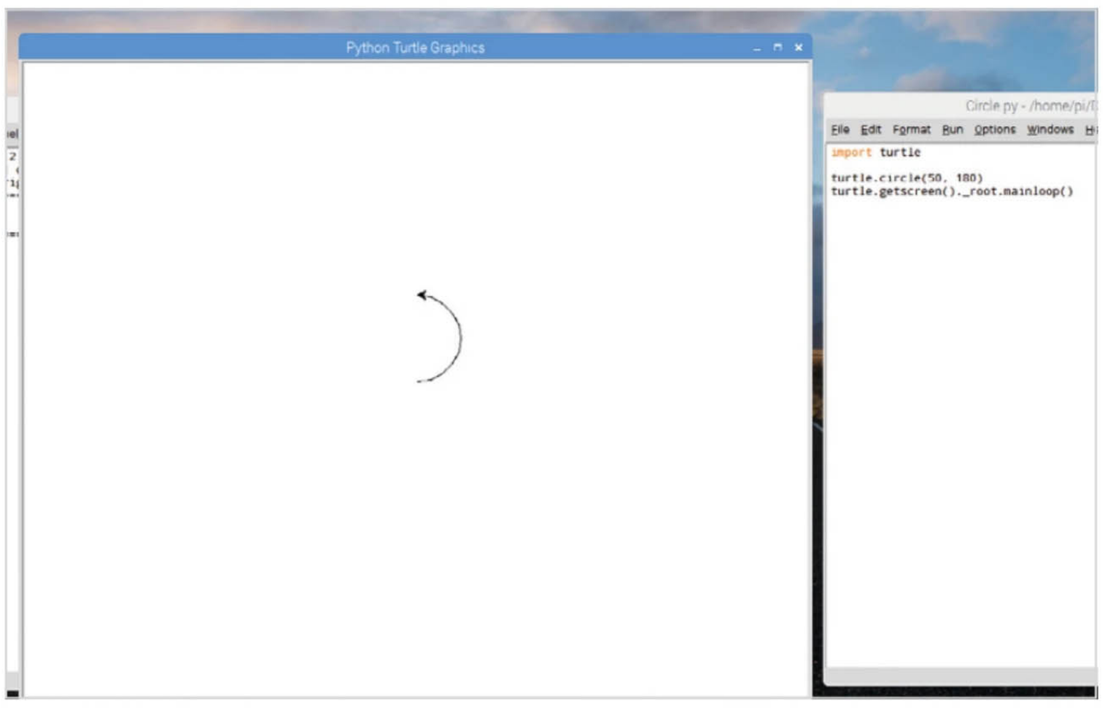

**STEP 2** Let's begin by drawing a simple circle. Start a New File, then enter the following code:

```
import turtle

turtle.circle(50)
turtle.getscreen()._root.mainloop()
```

As usual press F5 to save the code and execute it. A new window will now open up and the 'Turtle' will draw a circle.



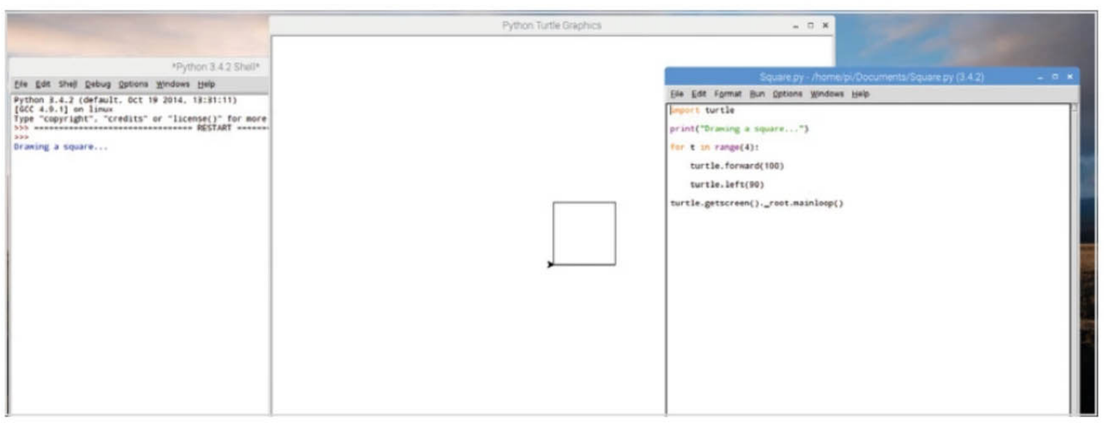

**STEP 3** The command turtle.circle(50) is what draws the circle on the screen, with 50 being the size. You can play around with the sizes if you like, going up to 100, 150 and beyond; you can draw an arc by entering: `turtle.circle(50, 180)`, where the size is 50, but you're telling Python to only draw 180° of the circle.




**STEP 4** The last part of the circle code tells Python to keep the window where the drawing is taking place to remain open, so the user can click to close it. Now, let's make a square:

```
import turtle

print("Drawing a square...")
for t in range(4):
    turtle.forward(100)
    turtle.left(90)
turtle.getscreen()._root.mainloop()
```

You can see that we've inserted a loop to draw the sides of the square.



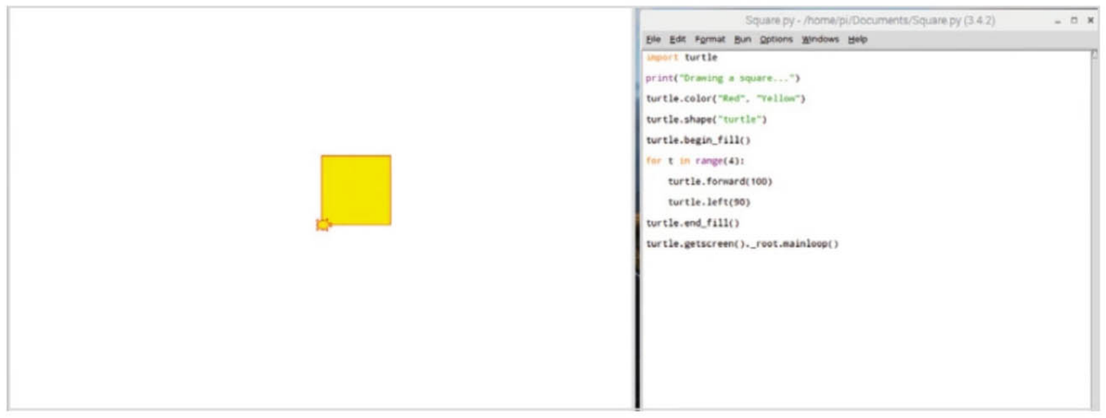
**STEP 5** You can add a new line to the square code to add some colour:

```
turtle.color("Red")
```

Then you can even change the character to an actual turtle by entering:

```
turtle.shape("turtle")
```

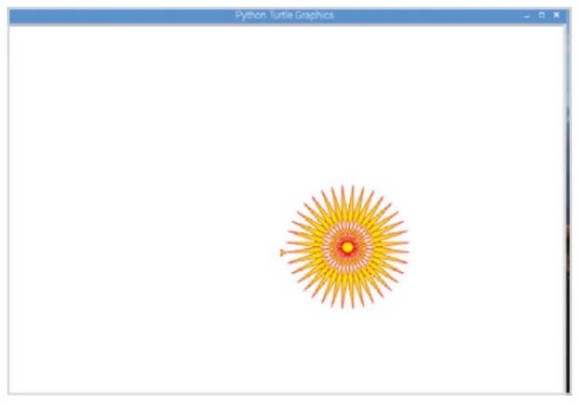
You can also use the command `turtle.begin_fill()`, and `turtle.end_fill()` to fill in the square with the chosen colours; red outline, yellow fill in this case.

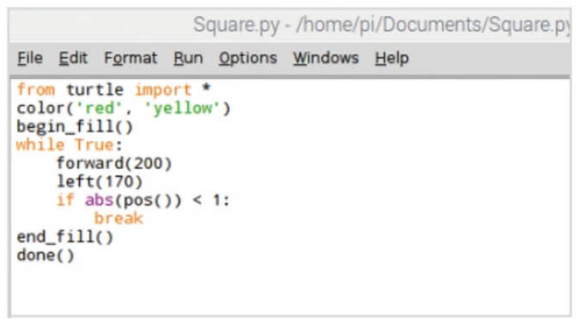



**STEP 6** You can see that the Turtle module can draw out some pretty good shapes and become a little more complex as you begin to master the way it works. Enter this example:

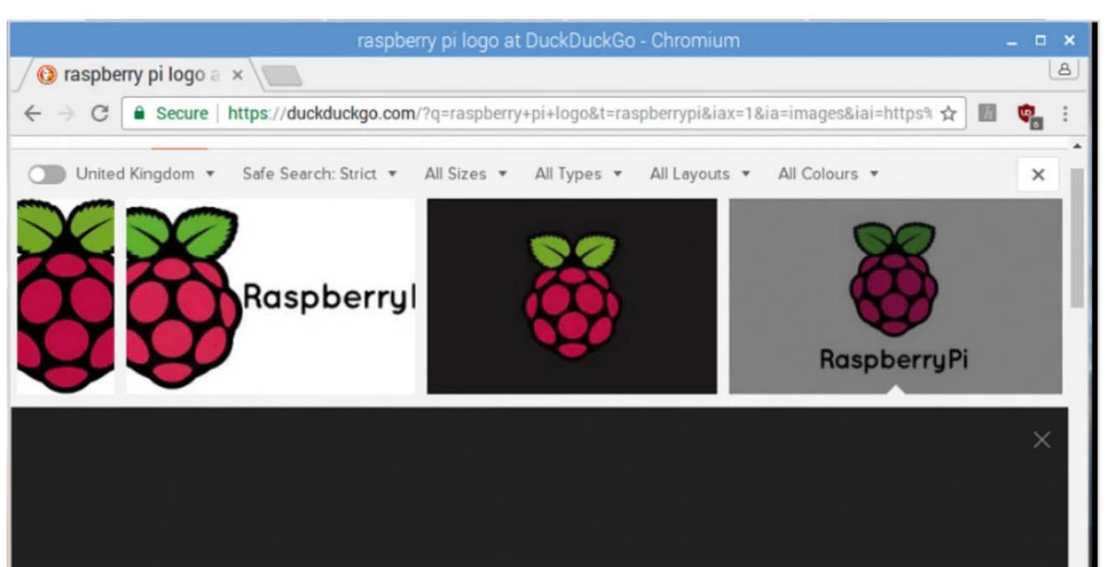
```
from turtle import *
color('red', 'yellow')
begin_fill()
while True:
  forward(200)
  left(170)
  if abs(pos()) < 1:
    break
end_fill()
done()
```

It's a different method, but very effective.




**STEP 7** Another way in which you can display graphics is by using the Pygame module. There are numerous ways in which pygame can help you output graphics to the screen but for now let's look at displaying a predefined image. Start by opening a browser and finding an image, then save it to the folder where you save your Python code.



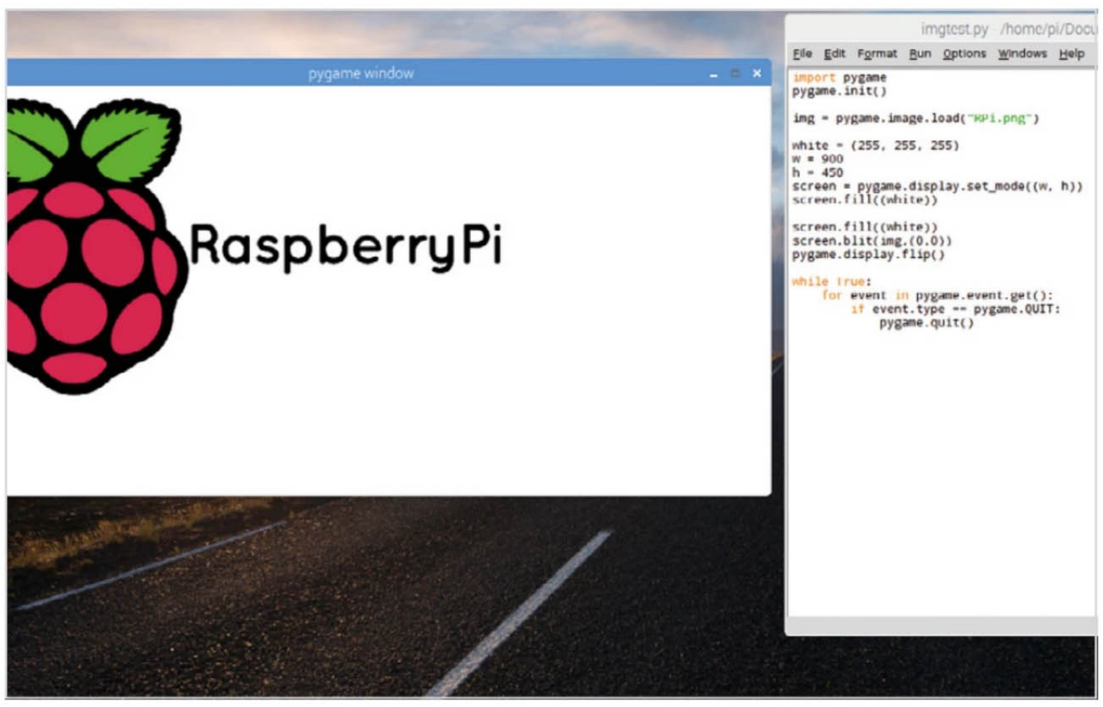

**STEP 8** Now let's get the code by importing the Pygame module:

```
import pygame
pygame.init()

img = pygame.image.load("RPi.png")

white = (255, 255, 255)
w = 900
h = 450
screen = pygame.display.
set_mode((w, h))
screen.fill((white))

screen.fill((white))
screen.blit(img,(0,0))
pygame.display.flip()

while True:
  for event in pygame.event.get():
    if event.type == pygame.QUIT:
      pygame.quit()
```

**STEP 9** In the previous step you imported pygame, initiated the pygame engine and asked it to import our saved Raspberry Pi logo image, saved as RPi.png. Next you defined the background colour of the window to display the image and the window size as per the actual image dimensions. Finally you have a loop to close the window.

```
w = 900
h = 450
screen = pygame.display.set_mode((w, h))
screen.fill((white))

screen.fill((white))
screen.blit(img,(0,0))
pygame.display.flip()

while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
```

**STEP 10** Press F5 to save and execute the code and your image will be displayed in a new window. Have a play around with the colours, sizes and so on and take time to look up the many functions within the Pygame module too.



"First, solve the problem.
Then, write the code."
– John Johnson (Developer and author)

# Using Modules

A Python module is simply a Python-created source file which contains the necessary and unique code for custom classes, functions and global variables. You can bind and reference modules to extend functionality and create even more spectacular Python programs.

Want to see how to better use these modules to add a little something extra to your code? Then read on and learn how they can be used to fashion fantastic code. There's even some basic animations in this section, to help you get more from the Python modules.

# Calendar Module

Beyond the Time module, the Calendar module can produce some interesting results when executed within your code. It does far more than simply display the date in the Time module-like format; you can actually call up a wall calendar-type display.

## WORKING WITH DATES

**The Calendar module is built into Python 3. However, if for some reason it's not installed, you can add it using `pip install calendar` as a Windows administrator, or `sudo pip install calendar` for Linux and macOS.**

**STEP 1** Launch Python 3 and enter: `import calendar` to call up the module and its inherent functions. Once it is loaded into memory, start by entering:

```
sep=calendar.TextCalendar(calendar.SUNDAY)
sep.prmonth(2019, 9)
```

*Python 3.5.3 Shell*

File Edit Shell Debug Options Window Help

```
Python 3.5.3 (default, Sep 27 2018, 17:25:39)
[GCC 6.3.0 20170516] on linux
Type "copyright", "credits" or "license()" for more information.
>>> import calendar
>>> sep=calendar.TextCalendar(calendar.SUNDAY)
>>> sep.prmonth(2019, 9)
   September 2019
Su Mo Tu We Th Fr Sa
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30
>>>
>>>
```

**STEP 2** You can see that the days of September 2019 are displayed in a wall calendar fashion. Naturally you can change the 2019, 9 part of the second line to any year and month you want, a birthday for example (1973, 6). The first line configures TextCalendar to start its weeks on a Sunday; you can opt for Monday if you prefer.

**STEP 3** There are numerous functions within the Calendar module that may be of interest to you when forming your own code. For example, you can display the number of leap years between two specific years:

```
leaps=calendar.leapdays(1900, 2019)
print(leaps)
```

The result is 29, starting from 1904 onward.

Python 3.5.3 Shell

File Edit Shell Debug Options Window Help

```
Python 3.5.3 (default, Sep 27 2018, 17:25:39)
[GCC 6.3.0 20170516] on linux
Type "copyright", "credits" or "license()" for more information.
>>> import calendar
>>> leaps=calendar.leapdays(1900, 2019)
>>> print(leaps)
29
>>>
```

**STEP 4** You could even fashion that particular example into a piece of working, user interactive Python code:

```
import calendar
print(">>>>>>>>>>Leap Year
Calculator<<<<<<<<<<\n")
y1=int(input("Enter the first year: "))
y2=int(input("Enter the second year: "))
leaps=calendar.leapdays(y1, y2)
print("Number of leap years between", y1, "and",
y2, "is:", leaps)
```

Python 3.5.3 Shell

File Edit Shell Debug Options Window Help

```
Python 3.5.3 (default, Sep 27 2018, 17:25:39)
[GCC 6.3.0 20170516] on linux
Type "copyright", "credits" or "license()" for more information.
>>> import calendar
>>> sep=calendar.TextCalendar(calendar.SUNDAY)
>>> sep.prmonth(2019, 9)
   September 2019
Su Mo Tu We Th Fr Sa
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30
>>>
>>> birthday=calendar.TextCalendar(calendar.MONDAY)
>>> birthday.prmonth(1973, 6)
     June 1973
Mo Tu We Th Fr Sa Su
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30
>>>
```

Python 3.5.3 Shell

File Edit Shell Debug Options Window Help

```
Python 3.5.3 (default, Sep 27 2018, 17:25:39)
[GCC 6.3.0 20170516] on linux
Type "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: /home/pi/Documents/leaps.py ==========
>>>>>>>>>>Leap Year Calculator<<<<<<<<<<

Enter the first year: 1756
Enter the second year: 2022
Number of leap years between 1756 and 2022 is: 65
>>>
```

leaps.py - /home/pi/Documents/leaps.py (3.5.3)

File Edit Format Run Options Window Help

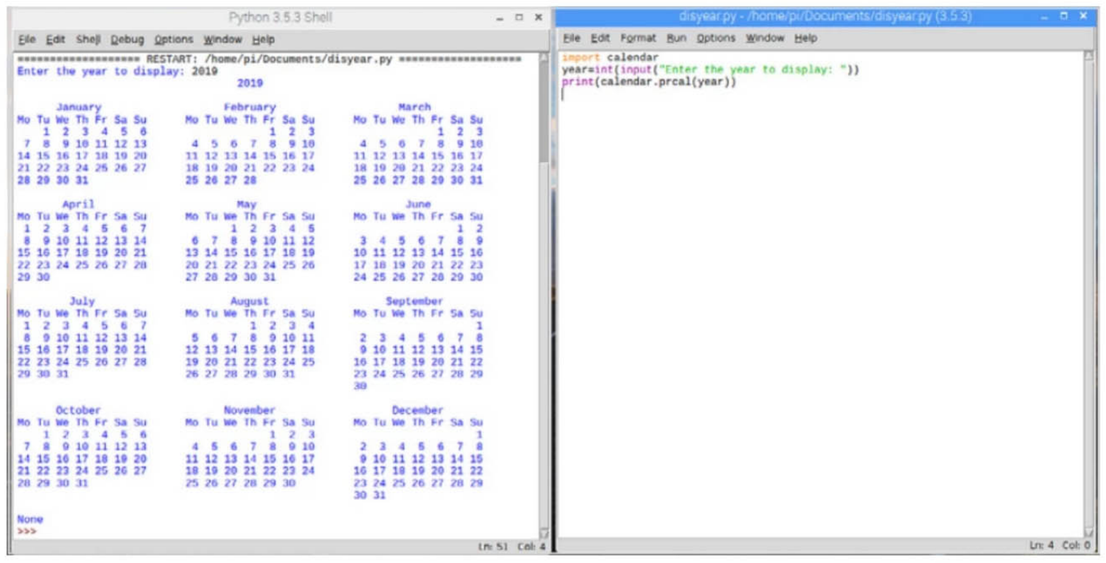
```
import calendar
print(">>>>>>>>>>Leap Year Calculator<<<<<<<<<<\n")
y1=int(input("Enter the first year: "))
y2=int(input("Enter the second year: "))
leaps=calendar.leapdays(y1, y2)
print("Number of leap years between", y1, "and", y2, "is:", leaps)
```

## STEP 5

You can also create a program that will display all the days, weeks and months, within a given year:

```
import calendar
year=int(input("Enter the year to display: ")
print(calendar.prcal(year))
```

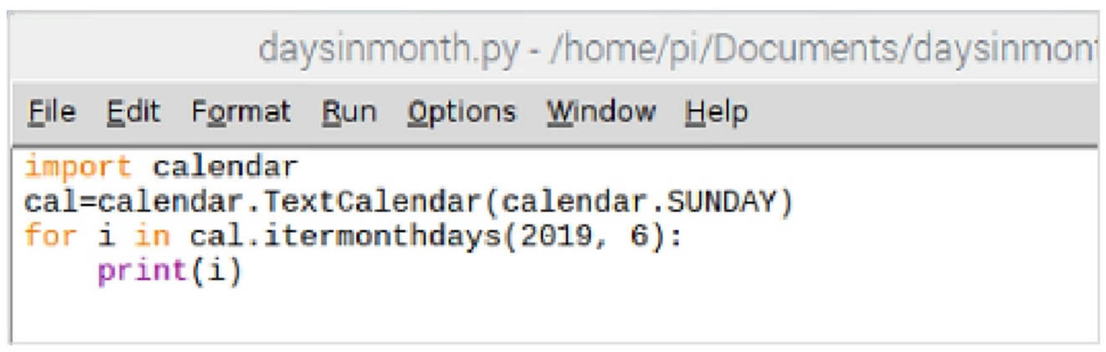
We're sure you'll agree that's quite a handy bit of code to have to hand.




