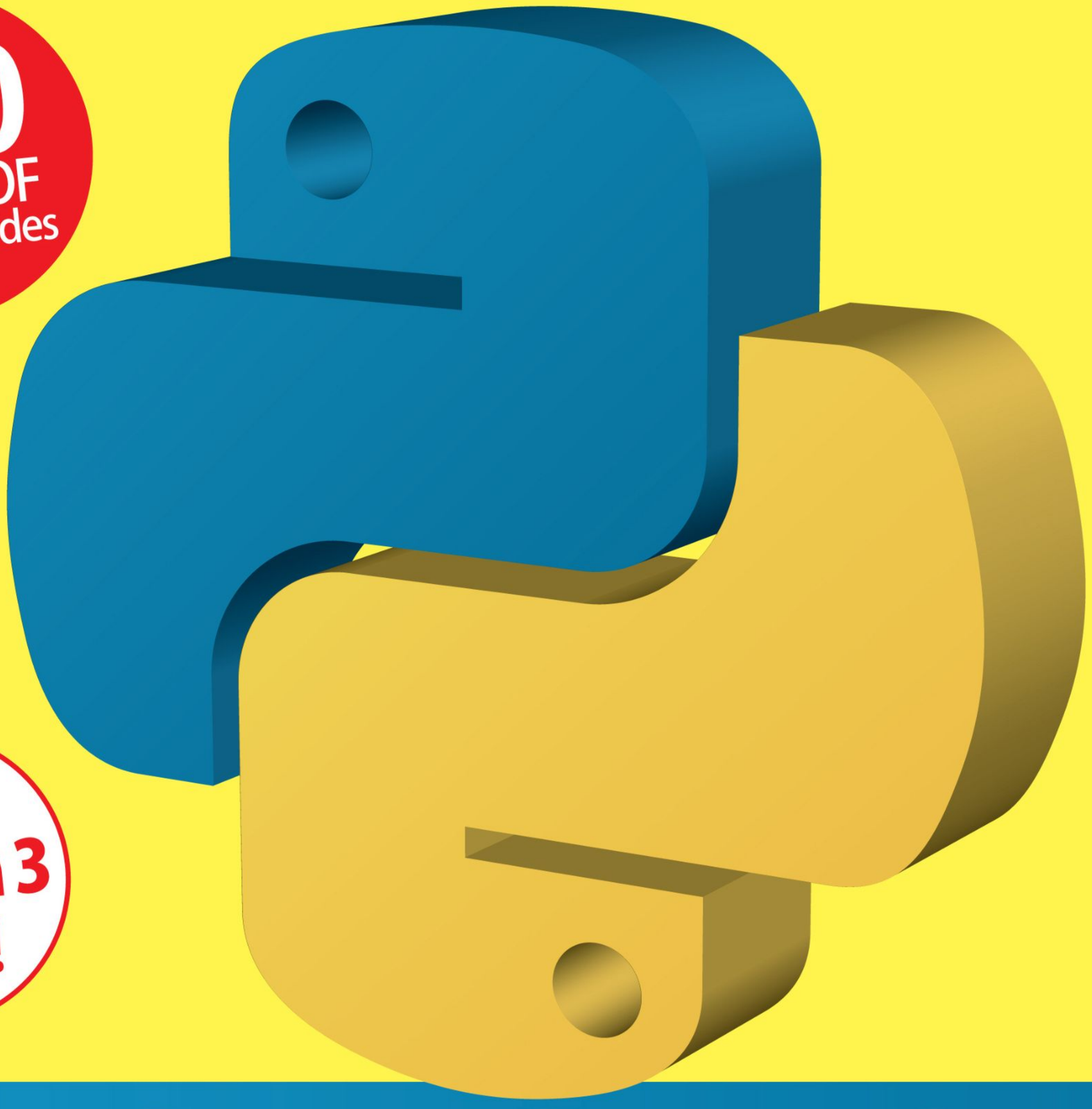
NEW Python 3 | Download over 60 Free Programs

BDM's Manual Series

The python™ Coding Manual

- ✓ *Step-by-step Guides*
- ✓ *Python 3*
- ✓ *Massive Code Repository*

Over 190 PAGES OF Python guides & tips



Learn Python 3 Now!

£12.99
Volume 20
US$24.99
Can$25.99
Aus$25.25
NZ$29.99

100% INDEPENDENT

9 772056 666004 20

- Packed with top tips and tutorials
- Learn the latest version of Python
- Featuring Python program listings
- Fantastic Python project ideas to use

Black Dog Media

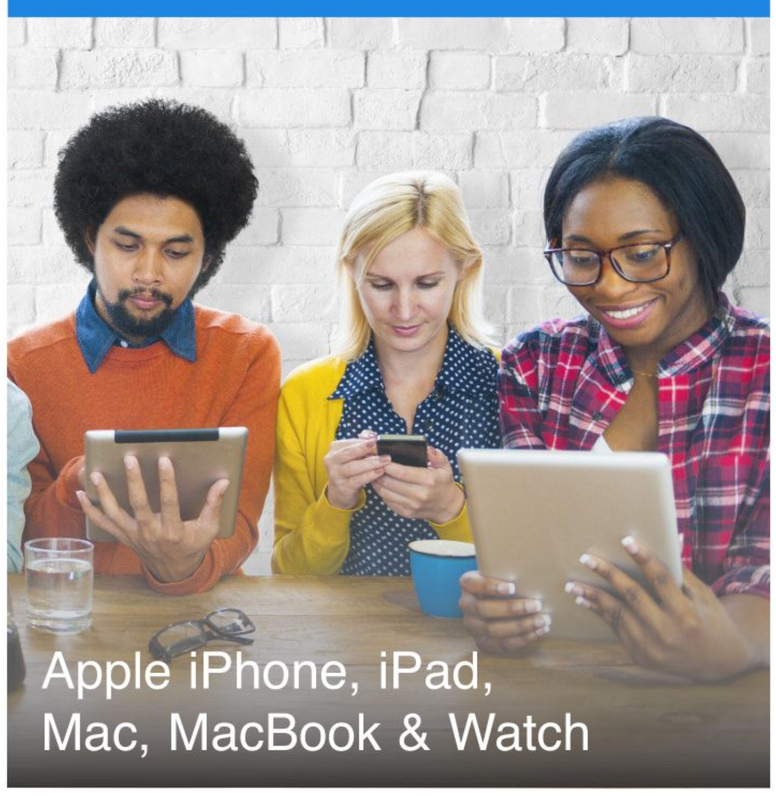
Master Your Tech

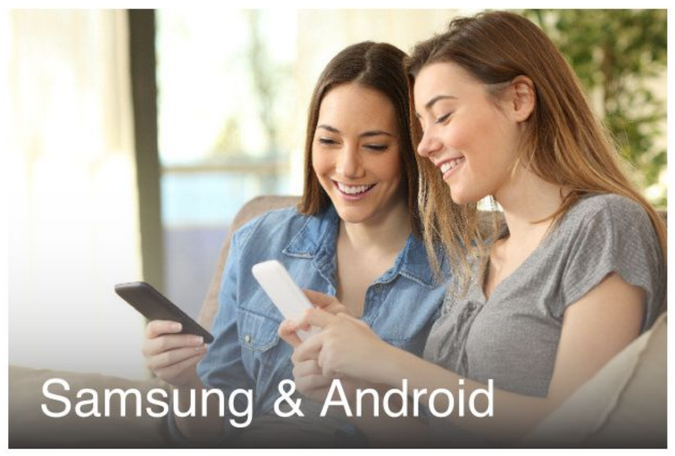
From Beginner to Expert

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!

BDM's Manual Series

The Python™ Coding Manual

The Only Python Manual You Need…

Anyone can learn how to code. All it takes is time, patience, and a good guide to help you get started. From there, you can take your coding experience as far as you like, but the most important factor is building a good foundation.

This manual aims to help you build a knowledge base around one of the best programming languages available: Python. Python is a fantastic programming language that has taken the world by storm. It's easy to use, powerful, useful, and fun. It works with other programming languages, and with all the major computer operating systems and platforms available today. It's the ideal choice for entry-level programmers who want to learn a new skill, or simply want to discover how they can create something on the computer that works for them, and it's used by data scientists and engineers who demand complexity and power.

Within these pages are the essential tutorials that will help you progress through the process of building your first piece of code, to tackling variables, numbers and expressions, user input, Python modules, and building and manipulating lists of data. We've even included a section called the Code Repository, covering a huge range of different types of Python programming, which you're free to delve into and use in your own unique programs. Learning Python is great fun, so what are you waiting for?

Let's get coding!

@bdmpubs BDM Publications www.bdmpublications.com

Contents

The Python Coding Manual

Check out BDM's Code Portal
60 FREE Python programs
21,500 lines of code
Visit: https://bdmpublications.com/code-portal

6

Print ("The World of Code")

8 Being a Programmer
10 A Brief History of Coding
12 Choosing a Programming Language
14 Creating a Coding Platform
16 Using Virtual Machines
18 Equipment You Will Need

20

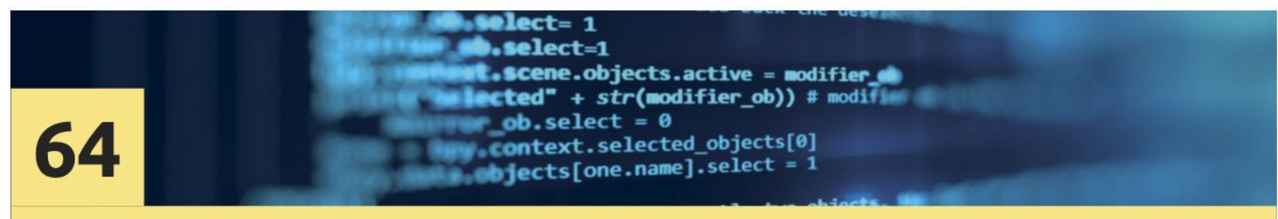
Welcome to Python

22 Why Python?
24 What can You Do with Python?
26 Python in Numbers
28 How to Set Up Python in Windows
30 How to Set Up Python in Linux
32 Python on the Pi
34 Getting to Know Python

36

First Steps into Python

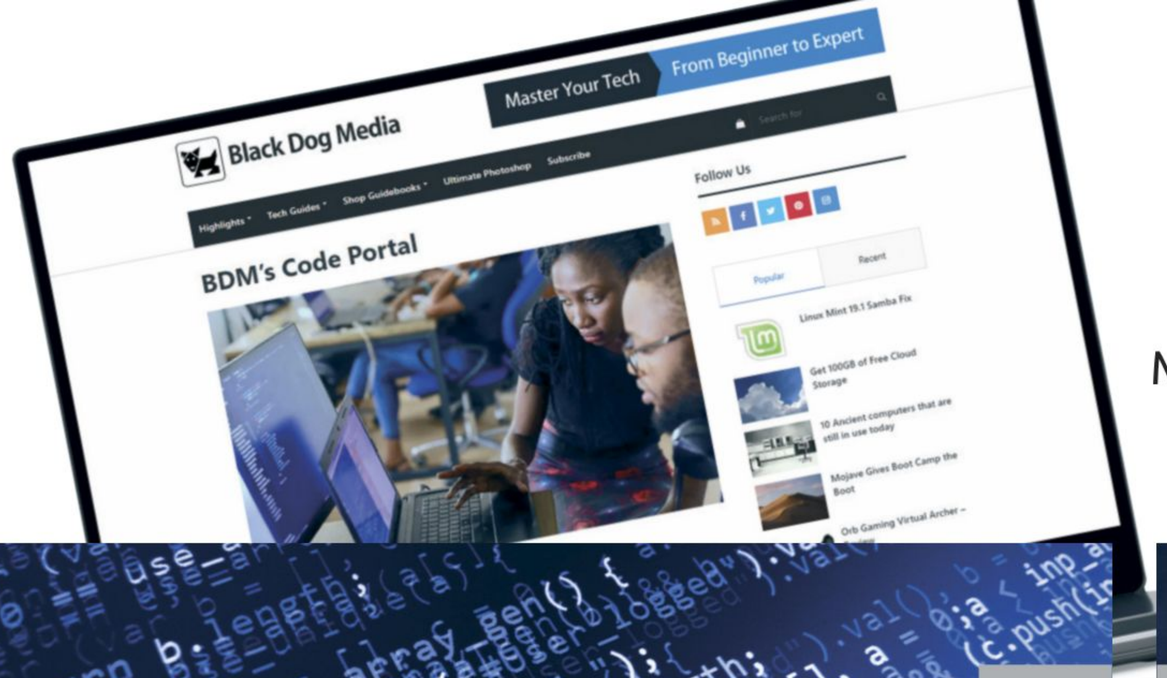
38 Starting Python for the First Time
40 Your First Code
42 Saving and Executing Your Code
44 Executing Code from the Command Line
46 Numbers and Expressions
48 Using Comments
50 Working with Variables
52 User Input
54 Creating Functions
56 Conditions and Loops
58 Python Modules
60 Python Errors
62 Combining What You Know So Far



64

Manipulating Data

66 Lists
68 Tuples
70 Dictionaries
72 Splitting and Joining Strings
74 Formatting Strings
76 Date and Time
78 Opening Files
80 Writing to Files
82 Exceptions
84 Python Graphics
86 Combining What You Know So Far



Contents

Master Python with the help of our fantastic Code Repository, featuring code for games, tools and more. **See page 148 for more details.**

88 Working with Modules

90 Calendar Module
92 OS Module
94 Using the Math Module
96 Random Module
98 Tkinter Module
100 Pygame Module
104 Basic Animation
106 Create Your Own Modules

108 Learning Linux

110 What is Linux?
112 Why Linux?
114 Using the Filesystem
116 Listing and Moving Files
118 Creating and Deleting Files
120 Create and Remove Directories
122 Copying, Moving and Renaming Files
124 Using the Man Pages
126 Editing Text Files
128 Getting to Know Users
130 Ownership and Permissions
132 Useful System and Disk Commands
134 Managing Programs and Processes
136 Input, Output and Pipes
138 Fun Things to Do in the Terminal
140 More Fun Things to Do in the Terminal
142 Linux Tips and Tricks
144 Command Line Quick Reference
146 A-Z of Linux Commands

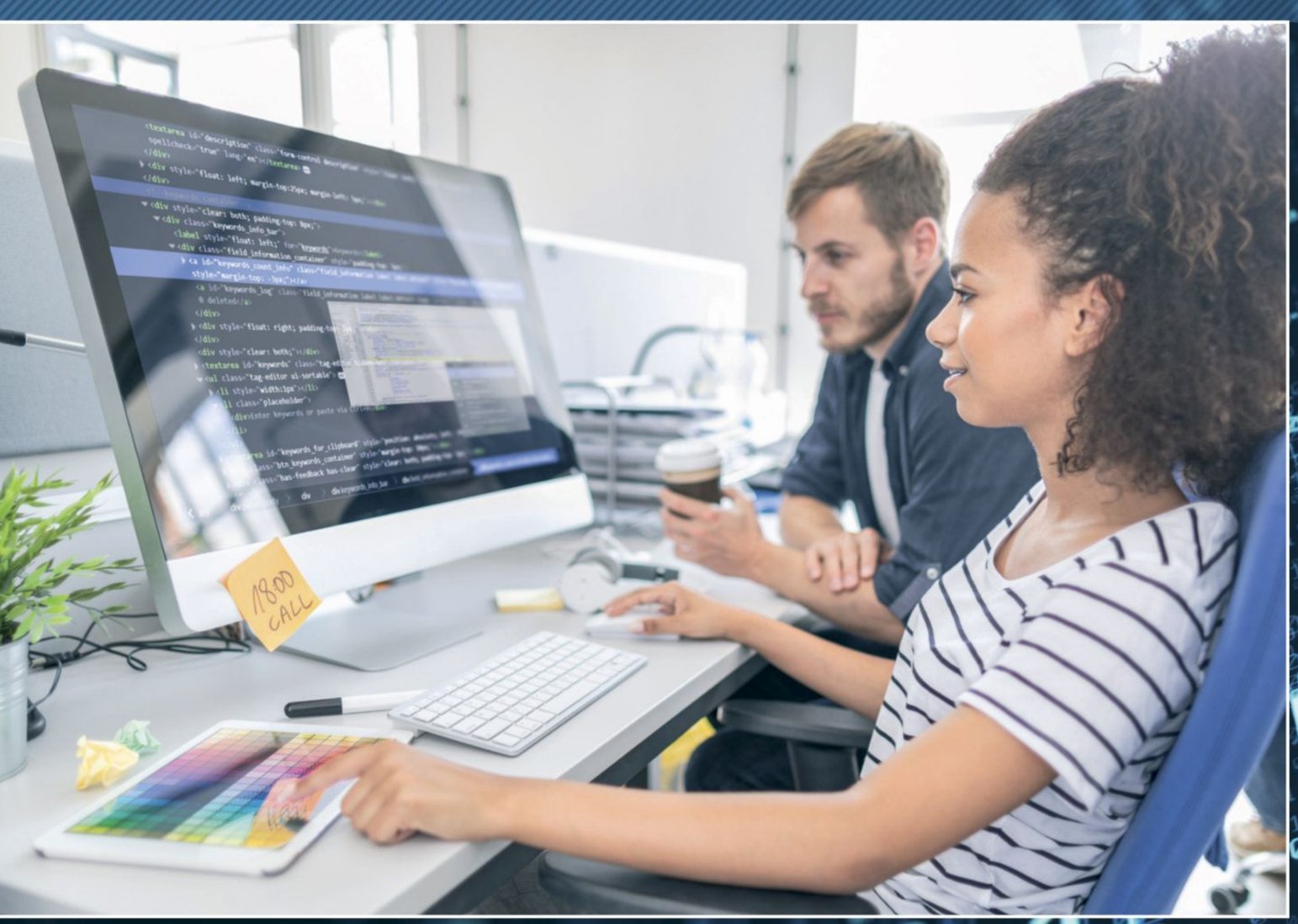
148 Code Repository

150 Python File Manager
152 Number Guessing Game
154 Polygon Circles
155 Random Number Generator
156 Random Password Generator
157 Keyboard Drawing Script
158 Pygame Text Examples
159 Google Search Script
160 Text to Binary Convertor
162 Text Adventure Script
164 Mouse Controlled Turtle
165 Python Alarm Clock
166 Vertically Scrolling Text
168 Python Digital Clock
170 Python Scrolling Ticker Script
171 Simple Python Calculator
172 Playing Music with the Winsound Module
174 Hangman Game Script

176 Coding Projects & Tips

178 Creating a Loading Screen
180 Text Animations
182 Tracking the ISS with Python
186 Using Text Files for Animation
188 Passing Variables to Python
190 Python Beginner's Mistakes
192 Glossary of Python Terms

Print("The World of Code")



Where to start learning how to code? Remarkably, this is the most difficult step. There are plenty of programming languages out there, and a seemingly unlimited number of tools to help you achieve what you want; but where do you begin?

In this section, we look at what you will need to take those first tentative steps into the world of coding. You won’t suddenly become a programmer in twenty-four hours, learning how to code properly takes time and patience. But with a little help, you can master the basics and start your coding journey.

8 Being a Programmer

10 A Brief History of Coding

12 Choosing a Programming Language

14 Creating a Coding Platform

16 Using Virtual Machines

18 Equipment You Will Need

Being a Programmer

Programmer, developer, coder, they’re all titles for the same occupation, someone who creates code. What they’re creating the code for can be anything from a video game to a critical element on-board the International Space Station. How do you become a programmer though?

Times have changed since programming in the ‘80s, but the core values still remain.

“It’s up to you how far to take your coding adventure!”

```
#include<stdio.h>
#include<dos.h>
#include<stdlib.h>
#include<conio.h>
void getup()
{
    textcolor(BLACK);
    textbackground(15);
    clrscr();
    window(10,2,70,3);
    cprintf("Press X to Exit, Press Space to Jump");
    window(62,2,80,3);
    cprintf("SCORE : ");
    window(1,25,80,25);
    for(int x=0;x<79;x++)
    cprintf("n");
    textcolor(0);
}

int t,speed=40;
void ds(int jump=0)
{
    static int a=1;

    if(jump==0)
        t=0;
    else if(jump==2)
        t--;
    else t++;
    window(2,15-t,18,25);
    cprintf("                ");
    cprintf("          лллллллм");
    cprintf("          ллллллллл");
    cprintf("          ллллллппп");
    cprintf(" л       мллллпппп ");
    cprintf(" ллм  мллллллммм ");
    cprintf(" пллллллллллл  п ");
    cprintf("   пллллллллп     ");
    if(jump==1 || jump==2){
    cprintf("    ллп пл       ");
    cprintf("    лм   лм      ");
    }else if(a==1)
    {
    cprintf("    пллп  ппп    ");
    cprintf("      лм         ");
    a=2;
    }
    else if(a==2)
    {
    cprintf("     плм пл      ");
    cprintf("          лм     ");
    a=1;
    }
    cprintf("                ");
    delay(speed);
}
void obj()
{
```

Being able to follow a logical pattern and see an end result is one of the most valued skills of a programmer.

MORE THAN CODE

For those of you old enough to remember the ‘80s, the golden era of home computing, the world of computing was a very different scene to how it is today. 8-bit computers that you could purchase as a whole, as opposed to being in kit form and you having to solder the parts together, were the stuff of dreams; and getting your hands on one was sheer bliss contained within a large plastic box. However, it wasn’t so much the new technology that computers then offered, moreover it was the fact that for the first time ever, you could control what was being viewed on the ‘television’.

Instead of simply playing one of the thousands of games available at the time, many users decided they wanted to create their own content, their own games; or simply something that could help them with their homework or home finances. The simplicity of the 8-bit home computer meant that creating something from a few lines of BASIC code was achievable and so the first generation of home-bred programmer was born.

From that point on, programming expanded exponentially. It wasn’t long before the bedroom coder was a thing of the past and huge teams of designers, coders, artists and musicians were involved in making a single game. This of course led to the programmer becoming more than simply someone who could fashion a sprite on the screen and make it move at the press of a key.

Naturally, time has moved on and with it the technology that we use. However, the fundamentals of programming remain the same; but what exactly does it take to be a programmer?

The single most common trait of any programmer, regardless of what they’re doing, is the ability to see a logical pattern. By this we mean someone who can logically follow something from start to finish and envisage the intended outcome. While you may not feel you’re such a person, it is possible to train your brain into this way of thinking. Yes, it takes time but once you start to think in this particular way you will be able to construct and follow code.

Second to logic is an understanding of mathematics. You don’t have to be at a genius level but you do need to understand the rudiments of maths. Maths is all about being able to solve a problem and code mostly falls under the umbrella of mathematics.

Being able to see the big picture is certainly beneficial for the modern programmer. Undoubtedly, as a programmer, you will be part of a team of other programmers, and more than likely part of an even bigger team of designers, all of whom are creating a final product. While you may only be expected to create a small element of that final product, being able to understand what everyone else is doing will help you create something that’s ultimately better than simply being locked in your own coding cubicle.

Finally, there’s also a level of creativity needed to be a good programmer. Again though, you don’t need to be a creative genius, just have the imagination to be able to see the end product and how the user will interact with it.

There is of course a lot more involved in being a programmer, including learning the actual code itself. However, with time, patience and the determination to learn, anyone can become a programmer. Whether you want to be part of a triple-A video game team or simply create an automated routine to make your computing life easier, it’s up to you how far to take your coding adventure!

A Brief History of Coding

It’s easy to think that programming a machine to automate a process or calculate a value is a modern concept that’s only really happened in the last fifty years or so. However, that assumption is quite wrong, coding has actually been around for quite some time.

01000011 01101111 01100100 01100101

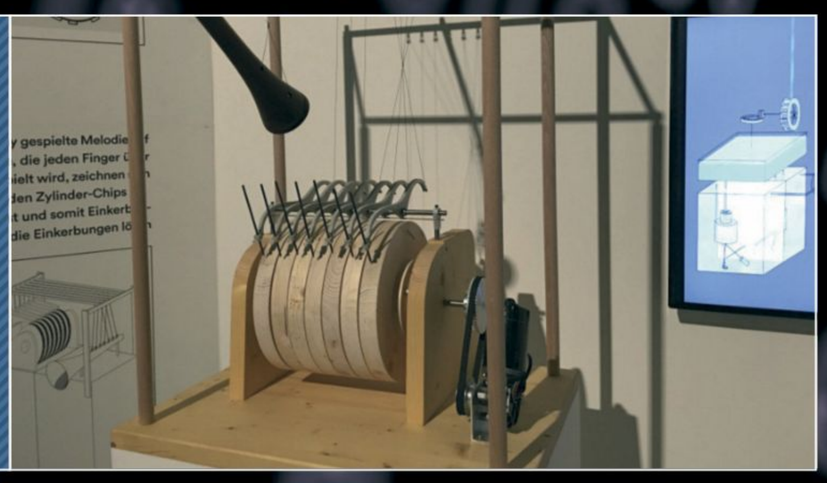
Essentially all forms of coding are made up of ones and zeros, on or off states. This works for a modern computer and even the oldest known computational device.

~87 BC

It’s difficult to pinpoint an exact start of when humans began to ‘program’ a device. However, it’s widely accepted that the Antikythera Mechanism is possibly the first ‘coded’ artefact. It’s dated to about 87 BC and is an ancient Greek analogue computer and orrery used to predict astronomical positions.

~850 AD

The Banū Mūsā brothers, three Persian scholars who worked in the House of Wisdom in Baghdad, published the Book of Ingenious Devices in around 850 AD. Among the inventions listed was a mechanical musical instrument: a hydro-powered organ that played interchangeable cylinders automatically.



1800

Joseph Marie Jacquard invents a programmable loom, which used cards with punched holes to create the textile design. However, it is thought that he based his design on a previous automated weaving process from 1725, by Basile Bouchon.

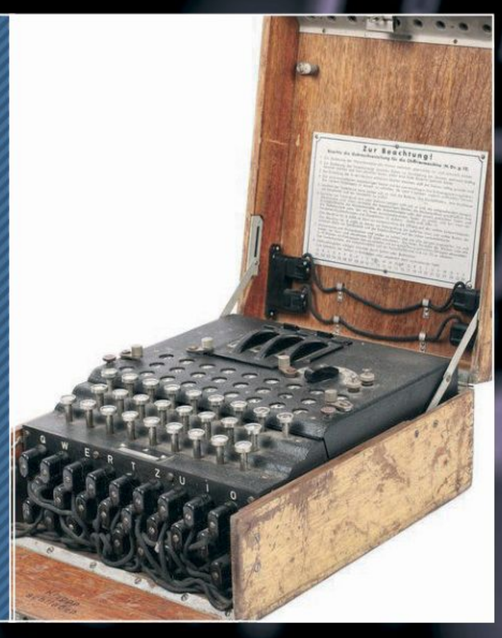
1842-1843

Ada Lovelace translated the memoirs of the Italian mathematician, Francis Maneclang, regarding Charles Babbage’s Analytical Engine. She made copious notes within her writing, detailing a method of calculating Bernoulli Numbers using the engine. This is recognised as the first computer program. Not bad, considering there were no computers available at the time.

1930-1950

1951-1958

During the Second World War, significant advances were made in programmable machines. Most notably, the cryptographic machines used to decipher military codes used by the Nazis. The Enigma was invented by the German engineer Arthur Scherbius, but was made famous by Alan Turing at Bletchley Park's codebreaking centre.




```
MONITOR FOR 6802 1.4          9-14-80  TSC ASSEMBLER  PAGE    2


C000                    ORG     ROM+$0000 BEGIN MONITOR
C000 8E 00 70  START    LDS     #STACK

                        ****************************************
                        * FUNCTION: INITA - Initialize ACIA
                        * INPUT: none
                        * OUTPUT: none
                        * CALLS: none
                        * DESTROYS: acc A

0013                    RESETA  EQU     %00010011
0011                    CTLREG  EQU     %00010001

C003 86 13      INITA   LDA A   #RESETA   RESET ACIA
C005 B7 80 04           STA A   ACIA
C008 86 11              LDA A   #CTLREG   SET 8 BITS AND 2 STOP
C00A B7 80 04           STA A   ACIA

C00D 7E C0 F1           JMP     SIGNON    GO TO START OF MONITOR

                        ****************************************
                        * FUNCTION: INCH - Input character
                        * INPUT: none
                        * OUTPUT: char in acc A
                        * DESTROYS: acc A
                        * CALLS: none
                        * DESCRIPTION: Gets 1 character from terminal

C010 B6 80 04   INCH    LDA A   ACIA      GET STATUS
C013 47                 ASR A             SHIFT RDRF FLAG INTO CARRY
C014 24 FA              BCC     INCH      RECIEVE NOT READY
C016 B6 80 05           LDA A   ACIA+1    GET CHAR
C019 84 7F              AND A   #$7F      MASK PARITY
C01B 7E C0 79           JMP     OUTCH     ECHO & RTS

                        ****************************************
                        * FUNCTION: INHEX - INPUT HEX DIGIT
                        * INPUT: none
                        * OUTPUT: Digit in acc A
                        * CALLS: INCH
                        * DESTROYS: acc A
                        * Returns to monitor if not HEX input

C01E 8D F0      INHEX   BSR     INCH      GET A CHAR
C020 81 30              CMP A   #'0       ZERO
C022 2B 11              BMI     HEXERR    NOT HEX
C024 81 39              CMP A   #'9       NINE
C026 2F 0A              BLE     HEXRTS    GOOD HEX
C028 81 41              CMP A   #'A
C02A 2B 09              BMI     HEXERR    NOT HEX
C02C 81 46              CMP A   #'F
C02E 2E 05              BGT     HEXERR
C030 80 07              SUB A   #7        FIX A-F
C032 84 0F      HEXRTS  AND A   #$0F      CONVERT ASCII TO DIGIT
C034 39                 RTS

C035 7E C0 AF   HEXERR  JMP     CTRL      RETURN TO CONTROL LOOP
```

The first true computer code was Assembly Language (ASM) or Regional Assembly Language. ASM was specific to the architecture of the machine on which it was being used. In 1951, programming languages fell under the generic term Autocode. Soon languages such as IPL, FORTRAN and ALGOL 58 were developed.

1959

Admiral Grace Hopper was part of the team that developed the UNIVAC I computer and she eventually developed a compiler for it. In time, the compiler she developed became COBOL (Common Business-oriented Language), a computer language that's still in use today.

1960-1970

Computer programming was mainly utilised by universities, the military and big corporations during the '60s and the '70s. A notable step toward a more user-friendly, or home user, language was the development of BASIC (Beginners All-purpose Symbolic Instruction Code) in the mid-sixties.

```
10 INPUT "What is your name: "; U$
20 PRINT "Hello "; U$
25 REM
30 INPUT "How many stars do you want: "; N
35 S$ = ""
40 FOR I = 1 TO N
50 S$ = S$ + "*"
55 NEXT I
60 PRINT S$
65 REM
70 INPUT "Do you want more stars? "; A$
80 IF LEN(A$) = 0 THEN GOTO 70
90 A$ = LEFT$(A$, 1)
100 IF (A$ = "Y") OR (A$ = "y") THEN GOTO 30
110 PRINT "Goodbye ";
120 FOR I = 1 TO 200
130 PRINT U$; " ";
140 NEXT I
150 PRINT
```

1970-1985

From the 1970s, the development of the likes of C, SQL, C with Classes (C++), MATLAB, Common Lisp and more, came to the fore. The '80s was undoubtedly the golden age of the home computer, a time when silicon processors were cheap enough for ordinary folk to buy. This led to a boom in home/bedroom coders with the rise of 8-bit machines.

1990s-Present Day

The Internet age brought a wealth of new programming languages and allowed people access to the tools and knowledge needed to learn coding in a better way. Not only could a user learn how to code, they could also freely share their code and source other code to improve their own.

Choosing a Programming Language

It would be impossible to properly explain every programming language in a single book of this size. New languages and ways in which to ‘talk’ to a computer or device and set it instructions are being invented almost daily; and with the onset of quantum computing, even more complex methods are being born. Here is a list of the more common languages along with their key features.

SQL

SQL stands for Structured Query Language. SQL is a standard language for accessing and manipulating databases. Although SQL is an ANSI (American National Standards Institute) standard, there are different versions of the SQL language. However, to be compliant, they all support at least the major commands such as Select, Update and Delete in a similar manner.

JAVASCRIPT

JavaScript (often shortened to JS) is a lightweight, interpreted, object-oriented language with first class functions. JavaScript runs on the client side of the web, that can be used to design or program how the web pages behave on the occurrence of an event. JavaScript is an easy to learn and also powerful scripting language, widely used for controlling web page behaviour.

JAVA

Java is the foundation for virtually every type of networked application and is the global standard for developing enterprise software, web-based content, games and mobile apps. The two main components of the Java platform are the Java Application Programming Interface (API) and the Java Virtual Machine (JVM) that translates Java code into machine language.

C#

C# is an elegant object-oriented language that enables developers to build a variety of secure and robust applications that run on the .NET Framework. You can use C# to create Windows client applications, XML Web services, client server applications, database applications and much more. The curly-brace syntax of C# will be instantly recognisable to anyone familiar with C, C++ or Java.

PYTHON

Python is a widely used high level programming language used for general purpose programming, created by Guido van Rossum and first released in 1991. An interpreted language, Python has a design philosophy that emphasises code readability and a syntax that allows programmers to express concepts in fewer lines of code. This can make it easier for new programmers to learn.

C++

C++ (pronounced cee plus plus) is a general purpose programming language. It has imperative, object-oriented and generic programming features. It was designed with a bias toward system programming and embedded, resource-constrained and large systems, with performance, efficiency and flexibility of use as its design highlights.

RUBY

Ruby is a language of careful balance. Its creator, Yukihiro “Matz” Matsumoto, blended parts of his favourite languages (Perl, Smalltalk, Eiffel, Ada and Lisp) to form a new language. From its release in 1995, Ruby has drawn devoted coders worldwide. Ruby is seen as a flexible language; essential parts of Ruby can be removed or redefined, at will. Existing parts can be added to.

PERL

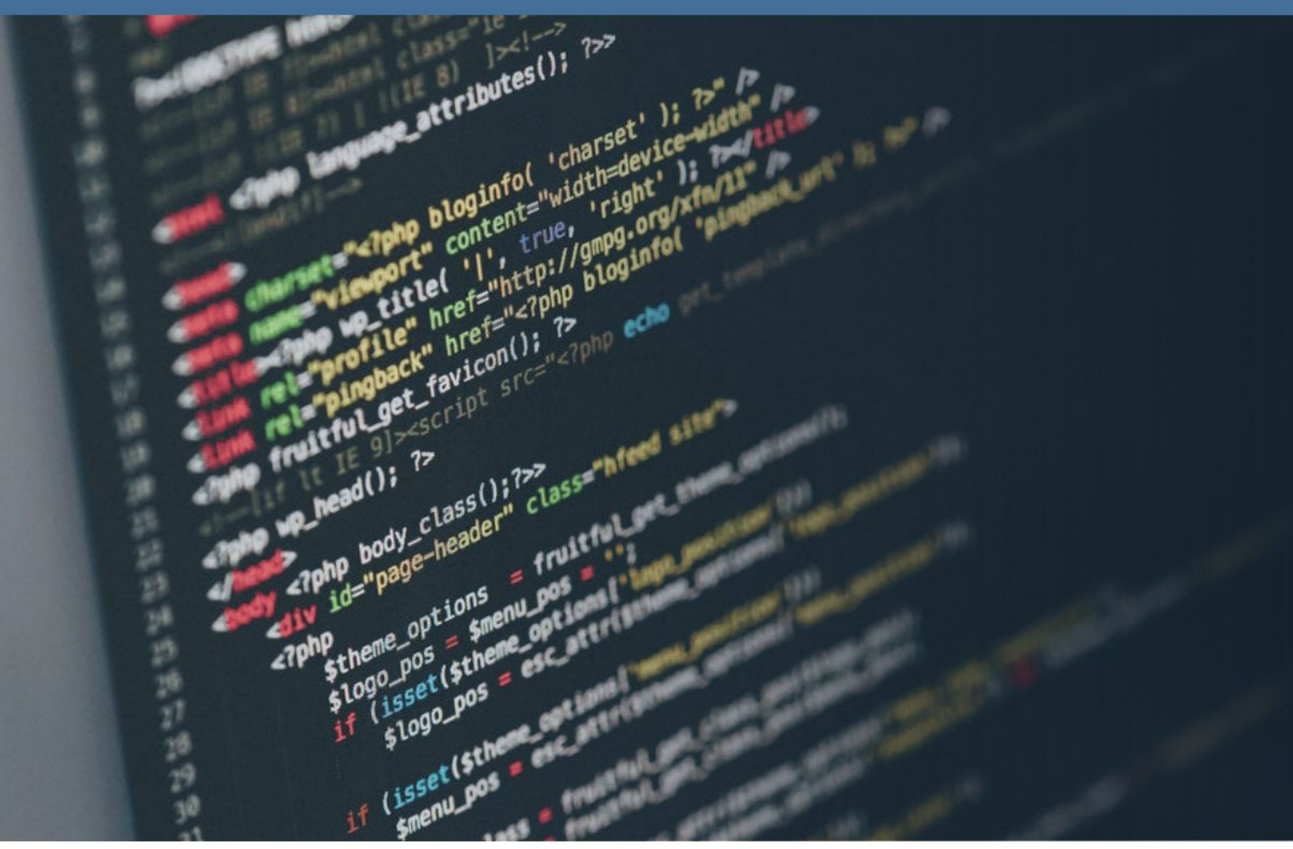
Perl is a general purpose programming language, used for a wide range of tasks including system administration, web development, network programming, GUI development and more. Its major features are that it’s easy to use, supports both procedural and object-oriented (OO) programming, has powerful built-in support for text processing and has one of the most impressive collections of third-party modules.

SWIFT

Swift is a powerful and intuitive programming language for macOS, iOS, watchOS and tvOS. Writing Swift code is interactive and fun; the syntax is concise yet expressive and Swift includes modern features that developers love. Swift code is safe by design, yet also produces software that runs lightning fast. A coding tutorial app, Swift Playgrounds, is available on iPad.

Creating a Coding Platform

The term ‘Coding Platform’ can denote a type of hardware, on which you can code, or a particular operating system, or even a custom environment that’s pre-built and designed to allow the easy creation of games. In truth it’s quite a loose term, as a Coding Platform can be a mixture of all these ingredients, it’s simply down to what programming language you intend to code in and what your end goals are.



Coding can be one of those experiences that sounds fantastic, but to get going with it, is often confusing. After all, there’s a plethora of languages to choose from, numerous apps that will enable you to code in a specific, or range, of languages and an equally huge amount of third-party software to consider. Then you access the Internet and discover that there are countless coding tutorials available, for the language in which you’ve decided you want to program, alongside even more examples of code. It’s all a little too much at times.

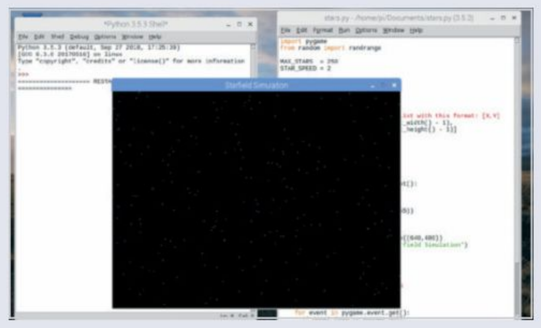
The trick is to slow down and, to begin with, not look too deeply into coding. Like all good projects, you need a solid foundation on which to build your skill and to have all the necessary tools available to hand to enable you to complete the basic steps. This is where creating a coding platform comes in, as it will be your learning foundation while you begin to take your first tentative steps into the wider world of coding.

HARDWARE

Thankfully, coding at the foundation level doesn’t require specialist equipment, or a top of the range, liquid hydrogen-cooled PC. If you own a computer, no matter how basic, you can begin to learn how to code. Naturally, if your computer in question is a Commodore 64 then you may have some difficulty following a modern language tutorial, but some of the best programmers around today started on an 8-bit machine, so there’s hope yet.

Access to the Internet is necessary to download, install and update the coding development environment, alongside a computer with either: Windows 10, macOS, or Linux installed. You can use other operating systems, but these are the ‘big three’ and you will find that most code resources are written with one, or all of these, in mind.

SOFTWARE



In terms of software, most of the development environments - the tools that allow you to code, compile the code and execute it - are freely available to download and install. There are some specialist tools available that will cost, but at this level they’re not necessary; so don’t be fooled into thinking you need to purchase any extra software in order to start learning how to code.

Over time, you may find yourself changing from the mainstream development environment and using a collection of your own, discovered, tools to write your code in. It’s all personal preference in the end and as you become more experienced, you will start to use different tools to get the job done. Some environments will enhance sections of code, making it easier to read, others will allow quick compiling and execution. It’s all a matter of testing and experience.

OPERATING SYSTEMS

Windows 10 is the most used operating system in the world, so it's natural that the vast majority of coding tools are written for Microsoft's leading operating system. However, don't discount macOS and especially Linux.

macOS users enjoy an equal number of coding tools to their Windows counterparts. In fact, you will probably find that a lot of professional coders use a Mac over a PC, simply because of the fact that the Mac operating system is built on top of Unix (the command-line OS that powers much of the world's filesystems and servers). This Unix layer lets you test programs in almost any language without using a specialised IDE.

Linux, however, is by far one of the most popular and important, coding operating systems available. Not only does it have a Unix-like backbone, but also it's also free to download, install and use and comes with most of the tools necessary to start learning how to code. Linux powers most of the servers that make up the Internet. It's used on nearly all of the top supercomputers, as well as specifically in organisations such as NASA, CERN and the military and it forms the base of Android-powered devices, smart TVs and in-car systems. Linux, as a coding platform, is an excellent idea and it can be installed inside a virtual machine without ever affecting the installation of Windows or macOS.

THE RASPBERRY PI

If you haven't already heard of the Raspberry Pi, then we suggest you head over to **www.raspberrypi.org**, and check it out. In short, the Raspberry Pi is a small, fully functional computer that comes with its own customised Linux-based operating system, pre-installed with everything you need to start learning how to code in Python, C++, Scratch and more.

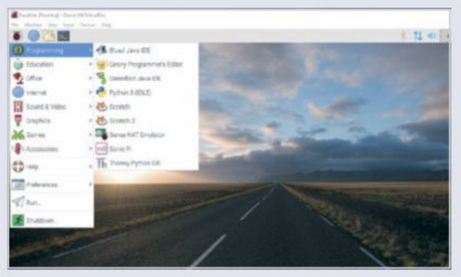
It's incredibly cheap, costing around £35 and allows you to utilise different hardware, in the form of robotics and electronics projects, as well as offering a complete desktop experience. Although not the most powerful computing device in the world, the Raspberry Pi has a lot going for it, especially in terms of being one of the best coding platforms available.

YOUR OWN CODING PLATFORM

Whichever method you choose, remember that your coding platform will probably change, as you gain experience and favour one language over another. Don't be afraid to experiment along the way, as you will eventually create your own unique platform that can handle all the code you enter into it.

VIRTUAL MACHINES

A virtual machine is a piece of software that allows you to install a fully working, operating system within the confines of the software itself. The installed OS will allocate user-defined resources from the host computer, providing memory, hard drive space etc, as well as sharing the host computer's Internet connection.



The advantage of a virtual machine is that you can work with Linux, for example, without it affecting your currently installed host OS. This means that you can have Windows 10 running, launch your virtual machine client, boot into Linux and use all the functionality of Linux while still being able to use Windows.

This, of course, makes it a fantastic coding platform, as you can have different installations of operating systems running from the host computer while using different coding languages. You can test your code without fear of breaking your host OS and it's easy to return to a previous configuration without the need to reinstall everything again.

Virtualisation is the key to most big companies now. You will probably find, for example, rather than having a single server with an installation of Windows Server, the IT team have instead opted for a virtualised environment whereby each Windows Server instance is a virtual machine running from several powerful machines. This cuts down on the number of physical machines, allows the team to better manage resources and enables them to deploy an entire server dedicated to a particular task in a fraction of the time.

MINIX NEO N42C-4

The NEO N42C-4 is an extraordinarily small computer from mini-PC developer, MINIX. Measuring just 139 x 139 x 30mm, this Intel N4200 CPU powered, Windows 10 Pro pre-installed computer is one of the best coding platforms we've come across.

The beauty, of course, lies in the fact that with increased storage and memory available, you're able to create a computer that can easily host multiple virtual machines. The virtual machines can cover Linux, Android and other operating systems, allowing you to write and test cross-platform code without fear of damaging, or causing problems, with other production or home computers.

The MINIX NEO N42C-4 starts at around £250, with the base 32GB eMMC and 4GB of memory. You'll need to add another hundred and fifty, or so, to increase the specifications, but consider that a license for Windows 10 Pro alone costs £219 from the Microsoft Store and you can begin to see the benefits of opting for a more impressive hardware foundation over the likes of the Raspberry Pi.

Using Virtual Machines

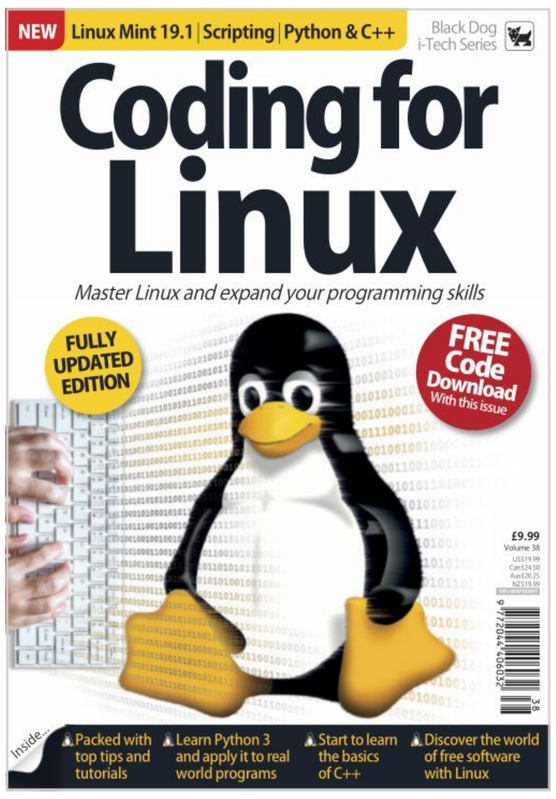
A Virtual Machine allows you to run an entire operating system from within an app on your desktop. This way, you’re able to host multiple systems in a secure, safe and isolated environment. In short, it’s an ideal way to code.

Sounds good, but what exactly is a Virtual Machine and how does it work?

The official definition of a virtual machine is ‘an efficient, isolated duplicate of a real computer machine’. This basically means that a virtual machine is an emulated computer system that can operate in exactly the same way as a physical machine, but within the confines of a dedicated virtual machine operator, or Hypervisor.

The Hypervisor itself, is an app that will allow you to install a separate operating system, creating a virtual computer system within itself complete with access to the Internet, your home network and so on.

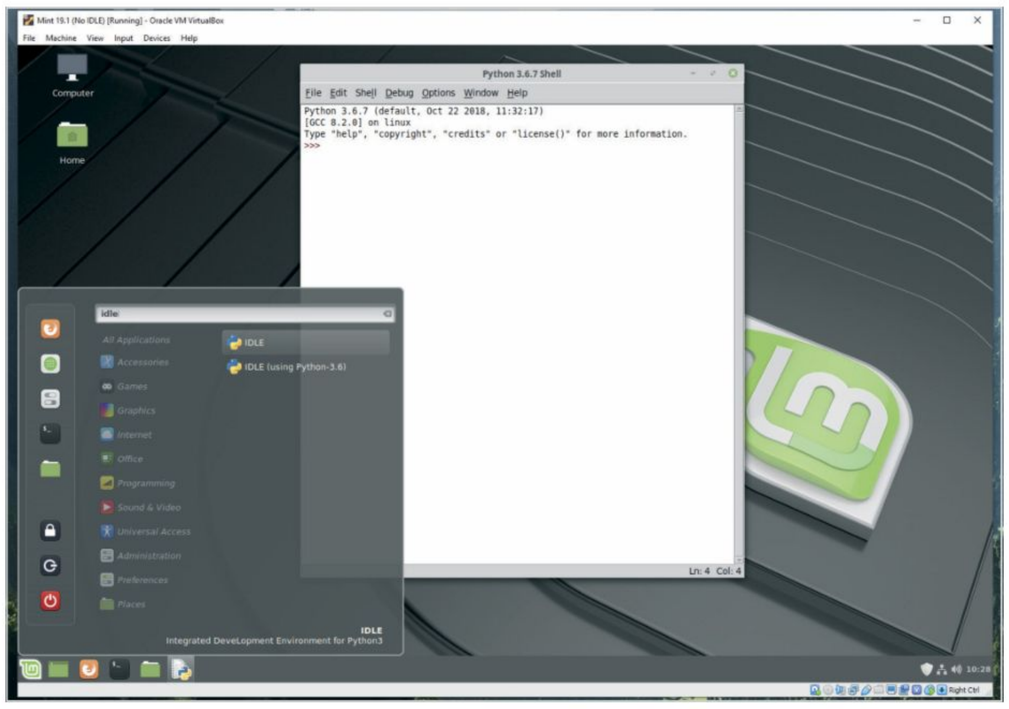
The Hypervisor will take resources from the host system - your physical computer, to create the virtual computer. This means that part of your physical computer’s: memory, CPU, hard drive space and other shared resources, will be set aside for use in the virtual machine and therefore won’t be available to the physical computer until the hypervisor has been closed down.



Our Linux titles contain steps on how to install a hypervisor and OS.

This resource overhead can be crippling for the physical machine if you don’t already have enough memory, or hard drive space available, or your computer has a particularly slow processor. While it’s entirely possible to run virtual machines on as little as 2GB of memory, it’s not advisable. Ideally, you will need a minimum of 8GB of memory (you can get away with 4GB, but again, your physical computer will begin to suffer with the loss of memory to the virtual machine), at least 25 to 50GB of free space on your hard drive and a quad-core processor (again, you can have a dual-core CPU, but that can cause a bottleneck on your physical computer).

The limit to how many different virtual machines you host on your physical computer is restricted, therefore, by the amount of physical system resources you can allocate to each, while still leaving enough for your physical computer to operate on.



You’re able to install Linux, and code inside a virtual machine on a Windows 10 host.

VIRTUAL OS

From within a hypervisor you’re able to run a number of different operating systems. The type of OS depends greatly on the hypervisor you’re running, as some are better at emulating a particular system over others. For example, VirtualBox, a free and easy to use hypervisor from Oracle, is great at running Windows and Linux virtual machines, but isn’t so good at Android or macOS. QEMU is good for emulating ARM processors, therefore ideal for Android and such, but it can be difficult to master.

There are plenty of hypervisors available to try for free, with an equal amount commercially available that are significantly more powerful and offer better features. However, for most users, both beginner and professional, VirtualBox does a good enough job.

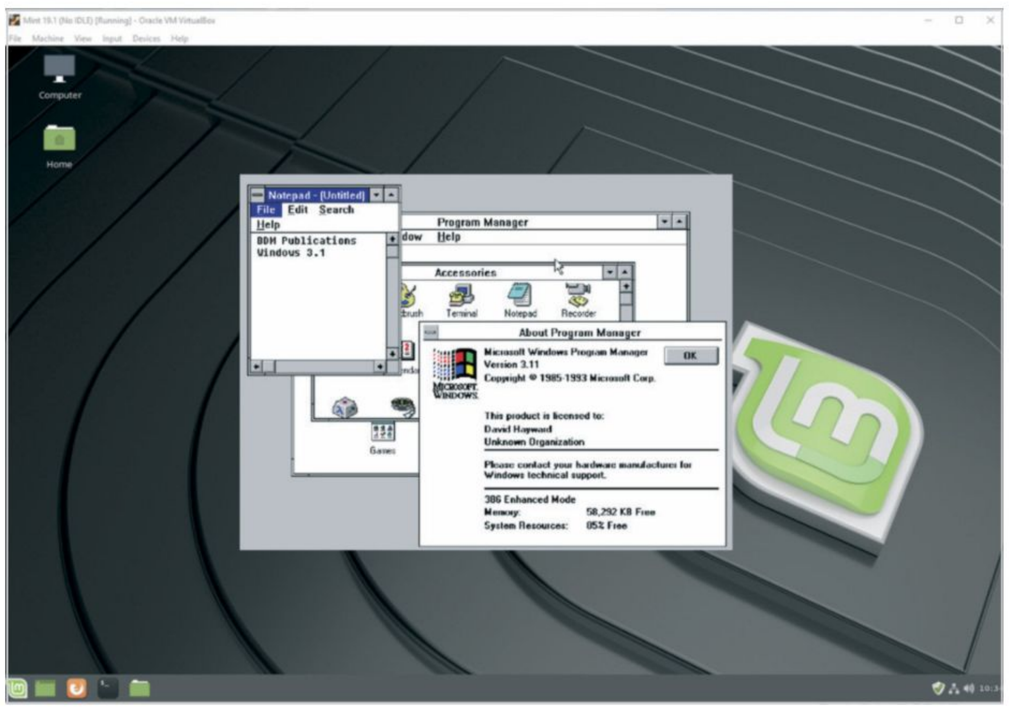
Within a hypervisor, you’re able to set up and install any of the newer distributions of Linux, or if you feel the need, you’re also able to install some of the more antiquated versions. You can install early versions of Windows, even as far back as Windows 3 complete with DOS 6.22 – although you may find some functionality of the VM lost due to the older drivers (such as access to the network).

With this in mind then, you’re able to have an installation of Linux Mint, or the latest version of Ubuntu, running in an app on your Windows 10 PC. This is the beauty of using a virtual machine. Conversely, if your physical computer has Linux as its installed operating system, then with a hypervisor you’re able to create a Windows 10 virtual machine – although you will need to have a licence code available to register and activate Windows 10.

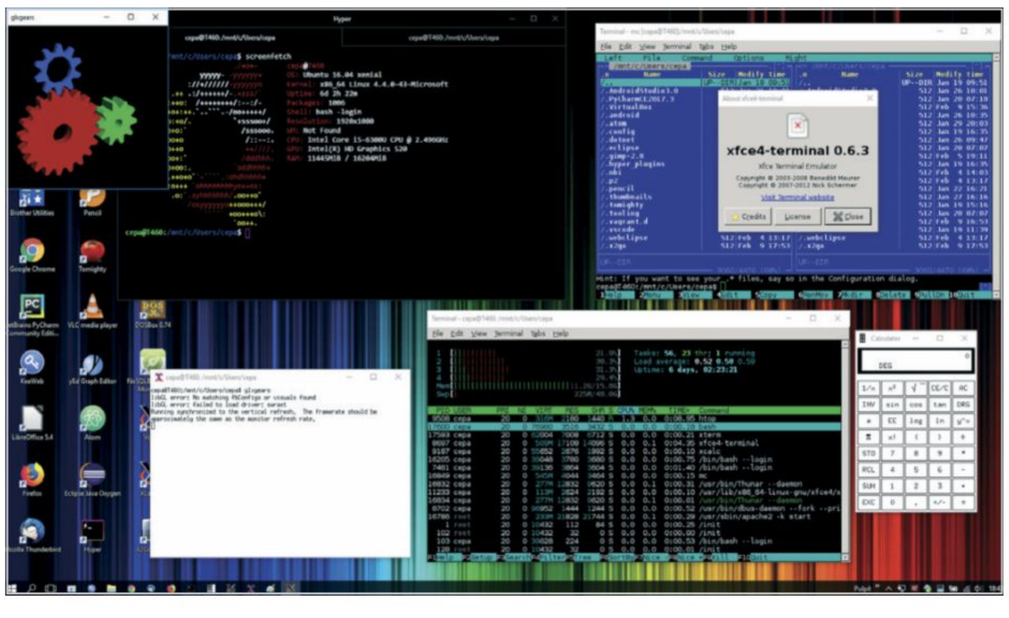
Using virtual machines removes the need to dual-boot. Dual-booting is having two, or more, physical operating systems installed on the same, or multiple, hard drives on a single computer. As the computer powers up, you're given the option to choose which OS you want to boot into. While this sounds like a more ideal scenario it isn't always as straight forward as it sounds, as all the operating systems that are booted into will have full access to the computer's entire system resources.

The problems with dual-booting come when one of the operating systems is updated. Most updates cover security patching, or bug fixing, however, some updates can alter the core - the kernel, of the OS. When these changes are applied, the update may alter the way in which the OS starts up, meaning the initial boot choice you made could be overwritten, leaving you without the ability to access the other operating systems installed on the computer. To rectify this, you'll need to access the Master Boot Record and alter the configuration to re-allow booting into the other systems. There's also the danger of possibly overwriting the first installed OS, or overwriting data and more often than not, most operating systems don't play well when running side-by-side. Indeed, while good, dual-booting has more than its fair share of problems. In contrast, using a virtual machine environment, while still problematic at times, takes out some of the more nasty and disastrous aspects of using multiple operating systems on a single computer.

Even old operating systems can be run inside a virtual machine.



Virtual machines can be as simple, or as complex as your needs require.

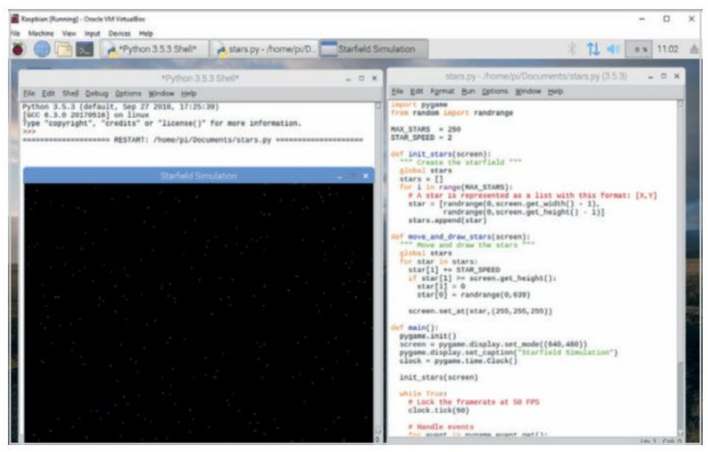


ADVANTAGES FOR CODERS

For the coder, having a virtual machine setup offers many advantages, the most popular being cross-platform code. Meaning if you write code within Windows 10, then with an installation of a Linux distro in a hypervisor, you're able to quickly and effortlessly power up the virtual machine and test your code in a completely different operating system. From this, you're able to iron out any bugs, tweak the code so it works better on a different platform and expand the reach of your code to non-Windows users.

The advantage of being able to configure a development environment, in specific ways for specific projects, is quite invaluable. Using a virtual machine setup greatly reduces the uncertainties that are inherent to having multiple versions of programming languages, libraries, IDEs and modules installed, to support the many different projects you may become involved in as a coder. Elements of code that 'talk' directly to specifics of an operating system can easily be overcome, without the need to clutter up your main, host system with cross-platform libraries, which in turn may have an affect on other libraries within the IDE.

Another element to consider is stability. If you're writing code that could potentially cause some instability to the core OS during its development phase, then executing and testing that code on a virtual machine makes more sense than testing it on your main computer; where having to repeatedly reboot, or reset something due to the code's instabilities, can become inefficient and just plain annoying.



Coding in Python on the Raspberry Pi Desktop OS inside a VM on Windows 10!

The virtual machine environment can be viewed as a sandbox, where you're able to test unsecure, or unstable code without it causing harm, or doing damage, to your main, working computer. Viruses and malware can be isolated within the VM without infecting the main computer, you're able to set up anonymity Internet use within the VM and you're able to install third-party software without it slowing down your main computer.

GOING VIRTUAL

While you're at the early stages of coding, using a virtual machine may seem a little excessive. However, it's worth looking into because coding in Linux can often be easier than coding in Windows, as some versions of Linux have IDEs pre-installed. Either way, virtualisation of an operating system is how many of the professional and successful coders and developers work, so getting used to it early on in your skill set is advantageous.

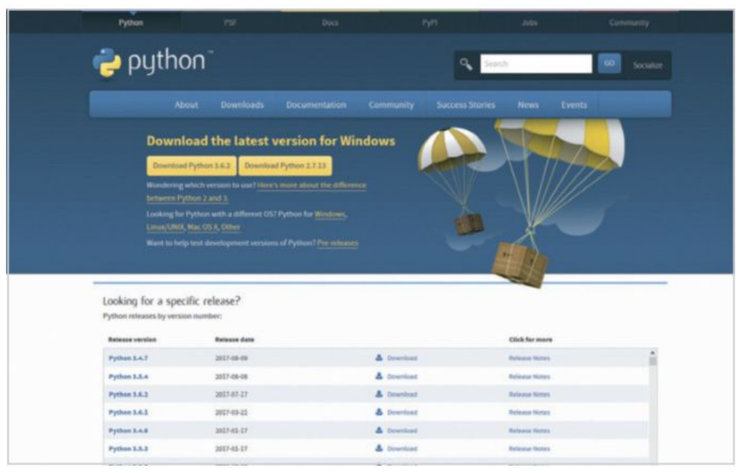
To start, look at installing VirtualBox. Then consider taking a look at our Linux titles, **https://bdmpublications.com/?s=linux&post_type=product**, to learn how to install Linux in a virtual environment and how best to utilise the operating system.

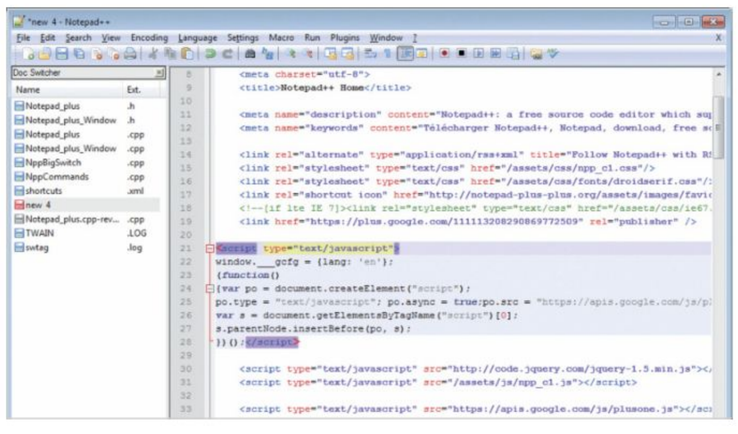
Equipment You Will Need

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

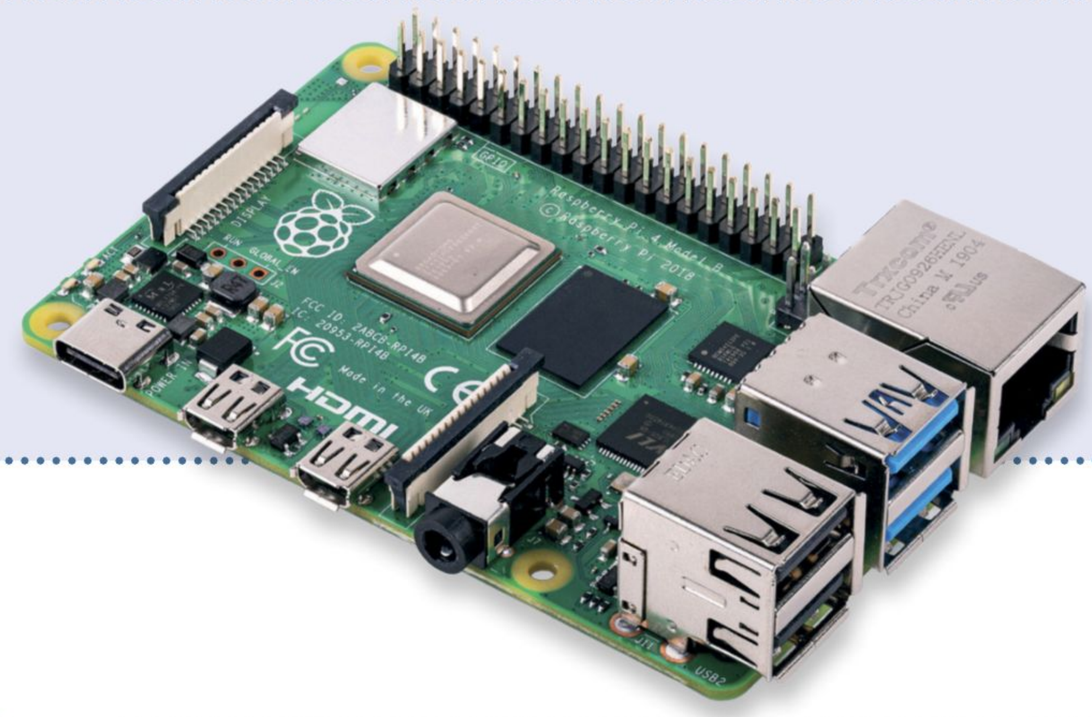
Despite what other books may lead you to believe, you won’t become a programmer in 24-hours. Learning to code in Python takes time, and patience. You may become stuck at times and other times the code will flow like water. Understand you’re learning something entirely new, and you will get there.

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 4 Model B is the latest version, incorporating a more powerful CPU, a choice of 1GB, 2GB or 4GB memory versions and Wi-Fi and Bluetooth support. You can pick up a Pi from around £33, increasing up to £54 for the 4GB memory version, or as a part of kit for £50+, depending on the kit you're interested in.

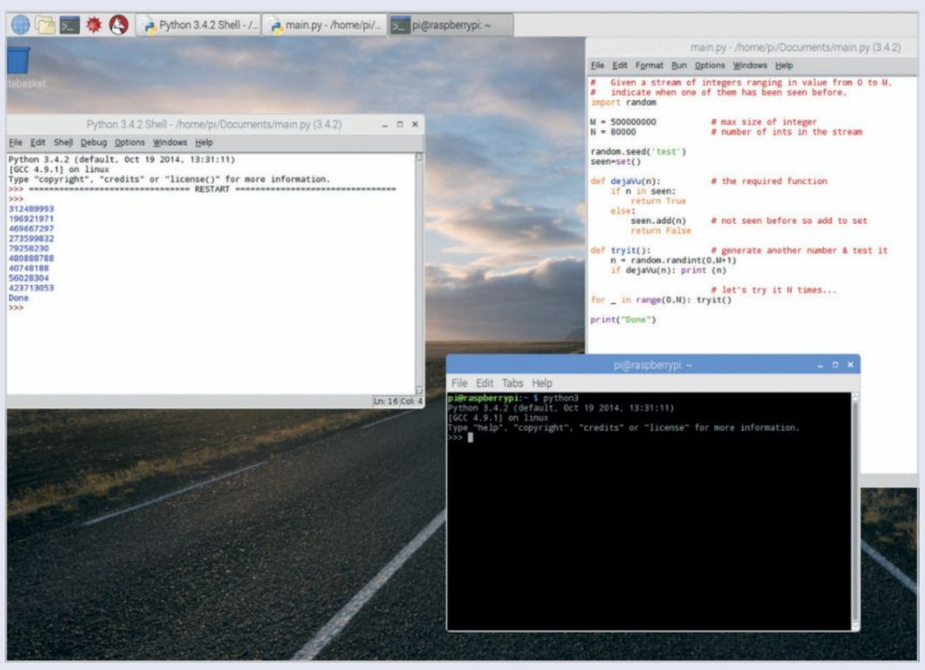


FUZE PROJECT

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.

RASPBIAN

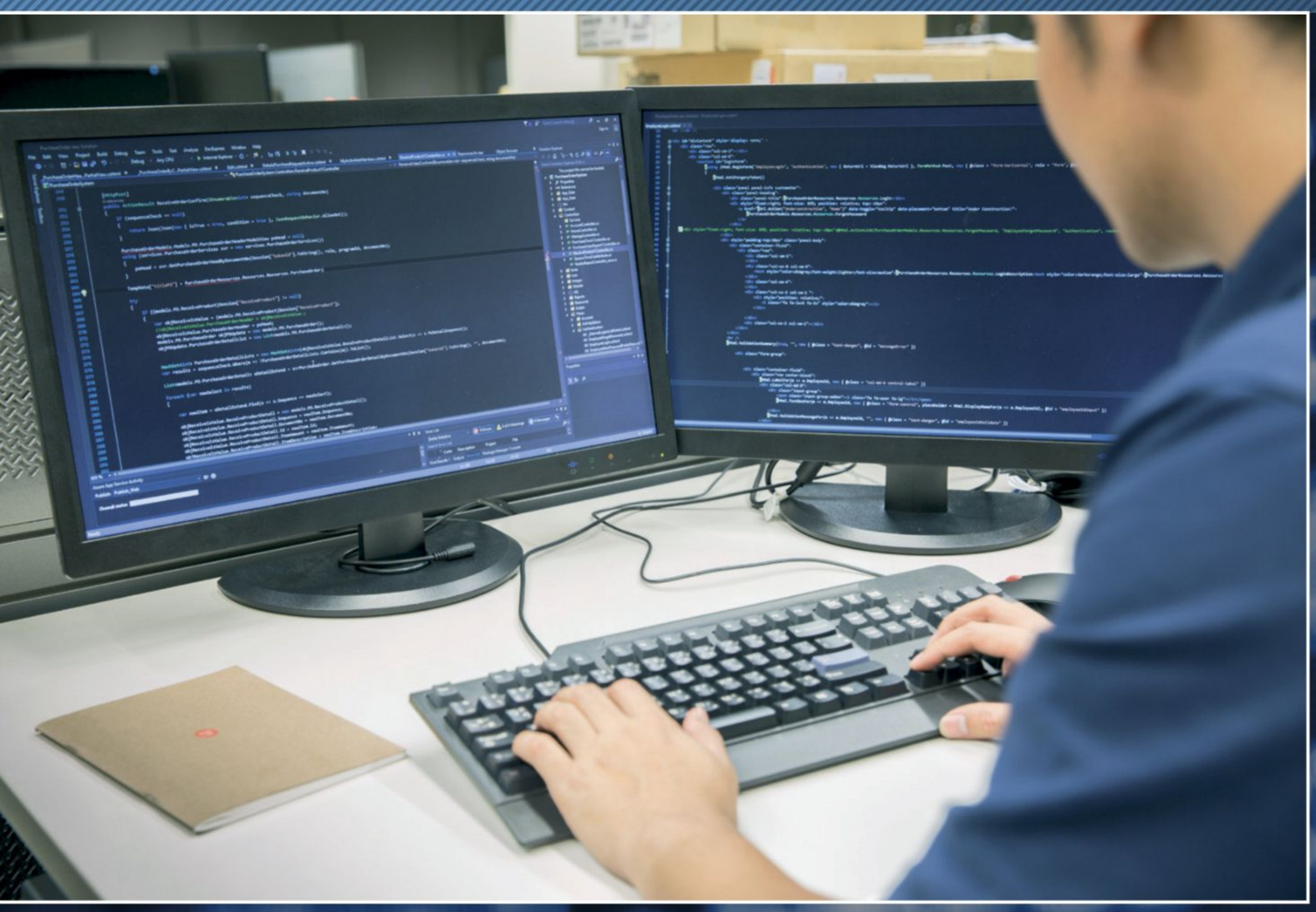
The Raspberry Pi's main operating system is a Debian-based Linux distribution that comes with everything you need in a simple to use package. It's streamlined for the Pi and is an ideal platform for hardware and software projects, Python programming and even as a desktop computer.



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.

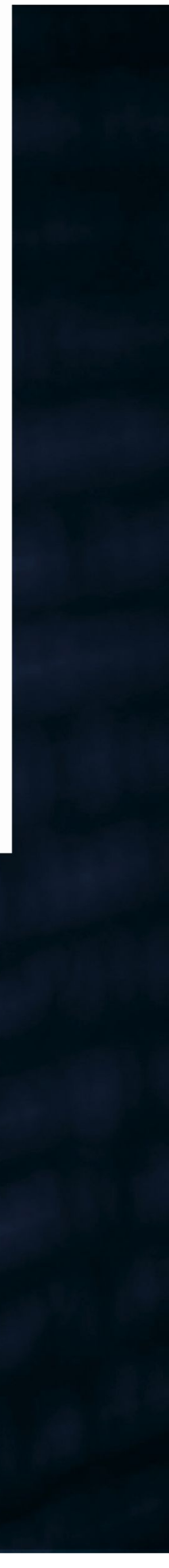
Welcome to Python



Python is a fantastic programming language. With it, you can create everything from a simple program to backup the pictures on your computer, through to analysing petabytes of data. Not only is Python the programming language of choice for some of the biggest companies in the world, it's also one of the easiest to learn.

This chapter will help you get Python set up on your computer, and introduce you to this remarkable and powerful language.

22 Why Python?
24 What can You Do with Python?
26 Python in Numbers
28 How to Set Up Python in Windows
30 How to Set Up Python in Linux
32 Python on the Pi
34 Getting to Know Python



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.

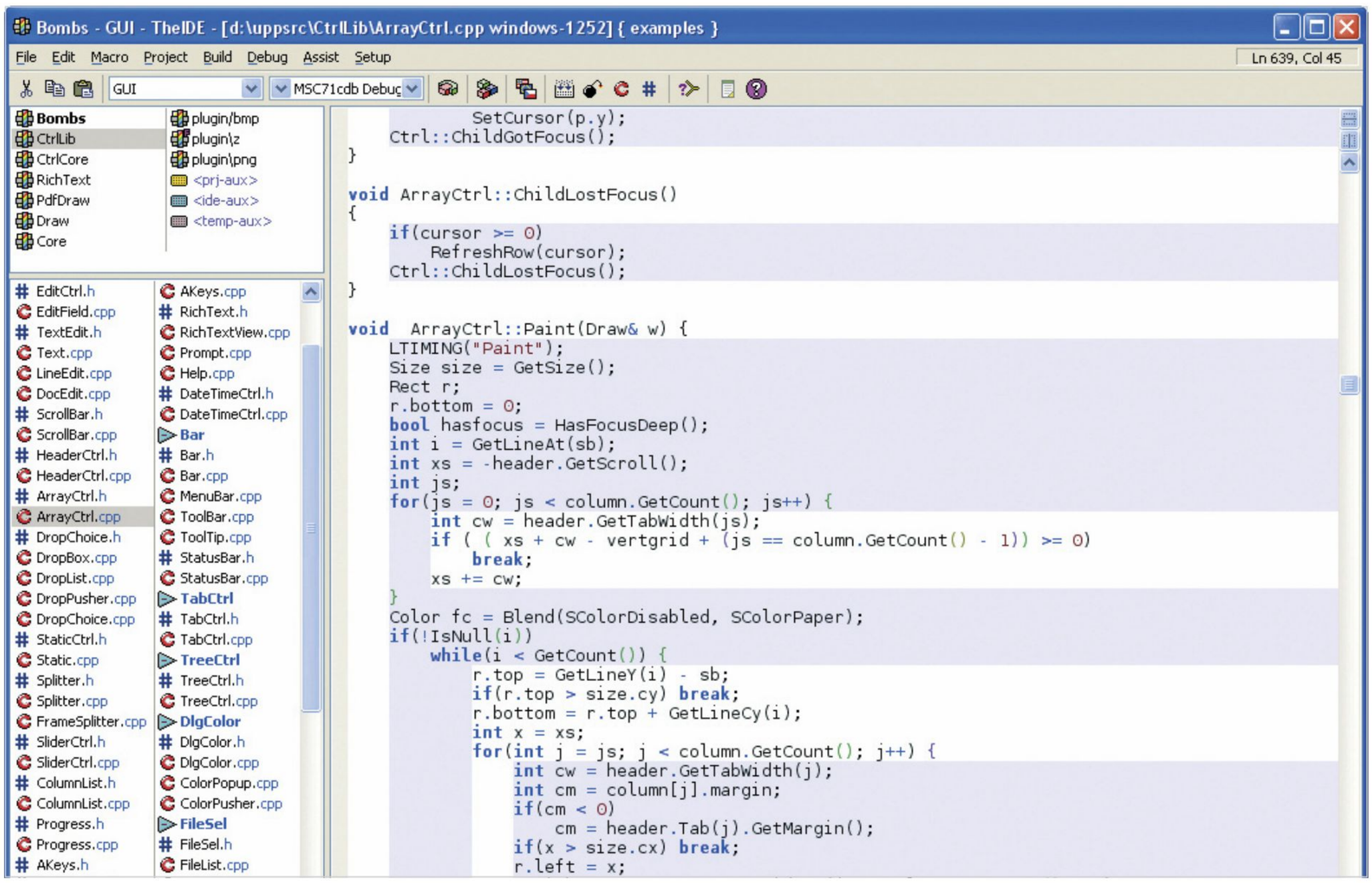
These pioneers of programming carved their way into a new frontier, forging small routines that enabled the letter 'A' to scroll across the screen. It may not sound terribly exciting to a generation that's used to ultra high-definition graphics and open world, multi-player online gaming. However, forty-something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



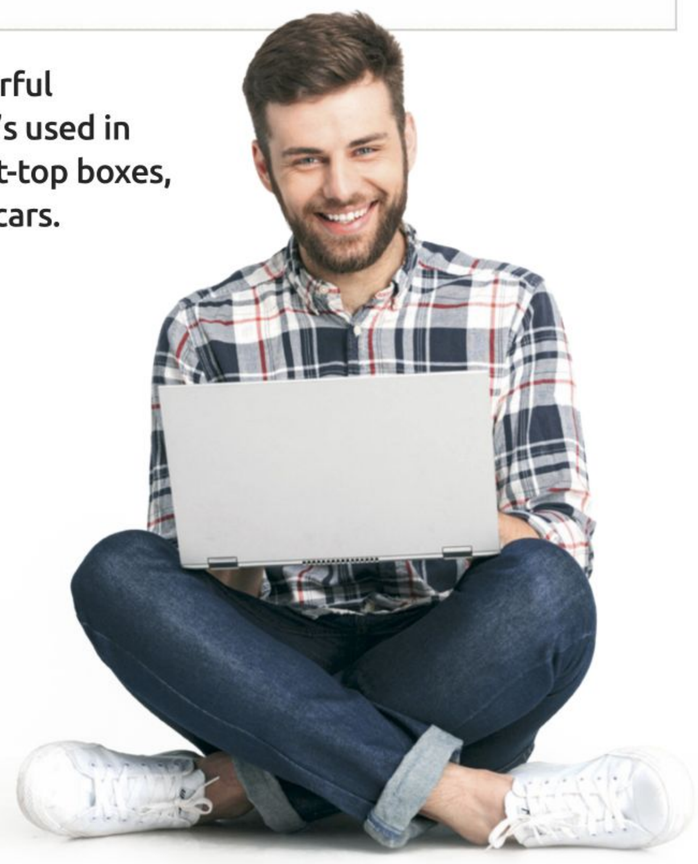
environment or users. Operating systems, such as Windows, macOS and such are usually a combination of C++, C#, assembly and some form of visual-based language. Games generally use C++ whilst web pages can use a plethora of available languages such as HTML, Java, Python and so on.

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.

```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 ) {
      // that method doesn't exist
    } catch ( IllegalAccessException e3 ) {
      // we don't have permission to invoke that
      method
    } catch ( InvocationTargetException e4 ) {
      // an exception ocurred while invoking that
      method
      System.out.println(
          "Method threw an: " + e4.
          getTargetException(  ) );
    }
  }
}
```

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



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.

```
  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.

```
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.

What can You Do with Python?

Python is an open-source, object-oriented programming language that's simple to understand and write with, yet also powerful and extremely malleable. It's these characteristics that help make it such an important language to learn.

Python's ability to create highly readable code within a small set of instructions has a considerable impact on our modern digital world. From the ideal, first programmers' choice to it's ability to create interactive stories and games; from scientific applications to artificial Intelligence and web-based applications, the only limit to Python is the imagination of the person coding in it.

It's Python's malleable design that makes it an ideal language for many different situations and roles. Even certain aspects of the coding world, that require more efficient code, still use Python. For example, NASA utilises Python both as a stand-alone language and as a bridge between other programming languages. This way, NASA scientists and engineers are able to get to the data they need without having to cross multiple language barriers; Python fills the gaps and provides the means to get the job done.

You'll find lots of examples of this, where Python is acting behind the scenes. This is why it's such an important language to learn.