## STEP 6

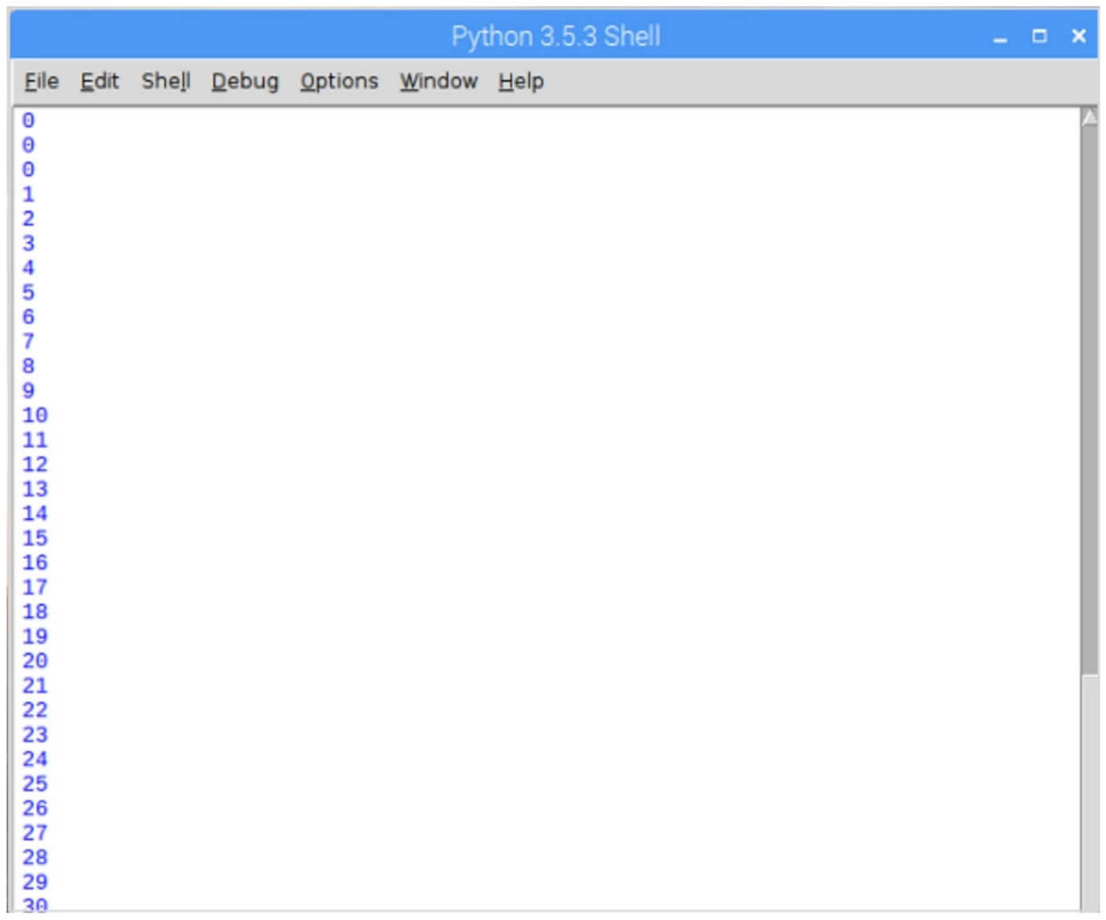
Interestingly we can also list the number of days in a month by using a simple for loop:

```
import calendar
cal=calendar.TextCalendar(calendar.SUNDAY)
for i in cal.itermonthdays(2019, 6):
        print(i)
```




## STEP 7

You can see that the code produced some zeros at the beginning; this is due to the starting day of the week, Sunday in this case, as well as overlapping days from the previous month. Therefore, the counting of the days will start on Saturday 1st June 2019 and will total 30, as the output correctly displays.

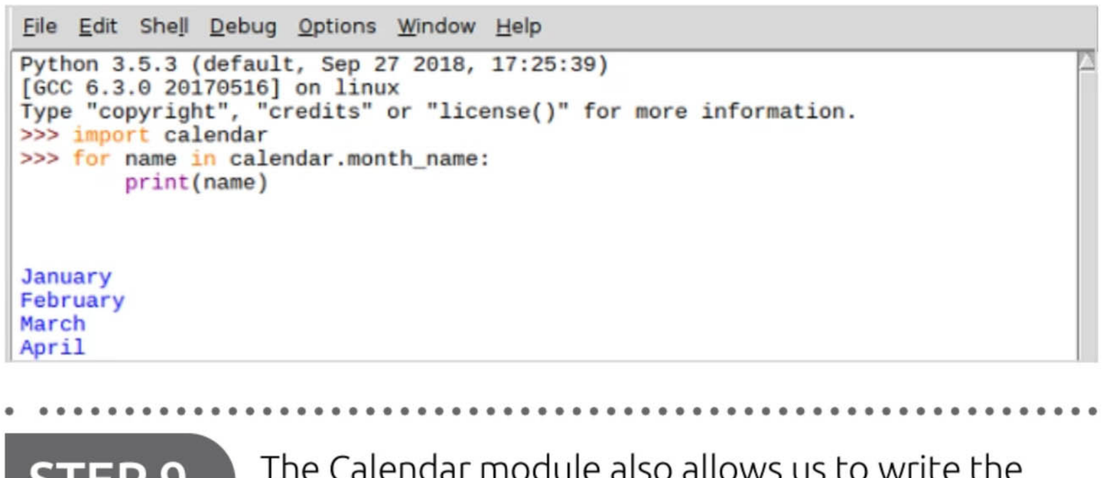



## STEP 8

You're also able to print the individual months or days of the week:

```
import calendar
for name in calendar.month_name:
        print(name)
```

```
import calendar
for name in calendar.day_name:
        print(name)
```

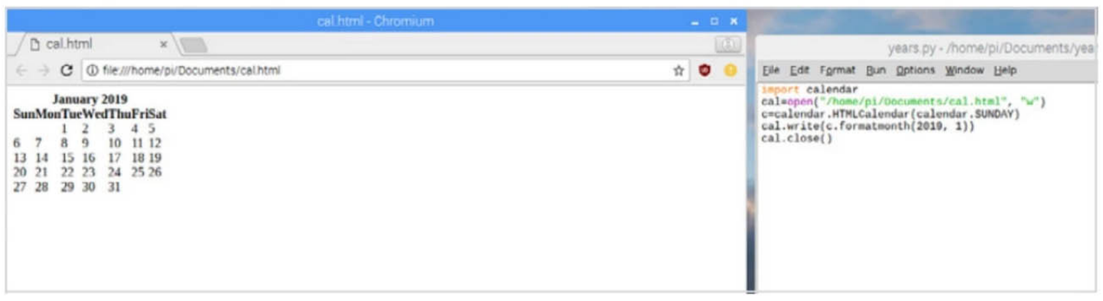



## STEP 9

The Calendar module also allows us to write the functions in HTML, so that you can display it on a website. Let's start by creating a new file:

```
import calendar
cal=open("/home/pi/Documents/cal.html", "w")
c=calendar.HTMLCalendar(calendar.SUNDAY)
cal.write(c.formatmonth(2019, 1))
cal.close()
```

This code will create an HTML file called cal, open it with a browser and it displays the calendar for January 2019.

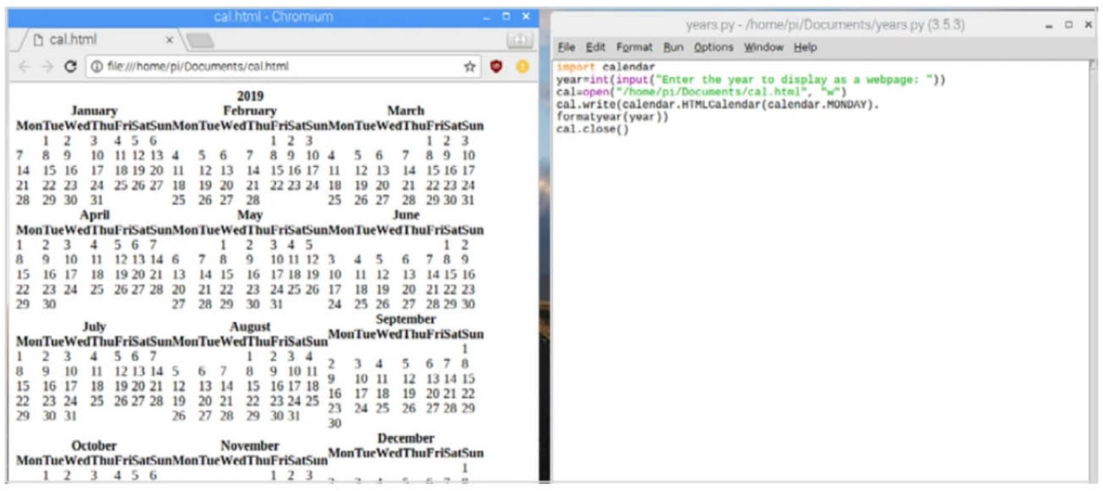



## STEP 10

Of course, you can modify that to display a given year as a web page calendar:

```
import calendar

year=int(input("Enter the year to display as a
webpage: "))
cal=open("/home/pi/Documents/cal.html", "w")
cal.write(calendar.HTMLCalendar(calendar.MONDAY).
formatyear(year))
cal.close()
```

This code asks the user for a year, and then creates the necessary webpage. Remember to change your file destination.



# OS Module

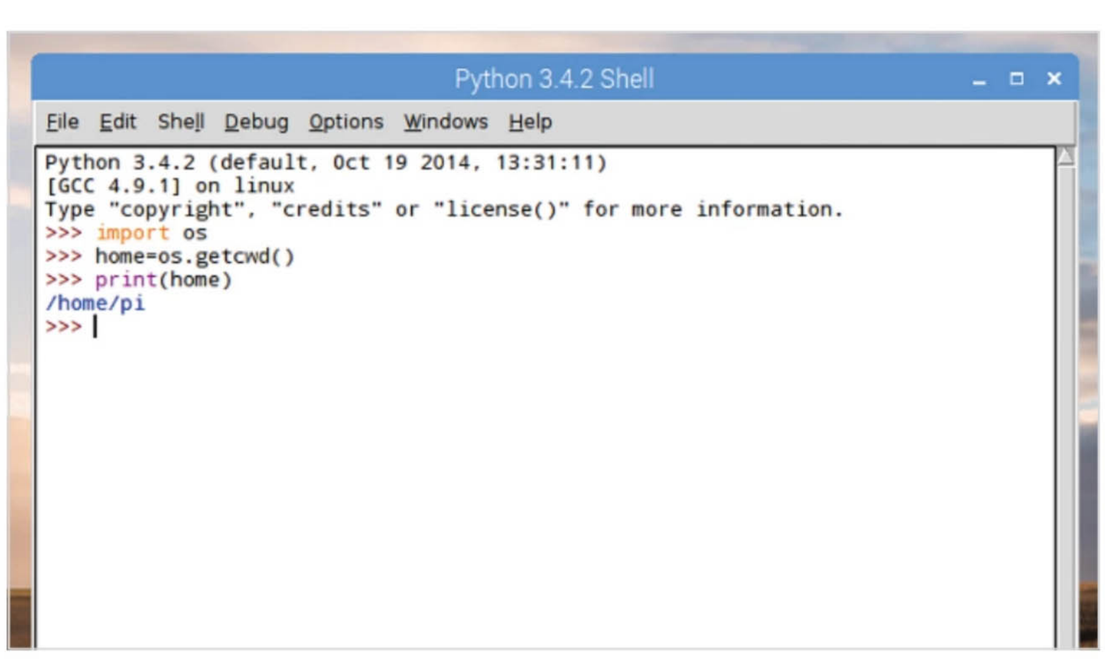
The OS module allows you to interact directly with the built-in commands found in your operating system. Commands vary depending on the OS you're running, as some will work with Windows whereas others will work with Linux and macOS.

## INTO THE SYSTEM

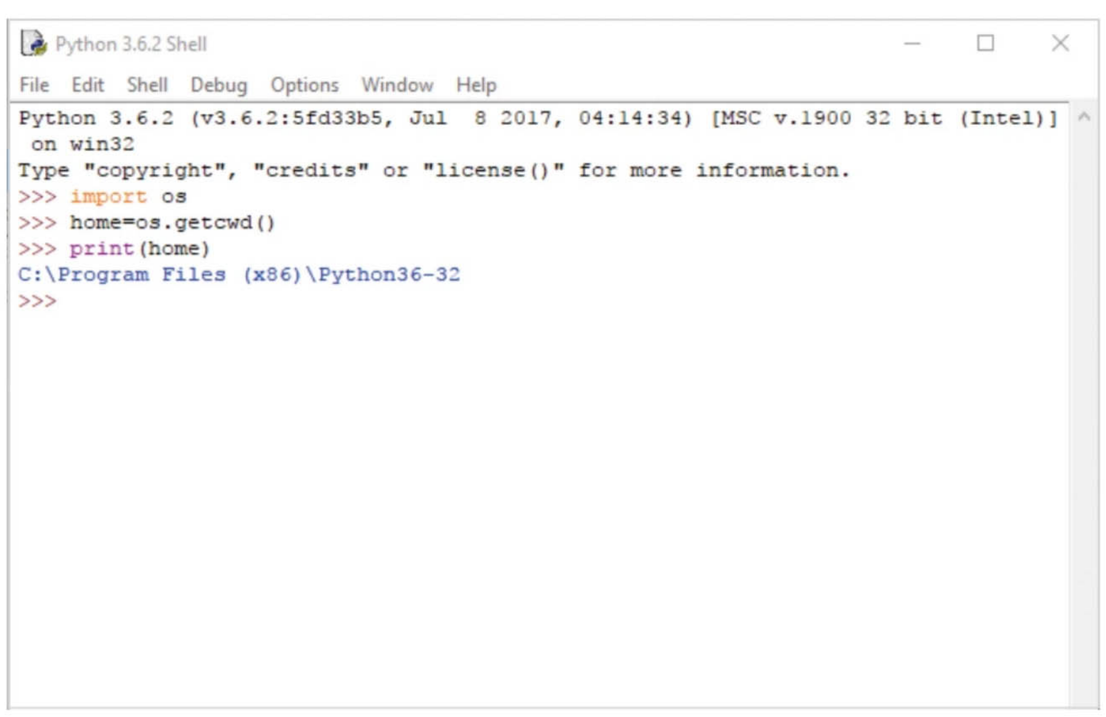
**One of the primary features of the OS module is the ability to list, move, create, delete and otherwise interact with files stored on the system, making it the perfect module for backup code.**

**STEP 1** You can start the OS module with some simple functions to see how it interacts with the operating system environment that Python is running on. If you're using Linux or the Raspberry Pi, try this:

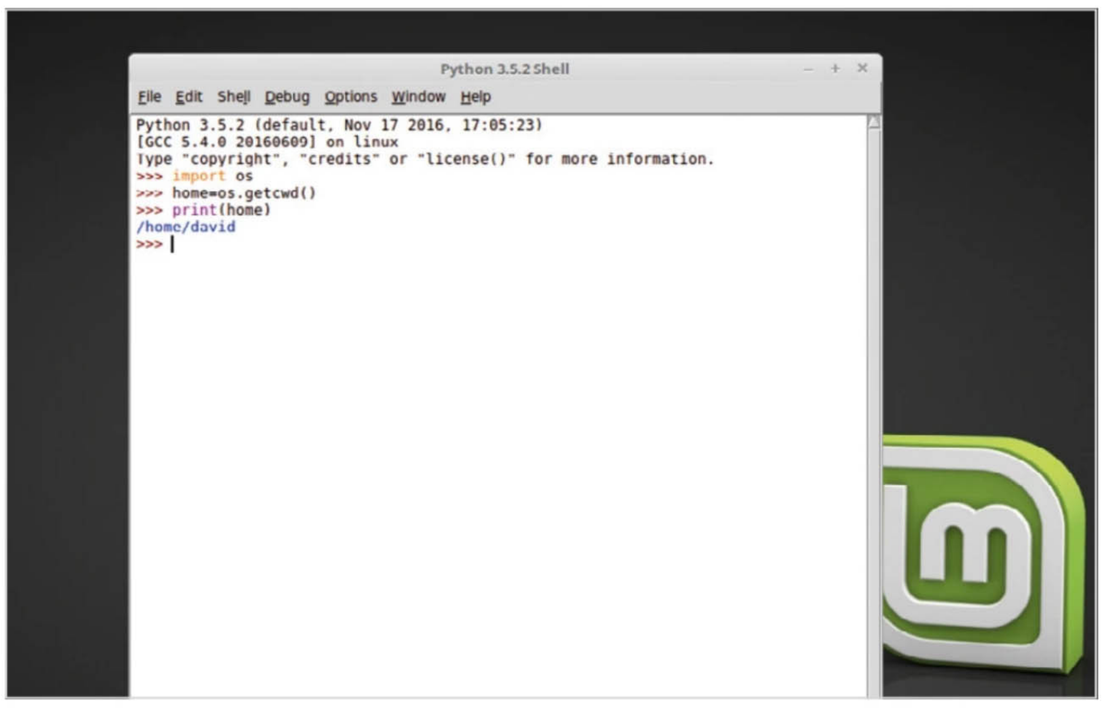
```
import os
home=os.getcwd()
print(home)
```




**STEP 2** The returned result from printing the variable home is the current user's home folder on the system. In our example that's /home/pi; it will be different depending on the user name you log in as and the operating system you use. For example, Windows 10 will output: C:\Program Files (x86)\Python36-32.

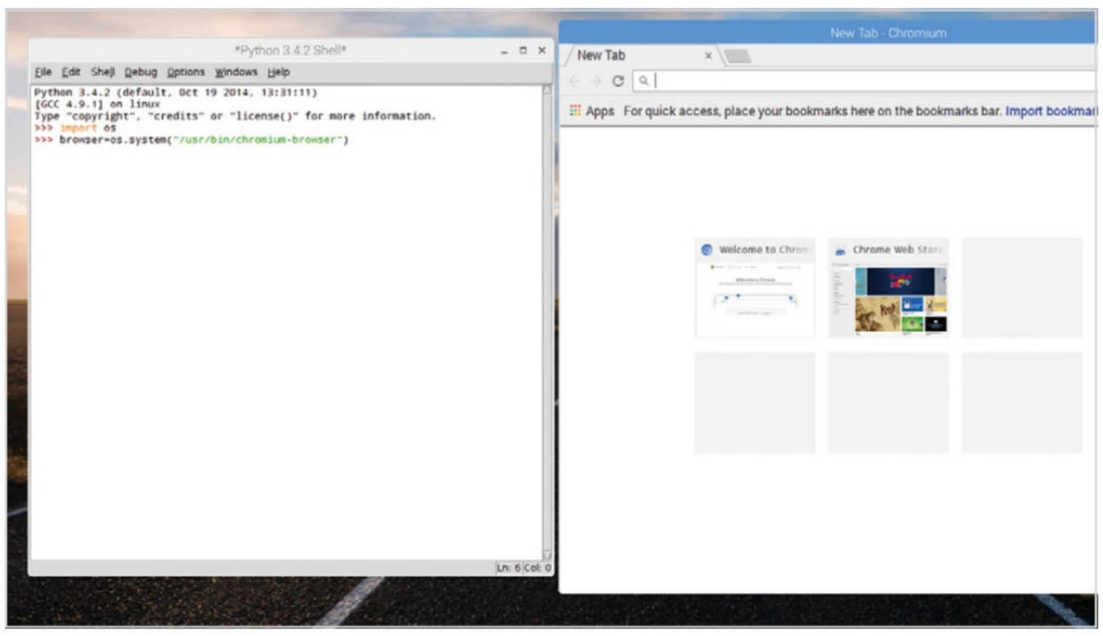



**STEP 3** The Windows output is different as that's the current working directory of Python, as determined by the system; as you might suspect, the os.getcwd() function is asking Python to retrieve the Current Working Directory. Linux users will see something along the same lines as the Raspberry Pi, as will macOS users.



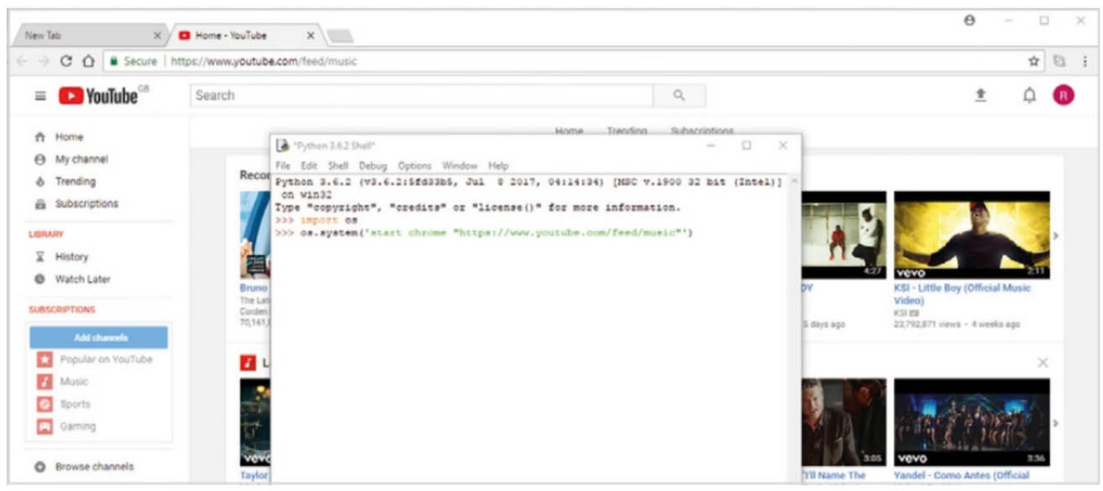

**STEP 4** Yet another interesting element to the OS module, is its ability to launch programs that are installed in the host system. For instance, if you wanted to launch the Chromium browser from within a Python program you can use the command:

```
import os
browser=os.system("/usr/bin/chromium-browser")
```



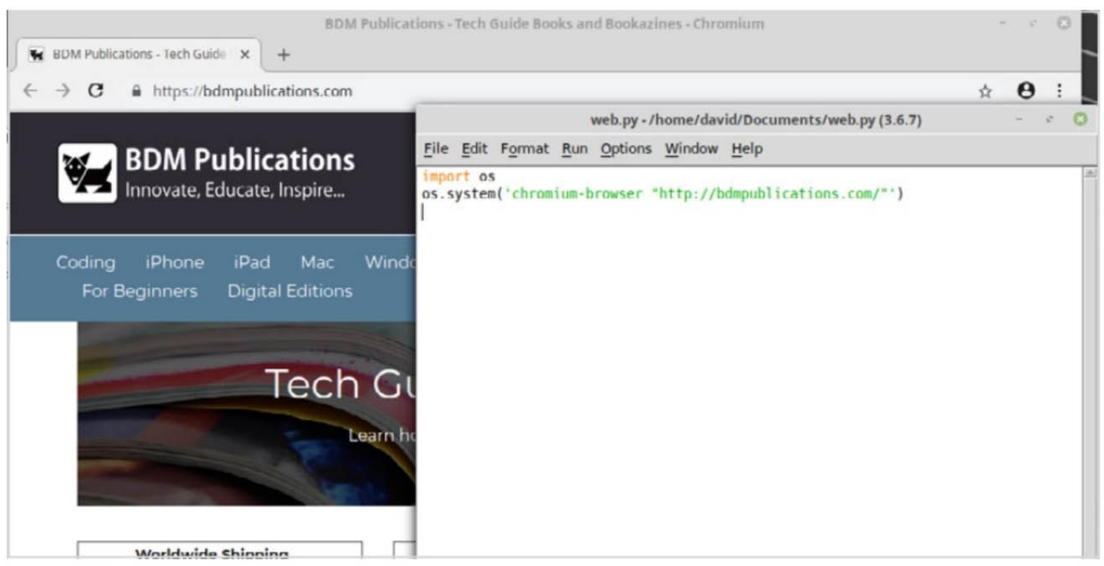
**STEP 5** The os.system() function is what allows interaction with external programs; you can even call up previous Python programs using this method. You will obviously need to know the full path and program file name for it to work successfully. However, you can use the following:

```
import os
os.system('start chrome "https://www.youtube.com/feed/music"')
```



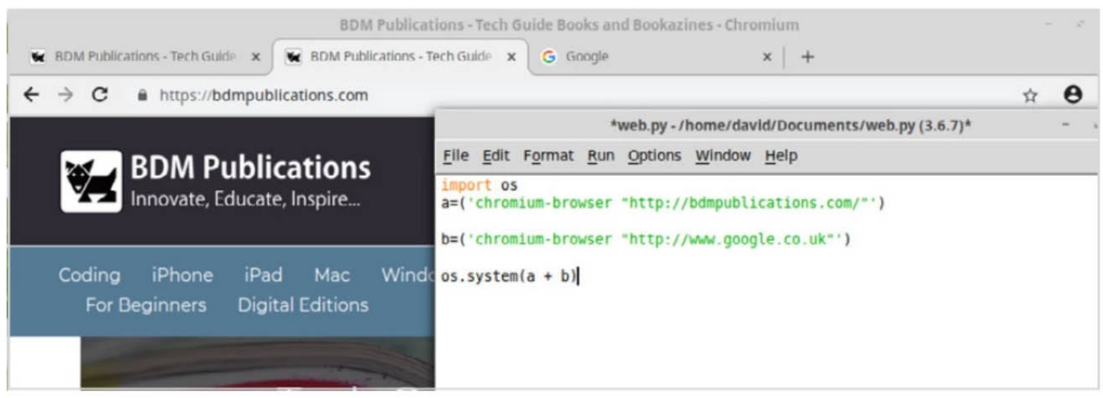

**STEP 6** For Step 5's example we used Windows, to show that the OS module works roughly the same across all platforms. In that case, we opened YouTube's music feed page, so it is therefore possible to open specific pages:

```
import os
os.system('chromium-browser "http://bdmpublications.com/"')
```




**STEP 7** Note in the previous step's example the use of single and double-quotes. The single quotes encase the entire command and launching Chromium, whereas the double quotes open the specified page. You can even use variables to call multiple tabs in the same browser:

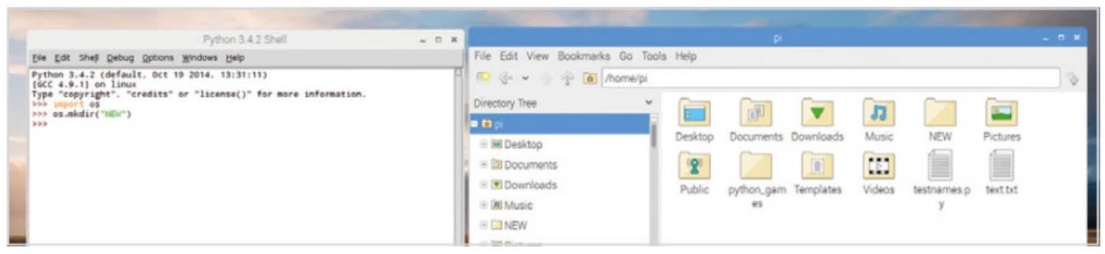
```
import os
a=('chromium-browser "http://bdmpublications.com/"')
b=('chromium-browser "http://www.google.co.uk"')
os.system(a + b)
```




**STEP 8** The ability to manipulate directories, or folders if you prefer, is one of the OS module's best features. For example, to create a new directory you can use:

```
import os
os.mkdir("NEW")
```

This creates a new directory within the Current Working Directory, named according to the object in the mkdir function.

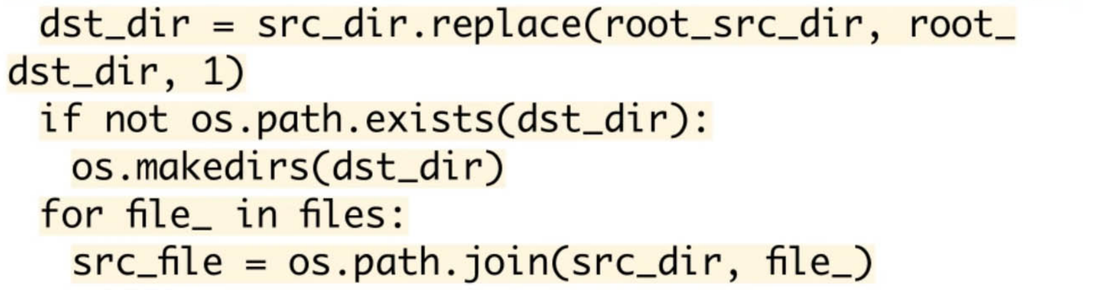



**STEP 9** You can also rename any directories you've created by entering:

```
import os
os.rename("NEW", "OLD")
```

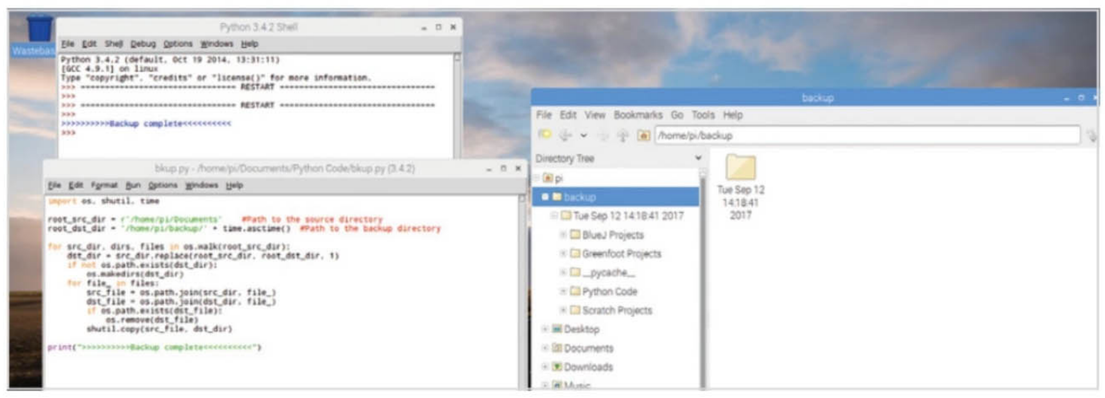
To delete them:

```
import os
os.rmdir("OLD")
```




**STEP 10** Another module that goes together with OS is shutil. You can use the Shutil module together with OS and time to create a time-stamped backup directory, and copy files into it:

```
import os, shutil, time

root_src_dir = r'/home/pi/Documents'
root_dst_dir = '/home/pi/backup/' + time.asctime()

for src_dir, dirs, files in os.walk(root_src_dir):
  dst_dir = src_dir.replace(root_src_dir, root_dst_dir, 1)
  if not os.path.exists(dst_dir):
    os.makedirs(dst_dir)
  for file_ in files:
    src_file = os.path.join(src_dir, file_)
    dst_file = os.path.join(dst_dir, file_)
    if os.path.exists(dst_file):
      os.remove(dst_file)
    shutil.copy(src_file, dst_dir)

print(">>>>>>>>>>Backup complete<<<<<<<<<<")
```



# Random Module

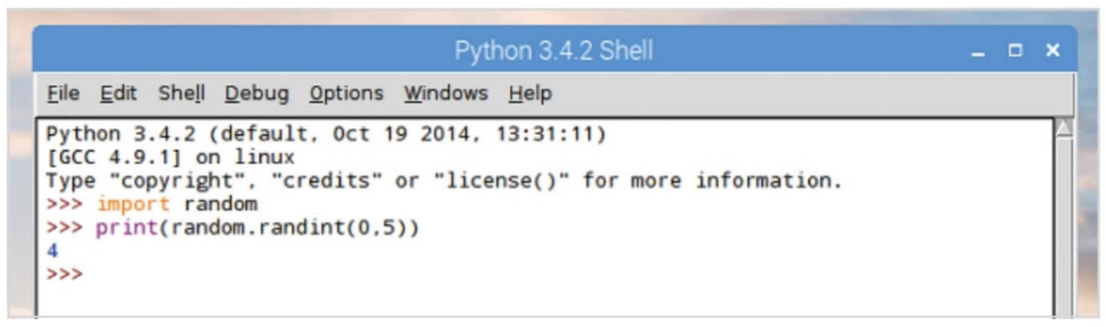
The Random module is one you will likely come across many times in your Python programming lifetime; as the name suggests, it's designed to create random numbers or letters. However, it's not exactly random but it will suffice for most needs.

## RANDOM NUMBERS

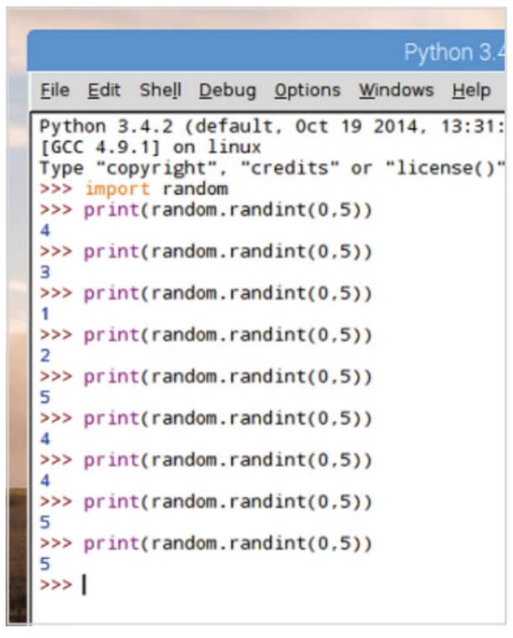
**There are numerous functions within the Random module, which when applied can create some interesting and very useful Python programs.**

**STEP 1** Just as with other modules you need to import random before you can use any of the functions we're going to look at in this tutorial. Let's begin by simply printing a random number from 1 to 5:

```
import random
print(randomint(0,5))
```



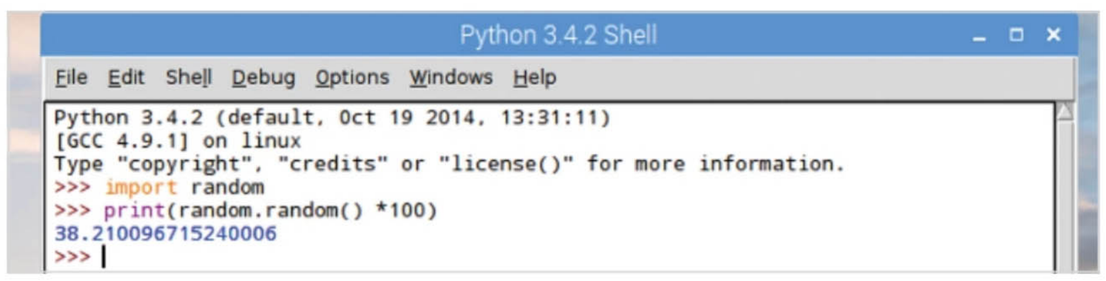

**STEP 2** In our example the number four was returned. However, enter the print function a few more times and it will display different integer values from the set of numbers given, zero to five. The overall effect, although pseudo-random, is adequate for the average programmer to utilise in their code.




**STEP 3** For a bigger set of numbers, including floating point values, you can extend the range by using the multiplication sign:

```
import random
print(random.random() *100)
```

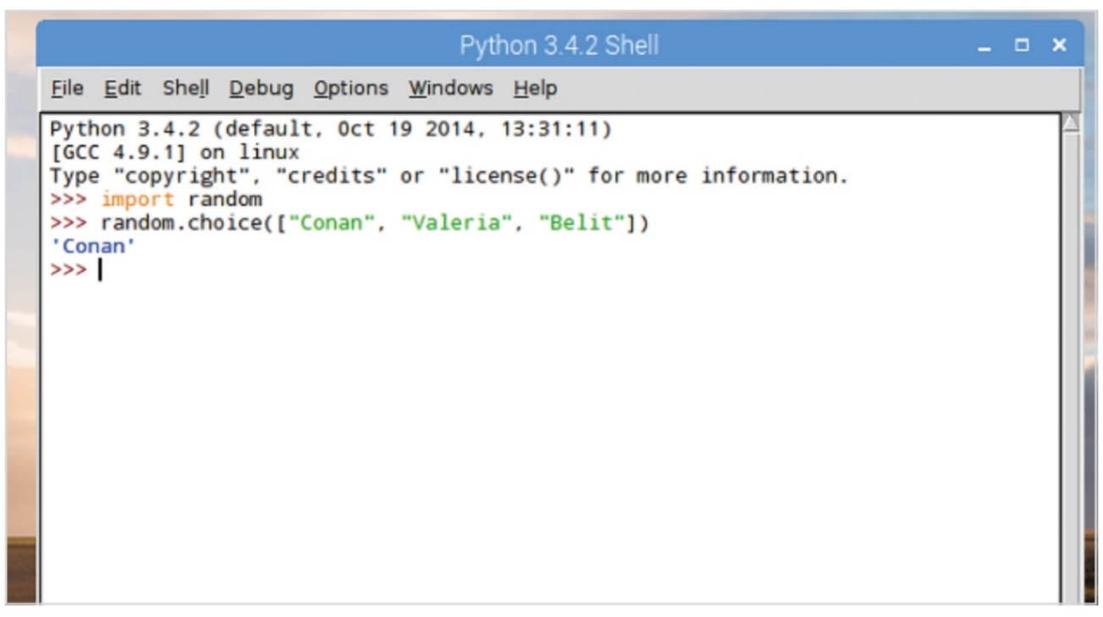
Will display a floating point number between 0 and 100, to the tune of around fifteen decimal points.




**STEP 4** However, the Random module isn't used exclusively for numbers. You can use it to select an entry from a list from random, and the list can contain anything:

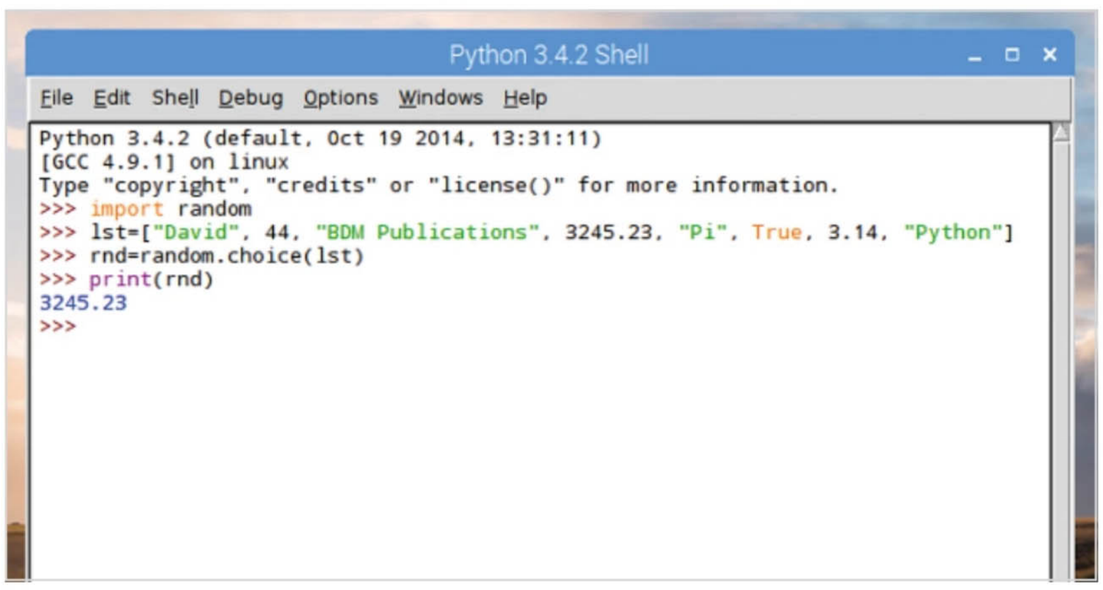
```
import random
random.choice(["Conan", "Valeria", "Belit"])
```

This will display one of the names of our adventurers at random, which is a great addition to a text adventure game.