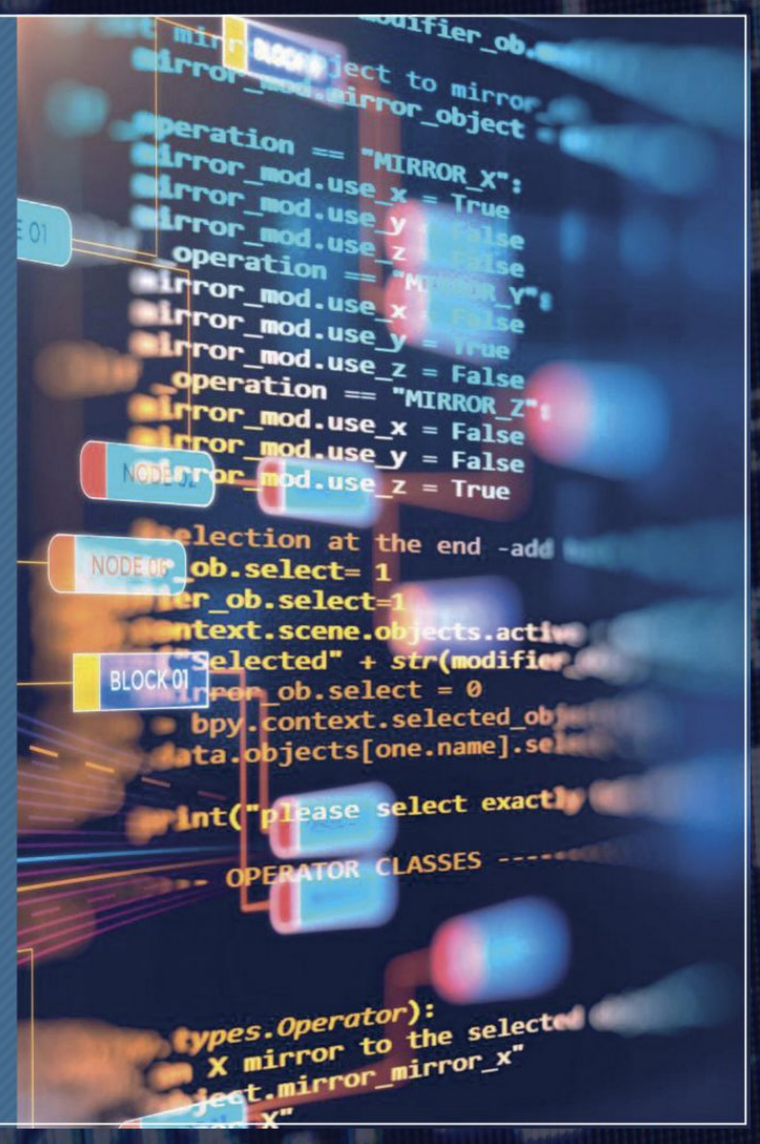
```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

BIG DATA

Big data is a buzzword you're likely to have come across in the last couple of years. Basically, it means extremely large data sets that are available for analysis to reveal patterns, trends and interactions between humans, society and technology. Of course, it's not just limited to those areas, big data is currently being used in a variety of industries, from social media to health and welfare, engineering to space exploration and beyond.

Python plays a substantial role in the world of big data. It's extensively used to analyse huge chunks of the available big data and extract specific information based on what the user/company requires from the wealth of numbers present. Thanks to an impressive set of data processing libraries, Python makes the act of getting to the data, in amongst the numbers, that counts and presenting it in a fashion that's readable and useable for humans.

There are countless libraries and freely available modules that enable fast, secure and more importantly, accurate processing of data from the likes of supercomputing clusters. For example, CERN uses a custom Python module to help analyse the 600 million collisions per second that the LHC produces. A different language handles the raw data, but Python is present to help sift through the data so scientists can get to the content they want without the need to learn a far more complex programming language.



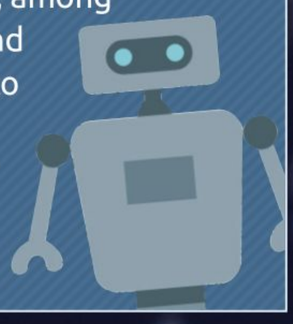
ARTIFICIAL INTELLIGENCE

Artificial Intelligence and Machine Learning are two of the most groundbreaking aspects of modern computing. AI is the umbrella term used for any computing process wherein the machine is doing something intelligent, working and reacting in similar ways to humans. Machine Learning is a subset of AI and provides the overall AI system with the ability to learn from its experiences.

However, AI isn't simply the creation of autonomous robots intent on wiping out human civilisation. Indeed, AI can be found in a variety of day-to-day computing applications where the 'machine', or more accurately the code, needs to learn from the actions of some form of input and anticipate what the input is likely to require, or do, next.

This model can be applied to Facebook, Google, Twitter, Instagram and so on. Have you ever looked up a celebrity on Instagram and then discovered that your searches within other social media platforms are now specifically targeted toward similar celebrities? This is a prime example of using AI in targeted advertising and behind the code and algorithms that predict what you're looking for, is Python.

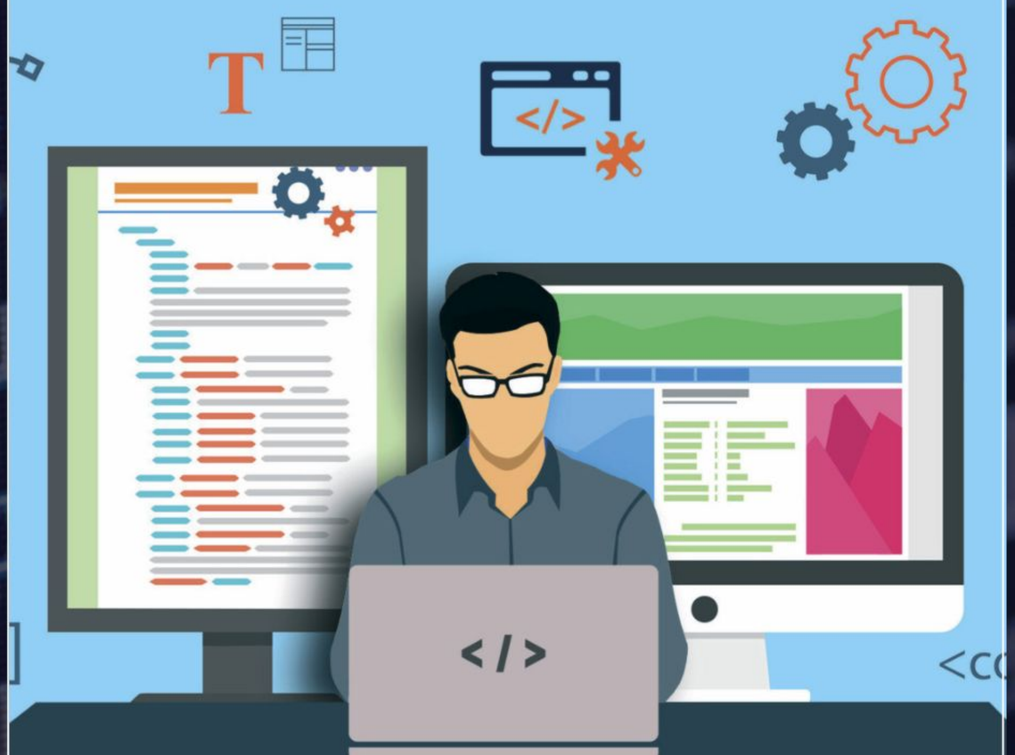
Spotify, for example, uses Python based code, among other things, to analyse your musical habits and offer playlists based on what you've listened to in the past. It's all clever stuff and, moving forward, Python is at the forefront of the way the Internet will work in the future.

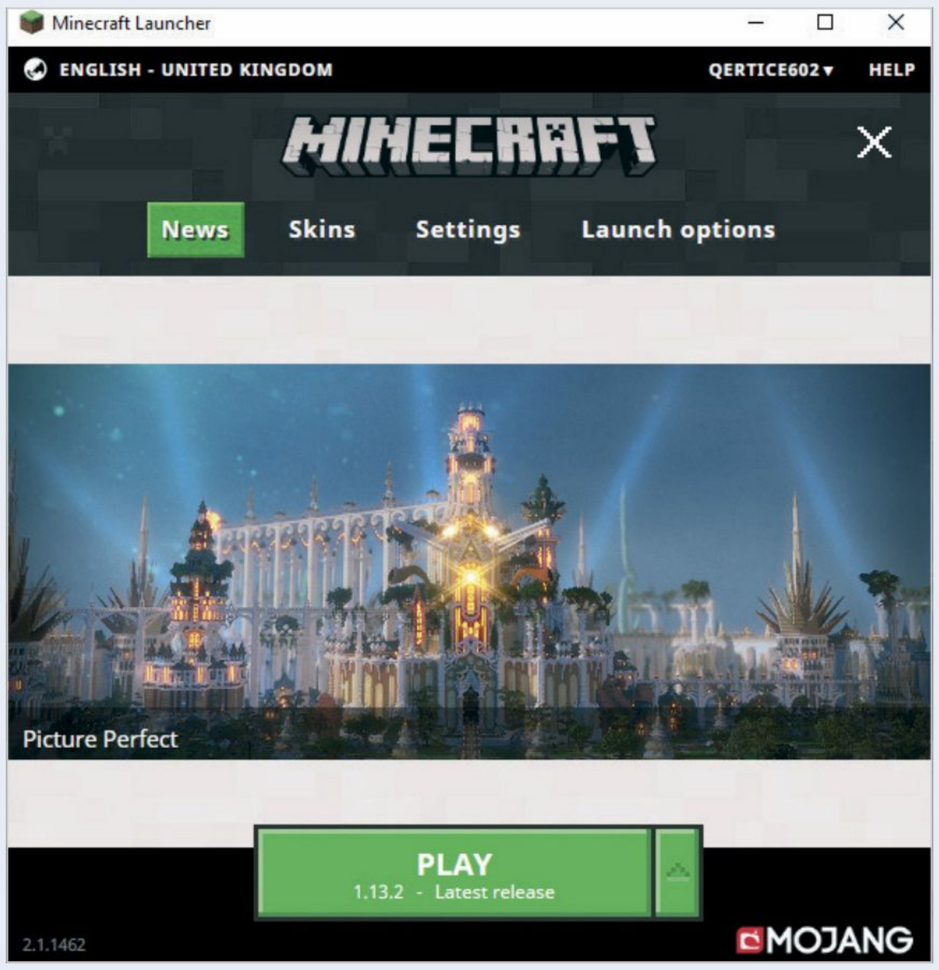


WEB DEVELOPMENT

Web development has moved on considerably since the early days of HTML scripting in a limited text editor. The many frameworks and web management services available now means that building a page has become increasingly complex.

With Python, the web developer has the ability to create dynamic and highly secure web apps, enabling interaction with other web services and apps such as Instagram and Pinterest. Python also allows the collection of data from other websites and even apps built within other websites.





GAMING

Although you won't find too many triple-A rated games coded using Python, you may be surprised to learn that Python is used as an extra on many of the high-ranking modern games.

The main use of Python in gaming comes in the form of scripting, where a Python script can add customisations to the core game engine. Many map editors are Python compatible and you will also come across it if you build any mods for games, such as The Sims.

A lot of the online, MMORPGs (Massively Multiplayer Online Role-Playing Games) available utilise Python as a companion language for the server-side elements. These include: code to search for potential cheating, load balancing across the game's servers, player skill matchmaking and to check whether the player's client-side game matches the server's versions. There's also a Python module that can be included in a Minecraft server, enabling the server admin to add blocks, send messages and automate a lot of the background complexities of the game.

PYTHON EVERYWHERE

As you can see, Python is quite a versatile programming language. By learning Python, you are creating a well-rounded skillset that's able to take you into the next generation of computing, either professionally or simply as a hobbyist.

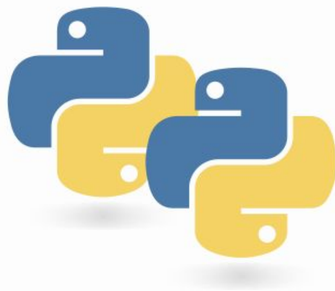
Whatever route you decide to take on your coding journey, you will do well to have Python in your corner.

PYTHON IN NUMBERS

Python creator **Guido Van Rossum** named Python after reading scripts from **Monty Python's Flying Circus.**

There's a lot to like about Python, but don't just take our word for it. Here are some amazing facts and figures surrounding one of the most popular programming languages of recent years.

Alexa, Amazon's Virtual Personal Assistant, uses Python to help with speech recognition.



Data analysis and Machine Learning are the two most used Python examples.

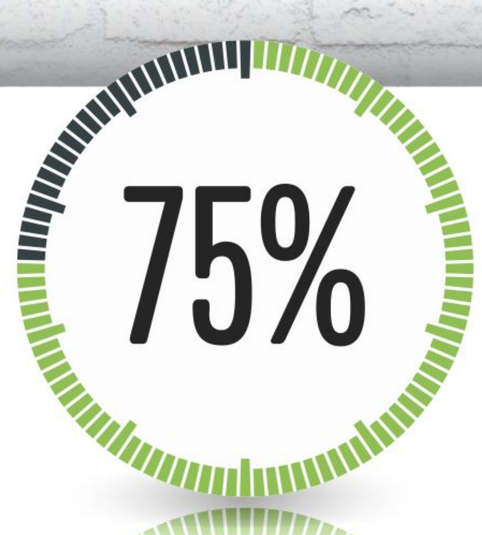
As of the end of 2018, Python was the most discussed language on the Internet.

PYTHON AND LINUX SKILLS ARE THE THIRD MOST POPULAR I.T. SKILLS IN THE UK.

Disney Pixar uses Python in its Renderman software to operate between other graphics packages.

OVER 75% OF RECOMMENDED CONTENT FROM NETFLIX IS GENERATED FROM MACHINE LEARNING – CODED BY PYTHON.

90% OF ALL FACEBOOK POSTS ARE FILTERED THROUGH PYTHON-CODED MACHINE LEARNING.



IT'S ESTIMATED THAT OVER 75% OF NASA'S WORKFLOW AUTOMATION SYSTEMS ON-BOARD THE I.S.S. USE PYTHON.

16,000

There are over 16,000 Python jobs posted every six months in the UK.

PYTHON SKILL-BASED POSITIONS ARE THE 16th MOST SOUGHT-AFTER JOBS IN THE UK.

Python Data Science is thought to become the most sought-after job in the coming years.

Google is the top company for hiring Python developers, closely followed by Microsoft.

Data Science, Blockchain and Machine Learning are the fastest growing Python coding skills.

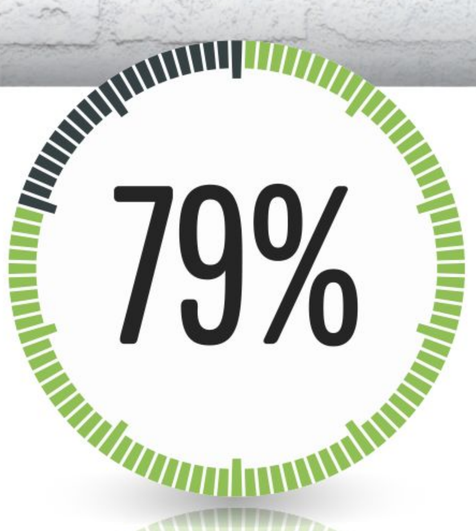
New York and San Francisco are the top Python developer cities in the world.

Python developers enjoy an average salary of

£60,000

95% OF ALL BEGINNER CODERS START WITH AND STILL USE, PYTHON AS THEIR PRIMARY OR SECONDARY LANGUAGE.

75% OF ALL PYTHON DEVELOPERS USE PYTHON 3, WHEREAS 25% STILL USE THE OUTDATED PYTHON 2 VERSION.



79% OF ALL PROGRAMMERS USE PYTHON REGULARLY, 21% USE IT AS A SECONDARY LANGUAGE.

49% OF WINDOWS 10 DEVELOPERS USE PYTHON 3 AS THEIR MAIN PROGRAMMING LANGUAGE.

How to Set Up Python in Windows

Windows users can easily install the latest version of Python via the main Python Downloads page. Whilst 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.

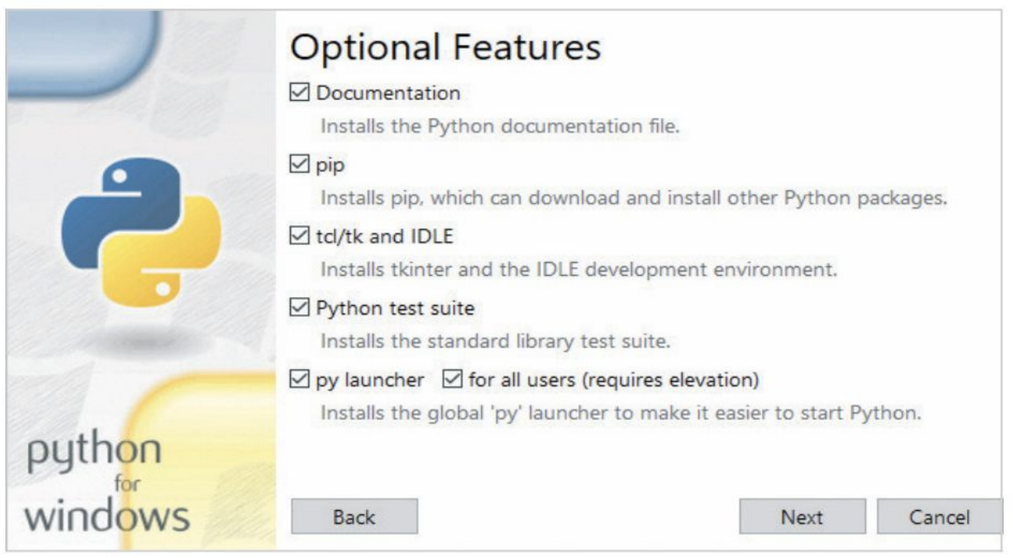
INSTALLING PYTHON 3.X

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.

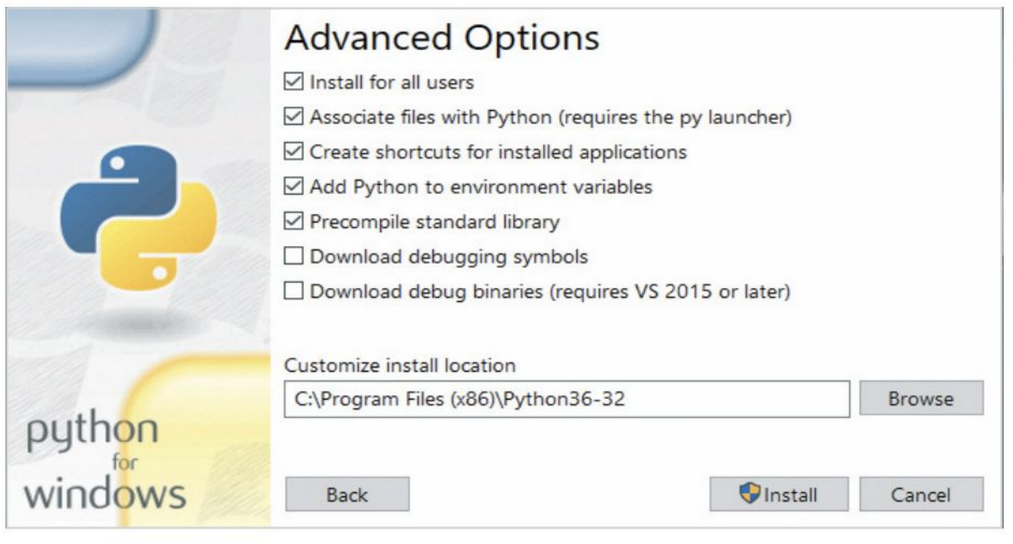
STEP 1 Start by opening your web browser to www.python.org/downloads/. Look for the button detailing the download link for Python 3.x. Python is regularly updated, changing the last digit for each bug fix and update. Therefore, don't worry if you see Python 3.7.3, or more, as long as it's Python 3, the code in this book will work fine.

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.

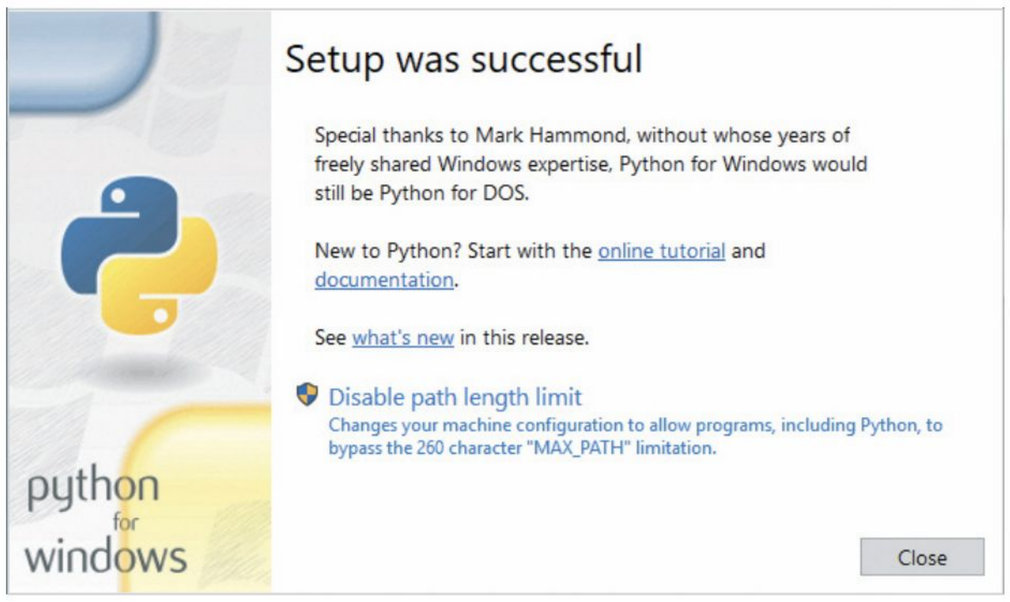
STEP 3 Choosing the Customise option allows you to specify certain parameters, and whilst you may stay with the defaults, it's a good habit to adopt as sometimes (not with Python, thankfully) installers can include unwanted additional features. On the first screen available, ensure all boxes are ticked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



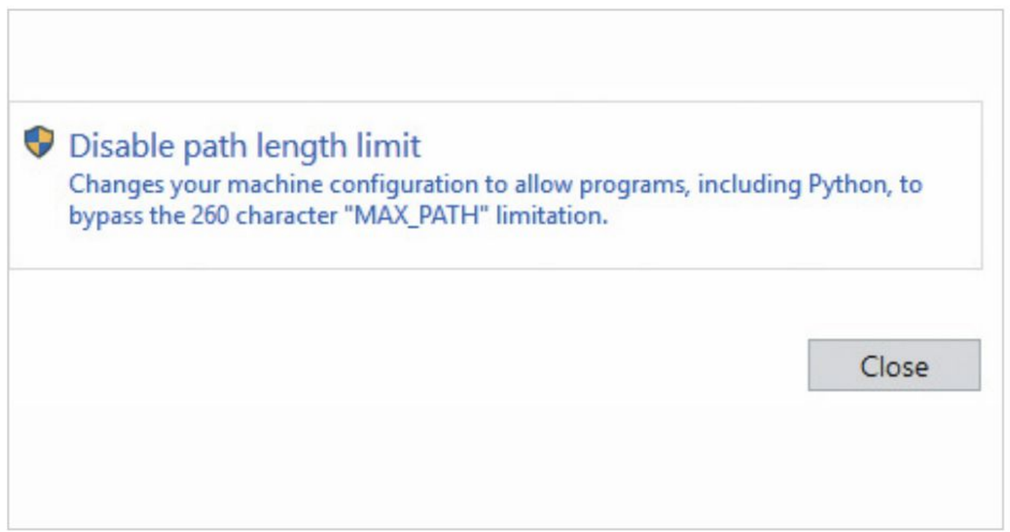
STEP 4 The next page of options include some interesting additions to Python. Ensure the Associate file with Python, Create Shortcuts, Add Python to Environment Variables, Precompile Standard Library and Install for All Users options are ticked. These make using Python later much easier. Click Install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



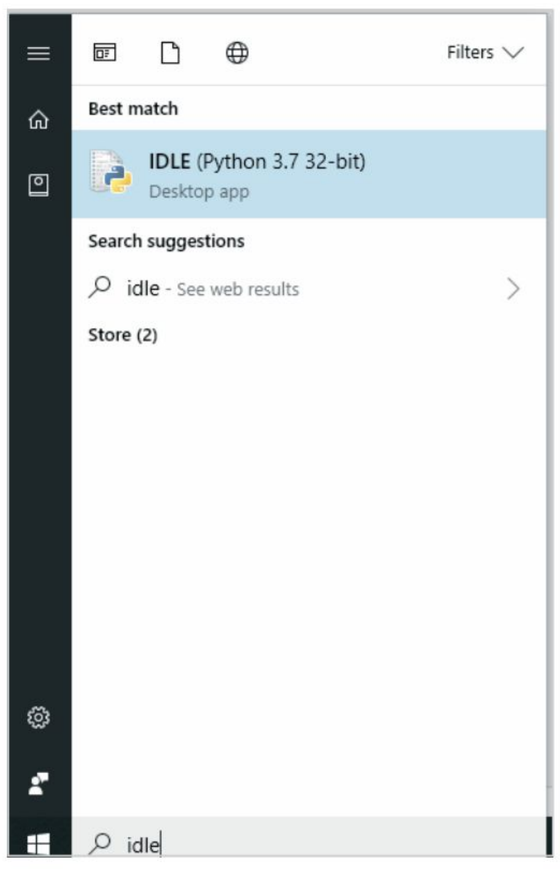
STEP 5 You may need to confirm the installation with the Windows authentication notification. Simply click Yes and Python will begin to install. Once the installation is complete the final Python wizard page will allow you to view the latest release notes, and follow some online tutorials.



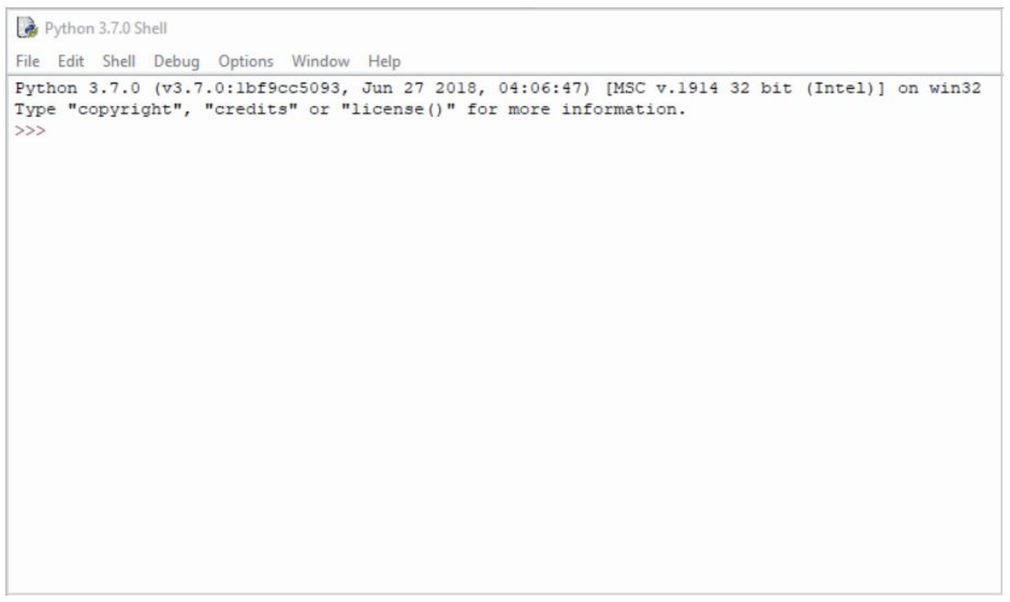
STEP 6 Before you close the install wizard window, however, it's best to click on the link next to the shield detailed Disable Path Length Limit. This will allow Python to bypass the Windows 260 character limitation, enabling you to execute Python programs stored in deep folders arrangements. Again,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



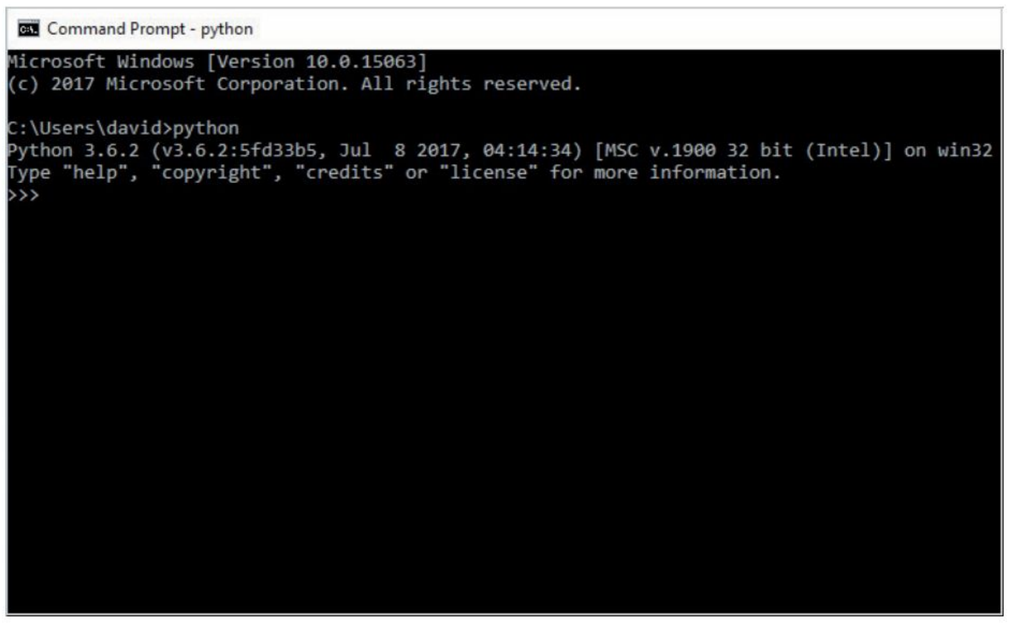
STEP 7 Windows 10 users can now find the installed Python 3.x within the Start button Recently Added section. The first link, Python 3.7 (32-bit) will launch the command line version of Python when clicked (more on that in a moment). To open the IDLE, type IDLE into Windows start.



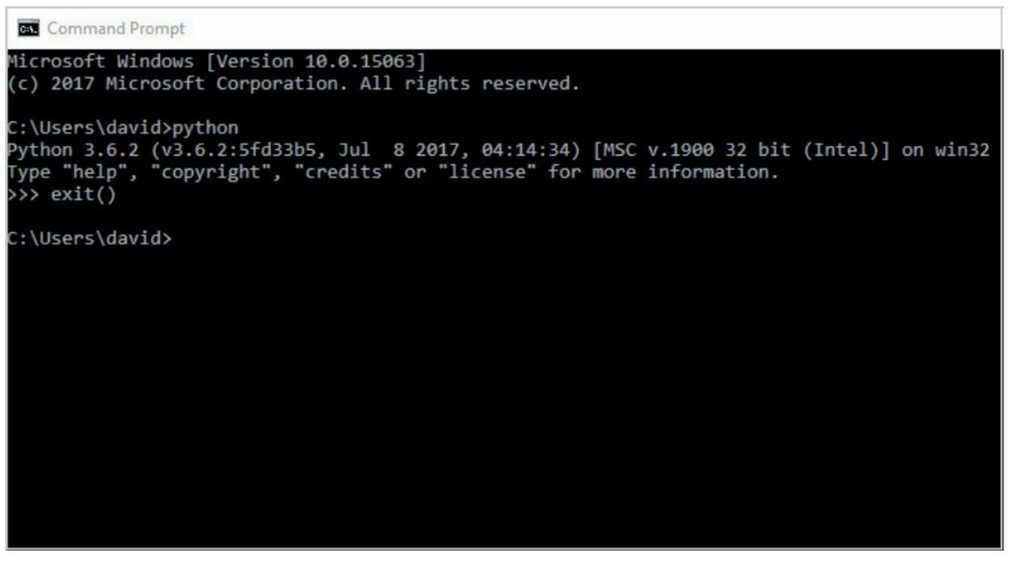
STEP 8 Clicking on the IDLE (Python 3.7 32-bit) link will launch the Python Shell, where you can begin your Python programming journey. Don't worry if your version is newer as long as it's Python 3.x our code works inside your Python 3 interface.



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.



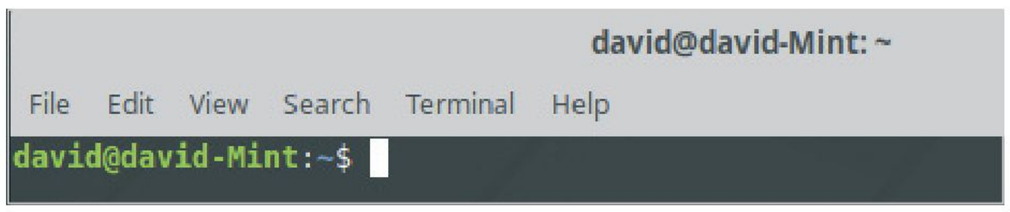
How to Set Up Python in Linux

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.

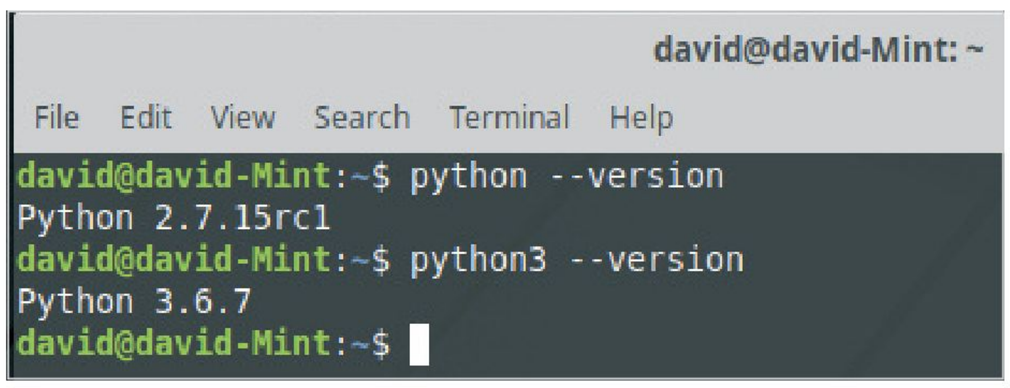
PYTHON PENGUIN

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, so for this particular tutorial we will stick to Linux Mint.

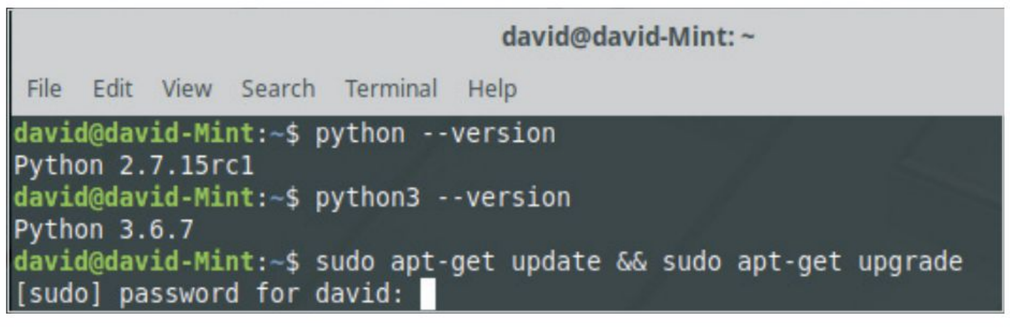
STEP 1 First you need to ascertain which version of Python is currently installed in your Linux system. To begin with, drop into a Terminal session from your distro's menu, or hit the Ctrl+Alt+T keys.



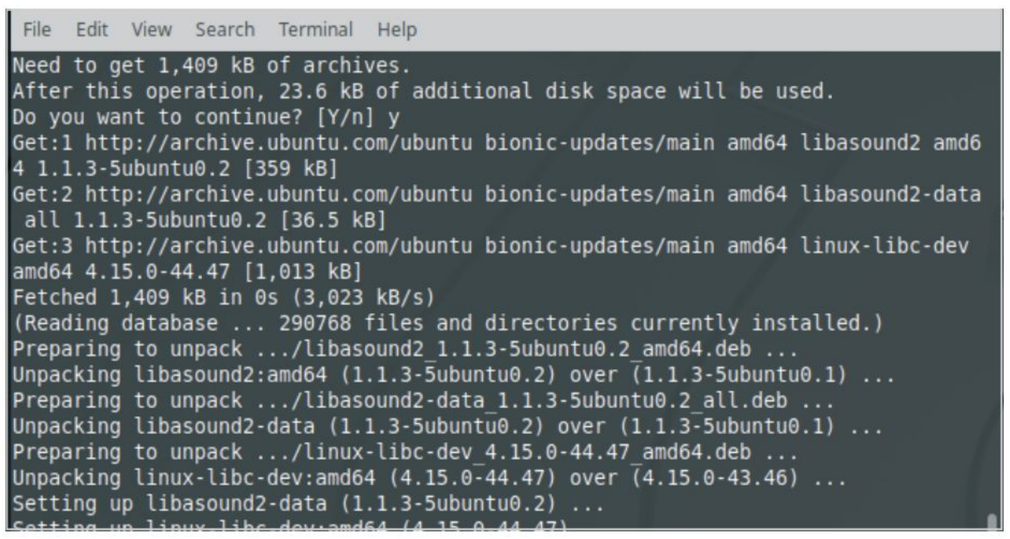
STEP 2 Next, enter: `python --version` into the Terminal screen. You should have the output relating to version 2.x of Python in the display. Most Linux distro come with both Python 2 and 3 by default, as there's plenty of code out there still available for Python 2. Now enter: `python3 --version`.



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.

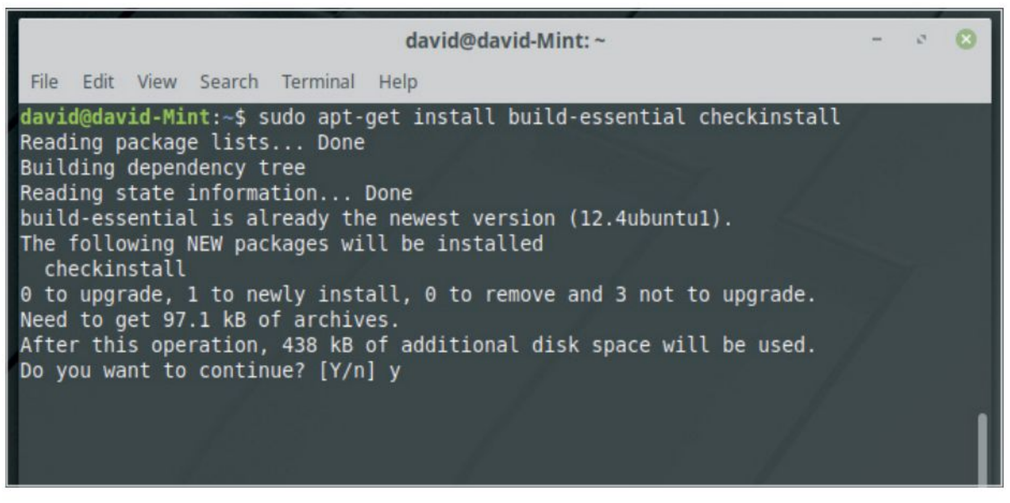


STEP 4 Once the update and upgrade completes, enter: `python3 --version` again to see if Python 3.x is updated, or even installed. As long as you have Python 3.x, you're running the most recent major version, the numbers after the 3. indicate patches plus further updates. Often they're unnecessary, but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

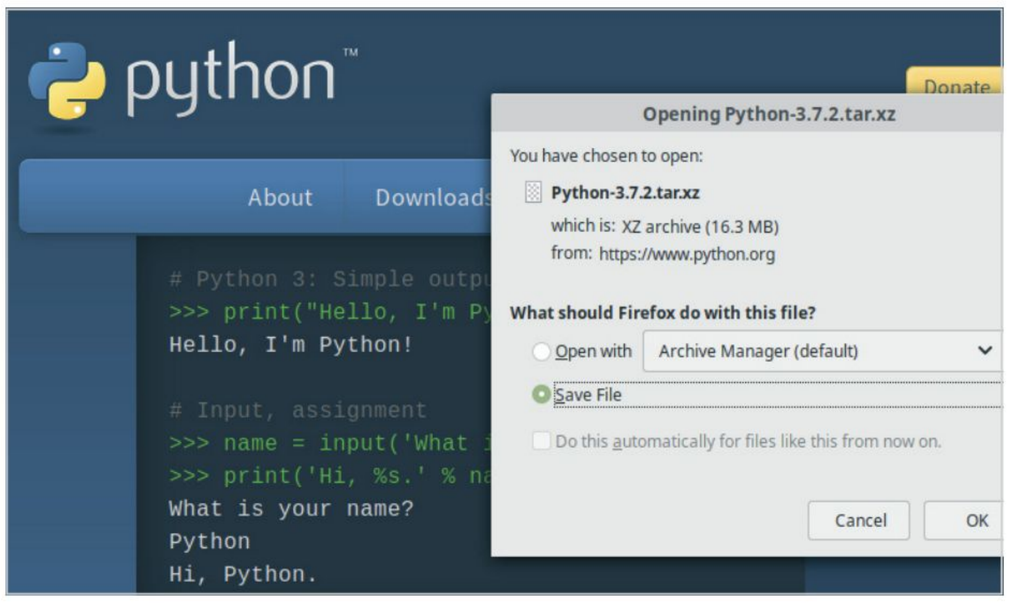


STEP 5 However, if you want the latest, cutting edge version, you'll need to build Python from source. Start by entering these commands into the Terminal:

```
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, then 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 with: `tar -xvf Python-3.Y.Y.tar.xz` (replace the Y's with the version numbers you've downloaded). Now enter the newly unzipped folder with: `cd Python-3.Y.Y/`.

```
Python-3.7.2/Objects/clinic/floatobject.c.h
Python-3.7.2/Objects/clinic/funcobject.c.h
Python-3.7.2/Objects/clinic/longobject.c.h
Python-3.7.2/Objects/clinic/dictobject.c.h
Python-3.7.2/Objects/clinic/structseq.c.h
Python-3.7.2/Objects/clinic/tupleobject.c.h
Python-3.7.2/Objects/clinic/moduleobject.c.h
Python-3.7.2/Objects/clinic/odictobject.c.h
Python-3.7.2/Objects/bytearrayobject.c
Python-3.7.2/Objects/typeobject.c
Python-3.7.2/Objects/lnotab_notes.txt
Python-3.7.2/Objects/methodobject.c
Python-3.7.2/Objects/tupleobject.c
Python-3.7.2/Objects/obmalloc.c
Python-3.7.2/Objects/object.c
Python-3.7.2/Objects/abstract.c
Python-3.7.2/Objects/listobject.c
Python-3.7.2/Objects/bytes_methods.c
Python-3.7.2/Objects/dictnotes.txt
Python-3.7.2/Objects/typeslots.inc
david@david-Mint:~/Downloads$ cd Python-3.7.2/
david@david-Mint:~/Downloads/Python-3.7.2$
```

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 now have Python 3.7 installed, alongside older Python 3.x.x and Python 2.

```
checking whether compiling and linking against OpenSSL works... no
checking for --with-ssl-default-suites... python
configure: creating ./config.status
config.status: creating Makefile.pre
config.status: creating Misc/python.pc
config.status: creating Misc/python-config.sh
config.status: creating Modules/ld_so_aix
config.status: creating pyconfig.h
creating Modules/Setup
creating Modules/Setup.local
creating Makefile


If you want a release build with all stable optimizations active (PGO, etc),
please run ./configure --enable-optimizations


david@david-Mint:~/Downloads/Python-3.7.2$ sudo make altinstall
```

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 to the one you installed from source.

```
david@david-Mint: ~/Downloads/Python-3.7.2
File  Edit  View  Search  Terminal  Help
david@david-Mint:~/Downloads/Python-3.7.2$ sudo apt-get install idle3
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  blt idle idle-python3.6 python3-tk tk8.6-blt2.5
Suggested packages:
  blt-demo tix python3-tk-dbg
The following NEW packages will be installed
  blt idle idle-python3.6 idle3 python3-tk tk8.6-blt2.5
0 to upgrade, 6 to newly install, 0 to remove and 3 not to upgrade.
Need to get 938 kB of archives.
After this operation, 4,221 kB of additional disk space will be used.
Do you want to continue? [Y/n]
```

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`

Once PIP is 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.

```
david@david-Mint: ~/Downloads/Python-3.7.2
File  Edit  View  Search  Terminal  Help
david@david-Mint:~/Downloads/Python-3.7.2$ sudo apt-get install python3-pip
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  python-pip-whl python3-distutils python3-lib2to3
Recommended packages:
  python3-dev python3-setuptools python3-wheel
The following NEW packages will be installed
  python-pip-whl python3-distutils python3-lib2to3 python3-pip
0 to upgrade, 4 to newly install, 0 to remove and 3 not to upgrade.
Need to get 1,984 kB of archives.
After this operation, 4,569 kB of additional disk space will be used.
Do you want to continue? [Y/n]
```

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, along with the necessary installers for macOS 64-bit for OS X 10.9 and later.

Python on the Pi

If you're considering on which platform to install and use Python, then give some thought to one of the best coding bases available: the Raspberry Pi. The Pi has many advantages for the coder: it's cheap, easy to use, and extraordinarily flexible.

THE POWER OF PI

While having a far more powerful coding platform on which to write and test your code is ideal, it's not often feasible. Most of us are unable to jump into a several hundred-pound investment when we're starting off and this is where the Raspberry Pi can help out.

While having a far more powerful coding platform on which to write and test your code is ideal, it's not often feasible. Most of us are unable to jump into a several hundred-pound investment when we're starting off and this is where the Raspberry Pi can help out.

The Raspberry Pi is a fantastic piece of modern hardware that has created, or rather re-created, the fascination we once all had about computers, how they work, how to code and foundation level electronics. Thanks to its unique mix of hardware and custom software, it has proved itself to be an amazing platform on which to learn how to code; specifically, using Python.

While you're able, with ease, to use the Raspberry Pi to learn to code with other programming languages, it's Python that has been firmly pushed to the forefront. The Raspberry Pi uses Raspbian as its recommended, default operating system. Raspbian is a Linux OS, or to be more accurate, it's a Debian-based distribution of Linux. This means that there's already a built-in element of Python programming, as opposed to a fresh installation of Windows 10, which has no Python-specific base. However, the Raspberry Pi Foundation has gone the extra mile to include a vast range of Python modules, extensions and even examples, out of the box. So, essentially, all you need to do is buy a Raspberry Pi, follow the instructions on how to set one up (by using one of our excellent Raspberry Pi guides found at **https://bdmpublications.com/?s=raspberry+pi&post_type=product)** and you can start coding with Python as soon as the desktop has loaded.

Significantly, there's a lot more to the Raspberry Pi, which makes it an excellent choice for someone who is starting to learn how to code in Python. The Pi is remarkably easy to set up as a headless node. This means that, with a few tweaks here and there, you're able to remotely connect to the Raspberry Pi from any other computer, or device, on your home network. For example, once you've set up the remote connectivity options, you can simply plug the Pi into the power socket anywhere in your house within range of your wireless router. As long as the Pi is connected, you will be able to remotely access the desktop from Windows or macOS as easily as if you were sitting in front of the Pi with a keyboard and mouse.

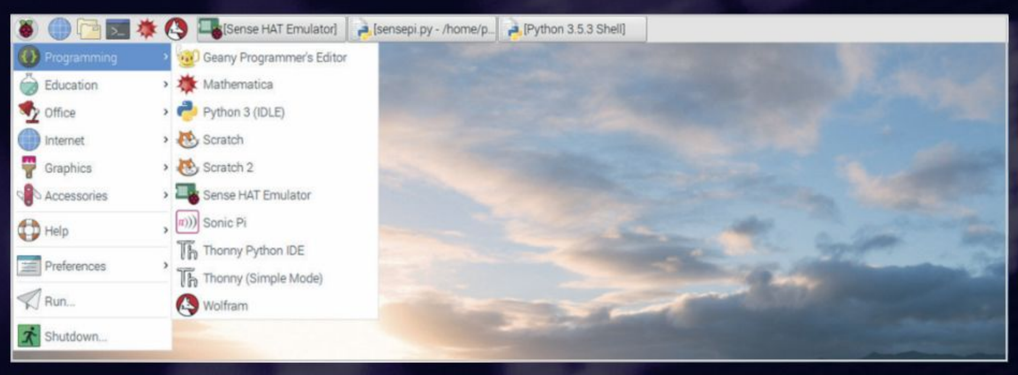
Using this method saves a lot of money, as you don't need another keyboard, mouse and monitor, plus, you won't need to allocate sufficient space to accommodate those extras either. If you're pushed for space and money, then for around £60, buying one of the many kits available will provide the Pi with a pre-loaded SD card (with the latest Raspbian OS), a case, power socket and cables, this is a good idea as you could, with very little effort, leave the Pi plugged into the wall under a desk, while still being able to connect to it and code.

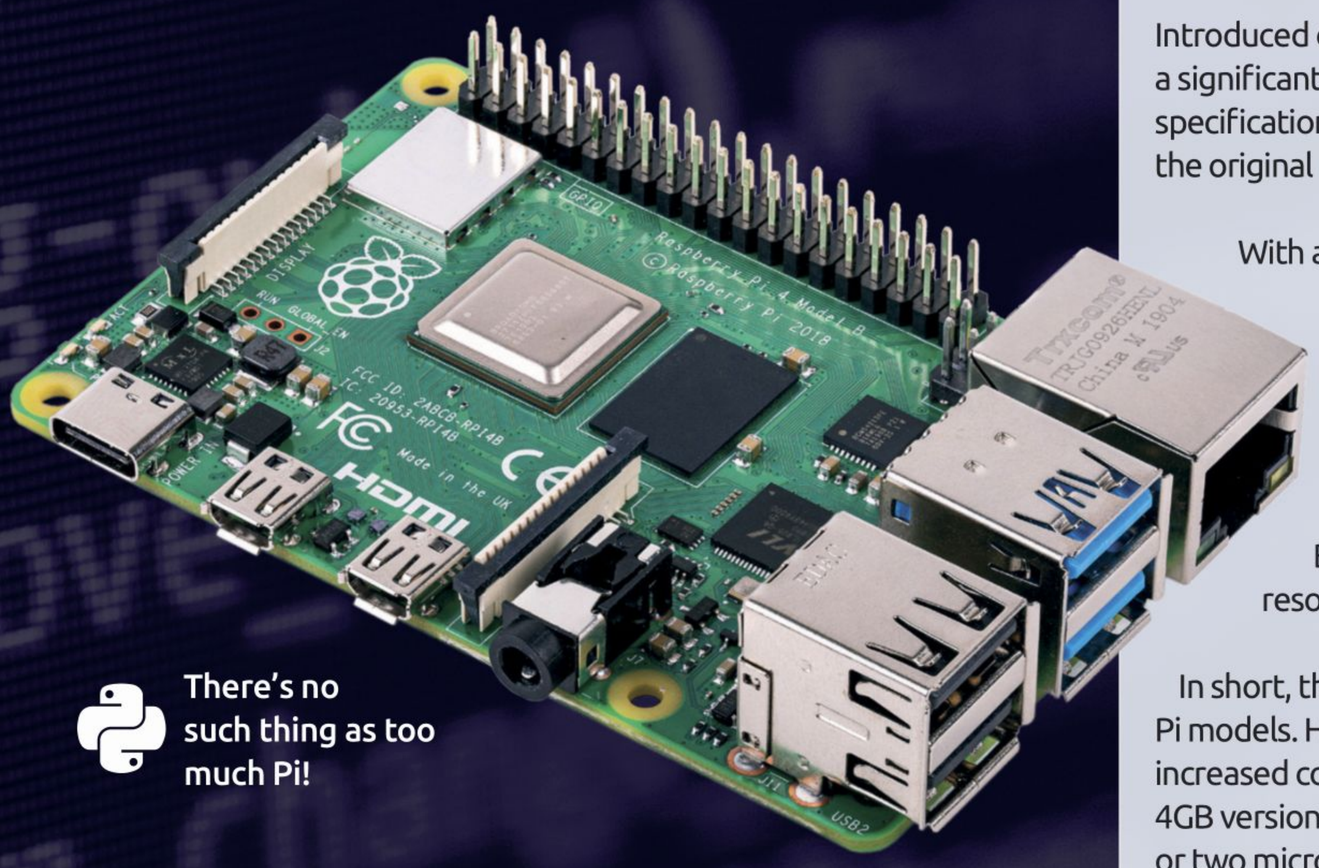
The main advantage is, of course, the extra content that the Raspberry Pi Foundation has included out of the box. The reason for this is that the Raspberry Pi's goal is to help educate the user, whether that's coding, electronics, or some other aspect of computing. To achieve that goal the Pi Foundation includes different IDEs for the user to compile Python code on; as well as both Python 2 and Python 3, there's even a Python library that allows you to communicate with Minecraft.

There are other advantages, such as being able to combine Python code with Scratch (an Object-Oriented programming language developed by MIT, for children to understand how coding works) and being able to code the GPIO connection on the Pi to further control any attached robotics or electronics projects. Raspbian also includes a Sense HAT Emulator (a HAT is a hardware attached piece of circuitry that offers different electronics, robotics and motorisation projects to the Pi), which can be accessed via Python code.

Consequently, the Raspberry Pi is an excellent coding base, as well as a superb project foundation. It is for these, and many other, reasons we've used the Raspberry Pi as our main Python codebase throughout this title. While the code is written and performed on a Pi, you're also able to use it in Windows, other versions of Linux and macOS. If the code requires a specific operating system, then, don't worry; we will let you know in the text.

Everything you need to learn to code with Python is included with the OS!





There's no such thing as too much Pi!

PI 4-EVER!

Introduced on 24th June 2019, the Raspberry Pi 4 Model B is a significant leap in terms of Pi performance and hardware specifications. It was also one of the quickest models, aside from the original Pi, to sell out.

With a new 1.5GHz, 64-bit, quad-core ARM Cortex-A72 processor, and a choice of 1GB, 2GB, or 4GB memory versions, the Pi 4 is one-step closer to becoming a true desktop computer. In addition, the Pi 4 was launched with the startling decision to include dual-monitor support, in the form of a pair of two micro-HDMI ports. You'll also find a pair of USB 3.0 ports, Bluetooth 5.0, and a GPU that's capable of handing 4K resolutions and OpenGL ES 3.0 graphics.

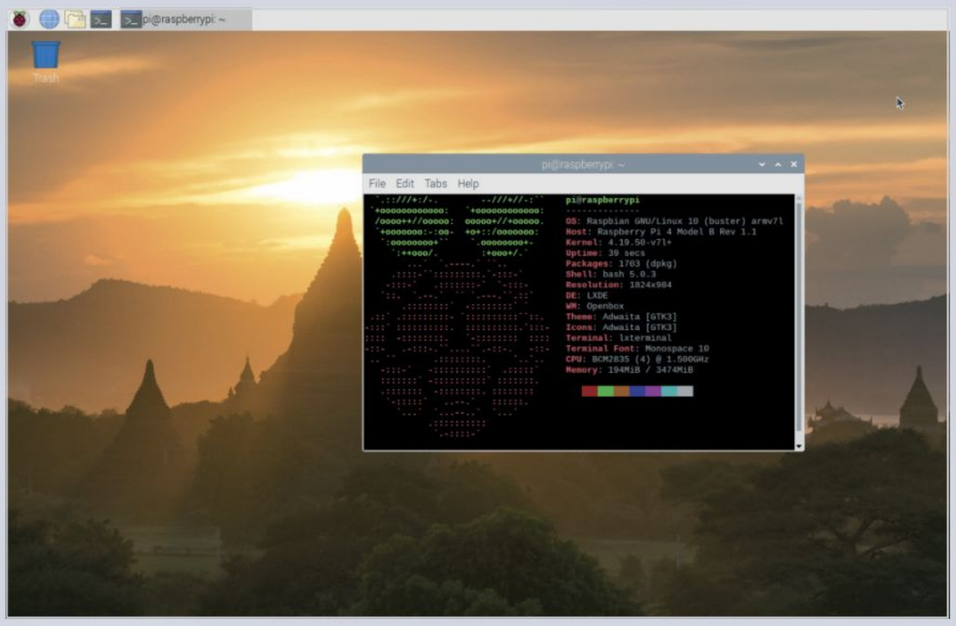
In short, the Pi 4 is the most powerful of the current Raspberry Pi models. However, the different memory versions have an increased cost. The 1GB version costs £34, 2GB is £44, and the 4GB version will set you back £54. Remember to also factor in one or two micro-HDMI cables with your order.

RASPBIAN BUSTER

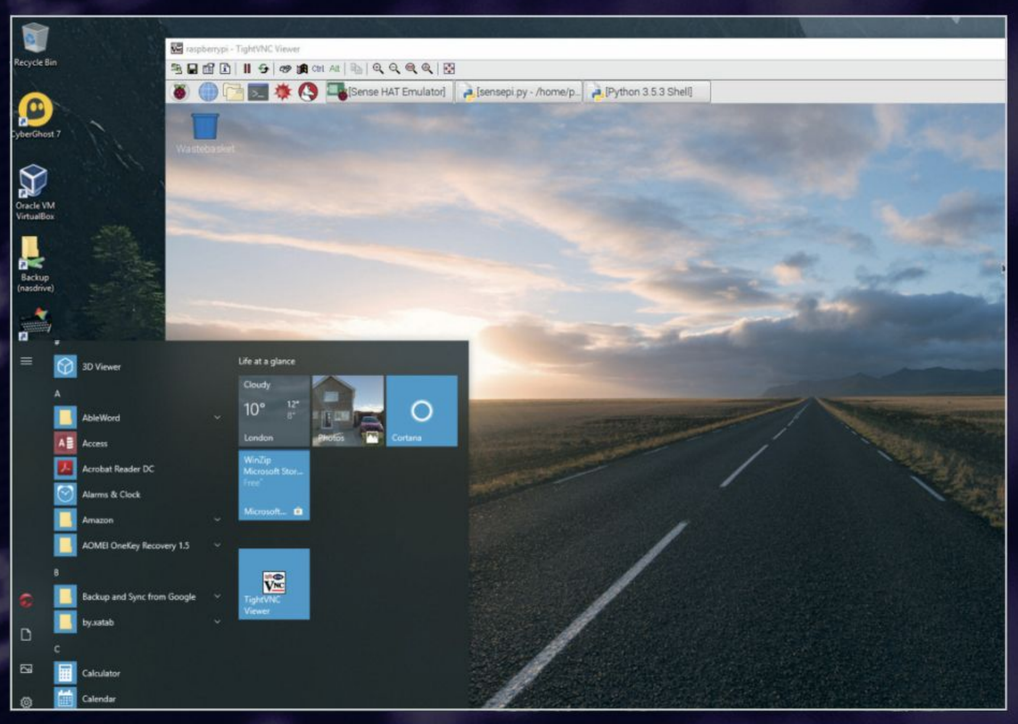
In addition to releasing the Pi 4, the Raspberry Pi team also compiled a new version of the Raspbian operating system, codenamed Buster.

In conjunction with the new hardware the Pi 4 boasts, Buster does offer a few updates. Although on the whole it's very similar in appearance and operation to the previous version of Raspbian. The updates are mainly in-line with the 4K's display and playback, giving the Pi 4 a new set of graphical drivers and performance enhancements.

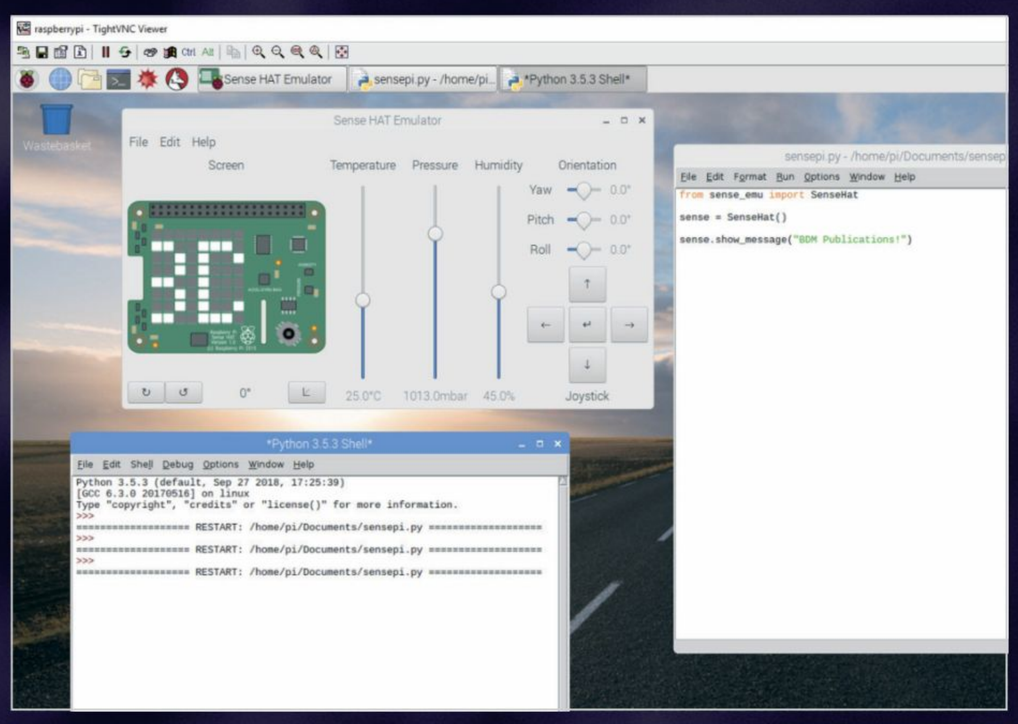
In short, what you see in this book will work with the Raspberry Pi 4 and Raspbian Buster!



Once set up, you can remotely connect to the Pi's desktop from any device/PC.



You can even test connected hardware with Python remotely, via Windows.



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

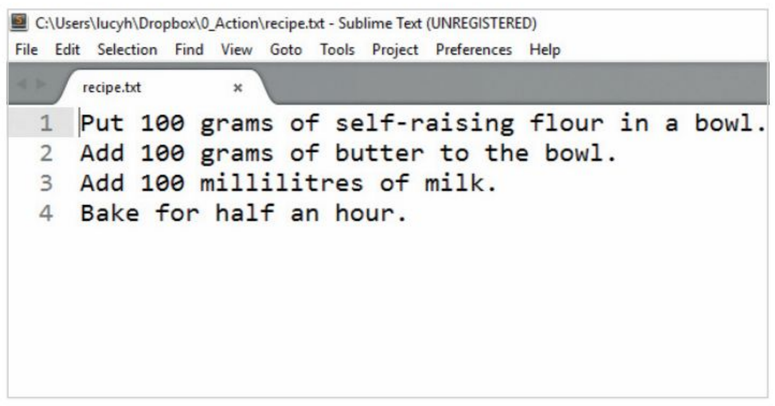
A programming language is a list of instructions that a computer follows. These instructions can be as simple as displaying your name or playing a music file, or as complex as building a whole virtual world. Python is a programming language conceived in the late 1980s by Guido van Rossum at Centrum Wiskunde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

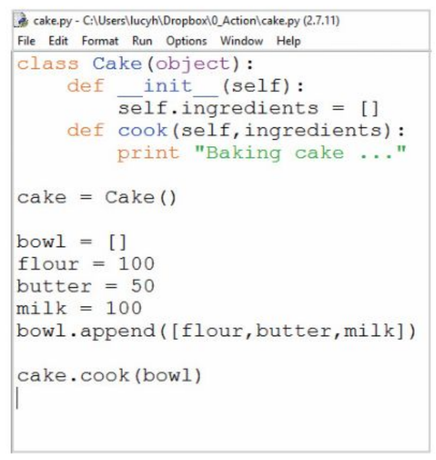
Put 100 grams of self-raising flour in a bowl.
Add 100 grams of butter to the bowl.
Add 100 millilitres of milk.
Bake for half an hour.



CODE

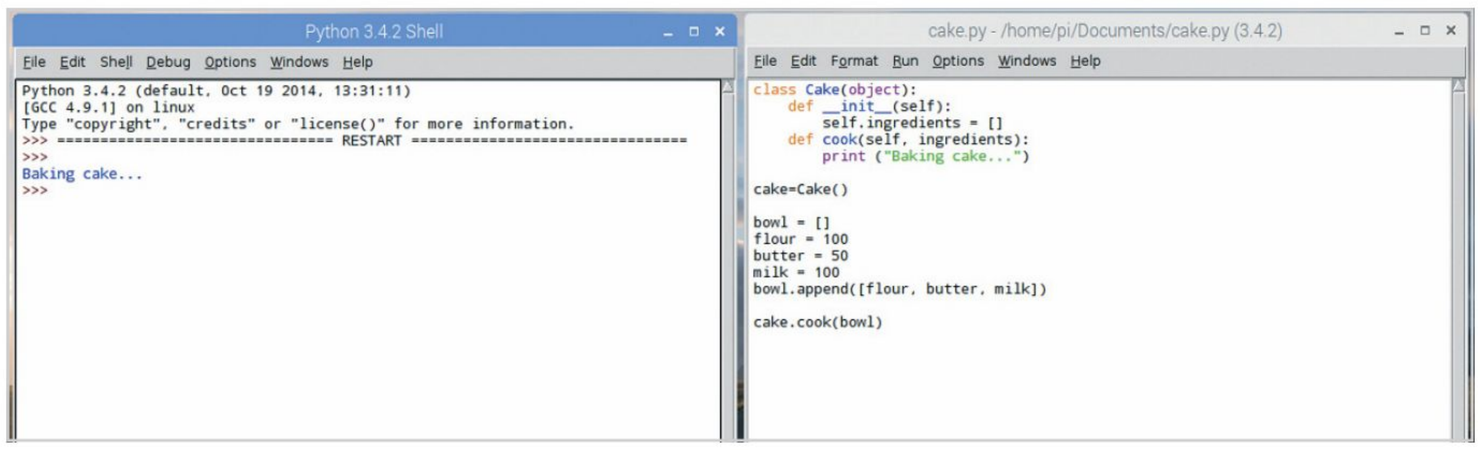
Just like a recipe, a program consists of instructions that you follow in order. A program that describes a cake might run like this:

```
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.

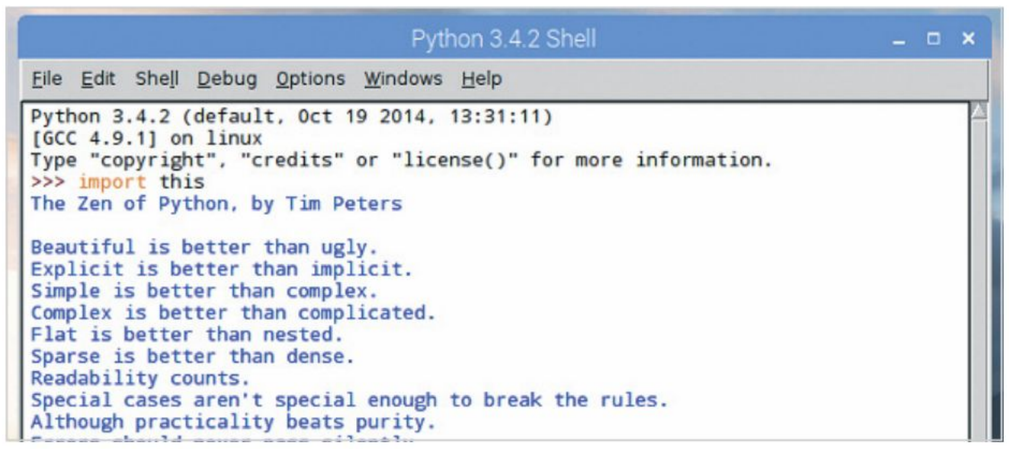


HIGH-LEVEL LANGUAGES

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

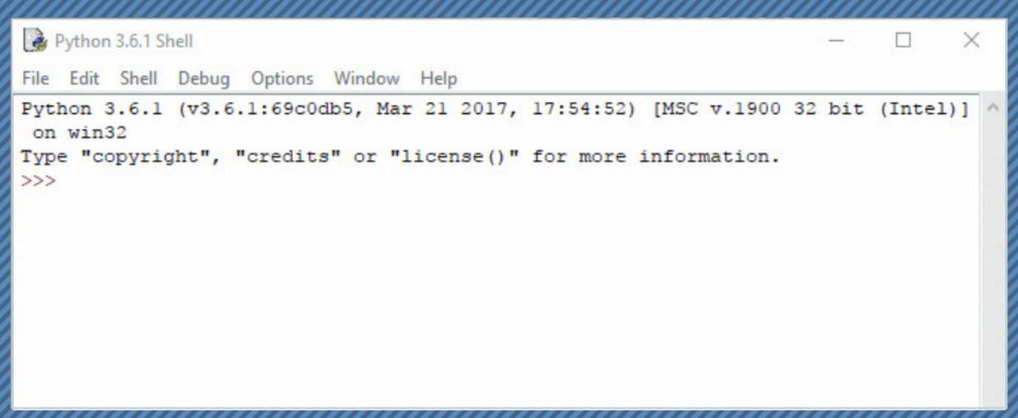
In a typical computing scenario, Python is complicated somewhat by the existence of two active versions of the language: Python 2 and Python 3.

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.

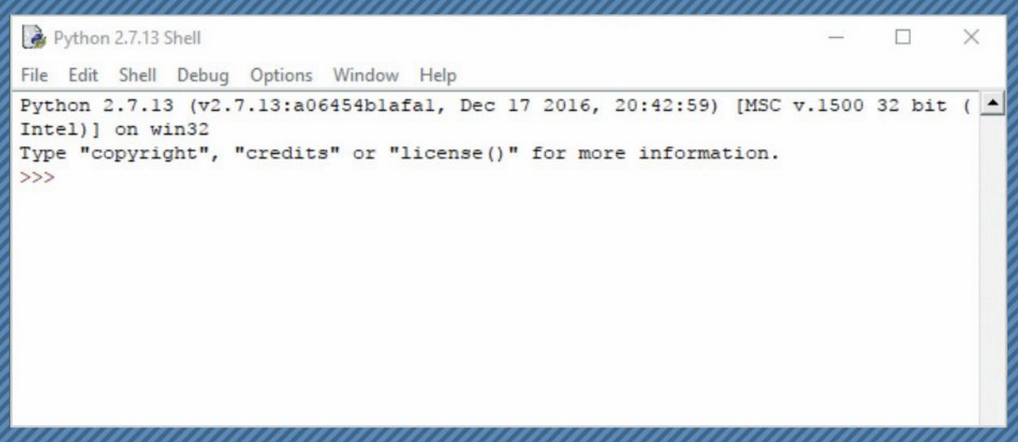
PYTHON 3.X

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.



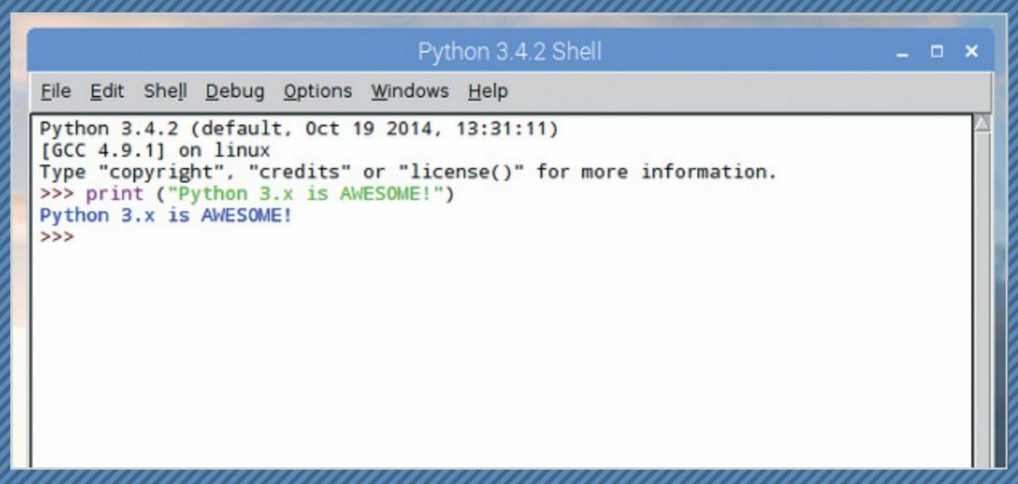
PYTHON 2.X

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.

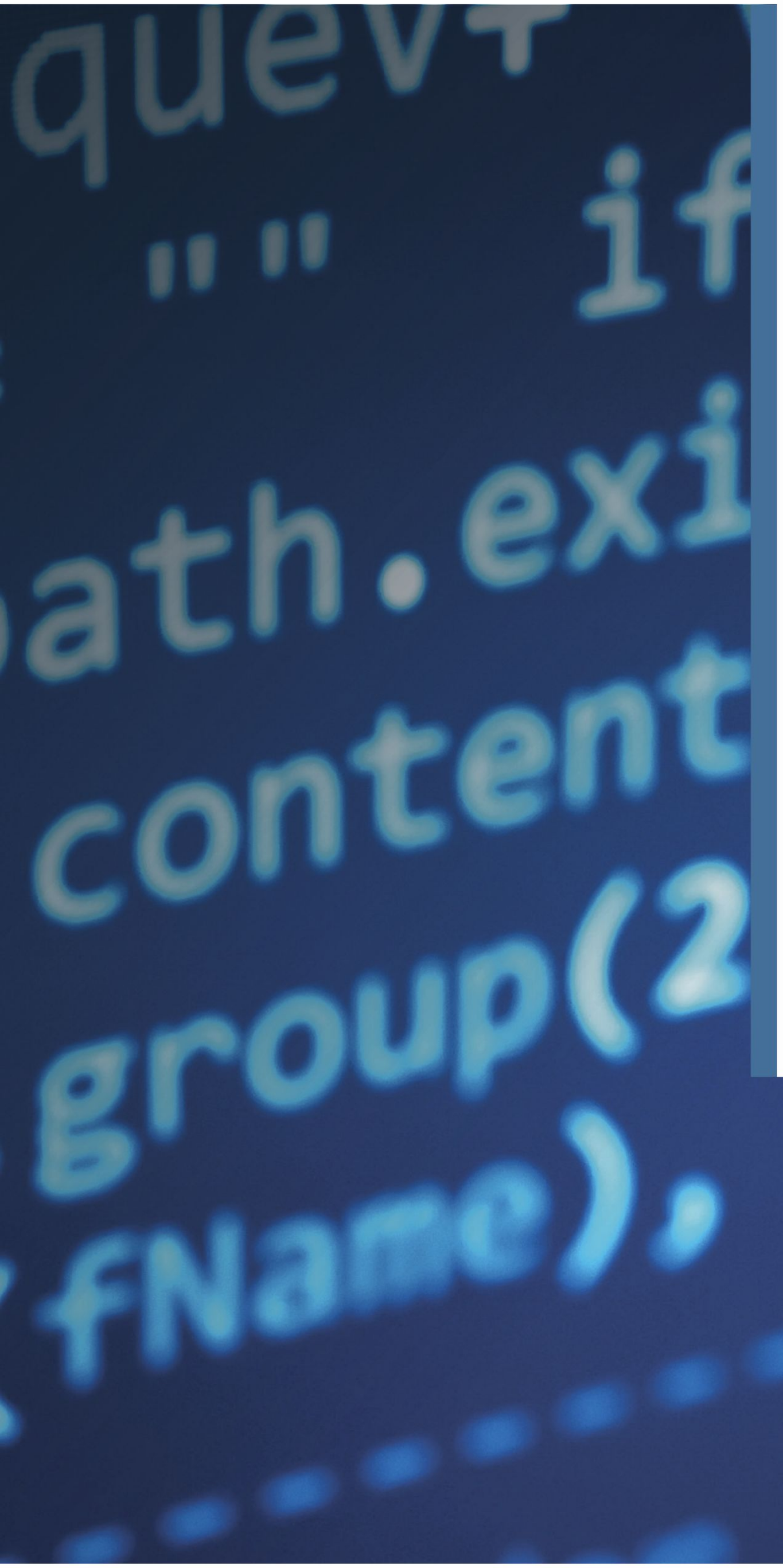


3.X WINS

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.



First Steps into Python



Now that you have the latest version of Python installed, you can begin to get programming. These are your first steps in the wider world of Python and we're here to help you write your first piece of code, save it, and run it in the Python IDLE Shell.

We cover variables, numbers and expressions, user input, conditions and loops, and the types of errors you will undoubtedly come across in your time with Python. Let's start and see how to get coding.

38 Starting Python for the First Time
40 Your First Code
42 Saving and Executing Your Code
44 Executing Code from the Command Line
46 Numbers and Expressions
48 Using Comments
50 Working with Variables
52 User Input
54 Creating Functions
56 Conditions and Loops
58 Python Modules
60 Python Errors
62 Combining What You Know So Far

Starting Python for the First Time

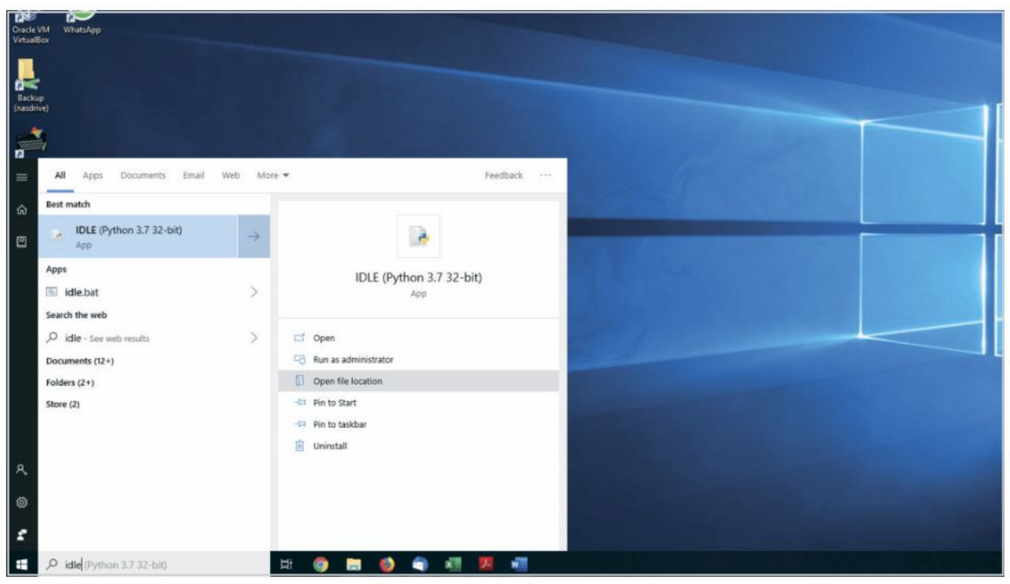
We're using Python 3 under Windows 10 for these following examples. Don't worry if your version of Python is 3.4.2, or something lesser than the current version, as long as you're using Python 3, the code will work.

STARTING PYTHON

As when learning anything new, you need to start slow. You can pick up the pace as your experience grows, but for now, let's just get something appearing on the screen. Don't worry, you'll soon be coding like a pro!

STEP 1

Click on the Windows Start button, and start typing 'idle'. The result will be the currently installed version of Python, **IDLE (Python 3.7 32-bit)**, for example. You can Pin it to the Start for convenience, otherwise simply click the icon to launch the Python Shell.



STEP 2

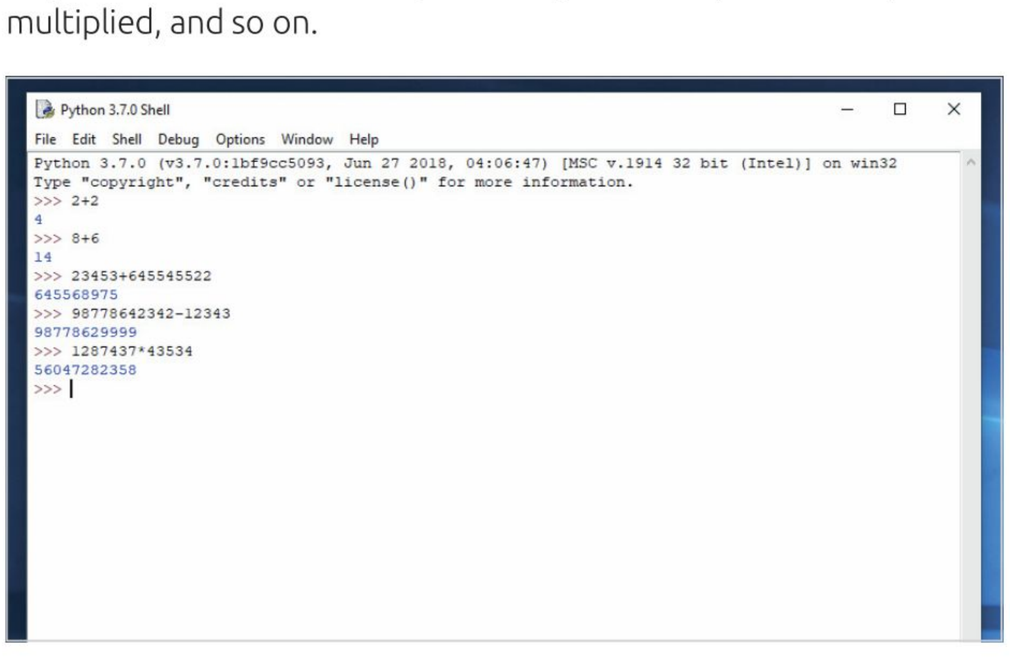
The Shell is where you can enter code and see the responses and output of code you've programmed into Python. This is a kind of sandbox, if you will, where you're able to try out some simple code and processes.

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

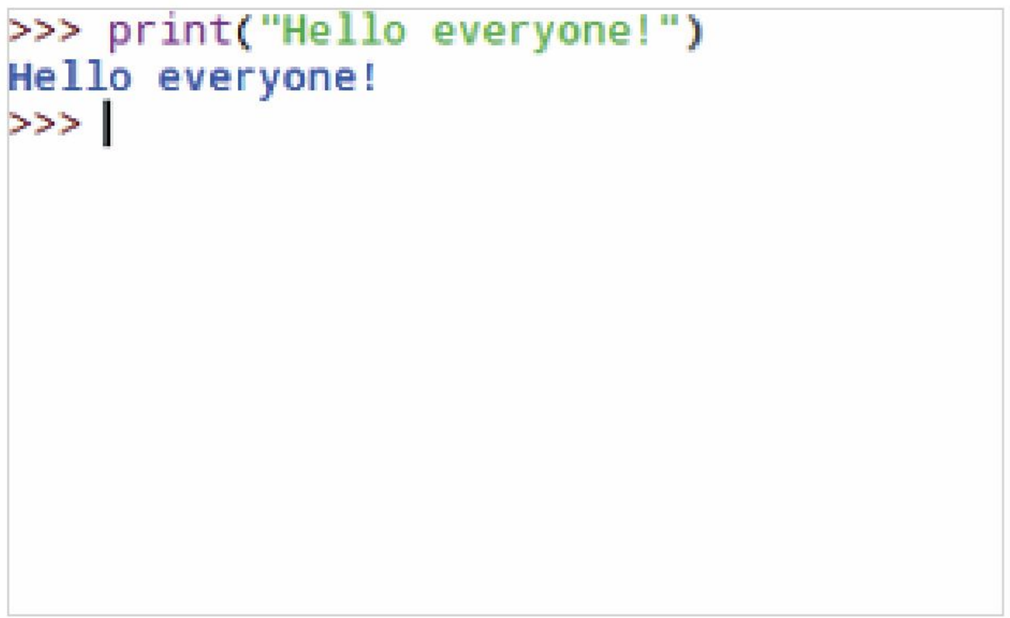
While that's very interesting, it's not particularly exciting. Instead, try this:

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

Just enter it into the IDLE as you've done 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 parentheses as well as quotes in order to output content to the screen, in this case the 'Hello everyone!' bit.



STEP 7

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

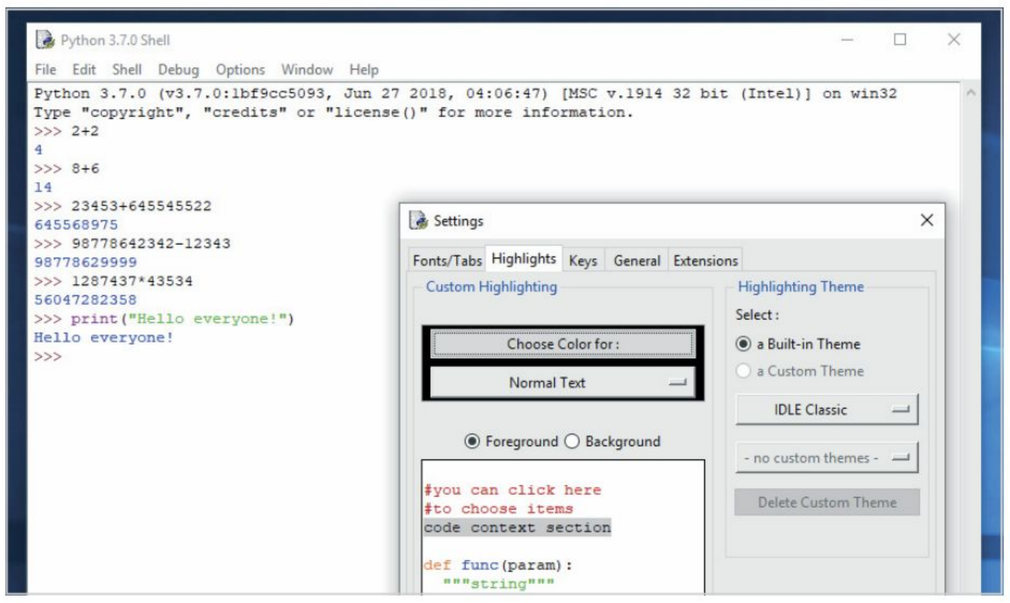
Black – Data and Variables
Green – Strings
Purple – Functions
Orange – Commands
Blue – User Functions
Dark Red – Comments
Light Red – Error Messages

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:`

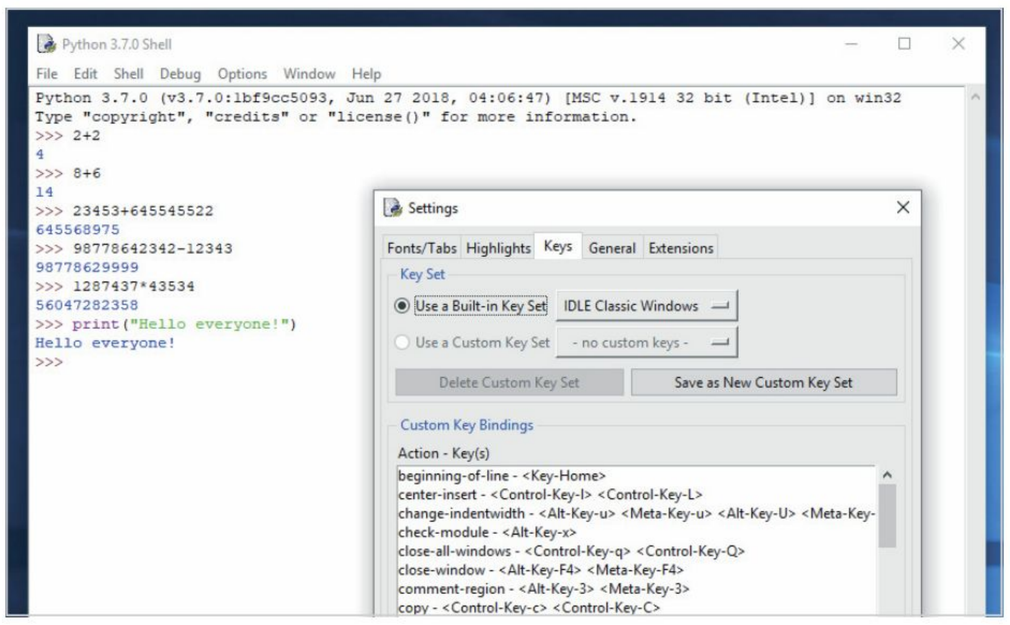
STEP 8

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.



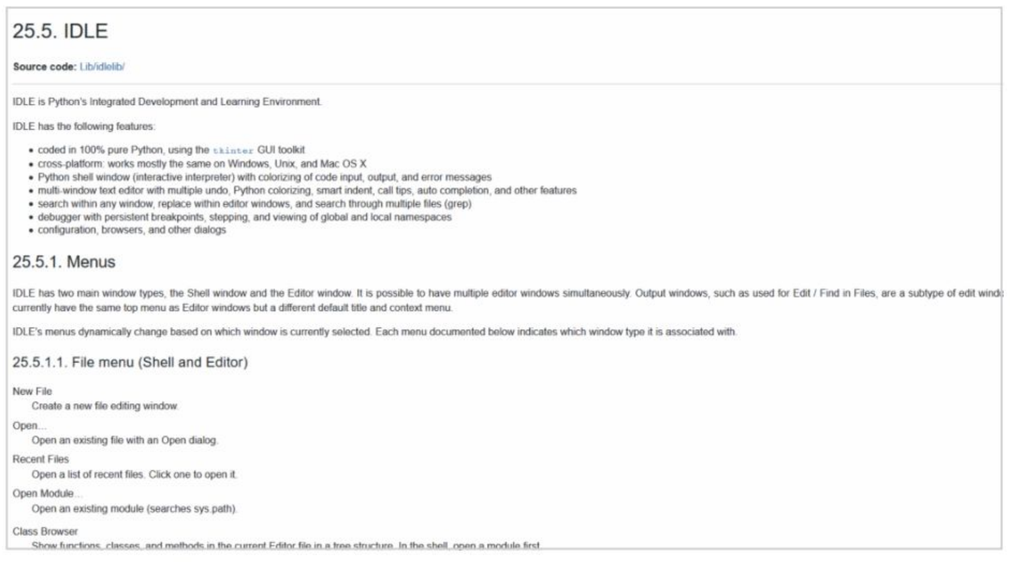
STEP 9

As with most programs available, regardless of the operating system, there are numerous shortcut keys. We don't have room for them all here, but within the Options > Configure IDLE and under the **Keys** tab, you'll 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 **https://docs.python.org/3/library/idle.html**, which details many of the IDLE's features.



Your First Code

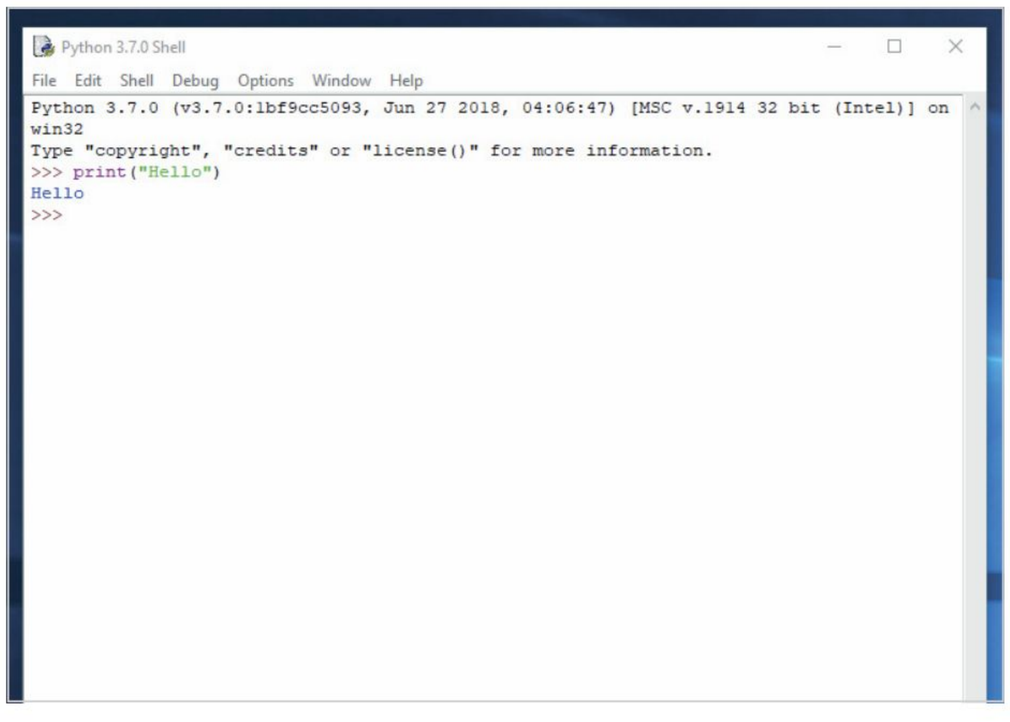
Essentially, you've already written your first piece of code with the print("Hello everyone!") function from the previous tutorial. However, let's expand that and look at entering your code and playing around with some other Python examples.

PLAYING WITH PYTHON

As 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, re-open it as you did in the previous page. In the Shell, enter the familiar following:

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



STEP 2 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'll notice 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'll need to remove the quotes:

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

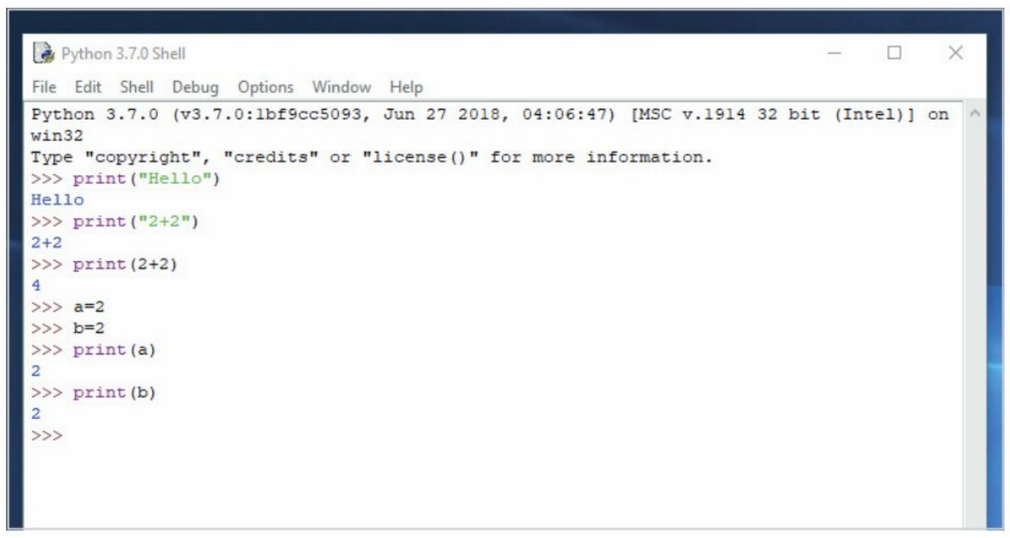
STEP 4 You can continue as such, printing 2+2, 464+2343 and so on to the Shell. An easier way is to use a variable, which is something we will cover in more depth later. For now, enter:

```
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)
```

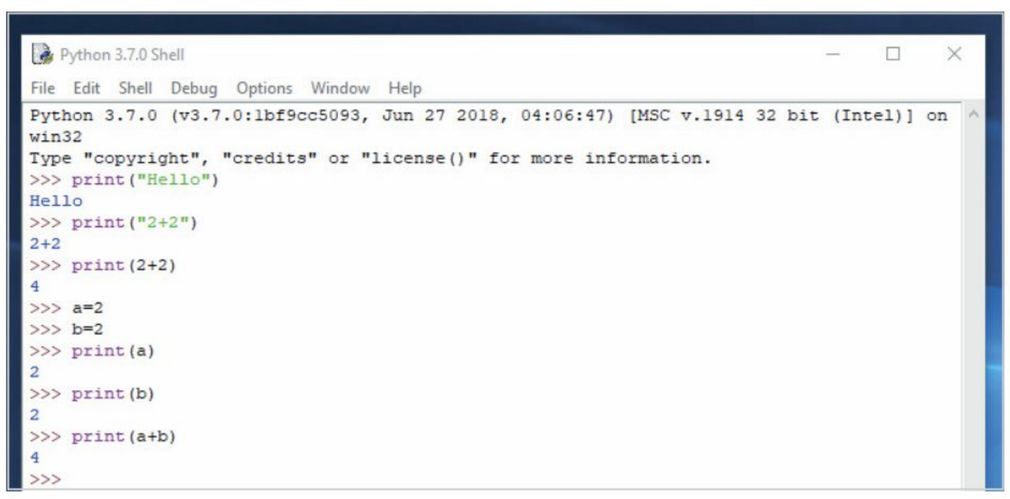


STEP 6

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

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

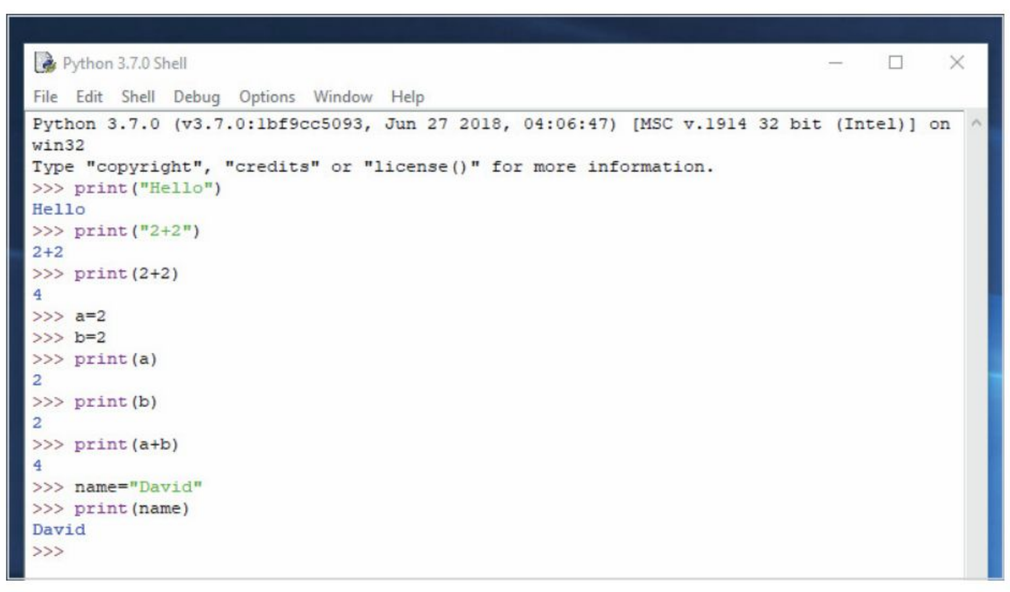
This code takes the value of both a and b, adds them together, and outputs the result.



STEP 7

You can play around with different kinds of variables together with the Print function. For example, we could assign variables for someone's name:

```
name="David"
print(name)
```

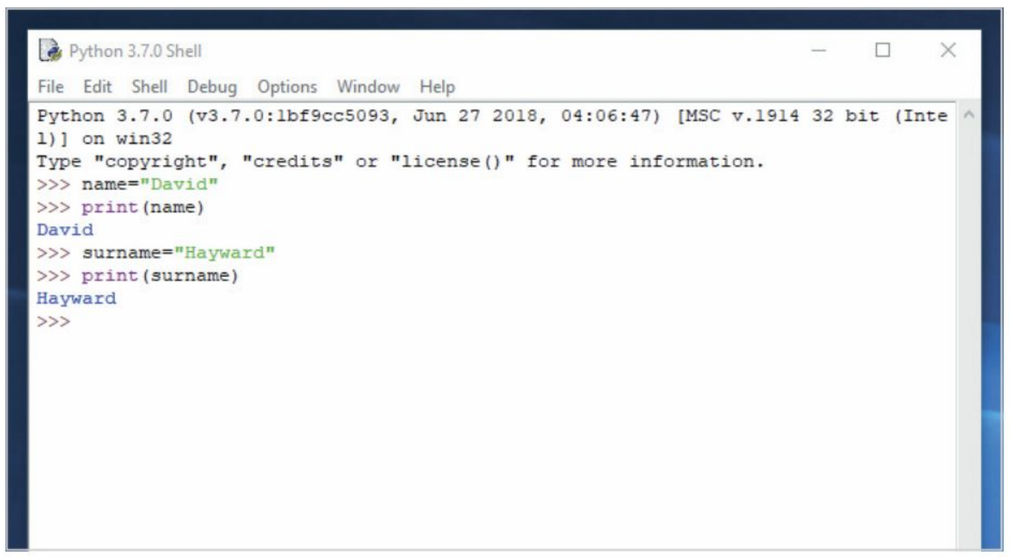


STEP 8

Now let's add a surname:

```
surname="Hayward"
print(surname)
```

We now have two variables containing both a first name and a surname, and we can print them independently.

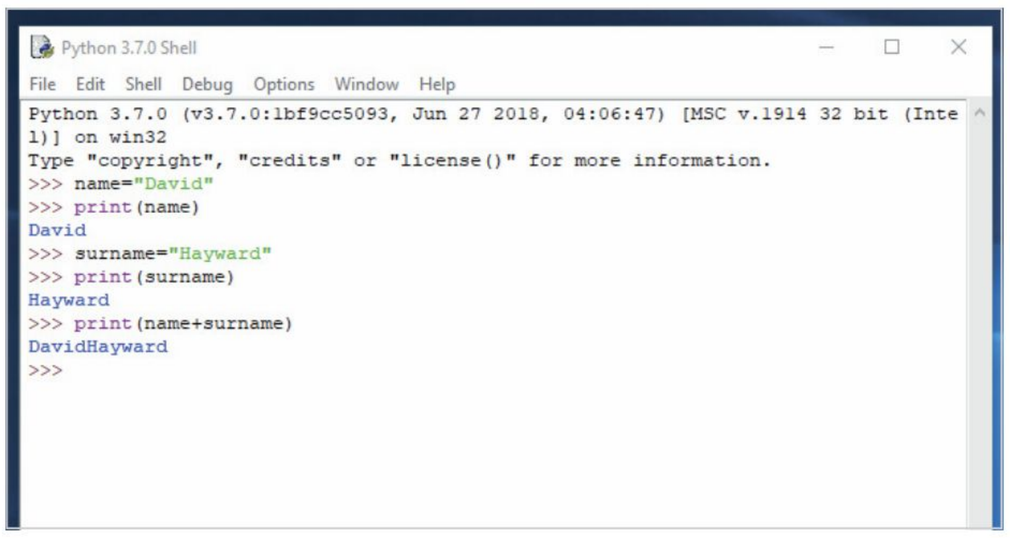


STEP 9

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)
```

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



STEP 10

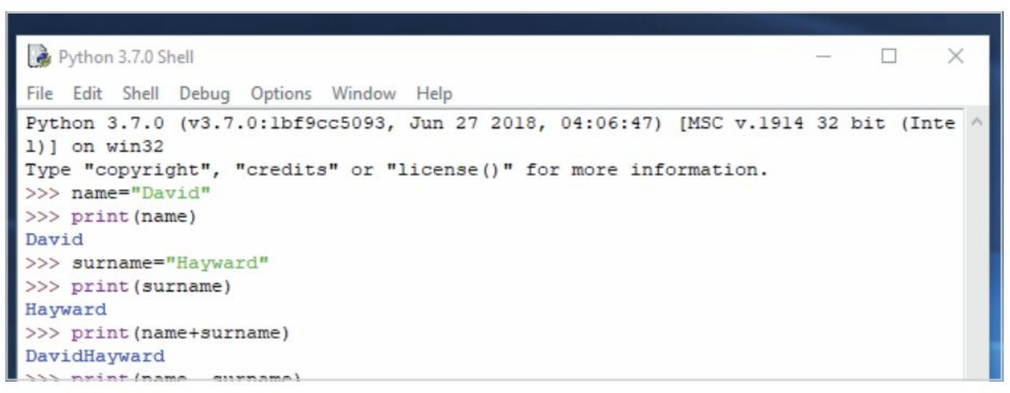
In Python 3 we can separate the two variables with a space by using a comma:

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

Alternatively, you can add the space yourself:

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

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



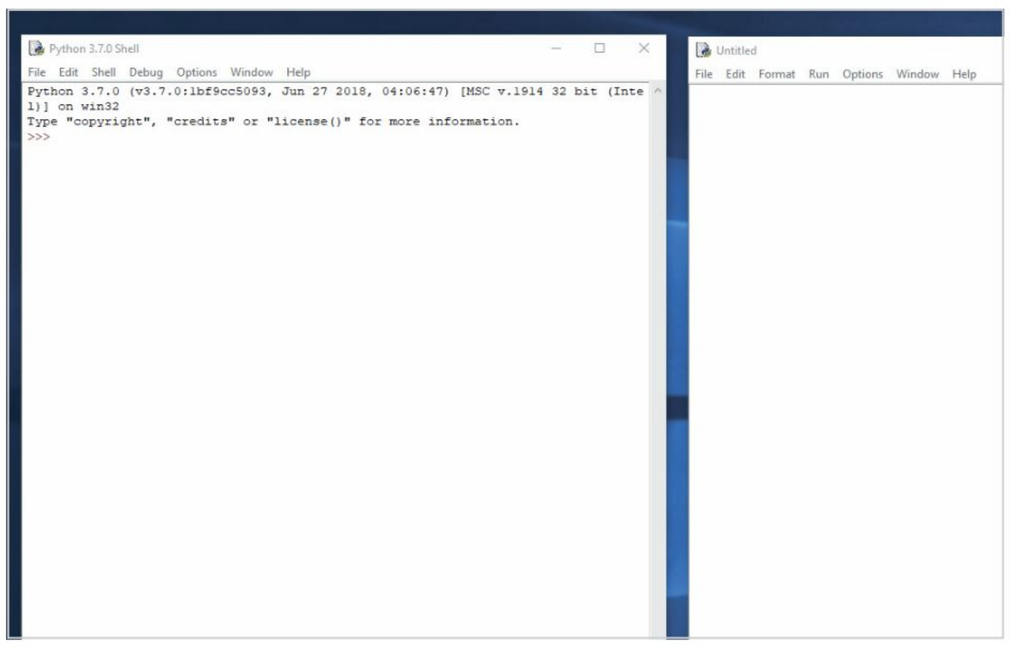
Saving and Executing Your Code

While working in the IDLE Shell is perfectly fine for snippets of code, it's not designed for entering longer program listings. In this section, we'll introduce you to the IDLE Editor, where most of our code will be entered from now on.

EDITING CODE

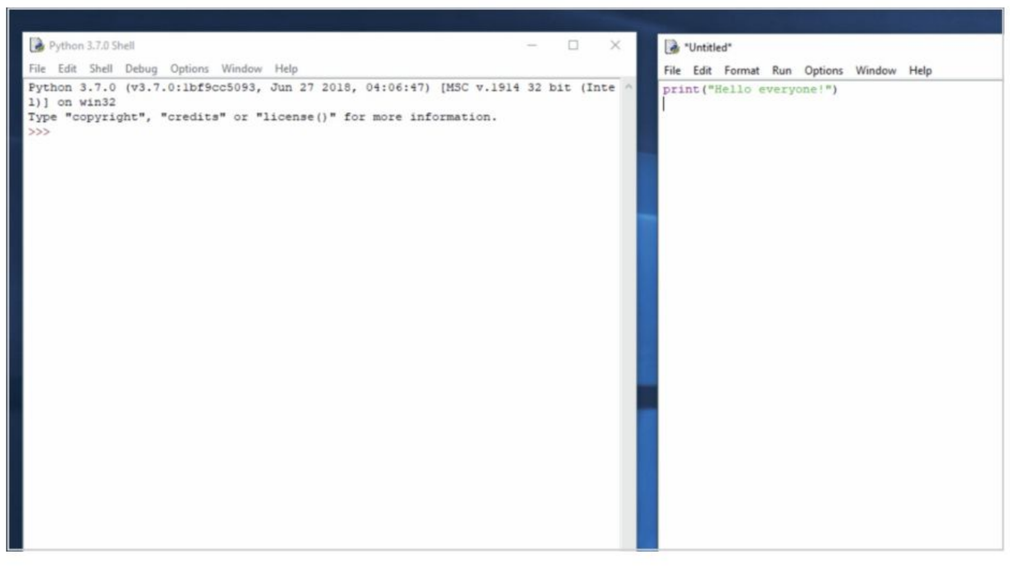
You will eventually reach a point where you have to move on from inputting single lines of code into the Shell. Instead, the IDLE Editor will allow you to save and execute your Python code.

STEP 1 First, open the Python IDLE Shell. 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 you need to create your future programs.