**STEP 5** You can extend the previous example somewhat by having random.choice() select from a list of mixed variables. For instance:

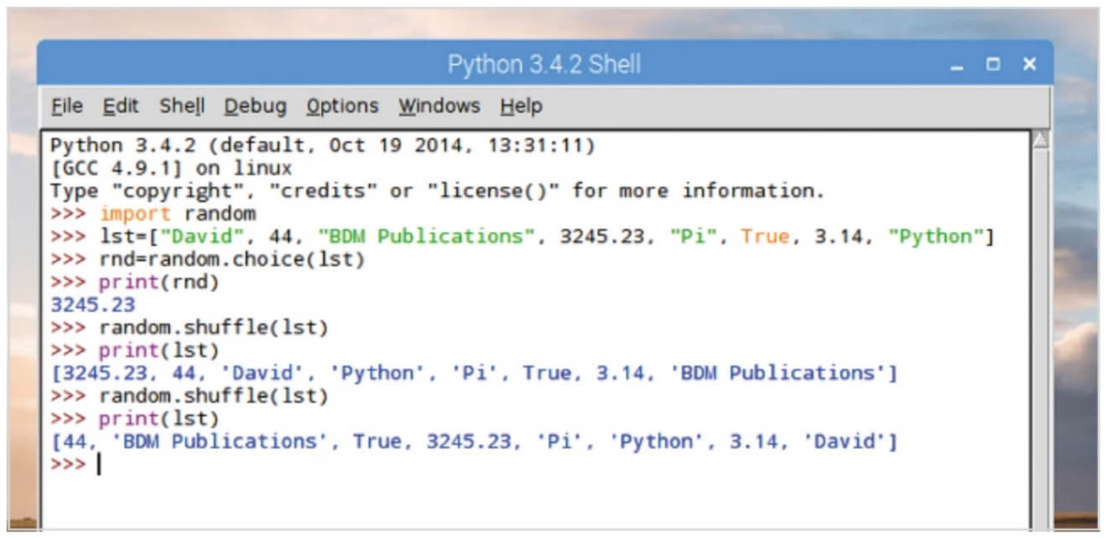
```
import random
lst=["David", 44, "BDM Publications", 3245.23,
"Pi", True, 3.14, "Python"]
rnd=random.choice(lst)
print(rnd)
```



**STEP 6** Interestingly, you can also use a function within the Random module to shuffle the items in the list, thus adding a little more randomness into the equation:

```
random.shuffle(lst)
print(lst)
```

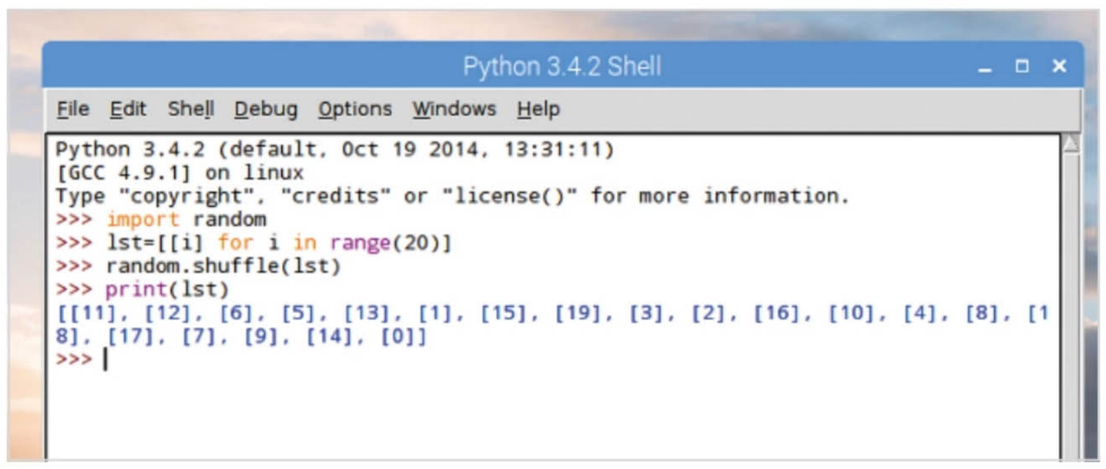
This way, you can keep shuffling the list before displaying a random item from it.




**STEP 7** Using shuffle, you can create an entirely random list of numbers. For example, within a given range:

```
import random
lst=[[i] for I in range(20)]
random.shuffle(lst)
print(lst)
```

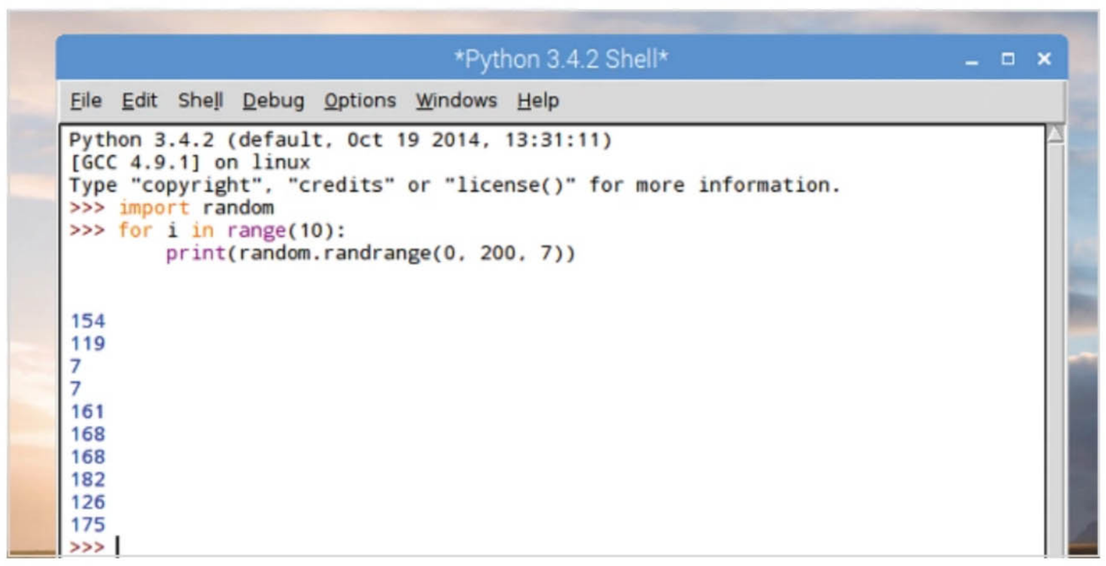
Keep shuffling the list and you can have a different selection of items from 0 to 20 every time.




**STEP 8** You can also select a random number from a given range in steps, using the start, stop, step loop:

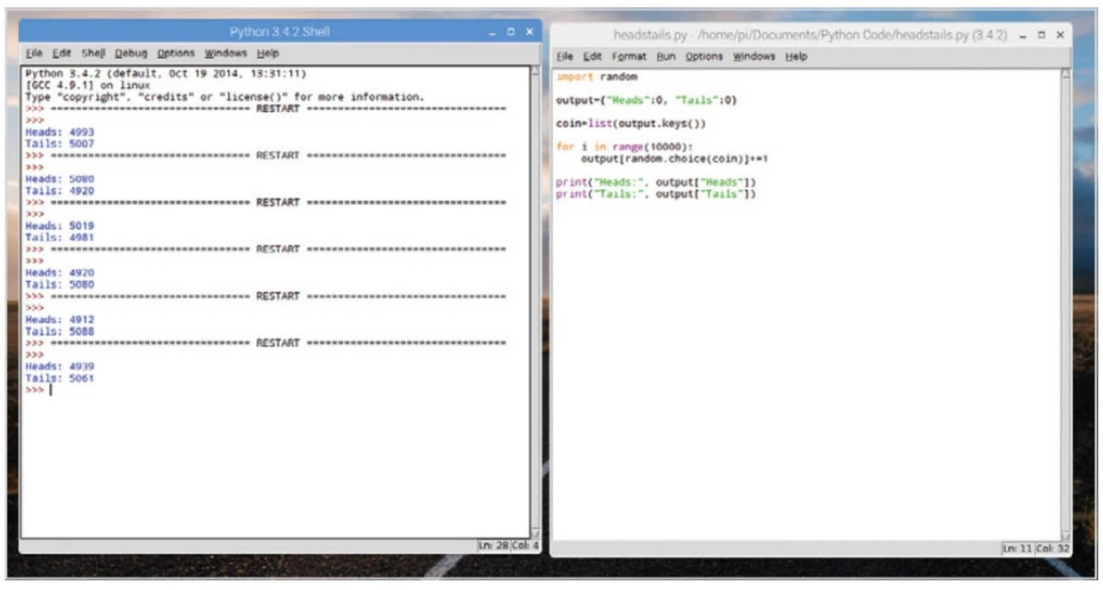
```
import random
for i in range(10):
  print(random.randrange(0, 200, 7))
```

Results will vary but you get the general idea as to how it works.



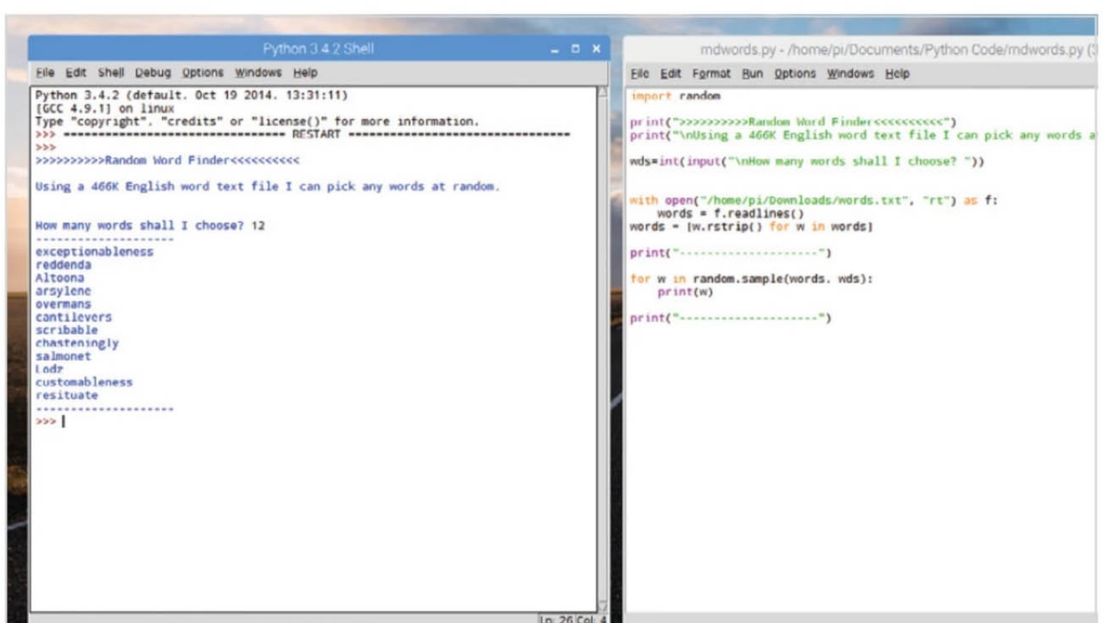

**STEP 9** Let's use an example piece of code which flips a virtual coin ten thousand times and counts how many times it will land on heads or tails:

```
import random
output={"Heads":0, "Tails":0}
coin=list(output.keys())

for i in range(10000):
  output[random.choice(coin)]+=1

print("Heads:", output["Heads"])
print("Tails:", output["Tails"])
```




**STEP 10** Here's an interesting piece of code. Using a text file containing 466 thousand words, you can pluck a user generated number of words from the file (text file found at: **www.github.com/dwyl/english-words**):

```
import random

print(">>>>>>>>>>Random Word Finder<<<<<<<<<<")
print("\nUsing a 466K English word text file I can
pick any words at random.\n")

wds=int(input("\nHow many words shall I choose?
"))

with open("/home/pi/Downloads/words.txt", "rt") as
f:
  words = f.readlines()
words = [w.rstrip() for w in words]

print("--------------------")

for w in random.sample(words, wds):
  print(w)

print("--------------------")
```



# Tkinter Module

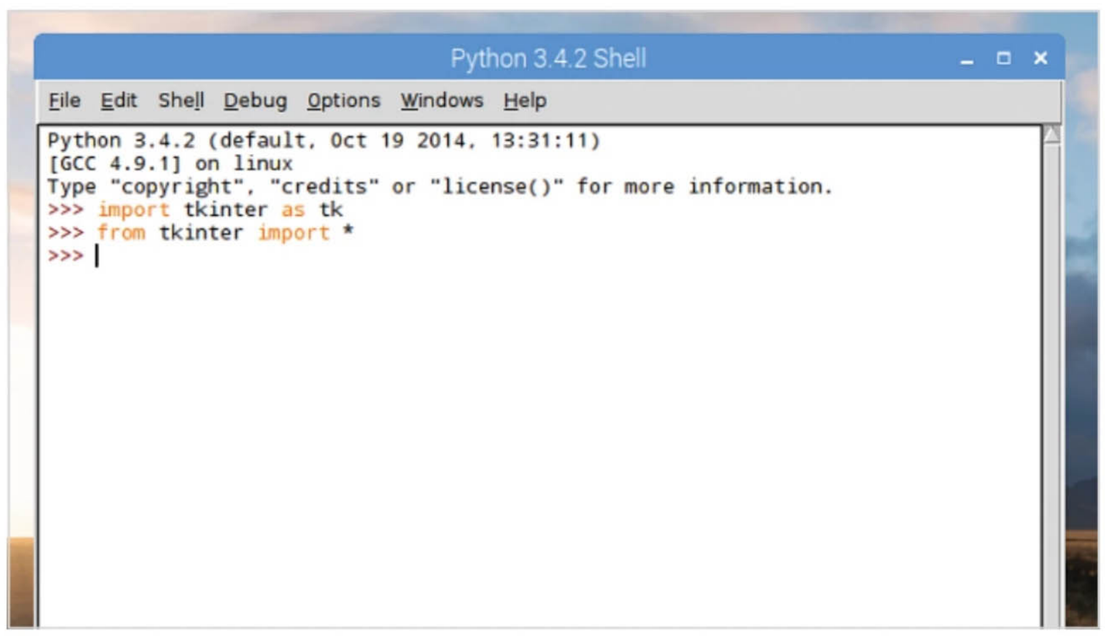
While running your code from the command line, or even in the Shell, is perfectly fine, Python is capable of so much more. The Tkinter module enables the programmer to set up a Graphical User Interface to interact with the user, and it's surprisingly powerful too.

## GETTING GUI

**Tkinter is easy to use but there's a lot more you can do with it. Let's start by seeing how it works and getting some code into it. Before long you will discover just how powerful this module really is.**

**STEP 1** Tkinter is usually built into Python 3. However, if it's available when you enter: `import tkinter`, then you need to `pip install tkinter` from the command prompt. We can start to import modules differently than before, to save on typing and by importing all their contents:

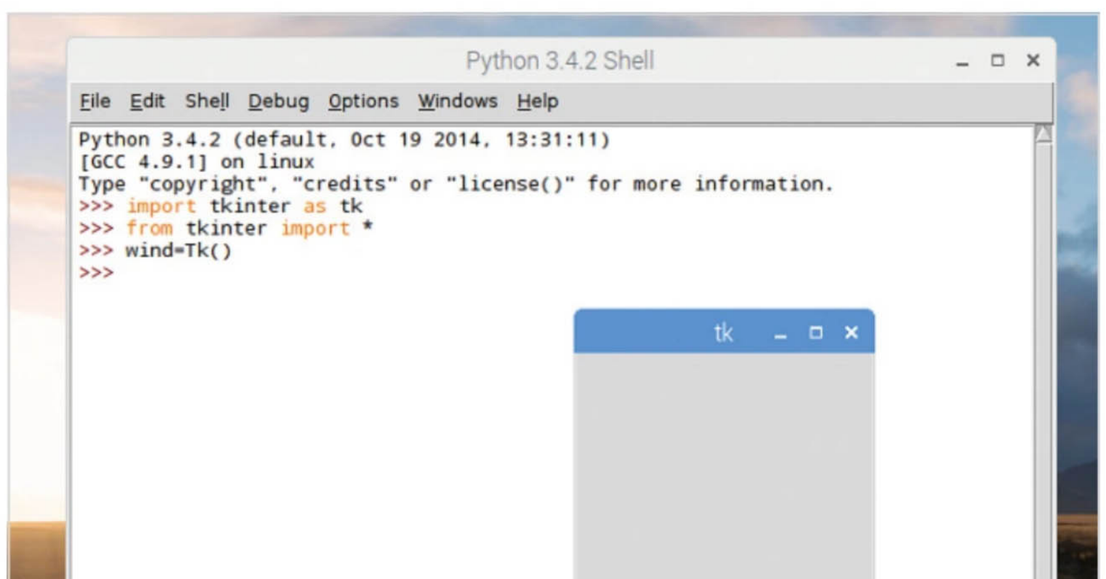
```
import tkinter as tk
from tkinter import *
```




**STEP 2** It's not recommended to import everything from a module using the asterisk but it won't do any harm normally. Let's begin by creating a basic GUI window, enter:

```
wind=Tk()
```

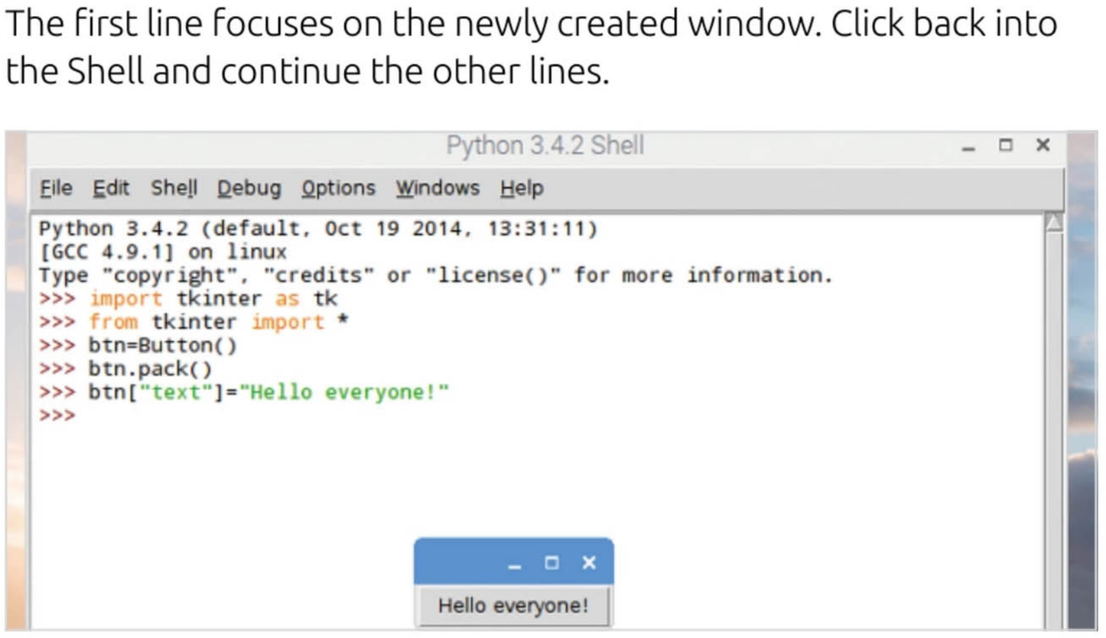
This creates a small, basic window. There's not much else to do at this point but click the X in the corner to close the window.




**STEP 3** The ideal approach is to add mainloop() into the code to control the Tkinter event loop, but we'll get to that soon. You've just created a Tkinter widget and there are several more we can play around with:

```
btn=Button()
btn.pack()
btn["text"]="Hello everyone!"
```

The first line focuses on the newly created window. Click back into the Shell and continue the other lines.