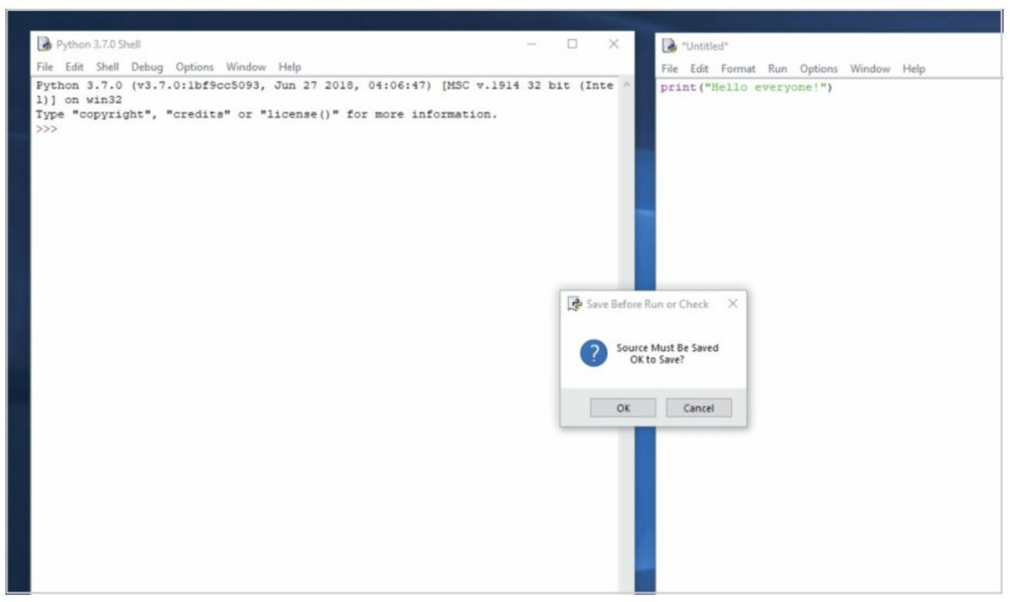


STEP 2 The IDLE Editor is, for all intents and purposes, a simple text editor with Python features, colour coding and so on. You enter code as you would within the Shell, so taking an example from the previous tutorial, enter:

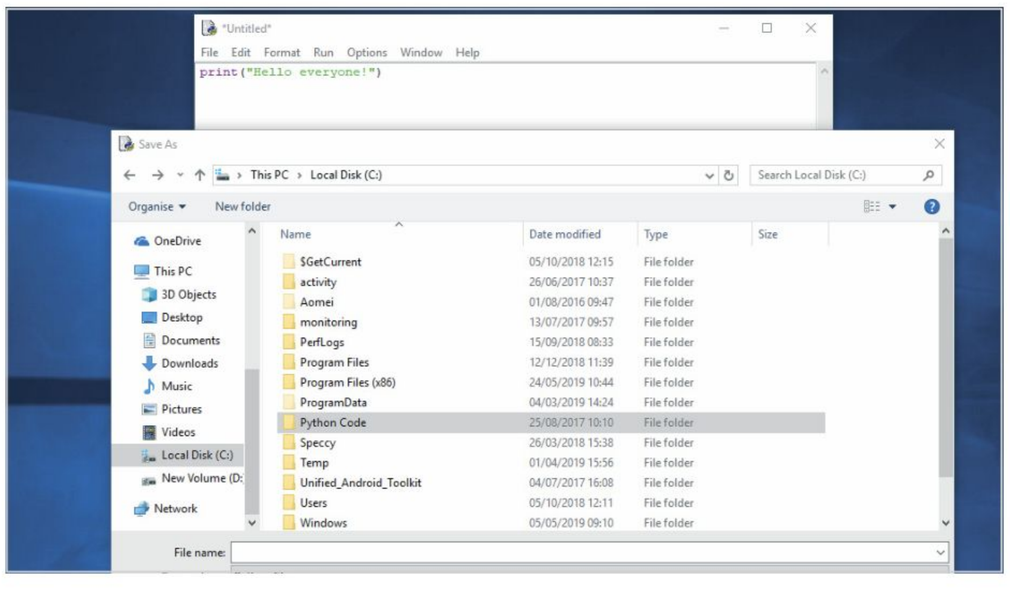
```
print("Hello everyone!")
```



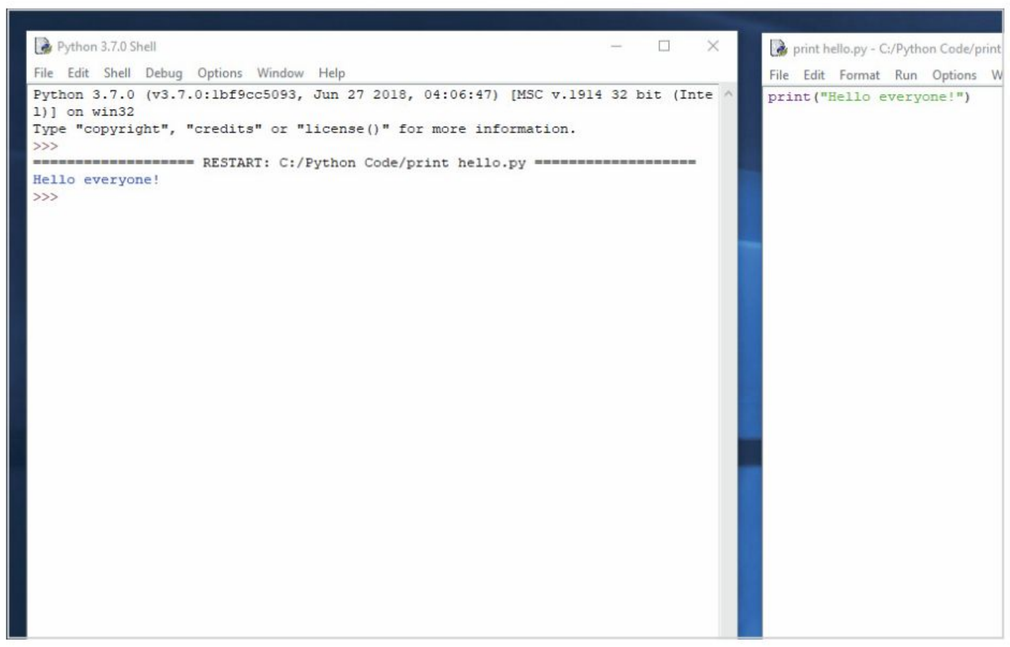
STEP 3 As you can see the same colour coding is in place in the IDLE Editor as it is in the Shell, enabling you to better understand what's going on with your code. To execute the code, however, you need to first save it. Press **F5** and you'll have a Save…Check box open.



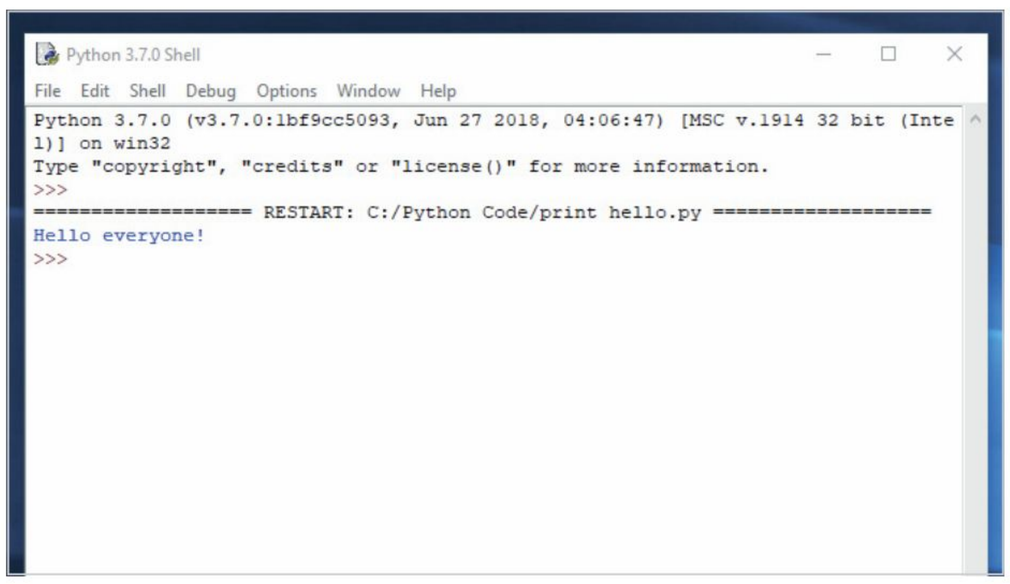
STEP 4 Click on the **OK** button in the Save box, and select a destination where you'll save all your Python code. The destination can be a dedicated folder called Python, or you can just dump it wherever you like. Remember to keep a tidy file system, though, it'll help you out in the future.



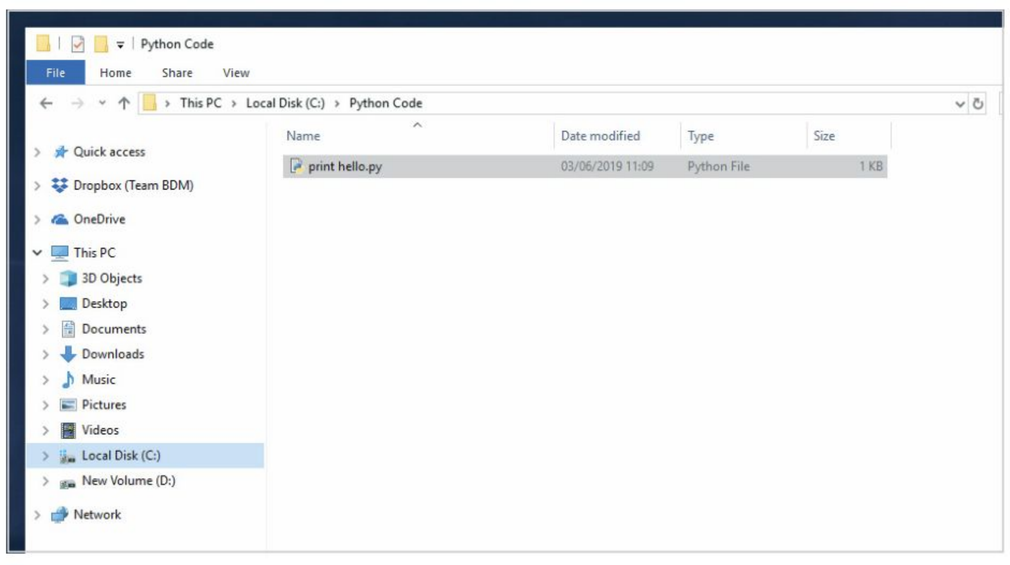
STEP 5 Enter a name for your code, 'print hello' for example, and click on the Save button. As soon as the Python code is saved, it's executed and the output will be detailed in the IDLE Shell; In this case, the words 'Hello everyone!'.



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 and, unless otherwise stated, this is the method we will use.



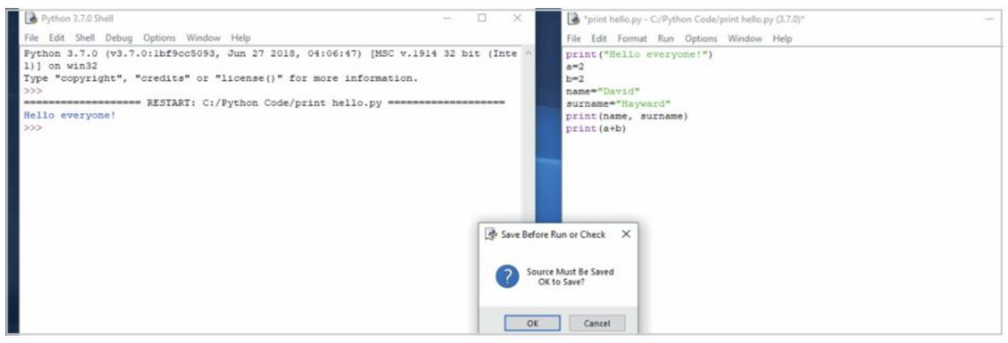
STEP 7 If you open the file location of the saved Python code, you'll notice that it ends in a .py extension. This is the default Python filename, any code you create will be whatever.py, and any code downloaded from the many Internet Python resource sites will be .py. Just ensure that the code is written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



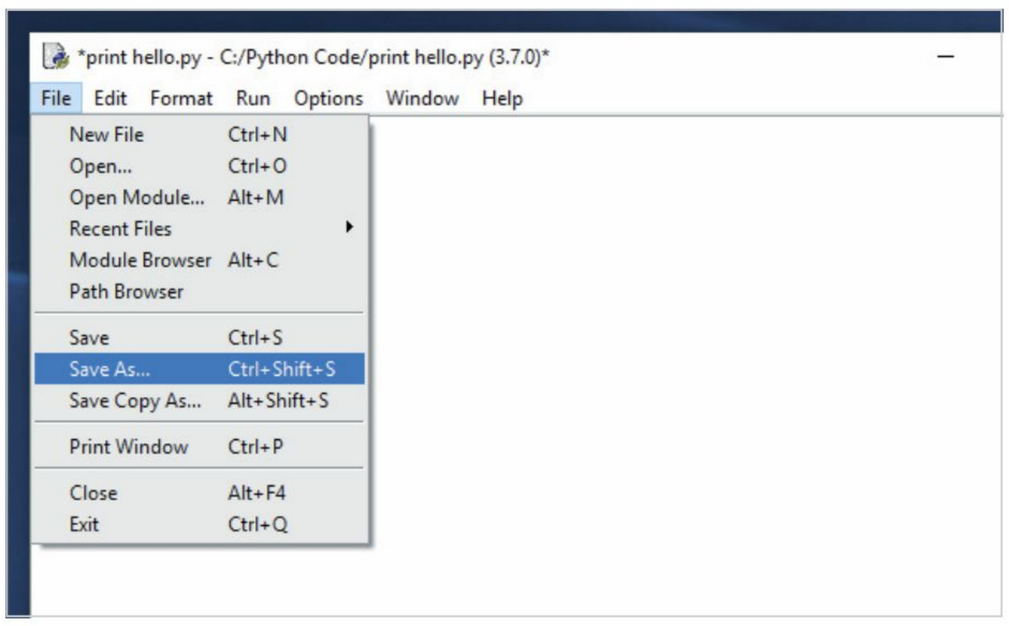
STEP 8 Let's extend the code and enter a few examples from the previous tutorial:

```
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.



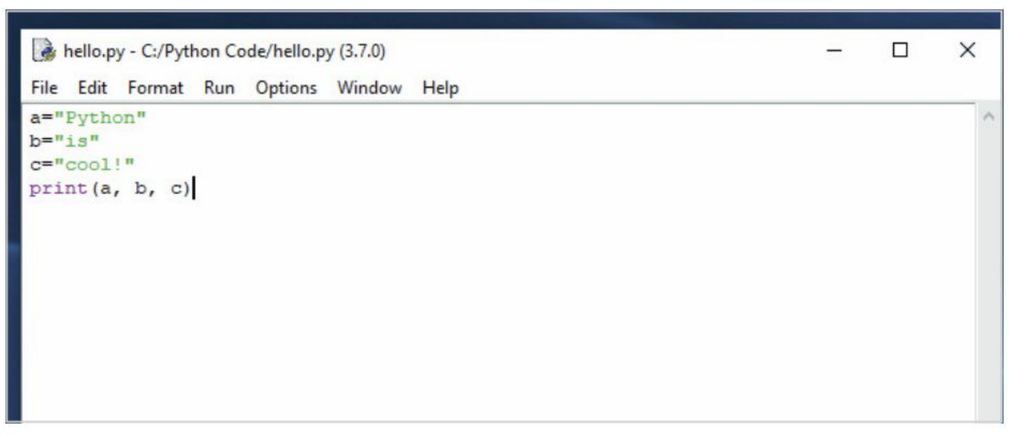
STEP 9 If you click the **OK** button the file will be overwritten with the new code entries, and executed; with the output in the Shell. It's not a problem with just these few lines, but if you were to edit a larger file overwriting can become an issue. Instead, use **File > Save As** from within the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

We will use this code in the next tutorial.



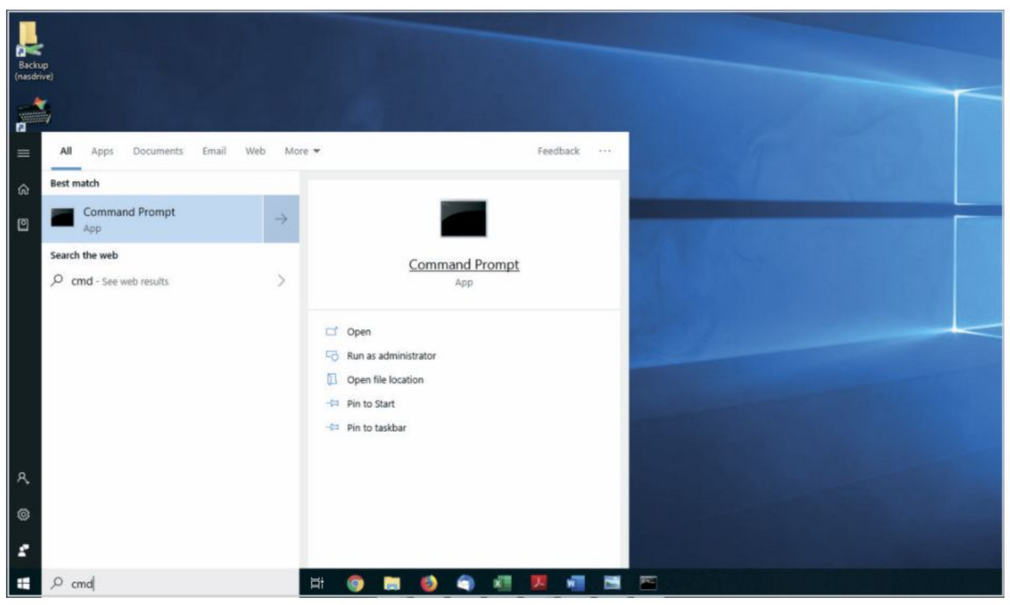
Executing Code from the Command Line

While we're going to be working from the GUI IDLE, it's worth taking a moment to look at Python's command line handling. Sometimes, depending on the code you write, executing via the command line is a better solution over the IDLE.

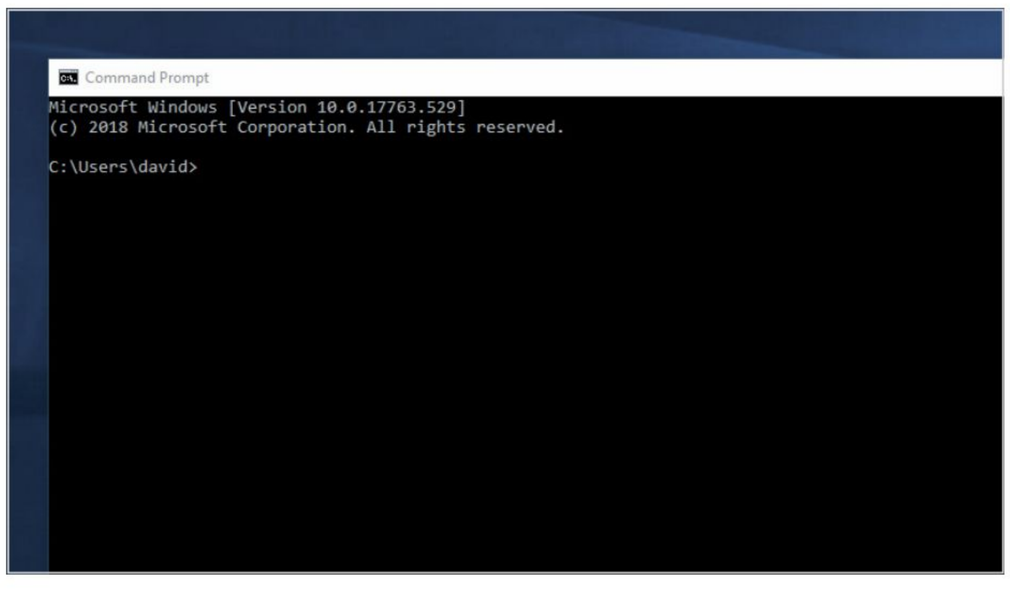
COMMAND THE CODE

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

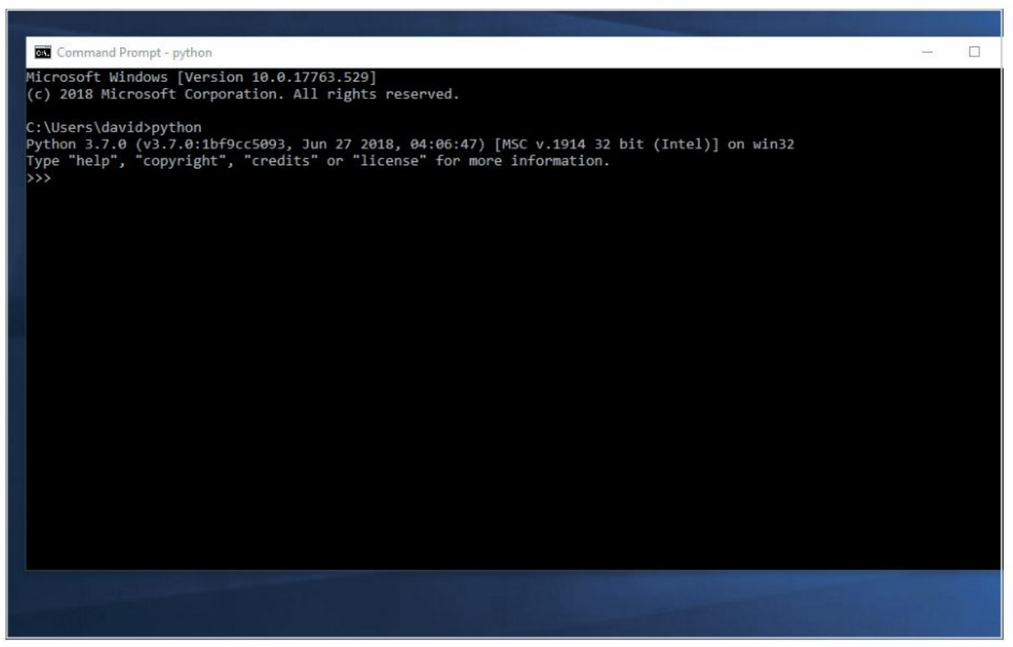
STEP 1 When you first installed Python, the installation routine automatically included all the necessary components to allow the execution of code outside of the GUI IDLE; in other words, the command line. To begin with, click on the Windows Start Button, and type: `cmd`.



STEP 2 As you did when launching the Python IDLE, click on the returned result from the search, the Command Prompt App. This will launch a new window, with a black background and white text. This is the command line, also called a Terminal in macOS, Linux, and Raspberry Pi operating systems.



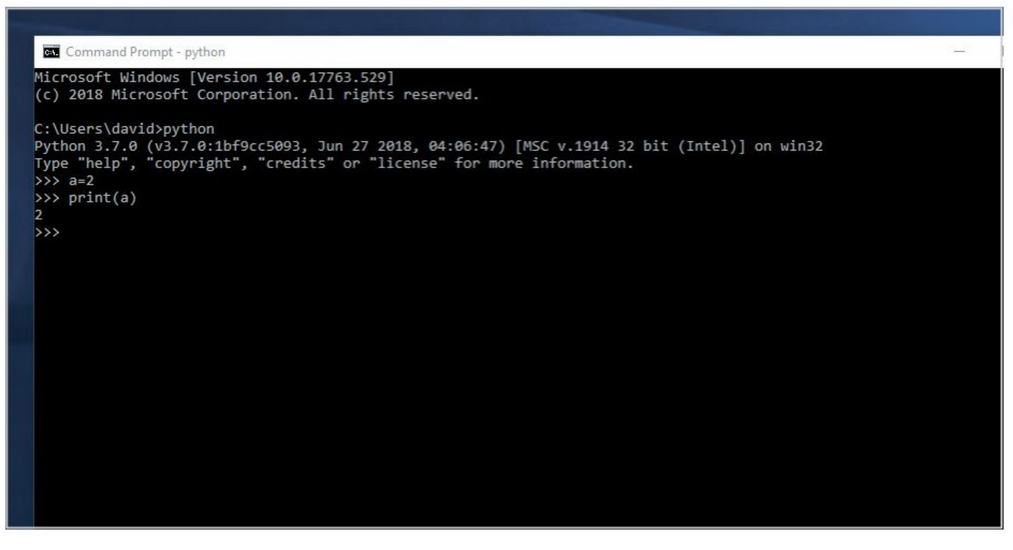
STEP 3 Now you're at the command line, we can start Python using the command `python` and pressing the Enter key. This will put you into the command line version of the Shell, with the familiar, three right-facing arrows as the cursor (>>>).



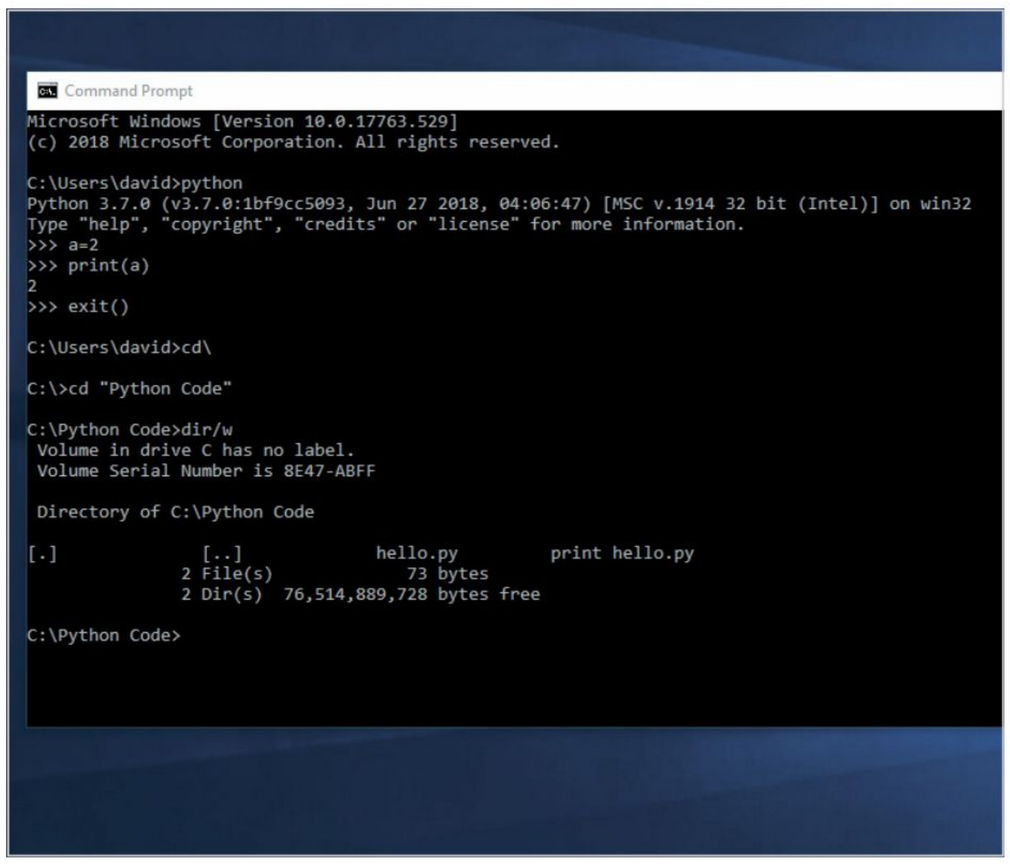
STEP 4 From here you're able to enter the code you've looked at previously, such as:

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

As you can see, it works exactly the same.



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

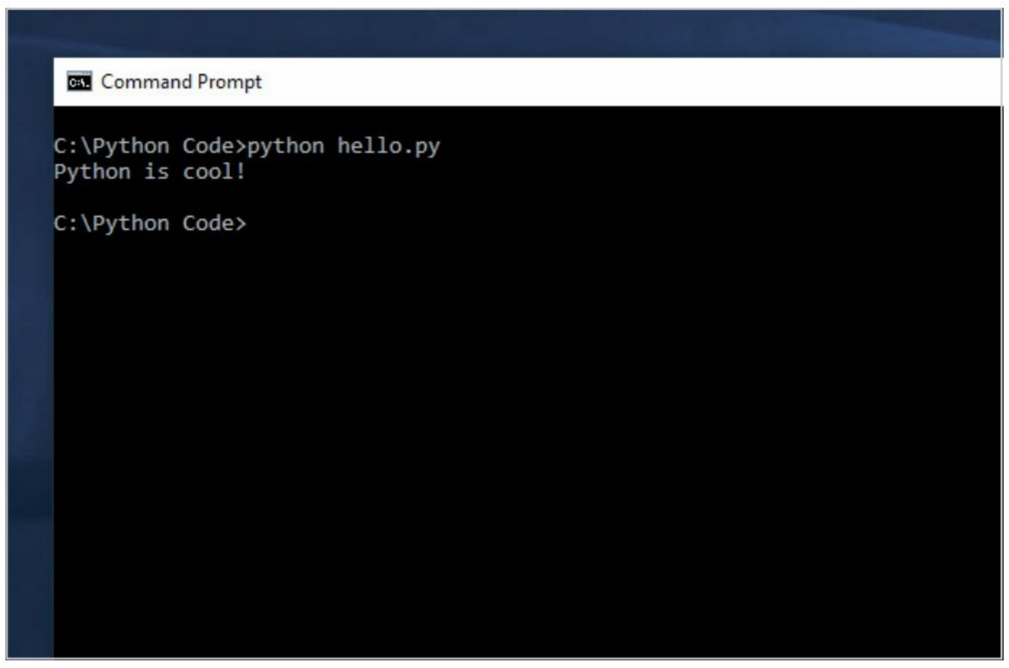


STEP 6 From within the same folder as the code you're going to run, enter the following into the command line:

```
python hello.py
```

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

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



DIFFERENT VERSIONS OF PYTHON

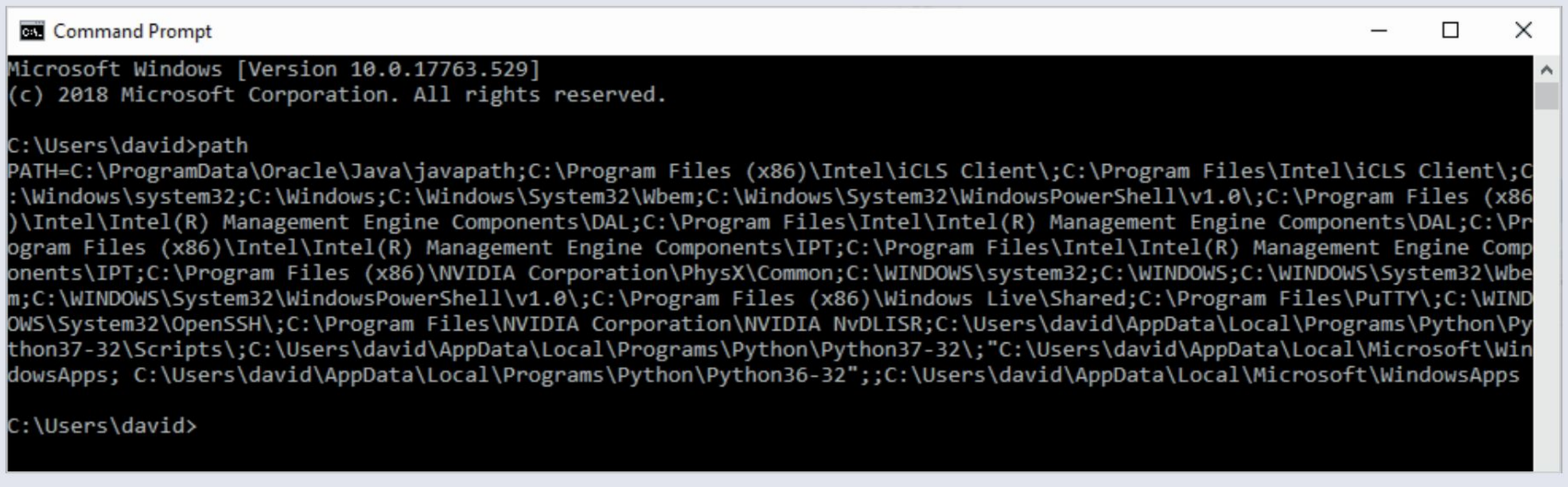
If you've previously used Python 3 on a Mac or Linux, and subsequently the Raspberry Pi, you may be a little confused as to why the Windows version of Python uses the command line: **python**, instead of **python3**.

The reason behind this is that UNIX-like systems, such as macOS and Linux, already have Python libraries pre-installed. These older libraries are present because some of the macOS and Linux system utilities rely on Python 2, and therefore installing a newer version of Python, and thus altering the executable name, could have dire consequences to the system.

As a result, developers decided that the best approach for macOS and Linux systems would be to leave the command line **'python'** as exclusive Python 2 use, and newer versions of user-installed Python would be **'python3'**.

This isn't an issue with Windows, as it doesn't use any Python libraries other than the ones installed by the user themselves when actually installing Python. When a Windows user installs Python, the installation wizard will auto-include the command line instance to the core Windows PATH variable, which you can view by entering: `path` into the command line. This points to the python.exe file required to execute Python code from the command line.

We don't recommend you install both Python 2 and Python 3 within Windows 10; naturally, you can if you want, but realistically, although Python 2 still has a foothold in the coding world, Python 3 is the newest version. However, if you do, then you will need to rename one of the Python versions names; as they will be installed in different folders and both use python.exe as the command line executable. It's a little long-winded, so unless there's a dire need to have both versions of Python installed, it's best to stick to Python 3.



18+

WHAT IS AVAXHOME?

AVAXHOME-

the biggest Internet portal, providing you various content: brand new books, trending movies, fresh magazines, hot games, recent software, latest music releases.

Unlimited satisfaction one low price
Cheap constant access to piping hot media
Protect your downloadings from Big brother
Safer, than torrent-trackers

18 years of seamless operation and our users' satisfaction

All languages
Brand new content
One site

AVXLIVE.ICU

AvaxHome - Your End Place

We have everything for all of your needs. Just open https://avxlive.icu

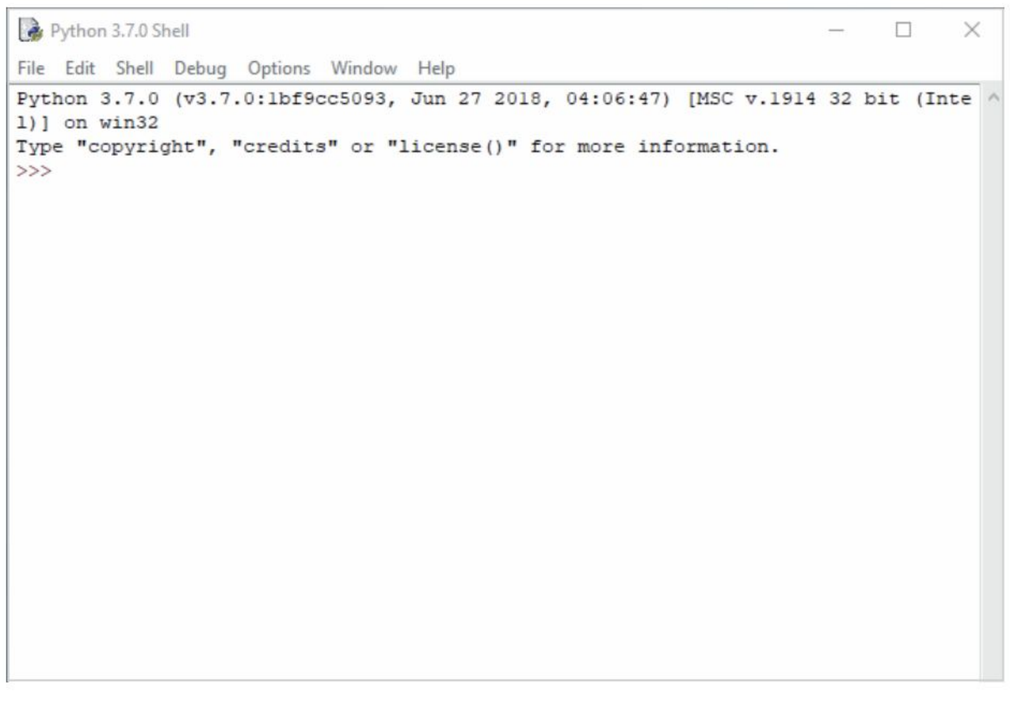
Numbers and Expressions

We've seen some basic mathematical expressions with Python, simple addition and the like. Now let's expand on that, 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 from the mathematical powers of Python, as maths is the driving force behind the code with most, if not all, programming languages.

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, we're going to use the Shell. If you've opted to use a third-party text editor, note that you need to get to the IDLE Shell for this part of the tutorial.



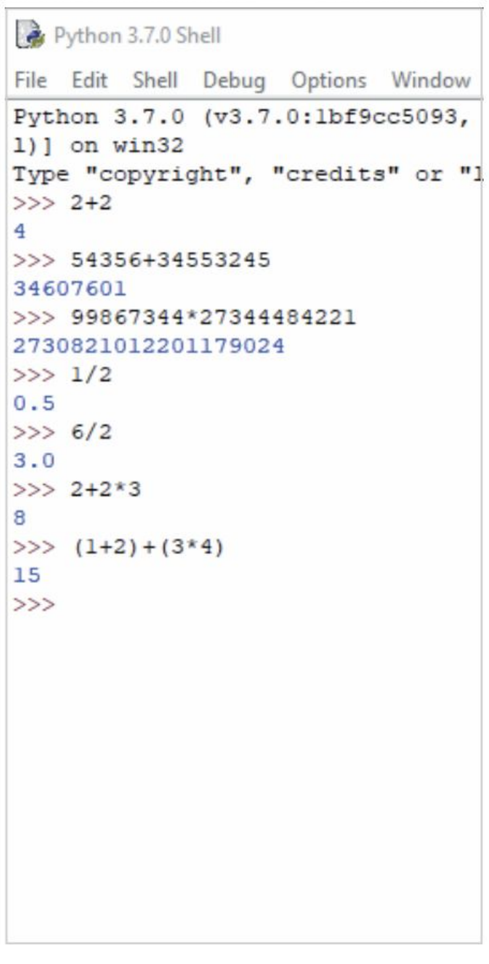
STEP 2 In the Shell enter the following:

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

As you can see, Python can handle some quite large numbers.

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

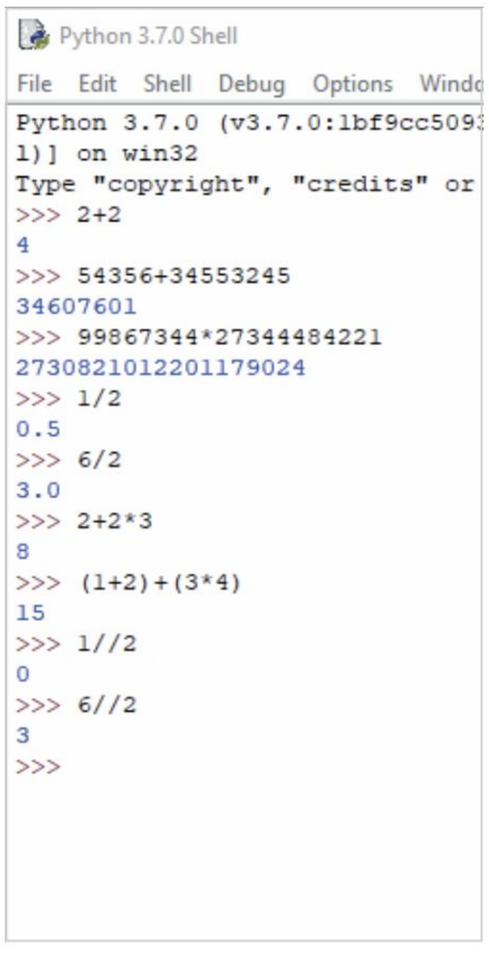
```
1/2
6/2
2+2*3
(1+2)+(3*4)
```



STEP 4 As you've no doubt noticed, division produces a decimal number. In Python, these are called floats, or floating point arithmetic. If however, 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

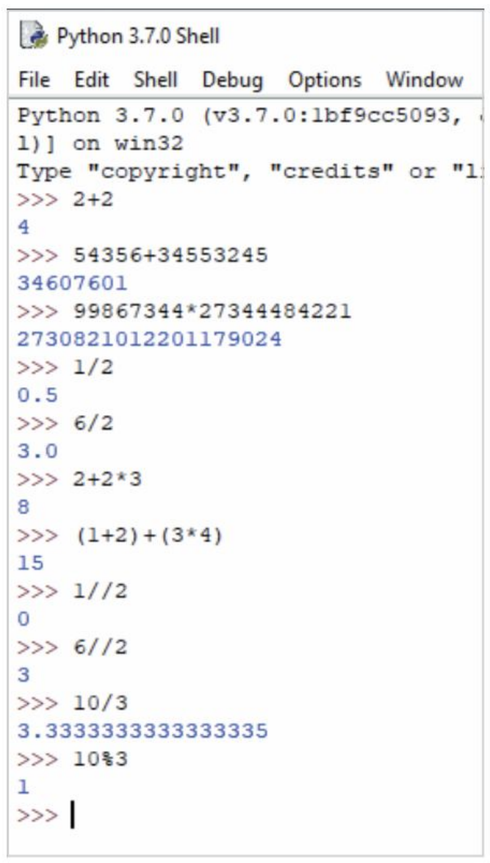
You can also use an operation to see the remainder left over from division. For example:

```
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 by 3.

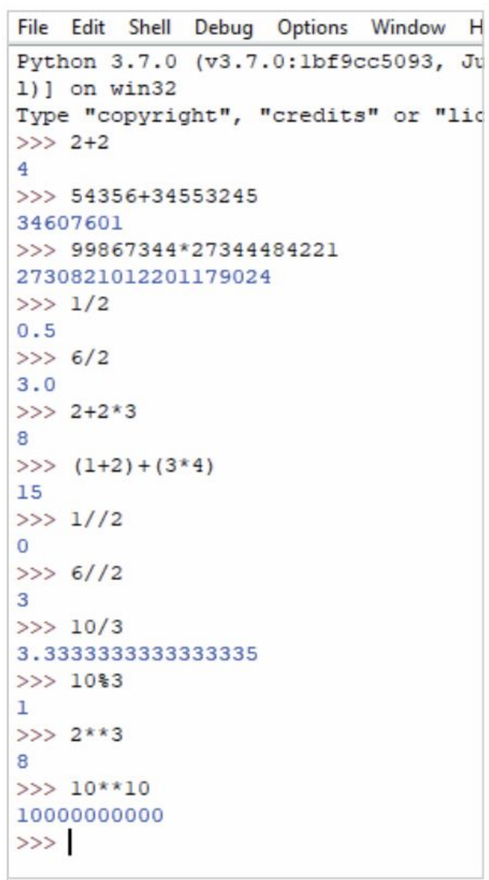


STEP 6

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
```

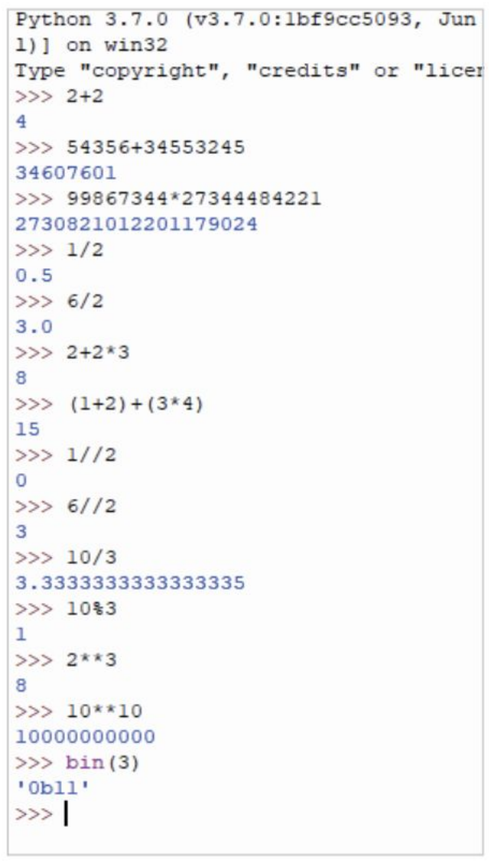
Essentially, it's 2x2x2, but we're sure you already know the basics behind maths operators. This is how you would work it out in Python.



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 Pythogarian tongue-twisters. For example, to convert a number to binary, use:

```
bin(3)
```

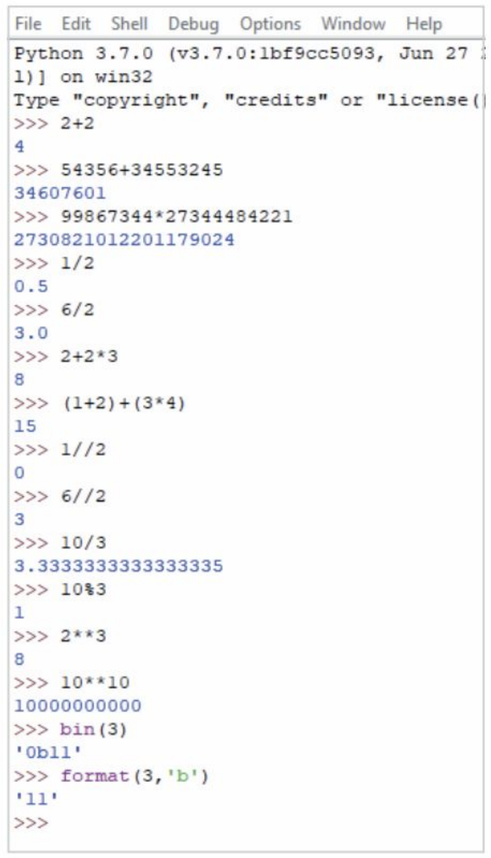


STEP 8

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')
```

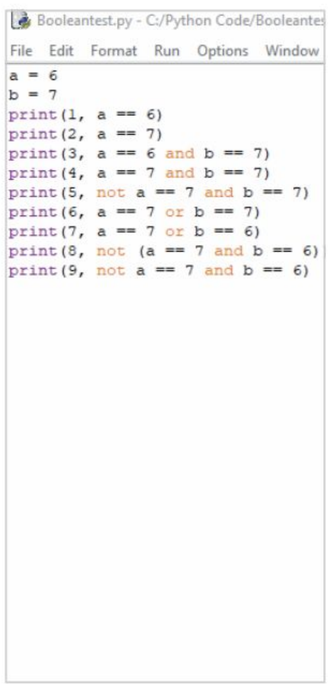
The Format command converts a value, the number 3, to a formatted representation as controlled by the format specification, the 'b' part.



STEP 9

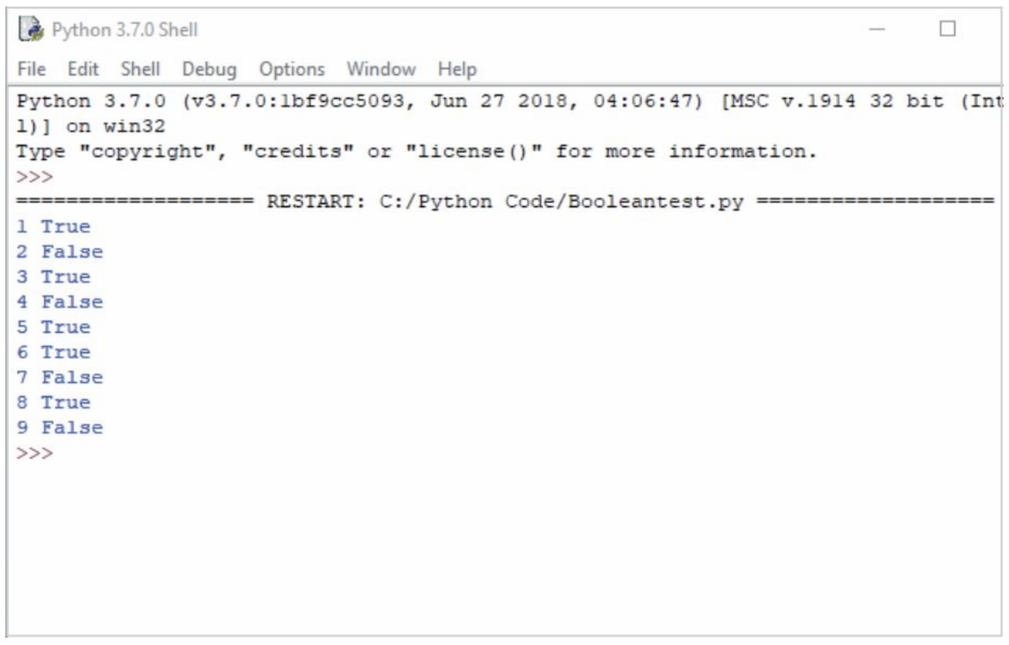
A Boolean Expression is a logical statement that will either be true or false. We can use these to compare data, and test to see if it's equal to, less than, or greater than. Try this in a New File:

```
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'll 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 we've looked at, and an important part of programming.



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 when the code starts to hit thousands of lines, things get a little difficult to read.

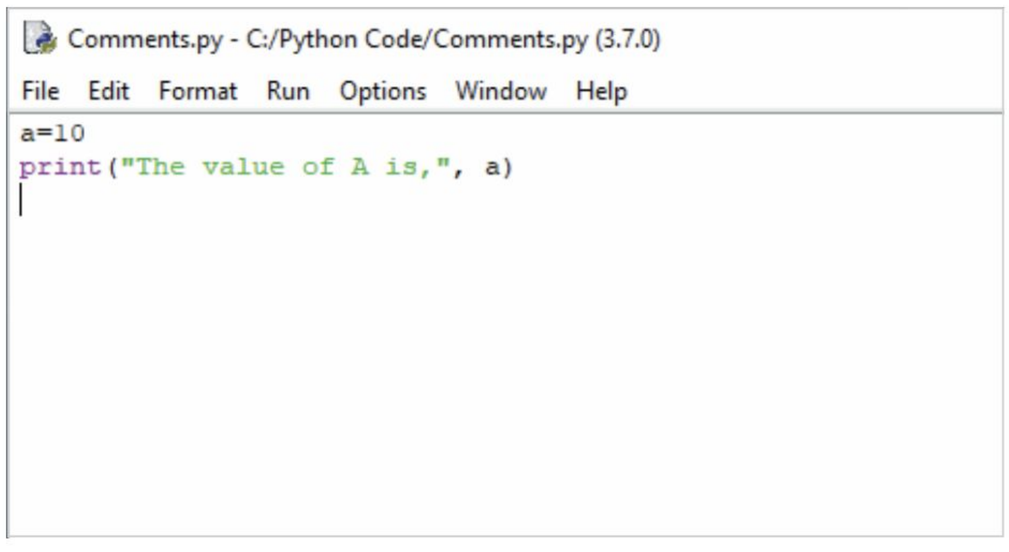
#COMMENTS!

A method used by most programmers for keeping their code readable, is by commenting on certain sections. For example, if a variable is used, the programmer comments on what it's supposed to do. It's just good practise.

STEP 1 We'll start by creating a new instance of the IDLE Editor (**File > New File**), and then create a simple variable and print command:

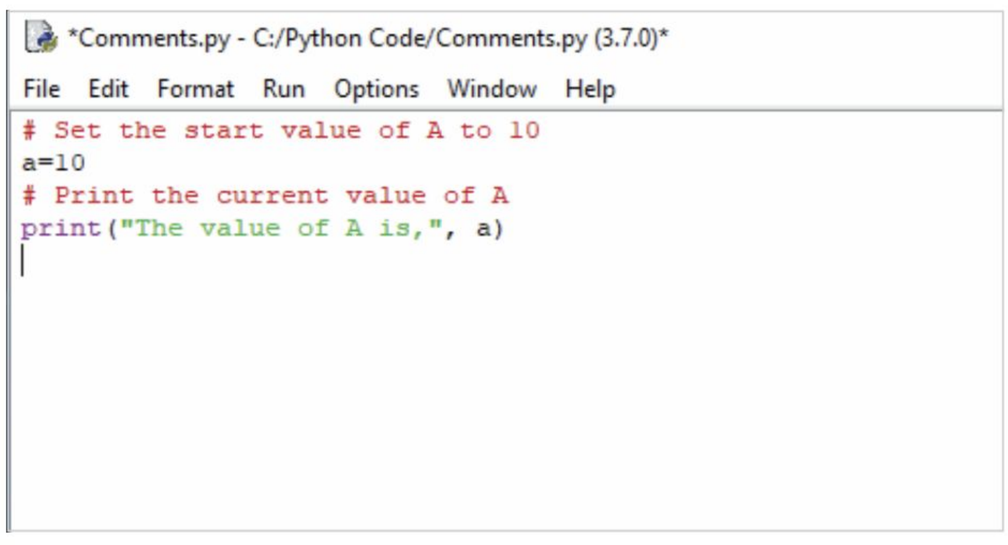
```
a=10
print("The value of A is,", a)
```

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 let's add some of the types of comments you'd normally see within code:

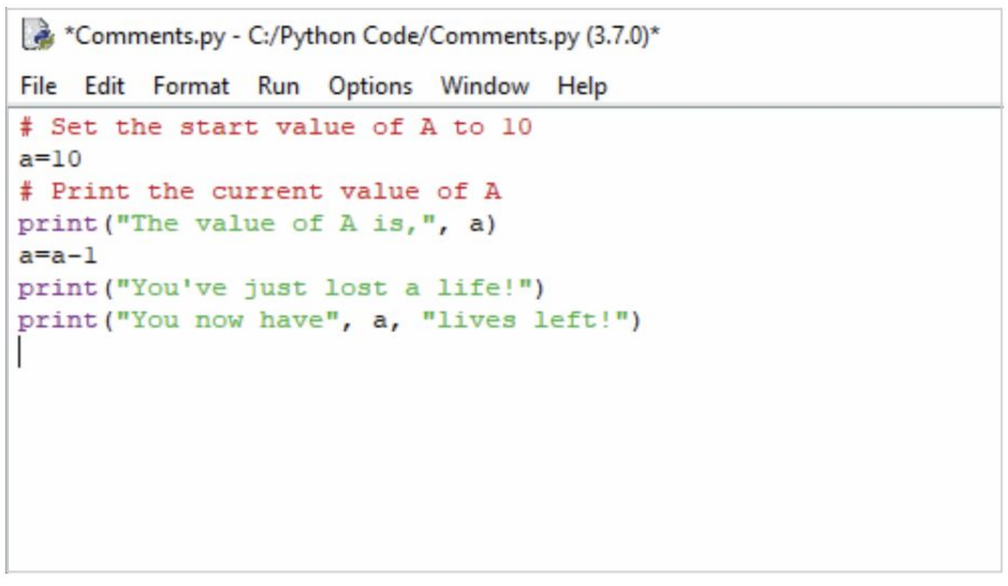
```
# Set the start value of A to 10
a=10
# Print the current value of A
print("The value of A is,", a)
```



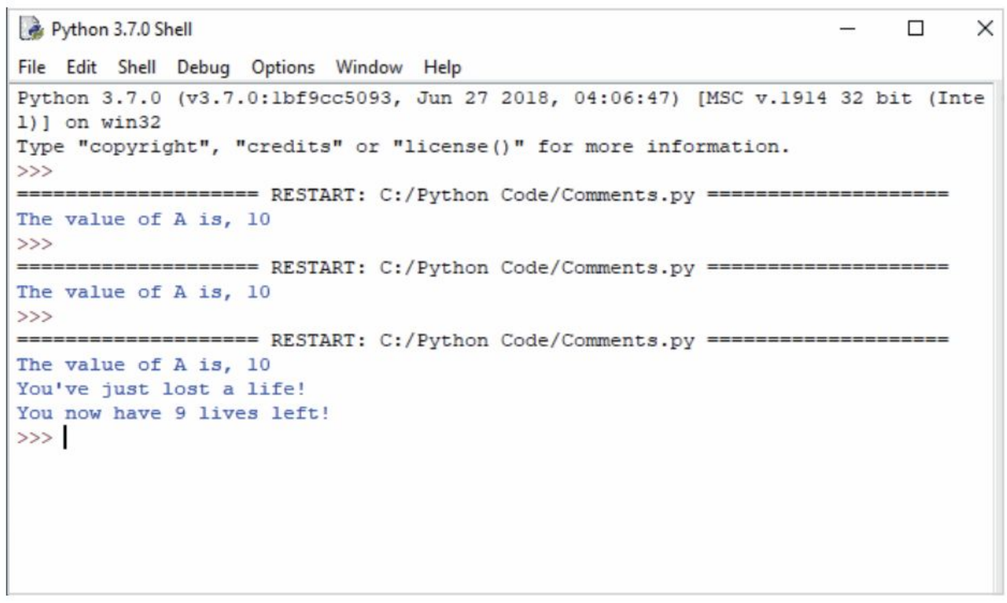
STEP 3 Re-save the code and execute it. You'll see that the output in the IDLE Shell is still the same as before, despite the extra lines being added. Simply put, the hash symbol (#) denotes a line of text the programmer can insert, to inform them and others of what's going on, without the user being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

STEP 4 Let's assume that the variable A we've created is the number of lives in a game. Every time the player dies, the value decreases 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!")
```

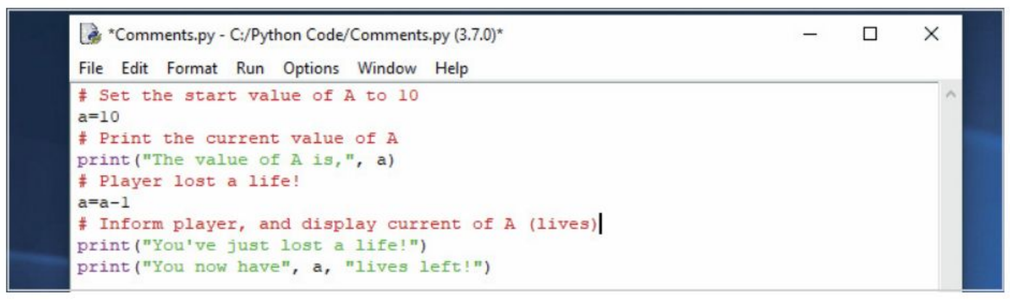


STEP 5 While we know that the variable A denotes number of lives and the player has just lost one, a casual viewer, or someone checking the code, may not know. Imagine for a moment that the code is twenty thousand lines long, instead of just our seven. You can see how handy comments are.



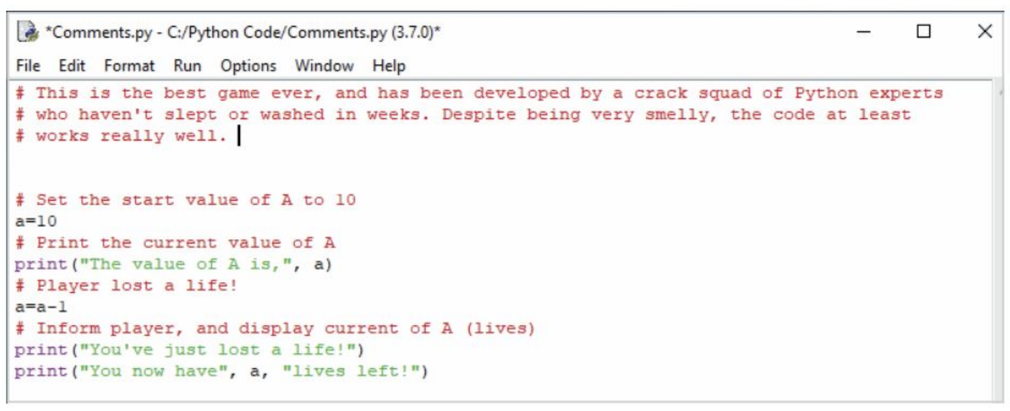
STEP 6 Essentially, the new code together with comments could look like:

```
# 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!")
```



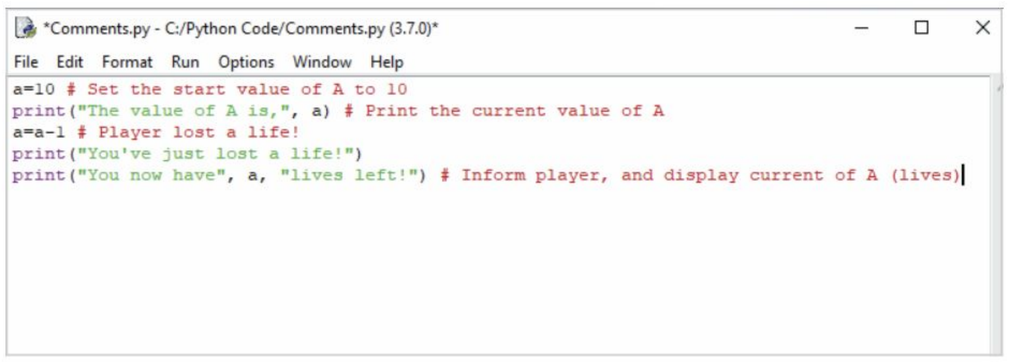
STEP 7 You can use comments in different ways. For example, Block Comments are a large section of text that details what's going on in the code, such as telling the code reader which variables you're planning on using:

```
# 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.
```



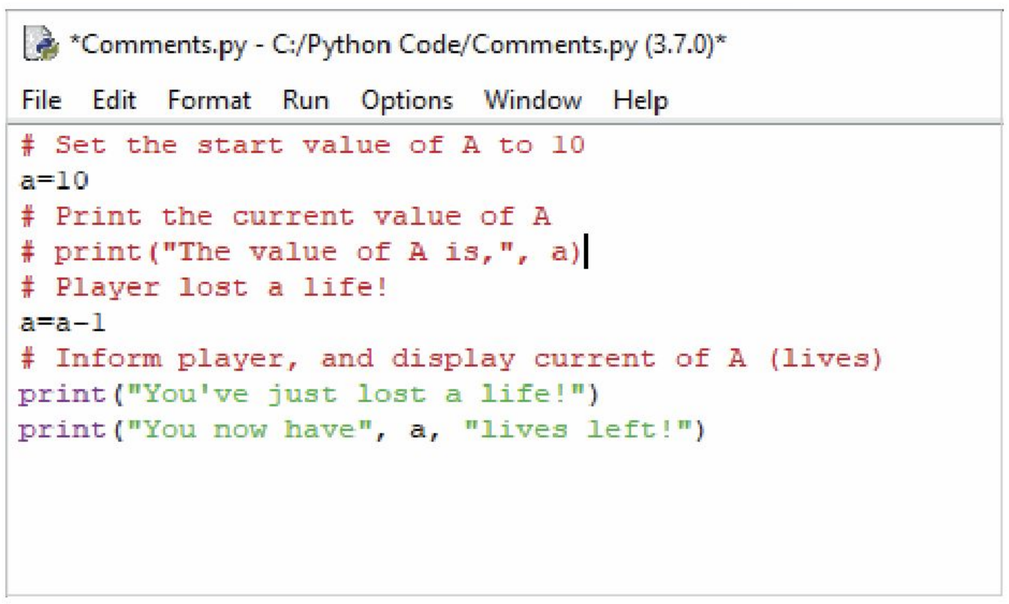
STEP 8 Inline Comments are comments that follow a section of code. Take our examples from above, instead of inserting the code on a separate line, we could use:

```
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)
```



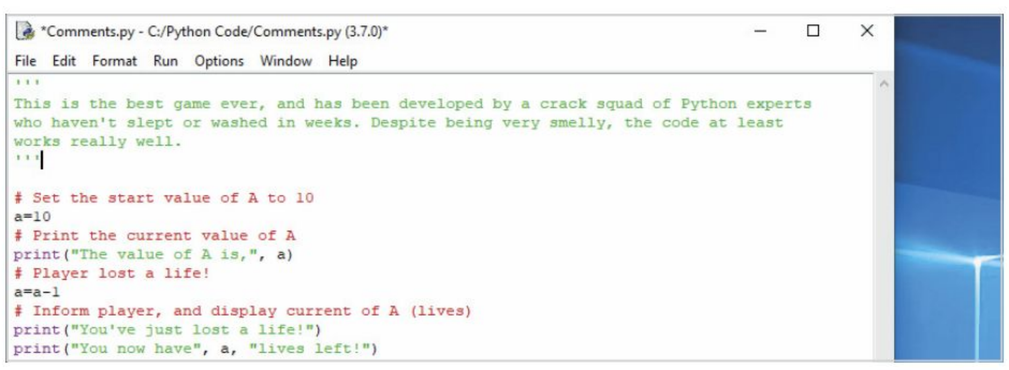
STEP 9 The comment, the hash symbol, can also be used to comment out sections of code you don't want to be executed in your program. For instance, if you wanted to remove the first print statement, you would use:

```
# 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. For them to work, place them before and after the areas you want to comment:

```
'''
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

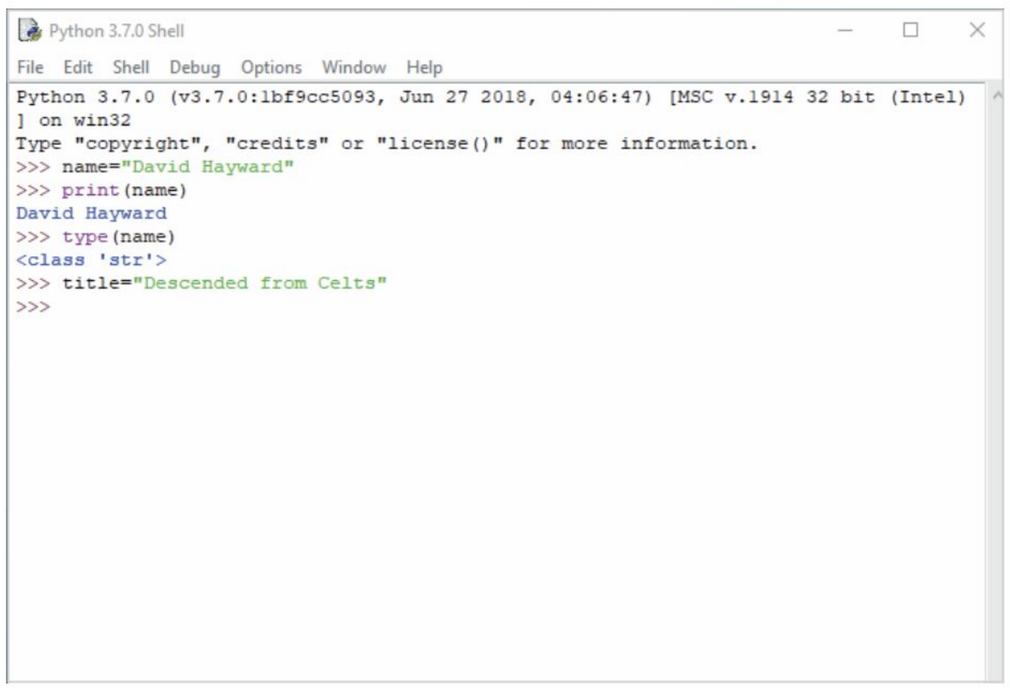
We've seen some examples of variables in our Python code already, but it's always worth going through the way they operate, and how Python creates and assigns certain values to a variable.

VARIOUS VARIABLES

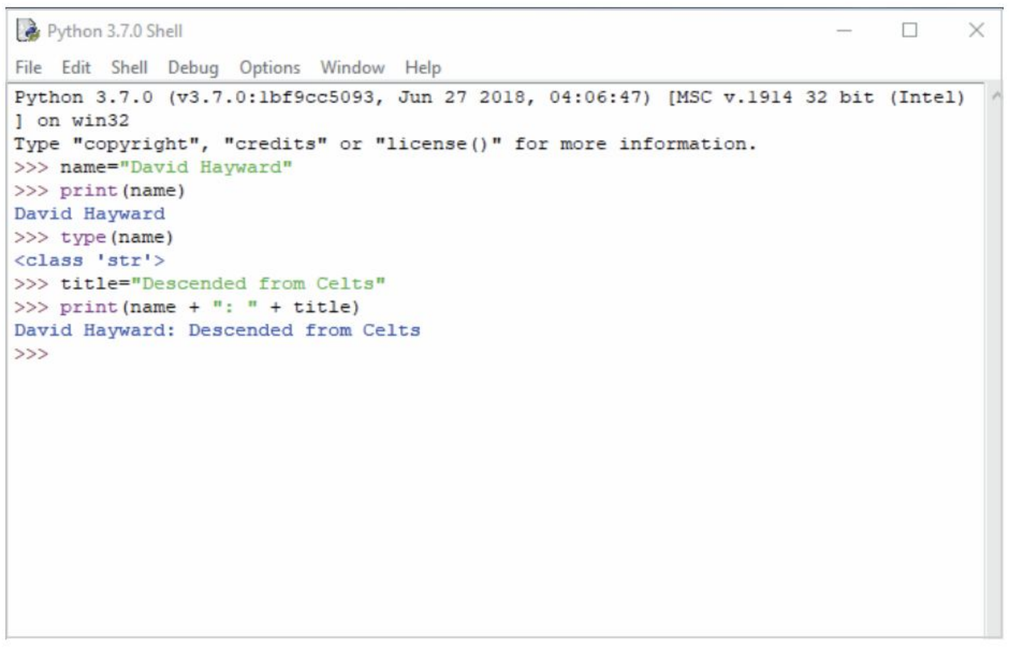
We'll be working with the Python 3 IDLE Shell in this tutorial. If you haven't already, open Python 3 or close down the previous IDLE Shell to clear up any old code.

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, so, for example, in the Shell enter: `name="David Hayward"` (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 Celts"`.



STEP 3 As we've seen previously, 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, we need to add them manually.



STEP 4 We 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, as we know, are integers.

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'll need to turn one into the other, or vice versa. When you do try to combine both, you'll get an error message:

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

```
] on win32
Type "copyright", "credits" or "license()" for more information.
>>> name="David Hayward"
>>> print(name)
David Hayward
>>> type(name)
<class 'str'>
>>> title="Descended from Celts"
>>> print(name + ": " + title)
David Hayward: Descended from Celts
>>> character=name + ": " + title
>>> print(character)
David Hayward: Descended from Celts
>>> age=46
>>> type(age)
<class 'int'>
>>> print (name+age)
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    print (name+age)
TypeError: can only concatenate str (not "int") to str
>>>
```

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

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

Alternatively, 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.

```
David Hayward: Descended from Celts
>>> age=46
>>> type(age)
<class 'int'>
>>> print (name+age)
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    print (name+age)
TypeError: can only concatenate str (not "int") to str
>>> print (character + " is " + str(age) + " years old.")
David Hayward: Descended from Celts is 46 years old.
>>> print (character, "is", age, "years old.")
David Hayward: Descended from Celts is 46 years old.
>>>
```

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

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

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

Python 3.7.0 Shell

File Edit Shell Debug Options Window Help

```
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [MSC v.1914 32 bit (Inte
l)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> age=input("How old are you? ")
How old are you? 46
>>> type(age)
<class 'str'>
>>>
```

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

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

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

Python 3.7.0 Shell

File Edit Shell Debug Options Window Help

```
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [MSC v.1914 32 bit (Inte
l)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> age=input("How old are you? ")
How old are you? 46
>>> type(age)
<class 'str'>
>>> age + 10
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    age + 10
TypeError: can only concatenate str (not "int") to str
>>> int(age) + 10
56
>>>
```

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.

Python 3.7.0 Shell

File Edit Shell Debug Options Window Help

```
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [MSC v.1914 32 bit (Inte
l)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> shirt=19.99
>>> type(shirt)
<class 'float'>
>>>
```

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.

Python 3.7.0 Shell

File Edit Shell Debug Options Window Help

```
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [MSC v.1914 32 bit (Inte
l)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> shirt=19.99
>>> type(shirt)
<class 'float'>
>>> int(shirt)
19
>>>
```

User Input

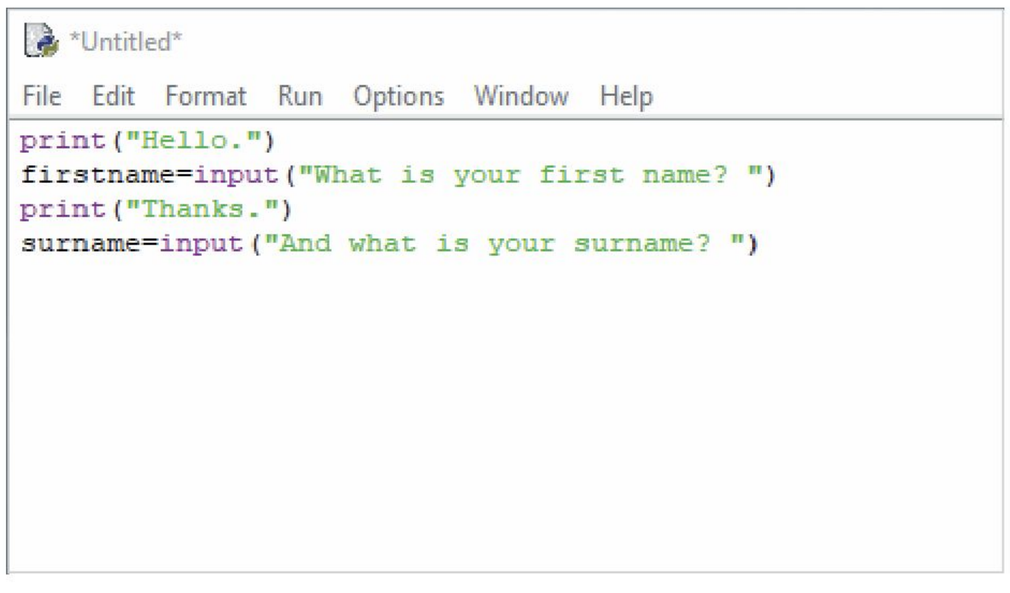
We've seen some basic user interaction with the code from a few of the examples earlier, so now would be a good time to focus solely on how you get information from the user, then store and present it.

USER FRIENDLY

The type of input you want from the user will depend greatly on the type of program you're coding. A game, for example, 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:

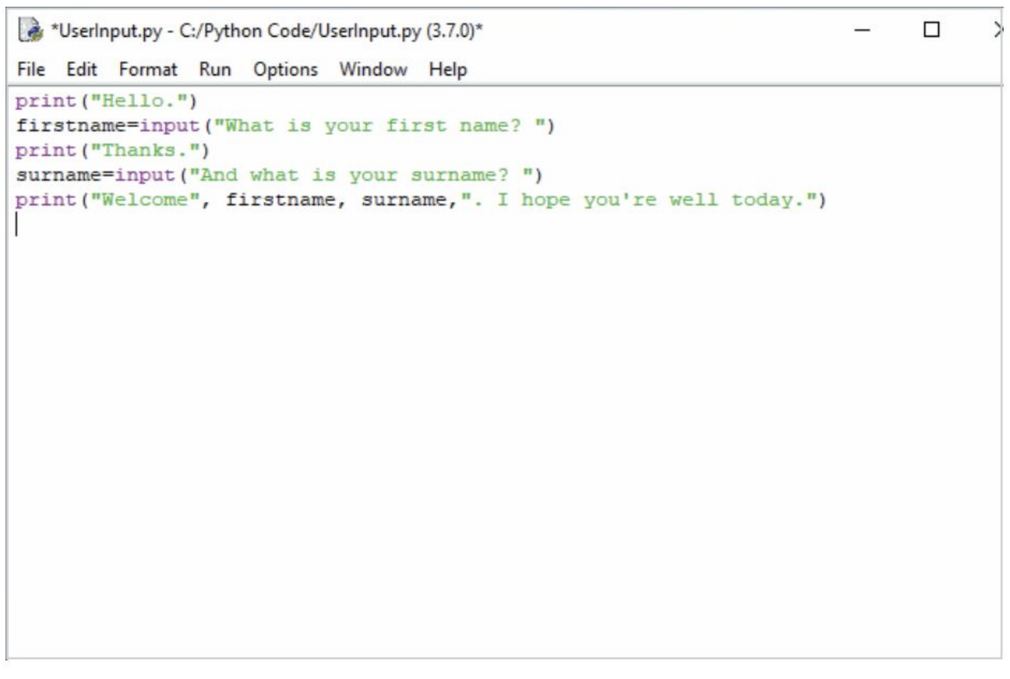
```
print("Hello")
firstname=input("What is your first name? ")
print("Thanks.")
surname=input("And what is your surname? ")
```



STEP 2 Save and execute the code, and, as you 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)**.

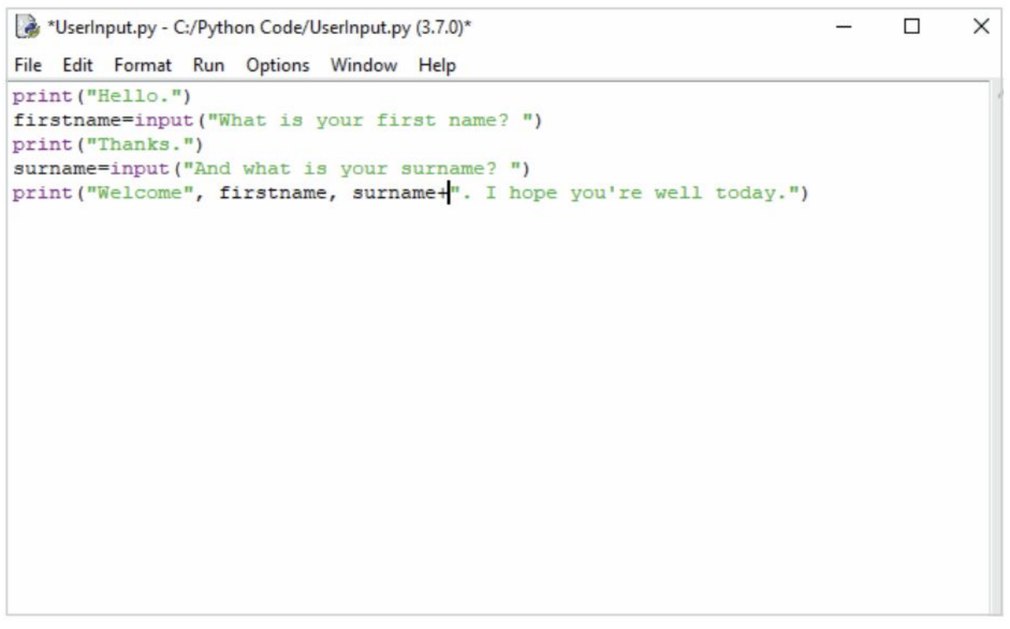
STEP 3 Now that we have the user's name stored in a couple of variables, we can call them up whenever we want:

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



STEP 4 Run the code and you'll notice 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:

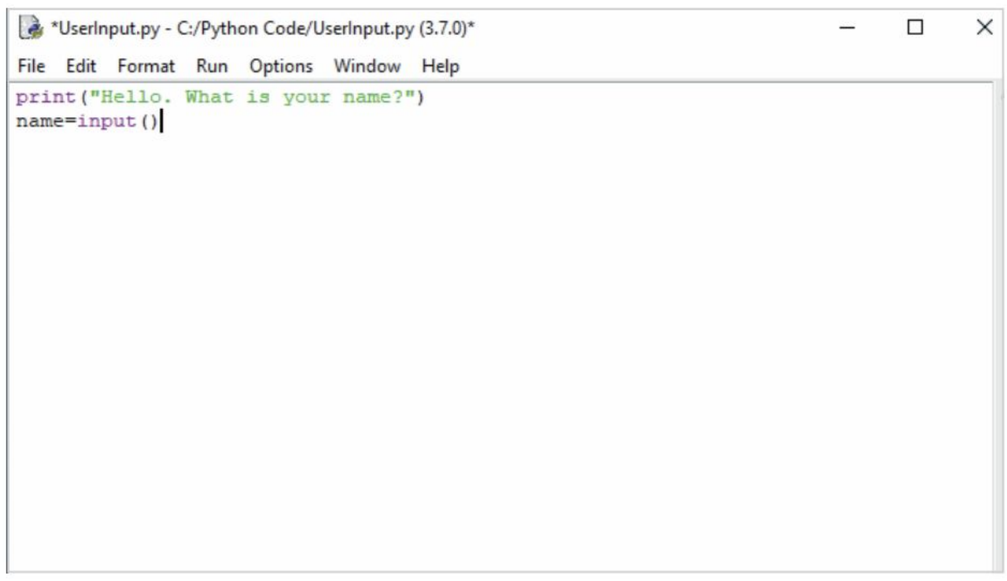
```
print("Welcome", firstname, surname+". I hope you're well today.")
```



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

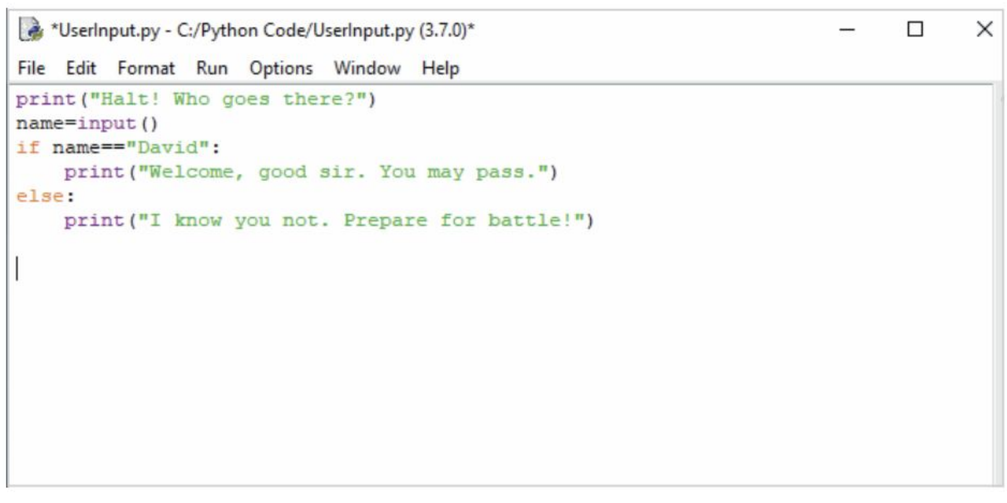
The code from the previous step is often regarded as being a little neater than having a lengthy amount of text in the input command, but it's not a rule that's set in stone, so do as you like in these situations. Expanding on the code, try this:

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

STEP 7

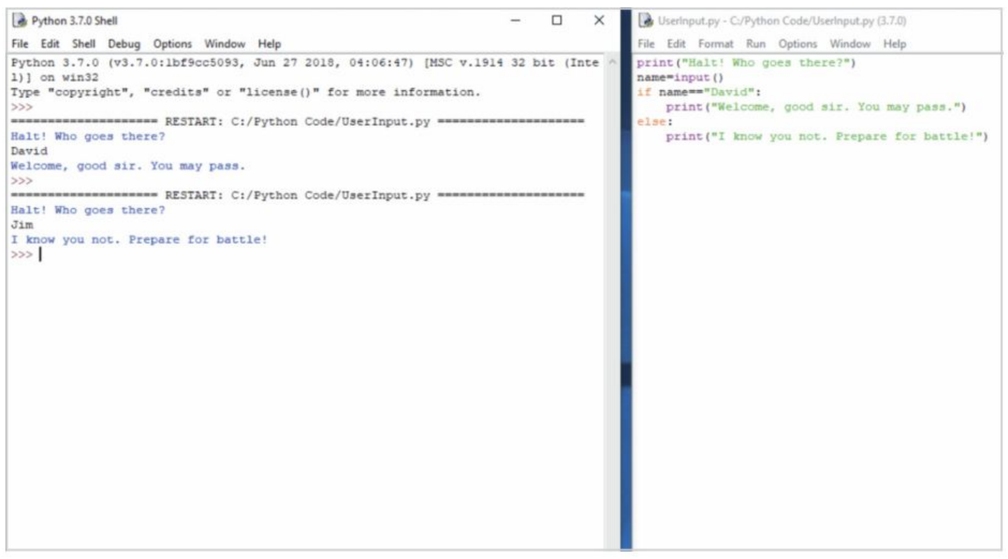
It's a good start to a text adventure game, perhaps? Now we can expand on it, and use the raw input from the user to flesh out the game a little:

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



STEP 8

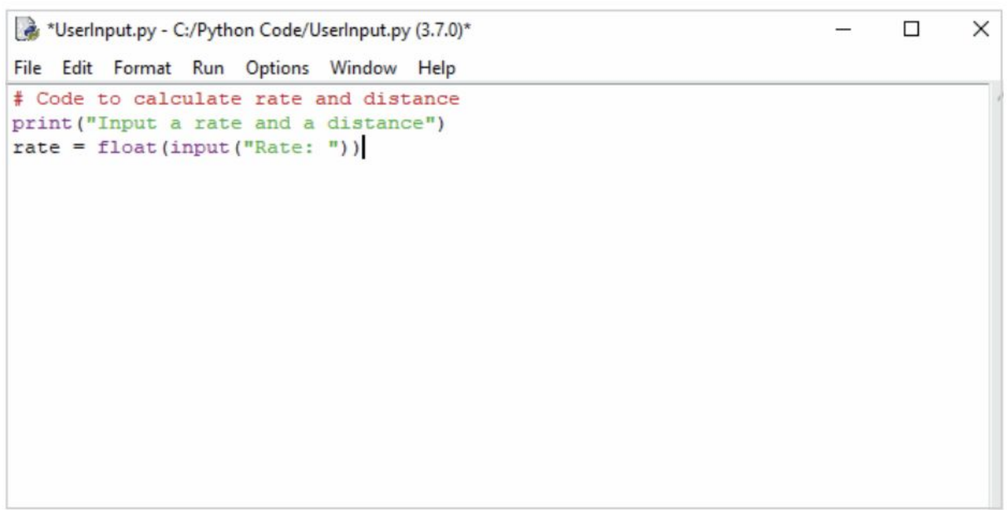
What you've created here is a condition, based on the user's input. In short, we're using the input from the user and measuring it against a condition. Therefore, if the user enters David as their name, the guard will allow them to pass unhindered. If, however, they enter a name other than David, the guard challenges them to a fight.



STEP 9

As you learned previously, any input from a user is automatically a string, so you'll 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: "))
```

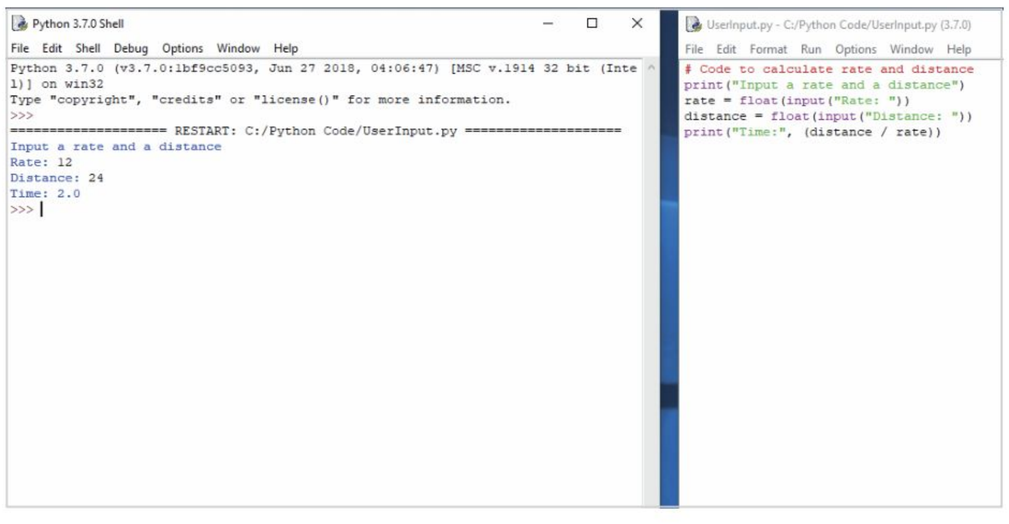


STEP 10

And 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

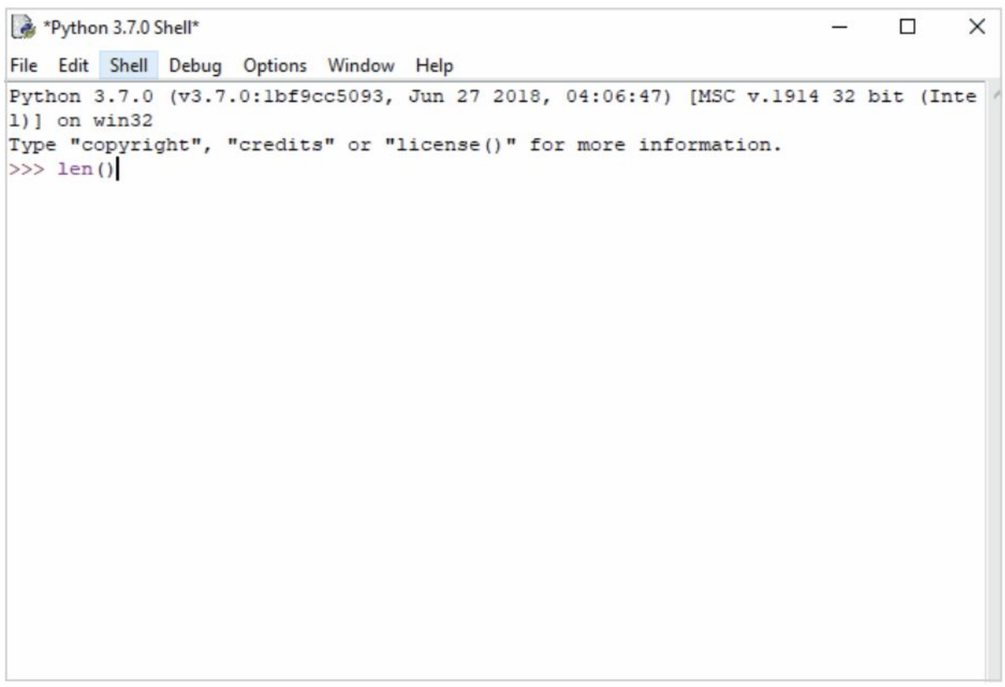
Now that you've mastered the use of variables and user input, the next step is to tackle functions. You've already used a few functions, such as the print command, but Python enables you to define your own function.

FUNKY FUNCTIONS

A function is a command that you enter into Python in order 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 only data that a function works on. Functions 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 followed by brackets. For example, **len()**, **list()**, or **type()**.



STEP 2

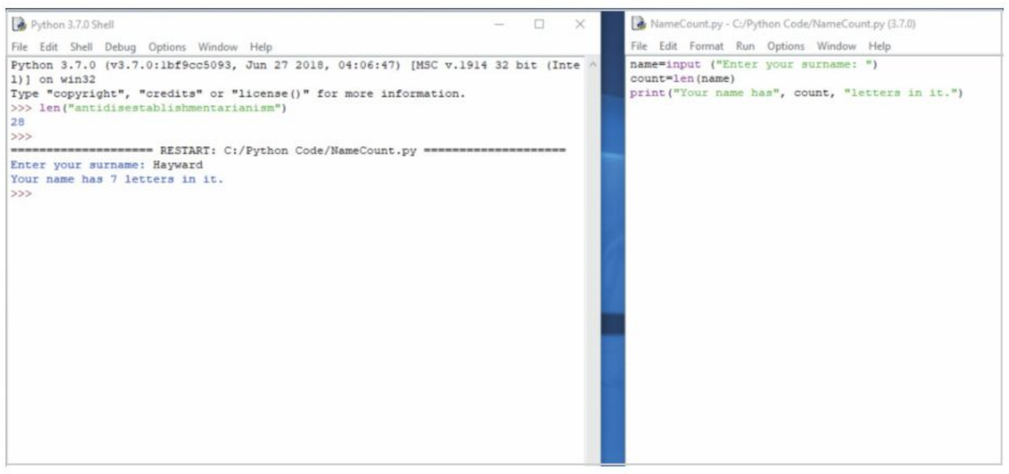
A function takes data, usually a variable, works on it depending on what the function is programmed to do, and returns the end value. The data being worked on goes inside the brackets, so if you wanted to know how many letters are in the word antidisestablishmentarianism, then you'd enter: `len("antidisestablishmentarianism")`,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

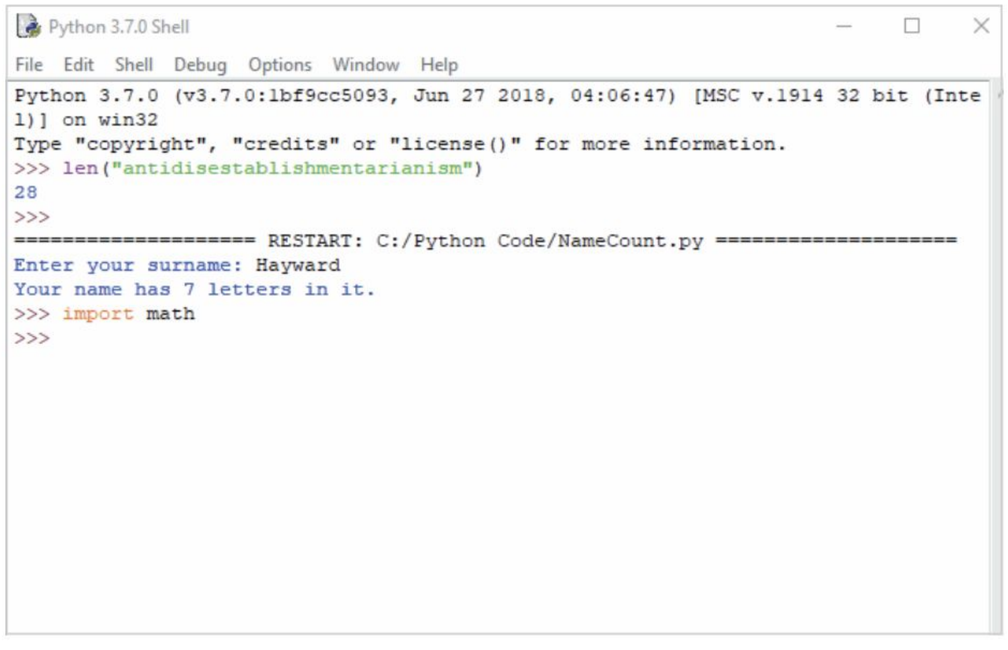
```
name=input ("Enter your surname: ")
count=len(name)
print ("Your surname has", count, "letters in it.")
```

Press **F5** and save the code to execute it.



STEP 4

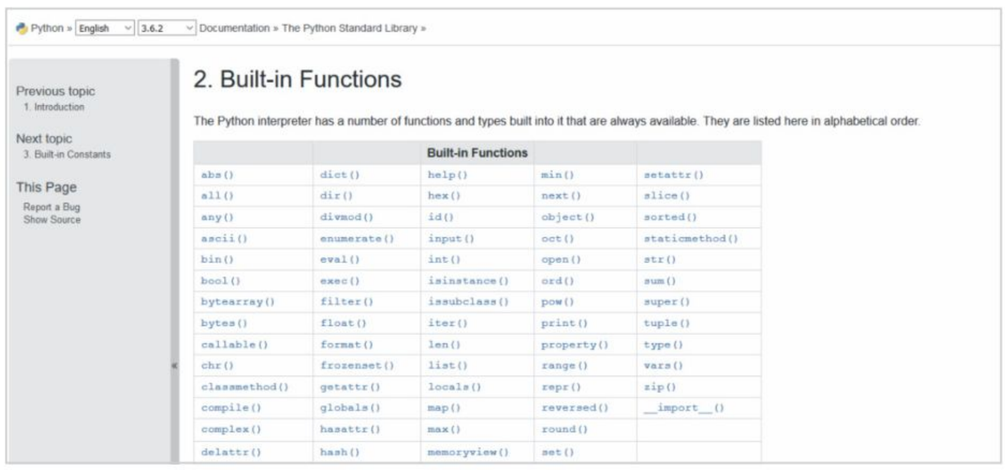
Python has tens of functions built into it, far too many to get into in the limited space available here. However, to view the list of built-in functions available to Python 3, navigate to **https://docs.python.org/3/library/functions.html**. These are the pre-defined functions, but since users have 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
```

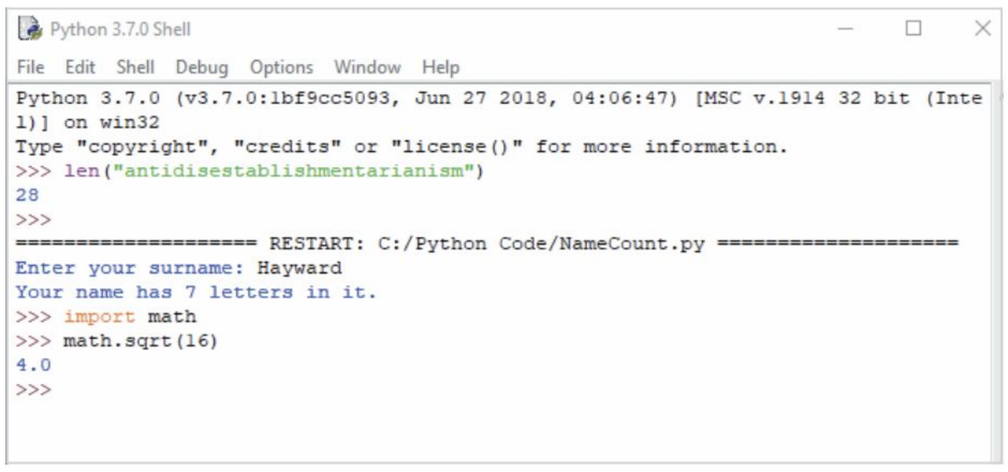
Once entered, you'll have access to all the Math module functions.



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 we've just imported it into Python, we can utilise the square root function. To do so, enter:

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

As you can see, the code is presented as **module.function(data)**.



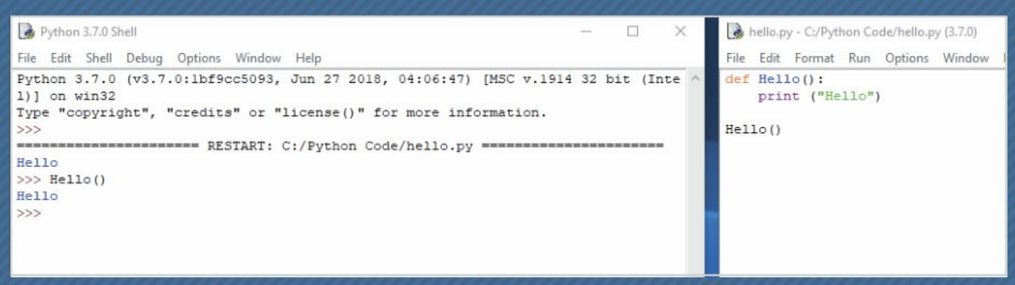
FORGING FUNCTIONS

There are many different functions, created by other Python programmers, which you can import and you'll undoubtedly come across some excellent examples in the future. However, 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 will greet a user. Enter:

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

Hello()
```

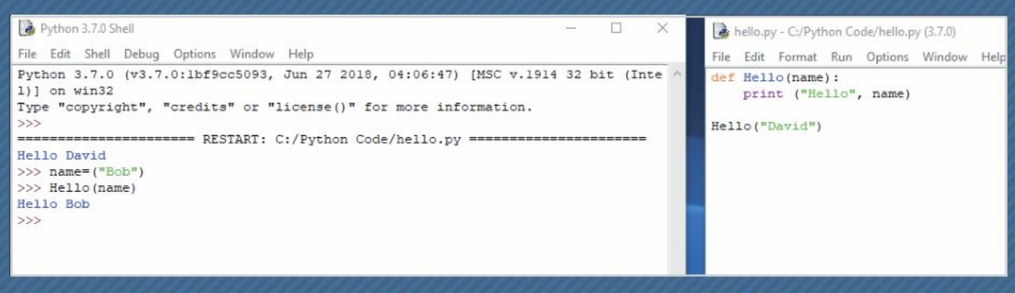
Press **F5** to save and run the script. You'll see Hello in the Shell, type in **Hello()** and it'll return the new function.



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")
```

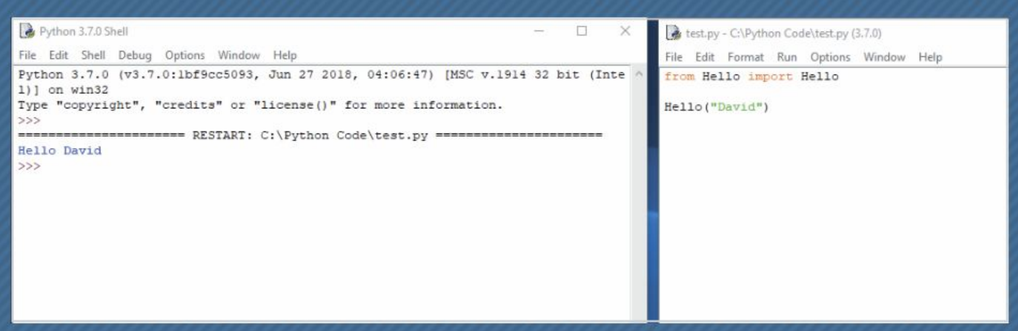
This will now accept the variable name, otherwise it will print Hello David. In the Shell, enter: **name=("Bob")**, then, **Hello(name)**. Your function can now pass variables through it.



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:

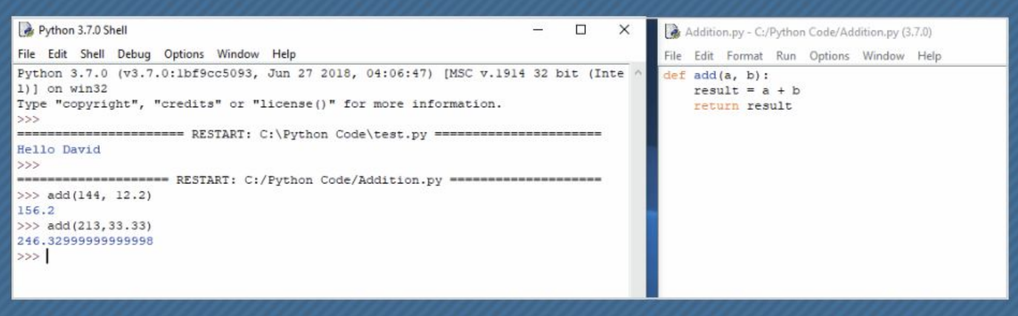
```
from Hello import Hello

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

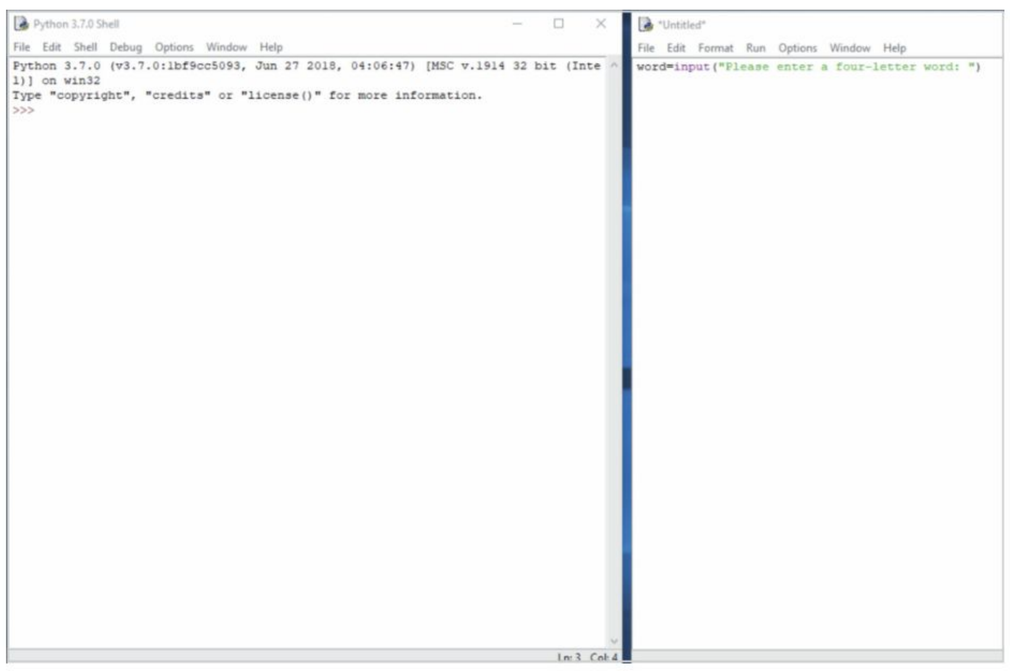
Conditions and loops are what make a program interesting, they can be simple or rather complex. How you use them depends greatly on what the program is trying to achieve, they could be the number of lives left in a game, or just displaying a countdown.

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.

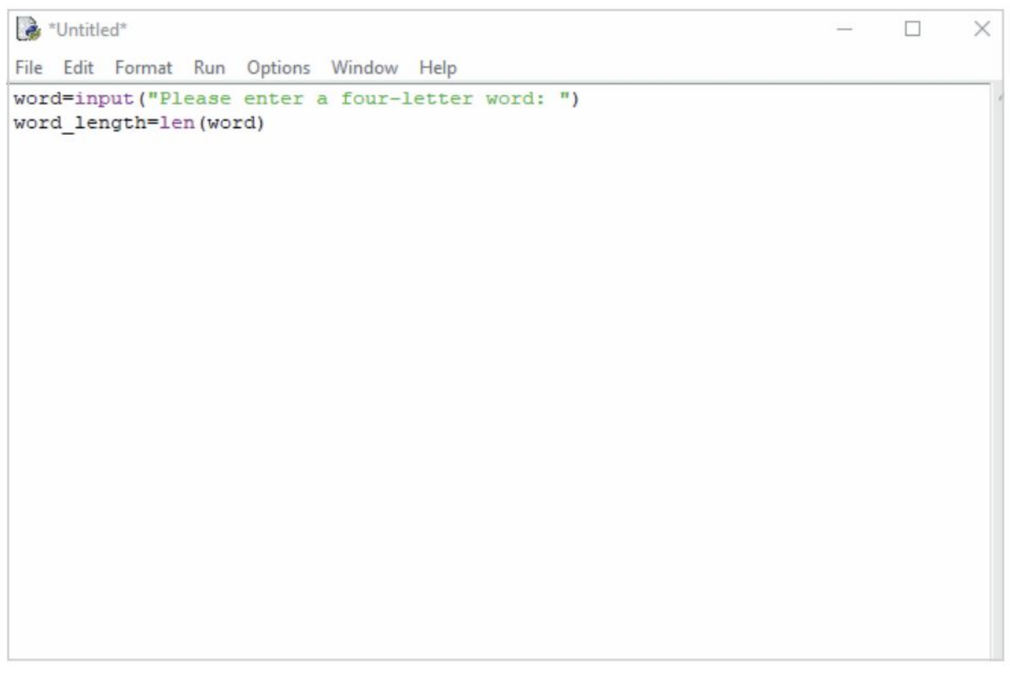
STEP 1 Let's create a new Python program that will ask the user to input a word, then check it to see if it's a four-letter word or not. Start with **File > New File**, and begin with the input variable:

```
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:

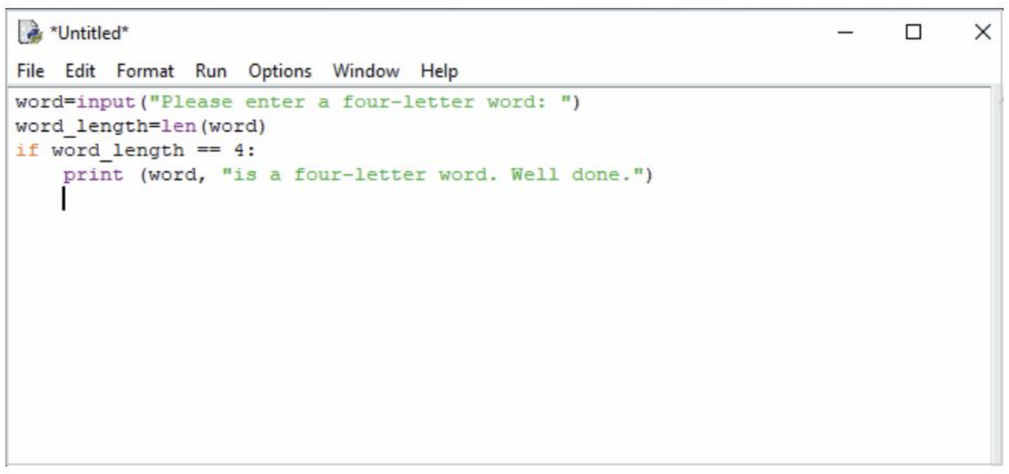
```
word=input("Please enter a four-letter word: ")
word_length=len(word)
```



STEP 3 Now we'll 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:

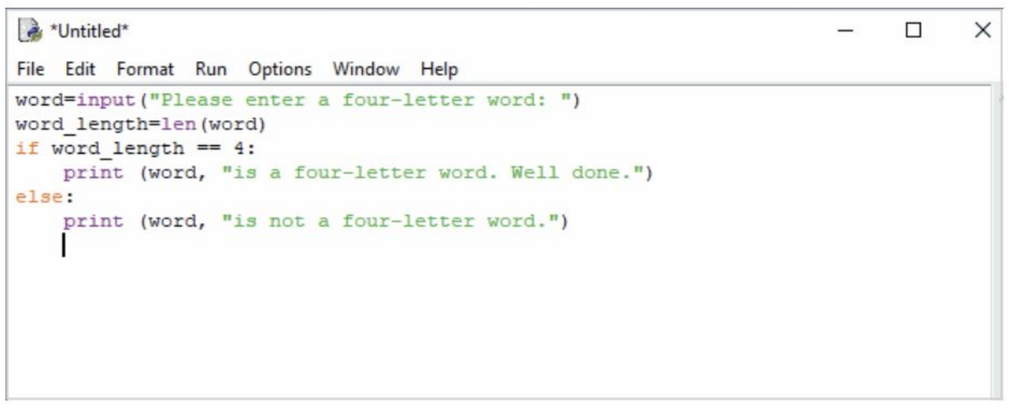
```
word=input("Please enter a four-letter word: ")
word_length=len(word)
if word_length == 4:
   print (word, "is a four-letter word. Well done.")
```

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

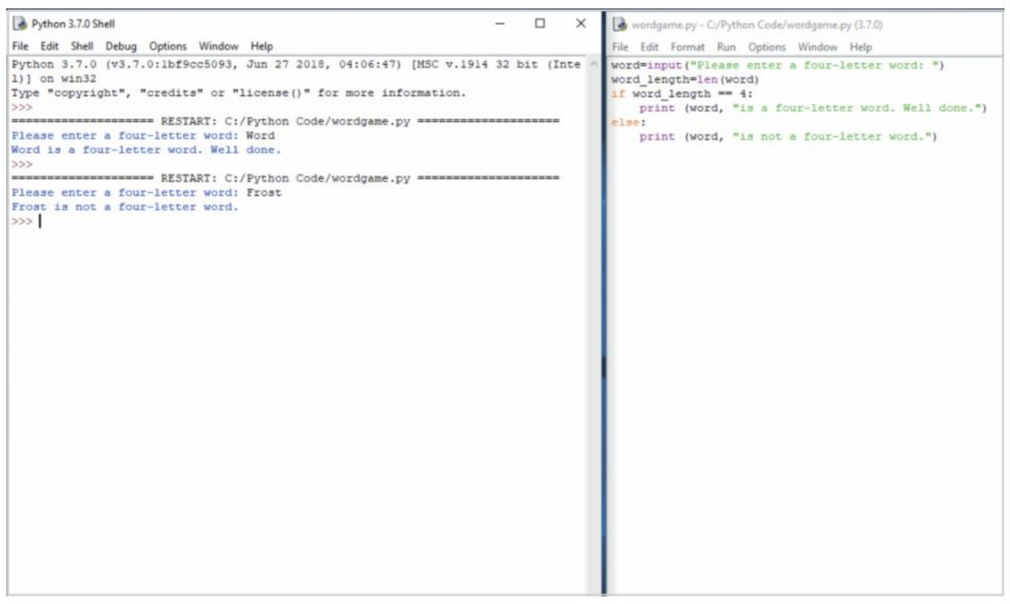


STEP 4 The colon at the end of if tells Python that if this statement is true, do everything after the colon that's indented. Next, move the cursor back to the beginning of the Editor:

```
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.")
```

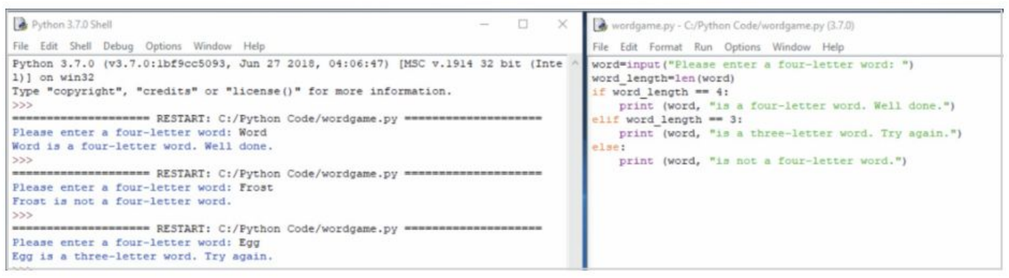


STEP 5 Press **F5** and save the code to execute it. Enter a four-letter word in the Shell to begin with, you should have the returned message that the word is four letters. Now press **F5** again, and re-run the program, but this time, enter a five-letter word. The Shell will display that it's not a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



STEP 6 Now expand the code to include other 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.")
```

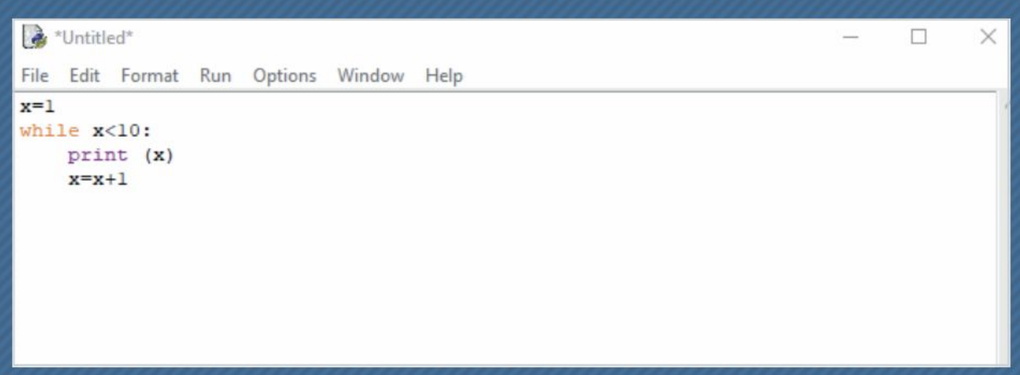


LOOPS

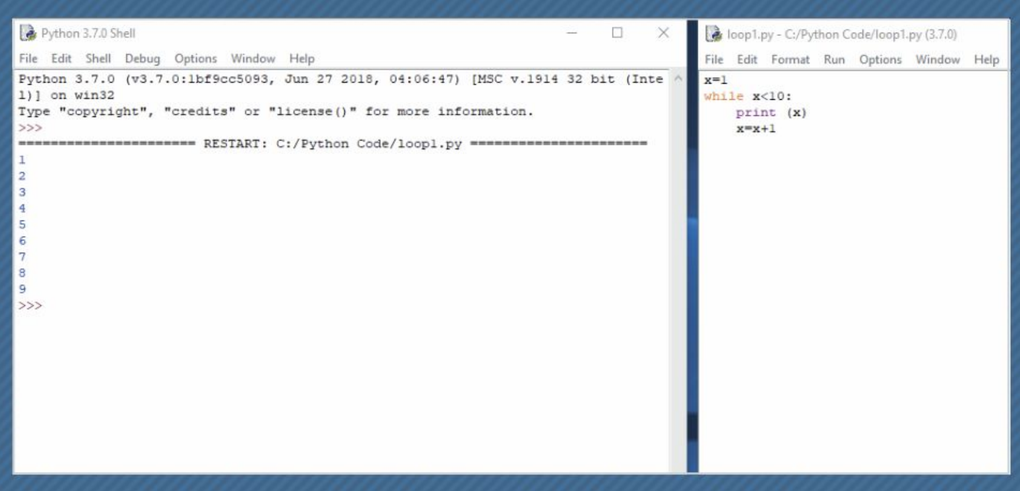
Although a loop looks quite similar to a condition, they are somewhat different in their operation. A loop will run through the same block of code a number of times, usually with the support of a condition.

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
```

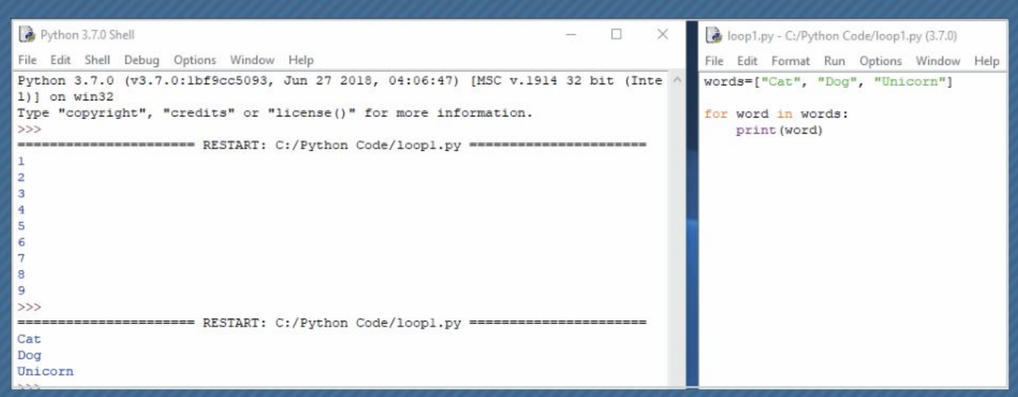


STEP 2 The difference between **if** and **while** is that when **while** gets to the end of the indented code, it goes back and checks the statement is still true. In our example x is less than 10. With each loop, it prints the current value of x, then adds one to that value. When x does eventually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



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:

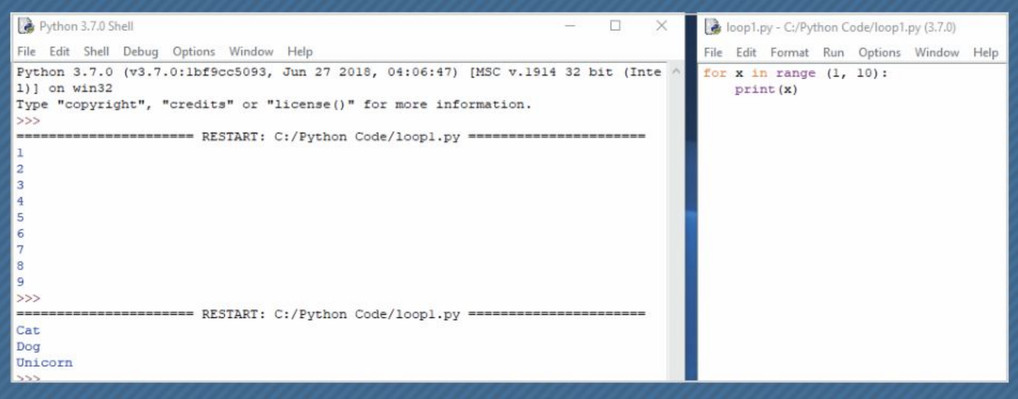
```
words=["Cat", "Dog", "Unicorn"]

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

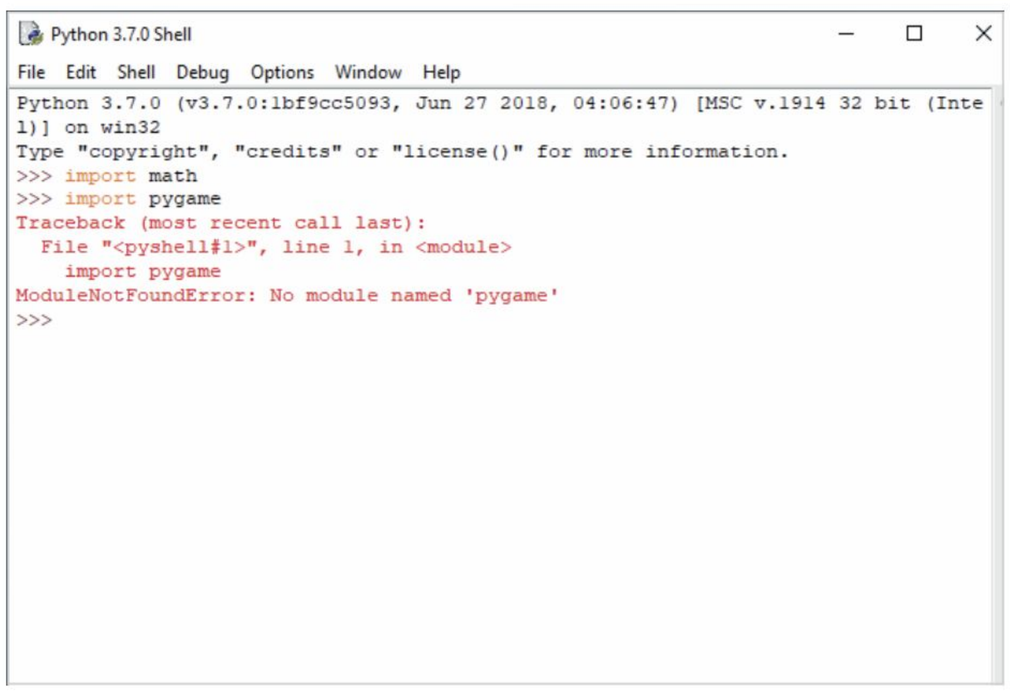
We've mentioned modules previously, using the Math module as an example, but since using modules is such a large part of getting the most from Python it's worth dedicating a little more time to them.

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.

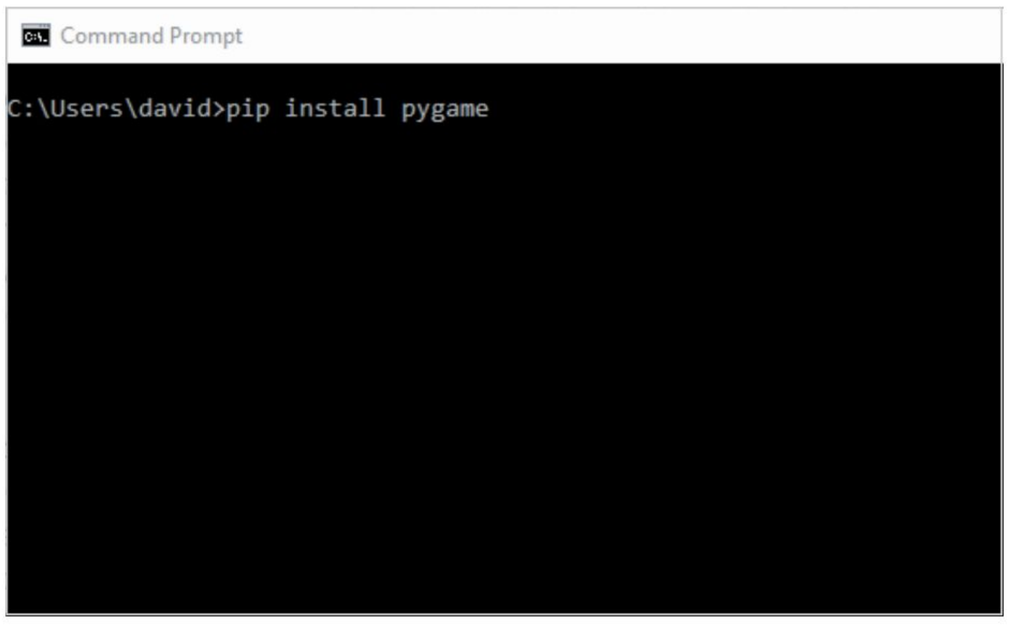
STEP 1 Although good, the built-in functions within Python are limited. The use of modules, however, allows us to make more sophisticated programs. As you are aware, modules are Python scripts that are imported, such as import math.

STEP 2 Some modules, especially on the Raspberry Pi, are included by default; the Math module is 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**.

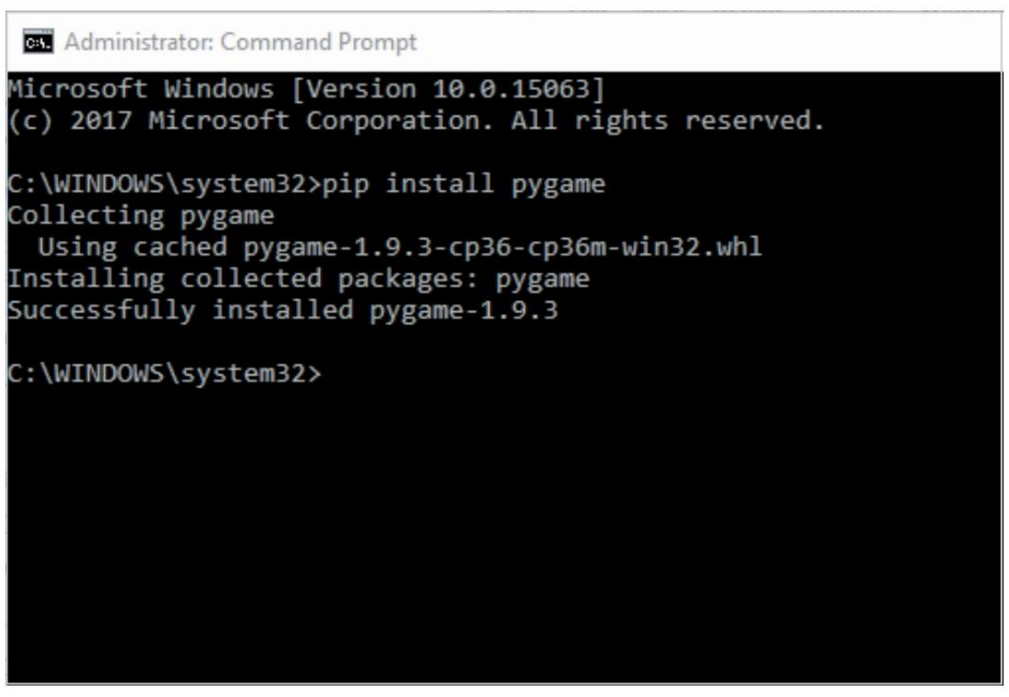


STEP 3 The result is an error in the IDLE Shell, as the Pygame module isn't recognised or installed in Python. To install a module we can use PIP (Pip Installs Packages). Close down the IDLE Shell and drop into a command prompt or Terminal session. At an elevated admin command prompt, enter:

```
pip install pygame
```



STEP 4 The PIP installation requires an elevated status due to it installing components at different locations. Start with a search for **CMD**, via the Start button, right-click the result, and 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 re-launch 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 or unique, and their absence is commonly the source of errors in execution.

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

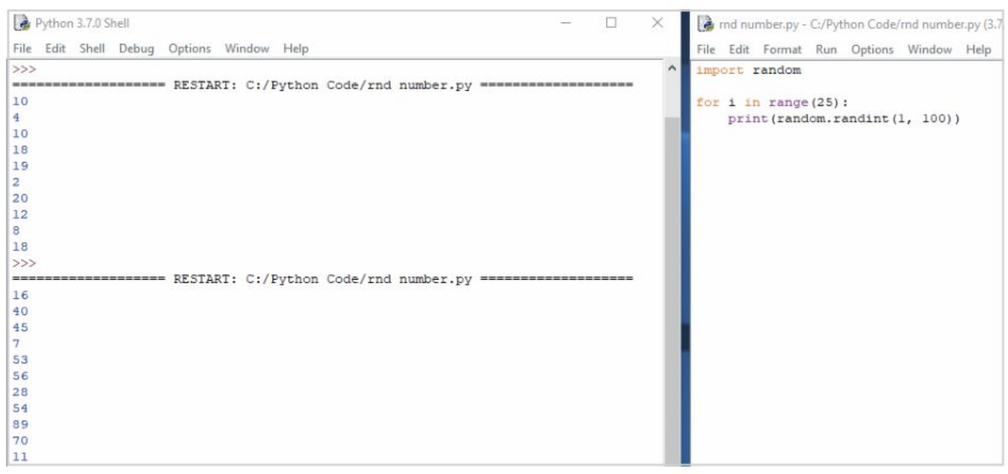
```
import random
```

Brings in the code from the Random number generator module. We can then use this module to create something like:

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

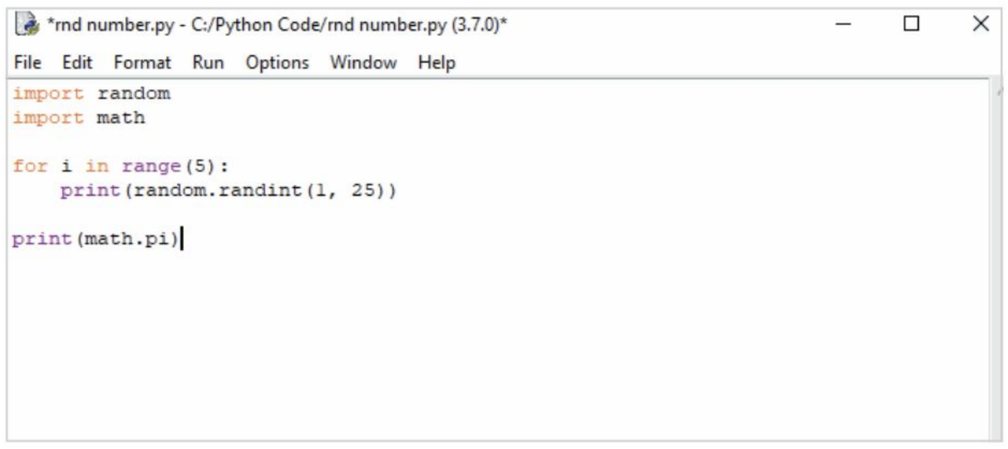
STEP 7 This code, when saved and executed, will display ten random numbers from 1 to 25. You can play around with the code to display more or less, and from a greater or lesser range. For example:

```
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.

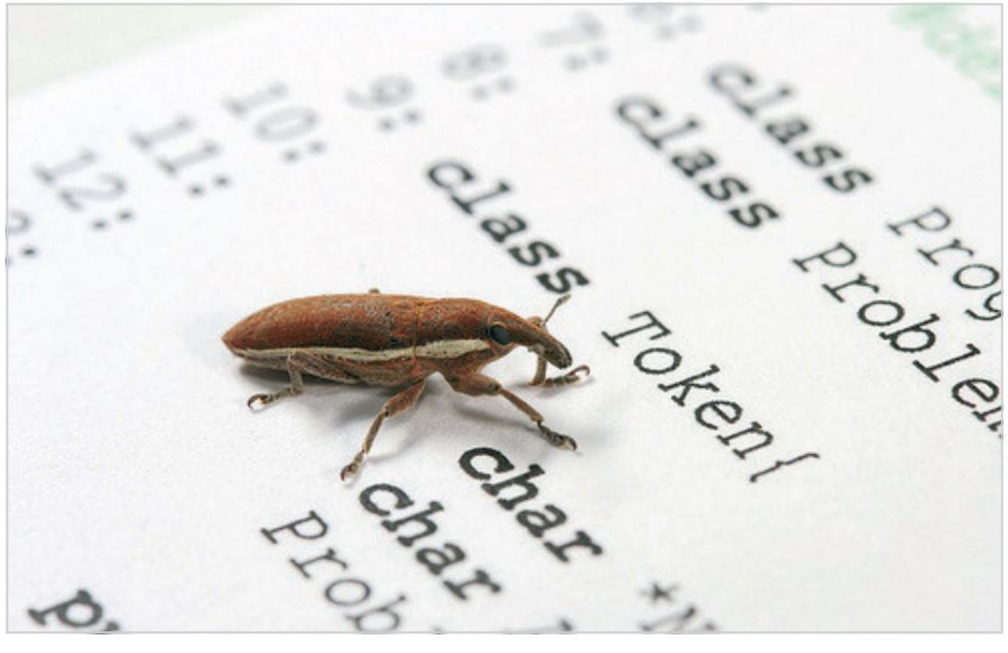
Python Errors

It goes without saying that you'll eventually come across an error in your code, where Python will declare it's not able to continue due to something being missed out, wrong, or simply unknown. Being able to identify these errors makes for a good programmer.

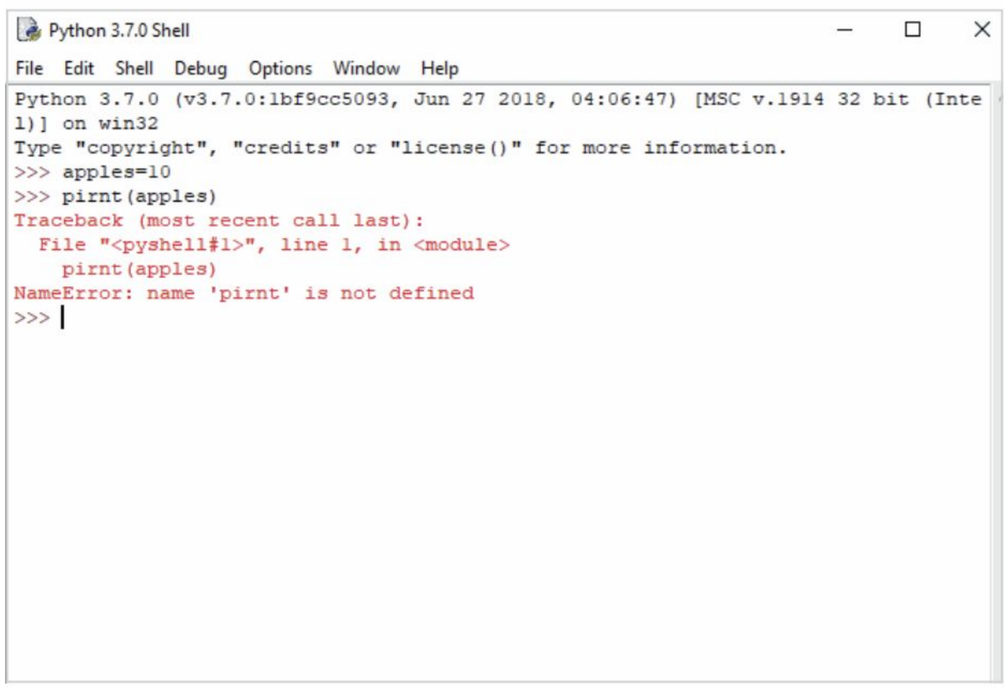
DEBUGGING

Errors in code are called bugs, they're perfectly normal and can often be easily rectified with a little patience. The import thing is to keep looking, experimenting, and testing. Eventually your code will be bug free.

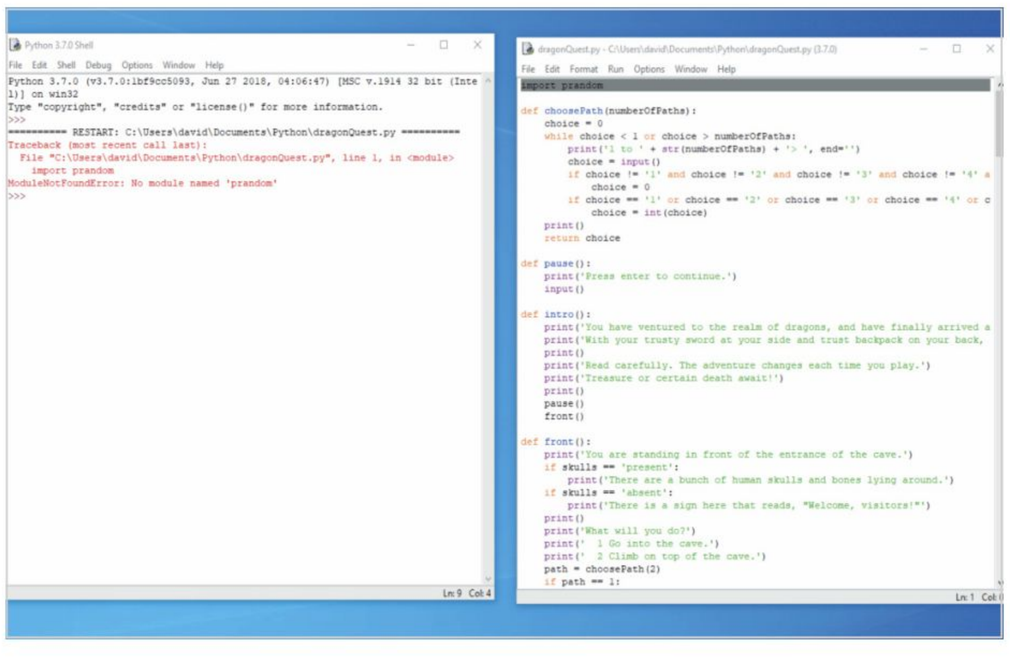
STEP 1 Code isn't as fluid as the written word, no matter how good the programming language is. Python is certainly easier than most languages, but even it is prone to some annoying bugs. The most common are typos by the user, and while easy to find in simple dozen-line code, imagine having to debug multi-thousand line code.



STEP 2 As we've mentioned, the most common of errors is the typo, often at the command level, mistyping the print command for example. However, they also occur when you've got numerous variables, all of which have lengthy names. The best advice is to simply go through the code and check your spelling.



STEP 3 Thankfully Python is helpful when it comes to displaying error messages. When you receive an error in red ink from the IDLE Shell, it will define the error itself, along with the line number where the error has occurred. While in the IDLE Editor this is a little daunting for lots of code, text editors help by including line numbering.



STEP 4 Syntax errors are probably the second most common errors you'll come across as a programmer. Even if the spelling is correct, the actual command itself is wrong. In Python 3 this often occurs when Python 2 syntaxes are applied. The most annoying of these is the print function. In Python 3, we use **print("words")**, whereas Python2 uses **print "words"**.

STEP 5

Pesky brackets are also a culprit of programming errors, especially when you have something like:

```
print(balanced_check(input()))
```

Remember that for every '(' there must be an equal number of ')'.

```
1  import sys
2
3  def balanced_check(data):
4      stack = []
5      characters = list(data)
6
7      for character in characters:
8          reference = {
9              '(': ')',
10             '{': '}',
11             '[': ']'
12         }
13         if character in reference.keys():
14             stack.append(character)
15
16     elif character in reference.values() and len(stack) > 0:
17         char = stack.pop()
18         if reference.get(char) != character:
19             return "NO"
20     else:
```

STEP 6

There are thousands of online Python resources, code snippets, and lengthy discussions across forums on how best to achieve something. While 99% of it is good code, don't always be lured into copying and pasting random code into your editor. More often than not, it won't work and frustratingly, you won't have learned anything.

8

You have a bare except clause; i.e.,

```
try:
    some_code()
except:
    clean_up()
```

The problem with a bare except is that it will catch *all* exceptions, including ones you really don't want to be ignoring (like KeyboardInterrupt and SystemExit). It would be much better if your except block only caught the specific exception you expect, and let all others bubble up as normal.

A few other general comments on your code:

- In line 200, you have this construction:

```
for letter in range(len(chosen_word)):
    if player_guess == chosen_word[letter]:
        word_guessed[letter] = player_guess
```

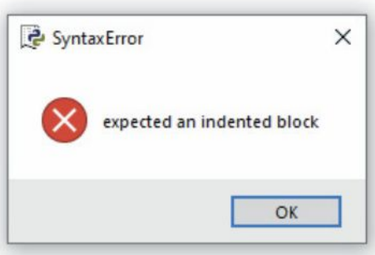
You're looping over the index variable, but also using the list element. It would be better to write:

```
for idx, letter in enumerate(chosen_word):
    if player_guess == letter:
```

STEP 7

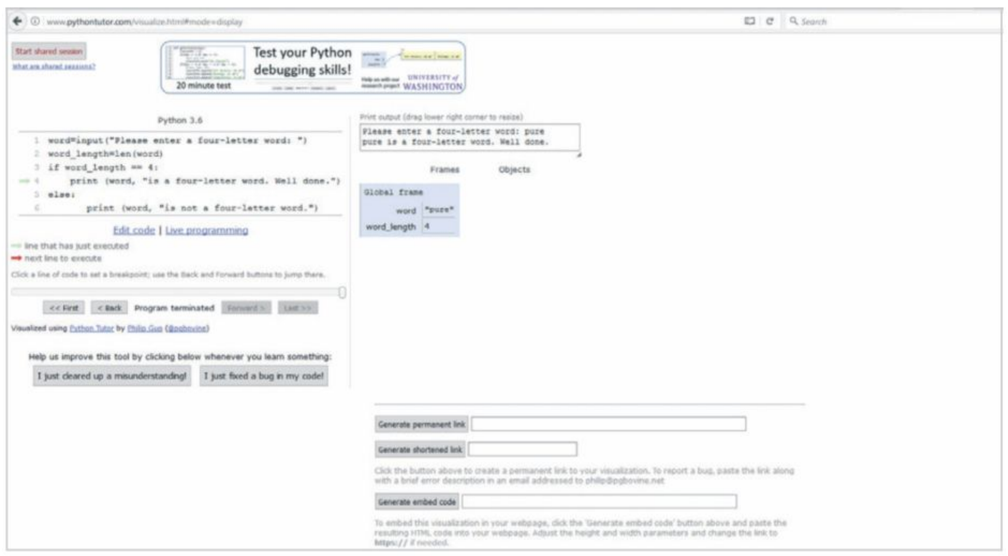
Indents are a tricky part of Python programming that a lot of beginners fall foul of. Recall the If loop from the Conditions and Loops section, where the colon means everything indented following the statement is to be executed as long as it's true? Missing the indent, or having too much of an indent, will come back with an error.

```
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.")
```



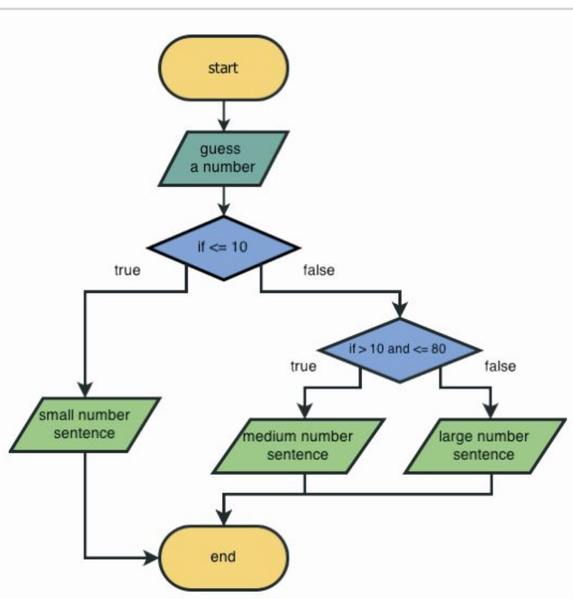
STEP 8

An excellent way to check your code step-by-step is to use Python Tutor's Visualise web page, found at **www.pythontutor.com/visualize.html#mode=edit**. Simply paste your code into the editor and click the Visualise Execution button to run the code line-by-line. This help to clear bugs, and any misunderstandings.



STEP 9

Planning makes for good code. While a little old school, it's a good habit to plan what your code will do before sitting down to type it out. List the variables that will be used, along with the modules, and write out a script for any user interaction or outputs.



STEP 10

Purely out of interest, the word debugging in computing terms comes from Admiral Grace Hopper, who, back in the '40s, was working on a monolithic Harvard Mark II electromechanical computer. According to legend, Hopper found a moth stuck in a relay, thus preventing the system from working. Removal of the moth was hence called: debugging.

Combining What You Know So Far

As we've reached the end of this section, let's take a moment to combine all we've looked at so far and apply it to writing a piece of code. This code can then be used to insert in your own programs in future, either in part or or as a whole.

PLAYING WITH PI

For this example, we're going to create a program that will calculate the value of Pi to a set number of decimal places; as described by the user. It combines much of what we've learned, and a little more.

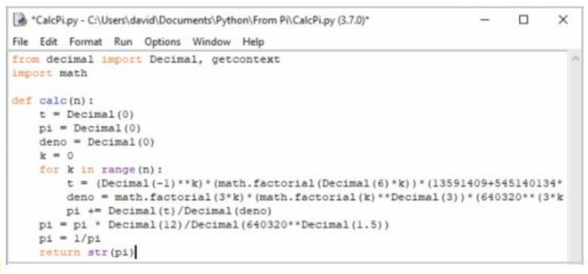
STEP 1 Start by opening Python and creating a New File in the Editor. First, we need to get hold of an equation that can accurately calculate Pi without rendering the computer's CPU useless for several minutes. The recommended calculation used in such circumstances is the Chudnovsky Algorithm, you can find more information about it at **en.wikipedia.org/wiki/Chudnovsky_algorithm**.

STEP 2 We can utilise the Chudnovsky Algorithm to create our own Python script based on the calculation. We'll begin by importing some important modules and functions within the modules:

```
from decimal import Decimal, getcontext
import math
```

This uses the decimal and getcontext functions from the decimal module, both of which deal with large decimal place numbers, and, naturally, the math module.

STEP 3 Now we can insert the Pi calculation algorithm part of the code. This is a version of the Chudnovsky Algorithm:




```
def calc(n):
    t = Decimal(0)
    pi = Decimal(0)
    deno = Decimal(0)
    k = 0
    for k in range(n):
        t = (Decimal(-1)**k)*(math.
factorial(Decimal(6)*k))*(13591409+545140134*k)
        deno = math.factorial(3*k)*(math.
factorial(k)**Decimal(3))*(640320**(3*k))
        pi += Decimal(t)/Decimal(deno)
    pi = pi * Decimal(12)/
Decimal(640320**Decimal(1.5))
    pi = 1/pi
    return str(pi)
```

STEP 4 The previous step defines the rules that make up both the algorithm and creating the string that will eventually display the value of Pi according to the Chudnovsky brother's algorithm. As you have no doubt already surmised, it would be handy to actually output the value of Pi to the screen. To rectify that we can add:

```
print(calc(1))
```

STEP 5 You can save and execute the code at this point, if you like. The output will print the value of Pi to 27 decimal places: **3.14159265358979342076684535 91**. While pretty impressive on its own, we want some user interaction, to ask the user as to how many places Pi should be calculated.

STEP 6 We can insert an input line before the Pi calculation Def command. It'll need to be an integer, as it will otherwise default to a string. We can call it numberofdigits, and use the getcontext function:

```
numberofdigits = int(input("please enter the
number of decimal place to calculate Pi to: "))
getcontext().prec = numberofdigits
```

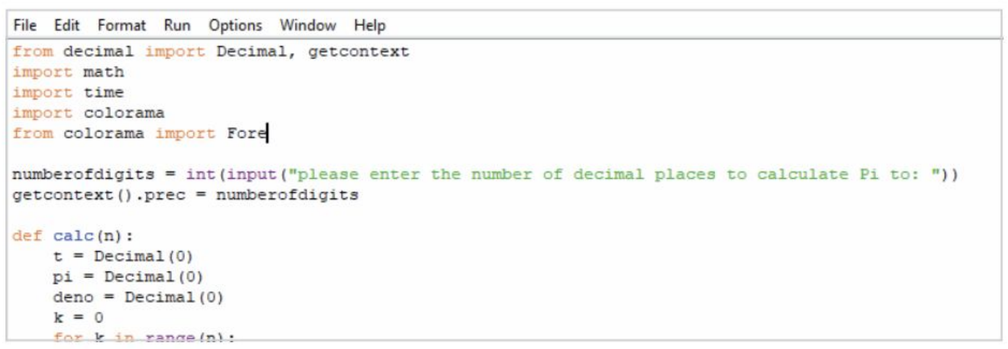
STEP 7 We can execute the code now, and it'll ask the user to how many decimal places they want to calculate Pi, and then output the result in the IDLE Shell. Try it with 1000 places, but don't go too high or else your computer will be locked up in calculating Pi.

STEP 8 Part of programming is being able to modify code, making it more presentable. Let's include an element that times how long it takes our computer to calculate the Pi decimal places, and present the information in a different colour. For this, drop into the command line and import the colorama module (RPi users already have it installed):

```
pip install colorama
```

STEP 9 Now we need to import the colorama module (which will output text in different colours), along with the Fore function (which dictates the foreground, ink, colour), and the time module to start a virtual stopwatch to see how long our calculations take:

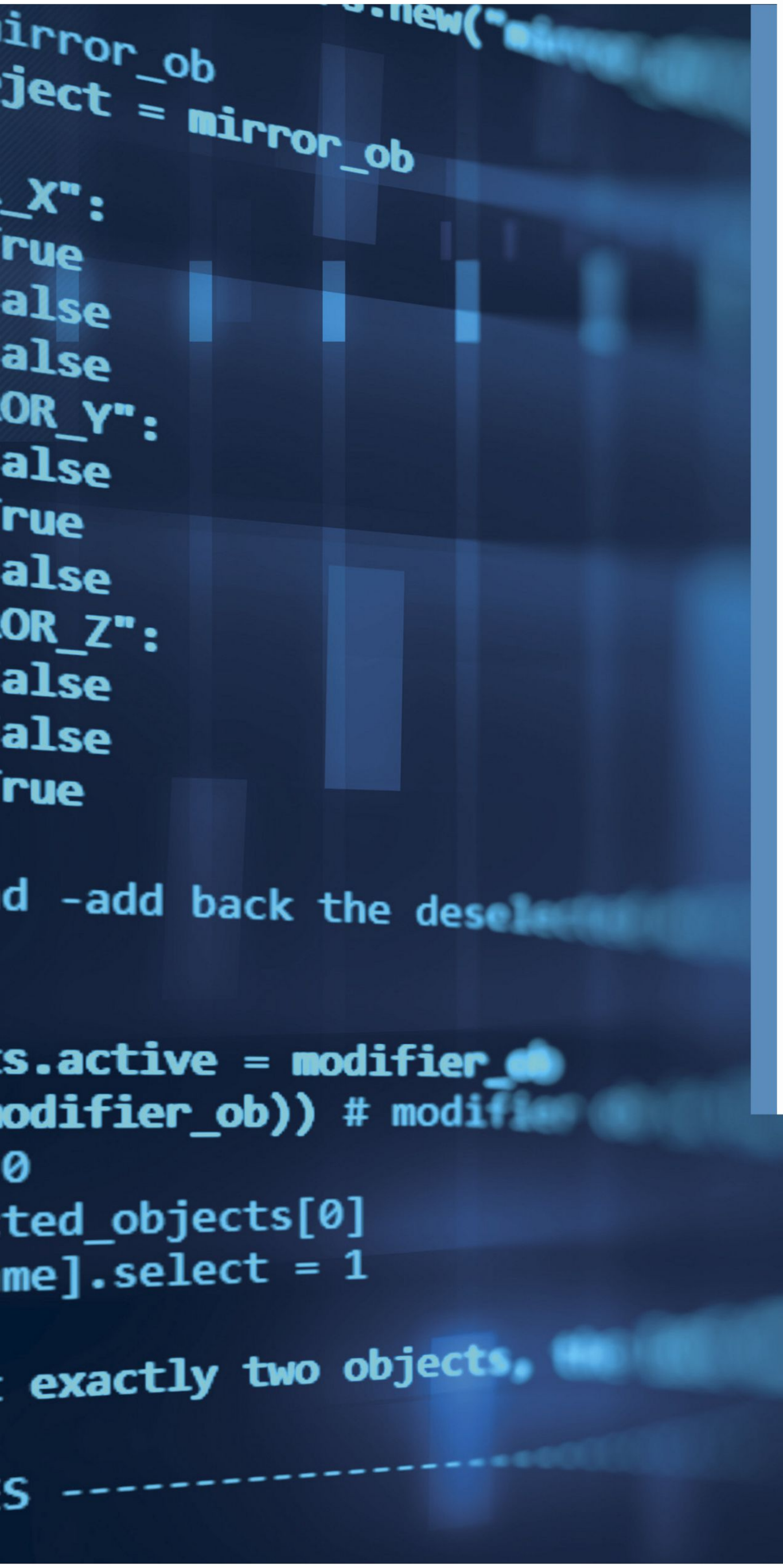
```
import time
import colorama
from colorama import Fore
```



STEP 10 To finish our code, we need to initialise the colorama module, and then start the time function at the point where the calculation starts, and when it finishes. The end result displays, in coloured ink, how long the process took (in the Terminal or command line):

```
from decimal import Decimal, getcontext
import math
import time
import colorama
from colorama import Fore
colorama.init()

numberofdigits = int(input("please enter the number
of decimal places to calculate Pi to: "))
getcontext().prec = numberofdigits

start_time = time.time()
def calc(n):
    t = Decimal(0)
    pi = Decimal(0)
    deno = Decimal(0)
    k = 0
    for k in range(n):
        t = (Decimal(-1)**k)*(math.
factorial(Decimal(6)*k))*(13591409+545140134*k)
        deno = math.factorial(3*k)*(math.
factorial(k)**Decimal(3))*(640320**(3*k))
        pi += Decimal(t)/Decimal(deno)
    pi = pi * Decimal(12)/
Decimal(640320**Decimal(1.5))
    pi = 1/pi
    return str(pi)

print(calc(1))
print(Fore.RED + "\nTime taken:", time.time() -
start_time)
```

Manipulating Data



Data is everything. It's more valuable than gold or oil. With data, governments can change the world, politicians can rise or fall, and companies, large or small, can impact our future in ways we wouldn't have imagined just a few years ago. Data is power, and learning how to manipulate and control it are essential aspects of any coding language.

Here we'll take a look at how you can use 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 open more doors for you.

66 Lists
68 Tuples
70 Dictionaries
72 Splitting and Joining Strings
74 Formatting Strings
76 Date and Time
78 Opening Files
80 Writing to Files
82 Exceptions
84 Python Graphics
86 Combining What You Know So Far

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, which 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 equal 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”]
```

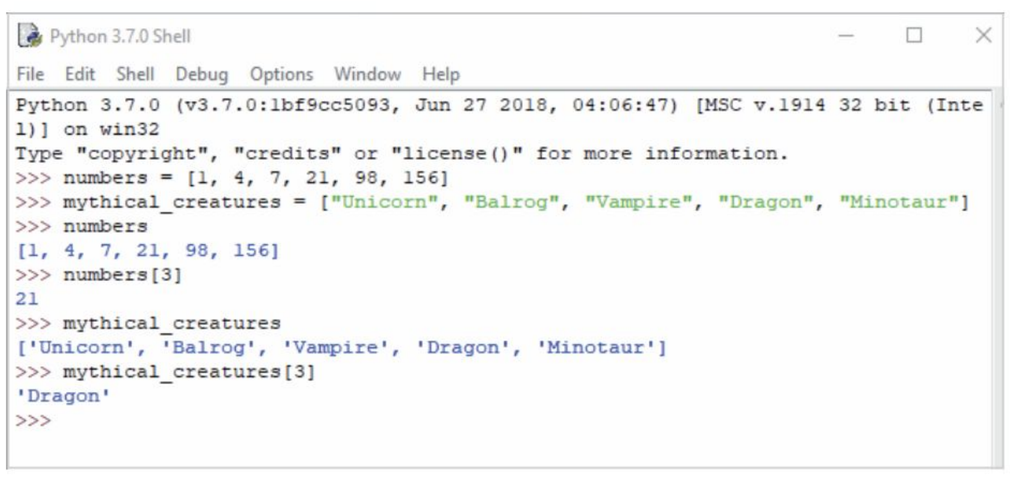
STEP 2 Once you’ve defined your list, you can call each by referencing its name followed by a number. Lists start the first item entry as 0, followed by 1, 2, 3, and so on. For example:

```
numbers
```

To call up the entire contents of the list.

```
numbers[3]
```

To call the item third from zero 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 4 and 7, those 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).

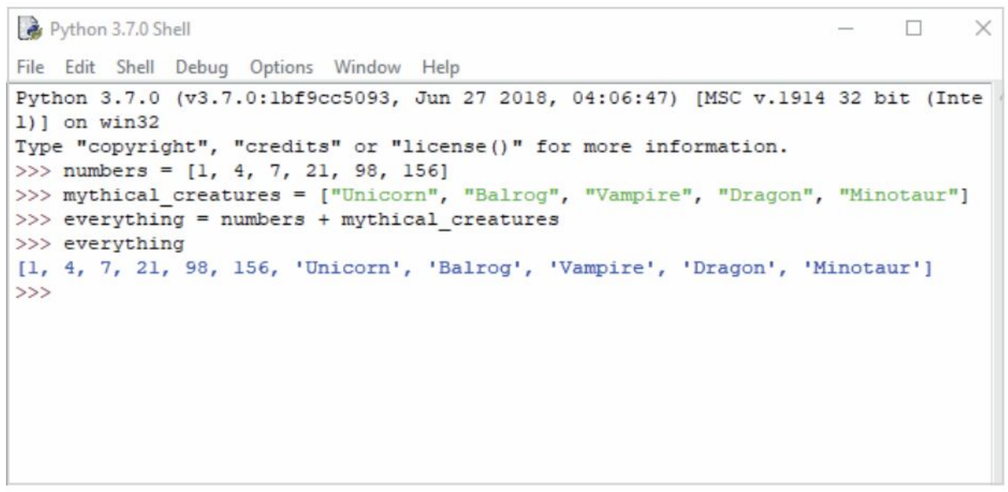
STEP 5

You can update items within an existing list, remove items, and even join lists together. For example, to join two lists we 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)
```

```
>>> numbers
[1, 4, 7, 21, 98, 156]
>>> mythical_creatures
['Unicorn', 'Balrog', 'Vampire', 'Dragon', 'Minotaur']
>>> numbers=numbers+[201]
>>> numbers
[1, 4, 7, 21, 98, 156, 201]
>>> mythical_creatures=mythical_creatures+["Griffin"]
>>> mythical_creatures
['Unicorn', 'Balrog', 'Vampire', 'Dragon', 'Minotaur', 'Griffin']
>>> mythical_creatures.append("Nessie")
>>> mythical_creatures
['Unicorn', 'Balrog', 'Vampire', 'Dragon', 'Minotaur', 'Griffin', 'Nessie']
>>> numbers.append(278)
>>> numbers
[1, 4, 7, 21, 98, 156, 201, 278]
>>>
```

STEP 7

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

```
del numbers[7]
```

The second, by item name:

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

```
>>> mythical_creatures
['Unicorn', 'Balrog', 'Vampire', 'Dragon', 'Minotaur']
>>> numbers=numbers+[201]
>>> numbers
[1, 4, 7, 21, 98, 156, 201]
>>> mythical_creatures=mythical_creatures+["Griffin"]
>>> mythical_creatures
['Unicorn', 'Balrog', 'Vampire', 'Dragon', 'Minotaur', 'Griffin']
>>> mythical_creatures.append("Nessie")
>>> mythical_creatures
['Unicorn', 'Balrog', 'Vampire', 'Dragon', 'Minotaur', 'Griffin', 'Nessie']
>>> numbers.append(278)
>>> numbers
[1, 4, 7, 21, 98, 156, 201, 278]
>>> del numbers[7]
>>> numbers
[1, 4, 7, 21, 98, 156, 201]
>>> mythical_creatures.remove("Nessie")
>>> mythical_creatures
['Unicorn', 'Balrog', 'Vampire', 'Dragon', 'Minotaur', 'Griffin']
>>>
```

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:

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

Inserts the number 62 at item index 4. And:

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

Will remove it.

```
educe__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__'
, '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear',
 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort
']
>>> numbers = [1, 4, 7, 21, 98, 156]
>>> numbers
[1, 4, 7, 21, 98, 156]
>>> numbers.insert(4, 62)
>>> numbers
[1, 4, 7, 21, 62, 98, 156]
>>> numbers.pop(4)
62
>>> numbers
[1, 4, 7, 21, 98, 156]
>>>
```

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
```

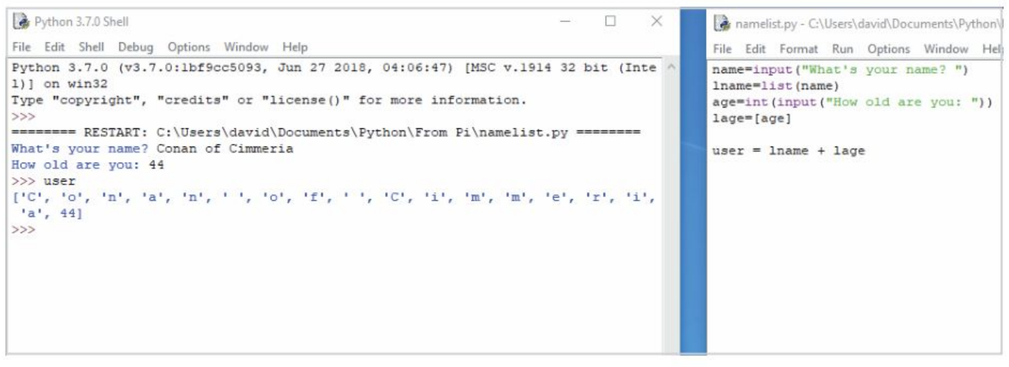
```
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [MSC v.1914 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> list("David")
['D', 'a', 'v', 'i', 'd']
>>> name=list("David Hayward")
>>> name
['D', 'a', 'v', 'i', 'd', ' ', 'H', 'a', 'y', 'w', 'a', 'r', 'd']
>>> age=[46]
>>> user = name + age
>>> user
['D', 'a', 'v', 'i', 'd', ' ', 'H', 'a', 'y', 'w', 'a', 'r', 'd', 46]
>>>
```

STEP 10

Based on that, we 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 virtually 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 it's perfect for storing fixed data items.

THE IMMUTABLE TUPLE

Reasons for having tuples vary depending on what the program is intended to do. Mostly a tuple is reserved for something special, but they're also used, as an 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 As with lists, the items within a named tuple can be indexed according to their position in the data range:

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

And so on. 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 we have 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], however 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
```

This, as you can imagine, gets a little confusing when you have a lot of tuple data to deal with.

```
>>> NPC[0][0]
'Conan'
>>> NPC[0][1]
100
>>> NPC[1]
('Belit', 80)
>>> NPC[1][0]
'Belit'
>>> NPC[1][1]
80
>>> NPC[2]
('Valeria', 95)
>>> NPC[2][0]
'Valeria'
>>> NPC[2][1]
95
>>>
```

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):

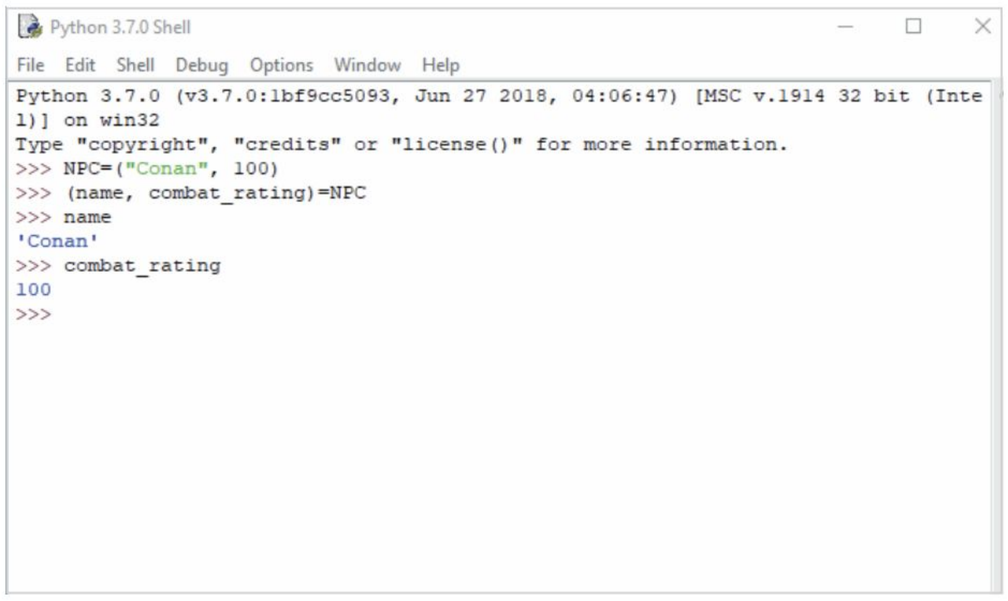
```
NPC=("Conan", 100)
```

STEP 8

Now unpack the tuple into two corresponding variables:

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

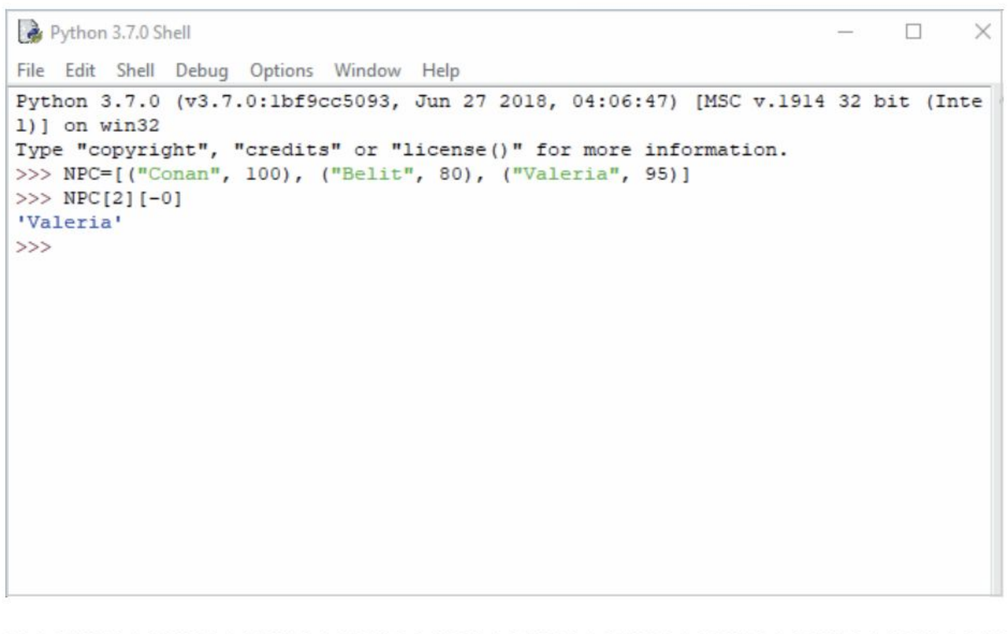
You can now check the values by entering **name** and **combat_rating**.



STEP 9

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

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



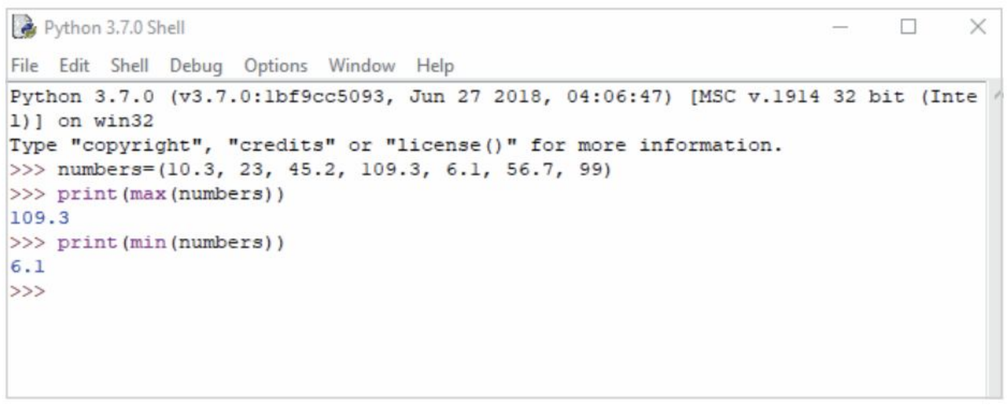
STEP 10

We 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. Although they can be tricky to get to grips with at first, 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, but the Value can be any data item you like.

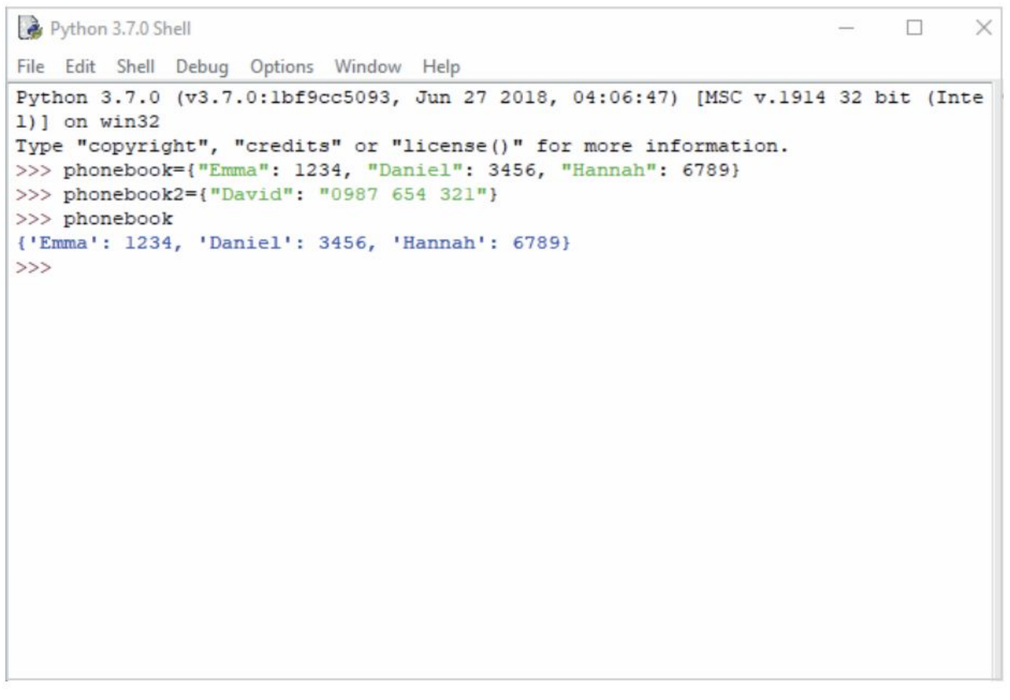
STEP 1 Let's say you want to create a phonebook in Python. You would create the dictionary name, and contain the data in curly brackets, separating the key and value by a colon `Key:Value`. For example:

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

STEP 2 As with most lists, tuples, and so on, strings need to be enclosed in quotes (single or double), while 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:

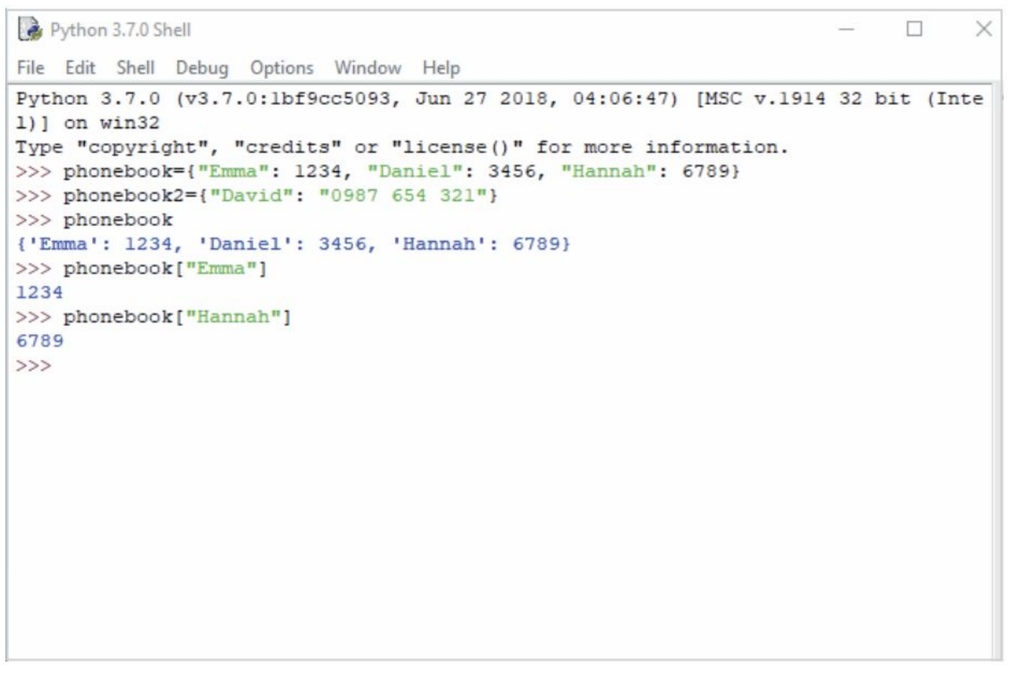
```
phonebook2={"David": "0987 654 321"}
```

STEP 3 As with lists and tuples, you can check the contents of a dictionary by calling the dictionary 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, we 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 as such:

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

STEP 6

And you can also remove items from a dictionary by issuing the del command followed by the item’s key – the value will also be removed as well, since both work as a pair of data items:

```
del phonebook[“David”]
```

STEP 7

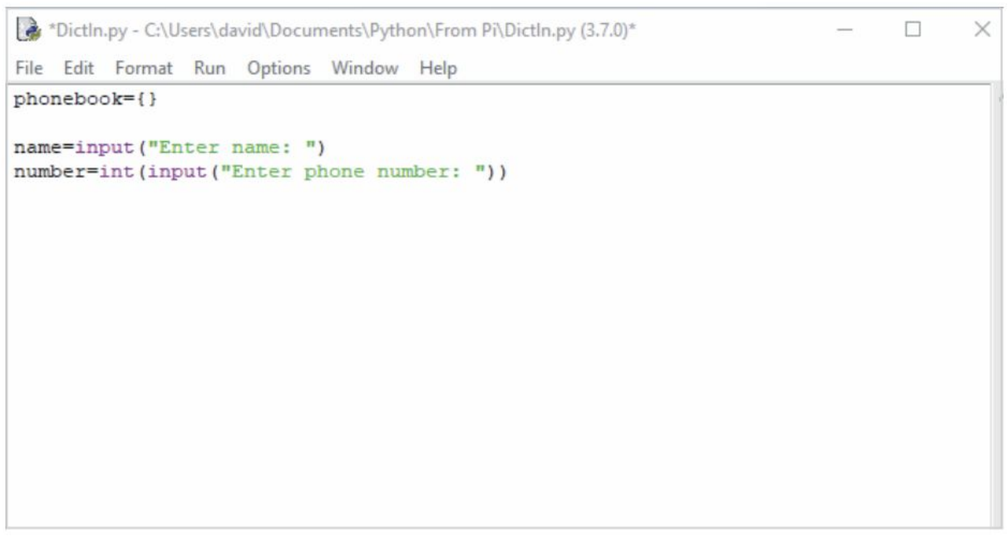
Taking this a step further, how about creating a piece of code that will ask the user for the dictionary key and value items? Create a new Editor instance, and start by coding in a new, blank dictionary:

```
phonebook={}
```

STEP 8

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

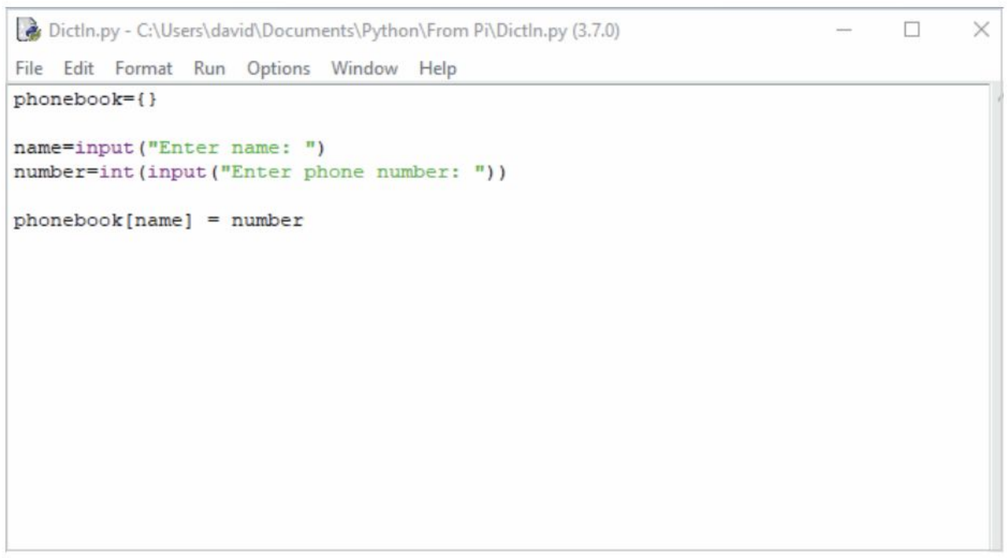
```
name=input(“Enter name: “)
number=int(input(“Enter phone number: “))
```



STEP 9

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

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

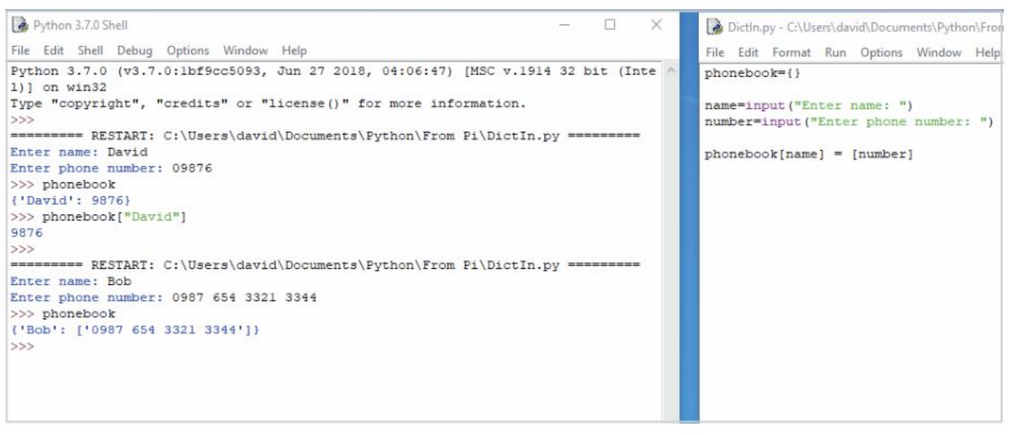


STEP 10

Now when we 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 we can test by entering into the Shell:

```
phonebook
phonebook[“David”]
```

If the number needs to contain spaces you’ll 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

We'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(" ")
```

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

STEP 3

Note that the text.split part contains 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 we would use:

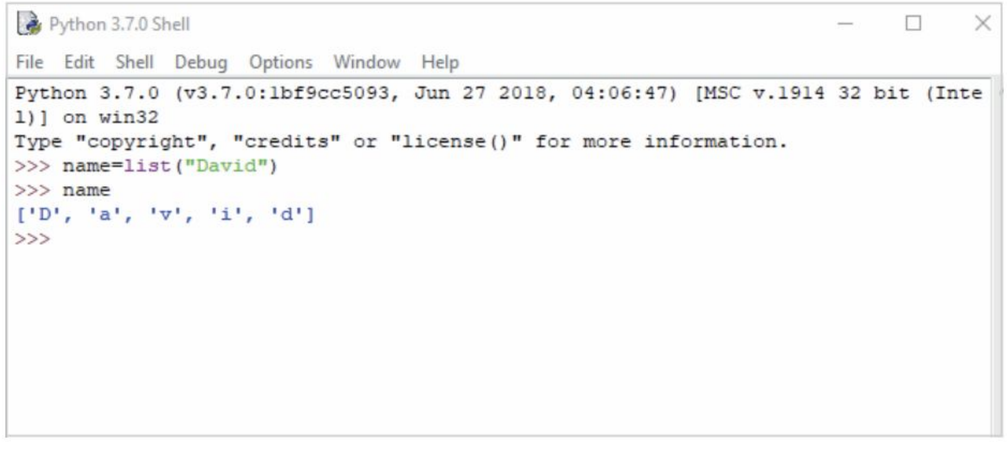
```
text="January,February,March,April,May,June"
months=text.split(",")
months
```

STEP 4

We've previously seen how, using a name, we can split a string into individual letters as a list:

```
name=list("David")
name
```

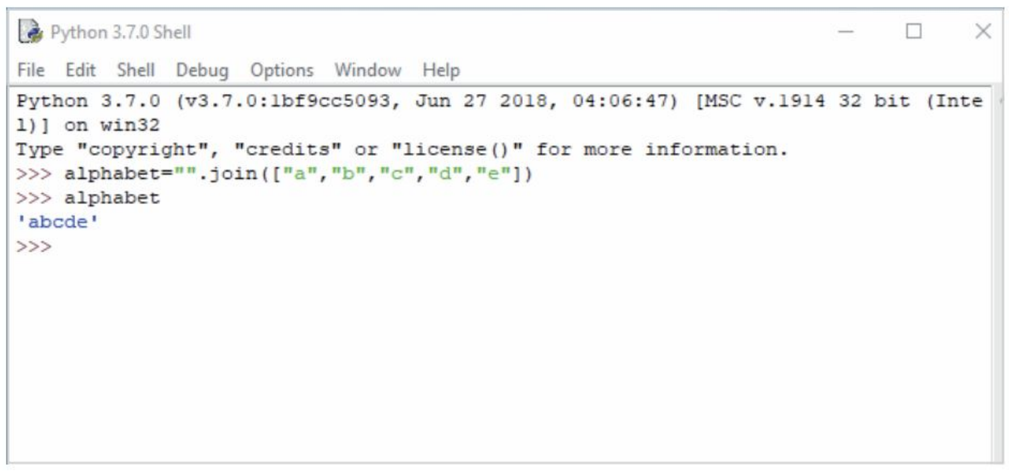
The returned value is 'D', 'a', 'v', 'i', 'd'. While it may seem a little useless under ordinary circumstances, it could be handy for creating a spelling game, for example.



STEP 5 The converse of the .split function is .join, where separate items in a string, can join 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
```

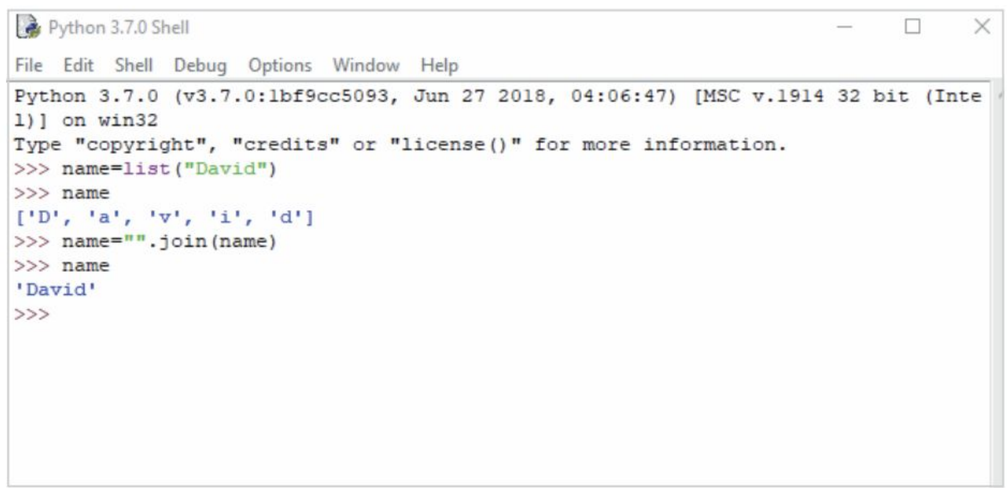
This will display 'abcde' in the Shell.



STEP 6 We can therefore apply .join to the separated name we made in Step 4, combining the letters again to form the name:

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

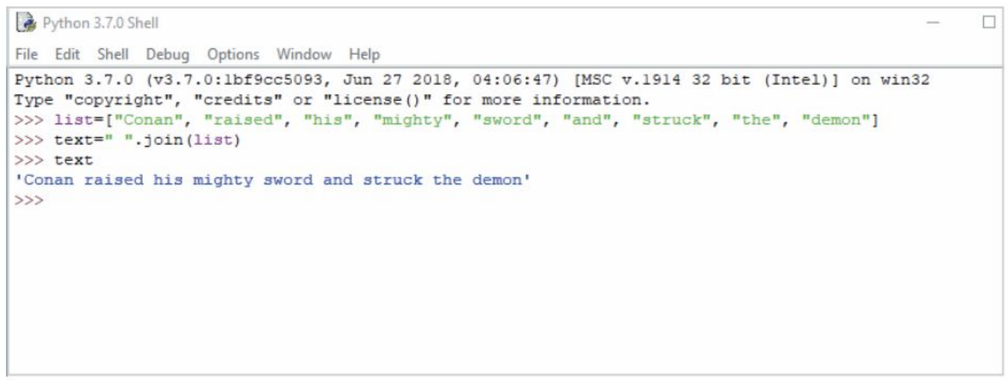
We've joined the string back together, and retained the list called name, passing it through the .join function.