**STEP 4** You can combine the above into a New File:

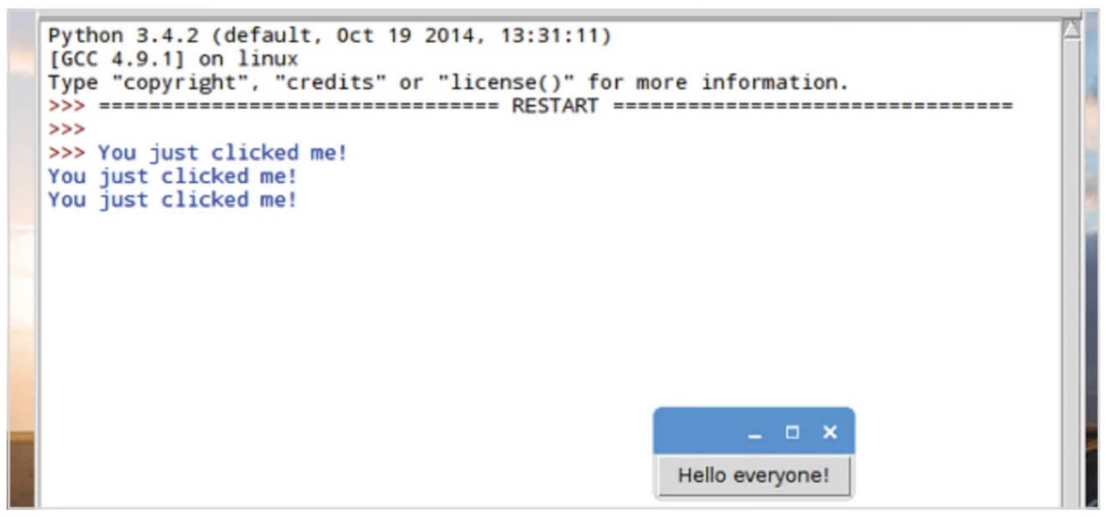
```
import tkinter as tk
from tkinter import *
btn=Button()
btn.pack()
btn["text"]="Hello everyone!"
```

Then add some button interactions:

```
def click():
  print("You just clicked me!")
btn["command"]=click
```

## STEP 5

Save and execute the code from Step 5 and a window appears with 'Hello everyone!' inside. If you click the Hello everyone! button, the Shell will output the text 'You just clicked me!'. It's simple but shows you what can be achieved with a few lines of code.



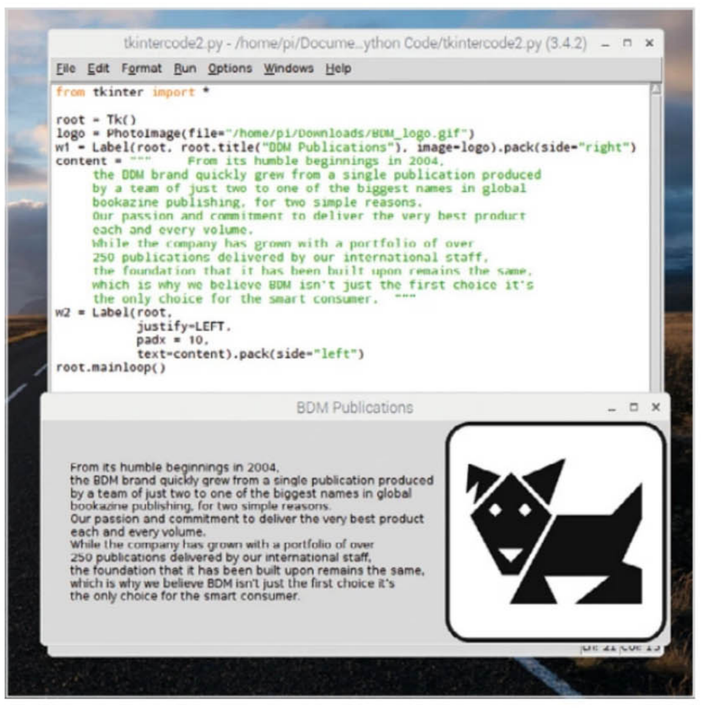

## STEP 6

You can also display both text and images within a Tkinter window. However, only GIF, PGM or PPM formats are supported. So find an image and convert it before using the code. Here's an example using the BDM Publishing logo:

```
from tkinter import *

root = Tk()
logo = PhotoImage(file="/home/pi/Downloads/BDM_logo.
gif")
w1 = Label(root, root.title("BDM Publications"),
image=logo).pack(side="right")
content = """  From its humble beginnings in 2004,
the BDM brand quickly grew from a single publication
produced by a team of just two to one of the biggest
names in global bookazine publishing, for two simple
reasons. Our passion and commitment to deliver the
very best product each and every volume. While
the company has grown with a portfolio of over 250
publications delivered by our international staff,
the foundation that it has been built upon remains
the same, which is why we believe BDM isn't just
the first choice it's the only choice for the smart
consumer.  """
w2 = Label(root,
    justify=LEFT,
    padx = 10,
    text=content).pack(side="left")
root.mainloop()
```

## STEP 7

The previous code is quite weighty, mostly due to the content variable holding a part of BDM's About page from the company website. You can obviously change the content, the root.title and the image to suit your needs.




## STEP 8

You can create radio buttons too. Try:

```
from tkinter import *

root = Tk()

v = IntVar()

Label(root, root.title("Options"), text="""Choose
a preferred language:""",
   justify = LEFT, padx = 20).pack()
Radiobutton(root,
     text="Python",
     padx = 20,
     variable=v,
     value=1).pack(anchor=W)
Radiobutton(root,
     text="C++",
     padx = 20,
     variable=v,
     value=2).pack(anchor=W)

mainloop()
```

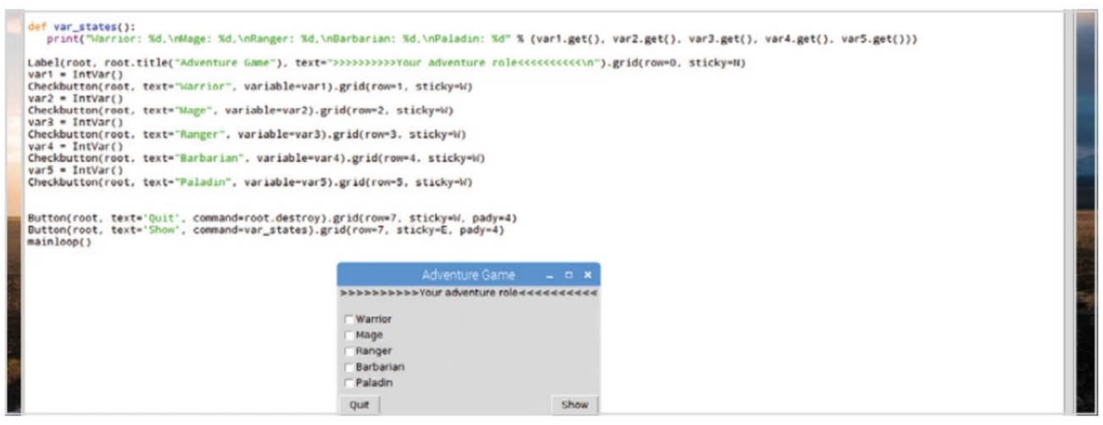
## STEP 9

You can also create check boxes, with buttons and output to the Shell:

```
from tkinter import *
root = Tk()

def var_states():
  print("Warrior: %d,\nMage: %d" % (var1.get(),
var2.get()))

Label(root, root.title("Adventure Game"),
text=">>>>>>>>>>Your adventure role<<<<<<<<<<").
grid(row=0, sticky=N)
var1 = IntVar()
Checkbutton(root, text="Warrior", variable=var1).
grid(row=1, sticky=W)
var2 = IntVar()
Checkbutton(root, text="Mage", variable=var2).
grid(row=2, sticky=W)
Button(root, text='Quit', command=root.destroy).
grid(row=3, sticky=W, pady=4)
Button(root, text='Show', command=var_states).
grid(row=3, sticky=E, pady=4)

mainloop()
```

## STEP 10

The code from Step 9 introduced some new geometry elements into Tkinter. Note the sticky=N, E and W arguments. These describe the locations of the check boxes and buttons (North, East, South and West). The row argument places them on separate rows. Have a play around and see what you get.



# Pygame Module

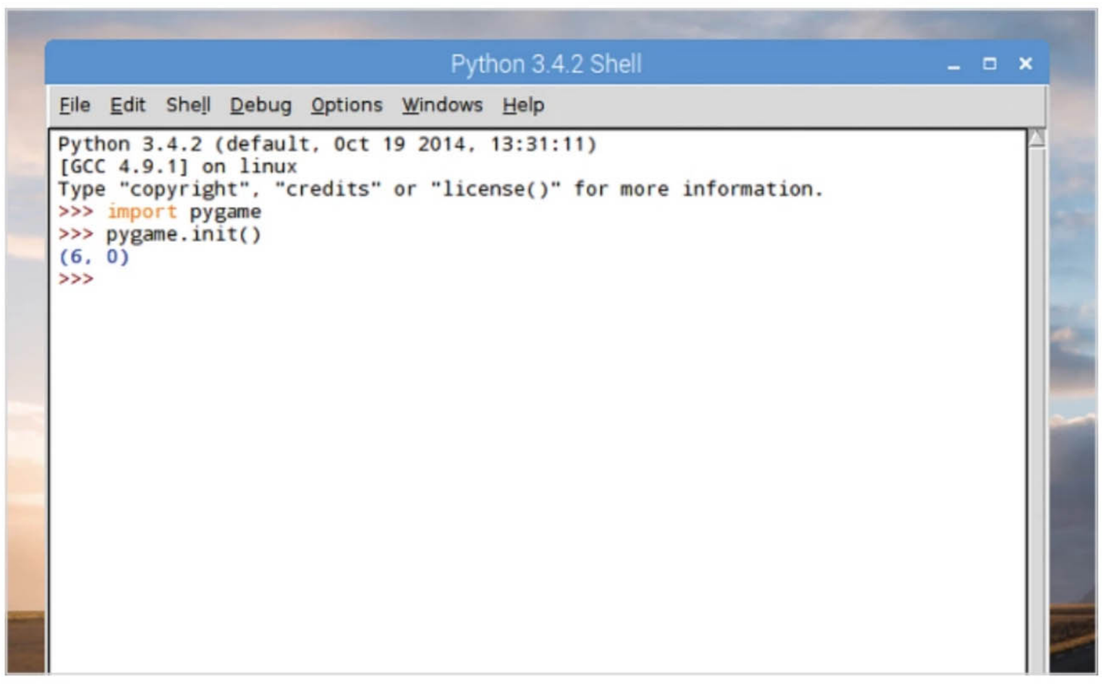
We've had a brief look at the Pygame module already but there's a lot more to it that needs exploring. Pygame was developed to help Python programmers create either graphical or text-based games.

## PYGAMING

**Pygame isn't an inherent module to Python but those using the Raspberry Pi will already have it installed. Everyone else will need to use: pip install pygame from the command prompt.**

**STEP 1** Naturally you need to load up the Pygame modules into memory before you're able to utilise them. Once that's done Pygame requires the user to initialise it prior to any of the functions being used:

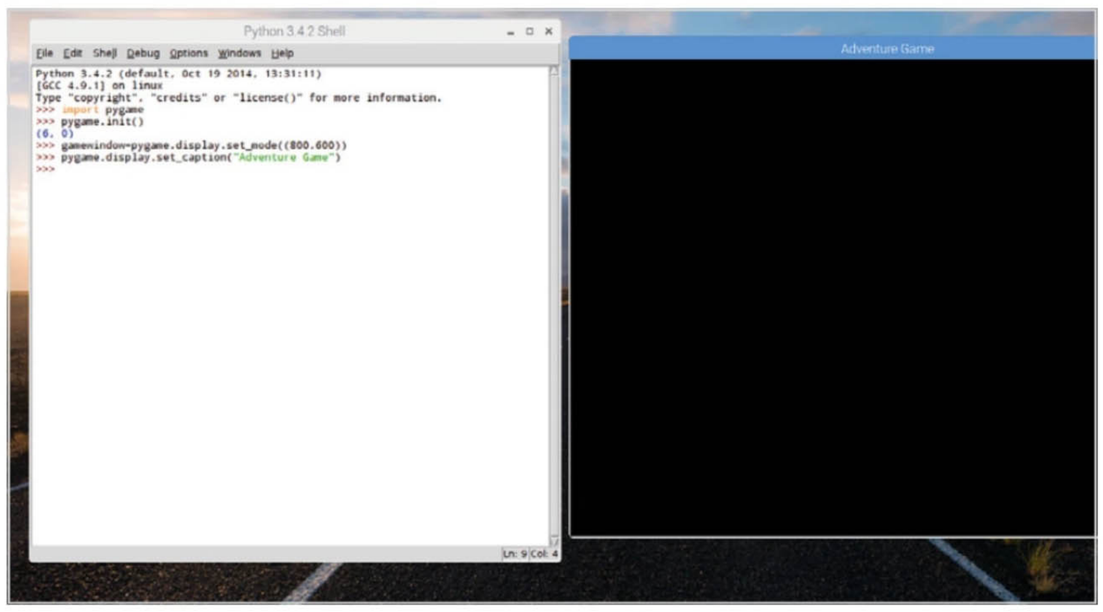
```
import pygame
pygame.init()
```




**STEP 2** Let's create a simple game ready window, and give it a title:

```
gamewindow=pygame.display.set_mode((800,600))
pygame.display.set_caption("Adventure Game")
```

You can see that after the first line is entered, you need to click back into the IDLE Shell to continue entering code; also, you can change the title of the window to anything you like.

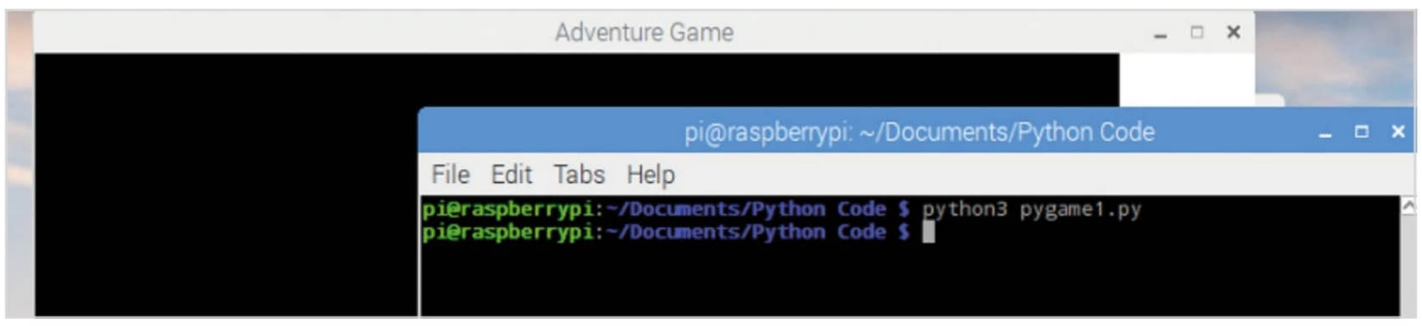



**STEP 3** Sadly you can't close the newly created Pygame window without closing the Python IDLE Shell, which isn't very practical. For this reason, you need to work in the editor (New > File) and create a True/False while loop:

```
import pygame
from pygame.locals import *
pygame.init()

gamewindow=pygame.display.set_mode((800,600))
pygame.display.set_caption("Adventure Game")

running=True

while running:
  for event in pygame.event.get():
    if event.type==QUIT:
      running=False
      pygame.quit()
```

## STEP 4

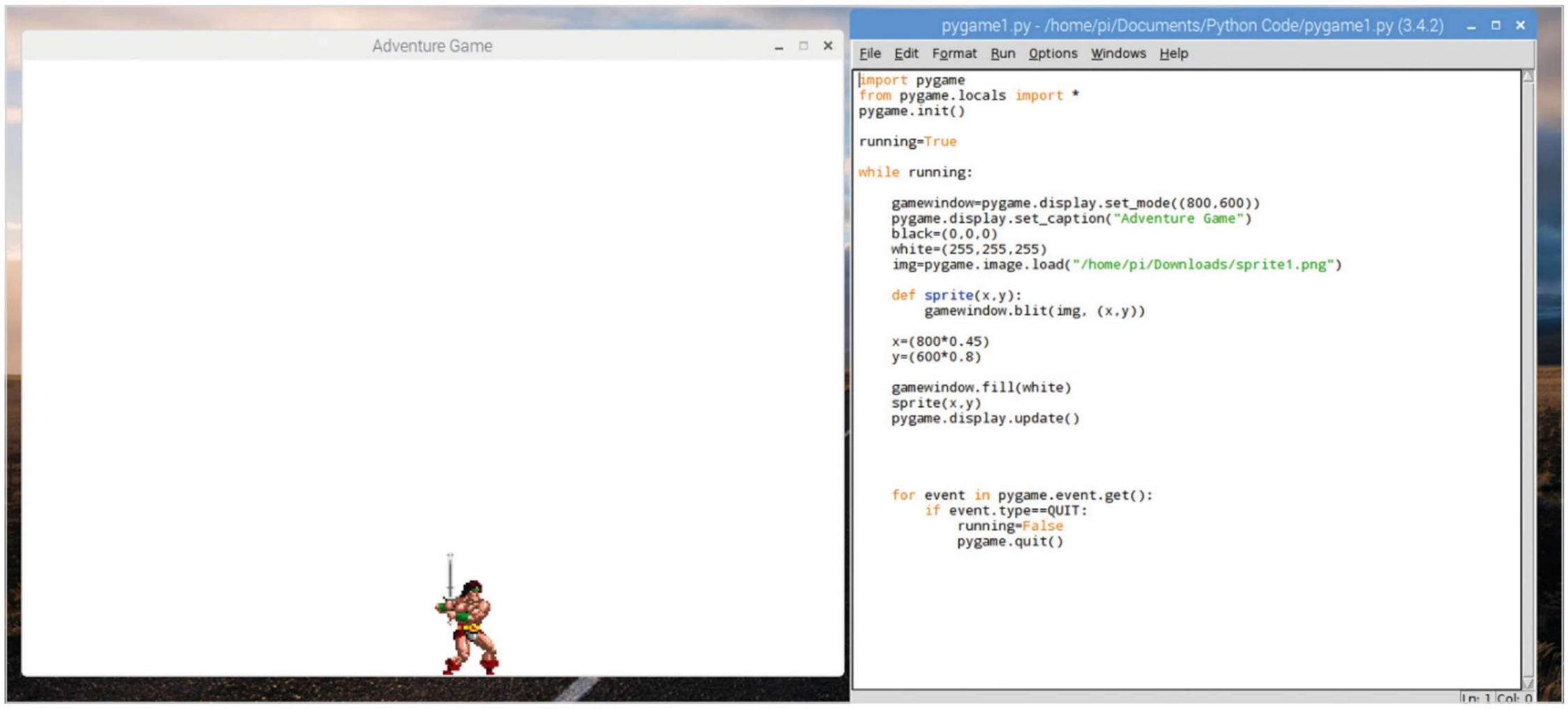
If the Pygame window still won't close don't worry, it's just a discrepancy between the IDLE (which is written with Tkinter) and the Pygame module. If you run your code via the command line, it closes perfectly well.