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:

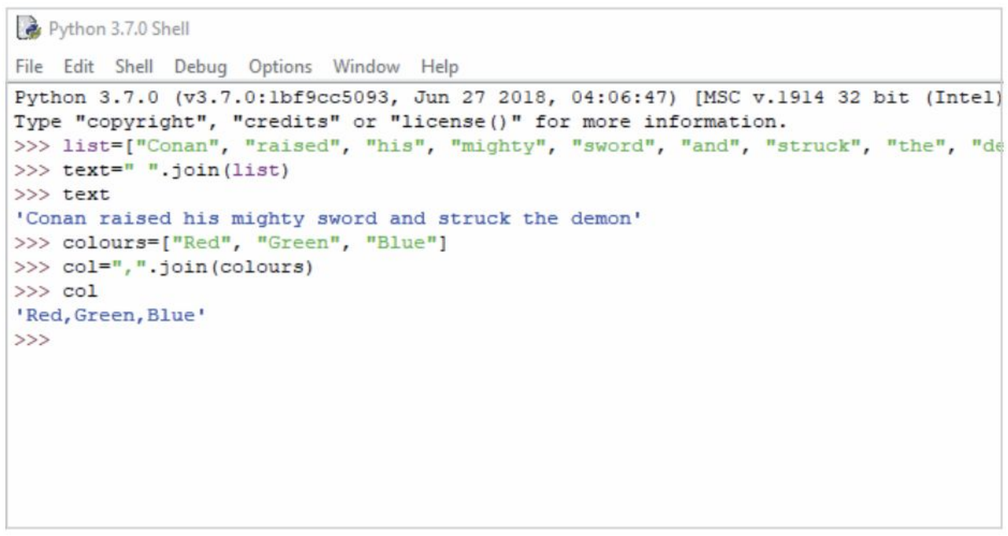
```
list=["Conan", "raised", "his", "mighty", "sword",
"and", "struck", "the", "demon"]
text=" ".join(list)
text
```

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, or a full stop, or a hyphen, or whatever you like:

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

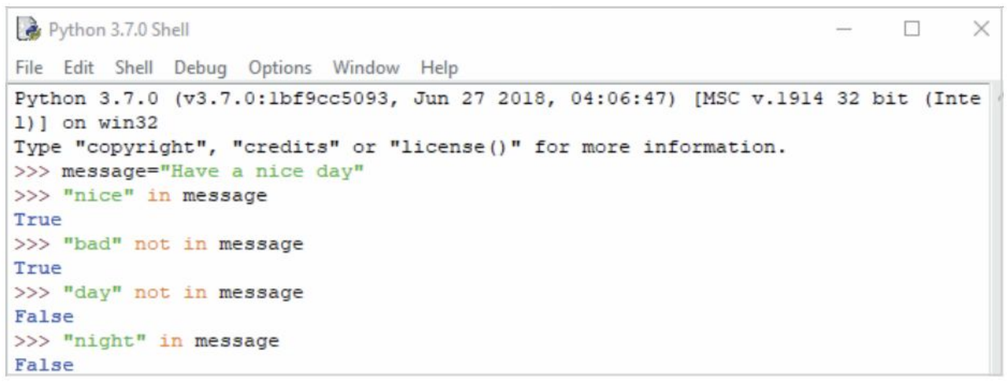


STEP 9 There's some interesting functions you can apply to a string, such as .capitalize and .title. For example:

```
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

As you work with data, creating lists, dictionaries, and objects you'll often want to print out the results. Merging strings with data is easy, especially with Python 3, as earlier versions of Pythons 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. This makes things easier to follow and, as with most coding, there are other ways to do things but this way is recommended.

STEP 1

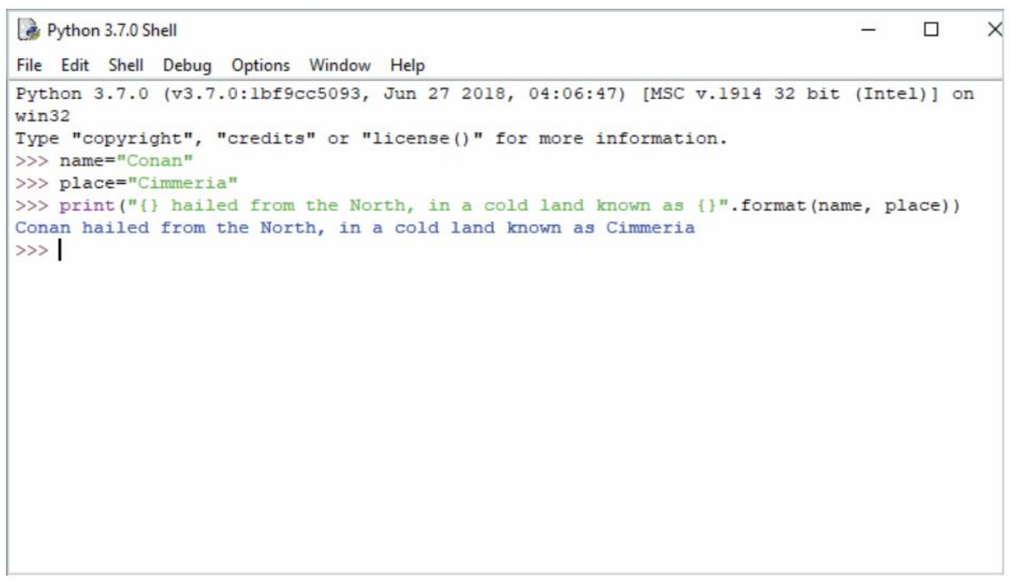
The basic formatting in Python is to call each variable into the string using the curly brackets:

```
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:

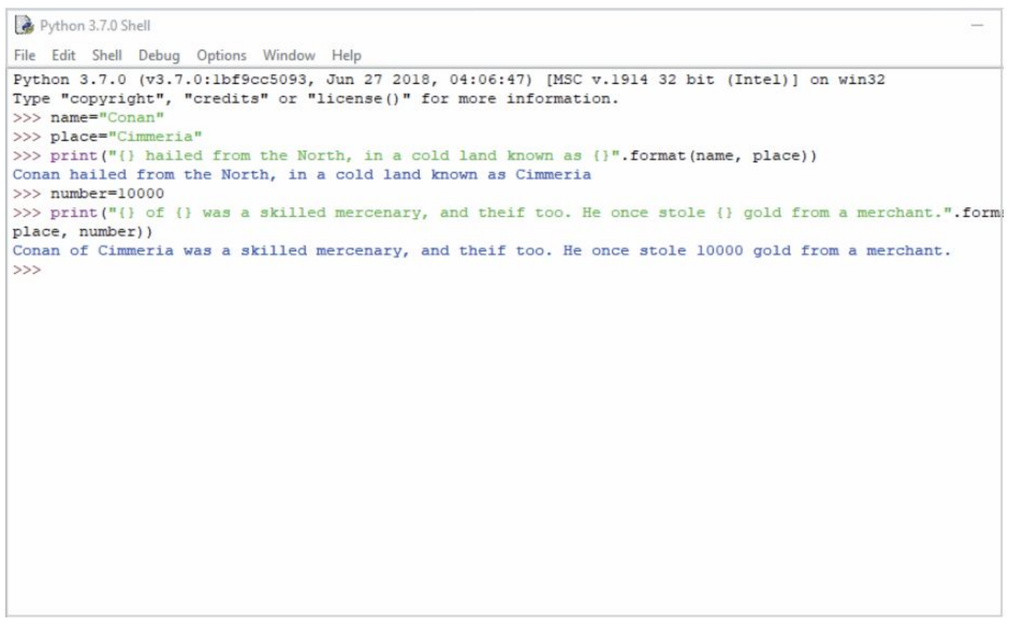
```
name="Conan"
place="Cimmeria"
print("{} hailed from the North, in a cold land known as {}".format(name, place))
```



STEP 3

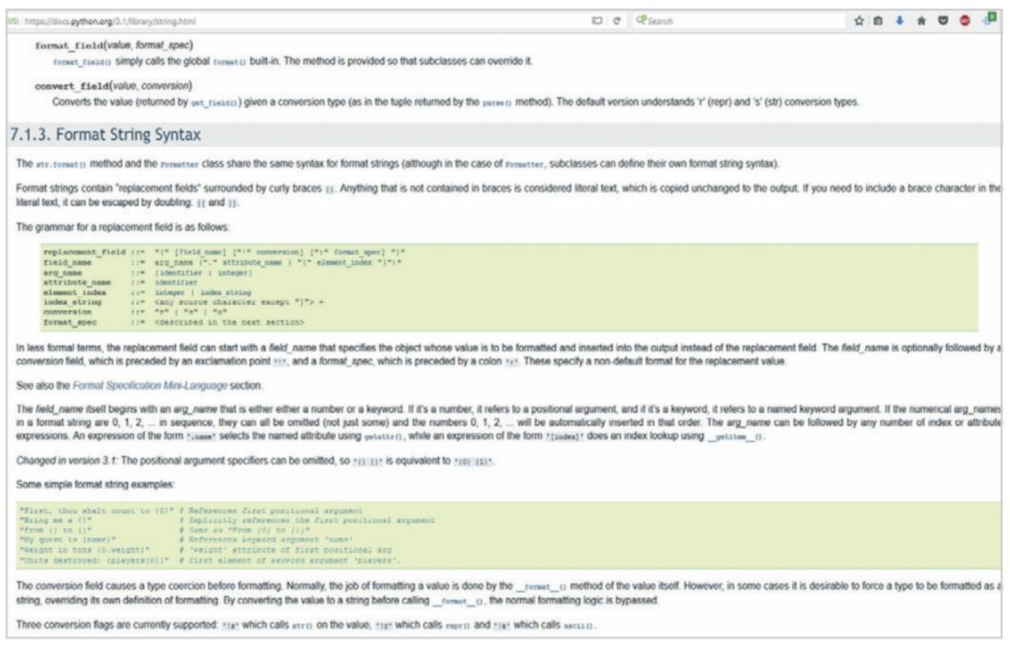
We can, of course, also include integers into the mix:

```
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, and 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 **https://docs.python.org/3.1/library/string.html**. Here, you will find tons of help.

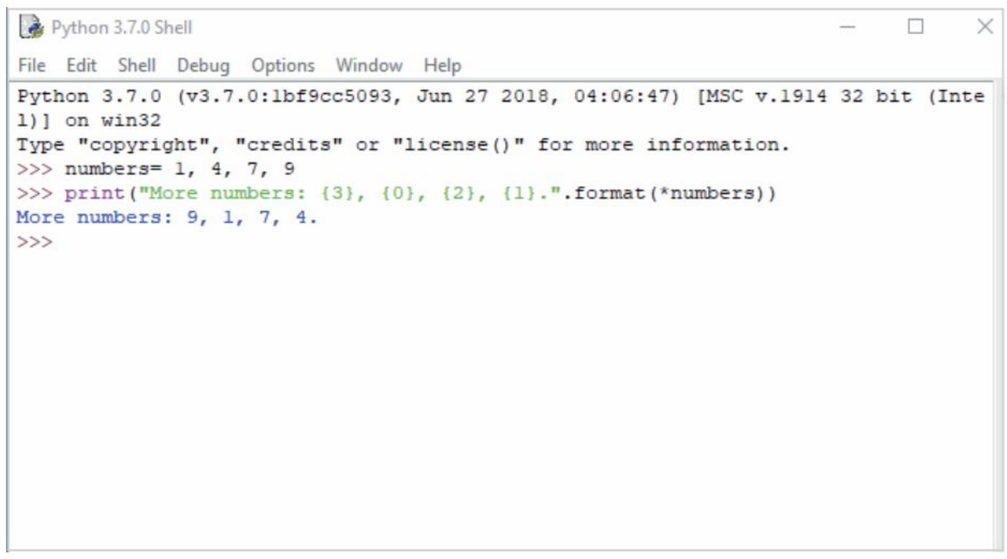


STEP 5 Interestingly, you can reference a list using the string formatting function. You need to place an asterisk in front of the list name:

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

STEP 6 As with indexing in lists, the same applies to calling a list using string formatting. We 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))
```

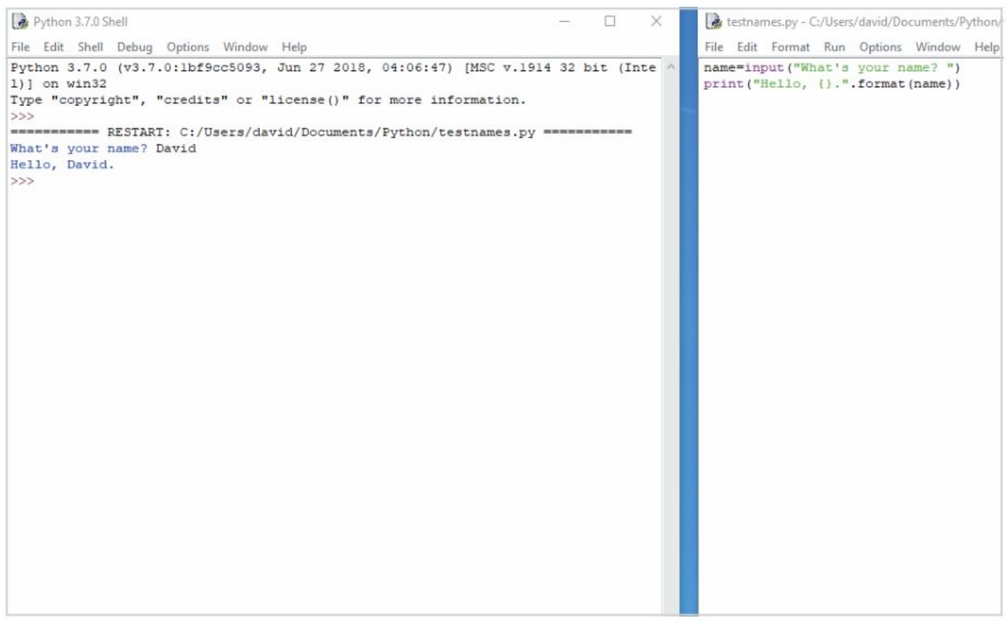


STEP 7 And as you probably suspect, you can mix strings and integers in a single list to be called in the .format function:

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

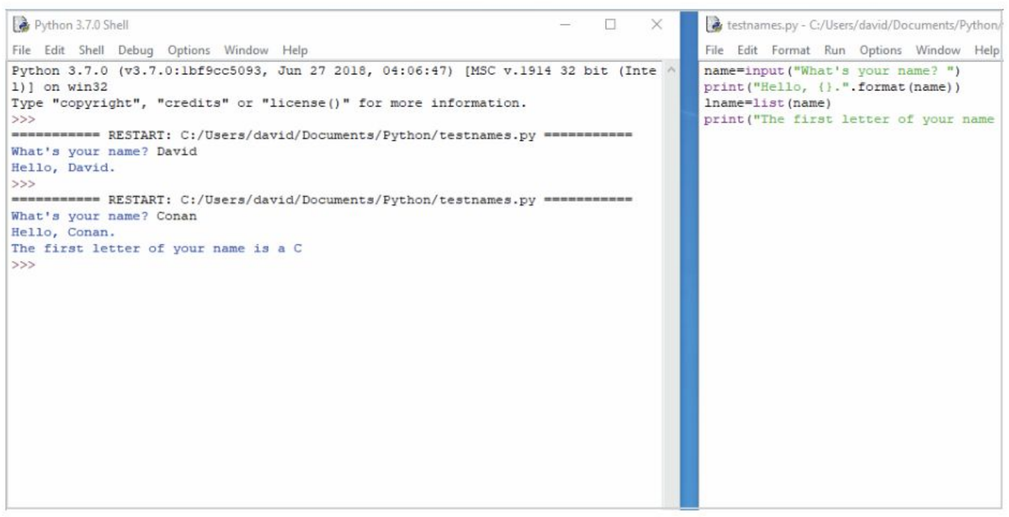
STEP 8 You can print out the content of a user's input in the same fashion:

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



STEP 9 And 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 at the code from Step 7, we can alter it with:

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

Creating two lists. Now we 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

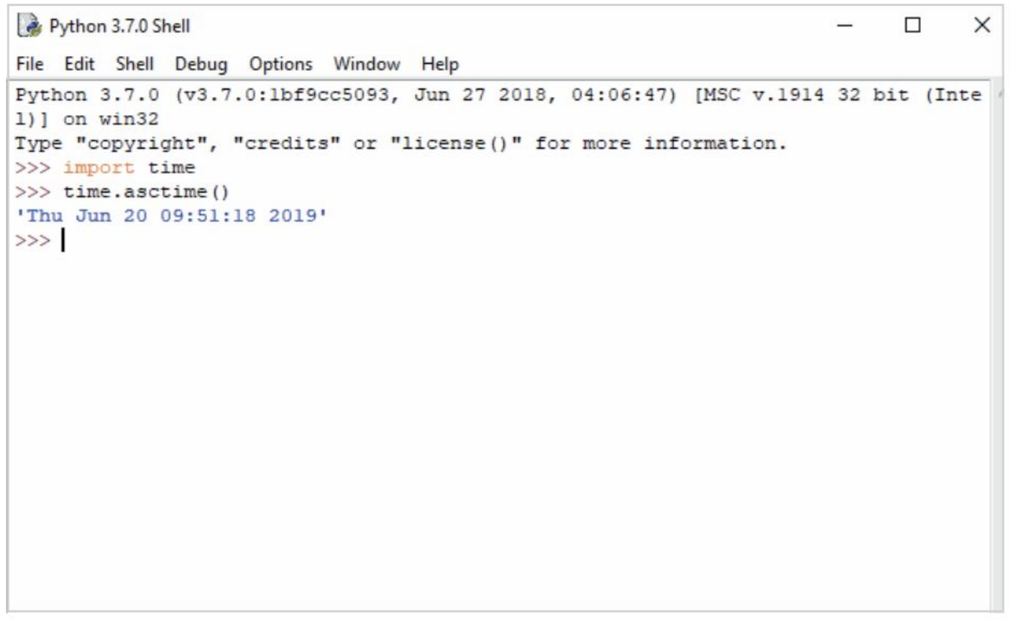
When working with data it's often handy to have access to the time. For example, you may want to time-stamp an entry, or see at what time a user logged into the system, and for how long. Thankfully, acquiring the time and date is easy thanks to the Time module.

TIME LORDS

The Time module contains functions that help you retrieve the current system time, read the date from strings, format 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, we 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=2019, 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 we can use:

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

STEP 5 Naturally, this 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")
```

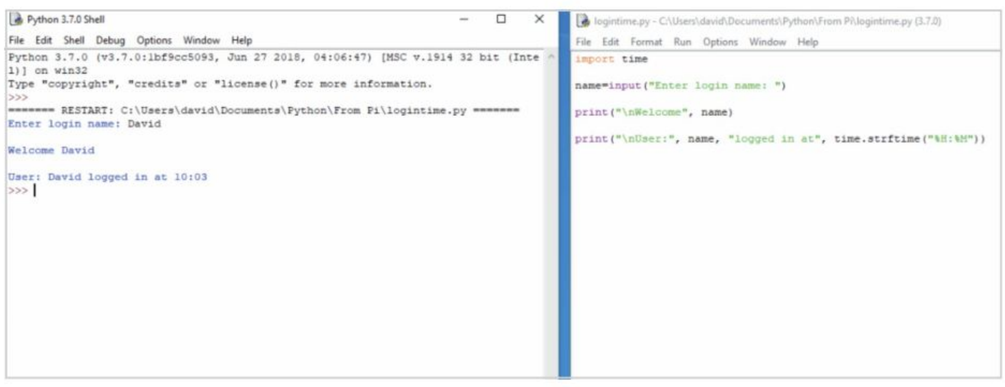
STEP 6 Note the last two entries, with %H and %H%M, as you can see, these are the hours and minutes and as the last entry indicates, entering them as %H%M doesn't display the time correctly in the Shell. We can easily rectify this with:

```
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.

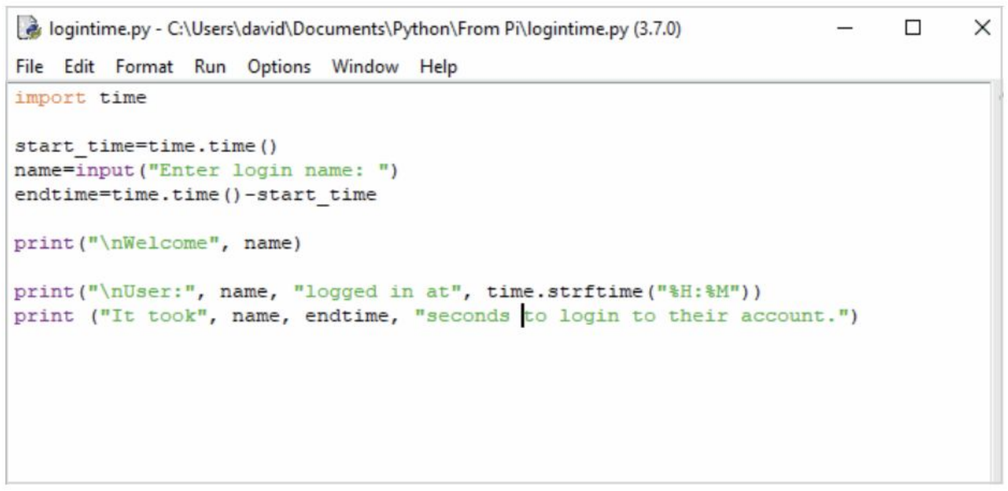


STEP 8 We can also use the Time module to display the amount of time taken for an event to happen. For example, taking the above code, we can alter it slightly by including:

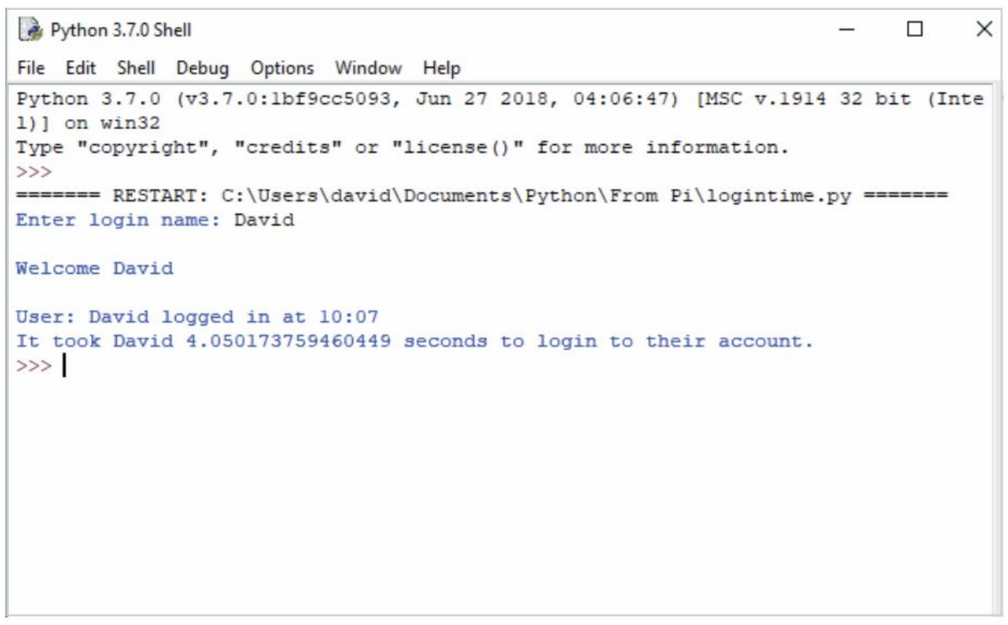
```
start_time=time.time()
```

And.

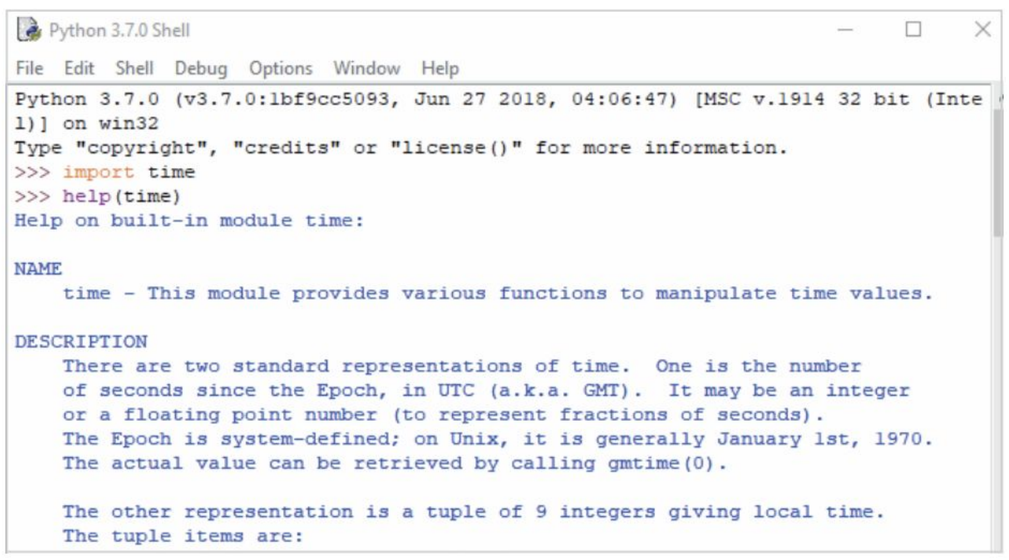
```
endtime=time.time()-start_time
```



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.



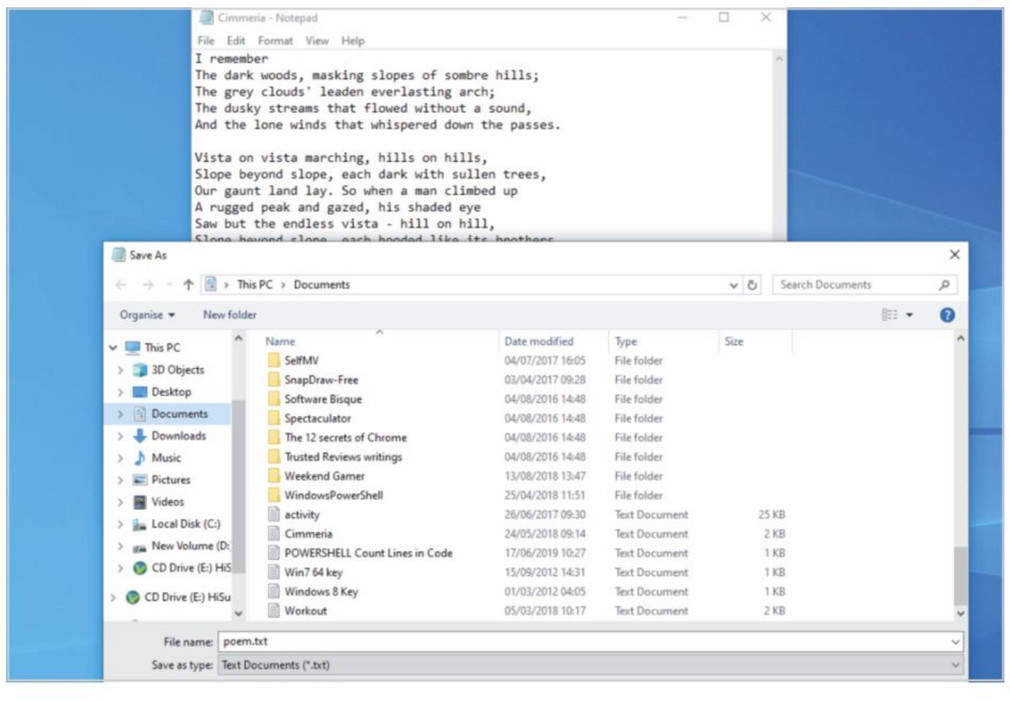
Opening Files

In Python, you can read text and binary files in your programs. This enables you to import data from one source to Python; handy if you have another program language running, that's creating an output, and you want to analyse the results in Python.

OPEN, READ AND WRITE

In Python, you create a file object, similar to creating a variable, only you 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 preferable to 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, and we've saved 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:

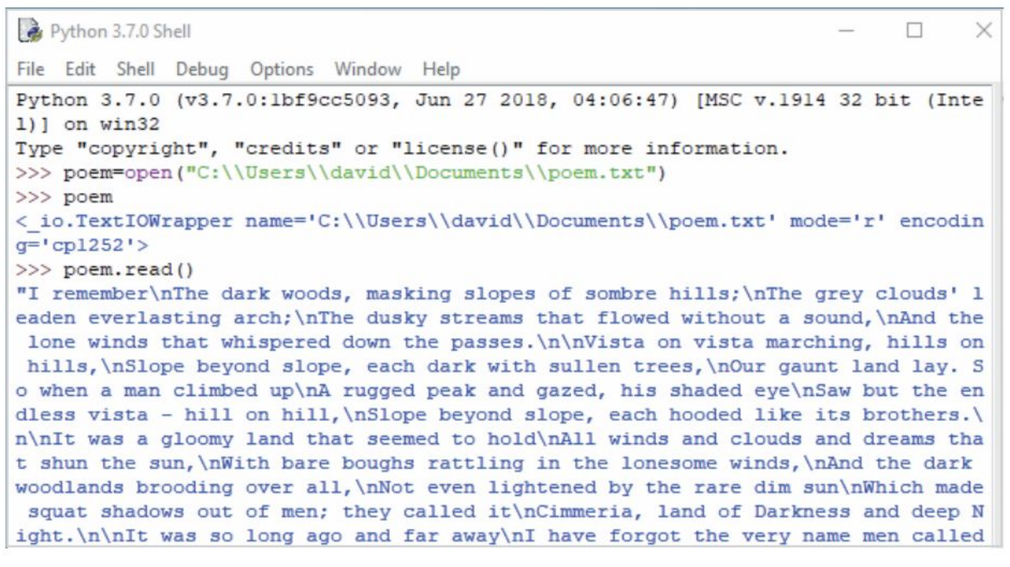
```
poem=open("C:\\Users\\david\\Documents\\poem.txt")
```

The reason for the double slash (\\) is because Python will read this as a Unicode Error, thinking you've entered: \U. This is Windows-only, Linux and Mac won't have this issue.

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. We can now use the **poem** variable to read the contents of the file:

```
poem.read()
```

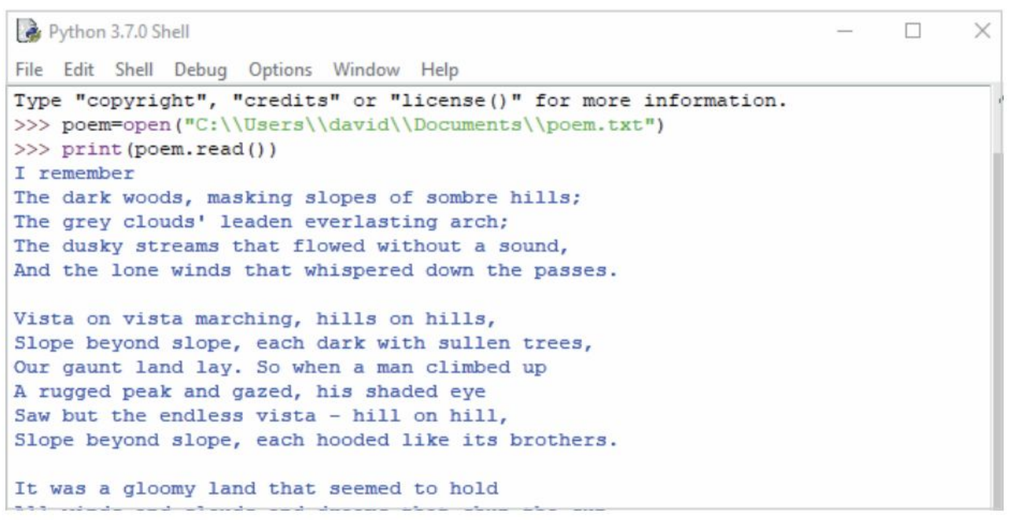
Note than a **/n** entry in the text represents a new line, as we have used previously.



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("C:\\Users\\david\\Documents\\poem.txt")` again to recreate the file. This time, however, enter:

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

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



STEP 5

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, while entering:

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

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

STEP 6

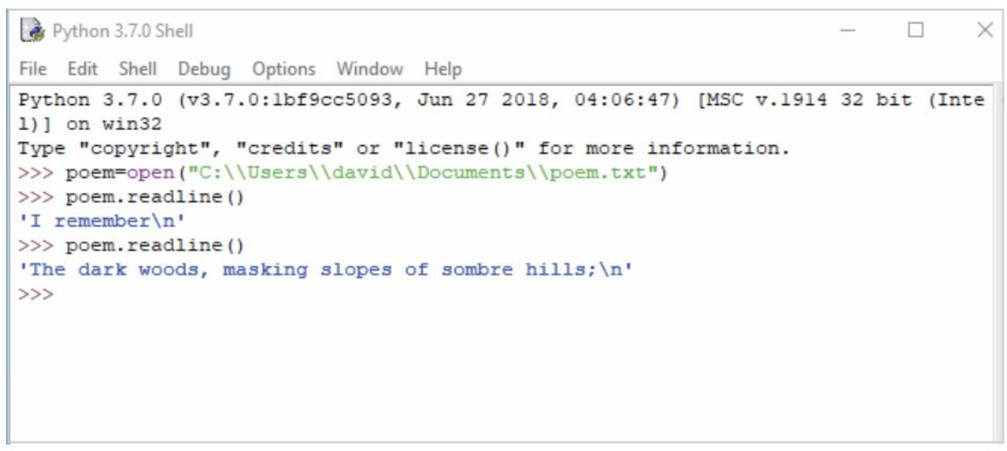
Similarly, you can display one line of text at a time by using the **readline()** function. For example:

```
poem=open("C:\\Users\\david\\Documents\\poem.txt")
poem.readline()
```

Will display the first line of the text. And:

```
poem.readline()
```

Will display the next line of text.



STEP 7

As you may suspect, you can pass the **readline()** function into a variable, allowing you to call it again, when needed:

```
poem=open("C:\\Users\\david\\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("C:\\Users\\david\\Documents\\poem.txt")
lines=poem.readlines()
lines[0]
lines[1]
lines[2]
```

STEP 9

We can also use the for statement to read the lines of text back to us:

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

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

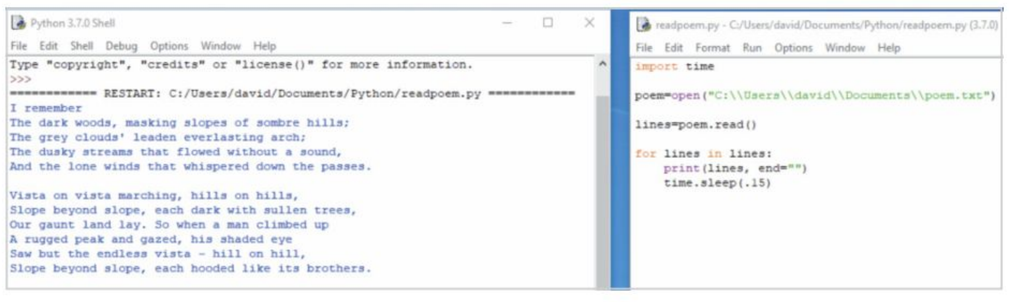
```
poem=open("C:\\Users\\david\\Documents\\poem.txt")
for lines in poem:
        print(lines)
```

STEP 10

Let's imagine that you wanted to print the text a character at a time, as would an old dot matrix printer. We can use the Time module mixed with what we've looked at here. Try this:

```
import time
poem=open("C:\\Users\\david\\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

Being able to read external files within Python is certainly handy, but writing to a file can be even more useful. Using the write() function, you're able to output the results of a program to a file, which you can then use to read() back into Python, or as a text file for perusal later.

WRITE AND CLOSE

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

STEP 1 Start by opening IDLE and enter the following (obviously entering your own username location):

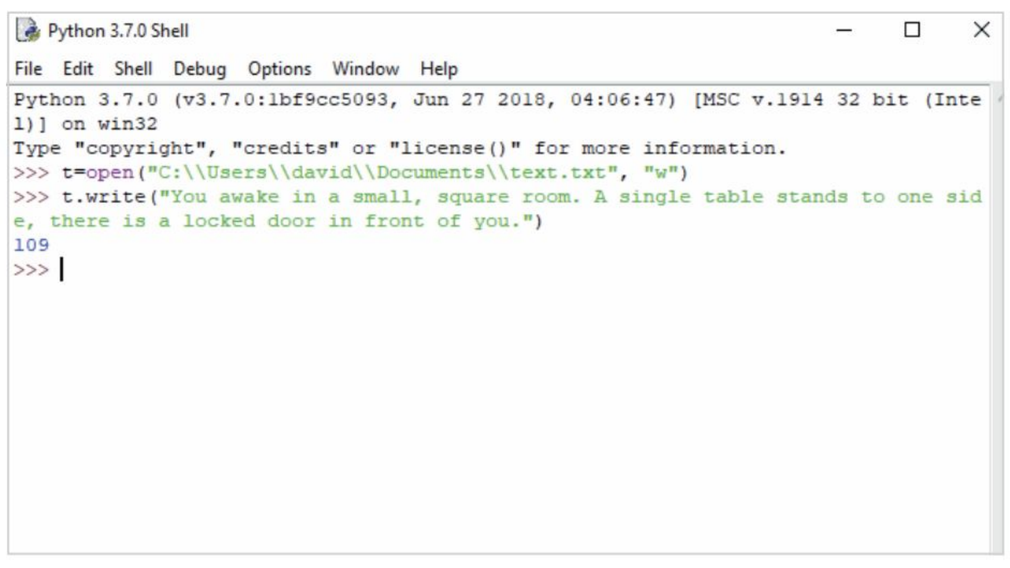
```
t=open("C:\\Users\\david\\Documents\\text.txt", "w")
```

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

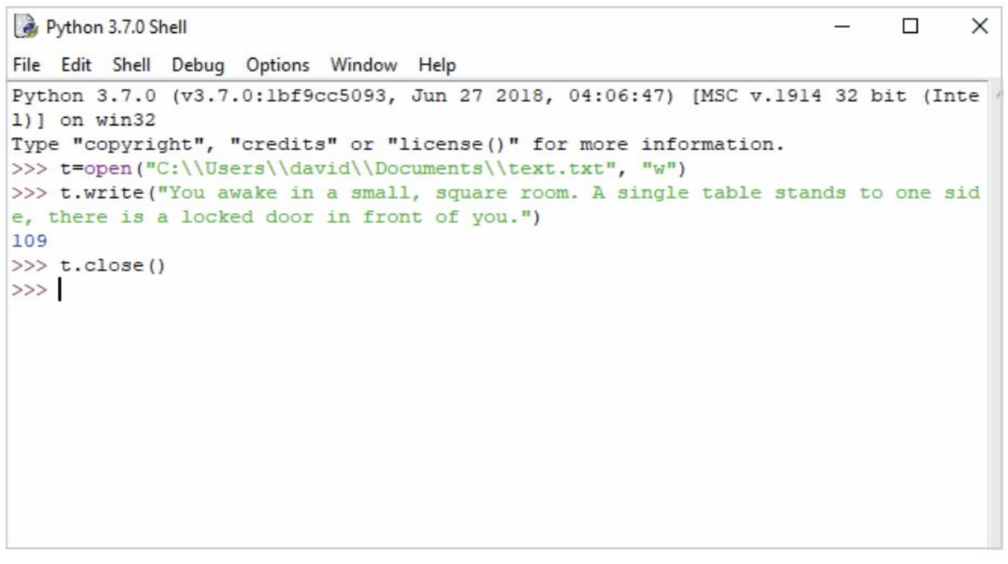
```
t.write("You awake in a small, square room. A single table stands to one side, there is a locked door in front of you.")
```

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

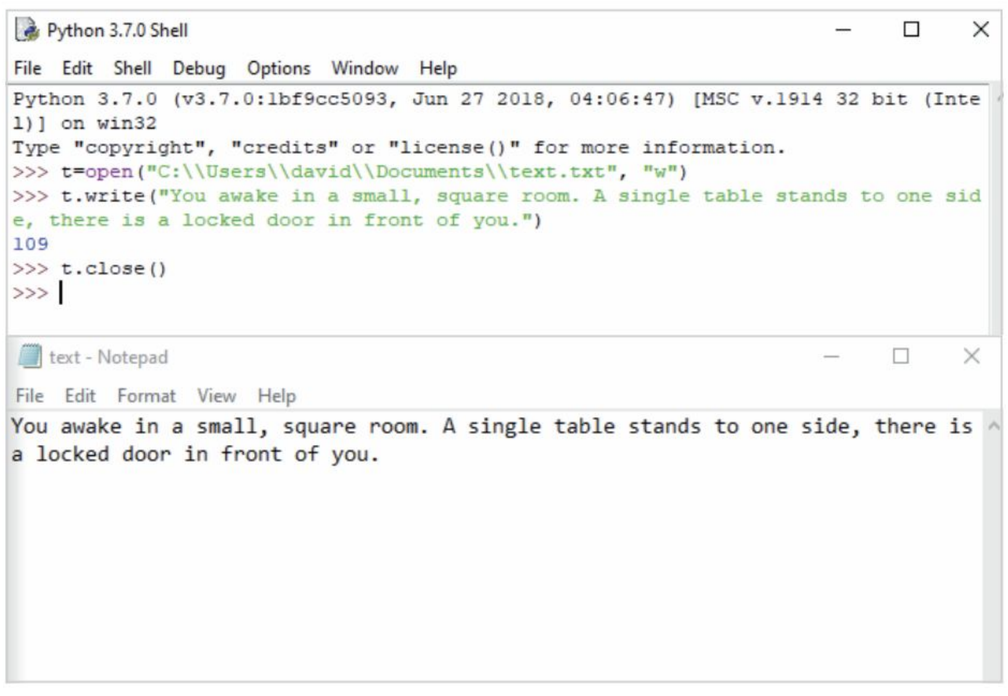


STEP 3 However, the actual text file is still blank (you can check by opening it up). This is because you've written the line of text to the file object, but not committed it to the file itself. Part of the write() function is that we need to commit the changes to the file, we can do this by entering:

```
t.close()
```



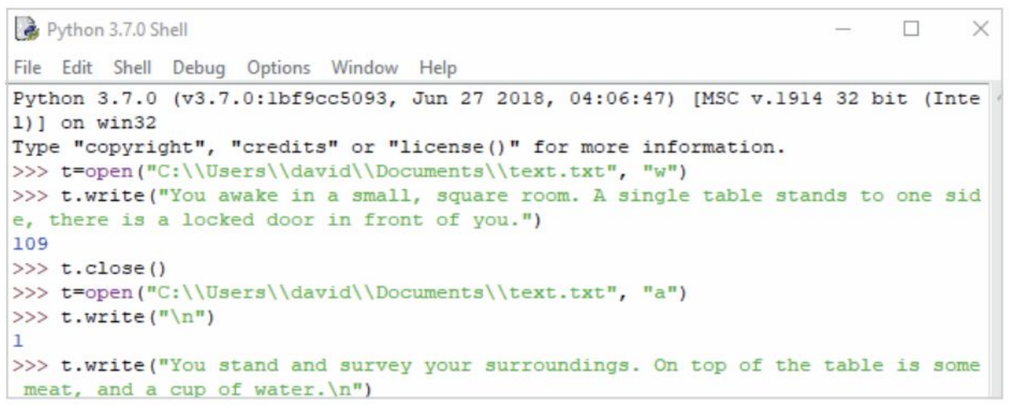
STEP 4 If you now open the text file with a text editor, you'll see that the line you created has been written to the file. This gives us the foundation for some interesting possibilities, perhaps the creation of your own log file, or even the beginning of an adventure game.



STEP 5

To expand this code, we can re-open 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:

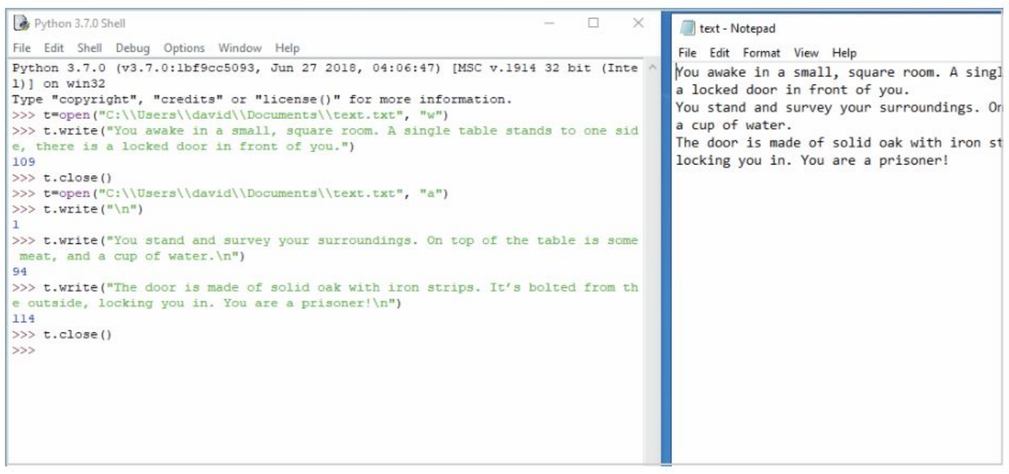
```
t=open("/home/pi/Documents/text.txt","a")
t.write("\n")
t.write(" You stand and survey your surroundings. On top of the table is some meat, and a cup of water.\n")
```



STEP 6

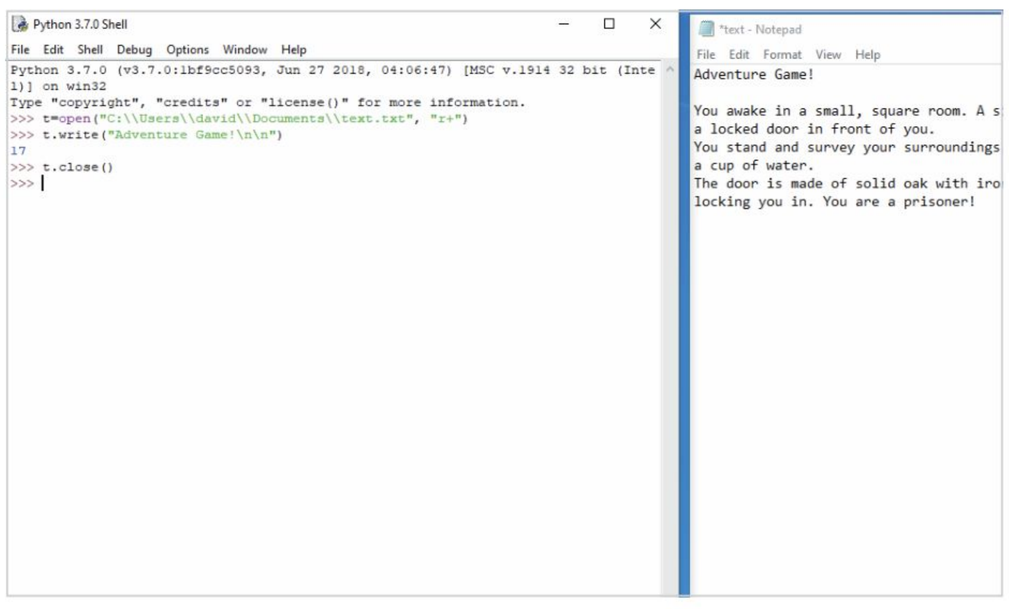
We can keep extending the text line by line, ending each with a new line (\n). When you're done, finish the code with t.close(), and open the file in a text editor to see the results:

```
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

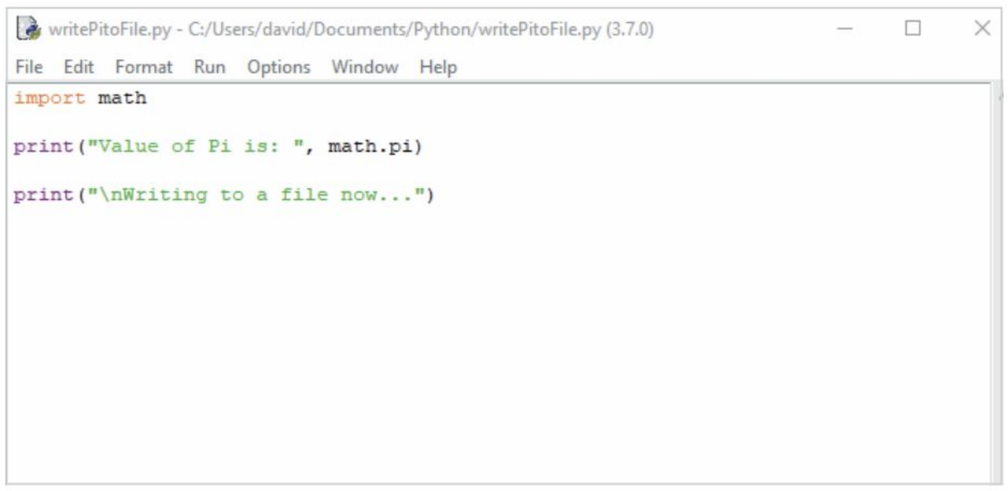
There are various types of file access to consider using the open() function. Each depends on how the file is accessed, and even the position of the cursor. For example, r+ opens a file in read and write, and places the cursor at the start of the file.



STEP 8

We can pass variables to a file that we've created in Python. Perhaps we want the value of Pi to be written to a file. We 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

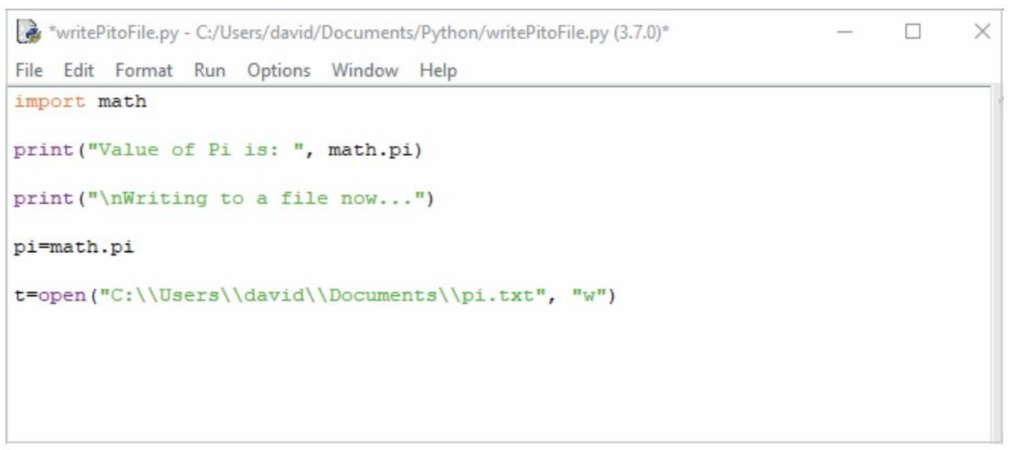
Now let's create a variable called pi, and assign it the value of Pi:

```
pi=math.pi
```

We also need to create a new file to write Pi to:

```
t=open("C:\\Users\\david\\Documents\\pi.txt", "w")
```

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

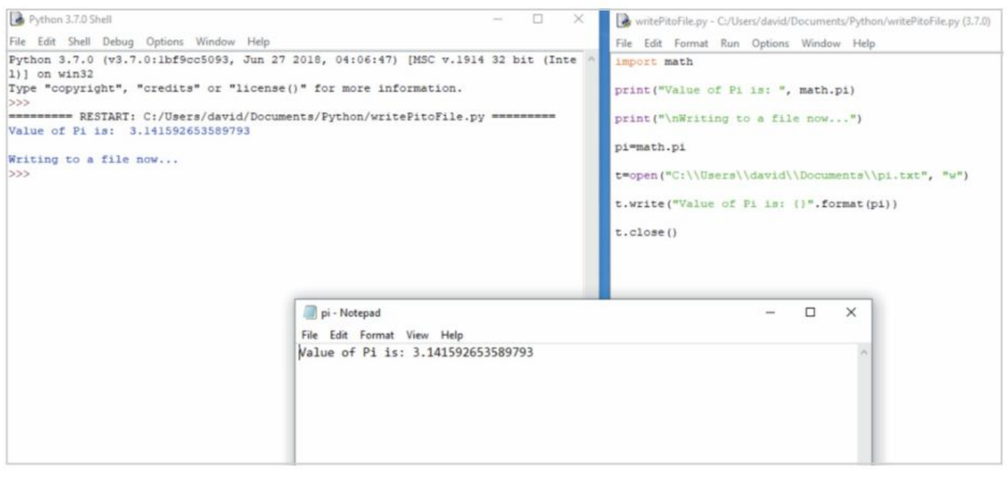


STEP 10

To finish, we 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()
```

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



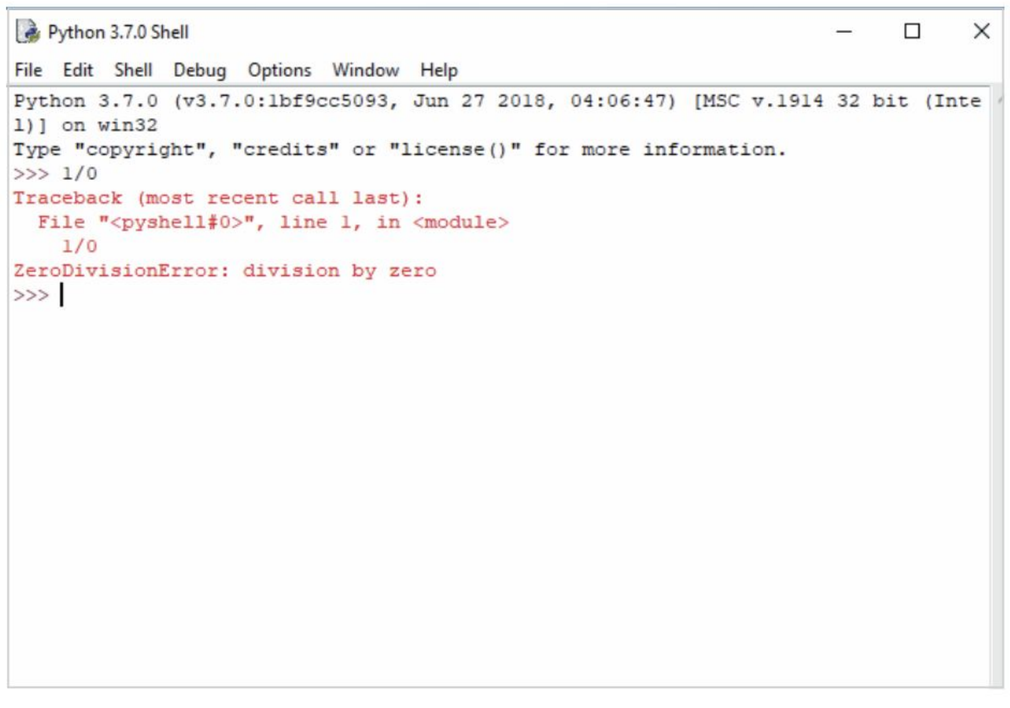
Exceptions

As your code begins to form and lengthen, you'll naturally come across some exceptional circumstances that are mostly 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. We 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.

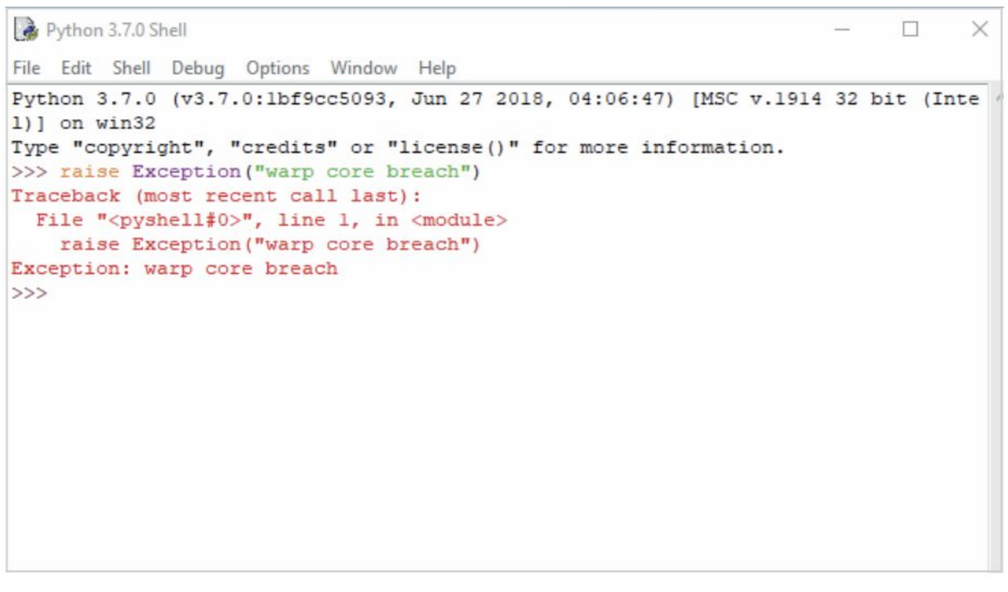


STEP 2 Most exceptions are raised automatically when Python comes across something that's inherently wrong with the code. However, we can create our own exceptions that are designed to contain the potential error and react to it, as opposed to letting the code fail.

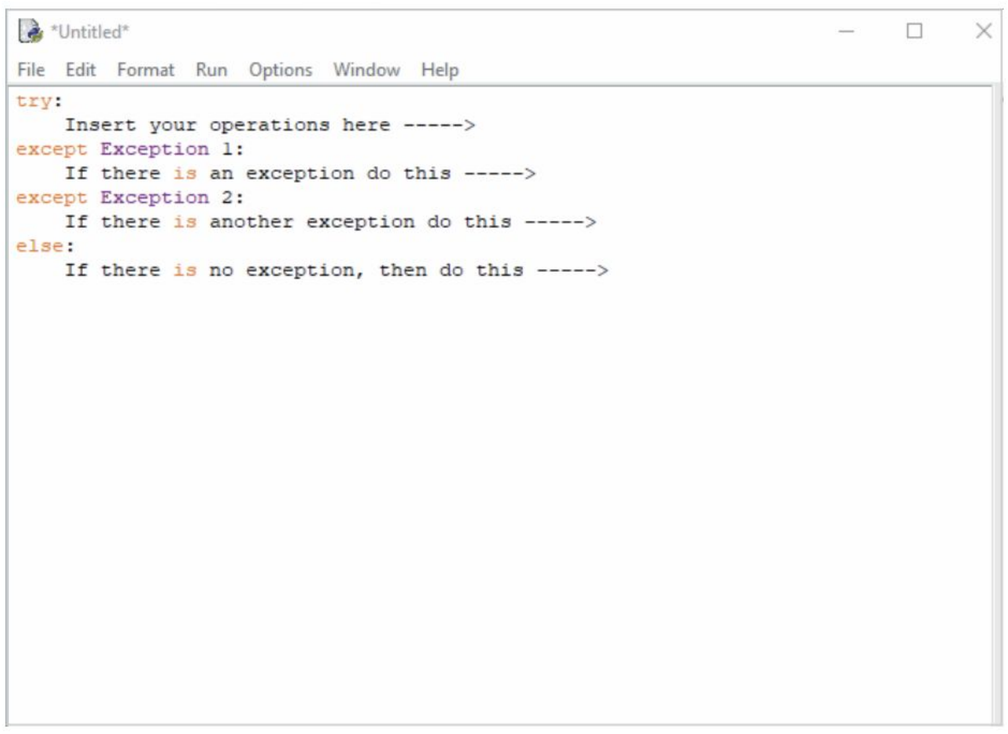
```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- 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 We 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, we can create a custom exception:

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

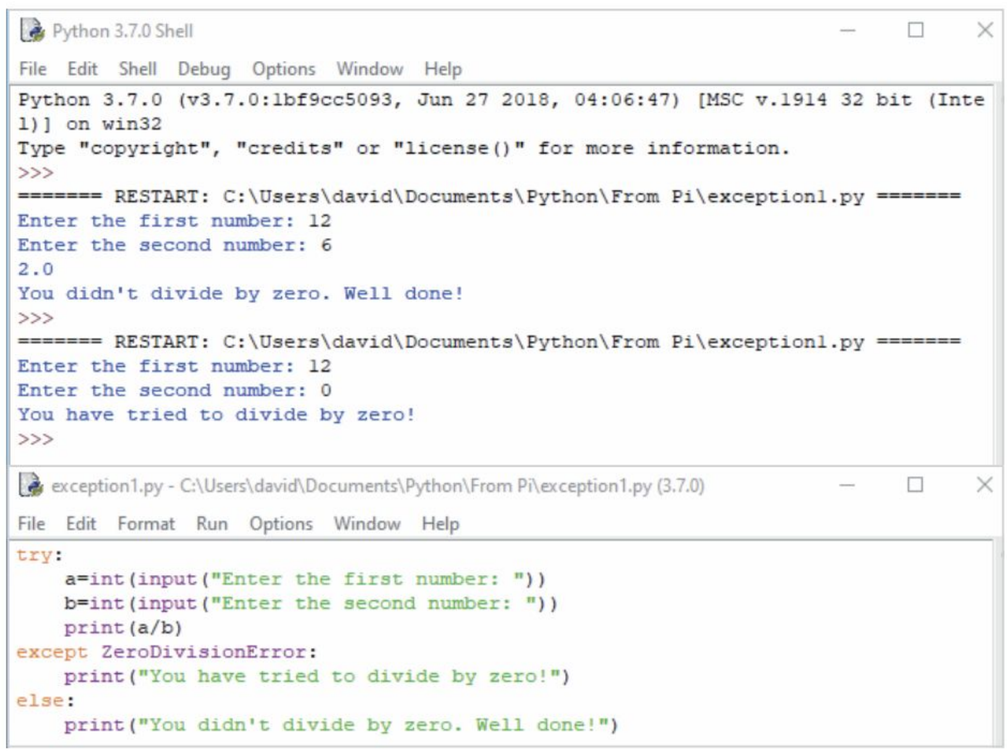


STEP 4 To trap any errors in the code we can encase the potential error within a **try**: block. This block consists of: try, except, else, where the code is held within try, then if there's an exception do something, or do something else.



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

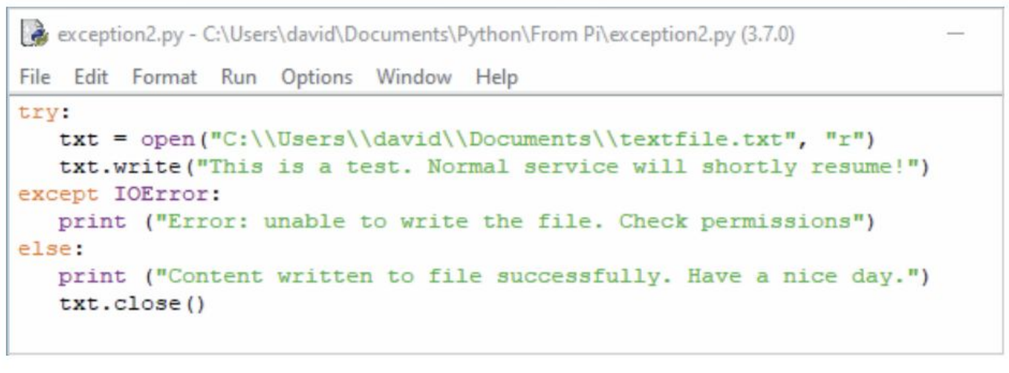
```
try:
 a=int(input("Enter the first number: "))
 b=int(input("Enter the second number: "))
 print(a/b)
except ZeroDivisionError:
 print("You have tried to divide by zero!")
else:
 print("You didn't divide by zero. Well done!")
```



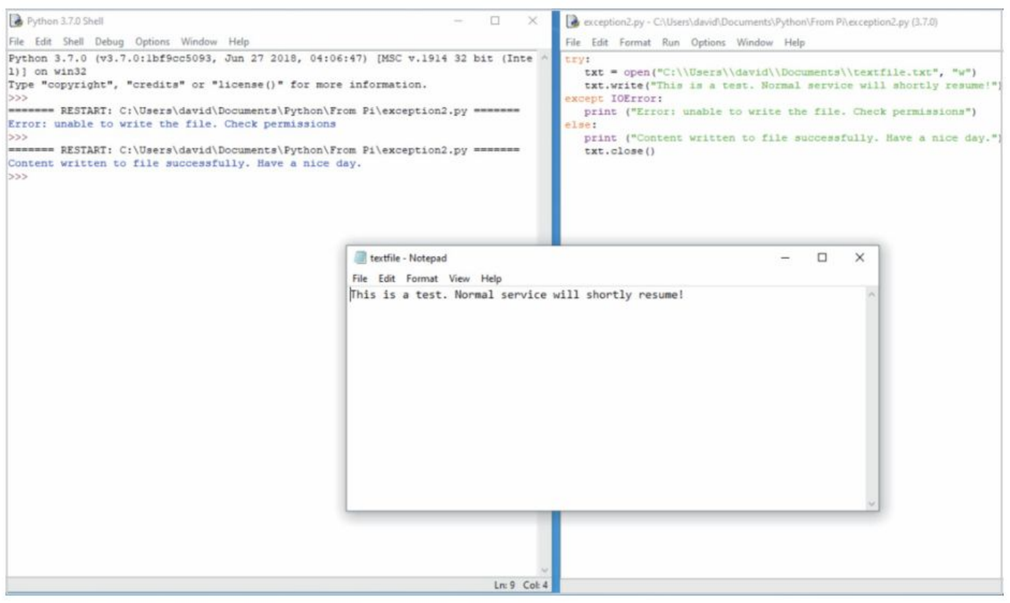
STEP 6 You can use exceptions to handle a variety of useful tasks. Using an example from our previous tutorials, let's assume you want to open a file and write to it:

```
try:
 txt = open("C:\\Users\\david\\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 us we're doing something wrong we've created an exception, using the **IOError** class, informing the user that the permissions are incorrect.



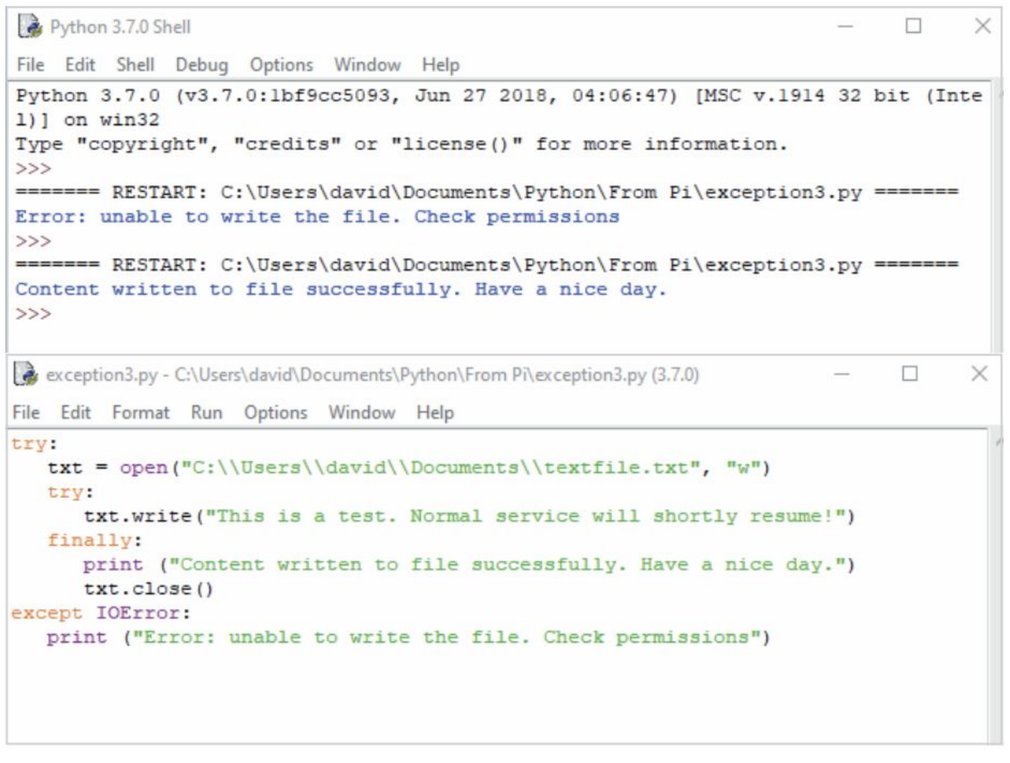
STEP 8 Naturally, we can quickly fix the issue by changing the "r" read only instance with a "w" for write. This, as you already know, will create the file and write the content then commit the changes to the file. The end result will report a different set of circumstances, in this case, a successful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



STEP 9 You can also use a **finally:** block, which works in a similar fashion, but you cannot use else with it. Hint: You'll need to delete the textfile.txt file from your folder.

```
try:
 txt = open C:\\Users\\david\\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 we've used the "r" read-only permission. If we 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.



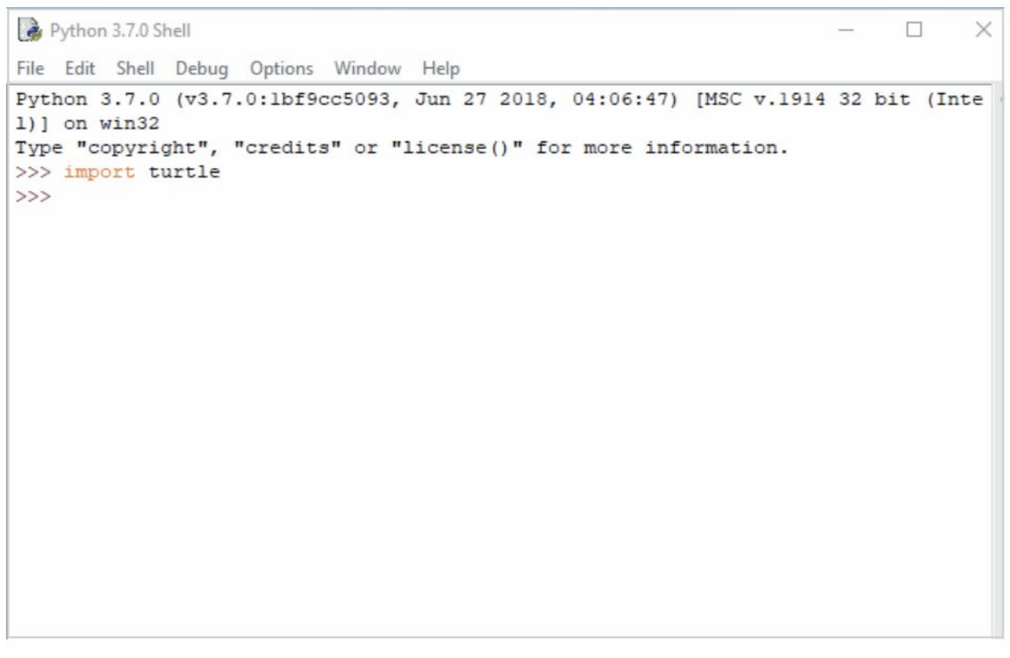
Python Graphics

While dealing with text on the screen, either as a game or in a program, is perfectly fine, there comes 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.

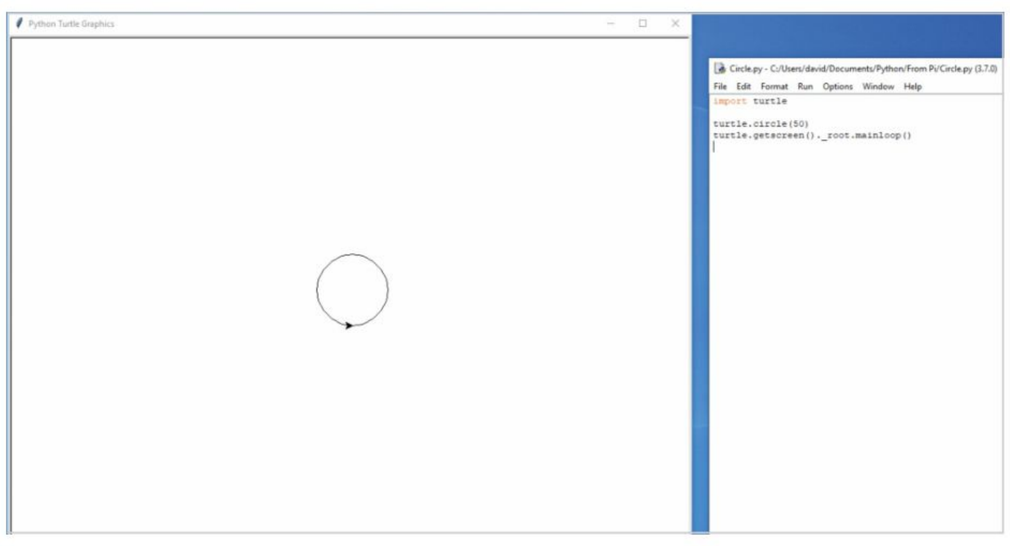
STEP 1 One of the best graphical modules to begin learning Python graphics is Turtle. The Turtle module is, as the name suggests, based on the turtle robots used in many schools that can be programmed to draw something on a large piece of paper on the floor. The Turtle module can be imported with: **import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



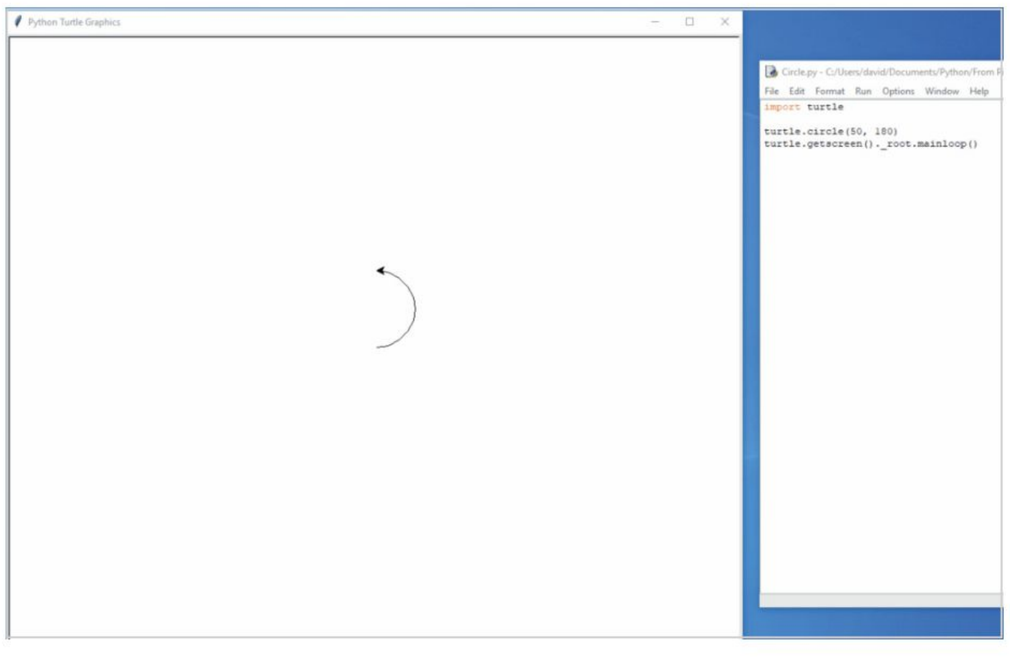
STEP 2 Let's begin by drawing a simple circle. Start a New File, then enter the following code:

```
import turtle

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

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



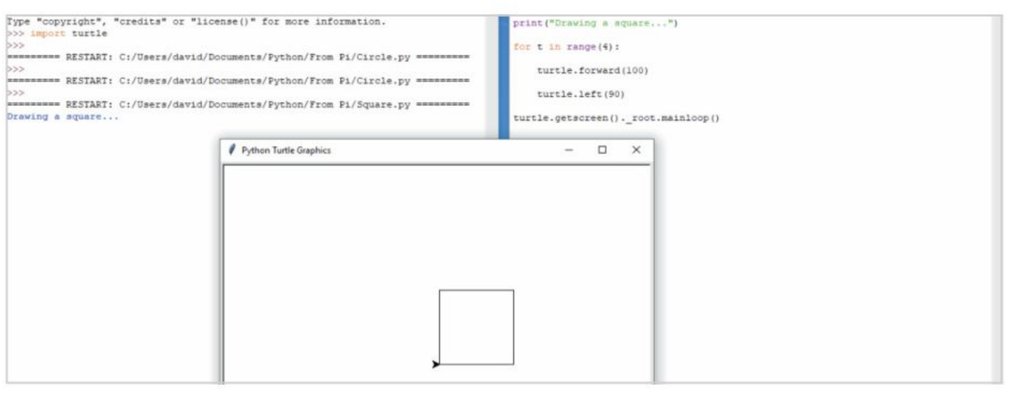
STEP 3 The command turtle.circle(50) is what draws the circle on the screen, with 50 being the size. You can play around with the sizes if you like, going up to 100, 150, and beyond; you can draw an arc by entering `turtle.circle(50, 180)`, where the size is 50, but you're telling Python to only draw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



STEP 5

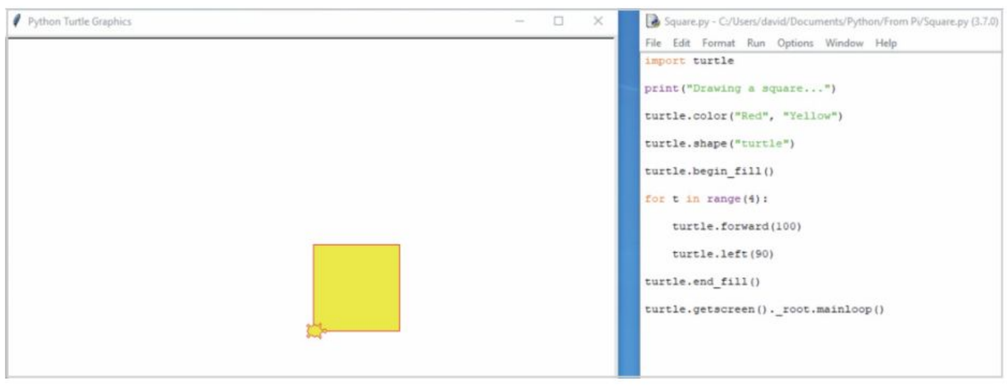
To add some colour, we can add a new line to the square code:

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

And we can even change the character to an actual turtle by entering:

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

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

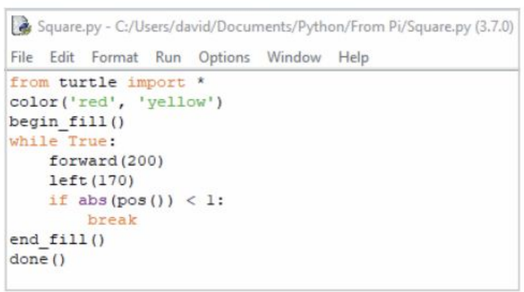


STEP 6

As you can see, 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:

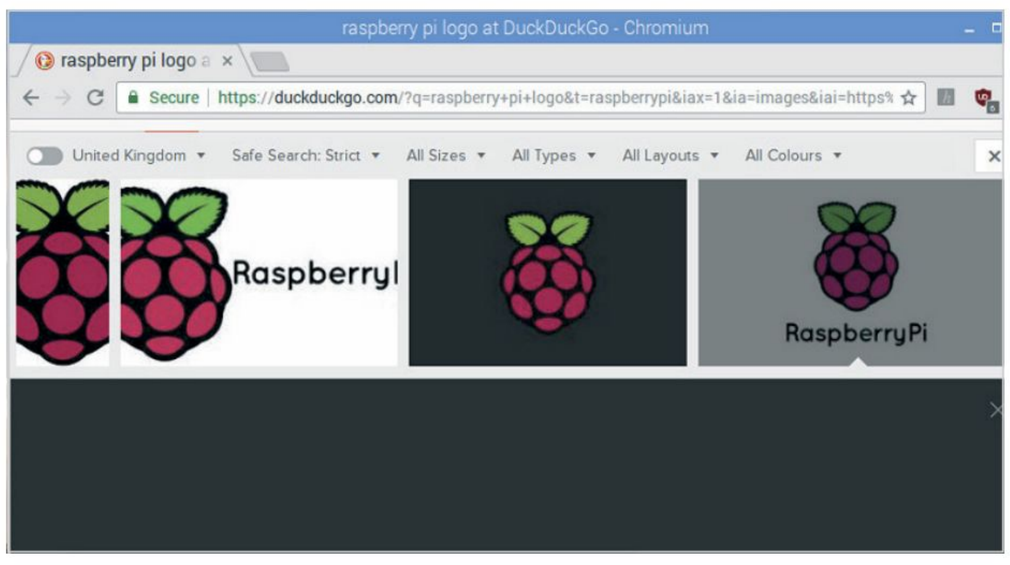
```
from turtle import *
color('red', 'yellow')
begin_fill()
while True:
  forward(200)
  left(170)
  if abs(pos()) < 1:
    break
end_fill()
done()
```

It's a different method, but very effective.



STEP 7

Another way 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 pre-defined image. Start by opening a browser and finding an image, then save it to the folder where you save your Python code.



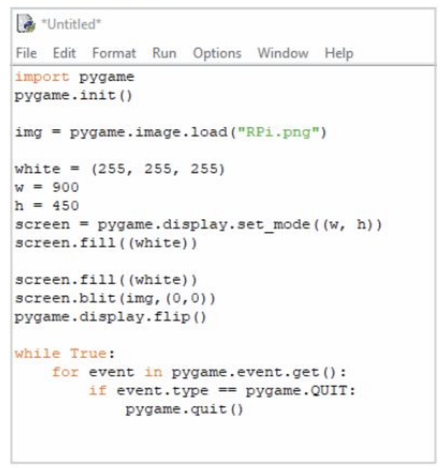
STEP 8

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()
```



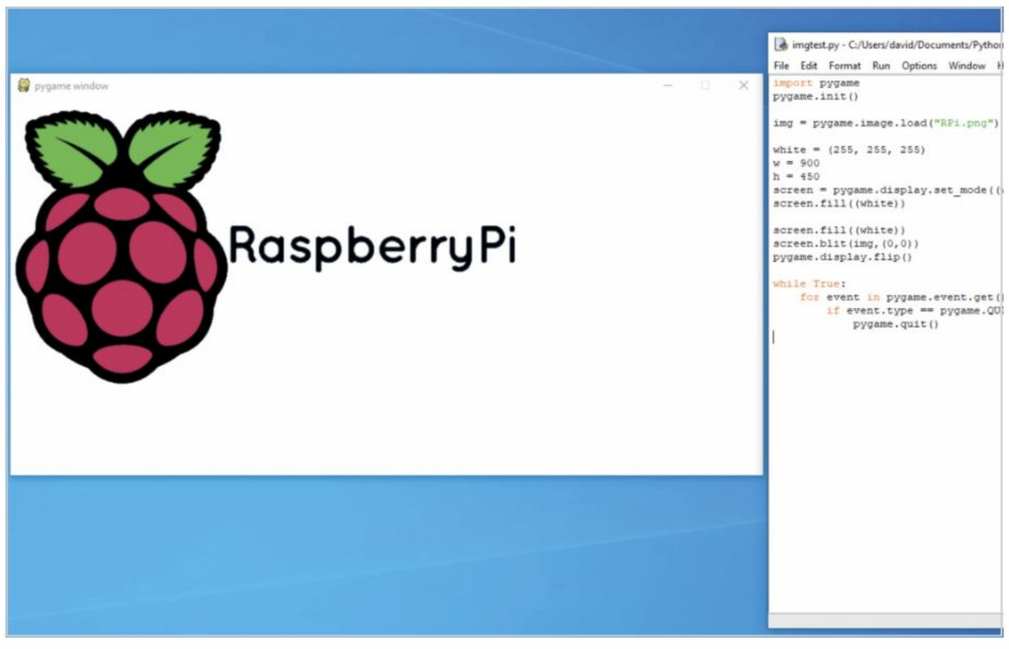
STEP 9

In the previous step we've imported pygame, initiated the pygame engine, and asked it to import our saved Raspberry Pi logo image, saved as RPi.png. Next, we defined the background colour of the window to display the image, and the window size as per the actual image dimensions. Finally, we 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.



Combining What You Know So Far

Based on what we've looked at over this section, let's combine it all and come up with a piece of code that can easily be applied into a real-world situation; or at the very least, something which you can incorporate into your programs.

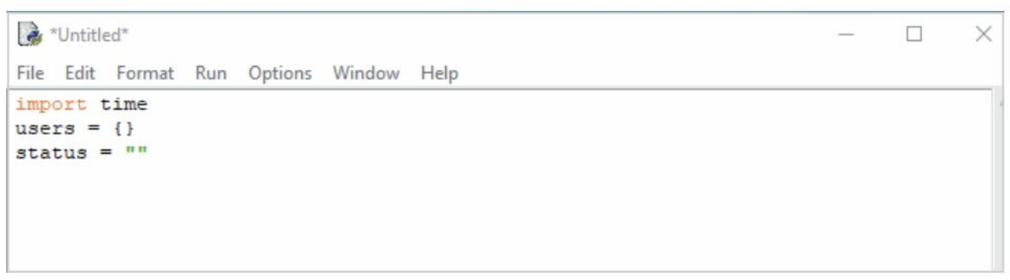
LOGGING IN

For this example, let's look to a piece of code that will create user logins then allow them to log into the system, and write the time at which they logged in. We can even include an option to quit the program by pressing 'q'.

STEP 1

Let's begin by importing the Time module, creating a new dictionary to handle the usernames and passwords, and creating a variable to evaluate the current status of the program:

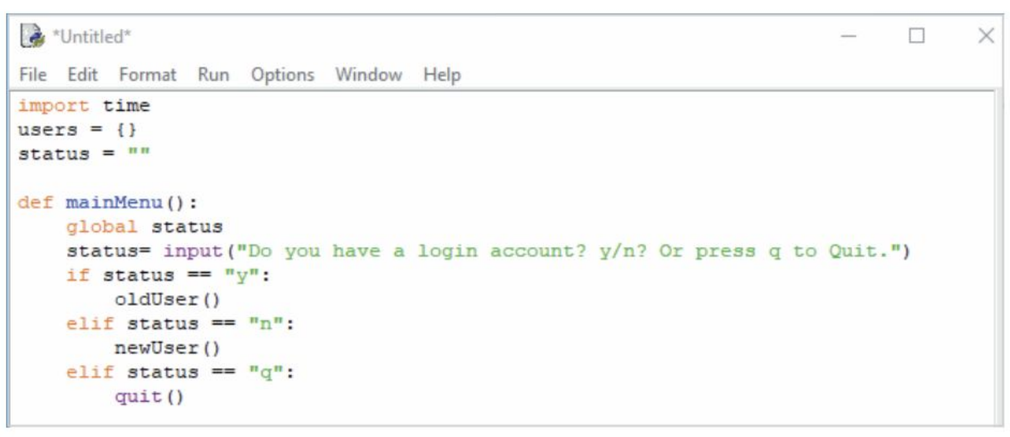
```
import time
users = {}
status = ""
```



STEP 2

Next we need to define some functions. We can begin with creating the main menu, to where, after selecting the available options, all users will return:

```
def mainMenu():
  global status
  status = input("Do you have a login account?
y/n? Or press q to quit.")
  if status == "y":
    oldUser()
  elif status == "n":
    newUser()
  elif status == "q":
    quit()
```



STEP 3

The **global status** statement separates a local variable from one that can be called throughout the code, this way we can use the q=quit element without it being changed inside the function. We've also referenced some newly defined functions: oldUser and newUser which we'll get to next.

```
def mainMenu():
    global status
    status = input("Do you have a login account? y/n? Or press q to quit.")
    if status == "y":
        oldUser()
    elif status == "n":
        newUser()
    elif status == "q":
        quit()
```

STEP 4

The newUser function is next:

```
def newUser():
  createLogin = input("Create a login name: ")

  if createLogin in users:
    print ("\nLogin name already exists!\n")
  else:
    createPassw = input("Create password: ")
    users[createLogin] = createPassw
    print("\nUser created!\n")
    logins=open("C:\\Users\\david\\Documents\\
logins.txt", "a")
    logins.write("\n" + createLogin + " " +
createPassw)
    logins.close()
```

This creates a new user and password, and writes the entries into a file called **logins.txt**.

```
def newUser():
    createLogin = input("Create a login name: ")

    if createLogin in users:
        print ("\nLogin name already exists!\n")
    else:
        createPassw = input("Create password: ")
        users[createLogin] = createPassw
        print("\nUser created!\n")
        logins=open("C:\\Users\\david\\Documents\\logins.txt", "a")
        logins.write("\n" + createLogin + " " + createPassw)
        logins.close()
```

STEP 5 Since we're using a Raspberry Pi, you will need to specify your own location for the logins.txt file. Essentially, this adds the username and password inputs, from the user, to the existing **users{}** dictionary. Therefore, the key and value structure remains; each user is the key, the password is the value.

```
def newUser():
    createLogin = input("Create a login name: ")

    if createLogin in users:
        print ("\nLogin name already exists!\n")
    else:
        createPassw = input("Create password: ")
        users[createLogin] = createPassw
        print("\nUser created!\n")
        logins=open("/home/pi/Documents/logins.txt", "a")
        logins.write("\n" + createLogin + " " + createPassw)
        logins.close()
```

STEP 6 Now to create the oldUser function:

```
def oldUser():
  login = input("Enter login name: ")
  passw = input("Enter password: ")

  # check if user exists and login matches password
  if login in users and users[login] == passw:
    print ("\nLogin successful!\n")
    print ("User:", login, "accessed the system on:", time.asctime())
  else:
    print ("\nUser doesn't exist or wrong password!\n")
```

```
    elif status == "n":
        newUser()
    elif status == "q":
        quit()

def newUser():
    createLogin = input("Create a login name: ")

    if createLogin in users:
        print ("\nLogin name already exists!\n")
    else:
        createPassw = input("Create password: ")
        users[createLogin] = createPassw
        print("\nUser created!\n")
        logins=open("C:\\Users\\david\\Documents\\logins.txt", "a")
        logins.write("\n" + createLogin + " " + createPassw)
        logins.close()

def oldUser():
    login = input("Enter login name: ")
    passw = input("Enter password: ")

    # check if user exists and login matches password
    if login in users and users[login] == passw:
        print("\n Login successful!\n")
        print("User:", login, "accessed the system on:", time.asctime())
    else:
        print("\nUser doesn't exist or wrong password!\n")
```

STEP 7 There's a fair bit happening here. We have login and passw variables, which are then matched to the users dictionary. If there's a match, then we have a successful login and the time and date of the login is outputted. If they don't match, then we print an error, and the process starts again.

```
def oldUser():
    login = input("Enter login name: ")
    passw = input("Enter password: ")

    # check if user exists and login matches password
    if login in users and users[login] == passw:
        print ("\nLogin successful!\n")
        print ("User:", login, "accessed the system on:", time.asctime())
    else:
        print ("\nUser doesn't exist or wrong password!\n")
```

STEP 8 Finally, we need to continually check that the 'q' key hasn't been pressed to exit the program. We can do this with:

```
while status != "q":
  status = displayMenu()
```

login.py - C:/Users/david/Documents/Python/login.py (3.7.0)

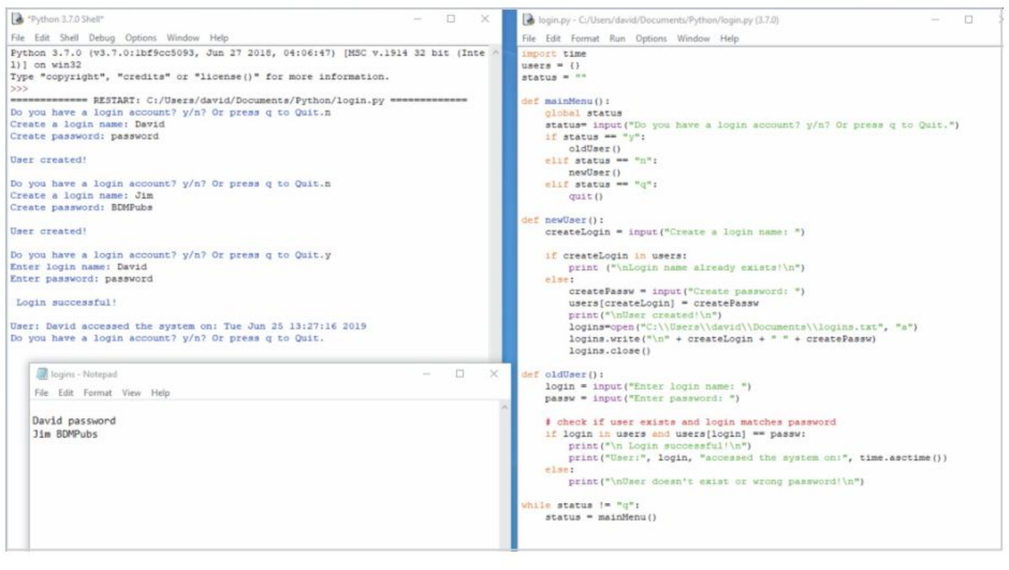
File Edit Format Run Options Window Help

```
import time
users = {}
status = ""

def mainMenu():
    global status
    status= input("Do you have a login account? y/n? Or press q to Quit.")
    if status == "y":
        oldUser()
    elif status == "n":
        newUser()
    elif status == "q":
        quit()

def newUser():
    createLogin = input("Create a login name: ")

    if createLogin in users:
        print ("\nLogin name already exists!\n")
    else:
        createPassw = input("Create password: ")
        users[createLogin] = createPassw
        print("\nUser created!\n")
        logins=open("C:\\Users\\david\\Documents\\logins.txt", "a")
        logins.write("\n" + createLogin + " " + createPassw)
        logins.close()

def oldUser():
    login = input("Enter login name: ")
    passw = input("Enter password: ")

    # check if user exists and login matches password
    if login in users and users[login] == passw:
        print("\n Login successful!\n")
        print("User:", login, "accessed the system on:", time.asctime())
    else:
        print("\nUser doesn't exist or wrong password!\n")
```

STEP 9 Although a seemingly minor two lines, the **while** loop is what keeps the program running. At the end of every function, it's checked against the current value of status. If that global value isn't 'q' then the program continues. If it's equal to 'q' then the program can quit.

```
while status != "q":
    status = mainMenu()
```

STEP 10 You can now create users, and then login with their names and passwords, with the logins.txt file being created to store the login data and successful logins being time-stamped. Now it's up to you to further improve the code. Perhaps you can import the list of created users from a previous session and, upon a successful login, display a graphic?



Working with Modules

Modules are where you can take Python programming to new heights; they are the key ingredients to better code. You can create your own modules, or you can use some of those already available, to help convert a mundane piece of code into something spectacular.

Want to see how to make better use of these modules to add a little something extra to your code? Then read on and learn how they can be used to forge fantastic programs.

90 Calendar Module

92 OS Module

94 Using the Math Module

96 Random Module

98 Tkinter Module

100 Pygame Module

104 Basic Animation

106 Create Your Own Modules

Calendar Module

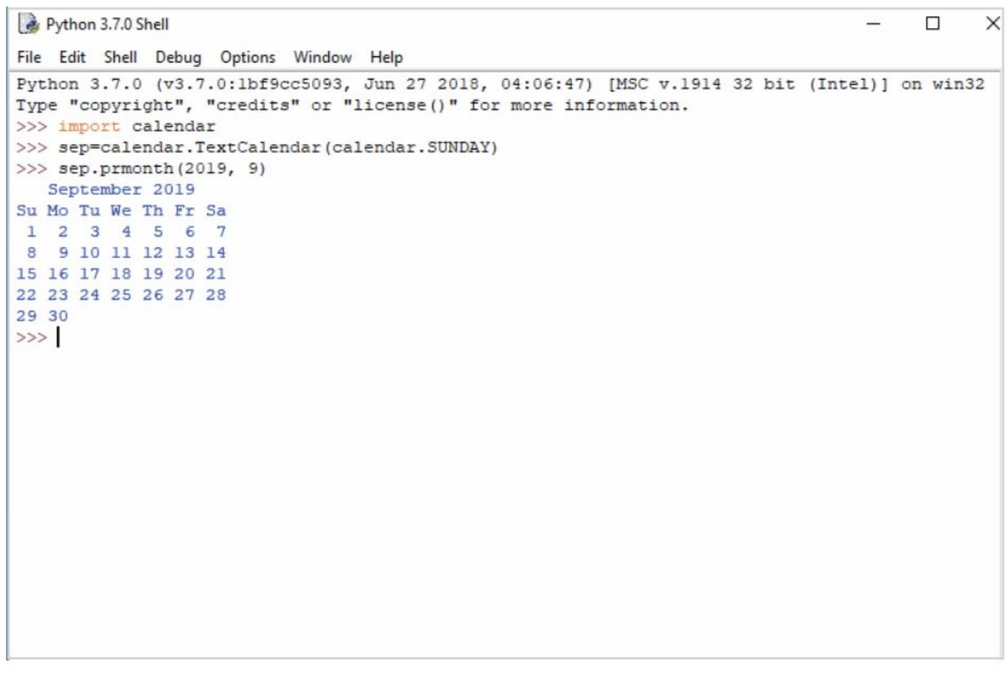
Beyond the Time module, the Calendar module can produce some interesting results when executed within your code. It does far more than simply display the date in the time module-like format, you can actually call up a wall-calendar type display.

WORKING WITH DATES

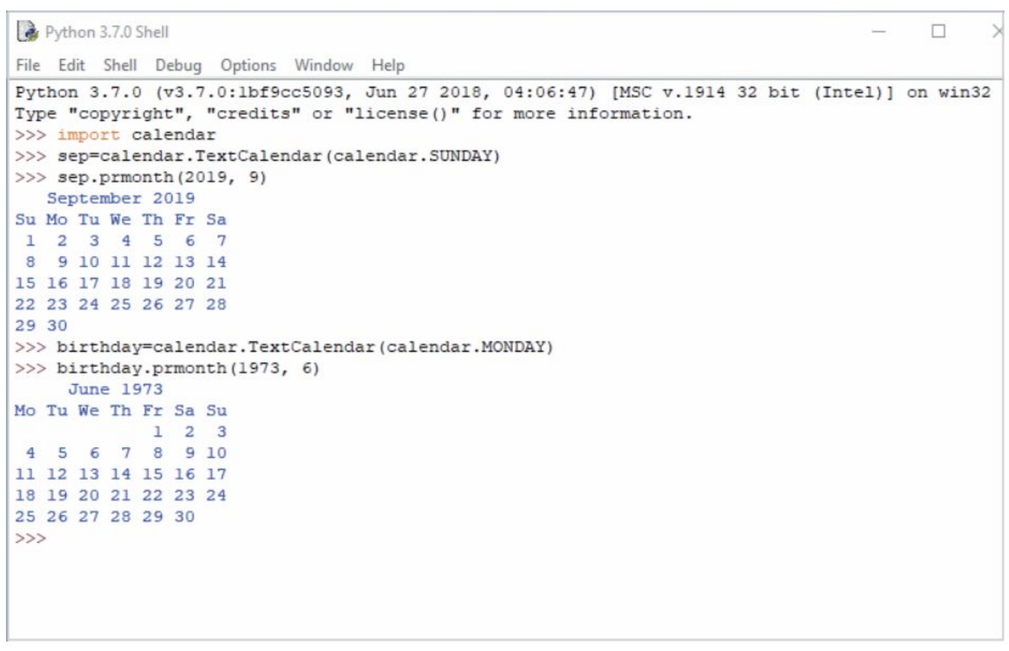
The Calendar module is built into Python 3. However, if, it's not installed, you can add it using pip install calendar, as a Windows administrator, or sudo pip install calendar for Linux and macOS.

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

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



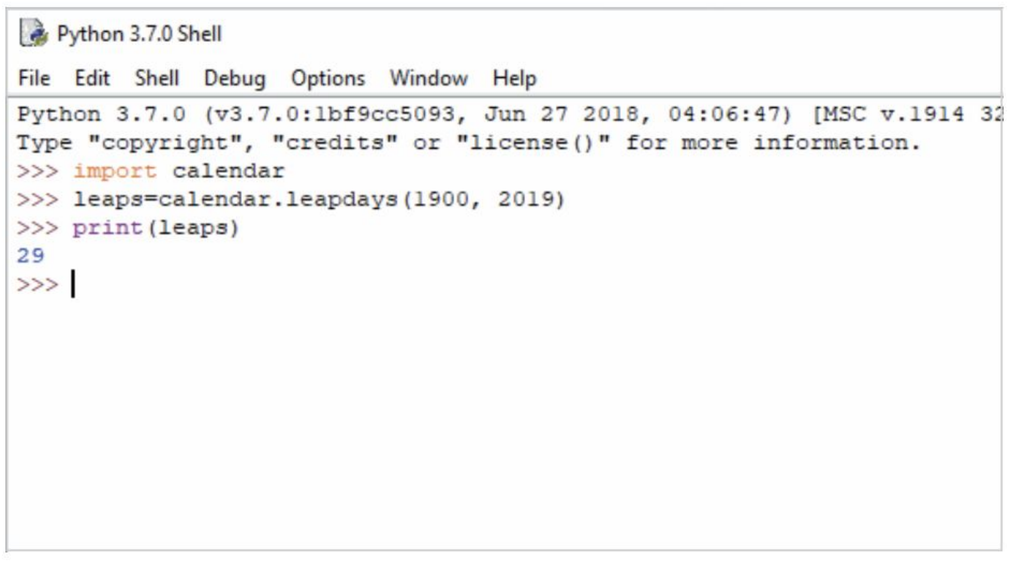
STEP 2 You can see that the days of September 2019 are displayed in a wall calendar fashion. Naturally you can change the 2019, 9 part of the second line, to any year and month you want, a birthday for example (1973, 6). The first line configures TextCalendar to start its weeks on a Sunday; you can opt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

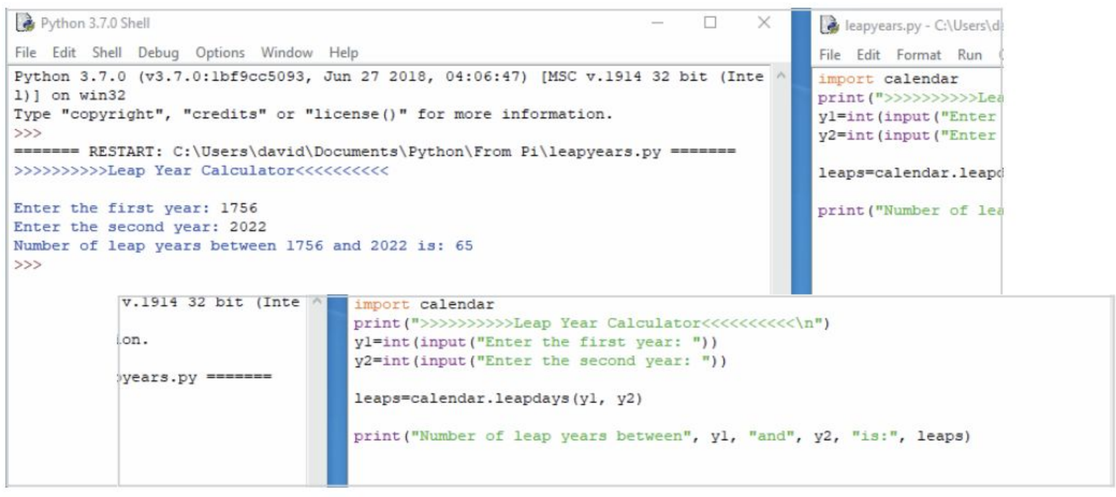
```
leaps=calendar.leapdays(1900, 2019)
print(leaps)
```

The result is 29, starting from 1904 onward.



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)
```

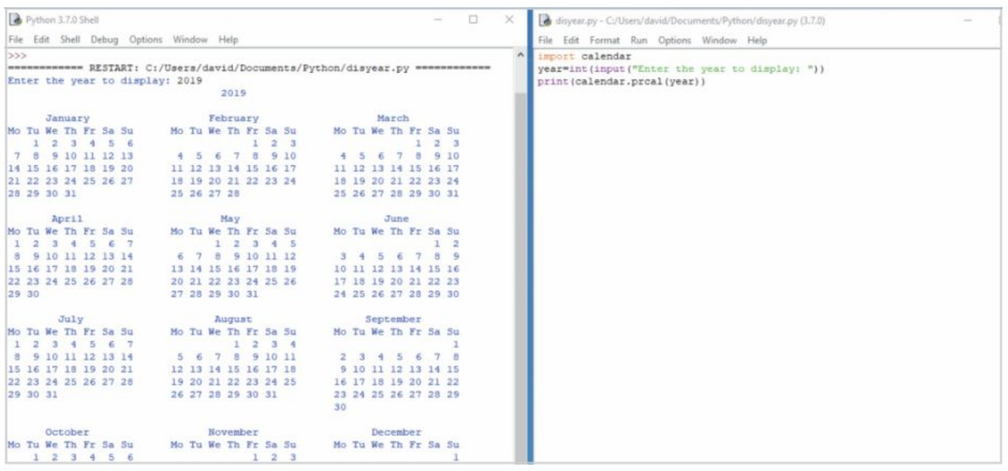


STEP 5

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))
```

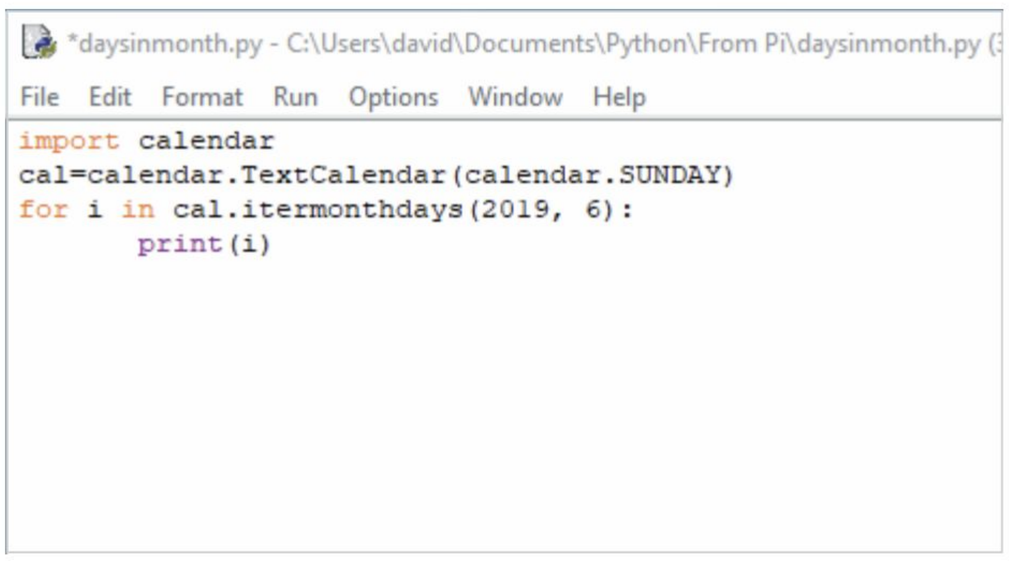
We're sure you'll agree that's quite a handy bit of code to have to hand.



STEP 6

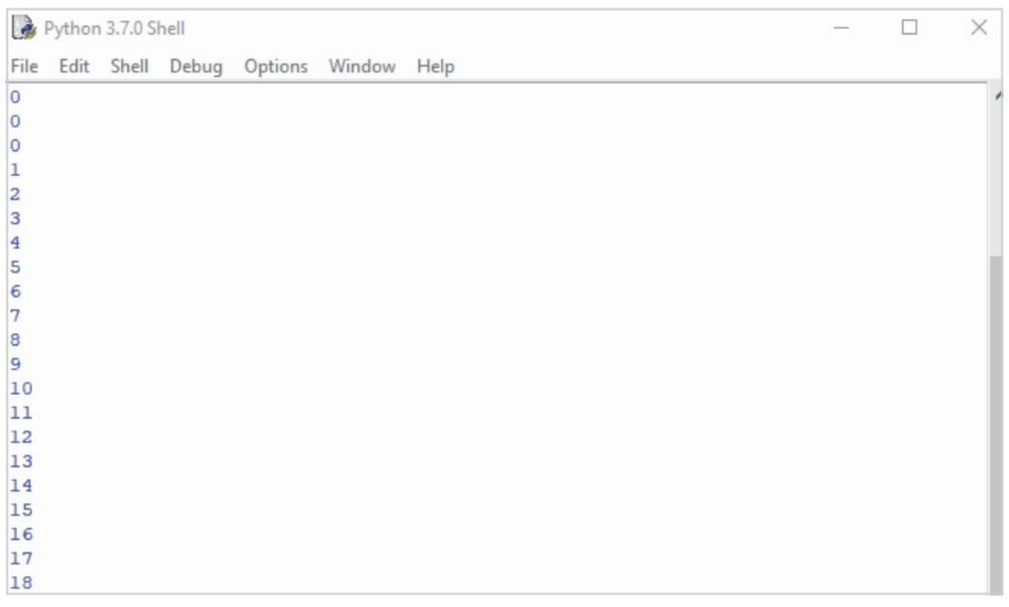
Interestingly we can also list the number of days in a month by using a simple for loop:

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



STEP 7

You can see that code produced some zeros at the beginning, this is due to the starting day of the week, Sunday in this case, and overlapping days from the previous month. So, 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)
```

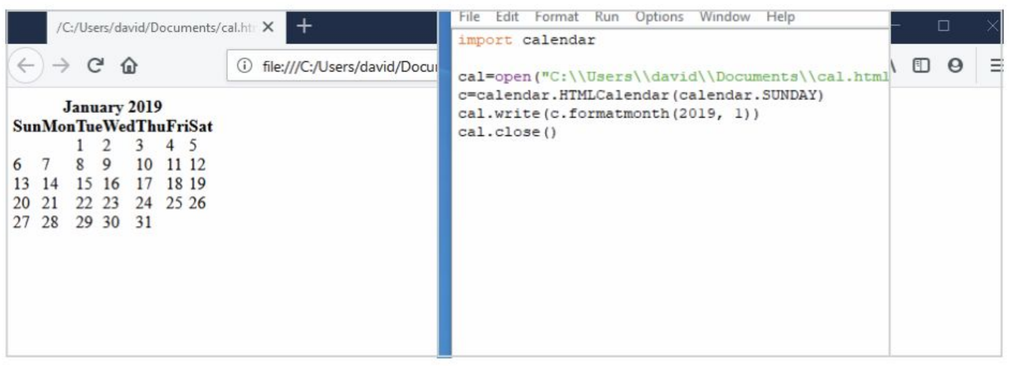
```
import calendar
for name in calendar.day_name:
   print(name)
```

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("C:\\Users\\david\\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.

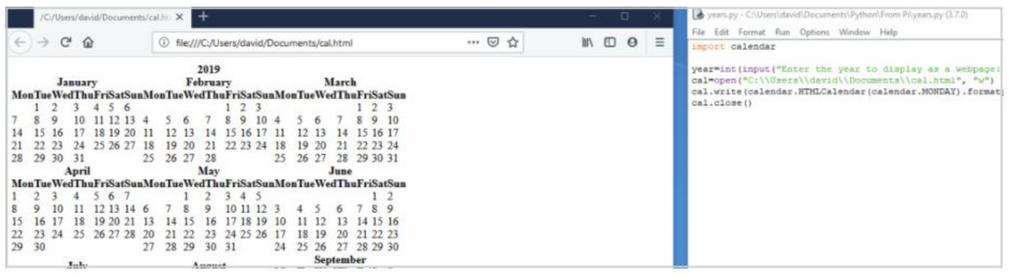


STEP 10

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("C:\\Users\\david\\Documents\\cal.html", "w")
cal.write(calendar.HTMLCalendar(calendar.MONDAY).
formatyear(year))
cal.close()
```

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



OS Module

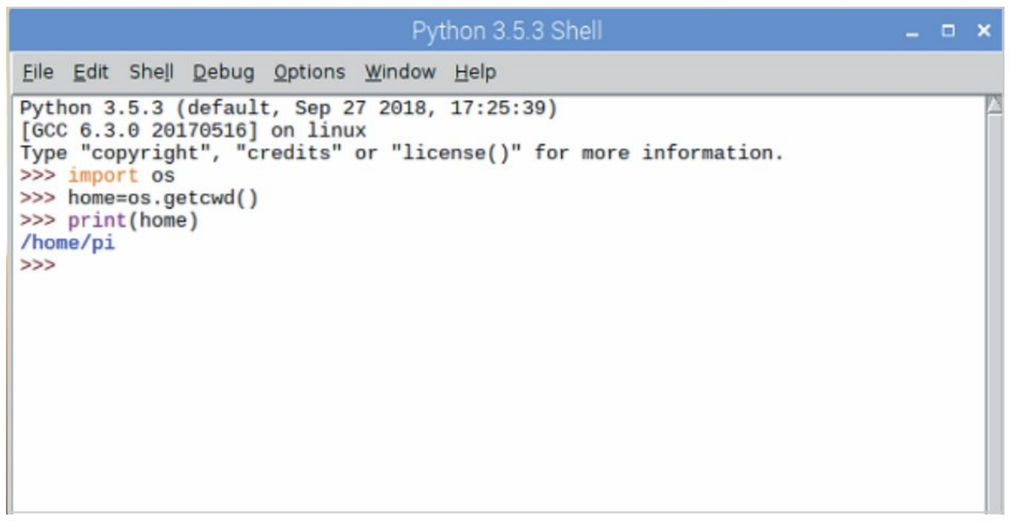
The OS module allows you to interact directly with the built-in commands found in your operating system. The commands can vary, depending on the OS on which you're running the module, as some will work with Windows whereas others will work with Linux and macOS.

INTO THE SYSTEM

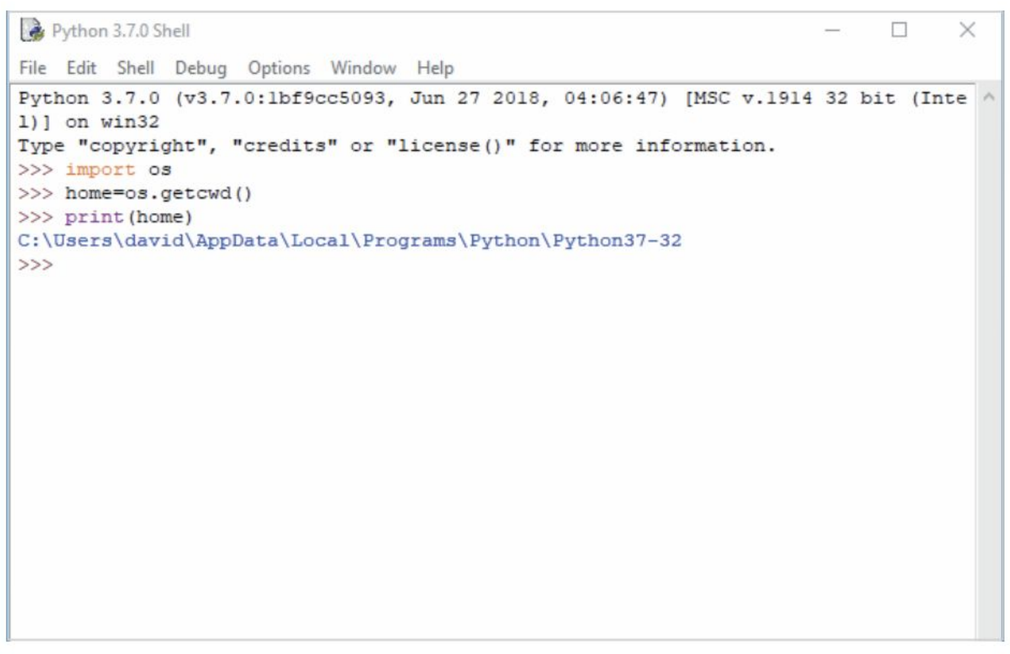
One of the primary features of the OS module is the ability to list, move, create, delete and otherwise interact with files stored on the system; making it the perfect module for backup code.

STEP 1 We 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:

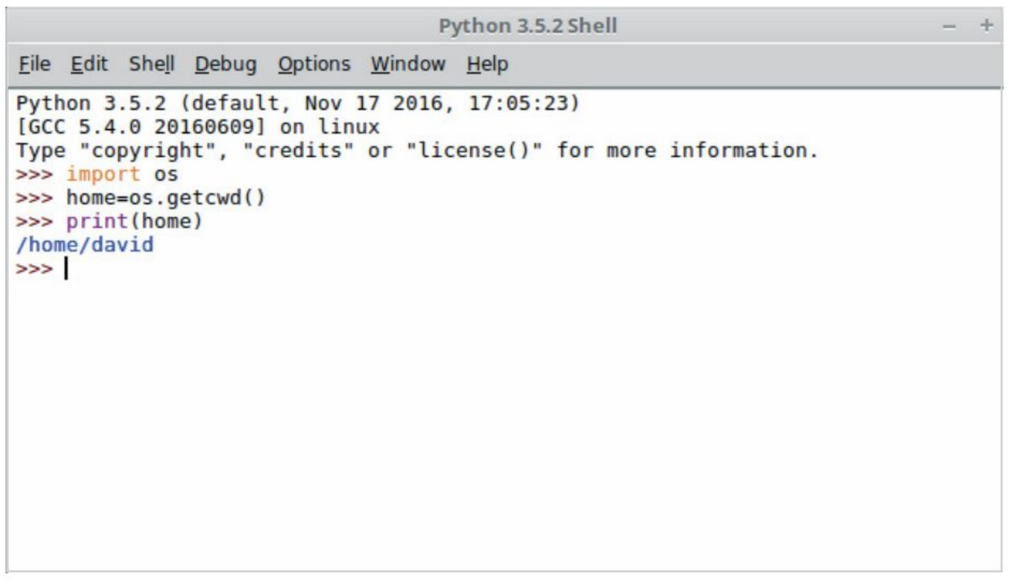
```
import os
home=os.getcwd()
print(home)
```



STEP 2 The returned result, from printing the variable home, is the current user's home folder on the system. In Step 1 that's **/home/pi**, it will be different depending on the user name you login as, and the operating system you use. For example, Windows 10 would output: **C:\Users\david\AppData\Local\Programs\Python\Python37-32**.

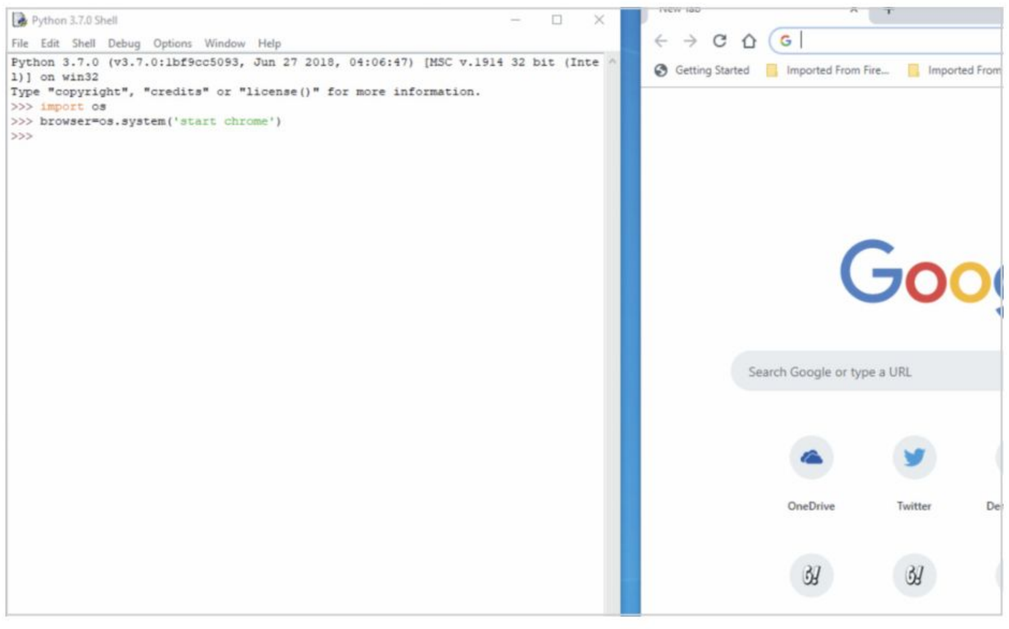


STEP 3 The Windows output is different as that's the current working directory of Python, as determined by the system. As you suspect, the **os.getcwd()** function is asking Python to retrieve the Current Working Directory. Linux users will see something along the same lines as the Raspberry Pi, as will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



STEP 4 Another interesting element to the OS module is its ability to launch programs that are installed in the host system. For instance, if we wanted to launch the Chromium Web Browser from within a Python program we can use the command:

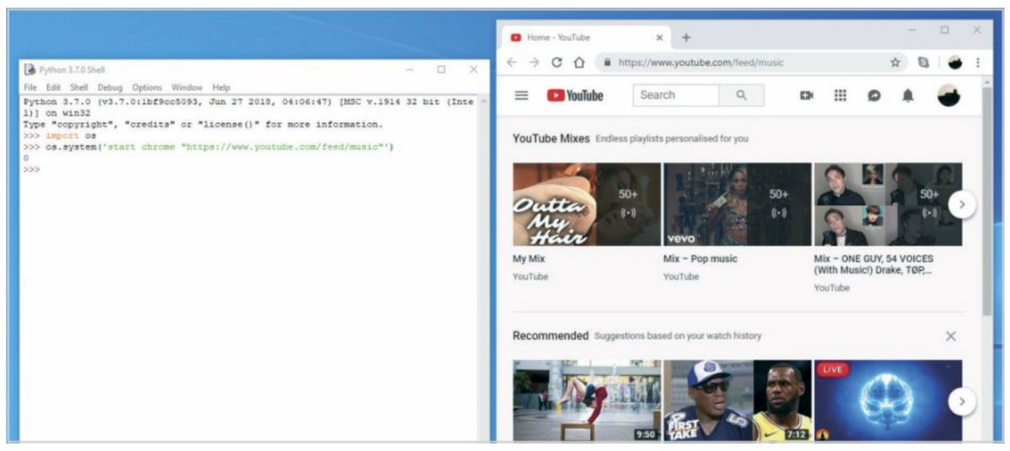
```
import os
browser=os.system('start chrome')
```



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 also use the following:

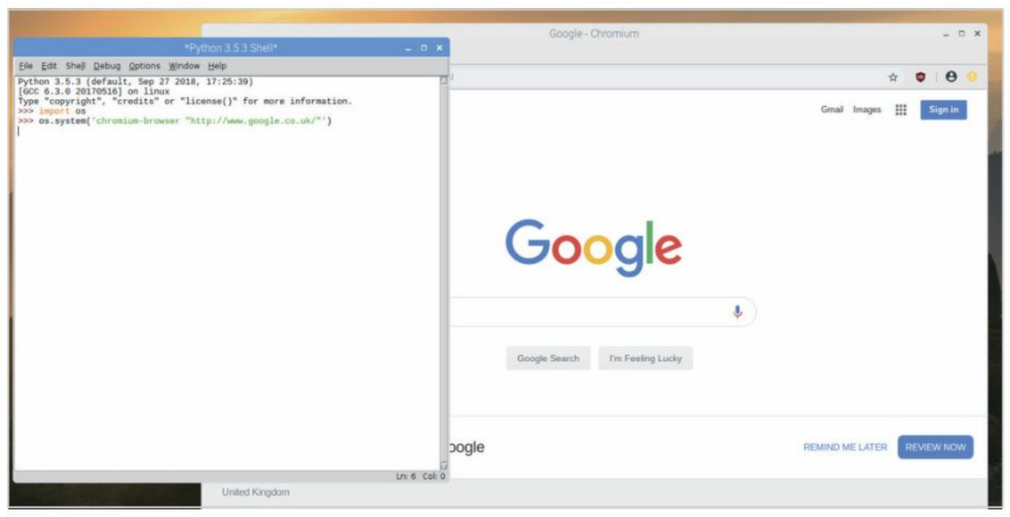
```
import os
os.system(‘‘start chrome “https://www.youtube.com/feed/music”’)
```



STEP 6

To show that the OS module works roughly the same across all platforms, we need to specify the name of the app according to the OS on which you’re working. For example, on a Raspberry Pi you would enter:

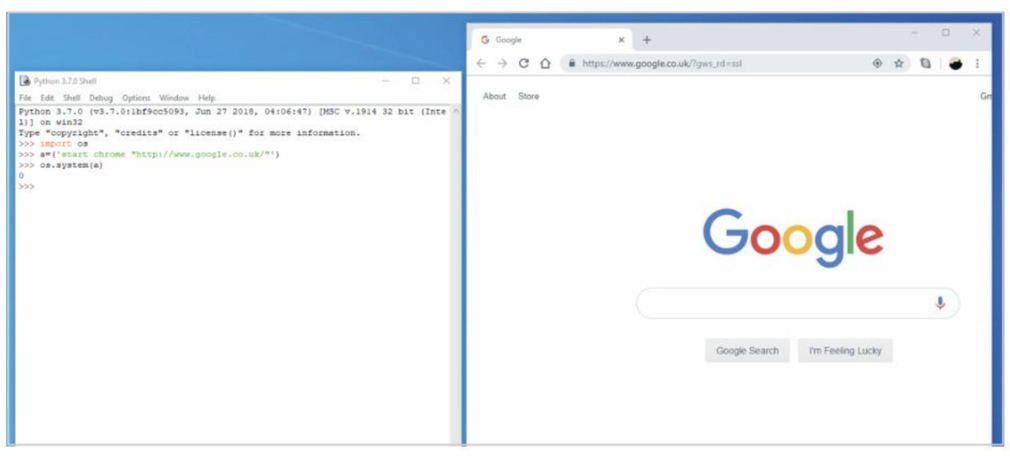
```
import os
os.system(‘chromium-browser “http://bdmpublications.com/”’)
```



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 open webpages:

```
import os
a=(‘start chrome “http://www.google.co.uk”’)
os.system(a)
```

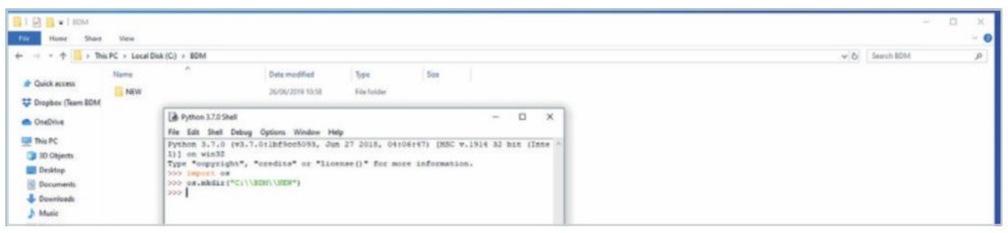


STEP 8

Being able 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(“C:\\BDM\\NEW”)
```

This creates a new directory within the specified Directory (C:\BDM\), named according to the object in the mkdir function (C:\BDM\NEW).



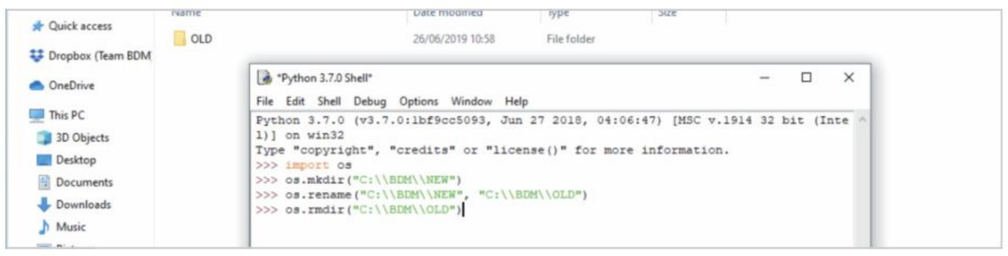
STEP 9

You can also rename any directories you’ve created by entering:

```
import os
os.rename(“C:\\BDM\\NEW”, “C:\\BDM\\OLD”)
```

Or delete them…

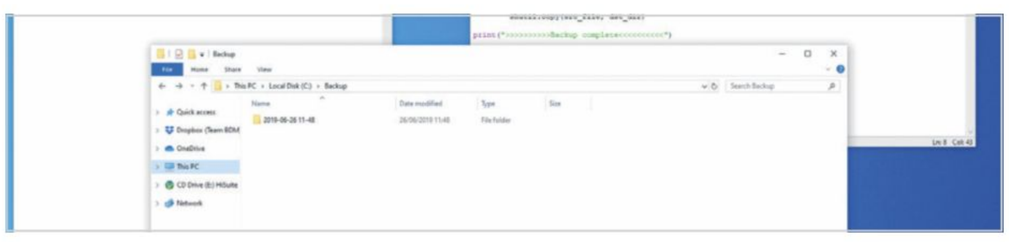
```
import os
os.rmdir(“C:\\BDM\\OLD”)
```



STEP 10

Another module that goes together with OS is shutil. We 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

t=str(time.strftime(“%Y-%m-%d %H-%M”))
root_src_dir = r’C:\\BDM\\Documents\\’    #Path to the source directory
root_dst_dir = ‘C:\\Backup\\’ + t #Path to the backup directory

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<<<<<<<<<<”)
```



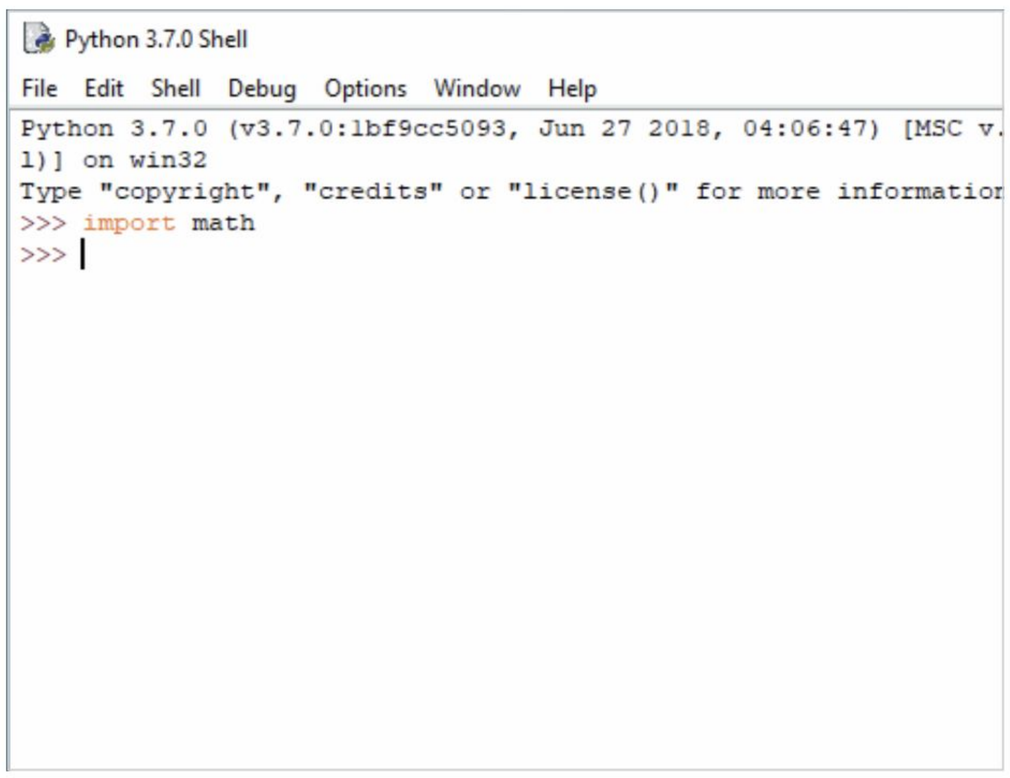
Using the Math Module

One of the more used modules you will come across is the Math module. As we've mentioned previously in this book, Mathematics is the backbone of programming and there's an incredible number of uses the Math module can have in your code.

$E = MC^2$

The Math module provides access to a plethora of Mathematical functions, from simply displaying the value of Pi, to helping you create complex 3D shapes.

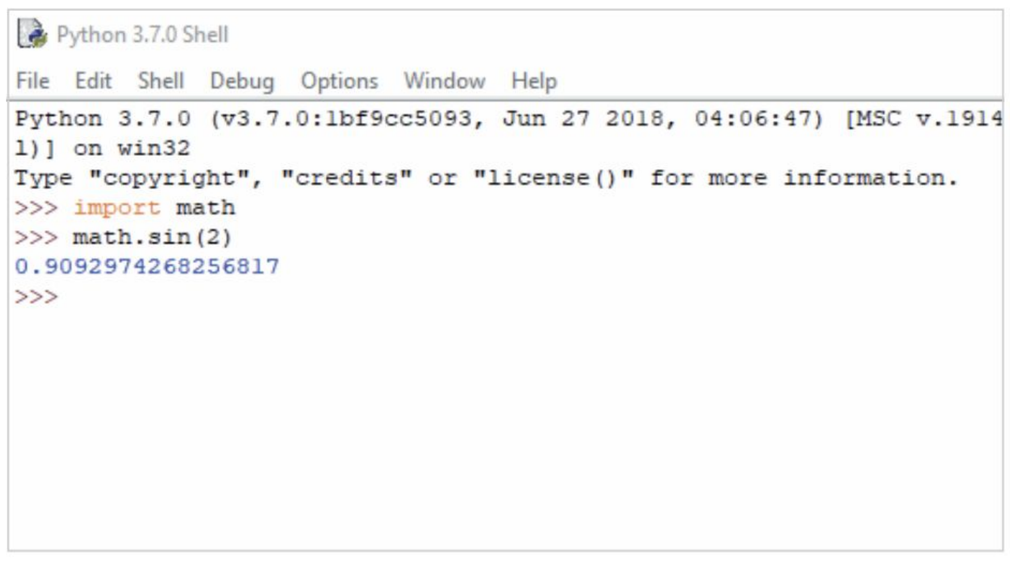
STEP 1 The Math module is built-in to Python 3; so there's no need to PIP install it. As with the other modules present, you can import the module's function by simply entering **import math** into the Shell, or as part of your code in the Editor.



STEP 2 Importing the Math module, as such, will give you access to the module's code. From there, you can call up any of the available functions within Math, by using **math**, followed by the name of the function in question. For example, enter:

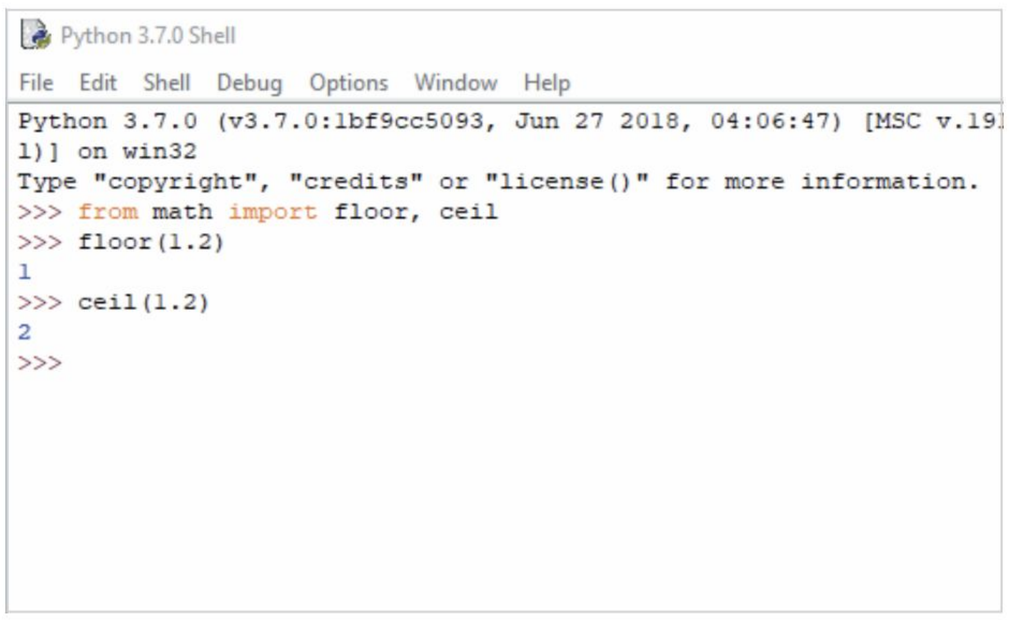
```
math.sin(2)
```

This will display the sine of 2.



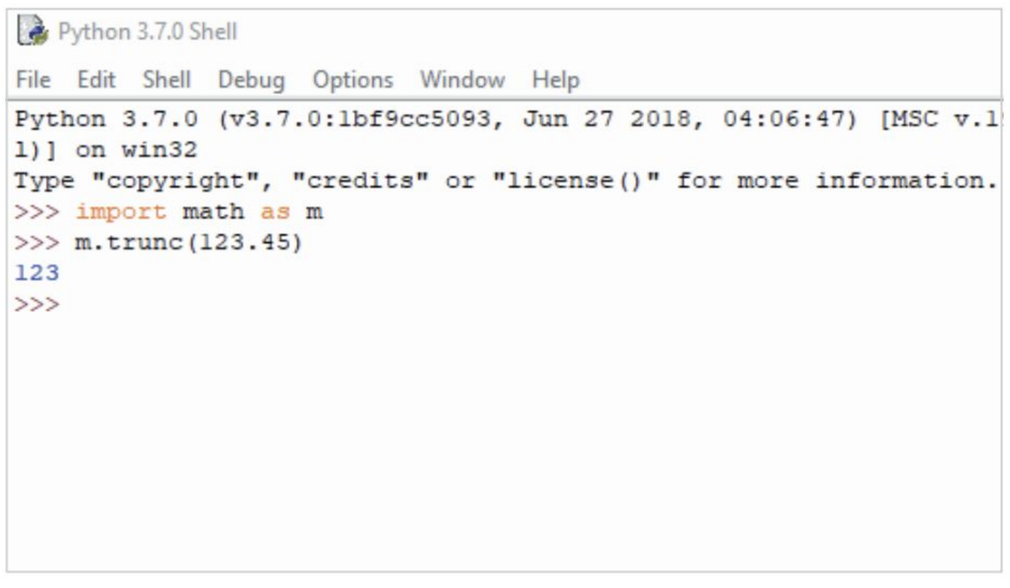
STEP 3 As you will no doubt be aware by now, if you know the name of the individual functions within the module you can specifically import them. For instance, the floor and ceil functions round a float down, or up:

```
from math import floor, ceil
floor(1.2) # returns 1
ceil(1.2) # returns 2
```



STEP 4 The Math module can also be renamed as you import it, as with the other modules on offer within Python. This often saves time, but don't forget to make a comment to show someone else looking at your code what you've done:

```
import math as m
m.trunc(123.45) # Truncate removes the fraction
```



STEP 5

Although it's not common practise, it is possible to import functions from a module and rename them. In this example, we're importing **floor** from Math and renaming it to **f**. This process can quickly become confusing though, where lengthy code is in use:

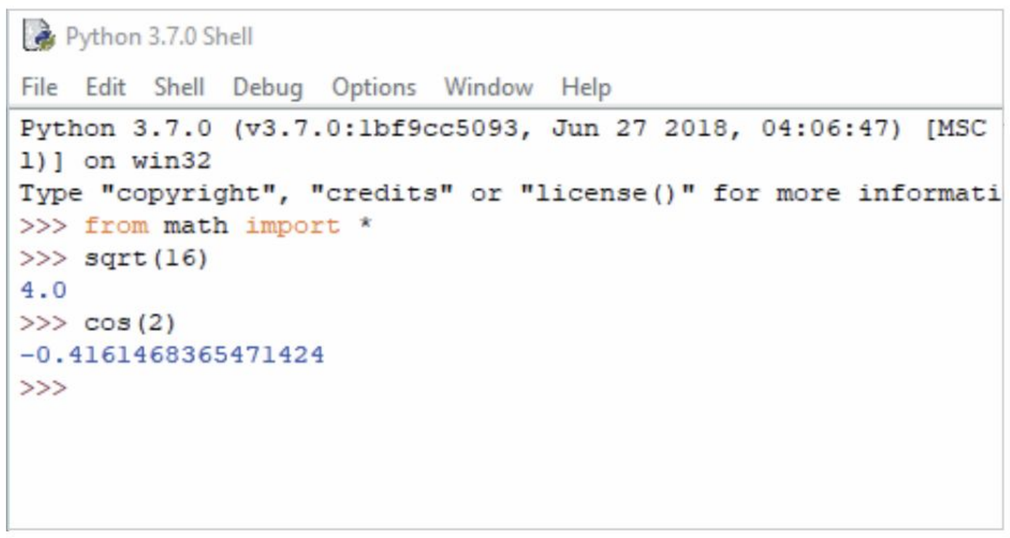
```
from math import floor as f
f(1.2)
```

STEP 6

Importing all the functions of the Math module can be achieved by entering:

```
from math import *
```

While certainly handy, this is often frowned upon by the developer community as it takes up unnecessary resources and isn't an efficient way of coding. However, if it works for you then go ahead.



STEP 7

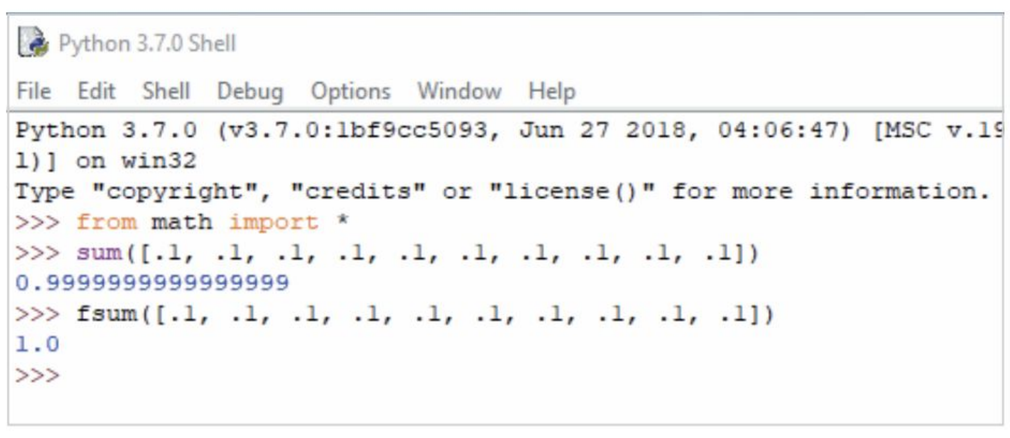
Interestingly, some functions within the Math module are more accurate, or to be more precise, are designed to return a more accurate value, than others. For example:

```
sum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
```

Will return the value of 0.999999999. Whereas:

```
fsum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
```

Returns the value of 1.0.



STEP 8

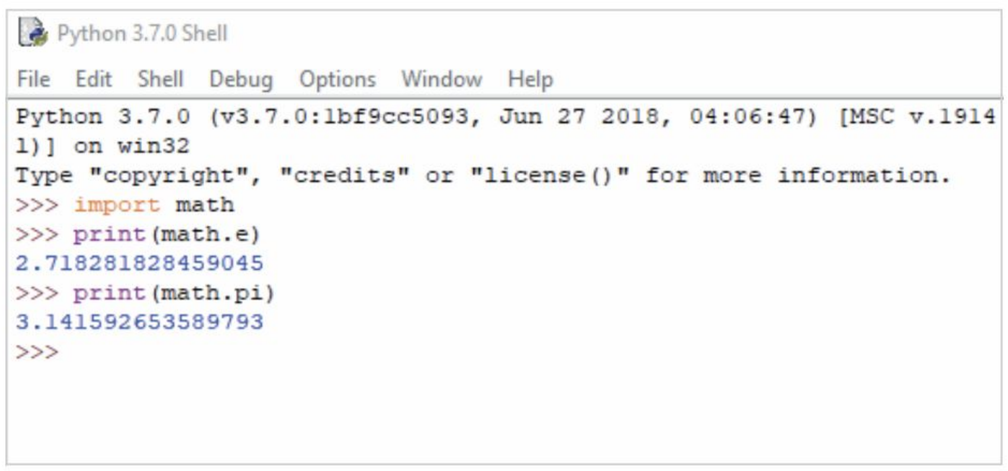
For further accuracy when it comes to numbers, the **exp** and **expm1** functions can be used to compute precise values:

```
from math import exp, expm1
exp(1e-5) - 1 # value accurate to 11 places
expm1(1e-5) # result accurate to full precision
```

STEP 9

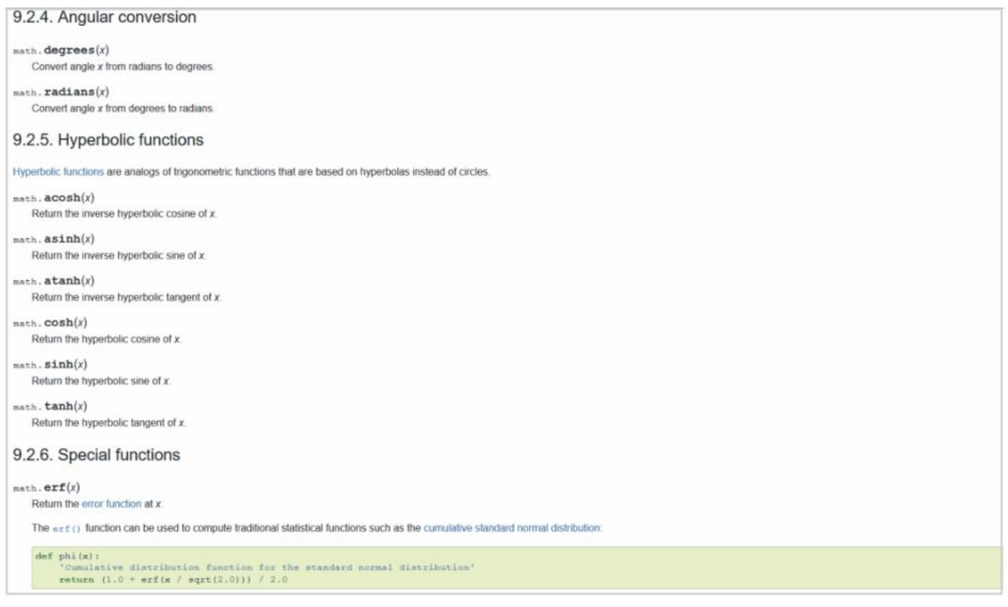
This level of accuracy is really quite impressive, but quite niche for the most part. Probably the two most used functions are **e** and **Pi**, where **e** is the numerical constant equal to 2.71828 (where the circumference of a circle is divided by its diameter):

```
import math
print(math.e)
print(math.pi)
```



STEP 10

The wealth of Mathematical functions available through the Math module is vast, and covers everything from factors to infinity, powers to trigonometry, and angular conversion to constants. Look up **https://docs.python.org/3/library/math.html#** for a list of available Math module functions.



9.2.4. Angular conversion

`math.`**degrees**(*x*)
Convert angle *x* from radians to degrees.

`math.`**radians**(*x*)
Convert angle *x* from degrees to radians.

9.2.5. Hyperbolic functions

Hyperbolic functions are analogs of trigonometric functions that are based on hyperbolas instead of circles.

`math.`**acosh**(*x*)
Return the inverse hyperbolic cosine of *x*.

`math.`**asinh**(*x*)
Return the inverse hyperbolic sine of *x*.

`math.`**atanh**(*x*)
Return the inverse hyperbolic tangent of *x*.

`math.`**cosh**(*x*)
Return the hyperbolic cosine of *x*.

`math.`**sinh**(*x*)
Return the hyperbolic sine of *x*.

`math.`**tanh**(*x*)
Return the hyperbolic tangent of *x*.

9.2.6. Special functions

`math.`**erf**(*x*)
Return the error function at *x*.

The `erf()` function can be used to compute traditional statistical functions such as the cumulative standard normal distribution:

```
def phi(x):
    'Cumulative distribution function for the standard normal distribution'
    return (1.0 + erf(x / sqrt(2.0))) / 2.0
```

Random Module

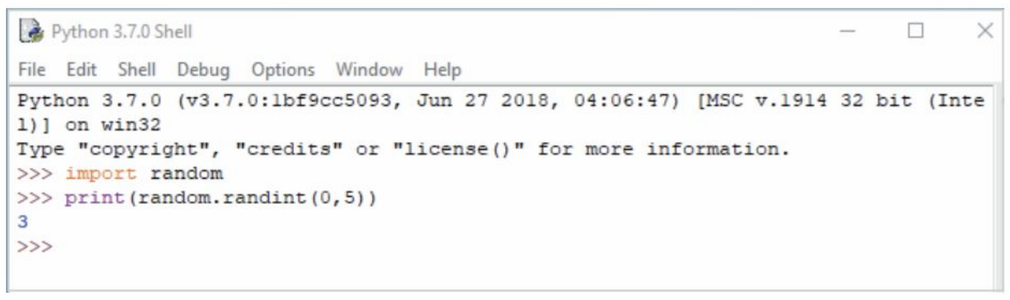
The Random module is one you will likely come across many times in your Python programming lifetime. As the name suggests, it's designed to create a random set of numbers from a given set. Although it's not exactly random, there is a pattern behind it, it will suffice for most needs.

RANDOM NUMBERS

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

STEP 1 As with other modules, you'll 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))
```

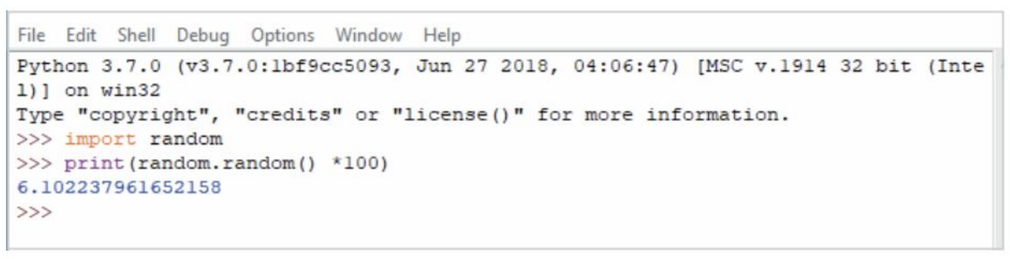


STEP 2 In our example the number three was returned. However, enter the print function a few more times and it will display different integer values from the set of numbers given, zero to five. The overall effect, although pseudo-random, is adequate for the average programmer to utilise in their code.

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

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

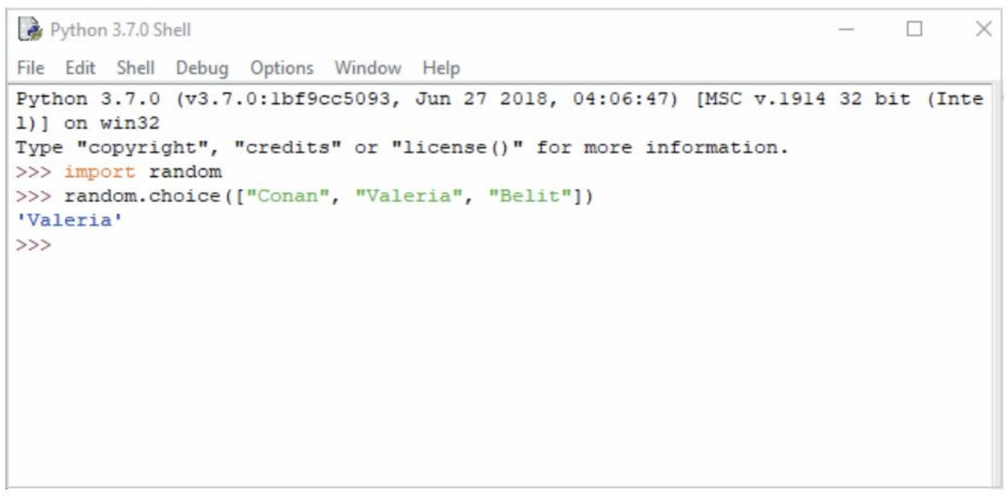
Will display a floating point number, between 0 and 100, to the tune of around fifteen decimal points.



STEP 4 However, the Random module isn't used exclusively for numbers. We can use it to select a list entry 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 We can extend the previous example somewhat by having random.choice() select from a list of mixed variables. For instance:

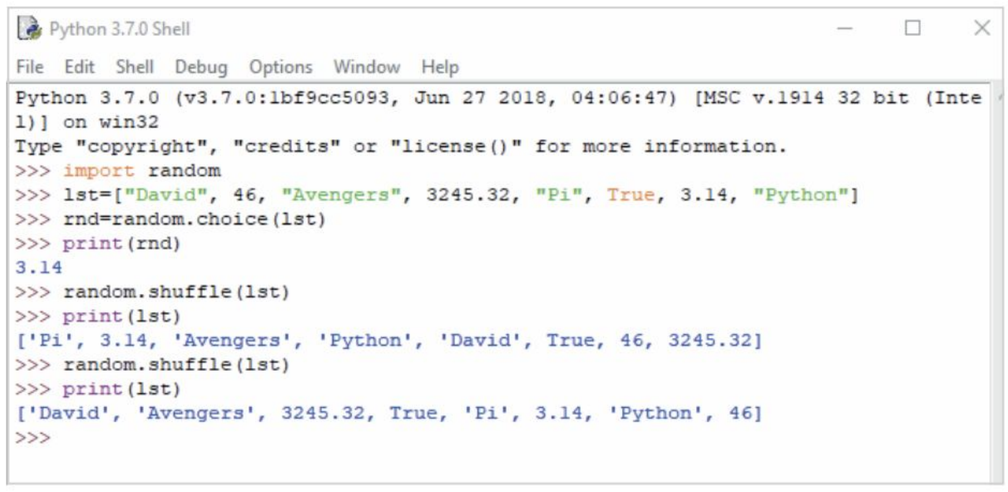
```
import random
lst=["David", 46, "Avengers", 3245.32, "Pi", True,
3.14, "Python"]
rnd=random.choice(lst)
print(rnd)
```

STEP 6

Interestingly, we 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.

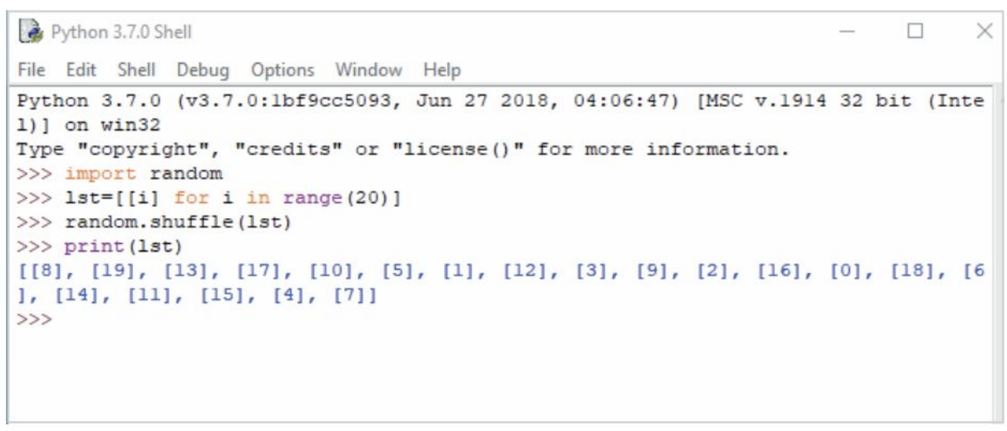


STEP 7

Using shuffle, we 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'll have a different selection of items from 0 to 20 every time.

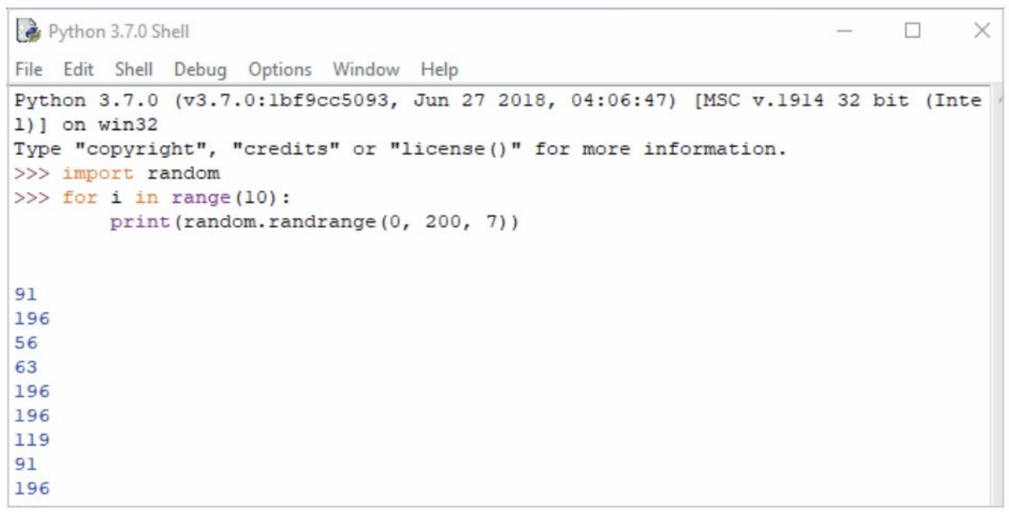


STEP 8

We 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))
```

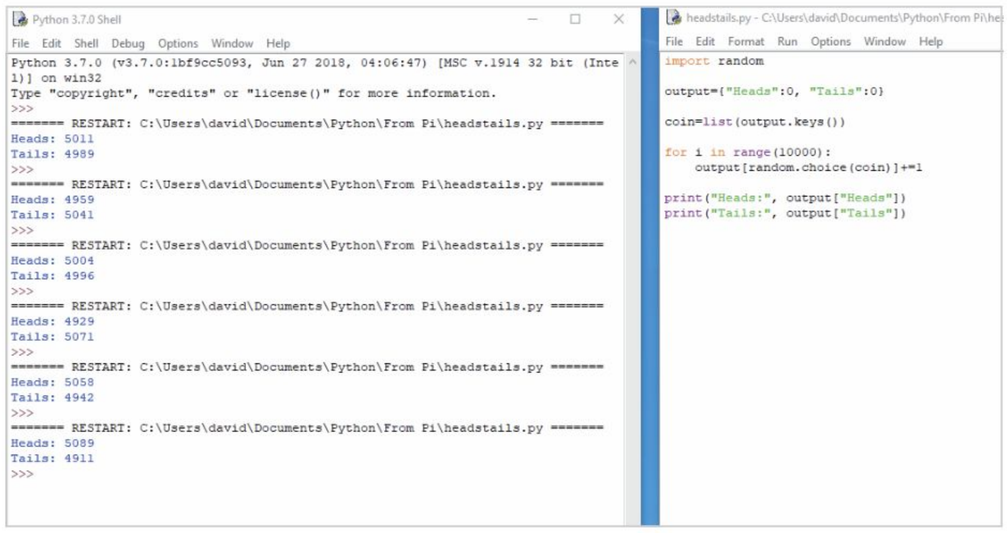
Result will vary, but you get the general idea as to how it works.



STEP 9

Let's use an example piece of code, which flips a virtual coin ten thousand times and counts how many times it'll land on Heads or Tails:

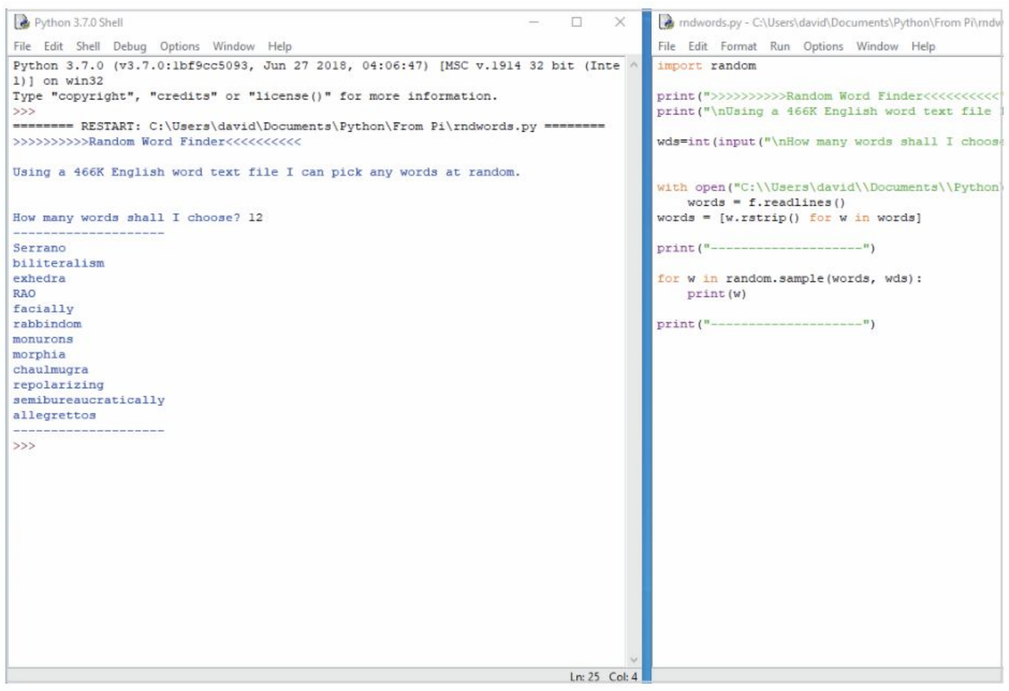
```
import random
output={"Heads":0, "Tails":0}
coin=list(output.keys())

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, we can pluck a user generated number of words from the file (text file found at: https://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 then getting some code into it.

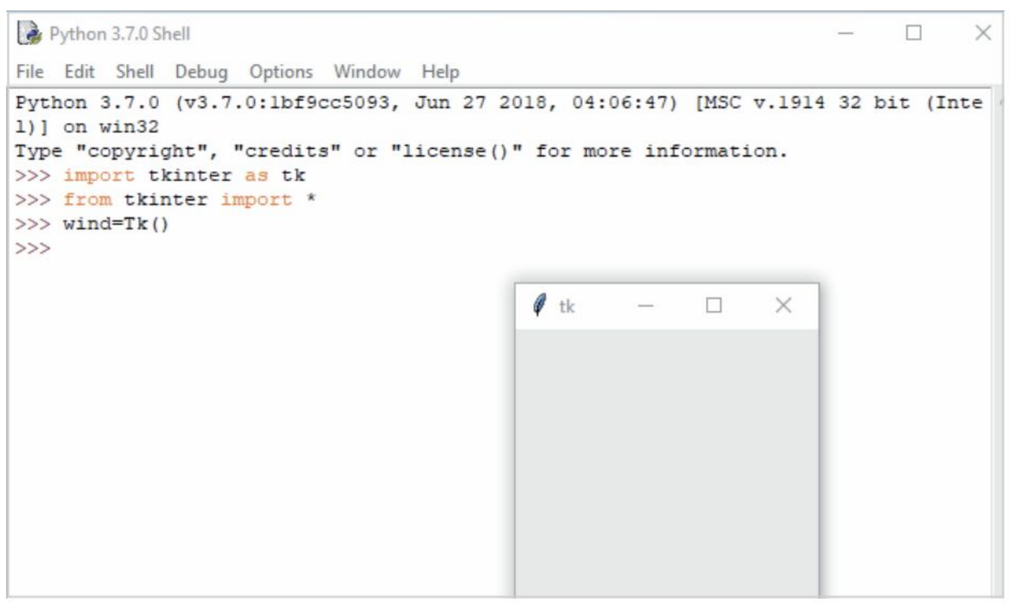
STEP 1 Tkinter is usually built into Python 3, however if it's available when you enter: `import tkinter`, then you'll need to **pip install tkinter** from the command prompt. We can start to import modules differently than before, to save on typing, by importing all their contents:

```
import tkinter as tk
from tkinter import *
```

STEP 2 Although it's not recommended to import everything from a module using the asterisk, normally it won't do any harm. Let's begin by creating a basic GUI window, enter:

```
wind=Tk()
```

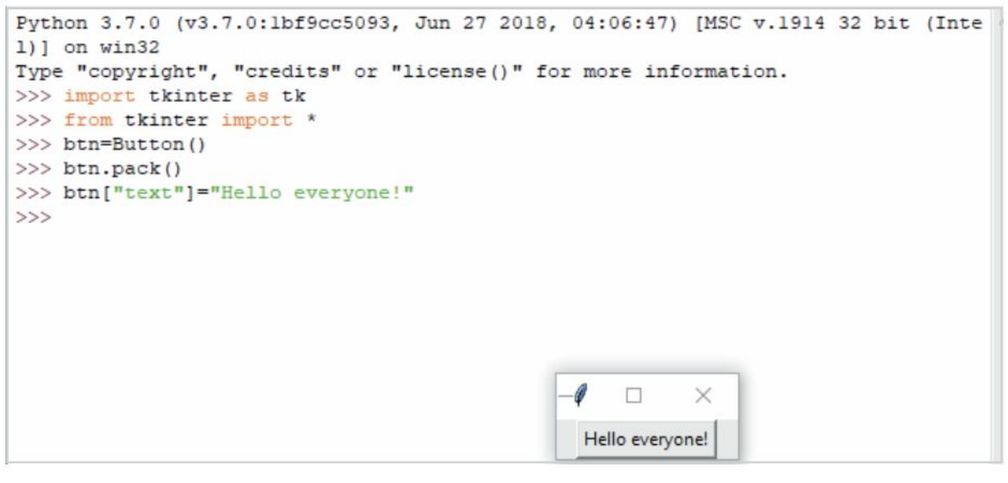
This creates a small, basic window. At this point, there's not much else to do but click the X in the corner to close the window.



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.

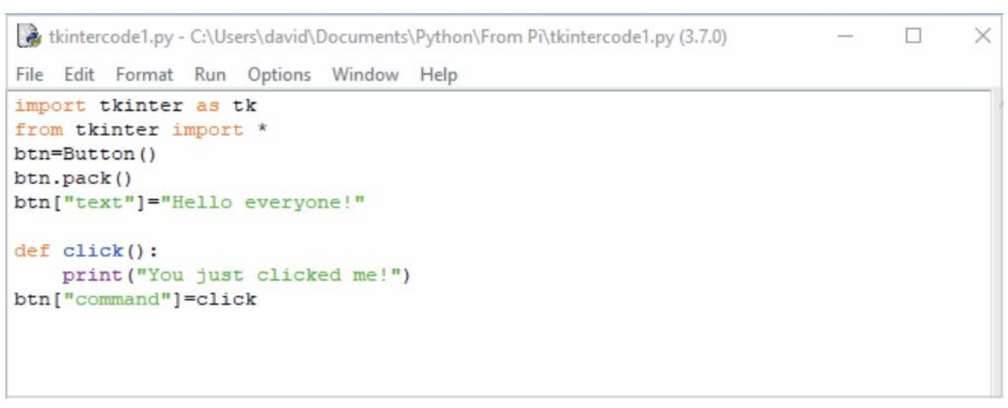


STEP 4 We can combine the above into a New File:

```
import tkinter as tk
from tkinter import *
btn=Button()
btn.pack()
btn["text"]="Hello everyone!"
```

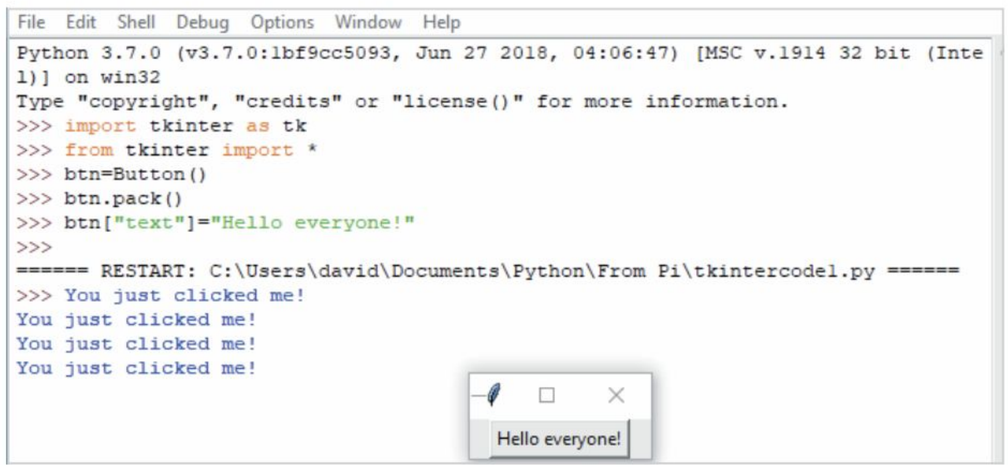
Then add some button interactions:

```
def click():
  print("You just clicked me!")
btn["command"]=click
```



STEP 5

Save and execute the code from Step 5, and a window will appear 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.



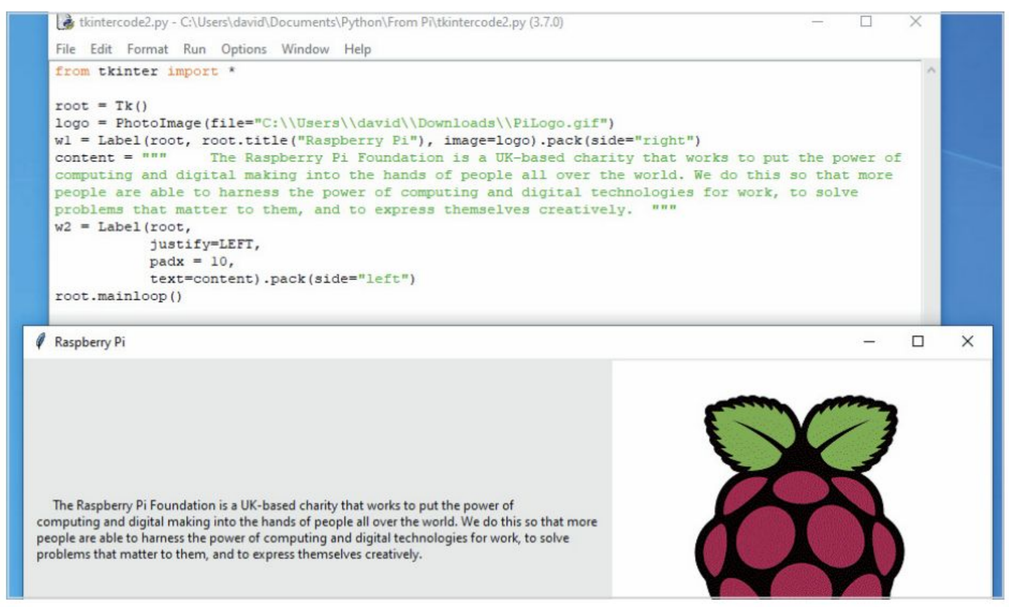
STEP 6

We 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 Raspberry Pi logo:

```
from tkinter import *

root = Tk()
logo = PhotoImage(file="C:\\Users\\david\\Downloads\\
PiLogo.gif")
w1 = Label(root, root.title("Raspberry Pi"),
image=logo).pack(side="right")
content = """     The Raspberry Pi Foundation is a
UK-based charity that works to put the power of
computing and digital making into the hands of
people all over the world. We do this so that more
people are able to harness the power of computing
and digital technologies for work, to solve
problems that matter to them, and to express
themselves creatively.  """
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 Raspberry Pi'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

We 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

And we can 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.

```
from tkinter import *
root = Tk()

def var_states():
   print("Warrior: %d,\nMage: %d" % (var1.get(), var2.get(), var3.get(), var4.get(), var5.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)
var3 = IntVar()
Checkbutton(root, text="Ranger", variable=var3).grid(row=3, sticky=W)
var4 = IntVar()
Checkbutton(root, text="Barbarian", variable=var4).grid(row=4, sticky=W)
var5 = IntVar()
Checkbutton(root, text="Paladin", variable=var5).grid(row=5, sticky=W)

Button(root, text='Quit', command=root.destroy).grid(row=7, sticky=W, pady=4)
Button(root, text='Show', command=var_states).grid(row=7, sticky=E, pady=4)
mainloop()
```

Pygame Module

We've had a brief look at the Pygame module already, but there's a lot more to it that needs to be explored. Pygame was developed to help Python programmers create games, whether they're graphical or not. You will, however, need to install Pygame before you can use it.

PYGAMING

As stated, Pygame isn't an inherent module to Python. 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, we need to load up the Pygame modules into memory before we're able to utilise them. Once that's done, Pygame requires the user to initialise it prior to any of the functions being used:

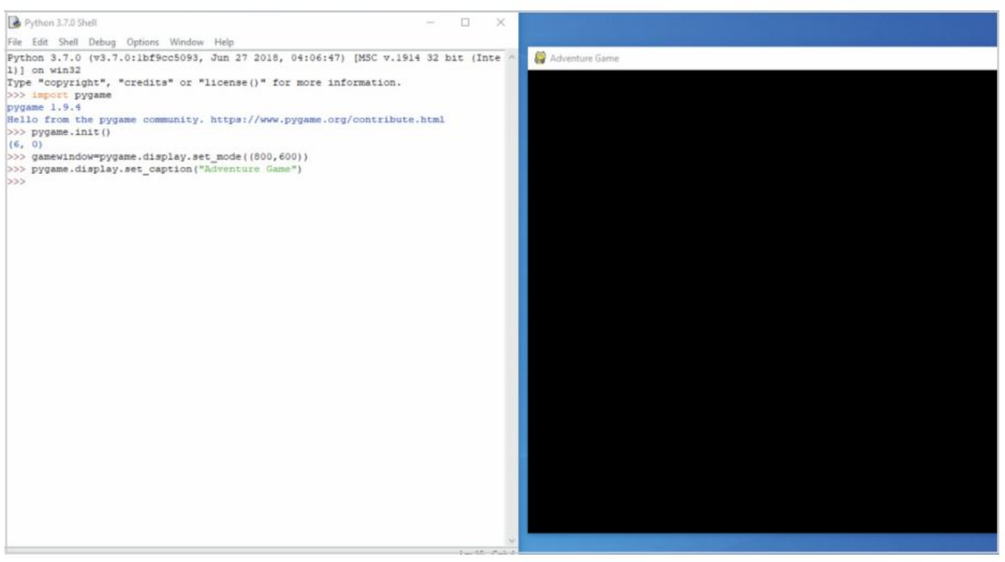
```
import pygame
pygame.init()
```

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 will notice that after the first line is entered, you'll need to click back into the IDLE Shell to continue entering code. In addition, 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, we 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()
```

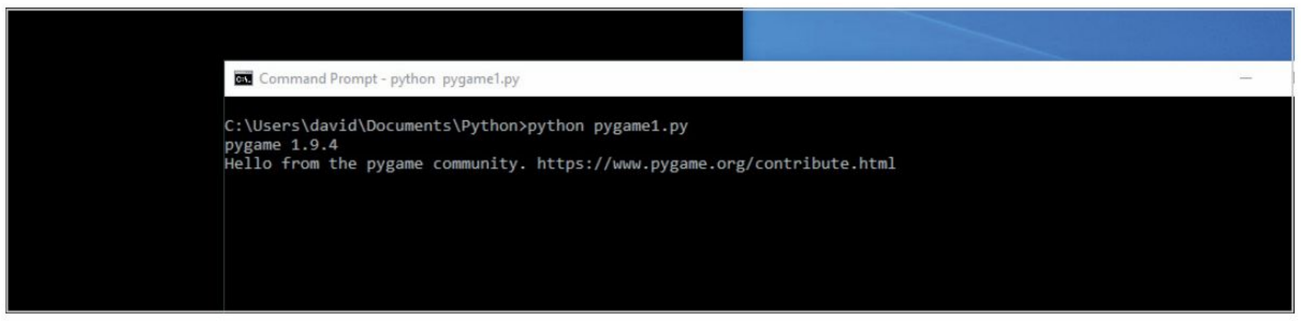
Untitled

File Edit Format Run Options Window Help

```
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

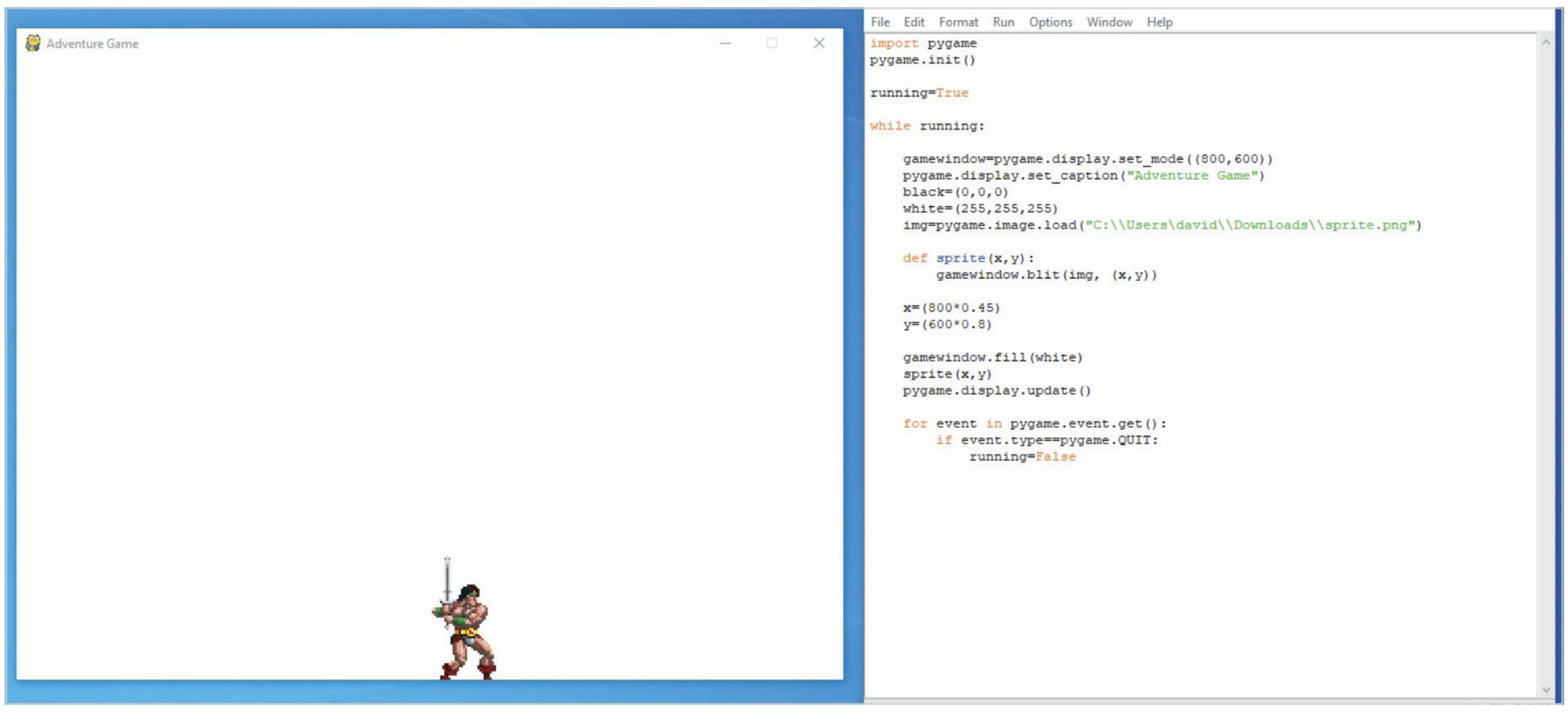
If the Pygame window still won't close, don't worry, it's just a discrepancy between the IDLE (which is written with Tkinter) and the Pygame module. If you run your code via the command line, it will close perfectly fine.



STEP 5

We'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. Also, 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("C:\\Users\david\\
Downloads\\sprite.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

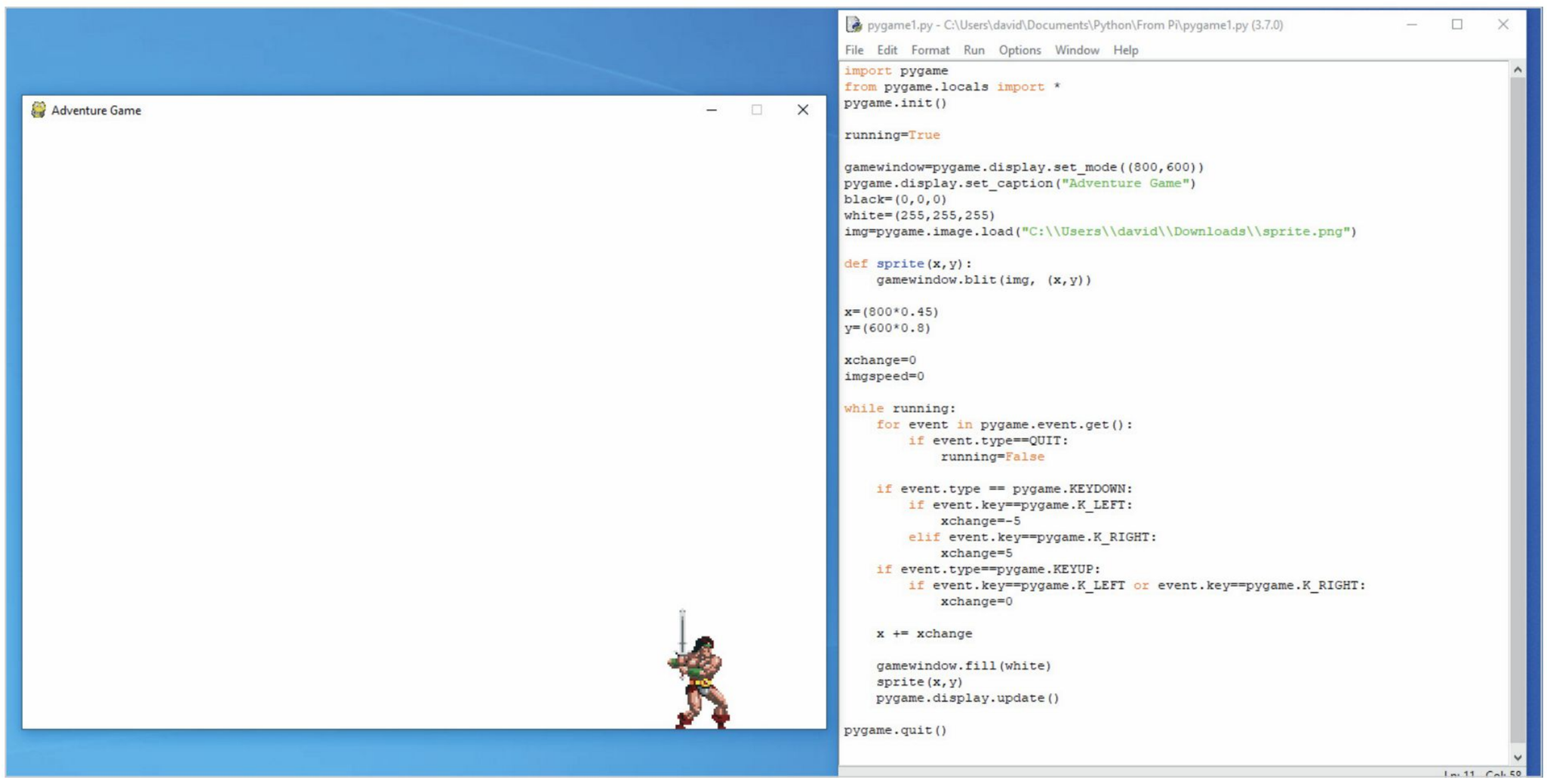
Let's quickly go through the code changes. We've defined two colours, black and white, together with their respective RGB colour values. Next, we've loaded the downloaded image called **sprite.png**, and allocated it to the variable img. We've also defined a sprite function, and the Blit function, which will allow us to eventually move the image.

```
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("C:\\Users\david\\Downloads\\sprite.p

    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 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("C:\\Users\\david\\
Downloads\\sprite.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 will be able to move your sprite across the bottom of the screen. It looks like we 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("C:\\Users\\david\\Downloads\\sprite.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 We can now implement a few additions, and utilise some previous tutorial code. The new elements are in 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:

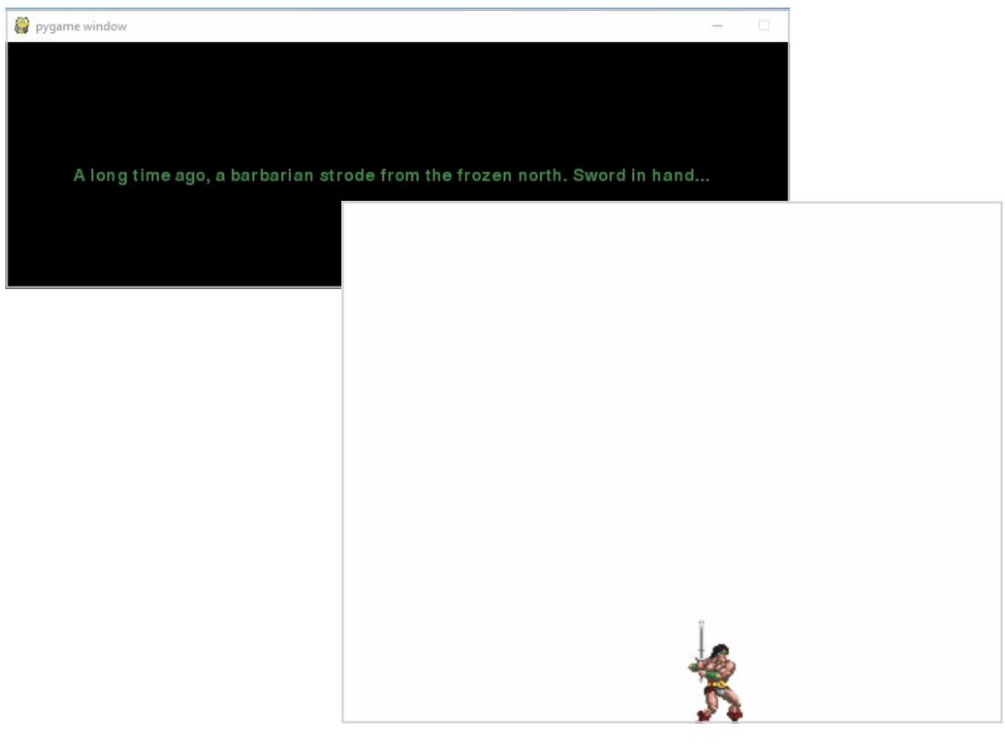
```
import pygame
import time
import subprocess
pygame.init()
screen = pygame.display.set_mode((800, 250))
clock = pygame.time.Clock()

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(“python C:\\Users\\david\\
Documents\\Python\\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()
```

pygametxt.py - C:\Users\david\Documents\Python\From Pi\pygametxt.py (3.7.0)

File Edit Format Run Options Window Help

```
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("python C:\\Users\\david\\Documents\\Python\\pygamel.p


    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...")
```

STEP 10 When we 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 we 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 display graphics, and animate them accordingly. Animation, though, can be a tricky element to get right in code. There are so 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 to 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 both 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. A blank window will appear, with a red ball sitting in the upper left corner of the window. While great, it's not very animated. Let's add the following code:

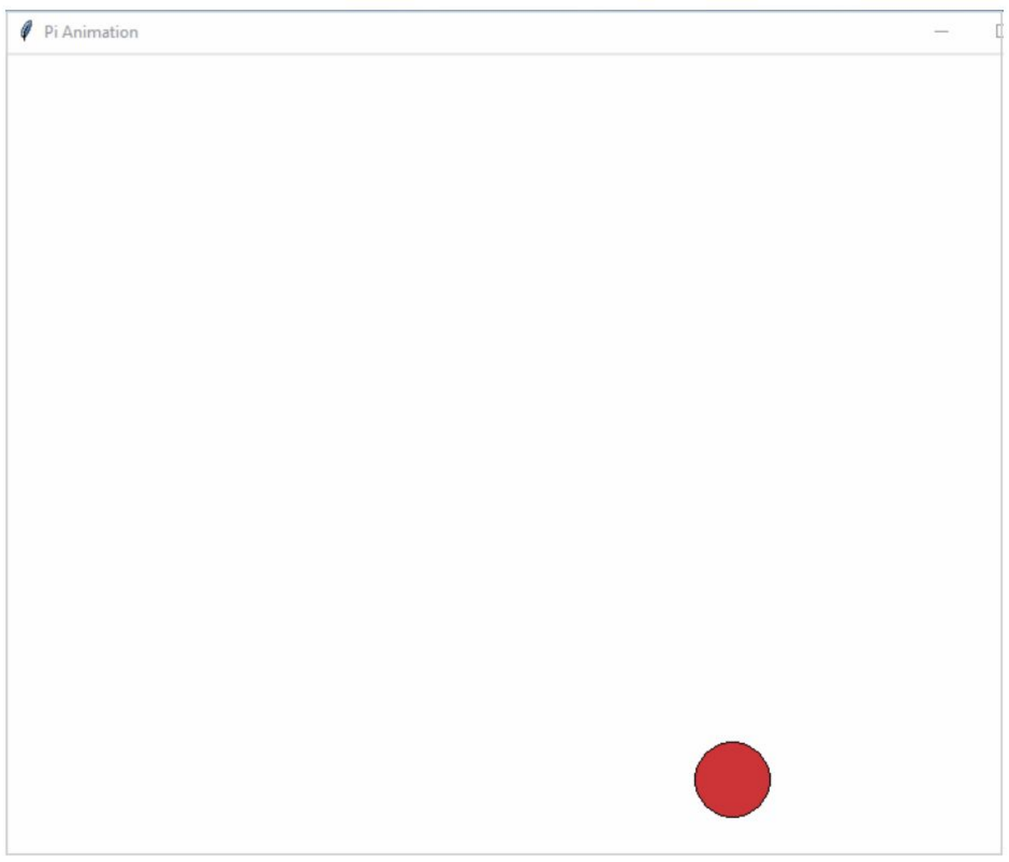
```
a = 5
b = 5

for x in range(0,100):
        canvas.move(ball1,a,b)
        gui.update()
        time.sleep(.01)
```

step1.py - C:/Users/david/Documents/Python/step1.py (3.7.0)

File Edit Format Run Options Window Help

```
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')

a = 5
b = 5

for x in range(0,100):
        canvas.move(ball1,a,b)
        gui.update()
        time.sleep(.01)

gui.mainloop()
```

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 what 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.

File Edit Format Run Options Window Help

```
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')
```

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, 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.

STEP 8

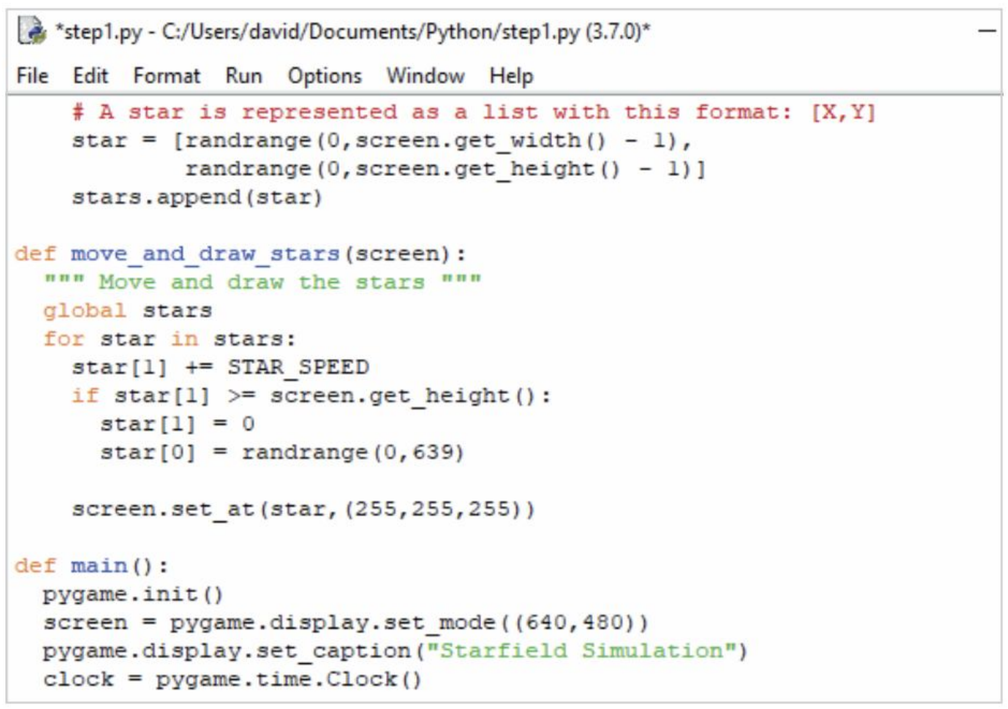
Pygame, however, is a much better module at producing higher-end animations. Begin by creating a New File, and entering:

```
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

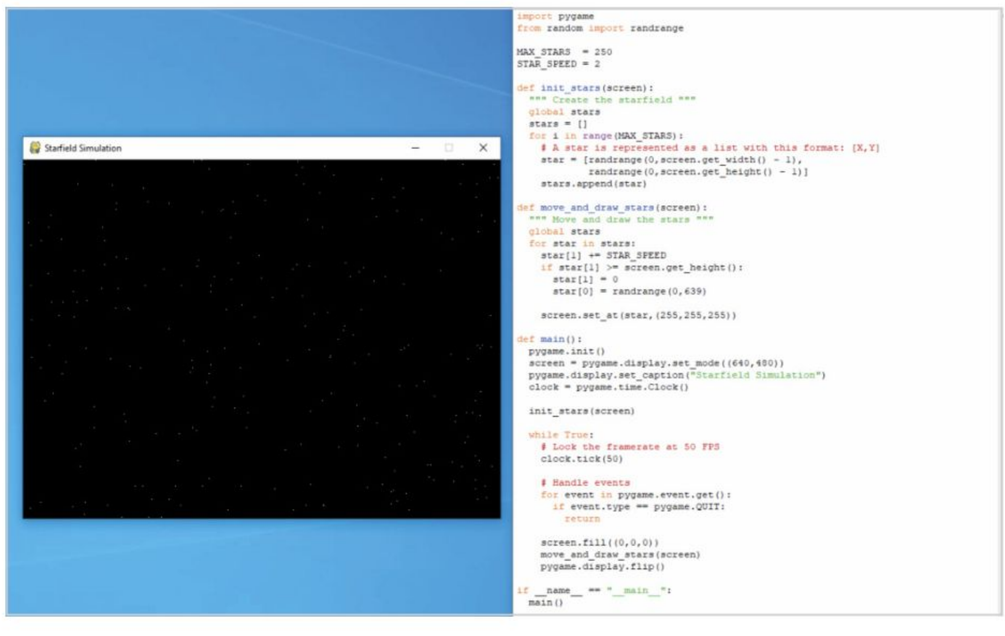
Now add the following:

```
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()
```



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

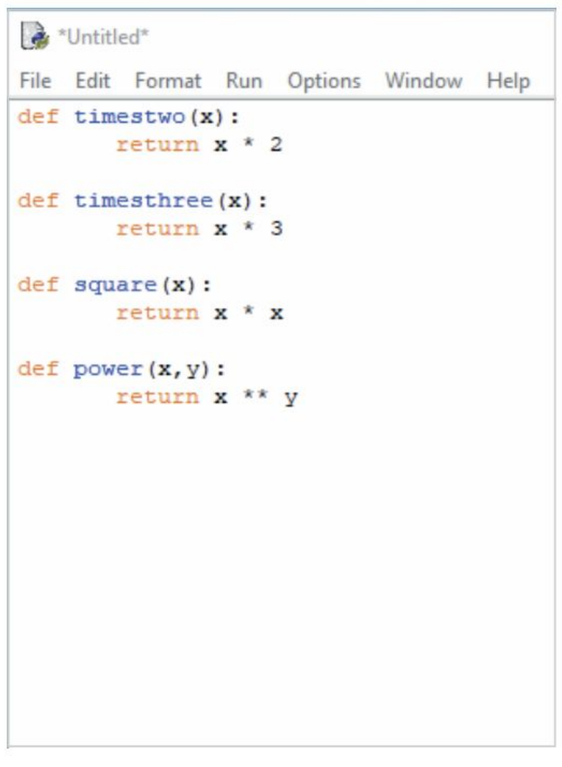
Large programs can be much easier to manage if you break them up into smaller parts and import the parts you need as modules. Learning to build your own modules also makes it easier to understand how modules work.