## STEP 5

You're going to shift the code around a bit now, running the main Pygame code within a while loop; it makes it neater and easier to follow. We've downloaded a graphic to use and we need to set some parameters for pygame:

```
import pygame
pygame.init()

running=True

while running:

  gamewindow=pygame.display.set_mode((800,600))
  pygame.display.set_caption("Adventure Game")
  black=(0,0,0)
  white=(255,255,255)
  img=pygame.image.load("/home/pi/Downloads/sprite1.png")

  def sprite(x,y):
    gamewindow.blit(img, (x,y))

  x=(800*0.45)
  y=(600*0.8)

  gamewindow.fill(white)
  sprite(x,y)
  pygame.display.update()

  for event in pygame.event.get():
    if event.type==pygame.QUIT:
      running=False
```




## STEP 6

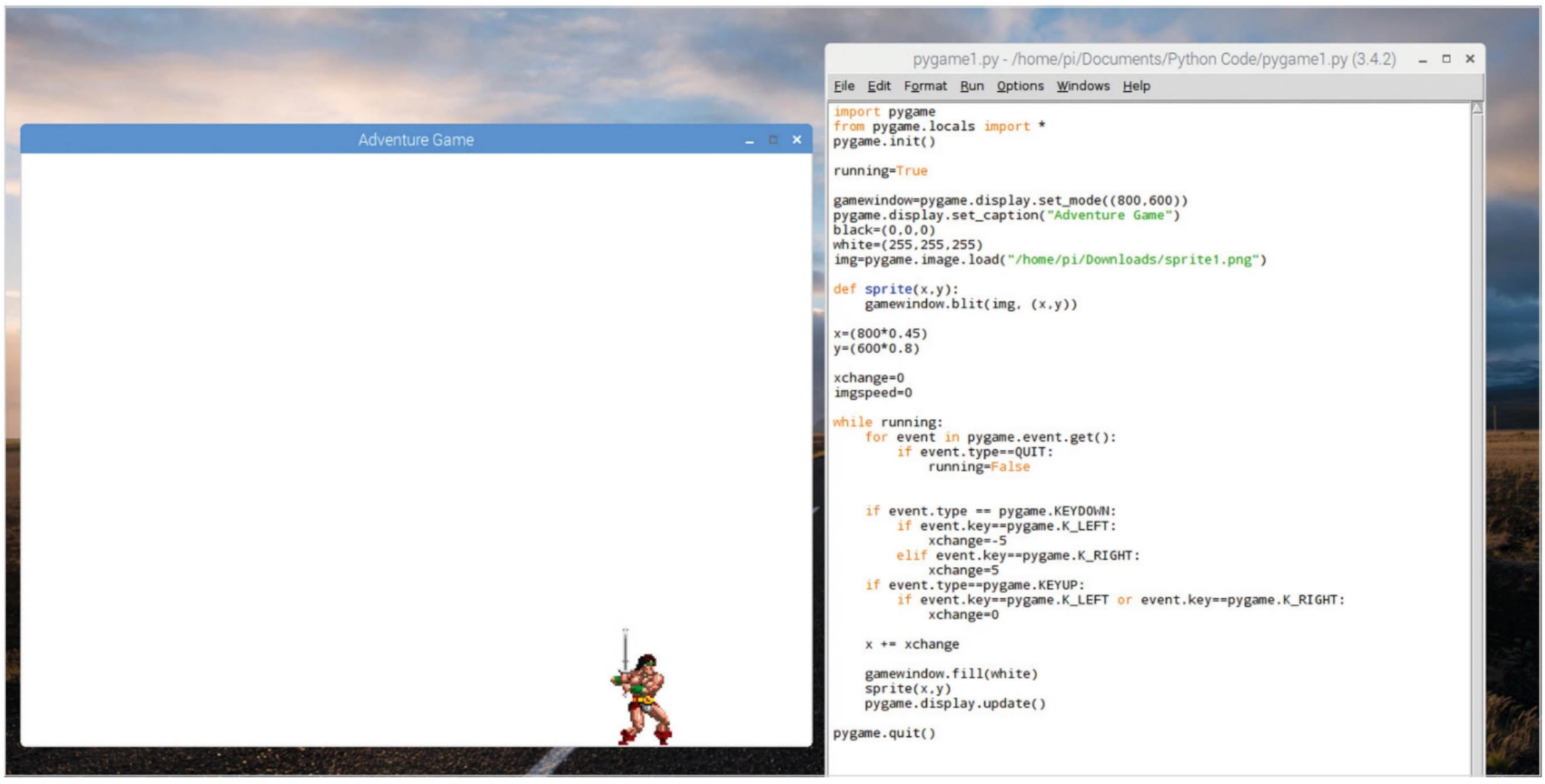
Let's quickly go through the code changes. We've defined two colours, black and white together with their respective RGB colour values. Next we've loaded the downloaded image called sprite1.png and allocated it to the variable img; and also defined a sprite function and the Blit function will allow us to eventually move the image.

```
import pygame
from pygame.locals import *
pygame.init()

running=True

while running:

    gamewindow=pygame.display.set_mode((800,600))
    pygame.display.set_caption("Adventure Game")
    black=(0,0,0)
    white=(255,255,255)
    img=pygame.image.load("/home/pi/Downloads/sprite1.png")

    def sprite(x,y):
        gamewindow.blit(img, (x,y))
```

```
    x=(800*0.45)
    y=(600*0.8)

    gamewindow.fill(white)
    sprite(x,y)
    pygame.display.update()



    for event in pygame.event.get():
        if event.type==QUIT:
            running=False
            pygame.quit()
```

**STEP 7** Now we can change the code around again, this time containing a movement option within the while loop, and adding the variables needed to move the sprite around the screen:

```
import pygame
from pygame.locals import *
pygame.init()

running=True

gamewindow=pygame.display.set_mode((800,600))
pygame.display.set_caption("Adventure Game")
black=(0,0,0)
white=(255,255,255)
img=pygame.image.load("/home/pi/Downloads/sprite1.
png")

def sprite(x,y):
  gamewindow.blit(img, (x,y))

x=(800*0.45)
y=(600*0.8)

xchange=0
imgspeed=0

while running:
  for event in pygame.event.get():
    if event.type==QUIT:
      running=False

  if event.type == pygame.KEYDOWN:
    if event.key==pygame.K_LEFT:
      xchange=-5
    elif event.key==pygame.K_RIGHT:
      xchange=5
  if event.type==pygame.KEYUP:
    if event.key==pygame.K_LEFT or event
key==pygame.K_RIGHT:
      xchange=0

  x += xchange

  gamewindow.fill(white)
  sprite(x,y)
  pygame.display.update()

pygame.quit()
```



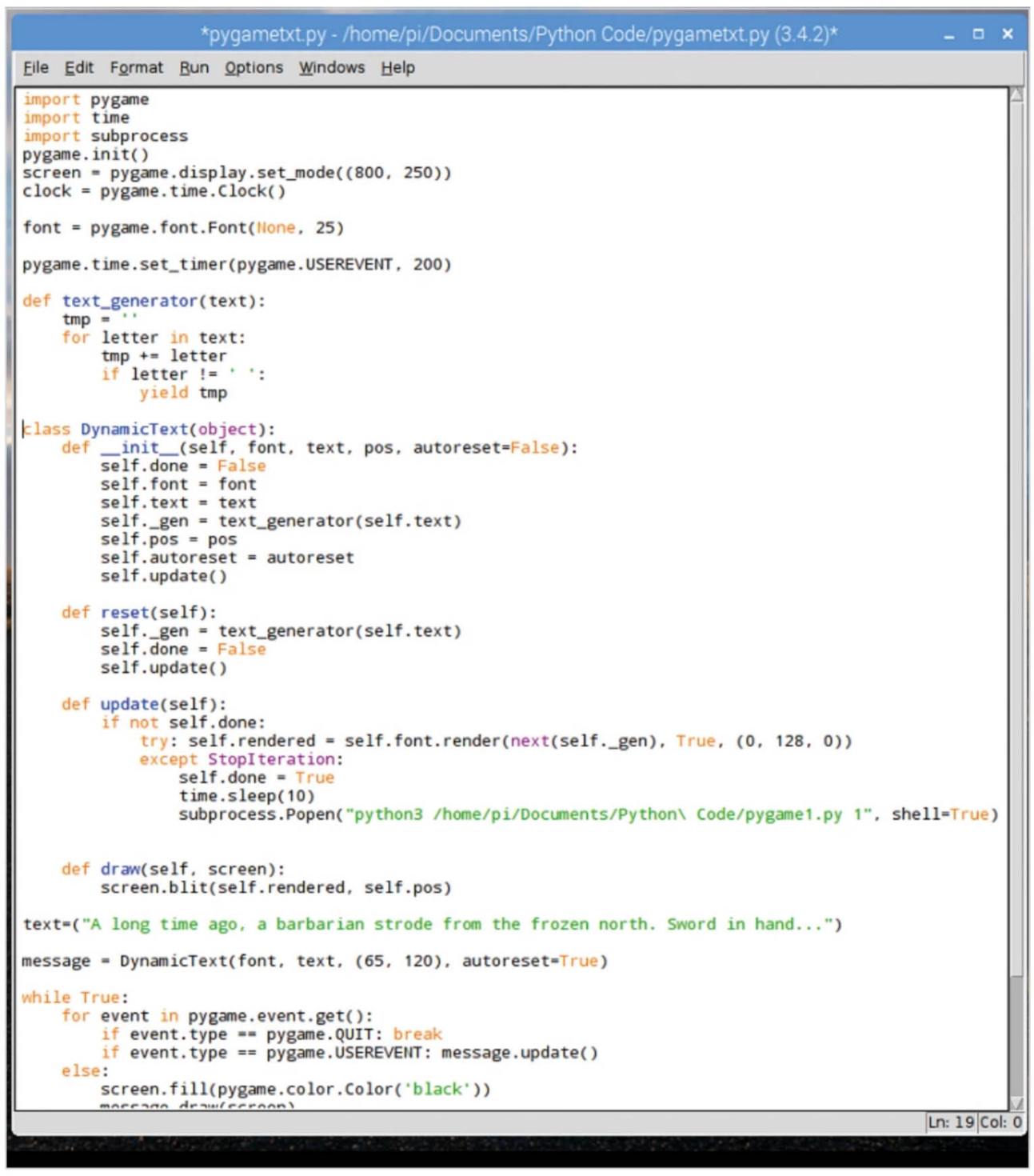

**STEP 8** Copy the code down and using the left and right arrow keys on the keyboard you can move your sprite across the bottom of the screen. Now, it looks like you have the makings of a classic arcade 2D scroller in the works.

```
import pygame
from pygame.locals import *
pygame.init()

running=True

gamewindow=pygame.display.set_mode((800,600))
pygame.display.set_caption("Adventure Game")
black=(0,0,0)
white=(255,255,255)
img=pygame.image.load("/home/pi/Downloads/sprite1.png")

def sprite(x,y):
    gamewindow.blit(img, (x,y))

x=(800*0.45)
y=(600*0.8)

xchange=0
imgspeed=0
```

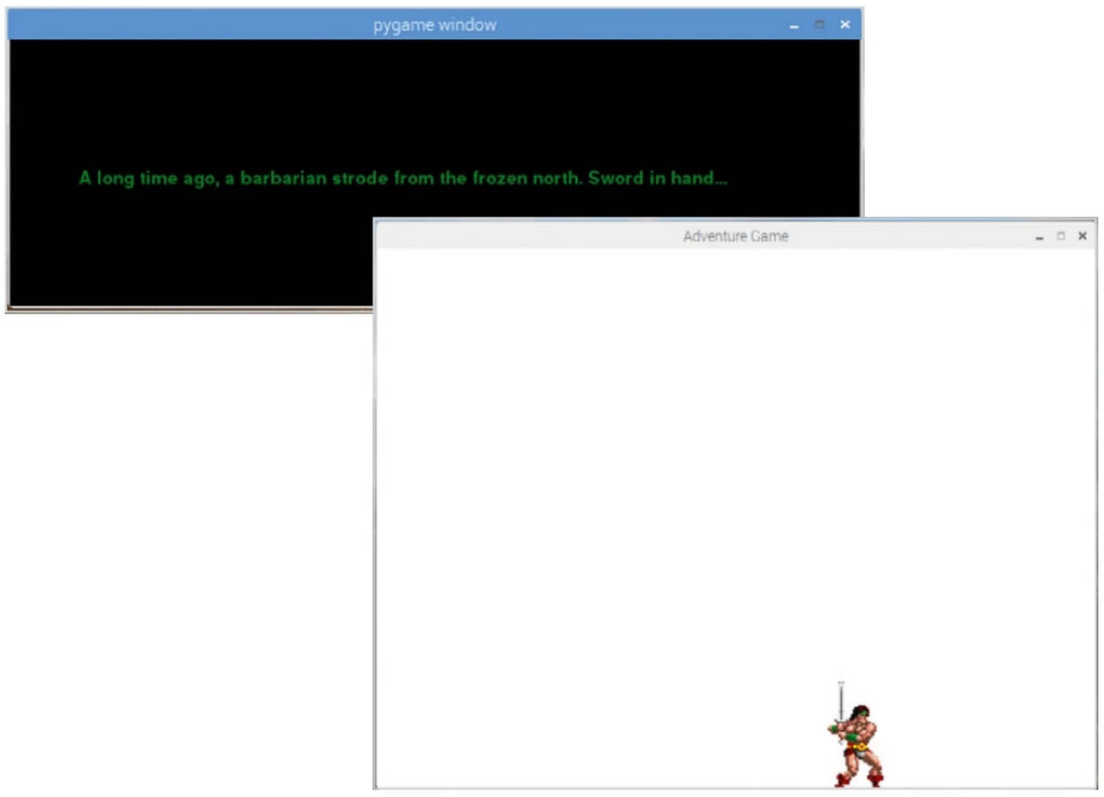
```
while running:
    for event in pygame.event.get():
        if event.type==QUIT:
            running=False


    if event.type == pygame.KEYDOWN:
        if event.key==pygame.K_LEFT:
            xchange=-5
        elif event.key==pygame.K_RIGHT:
            xchange=5
    if event.type==pygame.KEYUP:
        if event.key==pygame.K_LEFT or event.key==pygame.K_RIGHT:
            xchange=0

    x += xchange

    gamewindow.fill(white)
    sprite(x,y)
    pygame.display.update()

pygame.quit()
```

**STEP 9** You can now implement a few additions and utilise some previous tutorial code. The new elements are the Subprocess module, of which one function allows us to launch a second Python script from within another; and we're going to create a New File called pygametxt.py:

```
import pygame
import time
import subprocess
pygame.init()
screen = pygame.display.set_mode((800, 250))
clock = pygame.time.Clock()

font = pygame.font.Font(None, 25)

pygame.time.set_timer(pygame.USEREVENT, 200)

def text_generator(text):
  tmp = ‘’
  for letter in text:
    tmp += letter
    if letter != ‘ ‘:
      yield tmp

class DynamicText(object):
  def __init__(self, font, text, pos,
autoreset=False):
    self.done = False
    self.font = font
    self.text = text
    self._gen = text_generator(self.text)
    self.pos = pos
    self.autoreset = autoreset
    self.update()

  def reset(self):
    self._gen = text_generator(self.text)
    self.done = False
    self.update()

  def update(self):
    if not self.done:
      try: self.rendered = self.font.
render(next(self._gen), True, (0, 128, 0))
      except StopIteration:
        self.done = True
        time.sleep(10)
        subprocess.Popen(“python3 /home/pi/Documents/
Python\ Code/pygame1.py 1”, shell=True)

  def draw(self, screen):
    screen.blit(self.rendered, self.pos)

text=(“A long time ago, a barbarian strode from the
frozen north. Sword in hand...”)

message = DynamicText(font, text, (65, 120),
autoreset=True)

while True:
  for event in pygame.event.get():
    if event.type == pygame.QUIT: break
    if event.type == pygame.USEREVENT: message.
update()
  else:
    screen.fill(pygame.color.Color(‘black’))
    message.draw(screen)
    pygame.display.flip()
    clock.tick(60)
    continue
  break

pygame.quit()
```




**STEP 10** When you run this code it will display a long, narrow Pygame window with the intro text scrolling to the right. After a pause of ten seconds, it then launches the main game Python script where you can move the warrior sprite around. Overall the effect is quite good but there's always room for improvement.



# Basic Animation

Python's modules make it relatively easy to create shapes, or animated graphics. Animation, though, can be a tricky element to get right in code, as there are many different ways of achieving the same end result. Here's one such example.

## LIGHTS, CAMERA, ACTION

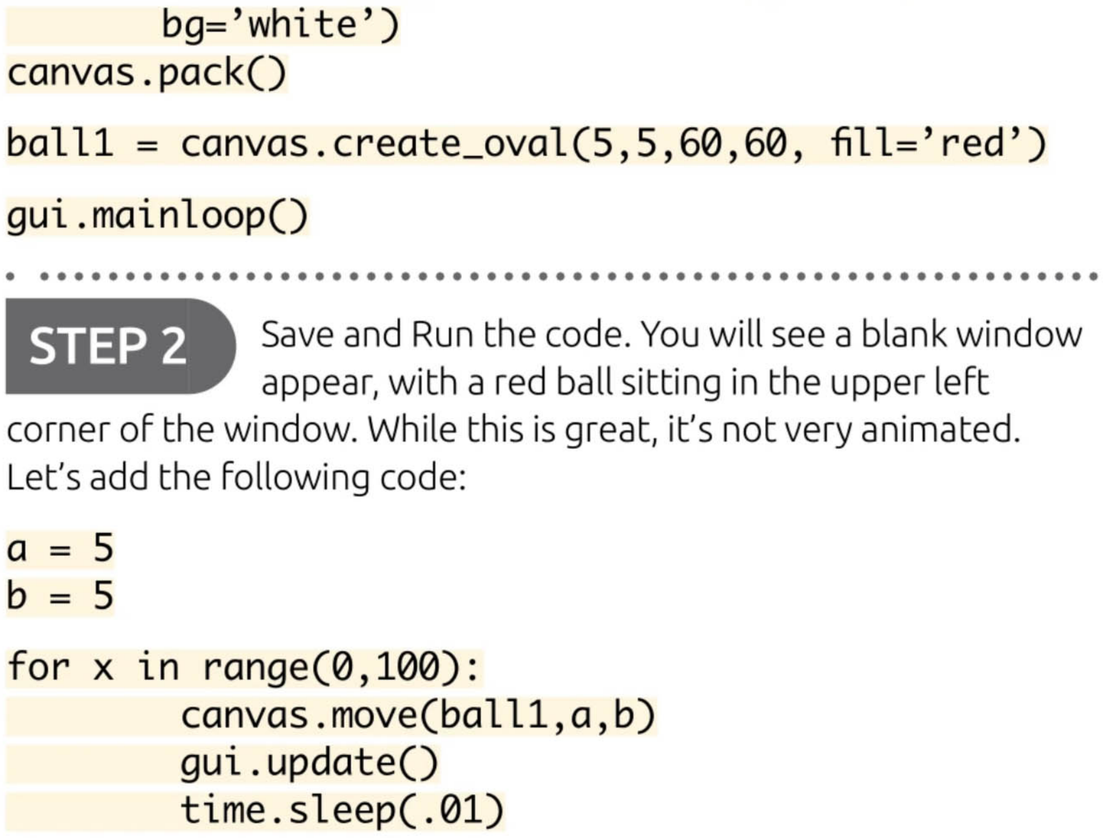
**The Tkinter module is an ideal starting point for learning animation within Python. Naturally, there are better custom modules out there, but Tkinter does the job well enough to get a grasp on what's needed.**

**STEP 1** Let's make a bouncing ball animation. First, we will need to create a canvas (window) and the ball to animate:

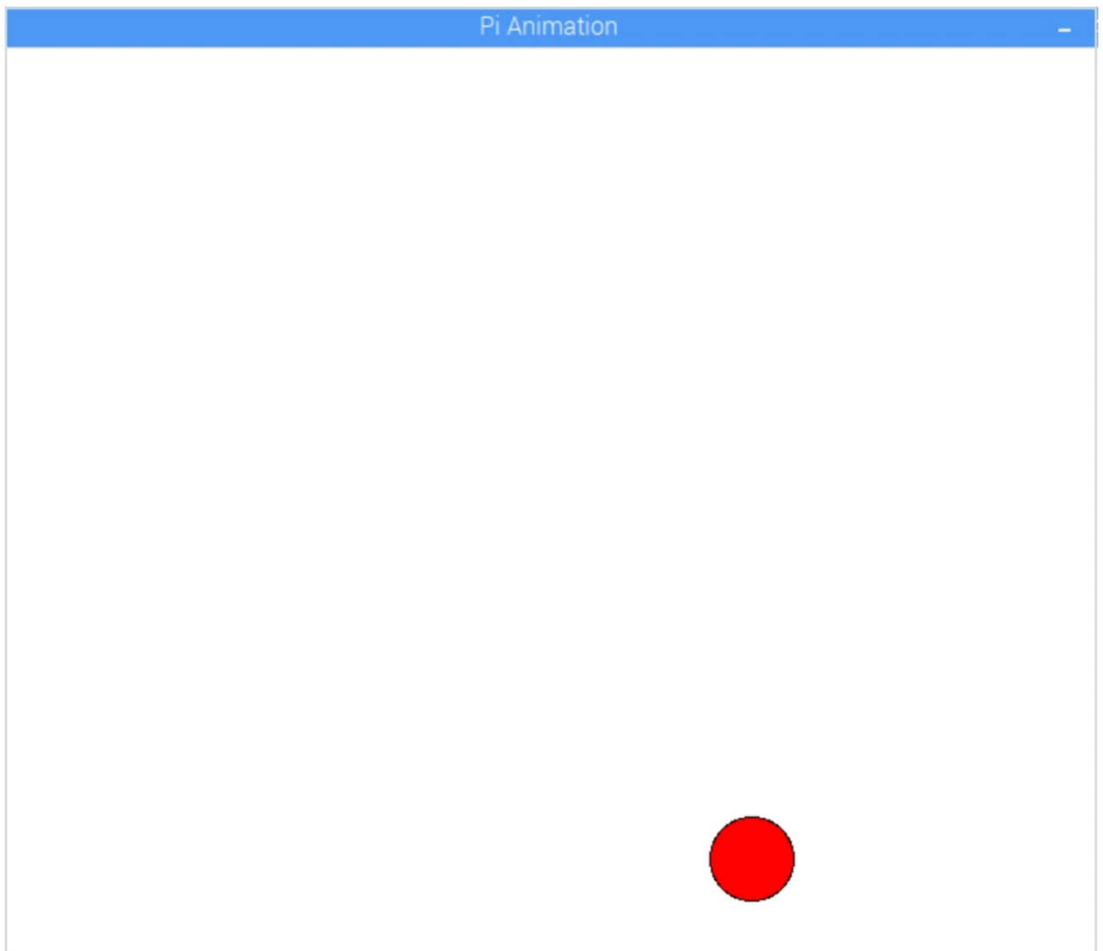
```
from tkinter import *
import time

gui = Tk()
gui.geometry("800x600")
gui.title("Pi Animation")
canvas = Canvas(gui, width=800,height=600,
        bg='white')
canvas.pack()

ball1 = canvas.create_oval(5,5,60,60, fill='red')

gui.mainloop()
```

**STEP 2** Save and Run the code. You will see a blank window appear, with a red ball sitting in the upper left corner of the window. While this is great, it's not very animated. Let's add the following code:

```
a = 5
b = 5

for x in range(0,100):
        canvas.move(ball1,a,b)
        gui.update()
        time.sleep(.01)
```




**STEP 3** Insert the new code between the `ball1 = canvas.create_oval(5,5,60,60, fill='red')` line and the gui.mainloop() line. Save it and Run. You will now see the ball move from the top left corner of the animation window, down to the bottom right corner. You can alter the speed in which the ball traverses the window by altering the `time.sleep(.01)` line. Try (.05).




**STEP 4** The `canvas.move(ball1,a,b)` line is the part that moves the ball from one corner to the other; obviously with both a and b equalling 5. We can change things around a bit already, such as the size and colour of the ball, with the line: `ball1 = canvas.create_oval(5,5,60,60, fill='red')` and we can change the values of a and b to something else.

```
ball1 = canvas.create_oval(7,7,60,60, fill='red')

a = 8
b = 3

for x in range(0,100):
    canvas.move(ball1,a,b)
    gui.update()
    time.sleep(.05)
```

## STEP 5

Let's see if we can animate the ball so that it bounces around the window until you close the program.

```
xa = 5
ya = 10

while True:
        canvas.move(ball1,xa,ya)
        pos=canvas.coords(ball1)
       if pos[3] >=600 or pos[1] <=0:
            ya = -ya
        if pos[2] >=800 or pos[0] <=0:
            xa = -xa
        gui.update()
       time.sleep(.025)
```

## STEP 6

Remove the code you entered in Step 2 and insert the code from Step 5 in its place; again, between the `ball1 = canvas.create_oval(5,5,60,60, fill='red')` and the `gui.mainloop()` lines. Save the code and Run it as normal. If you've entered the code correctly, then you will see the red ball bounce off the edges of the window until you close the program.

## STEP 7

The bouncing animation takes place within the `While True` loop. First, we have the values of xa and xy before the loop, both of 5 and 10. The `pos=canvas.coords(ball1)` line takes the value of the ball's location in the window. When it reaches the limits of the window, 800 or 600, it will make the values negative; moving the ball around the screen.

```
xa = 5
ya = 10

while True:
    canvas.move(ball1,xa,ya)
    pos=canvas.coords(ball1)
    if pos[3] >=600 or pos[1] <=0:
        ya = -ya
    if pos[2] >=800 or pos[0] <=0:
        xa =  xa
    gui.update()
    time.sleep(.025)
```

## STEP 8

Pygame, however, is a much better module at producing higher-end animations. Begin by creating a New File and entering:

```
import pygame
from random import randrange

MAX_STARS  = 250
STAR_SPEED = 2

def init_stars(screen):
  """ Create the starfield """
  global stars
  stars = []
  for i in range(MAX_STARS):
    # A star is represented as a list with this
format: [X,Y]
    star = [randrange(0,screen.get_width() - 1),
            randrange(0,screen.get_height() - 1)]
    stars.append(star)

def move_and_draw_stars(screen):
  """ Move and draw the stars """
  global stars
  for star in stars:
    star[1] += STAR_SPEED
    if star[1] >= screen.get_height():
      star[1] = 0
      star[0] = randrange(0,639)

    screen.set_at(star,(255,255,255))
```

## STEP 9

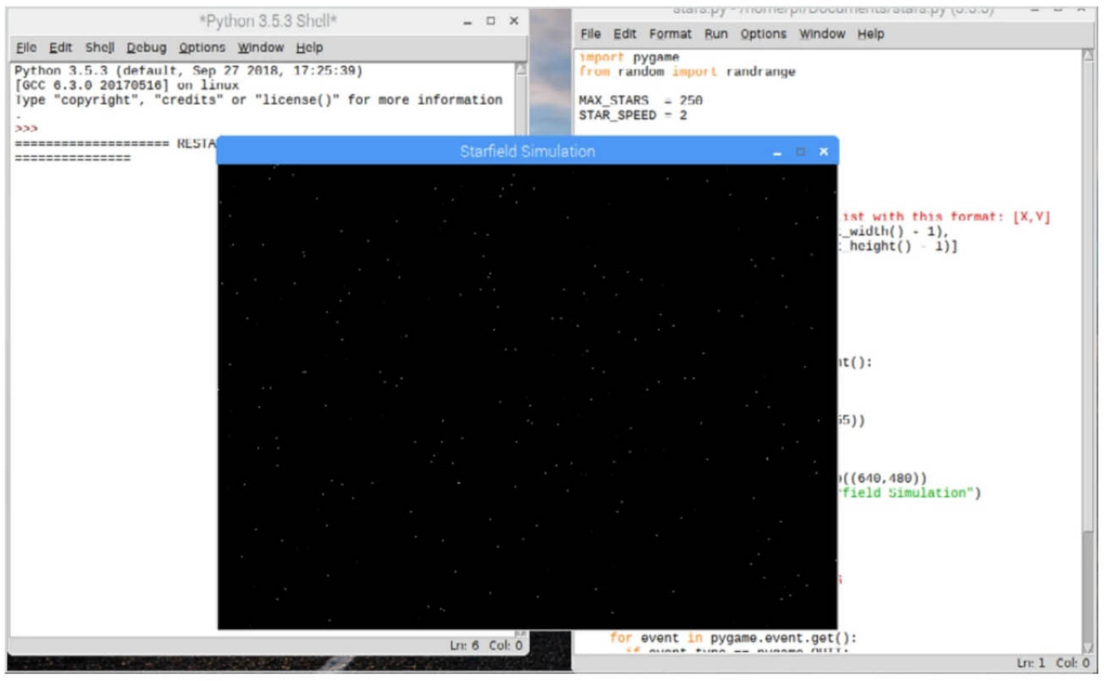
Now add the following:

```
def main():
  pygame.init()
  screen = pygame.display.set_mode((640,480))
  pygame.display.set_caption("Starfield
Simulation")
  clock = pygame.time.Clock()

  init_stars(screen)

  while True:
    # Lock the framerate at 50 FPS
    clock.tick(50)

    # Handle events
    for event in pygame.event.get():
      if event.type == pygame.QUIT:
        return

    screen.fill((0,0,0))
    move_and_draw_stars(screen)
    pygame.display.flip()

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

```
def main():
  pygame.init()
  screen = pygame.display.set_mode((640,480))
  pygame.display.set_caption("Starfield Simulation")
  clock = pygame.time.Clock()

  init_stars(screen)

  while True:
    # Lock the framerate at 50 FPS
    clock.tick(50)

    # Handle events
    for event in pygame.event.get():
      if event.type == pygame.QUIT:
        return

    screen.fill((0,0,0))
    move_and_draw_stars(screen)
    pygame.display.flip()

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

## STEP 10

Save and Run the code. You will agree that the simulated starfield code looks quite impressive. Imagine this as the beginning of some game code, or even the start to a presentation? Using a combination of Pygame and Tkinter, your Python animations will look fantastic.



# Create Your Own Modules

Large programs can be much easier to manage if you break them up into smaller parts and import the parts you need as modules. Learning to build your own modules also makes it easier to understand how they work.

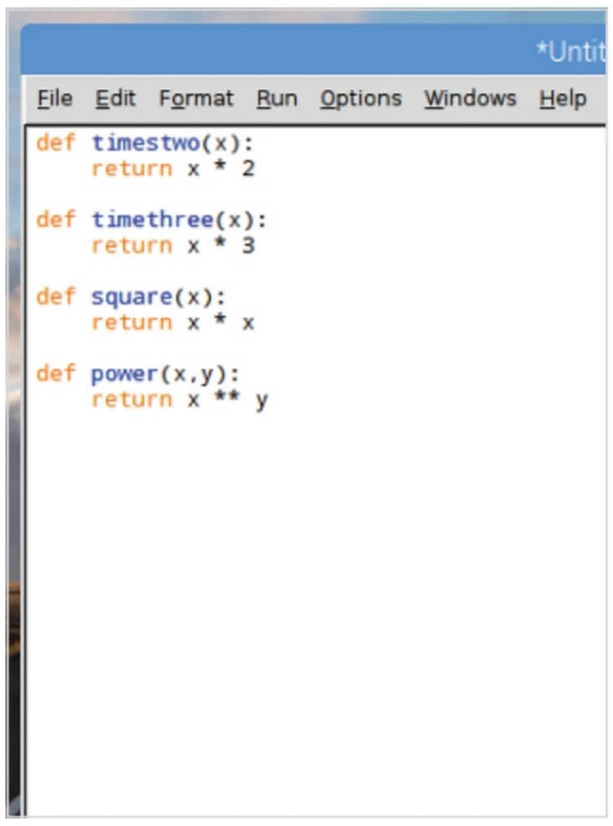
## BUILDING MODULES

**Modules are Python files, containing code, that you save using a .py extension. These are then imported into Python using the now familiar import command.**

### STEP 1

Let's start by creating a set of basic mathematics functions. Multiply a number by two, three and square or raise a number to an exponent (power). Create a New File in the IDLE and enter:

```
def timestwo(x):
        return x * 2

def timesthree(x):
        return x * 3

def square(x):
        return x * x

def power(x,y):
        return x ** y
```

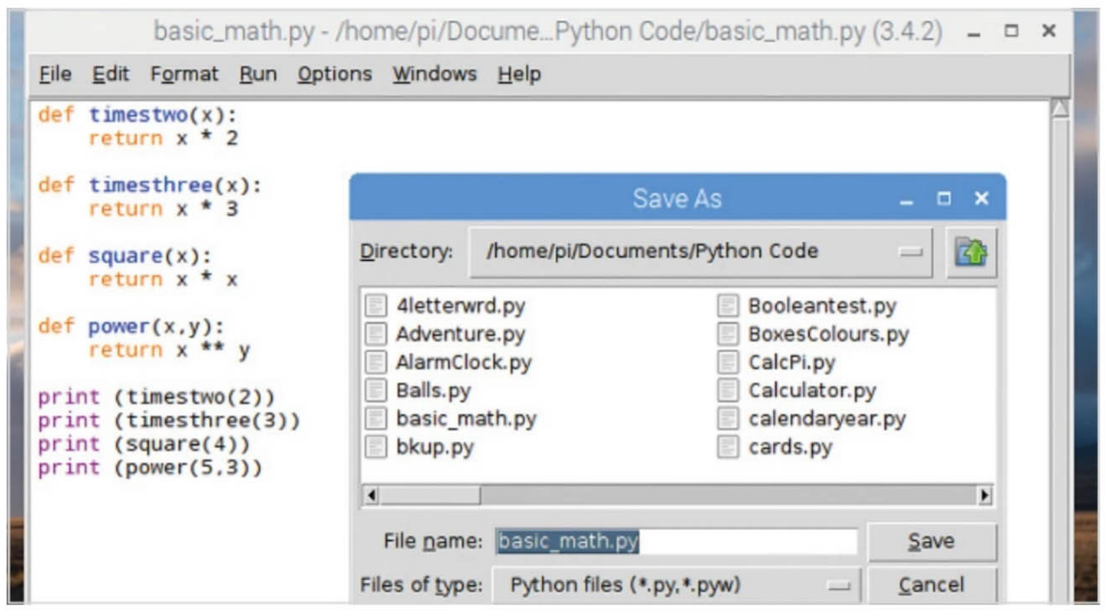



### STEP 2

Under the above code, enter functions to call the code:

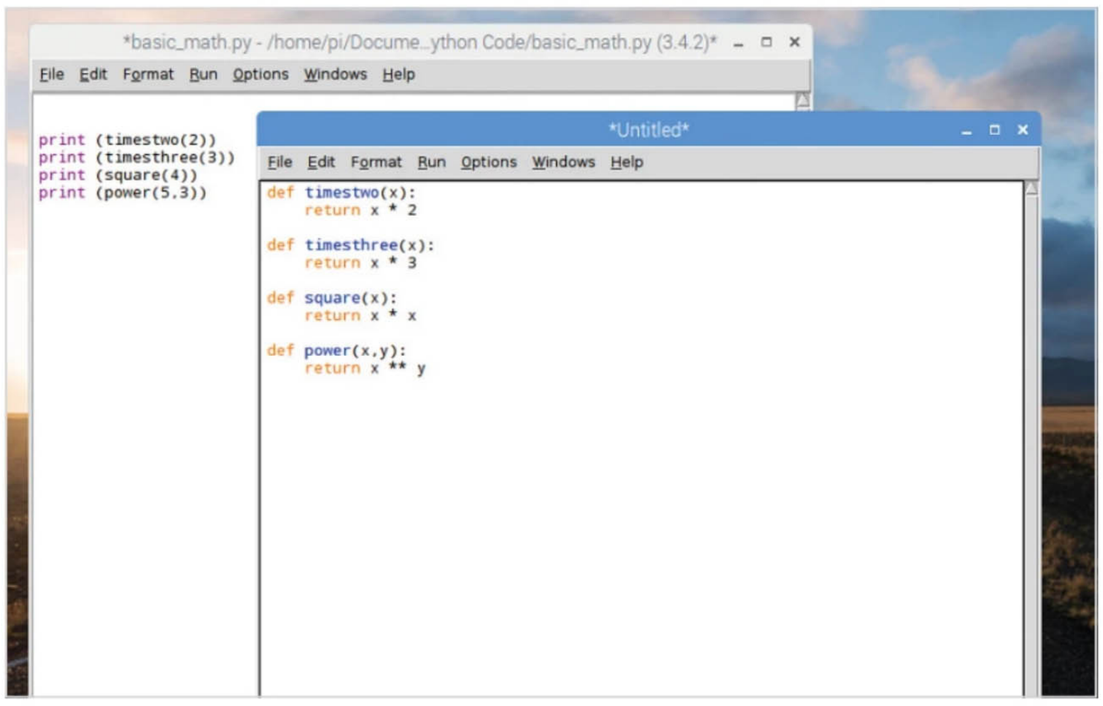
```
print (timestwo(2))
print (timesthree(3))
print (square(4))
print (power(5,3))
```

Save the program as basic_math.py and execute it to get the results.




### STEP 3

Now you're going to take the function definitions out of the program and into a separate file. Highlight the function definitions and choose Edit > Cut. Choose File > New File and use Edit > Paste in the new window. You now have two separate files, one with the function definitions, the other with the function calls.



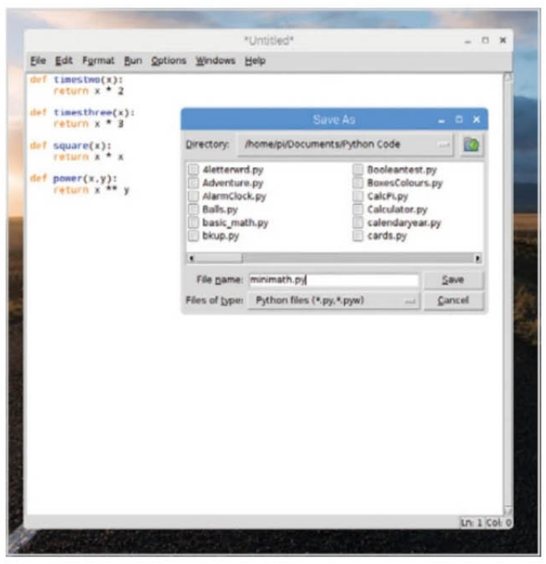

### STEP 4

If you now try and execute the basic_math.py code again, the error **'NameError: name 'timestwo' is not defined'** will be displayed. This is due to the code no longer having access to the function definitions.

```
Traceback (most recent call last):
  File "/home/pi/Documents/Python Code/basic_math.py", line 3, in <module>
    print (timestwo(2))
NameError: name 'timestwo' is not defined
>>>
```

### STEP 5

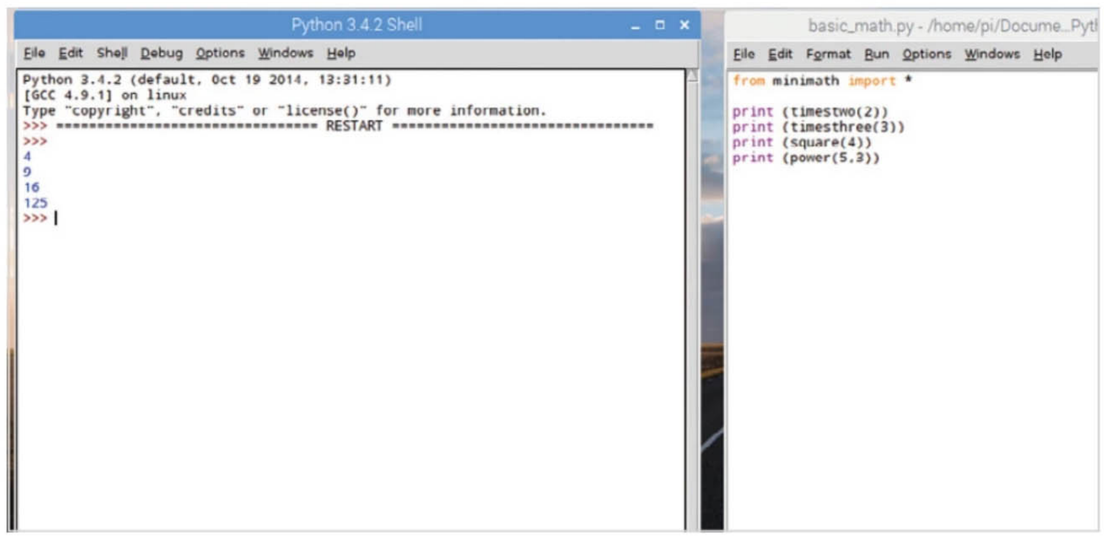
Return to the newly created window containing the function definitions, and click File > Save As. Name this **minimath.py** and save it in the same location as the original **basic_math.py** program. Now close the minimath.py window, so the basic_math.py window is left open.



**STEP 6** Back to the basic_math.py window: at the top of the code enter:

```
from minimath import *
```

This will import the function definitions as a module. Press F5 to save and execute the program to see it in action.