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 or three, and square or raise a number to an exponent (power). Create a New File in the IDLE and enter:

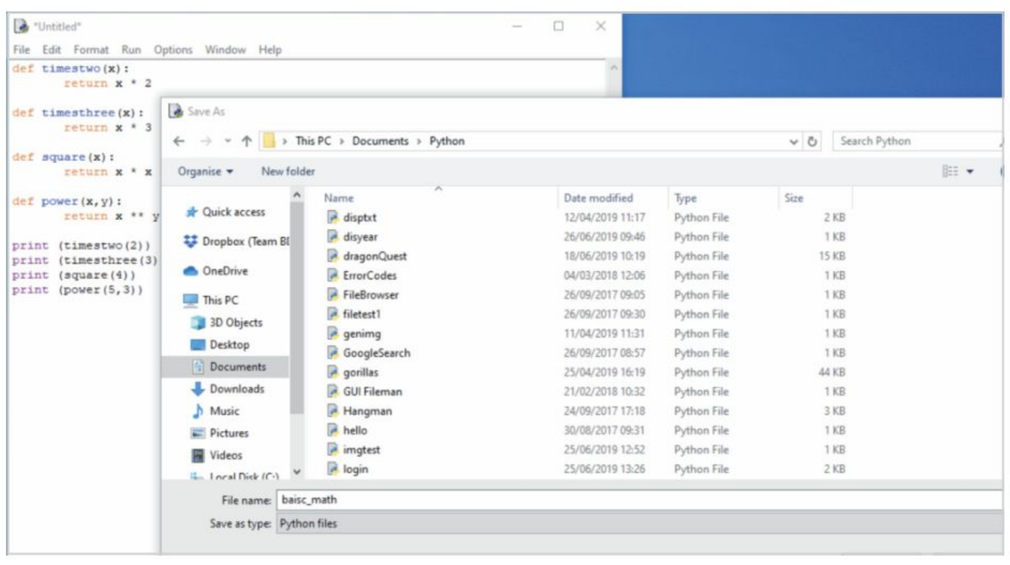
```
def timestwo(x):
        return x * 2

def timesthree(x):
        return x * 3

def square(x):
        return x * x

def power(x,y):
        return x ** y
```



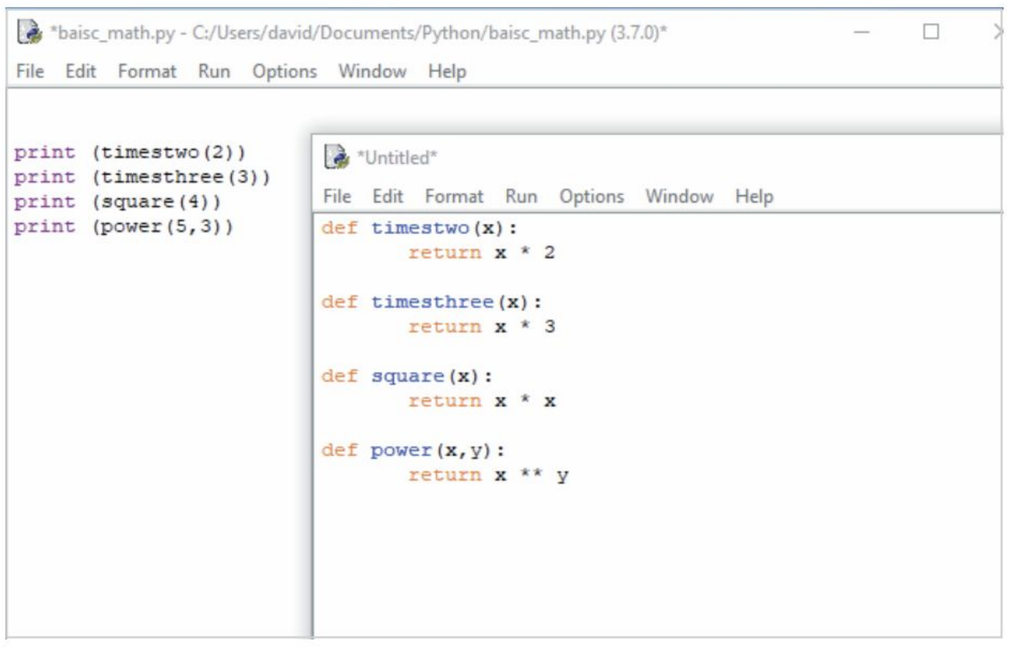
STEP 2 Under the above code, enter functions to call the code:

```
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.



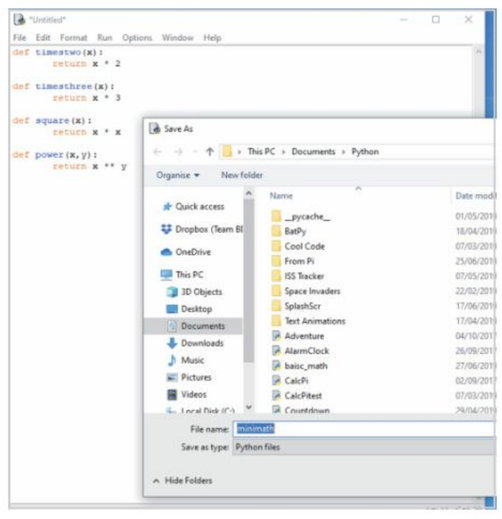
STEP 3 Now we're going to take the function definitions out of the program and into a separate file. Highlight the function definitions and choose **Edit > Cut**. Choose **File > New File** and use **Edit > Paste** in the new window. We now have two separate files, one with the function definitions, and 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 "C:/Users/david/Documents/Python/baisc_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.

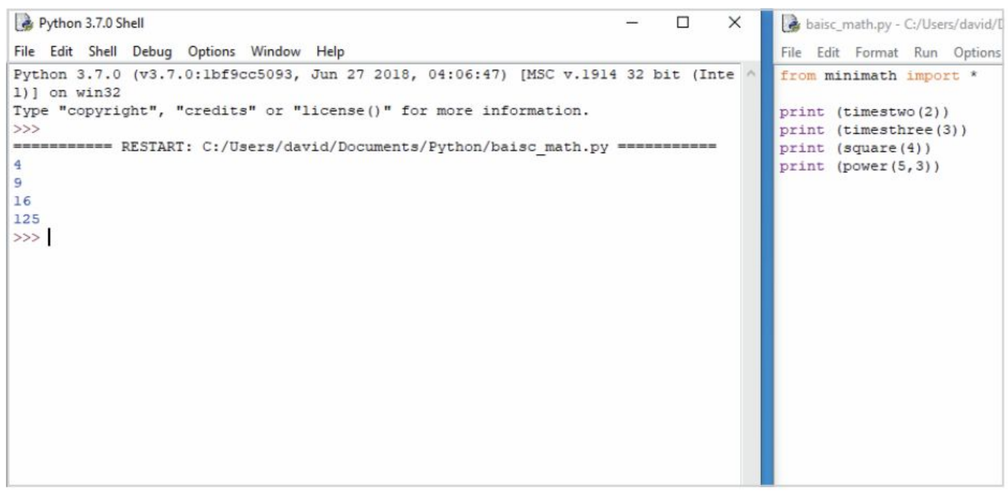


STEP 6

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, and see it in action.



STEP 7

We can now make the program a little more advanced, by utilising the newly created module to its full. Let's include some user interaction. Start by creating a basic menu from which the user can choose:

```
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**.

testmath.py - C:/Users/david/Documents/Python/testmath.py (3.7.0)

File Edit Format Run Options Window Help

```
from minimath import *

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):")

num1 = int(input("\nEnter number: "))
```

STEP 9

Finally, we 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")
```

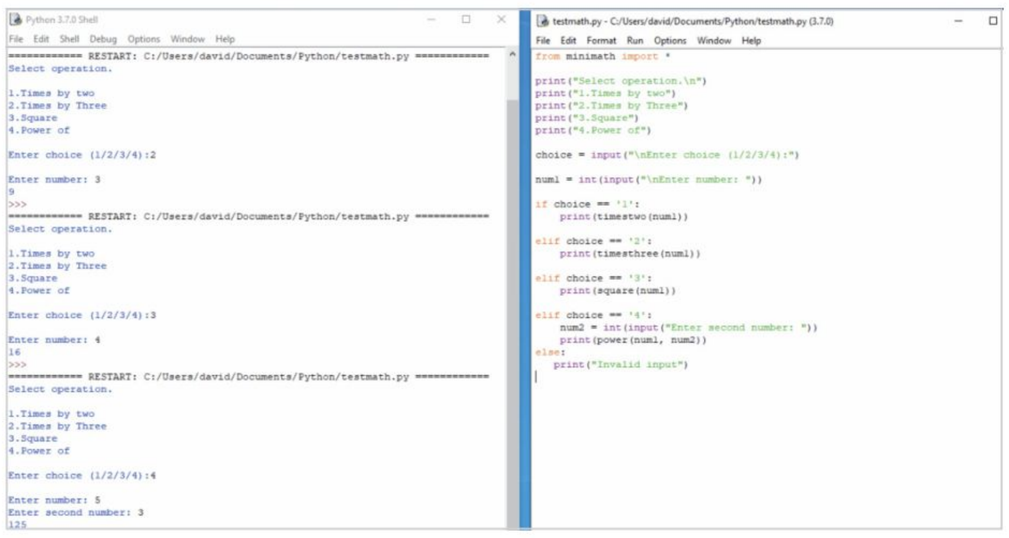
testmath.py - C:/Users/david/Documents/Python/testmath.py (3.7.0)

File Edit Format Run Options Window Help

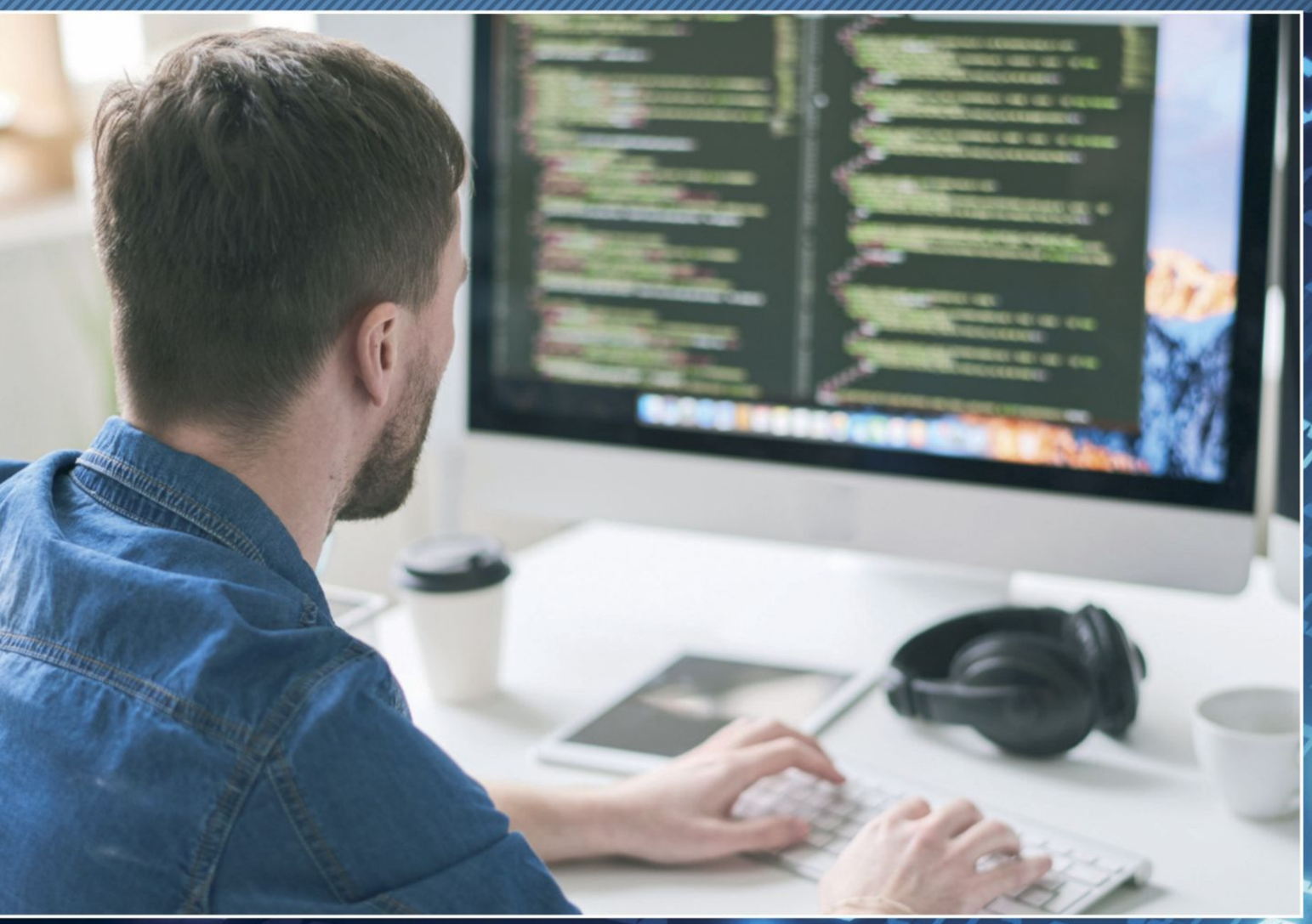
```
from minimath import *

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):")

num1 = int(input("\nEnter number: "))

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.



Learning Linux



Linux is a flexible and powerful operating system but it also has a unique filesystem and way of doing things. Therefore, learning how Linux works is essential to creating better Python code.

You'll find that Linux already has Python elements built-in and if you're working on a Raspberry Pi 4, then everything you need to code with Python is ready from the start. In this section we will mainly focus on using the Raspberry hardware with Linux but regardless of whether you're using an RPi, Linux Mint, or Ubuntu, these pages will prove invaluable for your Python learning.

110 What is Linux?
112 Why Linux?
114 Using the Filesystem
116 Listing and Moving Files
118 Creating and Deleting Files
120 Create and Remove Directories
122 Copying, Moving and Renaming Files
124 Using the Man Pages
126 Editing Text Files
128 Getting to Know Users
130 Ownership and Permissions
132 Useful System and Disk Commands
134 Managing Programs and Processes
136 Input, Output and Pipes
138 Fun Things to Do in the Terminal
140 More Fun Things to Do in the Terminal
142 Linux Tips and Tricks
144 Command Line Quick Reference
146 A-Z of Linux Commands

What is Linux?

The Raspberry Pi operating system is Raspbian, which is a Linux operating system. But, what exactly is Linux, where did it come from, and what does it do? In a world where Windows and macOS have supremacy of the desktop, it's easy to overlook Linux; but there's more here than you might imagine.

Linux is a surprisingly powerful, fast, secure, and capable operating system. It's used as the OS of choice for the Raspberry Pi, in the form of Raspbian OS, as well as in some of the most unlikely places.

Despite only enjoying 1.96% (according to netmarketshare.com) of the total desktop operating system market, Linux has a dedicated following of enthusiasts, users, and contributors. It was created in 1991 by then University of Helsinki student, Linus Torvalds, who had become frustrated with the limitations and licensing of the popular educational system in use called Minix; a miniature version of the UNIX operating system.

UNIX itself was released in the early '70s, as a multi-tasking, modular-designed operating system, originally developed for programmers who needed a stable platform on which to code. However, its performance, power, and portability meant that it soon became the system of choice for companies, and universities, where high-end computing tasks were needed.

Torvalds needed a system that could mirror UNIX's performance and features, without the licensing cost. Thus was born Linux, a UNIX-like operating system using freely available code from the GNU project. This enabled users around the world to utilise the power of a UNIX-like system, completely free of charge – an ethos that still holds today. Linux is free to download, install, and use.

Essentially, Linux is much like any other operating system, such as Windows or macOS. It manages the computer hardware, provides an interface for the user to access that hardware, and provides programs for productivity, communications, gaming, science, education and more. As an operating system, Linux can be broken up into a number of significant elements:

BOOTLOADER

The bootloader is the software that initialises and boots up your computer. It loads up the various modules the OS uses to begin to access the hardware in the system. You can modify a bootloader to load more than one OS installed on the system.

DAEMONS

Daemons are background services that will start as the operating system is booting. These can enable printing, sound, networking and so on. They run unobtrusively rather than under the direct control of the user; often waiting to be activated by an event or condition.

DESKTOP ENVIRONMENTAL

The Desktop Environment, or DE, is the main Graphical User Interface (GUI) with which users interact. It's the desktop that includes Internet browsers, productivity, games, and whatever program or app you're using. There are countless DEs available, Raspbian uses PIXEL.

GRAPHICAL SERVER

This is a module within Linux that provides a graphical output to your monitor. It's referred to as the X server, or simply X. X is an application that manages one or more graphical displays, and one or more input devices (keyboard, mouse, etc.) connected to the computer.

KERNEL

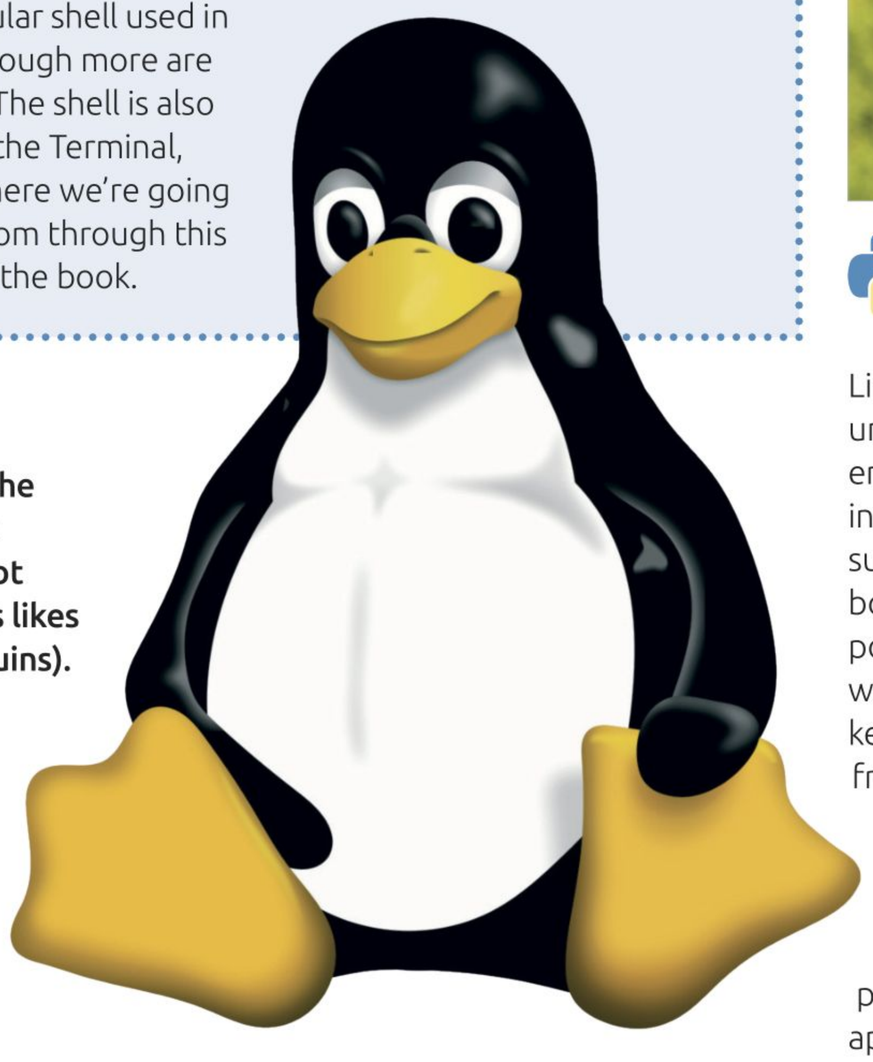
The kernel is the core of the system, and the single element that is actually called Linux. The Linux kernel manages the computer processor, memory, storage, and any peripherals you have attached to your computer. It provides the basic services for all other parts of the OS.

PROGRAMS/APPLICATIONS

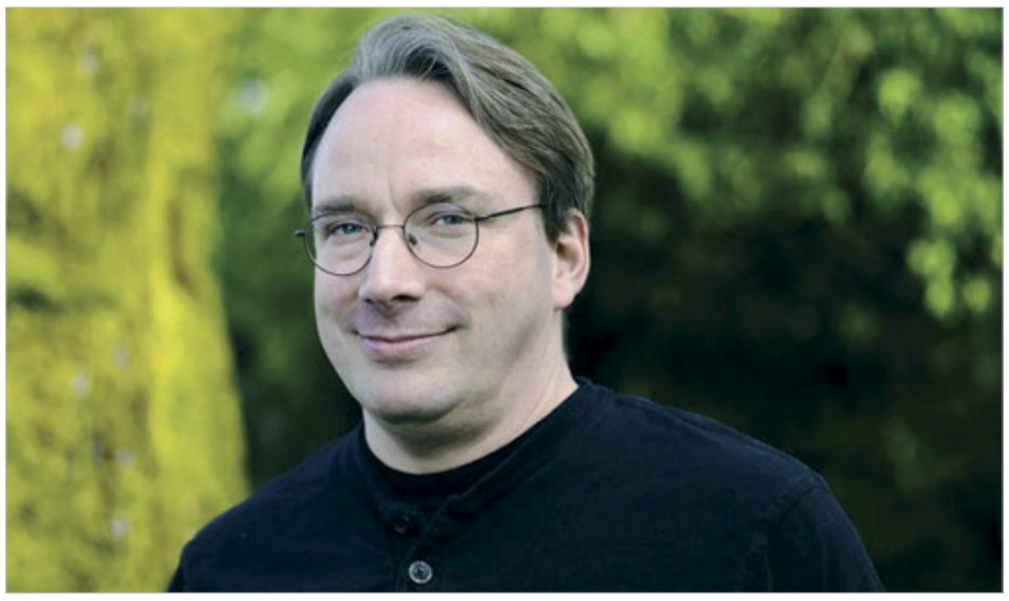
With Linux, begins not only an open source, and free, operating system, it also makes use of the tens of thousands of freely available applications too. The likes of LibreOffice, GIMP, and Python are just the tip of the iceberg.

SHELL

The Linux shell is a command-line interface environment that a Linux user can use to enter commands to the OS that directly affect it. Within the shell, you can add new users, reboot the system, create, and delete files and folders, and much more. BASH (Bourne-Again Shell) is the most popular shell used in Linux, although more are available. The shell is also known as the Terminal, and it's where we're going to work from through this section of the book.



Tux, the Linux mascot (Linus likes penguins).

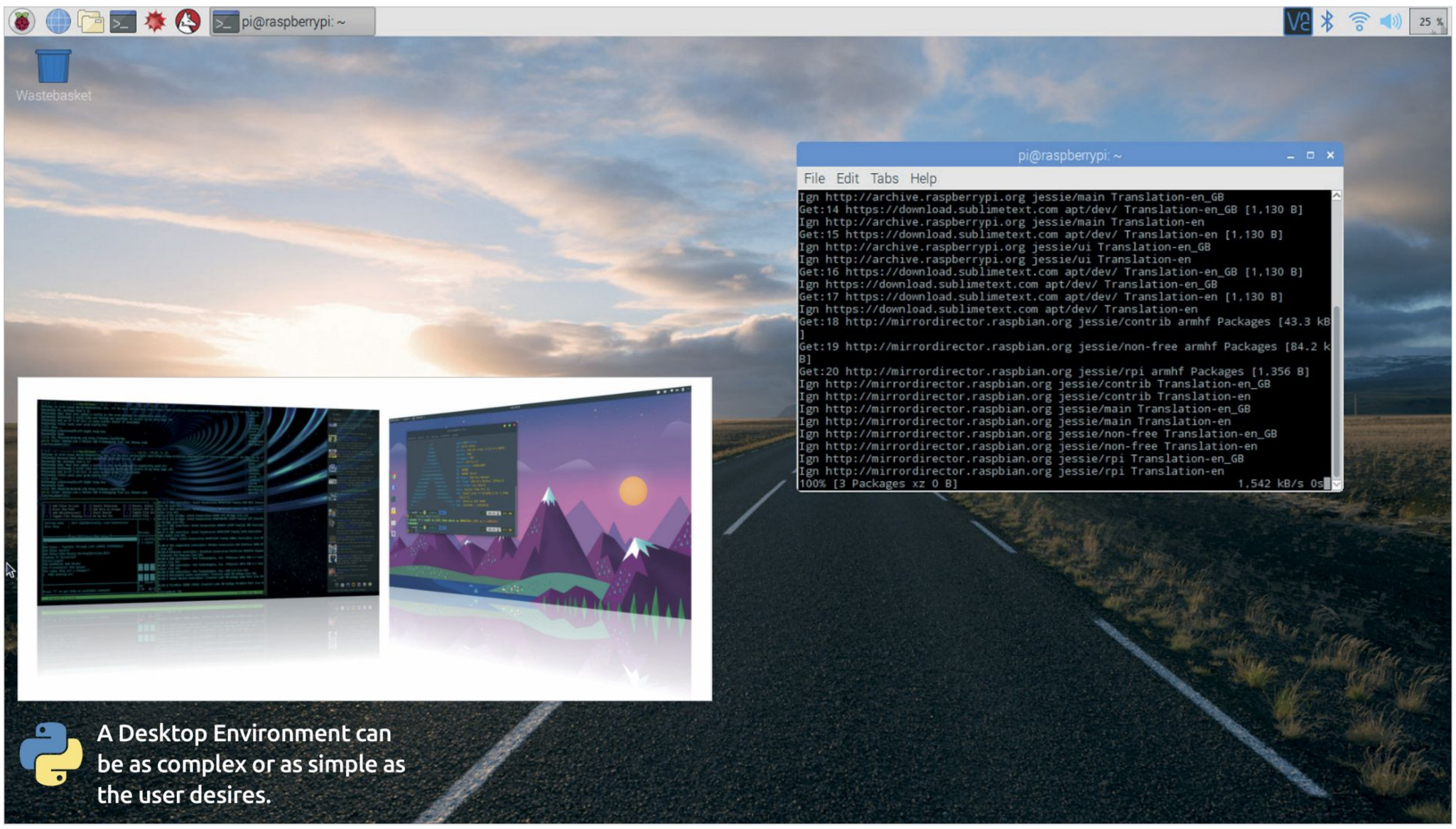


Linus Torvalds, the creator of the Linux kernel.

Linux is used throughout the world, in several basic, and quite unique, situations. While it may look radically different from one environment to the next, the actual Linux kernel can be found in modern smart TVs, in-car entertainment systems and GPS, supercomputers, IoT devices, and the Raspberry Pi. NASA uses it, both in the command centre and on board the ISS. Linux servers power the backbone of the Internet, along with most of the websites you visit daily. Android utilises components of the Linux kernel, as do set-top boxes, games consoles, and even some fridges, freezers, ovens, and washing machines.

Linux isn't just a free to use operating system. It's stable, powerful, fast, can easily be customised, and requires very little maintenance. However, it's more than just a set of on-paper performance stats. Linux means freedom from the walled-garden approach of other operating systems. It's a lively community of like-minded individuals who want more from their computers, without the shackles of price or conformity. Linux means choice.

Raspbian on the Raspberry Pi, is the Linux distribution of choice.



A Desktop Environment can be as complex or as simple as the user desires.

Why Linux?

Linux, like its parental UNIX, is a fantastic platform on which to code. Not only does Linux come pre-installed with Python modules and command-line execution, but it also has a wealth of other programming languages built-in to its framework.

FREE AND OPEN

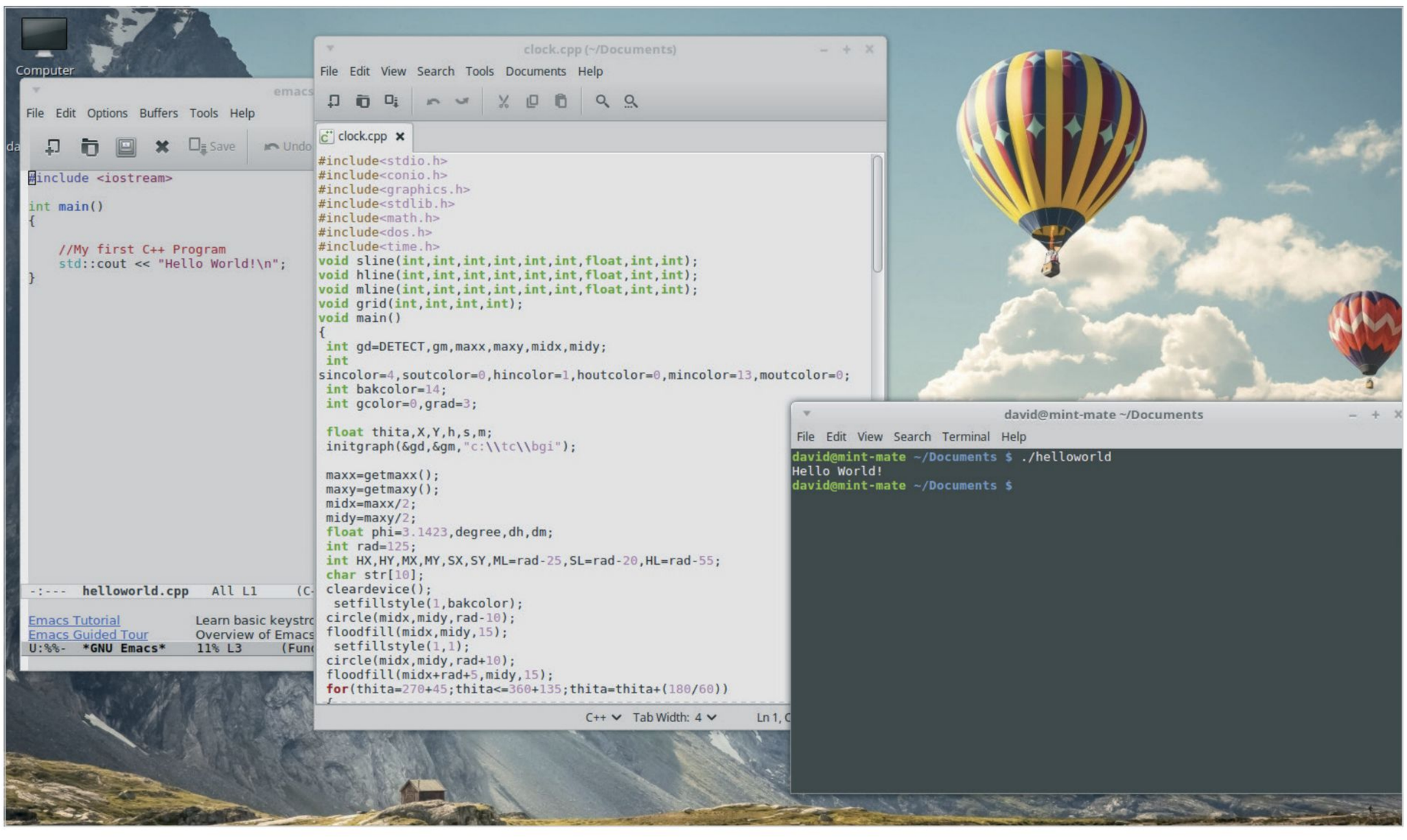
Linux is a fantastic fit for those who want something different. The efficiency of the system, the availability of applications, and its stability are just a few reasons why it's a great Python coding resource.

The first thing we need to address is that there is no such operating system called Linux. Linux is, in fact, the operating system kernel, the core component of an OS. When talking about Linux, what we, and others, are referring to is one of the many distributions, or distros, that use the Linux kernel. No doubt you've heard of at least one of the current popular distros: Ubuntu, Linux Mint, Fedora, openSUSE, Debian, Raspbian… the list goes on. Each one of these distros offers the user something a little different. While each has the Linux kernel at its core, they provide different looking desktop environments, different pre-loaded applications, different ways in which to update the system and get more apps installed, and a slightly different look and feel throughout the entire system. However, at the centre lies Linux; which is why we say, Linux.

Linux works considerably differently to Windows or macOS. It's free for a start, free to download, free to install on as many computers as you like, free to use for an unlimited amount of time, and free to upgrade and extend with, equally, free programs and applications. This free to use element is one of the biggest draws for the developer. While a Windows license can cost up to £100, and a Mac considerably more, a user, be they a developer, gamer, or someone who wants to put an older computer to use, can quickly download a distro and get to work in a matter of minutes.

Alongside the free to use aspect, comes a level of freedom to customise and mould the system to your own uses. Each of the distros available on the Internet have a certain 'spin', in that some offer

Linux is a great operating system in which to start coding.



increased security, a fancy-looking desktop, a gaming specific spin, or something directed toward students. This extensibility makes Linux a more desirable platform to use, as you can quickly mould the system into a development base, including many different kinds of IDEs for the likes of Python, web development, C++, Java and so on; or even a base for online anonymity, perhaps as a Minecraft server, media centre and much more.

Another remarkable advantage for those looking to learning how to code, is that Linux comes with most of the popular coding environments built-in. Both Python and C++ are pre-installed in a high percentage of Linux distros, which means you can start to program almost as soon as you install the system and boot it up for the first time.

Generally speaking, Linux doesn't take up as many system resources as Windows or macOS. By system resources, we mean memory, hard drive space, and CPU load, as the Linux code has been streamlined and is free from third-party 'bloatware', which hogs other systems resources. Of course, a more efficient system means more resources are available for the coding and testing environment, and the programs you will eventually create. Less use of resources also means you can use Linux on older hardware that would normally struggle, or even refuse, to run the latest versions of Windows or macOS. So rather than throwing away an old computer, it can be reused with a Linux distro.

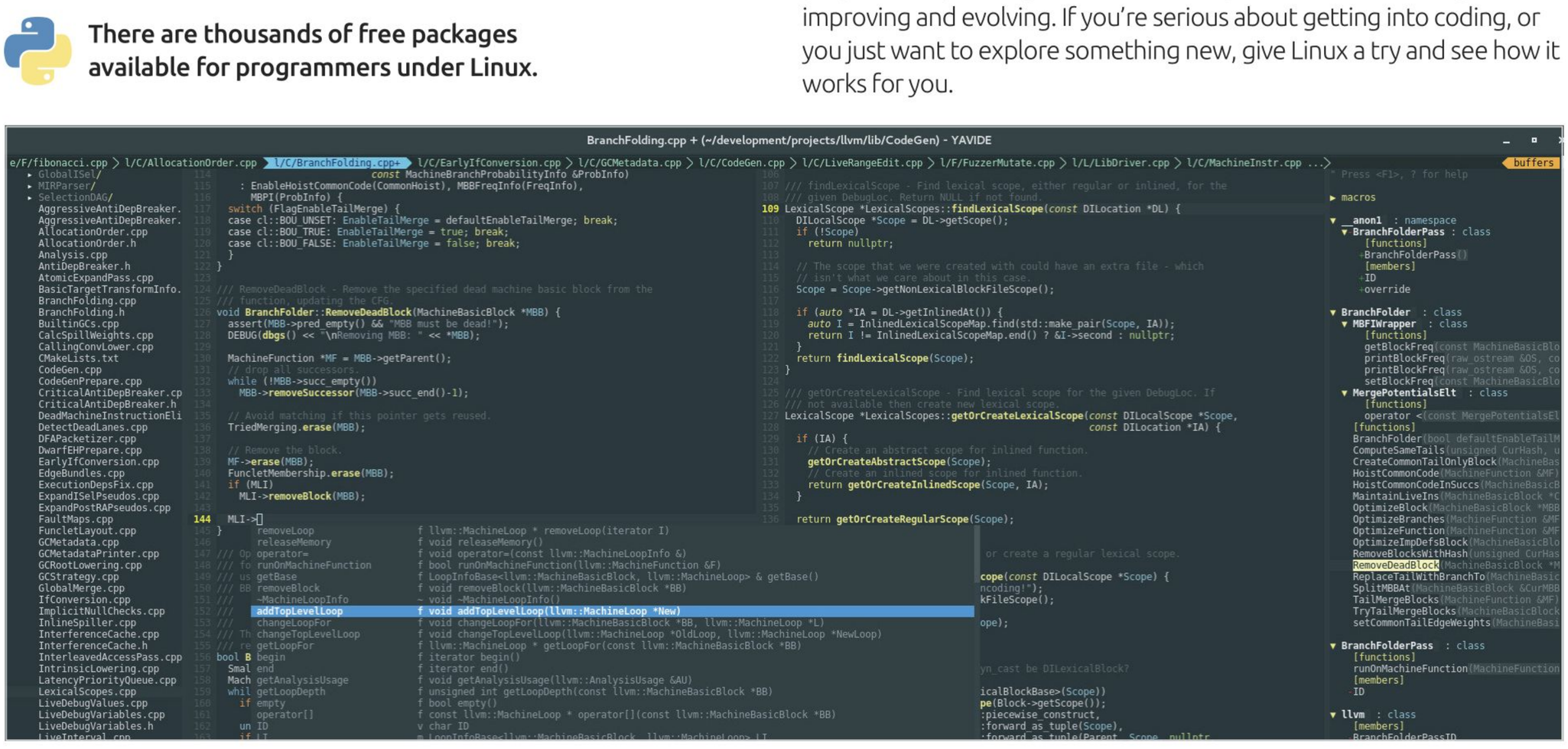
Each distro offers something unique to the user but all have Linux at the core.

There are thousands of free packages available for programmers under Linux.

It's not all about C++, Python, or any of the other more popular programming languages, however. Using the command line of Linux, also called the Terminal, you're able to create Shell scripts, which are programs designed to run from the command line and are made up of scripting languages. They are used mainly to automate tasks, or offer the user some form of input and output for a certain operation.

Finally, although there are many more advantages we could list, there are thousands and thousands of free programs and apps available that cover virtually every aspect of computing. Known as packages, there are (at the time of writing) over 8,700 specific programming applications just on Linux Mint alone, and an incredible 62,000+ overall packages catering from Amateur Radio to WWW tools.

Linux, therefore, is a great resource and environment in which to program. It's perfectly suited for developers and it's continually improving and evolving. If you're serious about getting into coding, or you just want to explore something new, give Linux a try and see how it works for you.



A Linux programming environment can be as simple or as complex as you need it to be.

Using the Filesystem

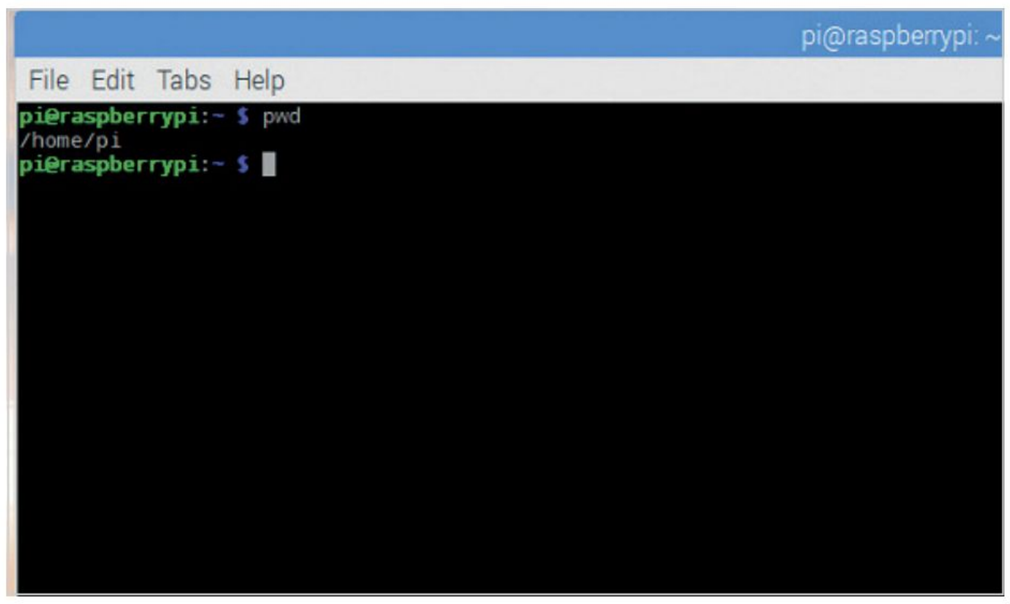
To master Linux, it's important to understand how the filesystem works. What's more, it's also important to become familiar with the Terminal, or shell. This command line environment may appear daunting at first, but with practise, it soon becomes easy to use.

GETTING AROUND

To drop into the Terminal, click on the fourth icon from the left along the top of the Raspberry Pi desktop, the one with a right-facing arrow and an underscore. This is the shell, or Terminal.

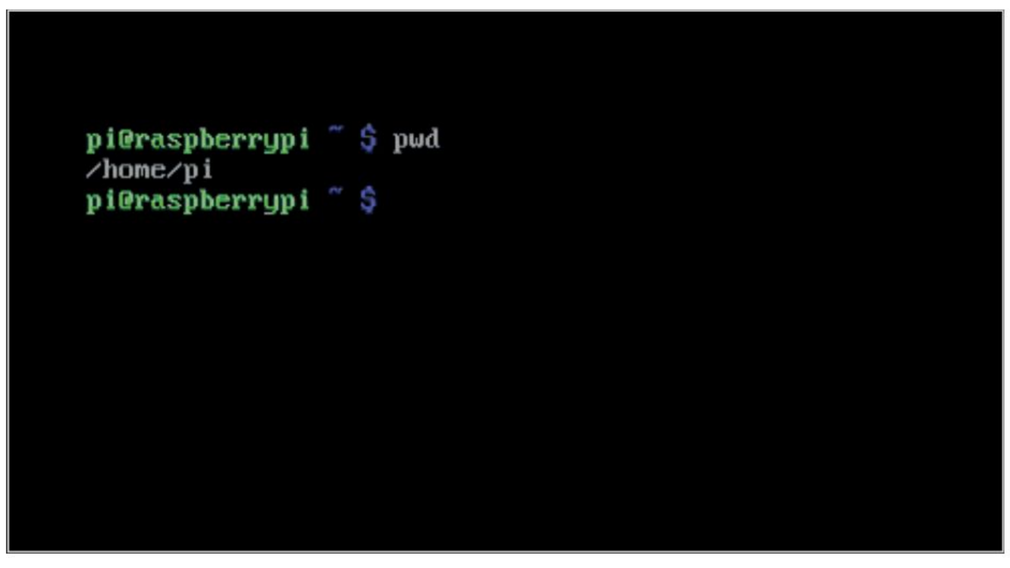
STEP 1

First, you're going to look at directories and the directory path. A directory is the same thing as a folder, however in Linux it's always called a directory. These are placed inside each other using a "/" character. So when you see /home/pi it means the pi directory is inside the home directory. Enter: `clear` and press return to clean the screen. Now enter: `pwd`. This stands for Print Working Directory and displays /home/pi..



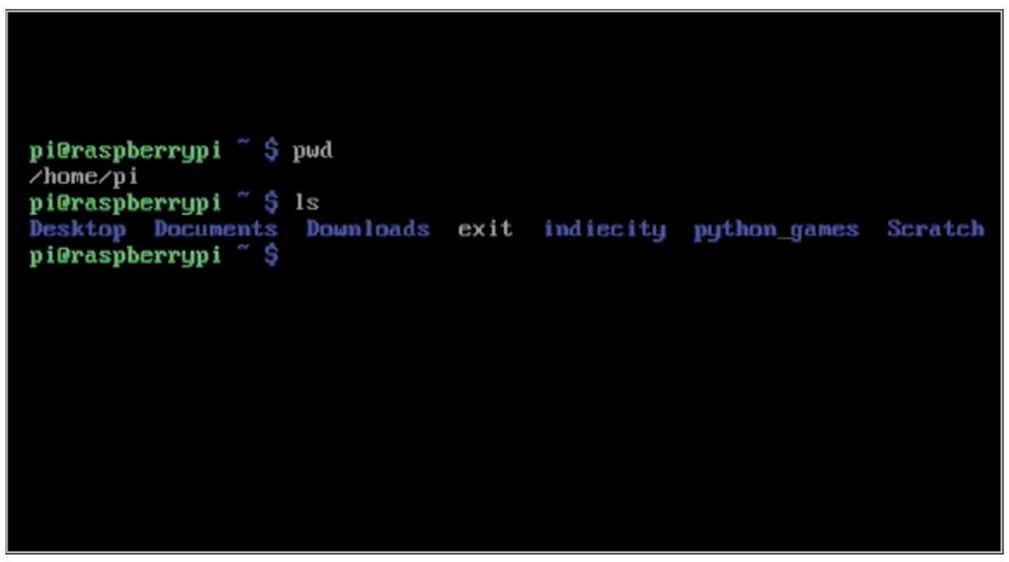
STEP 2

When you log in to your Raspberry Pi, you don't start at the base of the hard drive, known as the 'root' (also known as the topmost directory). Instead you begin inside your user directory, which is named 'pi' by default and is itself in a directory called 'home'. Directories are indicated by the '/' symbol. So, "'/home/pi'" tells you that in the root is a directory called home, and the next "'/'" says that inside "home" is a directory called "pi". That's where you start.



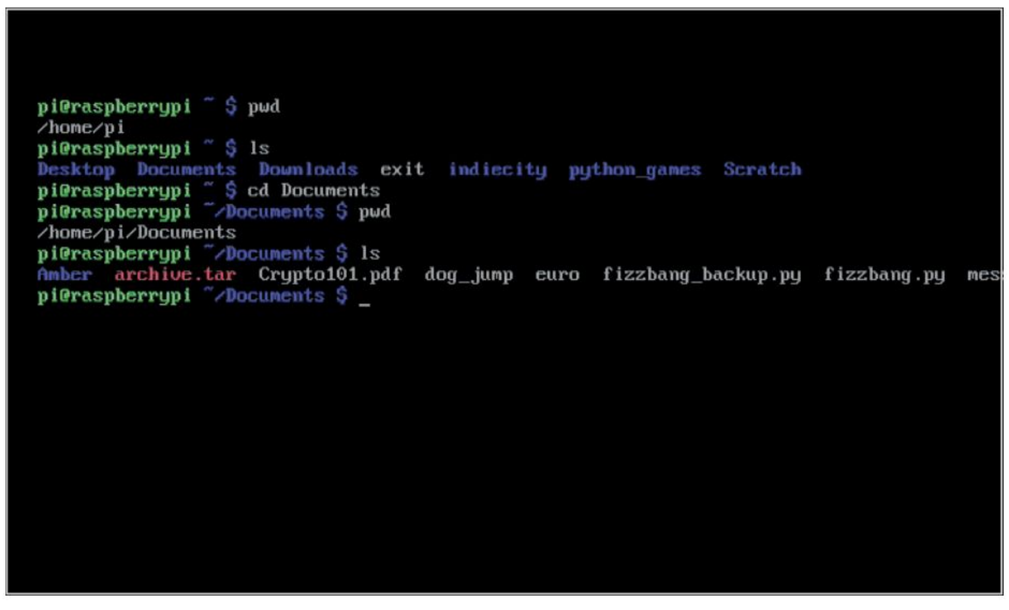
STEP 3

Enter: `ls` to view the contents of the current directory. You should see Desktop, Documents, and Downloads and Scratch in Blue. You may also see other items depending on how much you have used your Raspberry Pi. The colour code is worth knowing: directories are blue while most files are white. As you go on you'll see other colours: executable files (programs) are bright green, archived files are red and so on. Blue and white are the two you need to know to get started.



STEP 4

Now you're going to move from the pi directory into the Documents directory. Enter: `cd Documents`. Note the capital "D". Linux is case sensitive, which means you have to enter the exact name including correct capitalisation. The cd command stands for change directory. Now enter: `pwd` again to view the directory path. It will display /home/pi/ Documents. Enter: `ls` to view the files inside the Documents directory.



STEP 5 How do you get back up to the pi directory? By using a command "cd ..". In Linux two dots means the directory above, also known as the parent directory. Incidentally, a single dot "." is used for the same directory. You never use "cd ." to switch to the same directory but it's worth knowing because some commands need you to specify the current directory.

```
pi@raspberrypi ~/Documents $ pwd
/home/pi/Documents
pi@raspberrypi ~/Documents $ cd ..
pi@raspberrypi ~ $ pwd
/home/pi
pi@raspberrypi ~ $
```

STEP 6 The "ls" and "cd" commands can also be used with more complex paths. Enter: `ls Documents/Pictures` to view the contents of a Pictures directory inside your Documents directory. You can switch to this directory using `cd Documents/Pictures`; use `cd ../..` to move back up two parent directories.

```
pi@raspberrypi ~ $ ls Documents/Pictures
LEGO  LucyHattersley.jpg  raspberry_pi_2_photographs
pi@raspberrypi ~ $ cd Documents/Pictures
pi@raspberrypi ~/Documents/Pictures $ pwd
/home/pi/Documents/Pictures
pi@raspberrypi ~/Documents/Pictures $ cd ../..
pi@raspberrypi ~ $ pwd
/home/pi
pi@raspberrypi ~ $
```

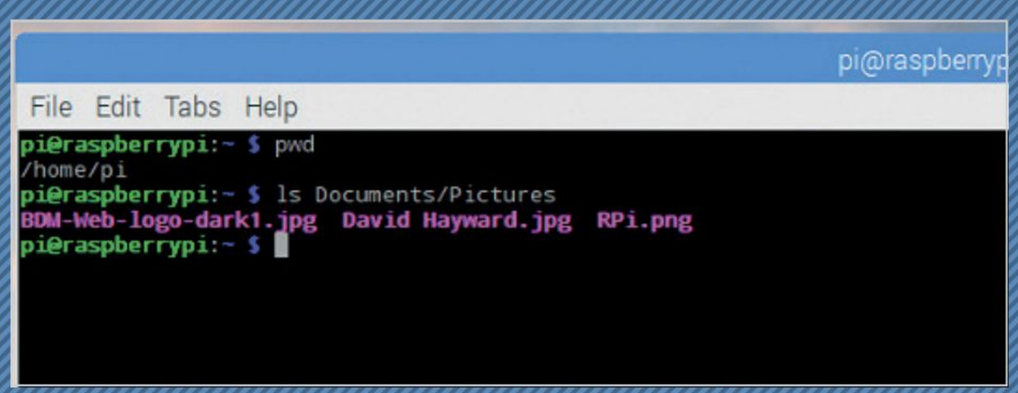
ABSOLUTE VS RELATIVE PATHS

It is important to know the difference between the working directory, root directory and home. There are also two types of path: Absolute and Relative. These are easier to understand than they sound. Let's take a look...

STEP 1 By default, commands like "ls" use the working directory. This is the current directory that you're looking at and is set to your home directory by default (/users/pi). Using "pwd" (Print Working Directory) lets you know what the working directory is, and using "cd" changes the working directory.

```
pi@raspberrypi ~ $ pwd
/home/pi
pi@raspberrypi ~ $
```

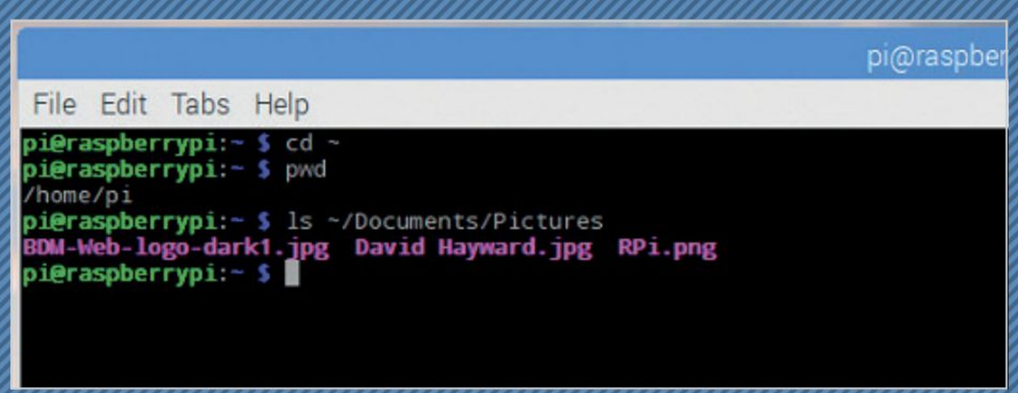
STEP 2 The root directory is always '/'. Entering: `ls /` lists the contents of root, and entering: `cd /` switches to the root directory. This is important because there is a difference between "ls Documents/Pictures" and "ls /Documents/Pictures". The first command lists the contents of the Pictures directory in Documents inside the working directory (which, if you are in the home directory, will work).



STEP 3 The second command ("ls /Documents/Pictures") attempts to list the content of Pictures in a directory called Documents inside the root directory (because the path started with '/', which is root). There is typically no Documents directory in root, so you will get a "No such file or directory" error. Starting a path with '/' is known as an "absolute path", while starting without the '/' is known as a "relative path" because it is relative to your working directory.

```
pi@raspberrypi ~ $ ls /
bin  boot  dev  etc  home  lib  lost+found  media  mnt  opt  proc  r
pi@raspberrypi ~ $ ls /Documents/Pictures
ls: cannot access /Documents/Pictures: No such file or directory
pi@raspberrypi ~ $ _
```

STEP 4 There is also an absolute path shortcut to your user directory, and that is the tilde "~" character. Entering: `ls ~` always lists the contents of your home directory, while "cd ~" moves straight to your home directory, no matter what your working directory is. You can also use this shortcut wherever you are: enter: `ls ~/Documents/Pictures` to display the contents of the Pictures.



Listing and Moving Files

Admittedly, using the desktop GUI to list and move files is much easier than using the Terminal and keyboard. However, it's an important skill that you will appreciate as you advance with the Raspberry Pi and Linux.

LOOKING AT FILES

Operating systems are built on files and folders, or directories if you prefer. While you're used to viewing your own files, most operating systems keep other files out of sight. In Raspbian, you have access to every file in the system.

STEP 1

We've already looked at "ls", which lists the files in the working directory, but you are more likely to use a command like "ls –l". The bit after the command (the '-lah') is known as the argument. This is an option that modifies the behaviour of the command.

```
pi@raspberrypi ~ $ ls -l_
```

STEP 2

The "-l" argument lists files and directories in long format. Each file and directory is now on a single line, and before each file is a lot of text. First you'll see lots of letters and dashes, like 'drwxr-xr-x'. Don't worry about these for now; they are known as 'permissions' and we'll come to those later.

```
pi@raspberrypi ~ $ ls -l
total 24
-rw-r--r-- 1 pi pi    0 May 11 20:56 articles.txt
drwxr-xr-x 2 pi pi 4096 Apr 21 17:55 Desktop
drwxr-xr-x 5 pi pi 4096 Apr 21 14:50 Documents
drwx------ 2 pi pi 4096 Apr 21 15:23 Downloads
drwxr-xr-x 3 pi pi 4096 Apr 17 18:48 indiecity
-rw-r--r-- 1 pi pi    0 May 11 20:56 names.txt
drwxrwxr-x 2 pi pi 4096 Jan  1  1970 python_games
drwxr-xr-x 2 pi pi 4096 Apr 17 12:53 Scratch
pi@raspberrypi ~ $
```

STEP 3

After the permission letters come a single number. This is the number of files in the item. If it's a file then it'll be 1, but if it's a directory it'll be at least 2. This is because each directory contains two hidden files; one with a single dot (.) and one with two dots (..). Directories containing files or other directories will have a higher number.

```
pi@raspberrypi ~ $ ls -l
total 24
-rw-r--r-- 1 pi pi    0 May 11 20:56 articles.txt
drwxr-xr-x 2 pi pi 4096 Apr 21 17:55 Desktop
drwxr-xr-x 5 pi pi 4096 Apr 21 14:50 Documents
drwx------ 2 pi pi 4096 Apr 21 15:23 Downloads
drwxr-xr-x 3 pi pi 4096 Apr 17 18:48 indiecity
-rw-r--r-- 1 pi pi    0 May 11 20:56 names.txt
drwxrwxr-x 2 pi pi 4096 Jan  1  1970 python_games
drwxr-xr-x 2 pi pi 4096 Apr 17 12:53 Scratch
pi@raspberrypi ~ $
```

STEP 4

Next you'll see the word "pi" listed twice on each line. This refers to the user rather than the name of your computer (your default username is "pi"). The first is the owner of the file, and the second is the group. Typically these will both be the same and you'll see either 'pi' or 'root'. You can enter: `ls -l /` to view the files and directories in the root directory that belong to the root account.

```
pi@raspberrypi ~ $ ls -l
total 28
-rw-r--r-- 1 pi pi    0 May 11 20:56 articles.txt
drwxr-xr-x 2 pi pi 4096 Apr 21 17:55 Desktop
drwxr-xr-x 5 pi pi 4096 Apr 21 14:50 Documents
drwx------ 2 pi pi 4096 Apr 21 15:23 Downloads
drwxr-xr-x 3 pi pi 4096 Apr 17 18:48 indiecity
-rw-r--r-- 1 pi pi    0 May 11 20:56 names.txt
drwxrwxr-x 2 pi pi 4096 Jan  1  1970 python_games
drwxr-xr-x 2 pi pi 4096 Apr 17 12:53 Scratch
drwxr-xr-x 3 pi pi 4096 May 11 21:15 test
pi@raspberrypi ~ $ ls -l /
total 74
drwxr-xr-x   2 root root  4096 Jan  1  1970 bin
drwxr-xr-x   3 root root  2048 Jan  1  1970 boot
drwxr-xr-x  12 root root  3280 May 11 09:03 dev
drwxr-xr-x 109 root root  4096 May 11 09:03 etc
drwxr-xr-x   3 root root  4096 Jan  1  1970 home
drwxr-xr-x  12 root root  4096 Jan  1  1970 lib
drwx------   2 root root 16384 Feb 15 11:21 lost+found
drwxr-xr-x   3 root root  4096 May 11 07:42 media
drwxr-xr-x   2 root root  4096 Jan 11 00:02 mnt
drwxr-xr-x   6 root root  4096 Jan  1  1970 opt
```

STEP 5 The next number relates to the size of the file, in bytes. In Linux each text file is made up of letters and each letter takes up a byte, so our names.txt file has 37 bytes and 37 characters in the document. Files and directories can be extremely large and hard to determine, so use "ls – lh". The "h" argument humanises the number, making it easier to read.

```
pi@raspberrypi ~ $ ls -l
total 32
-rw-r--r-- 1 pi pi    0 May 11 20:56 articles.txt
drwxr-xr-x 2 pi pi 4096 Apr 21 17:55 Desktop
drwxr-xr-x 5 pi pi 4096 Apr 21 14:50 Documents
drwx------ 2 pi pi 4096 Apr 21 15:23 Downloads
drwxr-xr-x 3 pi pi 4096 Apr 17 18:48 indiecity
-rw-r--r-- 1 pi pi   37 May 11 21:27 names.txt
drwxrwxr-x 2 pi pi 4096 Jan  1  1970 python_games
drwxr-xr-x 2 pi pi 4096 Apr 17 12:53 Scratch
drwxr-xr-x 3 pi pi 4096 May 11 21:15 test
pi@raspberrypi ~ $ ls -lh
total 32K
-rw-r--r-- 1 pi pi    0 May 11 20:56 articles.txt
drwxr-xr-x 2 pi pi 4.0K Apr 21 17:55 Desktop
drwxr-xr-x 5 pi pi 4.0K Apr 21 14:50 Documents
drwx------ 2 pi pi 4.0K Apr 21 15:23 Downloads
drwxr-xr-x 3 pi pi 4.0K Apr 17 18:48 indiecity
```

STEP 6 Finally, you should be aware that there are many hidden files in Linux. These are listed using the "-a" argument. Hidden files and directories begin with a dot (.), so you should never start a file or directory with a dot, unless you want to hide it. Typically, you can combine all three arguments together into the command "'s –lah".

```
pi@raspberrypi ~ $ ls -lah
total 520K
drwxr-xr-x 33 pi   pi       4.0K May 11 21:14 .
drwxr-xr-x  3 root root     4.0K Jan  1  1970 ..
drwx------  2 pi   pi       4.0K Apr 20 14:31 .aptitude
-rw-r--r--  1 pi   pi          0 May 11 20:56 articles.txt
-rw-------  1 pi   pi       8.7K May 11 09:03 .bash_history
-rw-r--r--  1 pi   pi        220 Feb 15 14:05 .bash_logout
-rw-r--r--  1 pi   pi       3.2K Feb 15 14:05 .bashrc
drwxr-xr-x 10 pi   pi       4.0K Apr 21 17:08 .cache
drwxr-xr-x 20 pi   pi       4.0K Apr 21 13:33 .config
drwx------  3 pi   pi       4.0K Feb 16 14:16 .dbus
drwxr-xr-x  2 pi   pi       4.0K Apr 21 17:55 Desktop
-rw-r--r--  1 pi   pi         35 Apr 17 12:17 .dmrc
drwxr-xr-x  5 pi   pi       4.0K Apr 21 14:50 Documents
drwx------  2 pi   pi       4.0K Apr 21 15:23 Downloads
drwxr-xr-x  2 pi   pi       4.0K Apr 20 13:45 .dreamchess
drwxr-xr-x  2 pi   pi       4.0K Apr 21 18:15 .fontconfig
```

SOME COMMON DIRECTORIES

Now that you know how to view the contents of your hard drive you'll start to notice a lot of directories with names like bin, sbin, var and dev. These are the files and directories that you are kept away from on a Mac, and won't encounter on a Windows PC.

STEP 1 Enter: `ls -lah /` to view all of the files and directories, including the hidden items, in the root directory of your hard drive. Here you will see all the items that make up your Raspbian OS (which is a version of Linux). It's worth taking the time to know some of them.

```
pi@raspberrypi ~ $ ls -lah /
total 82K
drwxr-xr-x  22 root root 4.0K May 11 21:23 .
drwxr-xr-x  22 root root 4.0K May 11 21:23 ..
drwxr-xr-x   2 root root 4.0K Jan  1  1970 bin
drwxr-xr-x   3 root root 2.0K Jan  1  1970 boot
drwxr-xr-x  12 root root 3.3K May 11 09:03 dev
drwxr-xr-x 109 root root 4.0K May 11 09:03 etc
drwxr-xr-x   3 root root 4.0K Jan  1  1970 home
drwxr-xr-x  12 root root 4.0K Jan  1  1970 lib
drwx------   2 root root  16K Feb 15 11:21 lost+found
drwxr-xr-x   3 root root 4.0K May 11 07:42 media
drwxr-xr-x   2 root root 4.0K Jan 11 00:02 mnt
drwxr-xr-x   6 root root 4.0K Jan  1  1970 opt
dr-xr-xr-x  85 root root    0 Jan  1  1970 proc
drwx------   9 root root 4.0K May 11 07:36 root
drwxr-xr-x  10 root root  460 May 11 09:03 run
drwxr-xr-x   2 root root 4.0K Jan  1  1970 sbin
drwxr-xr-x   2 root root 4.0K Jun 20  2012 selinux
```

STEP 3 Entering: `ls /home` displays the contents of your home directory, which contains pi; the directory that you start in. So, entering: `ls/home/pi` is the same as just "ls" from the default home directory. This is where you are expected to place most of the documents you create. Don't confuse home with "usr"; the /usr directory is where find you find program tools and libraries.

```
pi@raspberrypi ~ $ ls
articles.txt  Desktop  Documents  Downloads  indiecity  names.txt  python_g
pi@raspberrypi ~ $ ls /home/pi
articles.txt  Desktop  Documents  Downloads  indiecity  names.txt  python_g
pi@raspberrypi ~ $
```

STEP 2 Bin is a directory that stores binaries. This is the Linux way of saying programs or applications. Sbin is for system binaries, which are the programs that make up your system. Dev contains references to your devices: hard drive, keyboard, mouse and so on. Etc contains your system configuration files.

```
pi@raspberrypi ~ $ ls /bin
bash      bzfgrep      chgrp      dash          domainname  fgconsole  gze
bunzip2   bzgrep       chmod      date          dumpkeys    fgrep      gzi
bzcat     bzip2        chown      dd            echo        findmnt    hos
bzcmp     bzip2recover chvt       df            ed          fuser      ip
bzdiff    bzless       con2fbmap  dir           egrep       fusermount kbd
bzegrep   bzmore       cp         dmesg         false       grep       kil
bzexe     cat          cpio       dnsdomainname fbset       gunzip     kmo
pi@raspberrypi ~ $
```

STEP 4 Lib is a directory that contains libraries of code that are referred to by other programs (different programs share files in Lib). "Var" is short for various, which is mostly files used by the system, but you may need to work with items here. Finally there is a directory called "tmp", which is for temporary files; files placed here are on your system for the short term and can be deleted from the system.

```
pi@raspberrypi ~ $ ls /var
backups  cache  lib  local  lock  log  mail  opt  run  spool  swap  tmp  ww
pi@raspberrypi ~ $
```

Creating and Deleting Files

Being able to create and delete a file is an everyday computing skill. However, when using the Linux Terminal, there's an element of care required, chiefly because any deleted files aren't placed in the system recycle bin.

CREATING FILES

Once you learn to recognise the files and directories that make up Raspbian OS, it's time to discover how to make your own. Knowing how to make, edit and delete files and directories is essential if you want to make your own projects.

STEP 1

We're going to create a file using a command called Touch. Touch is an interesting command that reaches out to a file, or directory, and updates it (this changes the system time as if you'd just opened the file). You can see Touch in access using "ls –l" and checking the time next to a directory (such as Scratch).

```
pi@raspberrypi ~ $ ls -l
total 24
drwxr-xr-x 2 pi pi 4096 Apr 21 17:55 Desktop
drwxr-xr-x 5 pi pi 4096 May 13 10:57 Documents
drwx------ 2 pi pi 4096 May 13 11:01 Downloads
drwxr-xr-x 3 pi pi 4096 Apr 17 18:48 indiecity
drwxrwxr-x 2 pi pi 4096 Jan  1  1970 python_games
drwxr-xr-x 2 pi pi 4096 Apr 17 12:53 Scratch
pi@raspberrypi ~ $ _
```

STEP 2

Now enter: `touch Scratch` and `ls -l` again and notice that the time has changed. It now matches the current time. You might be wondering what this has to do with creating files or directories. Touch has a second, more popular, use, which is to create files.

```
pi@raspberrypi ~ $ ls -l
total 24
drwxr-xr-x 2 pi pi 4096 Apr 21 17:55 Desktop
drwxr-xr-x 5 pi pi 4096 May 13 10:57 Documents
drwx------ 2 pi pi 4096 May 13 11:01 Downloads
drwxr-xr-x 3 pi pi 4096 Apr 17 18:48 indiecity
drwxrwxr-x 2 pi pi 4096 Jan  1  1970 python_games
drwxr-xr-x 2 pi pi 4096 Apr 17 12:53 Scratch
pi@raspberrypi ~ $ touch Scratch
pi@raspberrypi ~ $ ls -l
total 24
drwxr-xr-x 2 pi pi 4096 Apr 21 17:55 Desktop
drwxr-xr-x 5 pi pi 4096 May 13 10:57 Documents
drwx------ 2 pi pi 4096 May 13 11:01 Downloads
drwxr-xr-x 3 pi pi 4096 Apr 17 18:48 indiecity
drwxrwxr-x 2 pi pi 4096 Jan  1  1970 python_games
drwxr-xr-x 2 pi pi 4096 May 13 11:05 Scratch
pi@raspberrypi ~ $ _
```

STEP 3

If you try to touch a file that doesn't exist, you create a blank file with that name. Try it now. Type `touch testfile` and `ls -l` to view the files. You'll now have a new file in your home directory called "testfile". Notice that the size of the file is 0, because it has nothing in it.

```
pi@raspberrypi ~ $ touch testfile
pi@raspberrypi ~ $ ls -l
total 24
drwxr-xr-x 2 pi pi 4096 Apr 21 17:55 Desktop
drwxr-xr-x 5 pi pi 4096 May 13 10:57 Documents
drwx------ 2 pi pi 4096 May 13 11:01 Downloads
drwxr-xr-x 3 pi pi 4096 Apr 17 18:48 indiecity
drwxrwxr-x 2 pi pi 4096 Jan  1  1970 python_games
drwxr-xr-x 2 pi pi 4096 May 13 11:05 Scratch
-rw-r--r-- 1 pi pi    0 May 13 11:10 testfile
pi@raspberrypi ~ $ _
```

STEP 4

A quick word about file names: remember that Linux is case sensitive, so if you now enter: `touch Testfile` (with a capital T), it doesn't update 'testfile'; instead, it creates a second file called 'Testfile'. Enter: `ls -l` to see both files. This is confusing, so most people stick with using lowercase letters at all times.

```
pi@raspberrypi ~ $ touch testfile
pi@raspberrypi ~ $ ls -l
total 24
drwxr-xr-x 2 pi pi 4096 Apr 21 17:55 Desktop
drwxr-xr-x 5 pi pi 4096 May 13 10:57 Documents
drwx------ 2 pi pi 4096 May 13 11:01 Downloads
drwxr-xr-x 3 pi pi 4096 Apr 17 18:48 indiecity
drwxrwxr-x 2 pi pi 4096 Jan  1  1970 python_games
drwxr-xr-x 2 pi pi 4096 May 13 11:05 Scratch
-rw-r--r-- 1 pi pi    0 May 13 11:08 testfile
pi@raspberrypi ~ $ touch Testfile
pi@raspberrypi ~ $ ls -l
total 24
drwxr-xr-x 2 pi pi 4096 Apr 21 17:55 Desktop
drwxr-xr-x 5 pi pi 4096 May 13 10:57 Documents
drwx------ 2 pi pi 4096 May 13 11:01 Downloads
drwxr-xr-x 3 pi pi 4096 Apr 17 18:48 indiecity
drwxrwxr-x 2 pi pi 4096 Jan  1  1970 python_games
drwxr-xr-x 2 pi pi 4096 May 13 11:05 Scratch
-rw-r--r-- 1 pi pi    0 May 13 11:08 testfile
-rw-r--r-- 1 pi pi    0 May 13 11:10 Testfile
pi@raspberrypi ~ $ _
```

STEP 5 Another important thing to know is never to use a space in your file names. If you try to enter: `touch test file`, you create a document called "test" and another called "file". Technically there are ways to create files containing a space but you should always use an underscore character ("_") instead of a space, such as "touch test_file".

```
pi@raspberrypi ~ $ touch test file
pi@raspberrypi ~ $ ls -l
total 24
drwxr-xr-x 2 pi pi 4096 Apr 21 17:55 Desktop
drwxr-xr-x 5 pi pi 4096 May 13 10:57 Documents
drwx------ 2 pi pi 4096 May 13 11:01 Downloads
-rw-r--r-- 1 pi pi    0 May 13 11:15 file
drwxr-xr-x 3 pi pi 4096 Apr 17 18:48 indiecity
drwxrwxr-x 2 pi pi 4096 Jan  1  1970 python_games
drwxr-xr-x 2 pi pi 4096 May 13 11:05 Scratch
-rw-r--r-- 1 pi pi    0 May 13 11:15 test
-rw-r--r-- 1 pi pi    0 May 13 11:10 testfile
-rw-r--r-- 1 pi pi    0 May 13 11:12 Testfile
pi@raspberrypi ~ $ _
```

STEP 6 Here are some other files names to avoid: #%&{}\<>*?/$!'":@+`|=. The full stop (.) is used to create an extension to a file; usually used to indicate a file type, such as textfile.txt or compressedfile.zip, and starting a file with a full stop makes it invisible. Don't use full stop in place of a space though; stick to underscores.

```
pi@raspberrypi ~ $ touch don't.use{odd}symbols&in<filenames>or=you'll^confu
```

REMOVING FILES

We've created some files that we don't want, so how do we go about removing them? It turns out that deleting files in your Raspberry Pi is really easy, which may be a problem, so be careful.

STEP 1 Enter: `ls -l` to view the files in your home directory. If you've followed the steps before then you should have three files: "test", "testfile", and "Testfile". We're going to get rid of these items because they were created as an example.

```
pi@raspberrypi ~ $ ls -l
total 24
drwxr-xr-x 2 pi pi 4096 Apr 21 17:55 Desktop
drwxr-xr-x 5 pi pi 4096 May 13 10:57 Documents
drwx------ 2 pi pi 4096 May 13 11:01 Downloads
-rw-r--r-- 1 pi pi    0 May 13 11:15 file
drwxr-xr-x 3 pi pi 4096 Apr 17 18:48 indiecity
drwxrwxr-x 2 pi pi 4096 Jan  1  1970 python_games
drwxr-xr-x 2 pi pi 4096 May 13 11:05 Scratch
-rw-r--r-- 1 pi pi    0 May 13 11:15 test
-rw-r--r-- 1 pi pi    0 May 13 11:10 testfile
-rw-r--r-- 1 pi pi    0 May 13 11:46 Testfile
pi@raspberrypi ~ $
```

STEP 2 To get rid of files you use the "rm" command. Enter: `rm Testfile` to delete the file called "Testfile" (with the uppercase "t"). Enter: `ls -l` and you'll find it's gone. Where is it? It's not in the Trash or Recycle Bin, like on a Mac or Windows PC. It's deleted completely and cannot be recovered. Bear this in mind and always think before deleting files.

```
pi@raspberrypi ~ $ rm Testfile
pi@raspberrypi ~ $ ls -l
total 24
drwxr-xr-x 2 pi pi 4096 Apr 21 17:55 Desktop
drwxr-xr-x 5 pi pi 4096 May 13 10:57 Documents
drwx------ 2 pi pi 4096 May 13 11:01 Downloads
-rw-r--r-- 1 pi pi    0 May 13 11:15 file
drwxr-xr-x 3 pi pi 4096 Apr 17 18:48 indiecity
drwxrwxr-x 2 pi pi 4096 Jan  1  1970 python_games
drwxr-xr-x 2 pi pi 4096 May 13 11:05 Scratch
-rw-r--r-- 1 pi pi    0 May 13 11:15 test
-rw-r--r-- 1 pi pi    0 May 13 11:10 testfile
pi@raspberrypi ~ $ _
```

STEP 3 We're going to use a wildcard (*) to delete our next two files, but again this is something you really need to do with care. First use "ls" to list the files and make sure it's the one you want to delete. Enter: `ls test*` to view files that match the word "test" and any other characters. The "*" character is called a "wildcard" and it means any characters here.

```
pi@raspberrypi ~ $ ls -l
total 24
drwxr-xr-x 2 pi pi 4096 Jul  9 08:36 Desktop
drwxr-xr-x 2 pi pi 4096 Jul  9 08:36 Documents
drwxr-xr-x 2 pi pi 4096 Jul  9 08:36 Downloads
-rw-r--r-- 1 pi pi    0 Jul  9 08:37 file
drwxr-xr-x 2 pi pi 4096 Jul  9 08:36 indiecity
drwxrwxr-x 2 pi pi 4096 Jan  1  1970 python_games
drwxr-xr-x 2 pi pi 4096 Jul  9 08:36 Scratch
-rw-r--r-- 1 pi pi    0 Jul  9 08:37 test
-rw-r--r-- 1 pi pi    0 Jul  9 08:37 testfile
pi@raspberrypi ~ $ ls test*
test  testfile
pi@raspberrypi ~ $ _
```

STEP 4 We see that "ls test*" matches two files: "test" and "testfile", but not the file called "file". That's because it didn't match the "test" part of "test*". Check carefully over groups of files you want to remove (remember you can't recover them) and replace the "ls" with "rm". Enter: `rm test*` to remove both files. Finally enter: `rm file` to get rid of the confusing file.

```
pi@raspberrypi ~ $ rm test*
pi@raspberrypi ~ $ ls -l
total 24
drwxr-xr-x 2 pi pi 4096 Jul  9 08:36 Desktop
drwxr-xr-x 2 pi pi 4096 Jul  9 08:36 Documents
drwxr-xr-x 2 pi pi 4096 Jul  9 08:36 Downloads
-rw-r--r-- 1 pi pi    0 Jul  9 08:37 file
drwxr-xr-x 2 pi pi 4096 Jul  9 08:36 indiecity
drwxrwxr-x 2 pi pi 4096 Jan  1  1970 python_games
drwxr-xr-x 2 pi pi 4096 Jul  9 08:36 Scratch
pi@raspberrypi ~ $ rm file
pi@raspberrypi ~ $
```

Create and Remove Directories

Creating, moving and deleting directories isn't as easy in the Terminal as it is within a desktop interface. You need to tell Linux to move the directories inside other directories, a process known as recursion. Sounds complex but you should quickly get the hang of it.

MANAGING FILES AND DIRECTORIES

Now that you know how to create files, you'll want to learn how to make directories, which are the same thing as folders, as well as move items around. If you are more used to working with a desktop interface, this can take a bit of getting used to.

STEP 1

Enter: `ls` to quickly view all the directories currently in in the home location. Directories are created using the "mkdir" command (make directory). Enter: `mkdir testdir` to create a new directory in your home directory. Enter: `ls` again to see it.

```
pi@raspberrypi ~ $ ls
Desktop  Documents  Downloads  indiecity  python_games  Scra
pi@raspberrypi ~ $ mkdir testdir
pi@raspberrypi ~ $ ls
Desktop  Documents  Downloads  indiecity  python_games  Scra
pi@raspberrypi ~ $ _
```

STEP 2

The "mkdir" command is different to touch, in that it doesn't update the timestamp if you use it with a directory that already exists. Enter: `mkdir testdir` again and you'll get the error "mkdir: cannot create directory 'testdir: File exists".

```
pi@raspberrypi ~ $ mkdir testdir
mkdir: cannot create directory `testdir': File exists
pi@raspberrypi ~ $ _
```

STEP 3

Like touch, you can create multiple directories at once with the mkdir command. Enter: `mkdir testdir2 testdir3` and enter: `ls`. You'll now find several directories called testdir. Also, like files, you should know this means you can't (and really shouldn't) create directories with spaces. As with files, use an underscore ("_") character instead of a space.

```
pi@raspberrypi ~ $ mkdir testdir2 testdir3
pi@raspberrypi ~ $ ls
Desktop  Documents  Downloads  indiecity  python_games  Scra
pi@raspberrypi ~ $ _
```

STEP 4

You can create directories inside of each other using the directory path. Enter: `mkdir Documents/photos` to create a new directory called "photos" inside your documents directory. The directory has to already exist, though, try to enter: `mkdir articles/reports` and you'll get an error because there is no articles directory.

```
pi@raspberrypi ~ $ ls
Desktop  Documents  Downloads  indiecity  python_games  Scrat
pi@raspberrypi ~ $ mkdir Documents/photos
pi@raspberrypi ~ $ mkdir articles/reports
mkdir: cannot create directory `articles/reports': No such fi
pi@raspberrypi ~ $
```

STEP 5 To create a directory path you need to pass in the "p" option to mkdir (which stands for "parents"). Options, if you remember, come after the command and start with a '-'. So enter: `mkdir -p articles/reports`. Enter: `ls` to view the articles directory, or "ls articles" to view the reports directory sitting inside.

```
pi@raspberrypi ~ $ mkdir -p articles/reports
```

STEP 6 Now you're starting to get a bit more advanced, we're going to just reiterate something. In Linux the command structure is always: command, option and argument, in that order. The command is the function, next are the options (typically single letters starting with "-") and finally the argument (often a file, or directory structure). It's always command, option then argument.

```
pi@raspberrypi ~ $ ls -l articles
total 4
drwxr-xr-x 2 pi pi 4096 May 13 12:36 reports
pi@raspberrypi ~ $ _
```

GETTING RID OF DIRECTORIES

Deleting directories is pretty easy in Linux, along with files, and this can be a problem. It's too easy to delete entire directories containing files and these are instantly removed, not sent to a trash directory. Tread carefully.

STEP 1 We're going to remove one of the directories we created earlier using the "rmdir" command. Enter: `ls` to view the files and directories in the current directory. We'll start by getting rid of one of the test directories. Enter: `rmdir testdir3` and `ls` again to confirm the directory has been removed.

```
pi@raspberrypi ~ $ ls
articles  Desktop  Documents  Downloads  indiecity  python_ga
pi@raspberrypi ~ $ rmdir testdir3_
```

STEP 3 To delete a directory containing files or other directories, you return to the "rm" command used to remove files, only now we need to use the "-R" option (which stands for "recursive".) Using "rm –R" removes all the files and directories to whatever you point it at. Enter: `rm -R articles` to remove the articles directory.

```
pi@raspberrypi ~ $ ls
articles  Desktop  Documents  Downloads  indiecity  python_ga
pi@raspberrypi ~ $ rm -R articles
pi@raspberrypi ~ $ ls
Desktop  Documents  Downloads  indiecity  python_games  Scrat
pi@raspberrypi ~ $
```

STEP 2 Now we'll try to get rid of the articles directory (containing the reports directory). Enter: `rmdir articles` and press return. You'll get an error saying "rmdir: failed to remove 'articles': Directory not empty". This is a puzzler; the rmdir command only removes directories that having nothing in them (no files or other directories).

```
pi@raspberrypi ~ $ rmdir articles
rmdir: failed to remove `articles': Directory not empty
pi@raspberrypi ~ $ _
```

STEP 4 As with multiple files, you can delete multiple directories inside the same directory using the "rm" command with the wildcard character (*). This should be done with care though so use the `-I` option (which stands for "interactive"). This will prompt you before each deletion. Enter: `rm -Ri test*` and press `Y` and `return` to each prompt. It's a good idea to use the `-i` option whenever using the rm command.

```
pi@raspberrypi ~ $ rm -Ri test*
rm: remove directory `testdir'? y
rm: remove directory `testdir2'? y
rm: remove directory `testdir3'? y_
```

Copying, Moving and Renaming Files

Taking command of the Terminal is essential when learning how your Raspberry Pi's operating system works. The copying, moving and renaming of files is equally important, as you'll be doing a lot of this throughout your Pi projects.

USING THE MOVE COMMAND

In Linux, renaming a file is simply moving it from one name to another and copying a file is moving it without deleting the original. Don't panic, it's quite easy to master.

STEP 1 Before we can move anything around, we need to have a few test items in our home directory. Enter: `touch testfile` and `mkdir testdir` to create a test file and test directory in your home directory. Enter: `ls` to check that they are both present.

```
pi@raspberrypi ~ $ touch testfile
pi@raspberrypi ~ $ mkdir testdir
pi@raspberrypi ~ $ ls
Desktop  Documents  Downloads  indiecity  python_games  Scra
pi@raspberrypi ~ $
```

STEP 2 Files and directories are moved using the mv command. This is different to the commands we've looked at so far because it has two arguments (remember Linux command line is command, option, argument). The first argument is the source (the file or directory to be moved) and the second is the destination.

```
pi@raspberrypi ~ $ ls
Desktop  Documents  Downloads  indiecity  python_games  Scrat
pi@raspberrypi ~ $ mv testfile testdir
```

STEP 3 Enter: `mv testfile testdir` and press return to move the testfile document into the testdir directory. Enter: `ls` to see that it's no longer in the home directory, and `ls testdir` to see the testfile now sitting in the testdir directory. Now enter: `mkdir newparent` to create a new directory.

```
pi@raspberrypi ~ $ ls
Desktop  Documents  Downloads  indiecity  python_games  Scrat
pi@raspberrypi ~ $ mv testfile testdir
pi@raspberrypi ~ $ ls
Desktop  Documents  Downloads  indiecity  python_games  Scrat
pi@raspberrypi ~ $ ls testdir
testfile
pi@raspberrypi ~ $
```

STEP 4 Directories with files are moved in the same way. Enter: `mv testdir newparent` to move the testdir directory inside the newparent directory. Let's move into the directory to find the file. Enter: `cd /newparent/testdir` and enter: `ls` to view the testfile sitting inside the directory.

```
pi@raspberrypi ~ $ ls
Desktop  Documents  Downloads  indiecity  python_games  Scrat
pi@raspberrypi ~ $ mkdir newparent
pi@raspberrypi ~ $ mv testdir newparent
pi@raspberrypi ~ $ cd newparent/testdir
pi@raspberrypi ~/newparent/testdir $ ls
testfile
pi@raspberrypi ~/newparent/testdir $
```

STEP 5 Files and directories can be moved up using the double dot ("..") as an argument. Enter: `ls -la` to view your testfile and the single and double dot files. The single dot is the current directory and the double dot is the parent directory. Enter: `mv testfile ..` to move the testfile up into the newparent directory. Enter: `cd ..` to move up to the parent directory.

```
pi@raspberrypi ~ $ cd newparent/testdir
pi@raspberrypi ~/newparent/testdir $ ls
testfile
pi@raspberrypi ~/newparent/testdir $ mv testfile ..
pi@raspberrypi ~/newparent/testdir $ cd
```

STEP 6 You can also move files using longer paths. Enter: `cd ~` to return to the home directory and `mv newparent/testfile newparent/testdir/testfile` to move the testfile from its current location back inside the testdir directory. Enter: `ls newparent/testdir` to view the file back in its current directory.

```
pi@raspberrypi ~/newparent $ cd ~
pi@raspberrypi ~ $ ls
Desktop  Documents  Downloads  indiecity  newparent  python
pi@raspberrypi ~ $ mv newparent/testfile newparent/testdir/
pi@raspberrypi ~ $ ls newparent/testdir
testfile
pi@raspberrypi ~ $ _
```

RENAMING FILES AND DIRECTORIES

The mv command isn't used just to move files; it also serves the purpose of renaming files (effectively it moves it from its old name to a new name). Let's see how to use mv to rename items.

STEP 1 Let's start by making a new test file called "names". Enter: `touch testfile` and then `ls` to make sure the testfile is present. We're going to turn this into a file that contains the names of some people. So let's call it something more appropriate, like "names".

```
pi@raspberrypi ~ $ ls
Desktop  Documents  Downloads  indiecity  newparent  python_
pi@raspberrypi ~ $ mv testfile names
pi@raspberrypi ~ $ ls
Desktop  Documents  Downloads  indiecity  names  newparent
pi@raspberrypi ~ $
```

STEP 2 Enter: `mv testfile names` and `ls`. Now we can see the new "names" file in our directory. The mv command can also be used to rename directories. We should still have our newparent directory in our home directory. Enter: `mv newparent people` to rename the newparent directory. Enter: `ls` to view it.

```
pi@raspberrypi ~ $ touch testfile
pi@raspberrypi ~ $ ls
Desktop  Documents  Downloads  indiecity  newparent  python_
pi@raspberrypi ~ $ mv newparent people
pi@raspberrypi ~ $ ls
Desktop  Documents  Downloads  indiecity  people  python_gam
pi@raspberrypi ~ $
```

STEP 3 You can rename directories inside other directories using paths. Let's rename the testdir directory, which is now inside the people directory. Enter: `mv names/testdir names/friends`. Now enter: `mv names people/friends` to move the names file inside the friends directory.

```
pi@raspberrypi ~ $ ls
Desktop  Documents  Downloads  indiecity  names  people  pyth
pi@raspberrypi ~ $ mv people/testdir people/friends
```

STEP 4 It is easy to overwrite files using the mv command, so if you have files with the same name use the "-n" option, which stands for "no overwrite". Enter: `touch testfile` to create a new file and `mv -n testfile people/friends`. There's no error report though, enter: `ls` and you'll find testfile still there.

```
pi@raspberrypi ~ $ touch testfile
pi@raspberrypi ~ $ mv -n testfile people/friends
pi@raspberrypi ~ $ ls
Desktop  Documents  Downloads  indiecity  people  python_gam
pi@raspberrypi ~ $ ls
Desktop  Documents  Downloads  indiecity  people  python_gam
pi@raspberrypi ~ $ ls people/friends
names  testfile
pi@raspberrypi ~ $
```

Using the Man Pages

Linux comes with man (manual) pages that explain each command and show you all the options you can use. Once you get the hang of reading the man pages, you'll be able to find and do just about anything in Linux.

HEY, MAN!

The man pages are one of the best features of Linux, and as a built-in tool it's invaluable for both beginner and senior level Linux administrators. Let's see how it works

STEP 1 Linux has a built-in manual, known as man for short. Using the man command you can obtain information on all the Linux commands we've talked about. Simply enter: `man` and the name of the command you want to learn more about. Start by entering: `man ls` in the command line.

```
pi@raspberrypi ~ $ man ls
```

STEP 2 The man pages are a bit more detailed than you might be used to. First you have a name, which tells you what the command is called; in this case "list directory contents" and then the synopsis shows you how it works. In this case: "ls [OPTION].. [FILE..]". So you enter: `ls` followed by options (such as `-la`) and the file or directory to list.

```
LS(1)

NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...

DESCRIPTION
       List information about the FILEs (the current directory by default).  Sort entries alphabetical

       Mandatory arguments to long options are mandatory for short options too.

       -a, --all
              do not ignore entries starting with .

       -A, --almost-all
              do not list implied . and ..

       --author
              with -l, print the author of each file

       -b, --escape
              print C-style escapes for nongraphic characters

       --block-size=SIZE
              scale sizes by SIZE before printing them.  E.g., `--block-size=M' prints sizes in units

       -B, --ignore-backups
              do not list implied entries ending with ~

       -c     with -lt: sort by, and show, ctime (time of last modification of file status information

       -C     list entries by columns

       --color[=WHEN]
              colorize the output.  WHEN defaults to `always' or can be `never' or `auto'.  More info

       -d, --directory
              list directory entries instead of contents, and do not dereference symbolic links

       -D, --dired
              generate output designed for Emacs' dired mode

       -f     do not sort, enable -aU, disable -ls --color

       -F, --classify
              append indicator (one of */=>@|) to entries

       --file-type
              likewise, except do not append `*'

       --format=WORD
              across -x, commas -m, horizontal -x, long -l, single-column -1, verbose -l, vertical -C

       --full-time
              like -l --time-style=full-iso

 Manual page ls(1) line 1 (press h for help or q to quit)
```

STEP 3 Most commands are pretty easy to figure out how to use, so what you spend most of the time in the man pages is looking under the Description. Here you will see all the options and the letters used to activate them. Most man pages are longer than a single page, so press any key, such as the space bar, to move to the next page of content.

```
-g     like -l, but do not list owner

--group-directories-first
       group directories before files.

       augment with a --sort option, but any use of --sort=none (-U) disables grouping

-G, --no-group
       in a long listing, don't print group names

-h, --human-readable
       with -l, print sizes in human readable format (e.g., 1K 234M 2G)

--si   likewise, but use powers of 1000 not 1024

-H, --dereference-command-line
       follow symbolic links listed on the command line

--dereference-command-line-symlink-to-dir
       follow each command line symbolic link that points to a directory

--hide=PATTERN
       do not list implied entries matching shell PATTERN (overridden by -a or -A)

--indicator-style=WORD
       append indicator with style WORD to entry names: none (default), slash (-p), file-type (--file-type), classify

-i, --inode
       print the index number of each file

-I, --ignore=PATTERN
       do not list implied entries matching shell PATTERN

-k     like --block-size=1K
```

STEP 4 Press the H key while looking at a man page to view the commands you can use to control the view. This is called the Summary of Less Commands (the less command is something we'll come to when we look at editing text). For now realise that you can move back and forward with Z and W. Press Q to quit this help screen and return to the man page.

```
                SUMMARY OF LESS COMMANDS

    Commands marked with * may be preceded by a number, N.
    Notes in parentheses indicate the behavior if N is given.

  h  H                 Display this help.
  q  :q  Q  :Q  ZZ     Exit.
 ---------------------------------------------------------------------------

                           MOVING

  e  ^E  j  ^N  CR  *  Forward  one line   (or N lines).
  y  ^Y  k  ^K  ^P  *  Backward one line   (or N lines).
  f  ^F  ^V  SPACE  *  Forward  one window (or N lines).
  b  ^B  ESC-v      *  Backward one window (or N lines).
  z                 *  Forward  one window (and set window to N).
  w                 *  Backward one window (and set window to N).
  ESC-SPACE         *  Forward  one window, but don't stop at end-of-file.
  d  ^D             *  Forward  one half-window (and set half-window to N).
  u  ^U             *  Backward one half-window (and set half-window to N).
  ESC-)  RightArrow *  Left  one half screen width (or N positions).
  ESC-(  LeftArrow  *  Right one half screen width (or N positions).
  F                    Forward forever; like "tail -f".
  r  ^R  ^L            Repaint screen.
  R                    Repaint screen, discarding buffered input.
        ---------------------------------------------------
        Default "window" is the screen height.
        Default "half-window" is half of the screen height.
 ---------------------------------------------------------------------------

                         SEARCHING

  /pattern          *  Search forward for (N-th) matching line.
  ?pattern          *  Search backward for (N-th) matching line.
  n                 *  Repeat previous search (for N-th occurrence).
  N                 *  Repeat previous search in reverse direction.
  ESC-n             *  Repeat previous search, spanning files.
  ESC-N             *  Repeat previous search, reverse dir. & spanning files.
  ESC-u                Undo (toggle) search highlighting.
```

STEP 5 Scroll to the bottom of the man page to discover more information. Typically you will find the author's name and information on reporting bugs, including web links that can be useful for more information. Press Q to exit the man page and return to the command line.

```
       assume tab stops at each COLS instead of 8

-u     with -lt: sort by, and show, access time with -l: show access time and sort by name otherwise: s

-U     do not sort; list entries in directory order

-v     natural sort of (version) numbers within text

-w, --width=COLS
       assume screen width instead of current value

-x     list entries by lines instead of by columns

-X     sort alphabetically by entry extension

-Z, --context
       print any SELinux security context of each file

-1     list one file per line

--help display this help and exit

--version
       output version information and exit
```

STEP 6 The man command can be used for just about every command you use in Linux. You can even enter: man man to get information on using the man tool. From now on, whenever you come across a new command in this book, such as "nano" or "chmod", take time to enter: `man nano` or `man chmod` and read the instructions.

```
MAN(1)                                                                    Man

NAME
       man - an interface to the on-line reference manuals

SYNOPSIS
       man  [-C  file] [-d] [-D] [--warnings[=warnings]] [-R encoding] [-L locale] [-m system[,...]] [
       [-r prompt] [-7] [-E encoding] [--no-hyphenation] [--no-justification] [-p string] [-t] [-T[dev
       man -k [apropos options] regexp ...
       man -K [-w|-W] [-S list] [-i|-I] [--regex] [section] term ...
       man -f [whatis options] page ...
       man -l [-C file] [-d] [-D] [--warnings[=warnings]] [-R encoding] [-L locale] [-P pager] [-r pro
       man -w|-W [-C file] [-d] [-D] page ...
       man -c [-C file] [-d] [-D] page ...
       man [-hV]

DESCRIPTION
       man is the system's manual pager. Each page argument given to man is normally the name of a pro
       section,  if  provided,  will  direct man to look only in that section of the manual.  The defa
       page found, even if page exists in several sections.

       The table below shows the section numbers of the manual followed by the types of pages they com
```

USING MAN OPTIONS

Because man doesn't change anything, like mv or mkdir, it is tempting not to see it as a command. But it is, and like all other commands it has options. These can be very handy to learn.