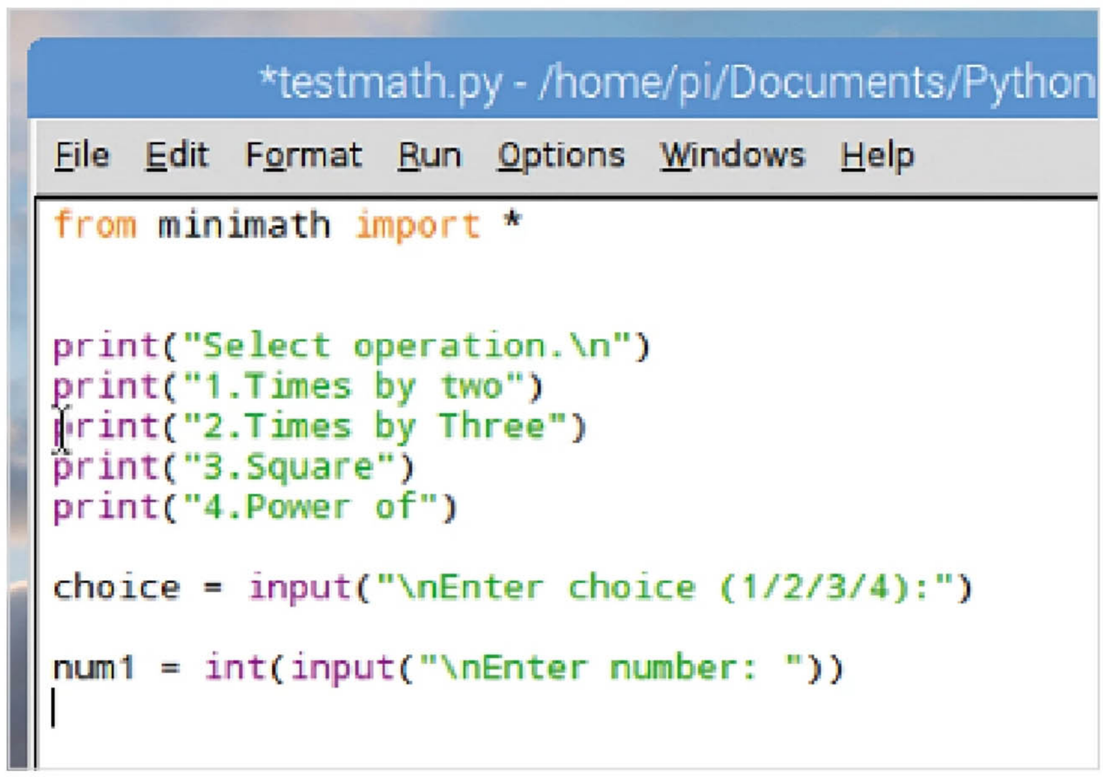

**STEP 7** You can now use the code further to make the program a little more advanced, utilising the newly created module to its full. Include some user interaction. Start by creating a basic menu the user can choose from:

```
print("Select operation.\n")
print("1.Times by two")
print("2.Times by Three")
print("3.Square")
print("4.Power of")

choice = input("\nEnter choice (1/2/3/4):")
```

**STEP 8** Now we can add the user input to get the number the code will work on:

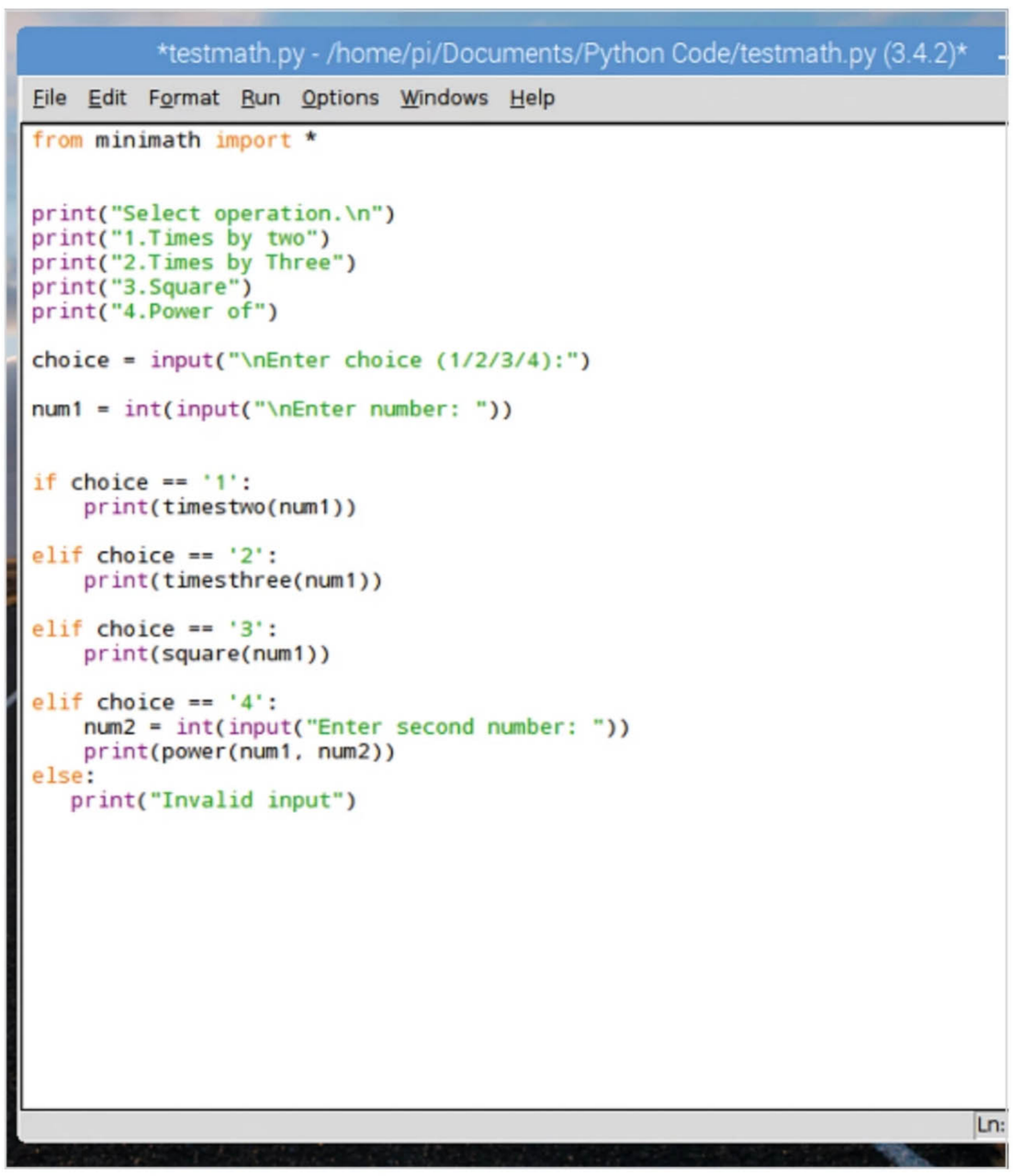
```
num1 = int(input("\nEnter number: "))
```

This will save the user-entered number as the variable num1.

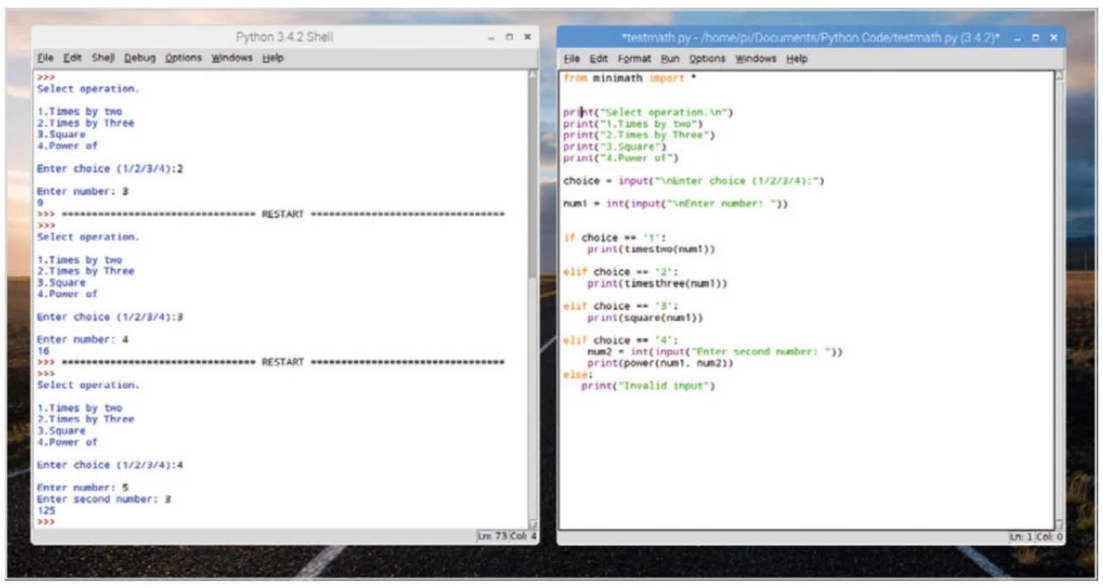



**STEP 9** Finally, you can now create a range of if statements to determine what to do with the number and utilise the newly created function definitions:

```
if choice == '1':
    print(timestwo(num1))

elif choice == '2':
    print(timesthree(num1))

elif choice == '3':
    print(square(num1))

elif choice == '4':
    num2 = int(input("Enter second number: "))
    print(power(num1, num2))
else:
   print("Invalid input")
```




**STEP 10** Note that for the last available options, the Power of choice, we've added a second variable, num2. This passes a second number through the function definition called power. Save and execute the program to see it in action.

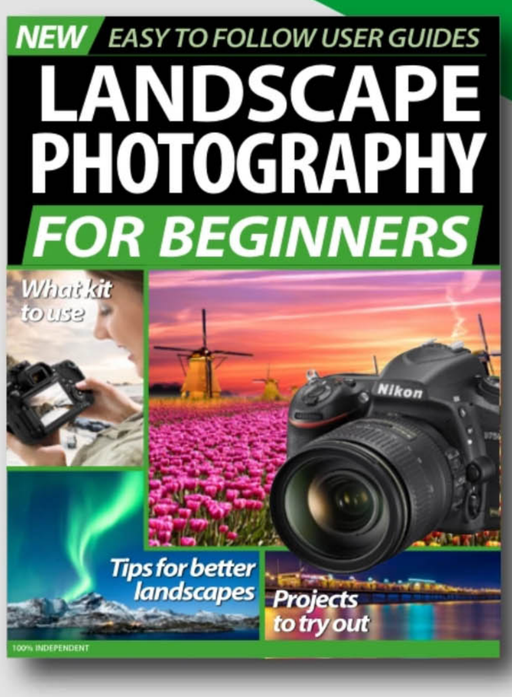


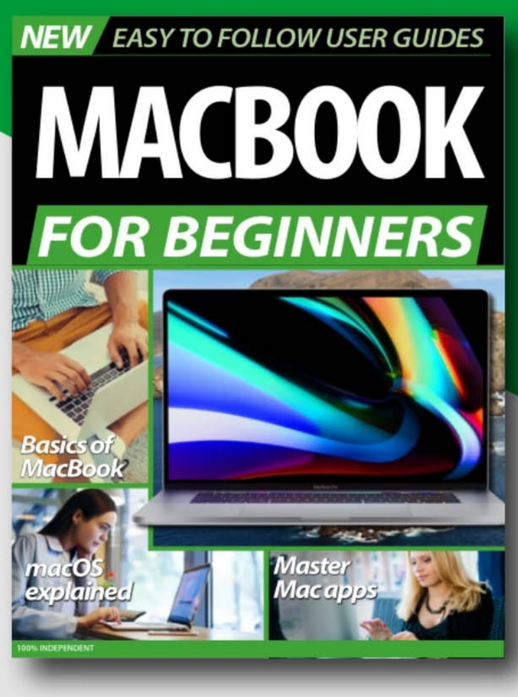
*Now you've got the basics down, you can improve and learn more essential skills in our next guide...*

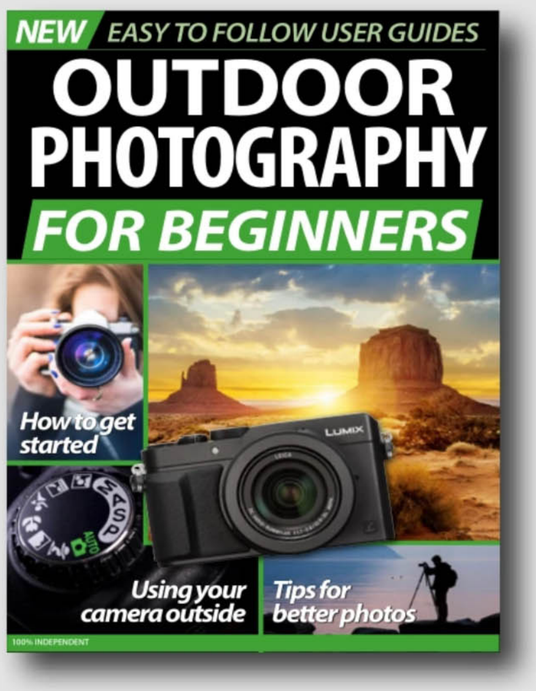
# PYTHON TRICKS AND TIPS

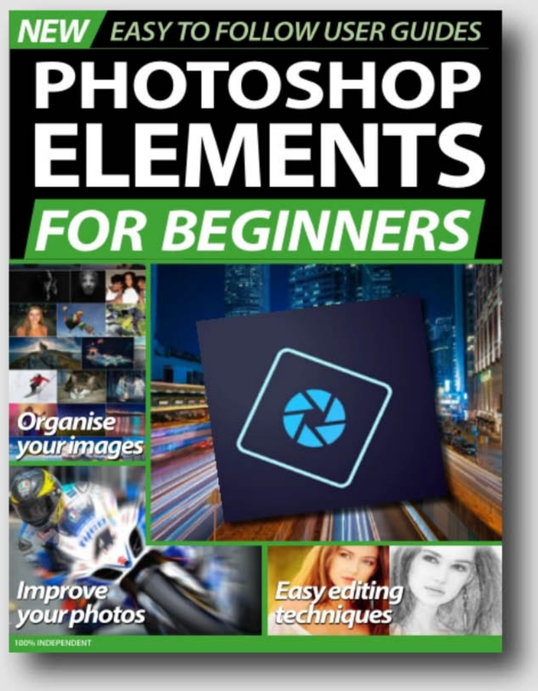
*Coming soon!*

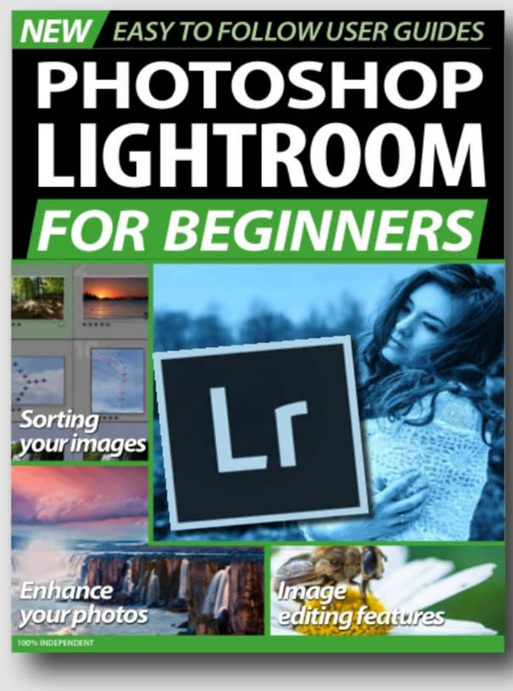
# Discover more of our guides...

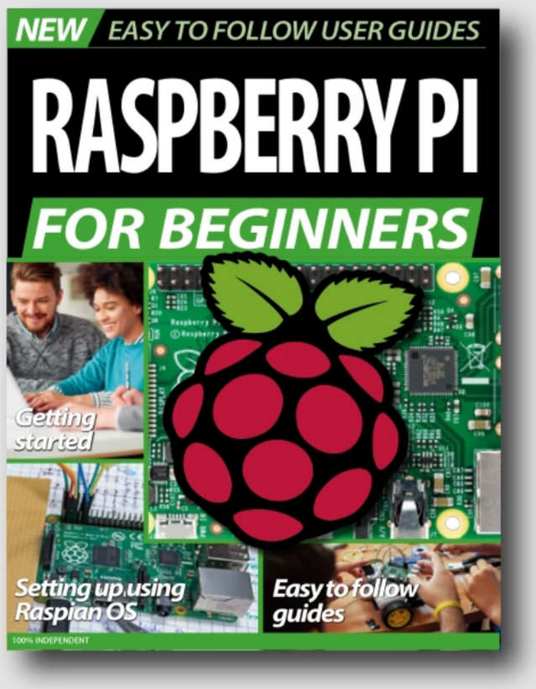




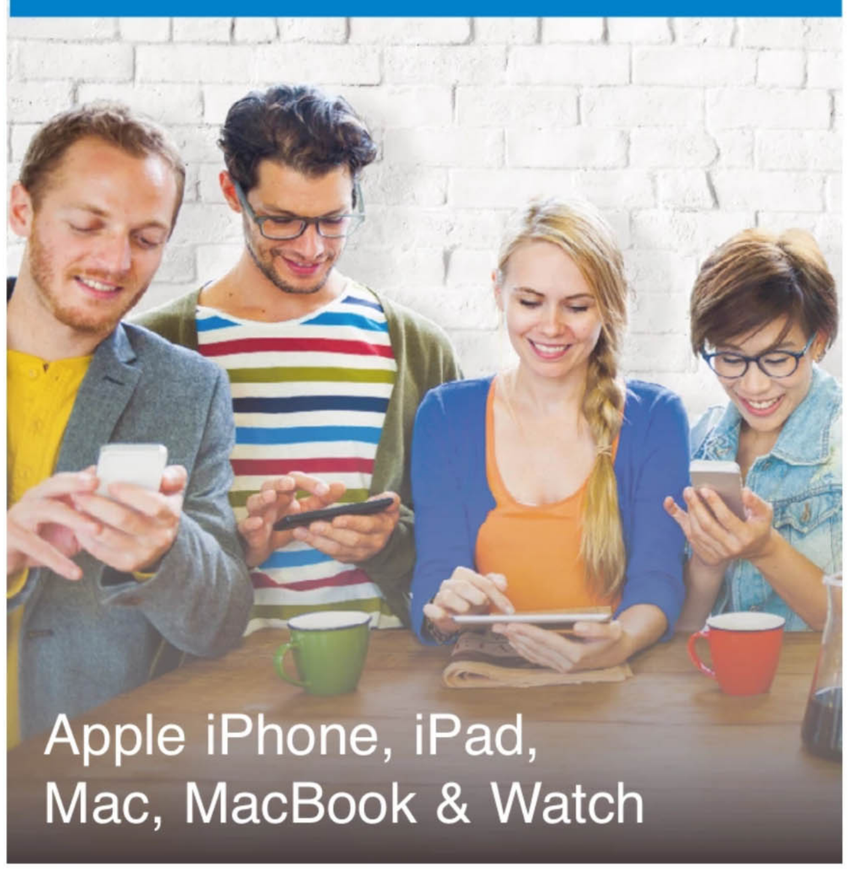
Black Dog Media

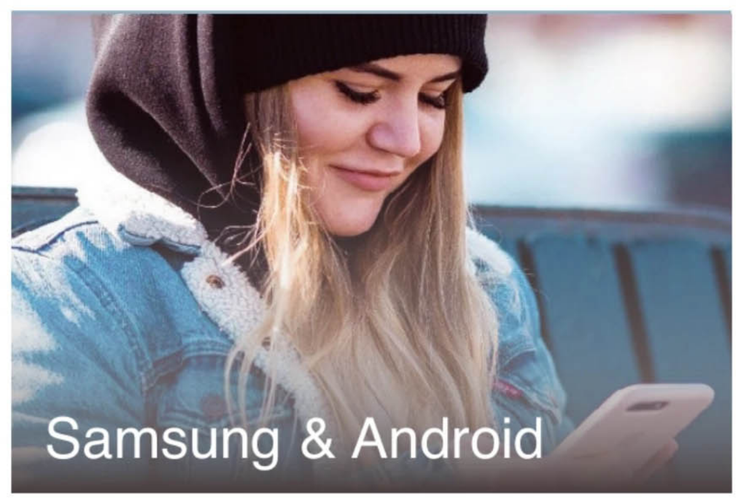
# Master Your Tech From Beginner to Expert

To continue learning more about your tech visit us at:

**www.bdmpublications.com**

## FREE Tech Guides





## EXCLUSIVE Offers on our Tech Guidebooks

- Print & digital editions
- Featuring the very latest updates
- Step-by-step tutorials and guides
- Created by BDM experts

*Check out our latest titles today!*

**bdmpublications.com/ultimate-photoshop**

Buy our Photoshop guides and download tutorial images for free!

*Simply sign-up and get creative.*

PLUS

## Special Deals and Bonus Content

Sign up to our monthly newsletter and get the latest updates, offers and news from BDM. We are here to help you Master Your Tech!