STEP 1 Entering: `man man` enables you to view some of the options, but sometimes you'll just want a quick overview. Fortunately man has a built-in help option that quickly lists the options. Press Q if you're in a man page and enter: `man –h` at the command line.

```
                              print physical location of cat file(s)

  -c, --catman                used by catman to reformat out of date cat pages
  -R, --recode=ENCODING       output source page encoded in ENCODING

Finding manual pages:
  -L, --locale=LOCALE         define the locale for this particular man search
  -m, --systems=SYSTEM        use manual pages from other systems
  -M, --manpath=PATH          set search path for manual pages to PATH

  -S, -s, --sections=LIST     use colon separated section list

  -e, --extension=EXTENSION   limit search to extension type EXTENSION

  -i, --ignore-case           look for pages case-insensitively (default)
  -I, --match-case            look for pages case-sensitively

      --regex                 show all pages matching regex
      --wildcard              show all pages matching wildcard

      --names-only            make --regex and --wildcard match page names only,
                              not descriptions

  -a, --all                   find all matching manual pages
  -u, --update                force a cache consistency check

      --no-subpages           don't try subpages, e.g. 'man foo bar' => 'man
                              foo-bar'

Controlling formatted output:
  -P, --pager=PAGER           use program PAGER to display output
  -r, --prompt=STRING         provide the `less' pager with a prompt

  -7, --ascii                 display ASCII translation of certain latin1 chars
  -E, --encoding=ENCODING     use selected output encoding
      --no-hyphenation, --nh turn off hyphenation
      --no-justification,                                --nj   turn off justification
  -p, --preprocessor=STRING   STRING indicates which preprocessors to run:
                              e - [neqn, p - pic, t - tbl,
g - grap, r - refer, v - vgrind

  -t, --troff                 use groff to format pages
  -T, --troff-device[=DEVICE]   use groff with selected device

  -H, --html[=BROWSER]        use www-browser or BROWSER to display HTML output
  -X, --gxditview[=RESOLUTION]   use groff and display through gxditview
                              (X11):
```

STEP 3 One of the most powerful man options is the -k option, which is for "apropos". This enables you to search a wider range of man pages than the exact command. Enter: `man -k directory` to view all of the man pages relating to directories "(man -k directory | less" to view one page at a time). Here you'll find commands like "ls", "mkdir" and "cd" along with their description.

```
fchdir (2)           - change working directory
fchmodat (2)         - change permissions of a file relative to a directory file descriptor
fchownat (2)         - change ownership of a file relative to a directory file descriptor
fdopendir (3)        - open a directory
find (1)             - search for files in a directory hierarchy
fstatat (2)          - get file status relative to a directory file descriptor
fstatat64 (2)        - get file status relative to a directory file descriptor
futimesat (2)        - change timestamps of a file relative to a directory file descriptor
get_current_dir_name (2) - get current working directory
get_current_dir_name (3) - get current working directory
getcwd (2)           - get current working directory
getcwd (3)           - get current working directory
getdents (2)         - get directory entries
getdents64 (2)       - get directory entries
getdirentries (3)    - get directory entries in a file system-independent format
getwd (3)            - get current working directory
git-clone (1)        - Clone a repository into a new directory
git-mv (1)           - Move or rename a file, a directory, or a symlink
git-stash (1)        - Stash the changes in a dirty working directory away
helpztags (1)        - generate the help tags file for directory
linkat (2)           - create a file link relative to directory file descriptors
lookup_dcookie (2)   - return a directory entry's path
ls (1)               - list directory contents
mkdir (2)            - create a directory
mkdirat (2)          - create a directory relative to a directory file descriptor
mkdtemp (3)          - create a unique temporary directory
mkfifoat (3)         - make a FIFO (named pipe) relative to a directory file descriptor
mkfontdir (1)        - create an index of X font files in a directory
mklost+found (8)     - create a lost+found directory on a mounted Linux second extended file system
mknodat (2)          - create a special or ordinary file relative to a directory file descriptor
mktemp (1)           - create a temporary file or directory
modprobe.d (5)       - Configuration directory for modprobe
mountpoint (1)       - see if a directory is a mountpoint
oldfind (1)          - search for files in a directory hierarchy
openat (2)           - open a file relative to a directory file descriptor
opendir (3)          - open a directory
pam_mkhomedir (8)    - PAM module to create users home directory
pwd (1)              - print name of current/working directory
pwdx (1)             - report current working directory of a process
readdir (2)          - read directory entry
readdir (3)          - read a directory
```

STEP 2 If you're fast you may have noticed the start of the text flew up off the page. This is because the "man –h" option doesn't use the less command by default (less is what enables you to move down text one screen at a time). We'll look into pipes ("|") later on, but for now just use "man -h | less" to read long text one page at a time.

```
pi@raspberrypi ~ $ man -h | less
```

STEP 4 Entering the man page for all the commands you come across can be a little long-winded, although ultimately productive. If you simply want to know what a command does you can read just the description using the "whatis" command. Enter: `whatis pwd` to read the description of the "pwd" command ("print name of current/working directory").

```
pi@raspberrypi ~ $ whatis pwd
pwd (1)              - print name of current/working directory
pi@raspberrypi ~ $
```

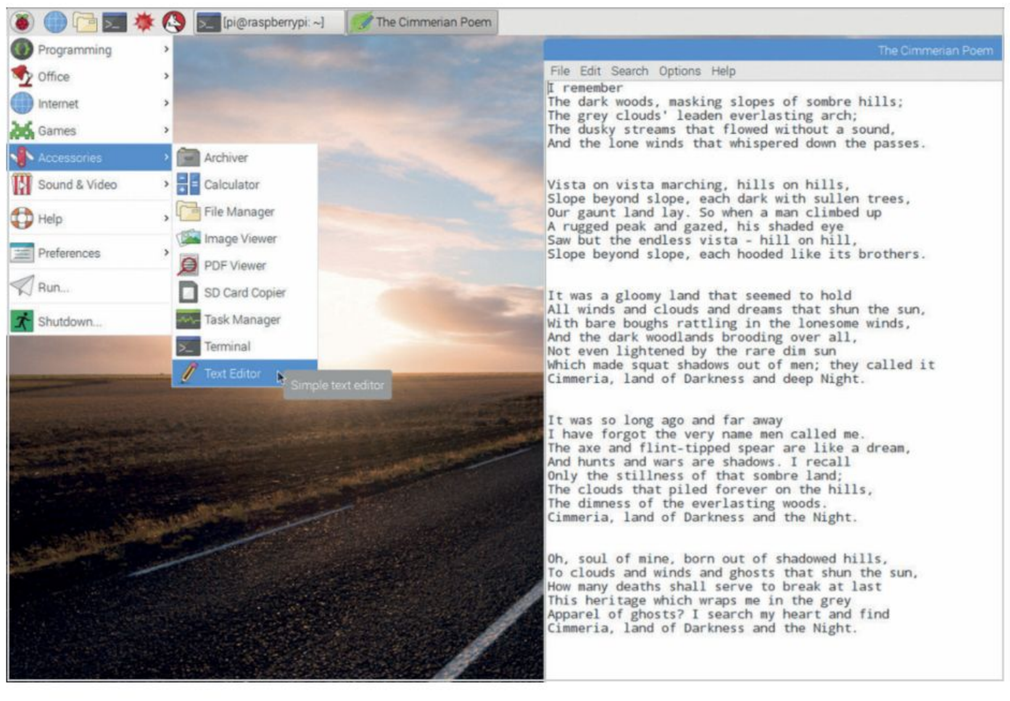
Editing Text Files

A text file in Linux can be anything from a simple set of instructions on how to use an app, to some complex Python, C++ or other programming language code. Text files can be used for scripting, automated executable files, as well as configuration files too.

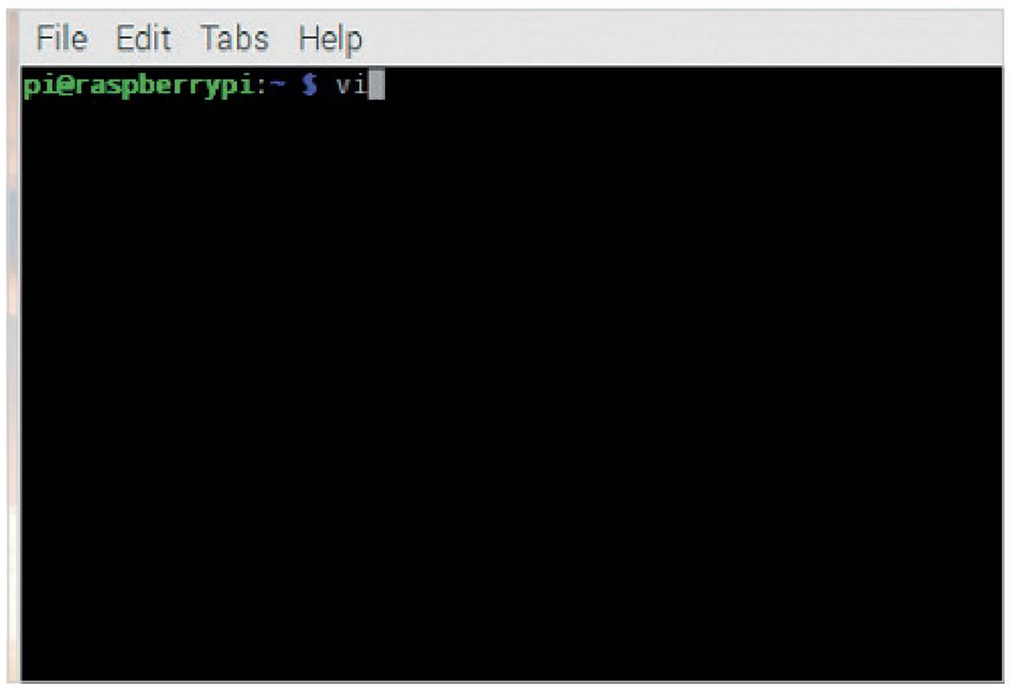
THE JOY OF TEXT

To be able to edit or create a text file, you need a good text editor. Linux has many but here are some in action on the Raspberry Pi.

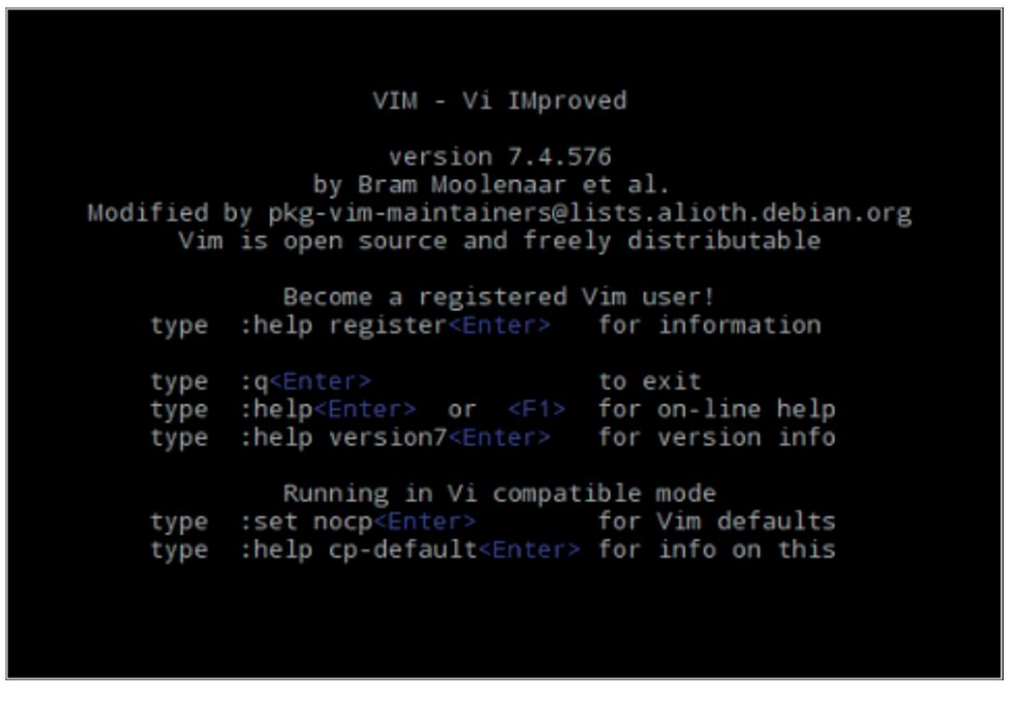
STEP 1 The first text editor for the Raspberry Pi is the default desktop environment app: Leafpad. To use, you can either double-click an existing text file or click the Raspberry Pi menu icon (in the top left of the desktop) and from the Accessories menu, choose Text Editor.



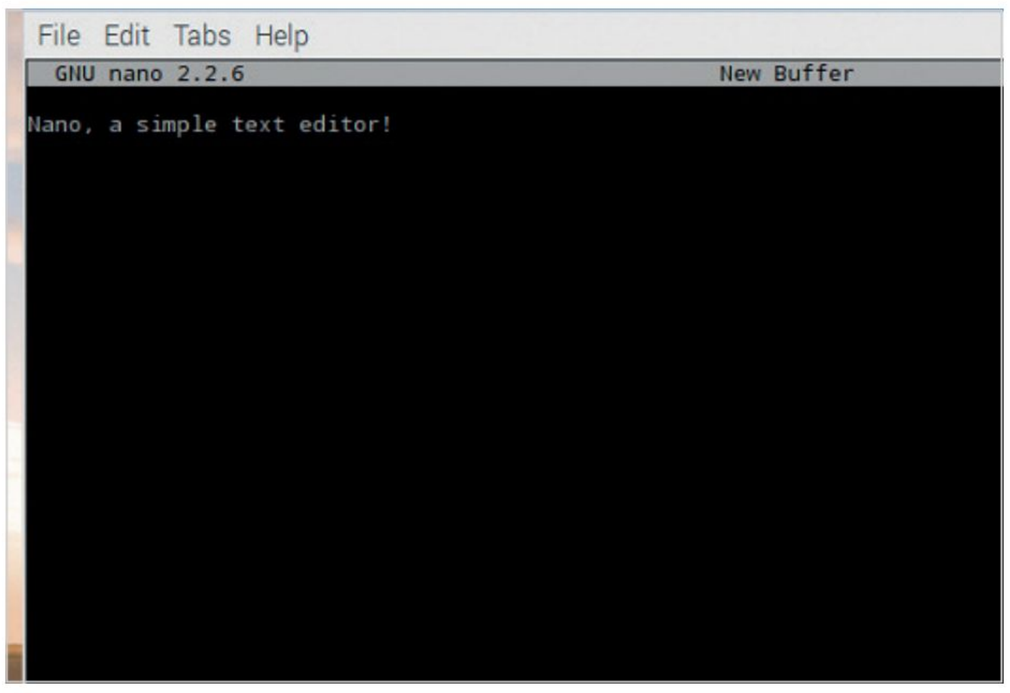
STEP 2 From the Terminal there are even more options, although using the correct command, you can launch any of the desktop apps via the Terminal. One of the simplest, and a classic text editor that's carried over from the Unix days, is vi. In the Terminal, enter: `vi`.



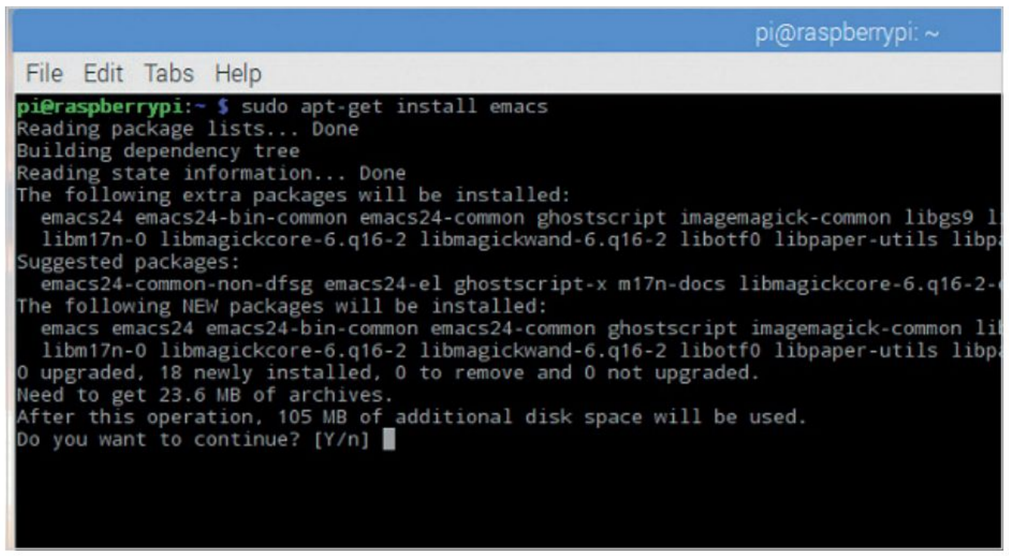
STEP 3 Vi is the original Unix command but in this case it launches VIM, the new Linux version of Vi. Although simple looking, Vi is considered, even by today's standards, to be one of the most widely used text editors, There's a lot you can do with it, so check out the man pages for more Vi information.



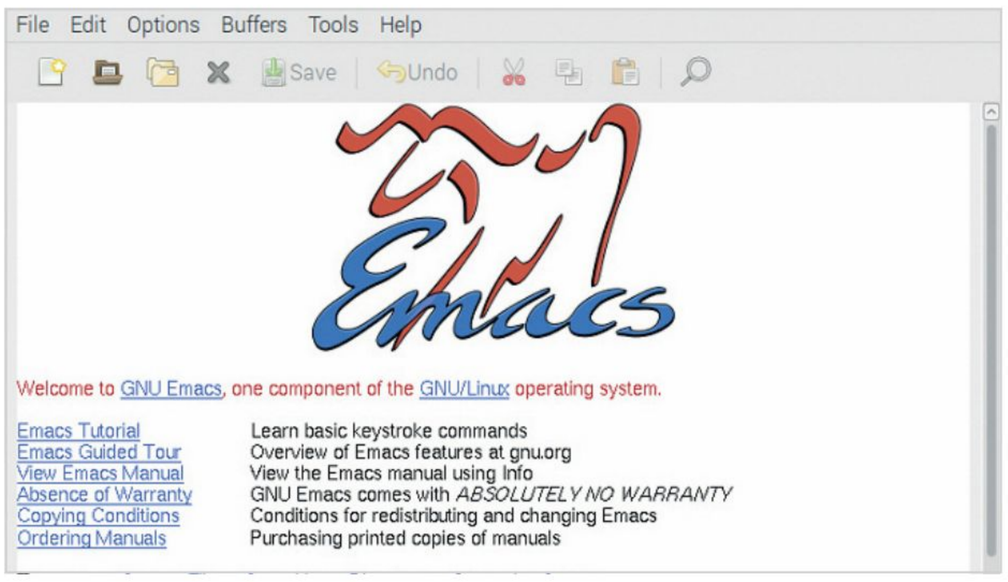
STEP 4 Nano is another favourite, and simple, text editor available for Linux. Enter: `nano` into the Terminal to launch it. You can use Nano for editing code, creating scripts or writing your own help files. To exit Nano, press Ctrl + X, followed by Y to save the file or N to exit without saving.



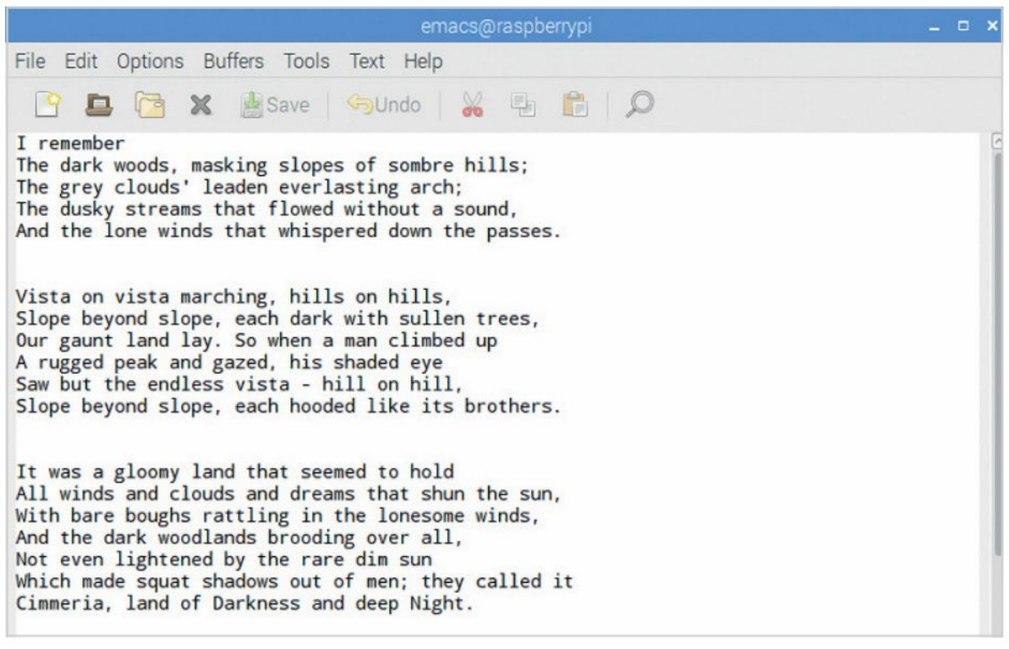
STEP 5 Emacs, or GNU Emacs, is an extensible and customisable, self-documenting, real-time display editor. It's a fantastic text editor and one that's worth getting used to as soon as you can. Sadly, it's not installed on the Pi by default, so you'll need to install it. In the Terminal, enter: `sudo apt-get install emacs`



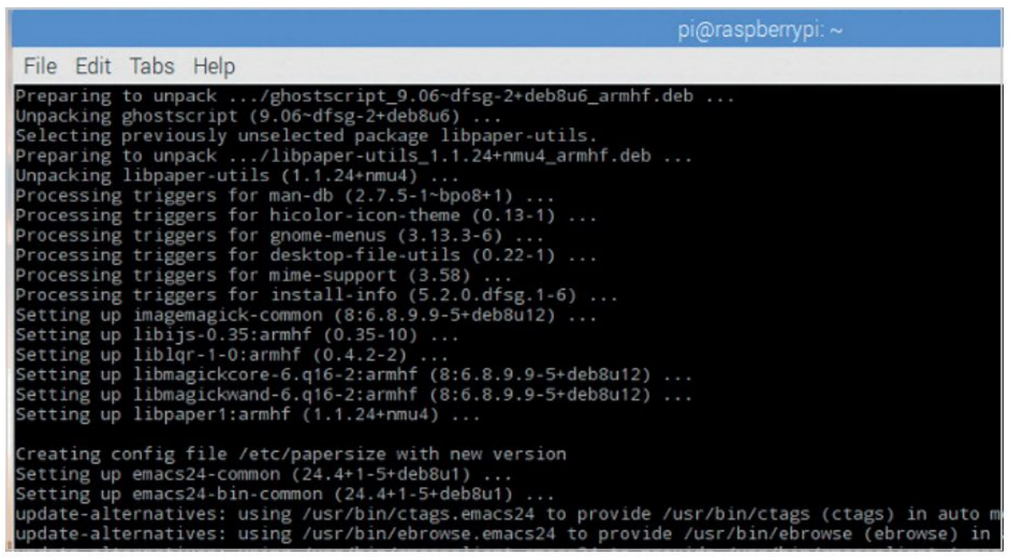
STEP 6 The previous command contacts the Debian (Raspbian is based on a Debian Linux distribution) repositories and pulls down the information needed to install Emacs. When the Pi asks to continue with the installation, press Y. This installs the latest version and when it's done, you'll be back to the command prompt.



STEP 7 Once the installation is complete, enter: `emacs` into the Terminal. The Emacs splash screen opens in a new window, offering a tutorial (which we recommend you run through) and a guided tour amongst other information.

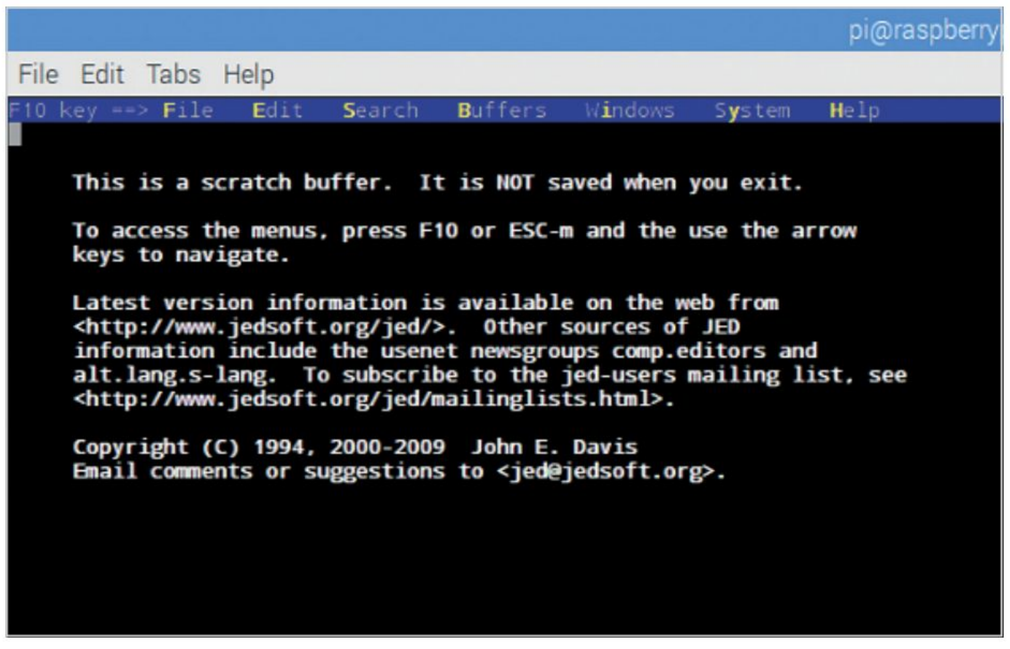


STEP 8 Emacs can offer an uncomplicated view of your text file or one with a plethora of information regarding the structure of the file in question; it's up to you to work out your own preference. There's also a hidden text adventure in Emacs, which we cover later in this book, why not see if you can find it without our help.



STEP 9 Gedit is another excellent text editor for Linux. Again, it's not installed by default on the Raspberry Pi; however, by entering: `sudo apt-get install gedit` and accepting the installation, the program can be on the Pi in a matter of seconds. Once it's installed, use `gedit` in the Terminal to launch it. Gedit is a great text editor for coding.

STEP 10 Finally, Jed is an Emacs-like, cross-platform text editor that's lightweight and comes with a wealth of features. To install it, enter: `sudo apt-get install jed`. Accept the installation and when it's complete, use: `jed` to launch.



Getting to Know Users

You might think you're the only person using your Raspberry Pi but there are several different users and even groups of users. Your main account is normally called Pi and there is an account above it called root, which is more powerful. You can also create users and groups.

WHAT IS A USER?

An important part of using Linux is the concept of users and understanding which user you are and which group you belong to. Like all modern computers, you can have multiple user accounts with each having different levels of access.

STEP 1 The first thing you need to do is get a concept of which user you are. Enter: `whoami` into the command line and press return. It should say "pi" (unless you set up your account name differently during setup). The "whoami" command might seem a bit simplistic, but it comes in very handy sometimes.

```
pi@raspberrypi ~ $ whoami
pi
pi@raspberrypi ~ $ _
```

STEP 2 When you are working in Linux, from time to time a 'Permission denied' error will occur, typically when you try to create, edit or execute (run) a file or directory outside of your area of privilege. If you want to see this, enter: `mkdir /testdir`. Attempting to create a new directory in your root directory isn't allowed.

```
pi@raspberrypi ~ $ mkdir /test
mkdir: cannot create directory `/test': Permission denied
pi@raspberrypi ~ $
```

STEP 3 To allow this, you need to use the sudo command. Sudo loosely stands for Substitute User Do; essentially it's the highest level of access to the system and you've already installed text editors using sudo. You'll come across sudo frequently in Linux, so let's create a second account to get the hang of it. Enter: `sudo useradd -m lucy` (or pick your name).

```
pi@raspberrypi ~ $ sudo useradd -m lucy
```

STEP 4 Now add a password for the new account. Enter: `sudo passwd lucy` and enter: a short password. Retype the same password and you'll now have two accounts on your Raspberry Pi. Now enter: `ls -l /home` to view the home directories for both users. Notice that the lucy directory lists lucy as the owner and group; and pi directory is belongs to pi.

```
pi@raspberrypi ~ $ sudo passwd lucy
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
pi@raspberrypi ~ $ ls -l /home
total 8
drwxr-xr-x  2 lucy lucy 4096 May 13 19:01 lucy
drwxr-xr-x 33 pi   pi   4096 May 13 18:32 pi
pi@raspberrypi ~ $ _
```

STEP 5 Let's try switching to our new account. Enter: `su lucy` and enter the password you just created for that account. Notice that the command line now says "lucy@raspberrypi" but the working directory is "still /home/pi" (check this using "pwd"). Enter: `whoami` to confirm that you are now the new user.

```
lucy@raspberrypi /home/pi $ su lucy
Password:
lucy@raspberrypi /home/pi $ pwd
/home/pi
lucy@raspberrypi /home/pi $ _
```

STEP 6 We'll look at permissions in the next tutorial, but for now try to create a file as before. Enter: `touch testfile` to create a file. It will say "touch: cannot touch 'testfile': Permission denied". This is because your new user account doesn't have the right to create files in the /home/pi directory. Enter: `su pi` to switch back to your pi account.

```
pi@raspberrypi ~ $ su lucy
Password:
lucy@raspberrypi /home/pi $ touch testfile
touch: cannot touch `testfile': Permission denied
lucy@raspberrypi /home/pi $ _
```

GETTING SUDO

We now have two accounts on our Raspberry Pi: lucy and pi. The lucy account can edit files in /home/lucy and the pi account can edit files in /home/pi. But there's also a third account, called "root", that sits above both lucy and pi. It can edit files anywhere.

STEP 1 The root account is all-powerful. It is possible, but not recommended, to switch to the root account, although you'll need to give it a password first (using "sudo passwd root"). Then just type "su" to switch to root. Please don't do this though: knowledge is a good thing but it's safer and wiser to use sudo instead.

```
pi@raspberrypi ~ $ sudo passwd root
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
pi@raspberrypi ~ $ su
Password: _
```

STEP 2 Most people think sudo stands for "super user", but it stands for "substitute user do". It enables you to perform a command as another user. Enter: `sudo -u lucy touch /home/lucy/test` to create a file inside the lucy home directory. You won't get an error because the lucy user has permission to edit that directory.

```
pi@raspberrypi ~ $ sudo -u lucy touch /home/lucy/test
pi@raspberrypi ~ $ _
```

STEP 3 It's rare that you use sudo to substitute another user. If you don't specify a user using the "-"u option with a username it defaults to the root account, as if you'd typed "sudo -u root". Enter: `sudo touch /home/lucy/anothertestfile` to create a file in the lucy directory while still using the pi account.

```
pi@raspberrypi ~ $ sudo touch /home/lucy/anothertestfile
pi@raspberrypi ~ $ ls /home/lucy/
anothertestfile  pistore.desktop  test
pi@raspberrypi ~ $ _
```

STEP 4 This step is optional. Only the pi user can use sudo. If we want to give the lucy account sudo privileges, it needs to be added to the sudoers file. Enter: `sudo visudo` to view the sudoers file. Add `lucy ALL=(ALL) NOPASSWD: ALL` to the last line and use Control+O to output the file. Remove the ".tmp" that is added to the file name as a security measure. Note that most accounts are not added to the sudoers file as a matter of course.

```
  GNU nano 2.2.6                                          File: /etc/sudoers.tmp

#
# This file MUST be edited with the 'visudo' command as root.
#
# Please consider adding local content in /etc/sudoers.d/ instead of
# directly modifying this file.
#
# See the man page for details on how to write a sudoers file.
#
Defaults        env_reset
Defaults        mail_badpass
Defaults        secure_path="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"

# Host alias specification

# User alias specification

# Cmnd alias specification

# User privilege specification
root    ALL=(ALL:ALL) ALL

# Allow members of group sudo to execute any command
%sudo   ALL=(ALL:ALL) ALL

# See sudoers(5) for more information on "#include" directives:

#includedir /etc/sudoers.d
pi ALL=(ALL) NOPASSWD: ALL
lucy ALL=(ALL) NOPASSWD: ALL
```

Ownership and Permissions

Once you've got the hang of users, you need to learn about ownership and permissions. Different users have different areas of ownership and can do different things with each file. Permissions in Linux can be quite complex but with careful thought it's not too difficult.

OWNER AND PRIVILEGE

Each user account in Linux is an owner of a section of the filesystem: their Home area. Within this area, they do what they like (within reason), as they have owner privileges. Elsewhere though, they usually just have read-only privileges.

STEP 1

If you followed the previous tutorial you should now have two accounts on your Raspberry Pi. One called "pi" and the other with a name (Lucy in our case). An essential aspect of Linux is the idea of file and directory ownership; who owns, and has access, to what. You need a test file so enter: `touch testfile`.

```
pi@raspberrypi ~ $ touch testfile.txt
```

STEP 2

Now enter: `ls -l` and let's have a good look at the default permissions file. Our testfile.txt files starts with the text "-rw-r--r—". Start with the first letter, which is a dash '-'. All the other items in our home directory are directories. You can tell because they are blue, and our testfile.txt file is white.

```
pi@raspberrypi ~ $ touch testfile.txt
pi@raspberrypi ~ $ ls -l
total 28
drwxr-xr-x 2 pi pi 4096 Apr 21 17:55 Desktop
drwxr-xr-x 3 pi pi 4096 May 13 18:08 Documents
drwx------ 2 pi pi 4096 May 13 11:01 Downloads
drwxr-xr-x 3 pi pi 4096 Apr 17 18:48 indiecity
drwxr-xr-x 3 pi pi 4096 May 13 14:53 people
drwxrwxr-x 2 pi pi 4096 Jan  1  1970 python_games
drwxr-xr-x 2 pi pi 4096 May 13 11:05 Scratch
-rw-r--r-- 1 pi pi    0 May 13 21:14 testfile.txt
pi@raspberrypi ~ $
```

STEP 3

The first letter in the permissions also indicates a directory or file. Notice that all the other files start with a 'd' and our textfile.txt file starts with a '-'. That's what the first letter means. It's either a 'd', in which case it's a directory, or a '-', in which case it's not; it's a file. Enter: `ls -l testfile.txt` to view the permissions for just this file.

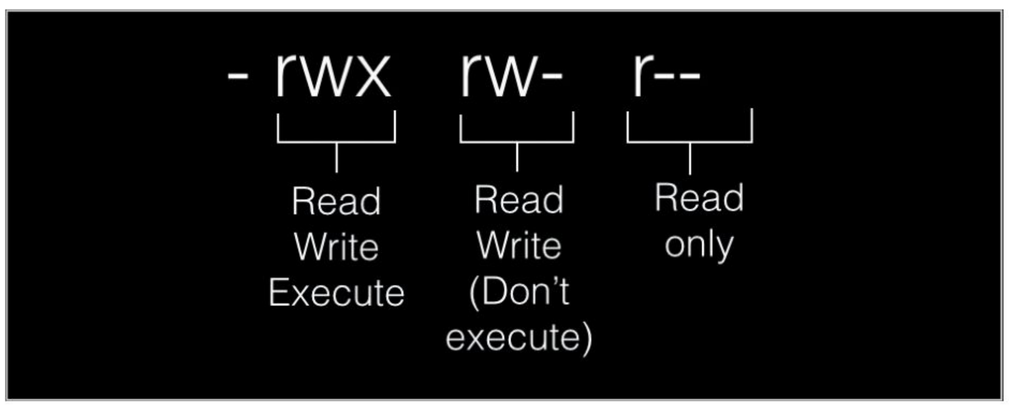
```
pi@raspberrypi ~ $ ls -l
total 28
drwxr-xr-x 2 pi pi 4096 Apr 21 17:55 Desktop
drwxr-xr-x 3 pi pi 4096 May 13 18:08 Documents
drwx------ 2 pi pi 4096 May 13 11:01 Downloads
drwxr-xr-x 3 pi pi 4096 Apr 17 18:48 indiecity
drwxr-xr-x 3 pi pi 4096 May 13 14:53 people
drwxrwxr-x 2 pi pi 4096 Jan  1  1970 python_games
drwxr-xr-x 2 pi pi 4096 May 13 11:05 Scratch
-rw-r--r-- 1 pi pi    0 May 13 21:14 testfile.txt
pi@raspberrypi ~ $ ls -l testfile.txt
-rw-r--r-- 1 pi pi 0 May 13 21:14 testfile.txt
pi@raspberrypi ~ $ _
```

STEP 4

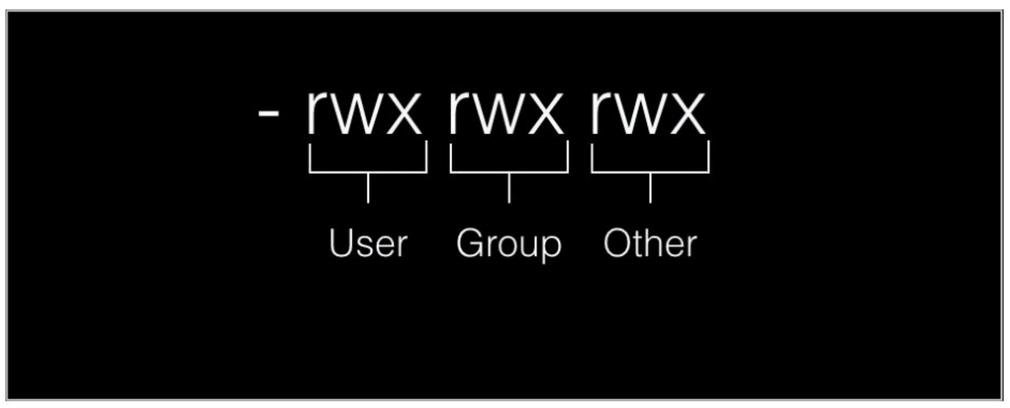
The next nine letters of the permissions are known as "alpha notation" because they let you know the permissions using letters. Each permission is either on, in which case you see a letter, or it is off, in which case you see a dash. The letter doesn't change for each place. So the first permission - the second letter after the directory one - is either an 'r' or a '-'. It's never any other letter.

```
pi@raspberrypi ~ $ ls -l testfile.txt
-rw-r--r-- 1 pi pi 0 May 13 21:14 testfile.txt
pi@raspberrypi ~ $ _
```

STEP 5 The 'r' means that particular permission is read and it's set to On. The nine letters here are divided into three groups of three letters: r, w, x. They stand for read, write and execute (run). Read means the file can be viewed (using cat or nano); w means the file can be edited or moved, and x means the file - typically a script or program - can be run.



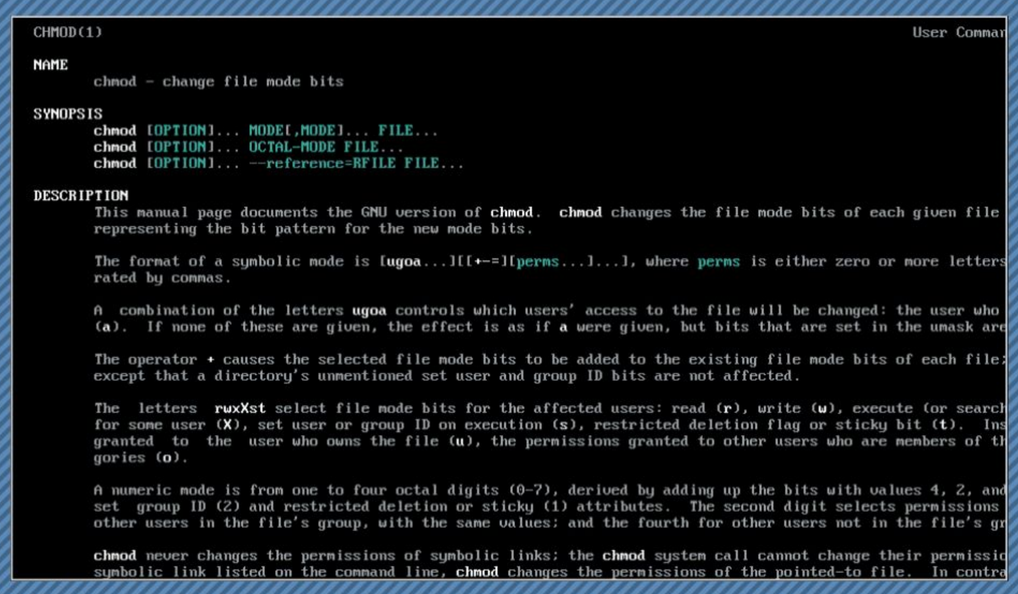
STEP 6 The presence of r, w, or x means that this aspect is possible, a dash means it isn't. Our testfile.txt has no x letter; so if it were a script it wouldn't run. So why are there so many letters? Why not just three; read, write and execute? The three blocks of three letters are for different sets of people: user, group and other.



CHANGING PERMISSIONS

Now that you know how groups of permissions work, it's time to look at how to change them.

STEP 1 The first block of three is the most important. This is the user who owns the file (typically pi); the second is for other people in the same group as the user, and the third is for other people on the system. Permissions are changed using the chmod (change file mode bit) command. Enter: `man chmod` to look at the manual.



```
CHMOD(1)                                                    User Comma

NAME
       chmod - change file mode bits

SYNOPSIS
       chmod [OPTION]... MODE[,MODE]... FILE...
       chmod [OPTION]... OCTAL-MODE FILE...
       chmod [OPTION]... --reference=RFILE FILE...

DESCRIPTION
       This manual page documents the GNU version of chmod.  chmod changes the file mode bits of each given file
       representing the bit pattern for the new mode bits.

       The format of a symbolic mode is [ugoa...][[+-=][perms...]...], where perms is either zero or more letters
       rated by commas.

       A  combination of the letters ugoa controls which users' access to the file will be changed: the user who
       (a).  If none of these are given, the effect is as if a were given, but bits that are set in the umask are

       The operator + causes the selected file mode bits to be added to the existing file mode bits of each file;
       except that a directory's unmentioned set user and group ID bits are not affected.

       The  letters  rwxXst select file mode bits for the affected users: read (r), write (w), execute (or search
       for some user (X), set user or group ID on execution (s), restricted deletion flag or sticky bit (t).  Ins
       granted  to  the  user who owns the file (u), the permissions granted to other users who are members of th
       gories (o).

       A numeric mode is from one to four octal digits (0-7), derived by adding up the bits with values 4, 2, and
       set  group ID (2) and restricted deletion or sticky (1) attributes.  The second digit selects permissions
       other users in the file's group, with the same values; and the fourth for other users not in the file's gr

       chmod never changes the permissions of symbolic links; the chmod system call cannot change their permissio
       symbolic link listed on the command line, chmod changes the permissions of the pointed-to file.  In contra
```

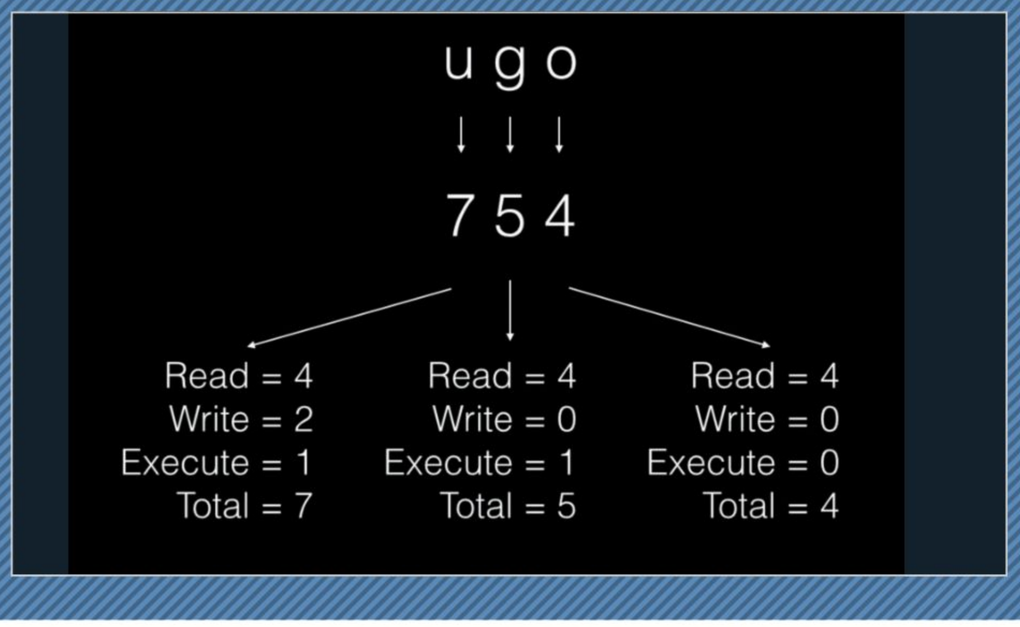
STEP 2 The chmod command is one of the trickier ones to understand. There are two ways you can adjust permissions; the first is using chmod with an option to target one of the three groups: owner, group, other. For these you use u, g or o followed by = and the letters or dashes you want. So enter: `chmod ugo=rx testfile.txt` to make all three groups read, write and execute.

```
pi@raspberrypi ~ $ chmod ugo=rwx testfile.txt
pi@raspberrypi ~ $ ls -l
total 28
drwxr-xr-x 2 pi pi 4096 Apr 21 17:55 Desktop
drwxr-xr-x 3 pi pi 4096 May 13 18:08 Documents
drwx------ 2 pi pi 4096 May 13 11:01 Downloads
drwxr-xr-x 3 pi pi 4096 Apr 17 18:48 indiecity
drwxr-xr-x 3 pi pi 4096 May 13 14:53 people
drwxrwxr-x 2 pi pi 4096 Jan  1  1970 python_games
drwxr-xr-x 2 pi pi 4096 May 13 11:05 Scratch
-rwxrwxrwx 1 pi pi    0 May 13 21:14 testfile.txt
pi@raspberrypi ~ $ _
```

STEP 3 Turning everything on is probably overkill, so you need to target each group. Do this by putting commas between each mode option. Enter: chmod u=rwx,g=rw,o=r testfile.txt to give users read, write and execute privileges, user read and write and other just read. Enter: `ls -l` to see your handiwork.

```
pi@raspberrypi ~ $ chmod u=rwx,g=rw,o=r testfile.txt
pi@raspberrypi ~ $ ls -l
total 28
drwxr-xr-x 2 pi pi 4096 Apr 21 17:55 Desktop
drwxr-xr-x 3 pi pi 4096 May 13 18:08 Documents
drwx------ 2 pi pi 4096 May 13 11:01 Downloads
drwxr-xr-x 3 pi pi 4096 Apr 17 18:48 indiecity
drwxr-xr-x 3 pi pi 4096 May 13 14:53 people
drwxrwxr-x 2 pi pi 4096 Jan  1  1970 python_games
drwxr-xr-x 2 pi pi 4096 May 13 11:05 Scratch
-rwxrw-r-- 1 pi pi    0 May 13 21:14 testfile.txt
pi@raspberrypi ~ $ _
```

STEP 4 Alpha notation is fine, but many Linux admins use octal notation instead. This is a three-digit number that represents permissions. This is the formula: read=4, write=2 and execute=1 and you add them up for each group, therefore if a group is read, write and execute it's 7, if it's read and write it's 6 or if it's just execute it's 1. A popular option is 755. Enter: `chmod 755 testfile.txt` to change the file using octal notation.



Useful System and Disk Commands

Understanding these core Linux commands will enable you to not only master the inner workings of your Raspberry Pi but also to transfer those skills to other Linux distros, such as Ubuntu or Linux Mint.

LOTS OF LINUX

Linux is a huge and versatile command line language and there are hundreds of commands you can learn and use. Here are a few that can help you get more from your Raspberry Pi.

STEP 1

The Raspberry Pi is a great little computer, so let's start by getting some information. Enter: `cat /proc/cpuinfo` to view some details on your Raspberry Pi processors. If you have a Raspberry Pi 3 you will see four processors, along with the model name and other info.

```
pi@raspberrypi ~ $ cat /proc/cpuinfo
processor       : 0
model name      : ARMv7 Processor rev 5 (v7l)
BogoMIPS        : 38.40
Features        : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae evtstrm
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x0
CPU part        : 0xc07
CPU revision    : 5

processor       : 1
model name      : ARMv7 Processor rev 5 (v7l)
BogoMIPS        : 38.40
Features        : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae evtstrm
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x0
CPU part        : 0xc07
CPU revision    : 5

processor       : 2
model name      : ARMv7 Processor rev 5 (v7l)
BogoMIPS        : 38.40
Features        : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae evtstrm
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x0
CPU part        : 0xc07
CPU revision    : 5

processor       : 3
model name      : ARMv7 Processor rev 5 (v7l)
BogoMIPS        : 38.40
Features        : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae evtstrm
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x0
```

STEP 2

Remember that cat is used to list the contents of a text file, which is what cpuinfo is. There are other text files with system info available. Try "cat /proc/meminfo" to get information about your memory, "cat /proc/partitions" for information about your SD card, and "cat /proc/version" shows which version of Raspberry Pi you are using.

```
pi@raspberrypi ~ $ cat /proc/meminfo
MemTotal:         884304 kB
MemFree:          784576 kB
MemAvailable:     834088 kB
Buffers:           11660 kB
Cached:            56340 kB
SwapCached:            0 kB
Active:            49148 kB
Inactive:          30072 kB
Active(anon):      11256 kB
Inactive(anon):      228 kB
Active(file):      37892 kB
Inactive(file):    29844 kB
Unevictable:           0 kB
Mlocked:               0 kB
SwapTotal:        102396 kB
SwapFree:         102396 kB
Dirty:                36 kB
Writeback:             0 kB
AnonPages:         11132 kB
Mapped:             7464 kB
Shmem:               260 kB
Slab:              10384 kB
SReclaimable:       4512 kB
SUnreclaim:         5872 kB
KernelStack:         672 kB
PageTables:          564 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:      544548 kB
Committed_AS:      43712 kB
VmallocTotal:    1171456 kB
VmallocUsed:        3880 kB
VmallocChunk:     922360 kB
pi@raspberrypi ~ $ cat /proc/partitions
major minor  #blocks  name

 179        0    7761920 mmcblk0
 179        1     835890 mmcblk0p1
 179        2          1 mmcblk0p2
 179        5      32768 mmcblk0p5
```

STEP 3

Enter: `uname` to view the name of the operating system's kernel, this is the element that sits between the interface and hardware. Just as you would suspect, the response from the command is Linux, as Raspbian is a Linux distro, which in itself is based on another Linux distro called Debian. While it may sound complicated, it actually demonstrates how versatile Linux is.

```
pi@raspberrypi ~ $ uname
Linux
pi@raspberrypi ~ $
```

STEP 4

Enter: `uname -a` to view some more detailed information. Here you'll see the kernel name, hostname and kernel version (3.18.7-v7 on ours). If you have a Raspberry Pi 2 you'll see SMP (symmetric multiprocessing), followed by the system date, CPU architecture and operating system (GNU/Linux).

```
pi@raspberrypi ~ $ uname
Linux
pi@raspberrypi ~ $ uname -a
Linux raspberrypi 3.18.7-v7+ #755 SMP PREEMPT Thu Feb 12 17:2
pi@raspberrypi ~ $ _
```

STEP 5 Enter: `vcgencmd measure_temp` to view the current operating system temperature of your Raspberry Pi. Enter: `vcgencmd get_mem arm` to view the RAM available, and `vcgencmd get_mem gpu` to view the memory available to the graphics chip. Finally try `ls usb` to view a list of attached USB devices.

```
pi@raspberrypi ~ $ vcgencmd measure_temp
temp=36.9'C
pi@raspberrypi ~ $ vcgencmd get_mem arm
arm=880M
pi@raspberrypi ~ $ vcgencmd get_mem gpu
gpu=128M
pi@raspberrypi ~ $ lsusb
Bus 001 Device 002: ID 0424:9514 Standard Microsystems Corp.
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 001 Device 003: ID 0424:ec00 Standard Microsystems Corp.
Bus 001 Device 004: ID 04d9:1503 Holtek Semiconductor, Inc. Shortboard Lefty
Bus 001 Device 005: ID 1a40:0101 Terminus Technology Inc. 4-Port HUB
Bus 001 Device 006: ID 276d:1105
pi@raspberrypi ~ $
```

STEP 6 One command you might be wondering about is how to switch off or restart your Raspberry Pi from the command line. Don't just hit the power switch. Enter: `sudo showdown –h` now to shut down the Raspberry Pi (the "-h" option stands for "halt"), or enter: `sudo shutdown –r` now to restart your Raspberry Pi.

```
pi@raspberrypi ~ $ sudo shutdown -r now

Broadcast message from root@raspberrypi (tty1) (Thu May 14 12:20:29 2015):
The system is going down for reboot NOW!
_
```

DISK COMMANDS

Learn the two commands that enable you to view your disk space and the files on it: df (disk free space) and du (disk usage). With these two commands you can view the file usage on your SD card.

STEP 1 Start by entering: `df` in the command line. It returns a list of the volumes contained on your SD card. You might be wondering what a volume is. It's best to think of your SD card as the drive. This contains partitions, which is where you split one drive to act like two or more drives. And each partition can contain volumes, which are storage spaces.


```
pi@raspberrypi ~ $ vcgencmd measure_temp
temp=36.9'C
pi@raspberrypi ~ $ vcgencmd get_mem arm
arm=880M
pi@raspberrypi ~ $ vcgencmd get_mem gpu
gpu=128M
pi@raspberrypi ~ $ lsusb
Bus 001 Device 002: ID 0424:9514 Standard Microsystems Corp.
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 001 Device 003: ID 0424:ec00 Standard Microsystems Corp.
Bus 001 Device 004: ID 04d9:1503 Holtek Semiconductor, Inc. Shortboard Lefty
Bus 001 Device 005: ID 1a40:0101 Terminus Technology Inc. 4-Port HUB
Bus 001 Device 006: ID 276d:1105
pi@raspberrypi ~ $
```


STEP 2 Enter: `df –h` to get the list in human readable form. The first two lines should read "rootfs" and "/dev/root" and have matching Size, Used, Avail and Use% listings. This is the main drive, and is an indication of how much space you have used, and have free, on your Raspbian OS. The other volumes are for booting and initialising devices (you can ignore these for now).

```
pi@raspberrypi ~ $ du -h | less
22M     ./.minecraft/games/com.mojang/minecraftWorlds/world
22M     ./.minecraft/games/com.mojang/minecraftWorlds
22M     ./.minecraft/games/com.mojang
22M     ./.minecraft/games
4.0K    ./.pulse
16K     ./.config/gedit
8.0K    ./.config/libfm
1.4M    ./.config/epiphany/adblock
1.5M    ./.config/epiphany
8.0K    ./.config/lxsession/LXDE-pi
12K     ./.config/lxsession
8.0K    ./.config/dconf
8.0K    ./.config/rncbc.org
8.0K    ./.config/lxterminal
8.0K    ./.config/uk.ac.cam.cl
8.0K    ./.config/IndieCity
4.0K    ./.config/enchant
8.0K    ./.config/lxpanel/LXDE-pi/panels
16K     ./.config/lxpanel/LXDE-pi
```

STEP 3 Now enter: `du`. You should see lots of text fly up the screen. This is the disk usage for the files contained in your home directory and their sub-directories. As with df, it is better to use du with the "-h" option to humanise the output. If you want to slow down the output, you'll also need to pipe it through less. Enter: `df -h | less` to view the files and their respective usage one page at a time.


```
pi@raspberrypi ~ $ sudo shutdown -r now

Broadcast message from root@raspberrypi (tty1) (Thu May 14 12:20:29 2015):
The system is going down for reboot NOW!
_
```


STEP 4 You don't typically enter: `du` on its own; most of the time you want to view the disk usage of a specific directory. Enter: `du -h python_games` to view how much space the python_games directory (installed alongside Raspbian) takes up. It should be 1.8M. If you want a more comprehensive breakdown of the files contained, use the "-a" option (all). Enter: `du -ha python_games` to view all the files contained and their disk usage.

```
pi@raspberrypi ~ $ ls
Desktop  Documents  Downloads  indiecity  people  python_games  Scratch  testfile.txt
pi@raspberrypi ~ $ du -h python_games
1.8M    python_games
pi@raspberrypi ~ $ du -ha python_games
12K     python_games/RedSelector.png
12K     python_games/4row_board.png
12K     python_games/Star.png
28K     python_games/4row_humanwinner.png
12K     python_games/Wall_Block_Tall.png
8.0K    python_games/princess.png
12K     python_games/Selector.png
8.0K    python_games/4row_black.png
4.0K    python_games/catanimation.py
20K     python_games/flippy.py
36K     python_games/match3.wav
24K     python_games/starpusher.py
4.0K    python_games/grass1.png
40K     python_games/beep2.ogg
12K     python_games/slidepuzzle.py
8.0K    python_games/gem5.png
16K     python_games/fourinarow.py
```

Managing Programs and Processes

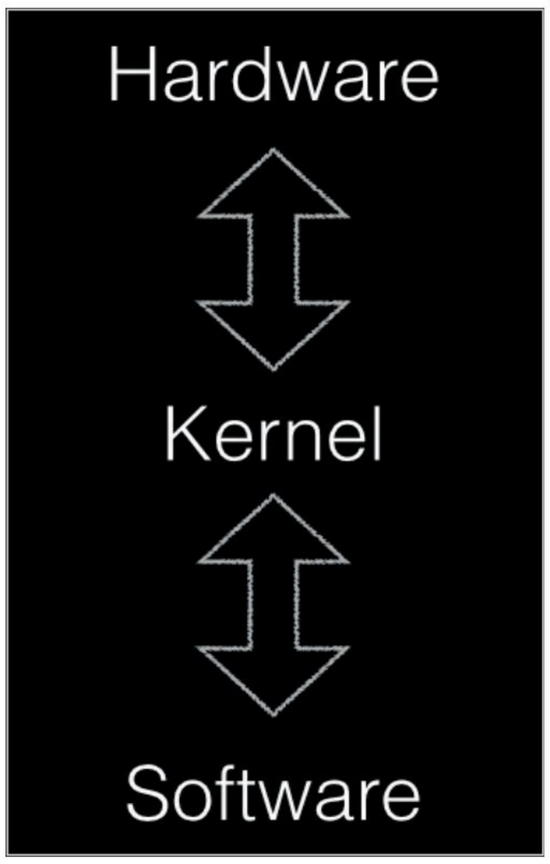
Being able to effectively manage the active programs and processes on your Raspberry Pi allows you to change the way the systems work. If you have a project requiring more memory, you can kill a process to free up the available system resources.

PROGRAMS AND PROCESSES

Linux has a trick up its sleeve when it comes to being able to manage programs and processes. When Windows closes a program, the allocated memory often isn't freed up again. However, Linux is far more streamlined.

STEP 1

As you get into Linux you'll start to hear more about processes and another thing called the "kernel". The kernel sits beneath the software and hardware. It passes instructions to the hardware running processes, which takes up memory, and when the process is finished it closes it and reclaims the memory.



STEP 2

You're probably used to thinking in terms of programs and most OS's tend to keep processes out of sight. In Linux on the other hand, you're right in at the deep end. A process is like a program, only it's a single task running on your computer. Programs as you know them, may be a single process, or multiple processes working together. In Linux, you should learn to manage processes. This is done using the "ps" (process status) command.

```
pi@raspberrypi ~ $ ps_
```

STEP 3

If you type in "ps" on its own you don't see much. You should see two items: bash and ps. Bash (Bourne Again Shell) is the command line environment you are typing in, and ps is the command you just entered. If that seems a little light, that's because it is. These are just the processes owned by the user and are running in the foreground.

```
pi@raspberrypi ~ $ ps
  PID TTY          TIME CMD
 2165 tty1     00:00:01 bash
 2376 tty1     00:00:00 ps
pi@raspberrypi ~ $ _
```

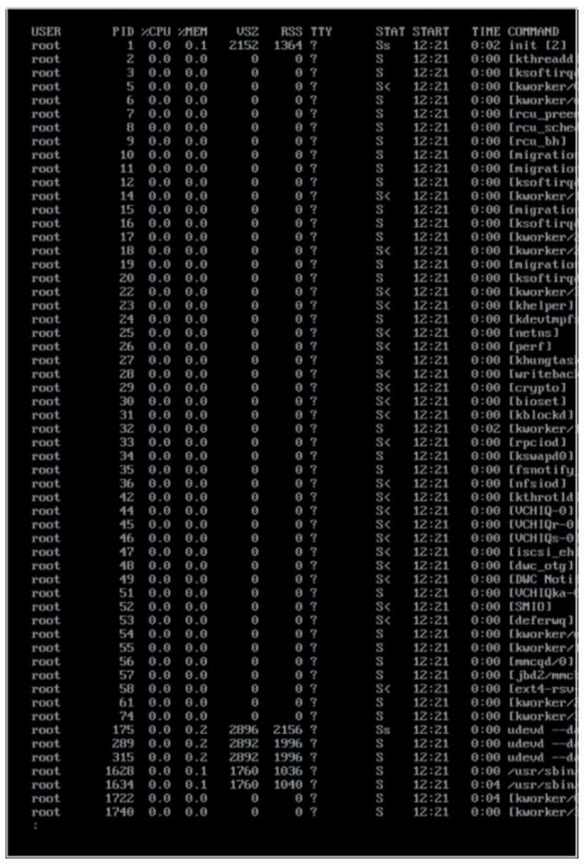
STEP 4

If you want to see processes used by other users (including those started by root) enter: `ps –a`. The option stands for all users. This still isn't everything though because it doesn't include background processes. For this you can enter either "ps –A" or "ps –e". This will show you every process on the system including the background processes. You may need to pipe it through less using "ps -e | less".

```
  PID TTY          TIME CMD
    1 ?        00:00:02 init
    2 ?        00:00:00 kthreadd
    3 ?        00:00:00 ksoftirqd/0
    5 ?        00:00:00 kworker/0:0H
    6 ?        00:00:00 kworker/u8:0
    7 ?        00:00:00 rcu_preempt
    8 ?        00:00:00 rcu_sched
    9 ?        00:00:00 rcu_bh
   10 ?        00:00:00 migration/0
   11 ?        00:00:00 migration/1
   12 ?        00:00:00 ksoftirqd/1
   14 ?        00:00:00 kworker/1:0H
   15 ?        00:00:00 migration/2
   16 ?        00:00:00 ksoftirqd/2
   17 ?        00:00:00 kworker/2:0
   18 ?        00:00:00 kworker/2:0H
   19 ?        00:00:00 migration/3
   20 ?        00:00:00 ksoftirqd/3
   22 ?        00:00:00 kworker/3:0H
   23 ?        00:00:00 khelper
   24 ?        00:00:00 kdevtmpfs
   25 ?        00:00:00 netns
   26 ?        00:00:00 perf
   27 ?        00:00:00 khungtaskd
   28 ?        00:00:00 writeback
   29 ?        00:00:00 crypto
   30 ?        00:00:00 bioset
   31 ?        00:00:00 kblockd
   32 ?        00:00:02 kworker/1:1
   33 ?        00:00:00 rpciod
   34 ?        00:00:00 kswapd0
   35 ?        00:00:00 fsnotify_mark
   36 ?        00:00:00 nfsiod
   42 ?        00:00:00 kthrotld
   44 ?        00:00:00 VCHIQ-0
   45 ?        00:00:00 VCHIQr-0
   46 ?        00:00:00 VCHIQs-0
   47 ?        00:00:00 iscsi_eh
   48 ?        00:00:00 dwc_otg
   49 ?        00:00:00 DWC Notificatio
```

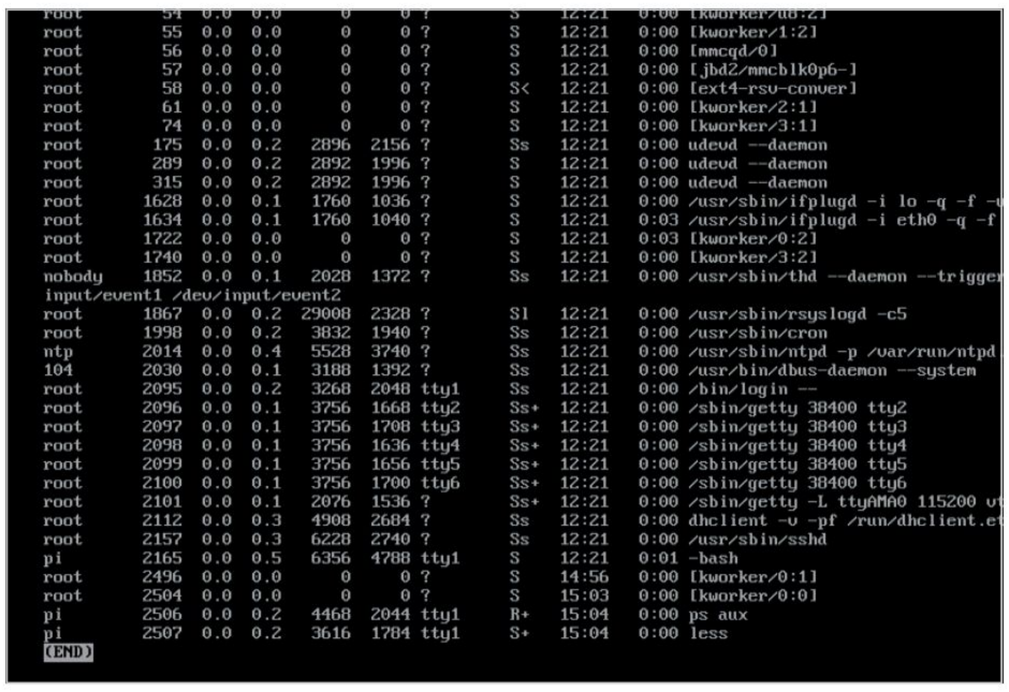
STEP 5

The ps command is one of the oldest known, and it has a lot of legacy. Because of this, it also has alternative options that can be used without the dash at the start. The most common way to use ps is to enter: `ps aux`. Piping this command through less is the best way to go about using ps. Enter: `ps aux | less` to view your processes.



STEP 6

The "ps aux" command displays the processes in columns. The first is the User followed by the Process ID (PID), then we see the percentage of CPU and memory.

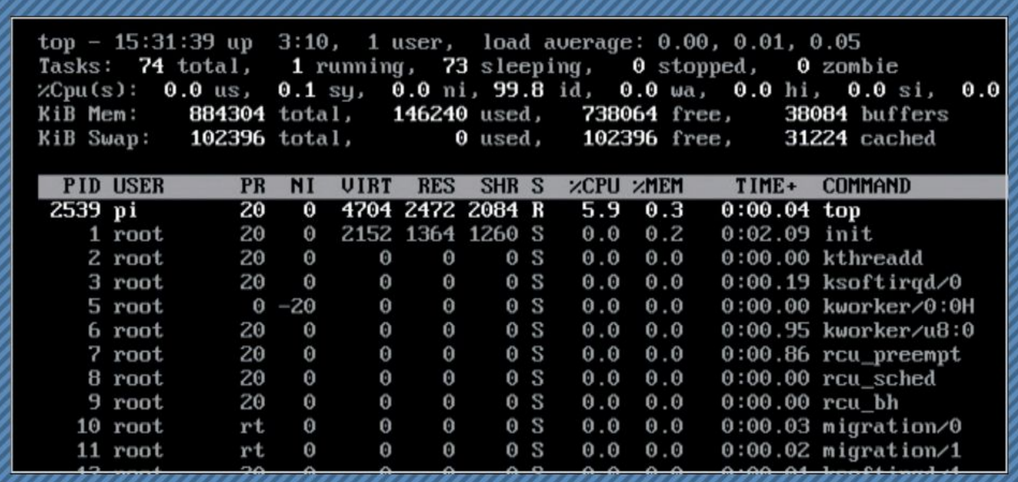


VIEWING AND QUITTING PROCESSES

Now that you've got a handle on processes, you need to know what to do with them. You can view processes running in real-time, and quit ones that you no longer need, or those that are problematic.

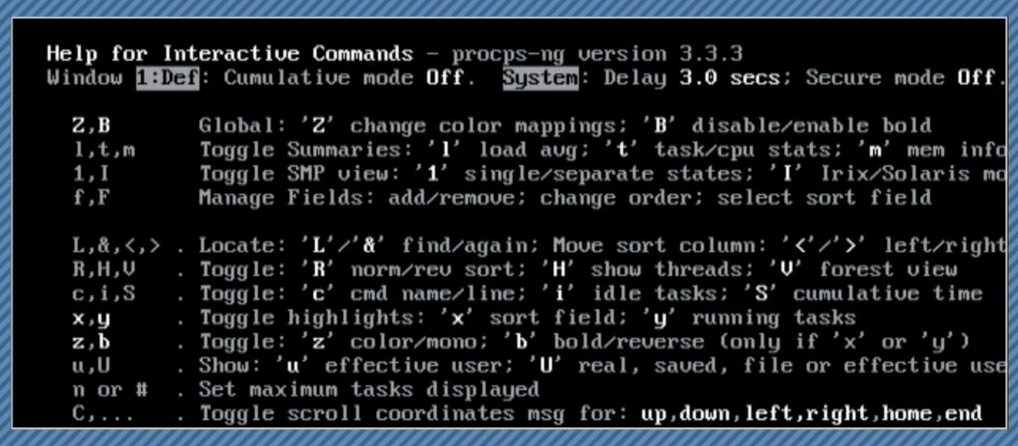
STEP 1

While ps is a great tool for checking all the processes on your Raspberry Pi, sometimes you need to view them in real-time. In particular you may want to check the CPU and memory usage of processes and see if any process is hogging all your computer's resources. Use "top" to do this.



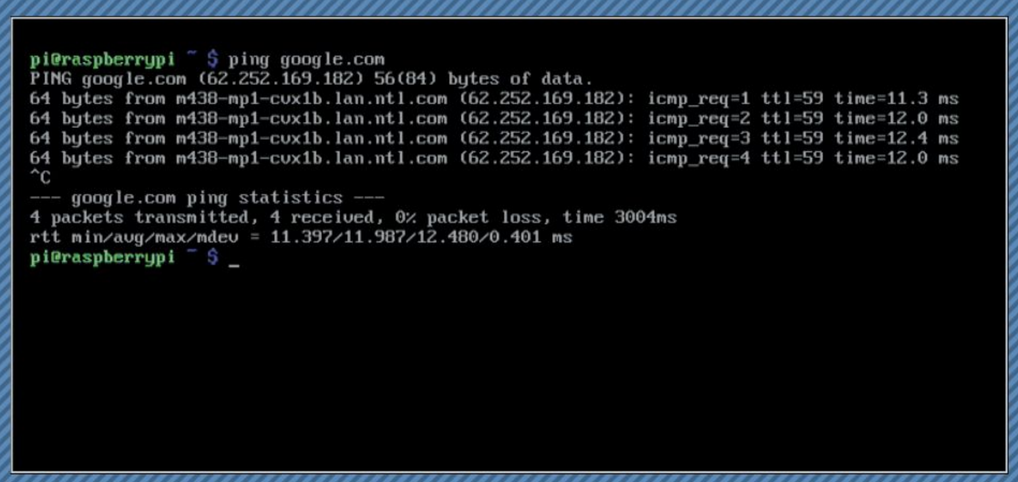
STEP 2

Top fills the display with processes and it fits as many as it can on one screen. These run in real-time so as you watch, the display items will move up and down according to their usage. You can interact with top as it is running: use < and > to change the sort column. Press H to view a help page with all the commands you can use.



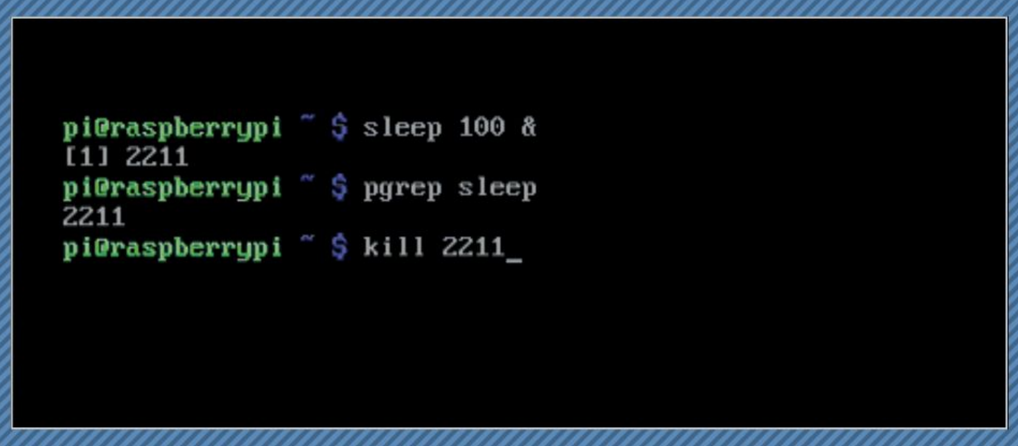
STEP 3

Now that you've got an idea of what processes are, how do you go about stopping them? Some processes run in the foreground until you quit them. Enter: `ping www.google.com` and you'll start a ping process continuously contacting Google. This sort of process runs until you press Control+C.



STEP 4

Processes are stopped using the kill command. Enter: `sleep 100 &` to create a dummy process. Enter: `pgrep sleep` to find its PID number (on ours it is 2192). Now enter: `kill 2192` and the sleep process will be stopped. You can also use `pkill sleep` if you're sure you know the process name.



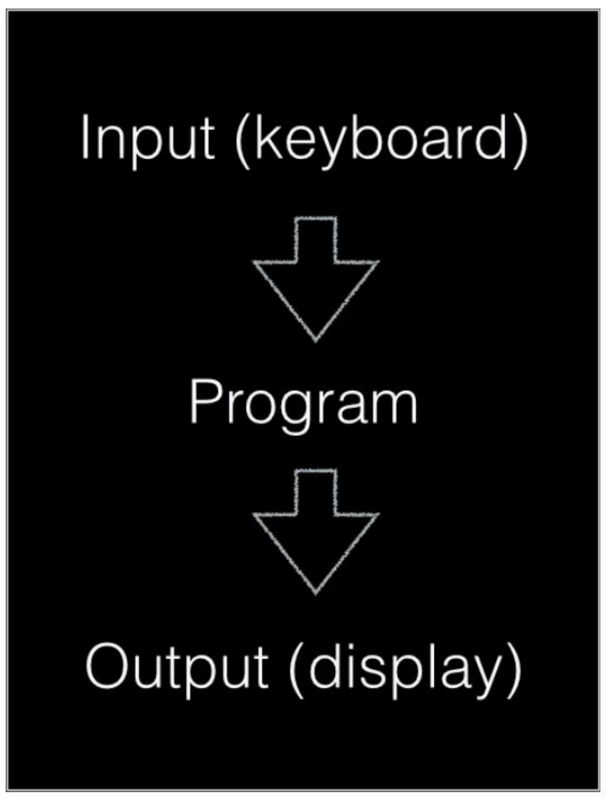
Input, Output and Pipes

Most operating systems allow you to direct the output of something on the screen to, for example, a file. Linux, with its ancestral history of Unix-like command-based arguments, goes a step further and offers much more control.

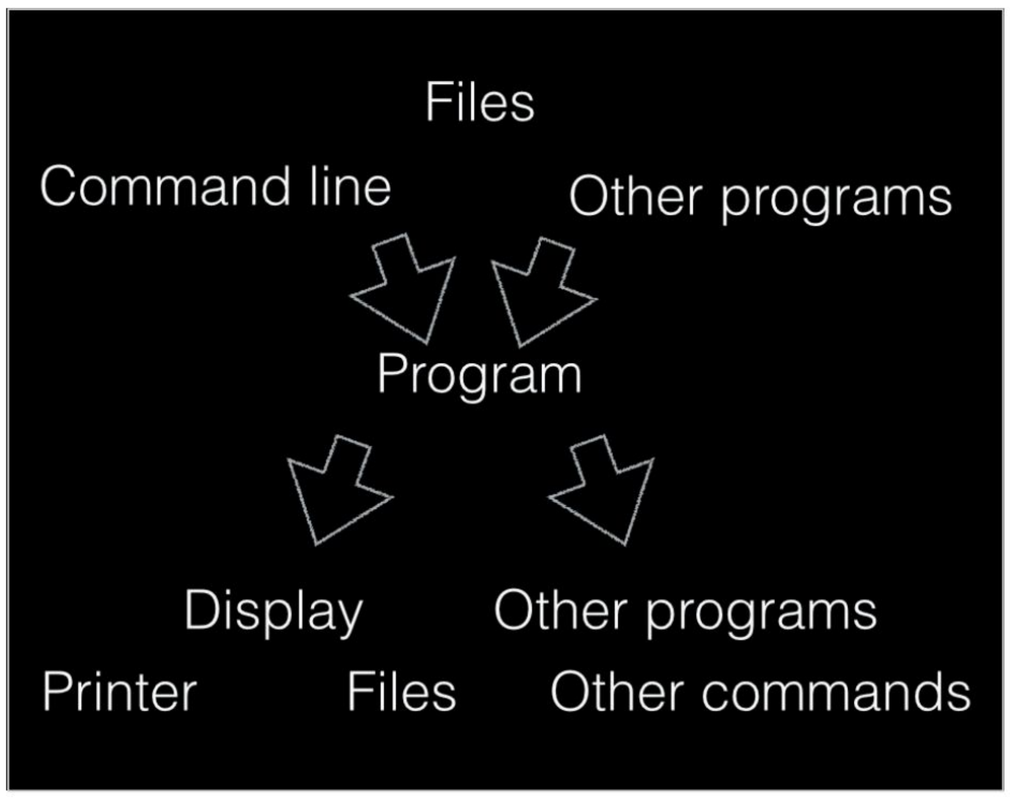
IN, OUT, LINUX ALL ABOUT

Everything on a computer is about input and output. You input things into the computer (press keys, move the mouse) and it makes calculations and outputs content (changes the display, makes a noise, for example).

STEP 1 When you enter commands into Linux (in the command line), you are using standard input and output. This is so obvious that most people don't even think about it. You enter commands using the keyboard (that's input) and it responds to the screen (output). This is the regular way of doing it, which is why it's "standard input and output" (often called "stdin" and "stdout" for short).



STEP 2 As far as the computer is concerned, input and output can be to and from a whole range of different sources that you might never even think about. A program can get input from other programs, from files stored on the hard drive and a whole host of other areas. It outputs back to the display line, but also to files, other programs and even other commands.



STEP 3 You can change the standard output to something else using the ">" character after your command. If we wanted to see all the items in the python_games directory we could use "ls -l python_games". Using "ls -l python_games > games.txt" outputs the list of items to a new text file called "games.txt".

```
pi@raspberrypi ~ $ ls -l python_games > games.txt
```

STEP 4 The games.txt file now contains the output from the ls command. You can check it using "nano games.txt". It's an editable text file containing all the permissions, user and file size information and the names of the files. The output from the ls -l command, normally displayed on the screen, was instead sent to this file. Press Control+X to quit nano.

```
 GNU nano 2.2.6

total 1800
-rw-rw-r-- 1 pi pi   9731 Jan 27 08:34 4row_arrow.png
-rw-rw-r-- 1 pi pi   7463 Jan 27 08:34 4row_black.png
-rw-rw-r-- 1 pi pi   8666 Jan 27 08:34 4row_board.png
-rw-rw-r-- 1 pi pi  18933 Jan 27 08:34 4row_computerwinner.png
-rw-rw-r-- 1 pi pi  25412 Jan 27 08:34 4row_humanwinner.png
-rw-rw-r-- 1 pi pi   8562 Jan 27 08:34 4row_red.png
-rw-rw-r-- 1 pi pi   8912 Jan 27 08:34 4row_tie.png
-rw-rw-r-- 1 pi pi  36908 Jan 27 08:34 badswap.wav
-rw-rw-r-- 1 pi pi  39782 Jan 27 08:34 beep1.ogg
-rw-rw-r-- 1 pi pi  39284 Jan 27 08:34 beep2.ogg
-rw-rw-r-- 1 pi pi  38581 Jan 27 08:34 beep3.ogg
-rw-rw-r-- 1 pi pi  39214 Jan 27 08:34 beep4.ogg
-rw-rw-r-- 1 pi pi    308 Jan 27 08:34 blankpygame.py
-rw-rw-r-- 1 pi pi   6581 Jan 27 08:34 boy.png
-rw-rw-r-- 1 pi pi   1034 Jan 27 08:34 catanimation.py
-rw-rw-r-- 1 pi pi   7270 Jan 27 08:34 catgirl.png
-rw-rw-r-- 1 pi pi  12574 Jan 27 08:34 cat.png
-rw-rw-r-- 1 pi pi   1178 Jan 27 08:34 drawing.py
-rw-rw-r-- 1 pi pi  83036 Jan 27 08:34 flippybackground.png
-rw-rw-r-- 1 pi pi 340760 Jan 27 08:34 flippyboard.png
-rw-rw-r-- 1 pi pi  19479 Jan 27 08:34 flippy.py
-rw-rw-r-- 1 pi pi  13080 Jan 27 08:34 fourinarow.py
-rw-rw-r-- 1 pi pi   2134 Jan 27 08:34 gameicon.png
-rw-rw-r-- 1 pi pi   4382 Jan 27 08:34 gem1.png
-rw-rw-r-- 1 pi pi   5665 Jan 27 08:34 gem2.png
-rw-rw-r-- 1 pi pi   3454 Jan 27 08:34 gem3.png
-rw-rw-r-- 1 pi pi   4217 Jan 27 08:34 gem4.png
-rw-rw-r-- 1 pi pi   5507 Jan 27 08:34 gem5.png
-rw-rw-r-- 1 pi pi   1681 Jan 27 08:34 gem6.png
-rw-rw-r-- 1 pi pi   3132 Jan 27 08:34 gem7.png
-rw-rw-r-- 1 pi pi  22489 Jan 27 08:34 gemgem.py
-rw-rw-r-- 1 pi pi   3009 Jan 27 08:34 grass1.png
-rw-rw-r-- 1 pi pi   3019 Jan 27 08:34 grass2.png
-rw-rw-r-- 1 pi pi   3009 Jan 27 08:34 grass3.png
-rw-rw-r-- 1 pi pi   3032 Jan 27 08:34 grass4.png
-rw-rw-r-- 1 pi pi   8244 Jan 27 08:34 Grass_Block.png
-rw-rw-r-- 1 pi pi   7186 Jan 27 08:34 horngirl.png
-rw-rw-r-- 1 pi pi  18855 Jan 27 08:34 inkspilllogo.png
-rw-rw-r-- 1 pi pi  18739 Jan 27 08:34 inkspill.py
-rw-rw-r-- 1 pi pi   7855 Jan 27 08:34 inkspillresetbutton.png
-rw-rw-r-- 1 pi pi  10746 Jan 27 08:34 inkspillsettingsbutton.png
```

STEP 5 So > enables you to output to files, but you can also get input from a file. Make a new directory called music ("mkdir music") and switch to it ("cd music"). Enter: `nano bands.txt` to create a new text file. Enter some band names and press Control+O to output the file. Press Control+X to quit nano.

```
  GNU nano 2.2.6

The Beatles
The Who
The Kinks
Credence Clearwater_Revival
Jefferson Airplane
Aerosmith
ZZ Top
_
```

STEP 6 We're going to use this text file as input to the sort command. Enter: `sort < bands.txt` and the content from the text file is used as input to sort. Because the output isn't specified, it uses the standard output (the screen) but you use input and output together. Enter: `sort < bands.txt > bands_sorted.txt` to create a new file with the band names in order to switch back to your pi account.

```
pi@raspberrypi ~/music $ sort < bands.txt
Aerosmith
Credence Clearwater_Revival
Jefferson Airplane
The Beatles
The Kinks
The Who
ZZ Top
pi@raspberrypi ~/music $ sort < bands.txt > bands_
```

USING PIPES

As well as directing input and output to and from files, you can send the output from one command directly into another. This is known as piping, and uses the pipe character "|".

STEP 1 As you start to get more advanced in Linux, you begin to create more powerful commands, and one way you do this is by using the pipe character ("|"). Take some time to find this character if you haven't already: it usually sits above or to the left of the Return key on most keyboards.

```
pi@raspberrypi ~/music $ ps aux | less
```

STEP 3 You can pipe commands multiple times. Enter: `cat bands.txt | sort | grep The*` to get the bands starting with "The" in alphabetical order. The output of the text from the bands.txt document is passed into sort, and the output from sort is passed into grep which filters out the bands starting with "The". These bands form the output.

```
pi@raspberrypi ~/music $ cat bands.txt | sort | grep The*
The Beatles
The Kinks
The Who
pi@raspberrypi ~/music $ _
```

STEP 2 We've used the pipe a few times in the book ("ps aux | less"), but you might not have understood what's actually happening. Enter: `cat bands.txt | wc`. The output from the cat command (the text inside the document) isn't displayed on the screen, instead it is piped into the wc (word count) function. This then tells us how many lines, words and characters are in the document.

```
pi@raspberrypi ~/music $ cat bands.txt | wc
       7      13      94
pi@raspberrypi ~/music $ _
```

STEP 4 You can combine pipes with input and output to create complex expressions. You can also use >> to append outputted data to a file that already exists. Enter: `cat bands.txt | wc >> bands.txt`. This takes the output from the bands.txt file and pipes it into the wc (word count) function. The output from wc is appended to the end of the bands.txt file. Enter: `cat bands.txt` to view it.

```
pi@raspberrypi ~/music $ cat bands.txt | wc >> bands.txt_
```

Fun Things to Do in the Terminal

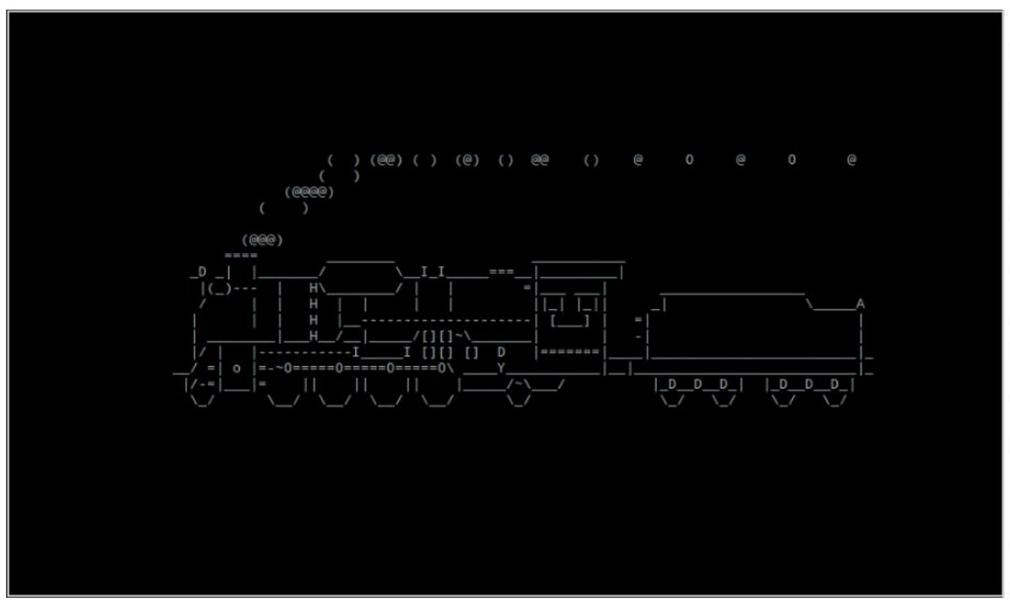
Despite the seriousness of an operating system, the Linux community are certainly no strangers to a bit of fun. Over the years, the developers have created and inserted all manner of quirky and entertaining elements into the Terminal.

TERMINAL FUN

All these commands are Linux-based, so not only can you use them on the Raspberry Pi but also on any of the Debian-based Linux distributions.

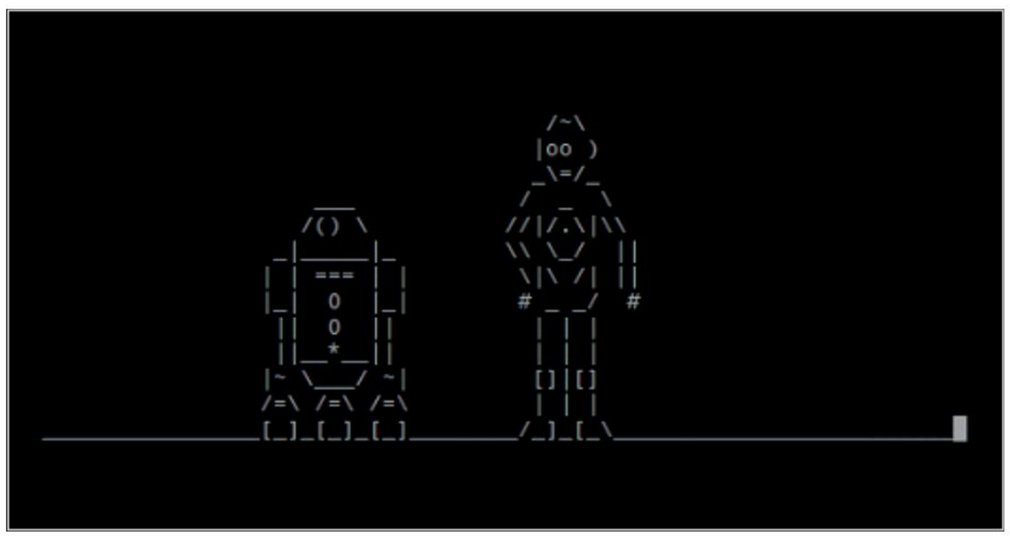
STEP 1

The first command we're going to use is **sl**, it's not installed by default so enter: `sudo apt-get install sl`. The command can be run with sl and when executed displays a Steam Locomotive travelling across the screen (hence 'sl'). Entering: `LS` (note the upper case) also works.



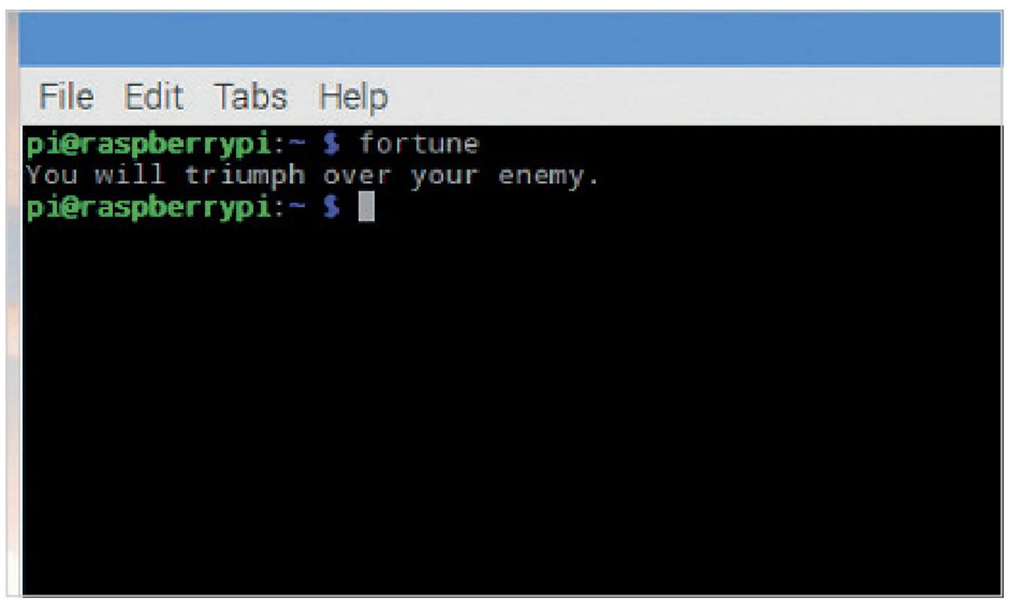
STEP 2

Fans of Star Wars even get a fix when it comes to the Terminal. By linking to a remote server via the telnet command, you can watch *Episode IV: A New Hope* being played out, albeit in ASCII. To view this spectacle, enter: `sudo apt-get install telnet`, followed by: `telnet towel.blinkenlights.nl`



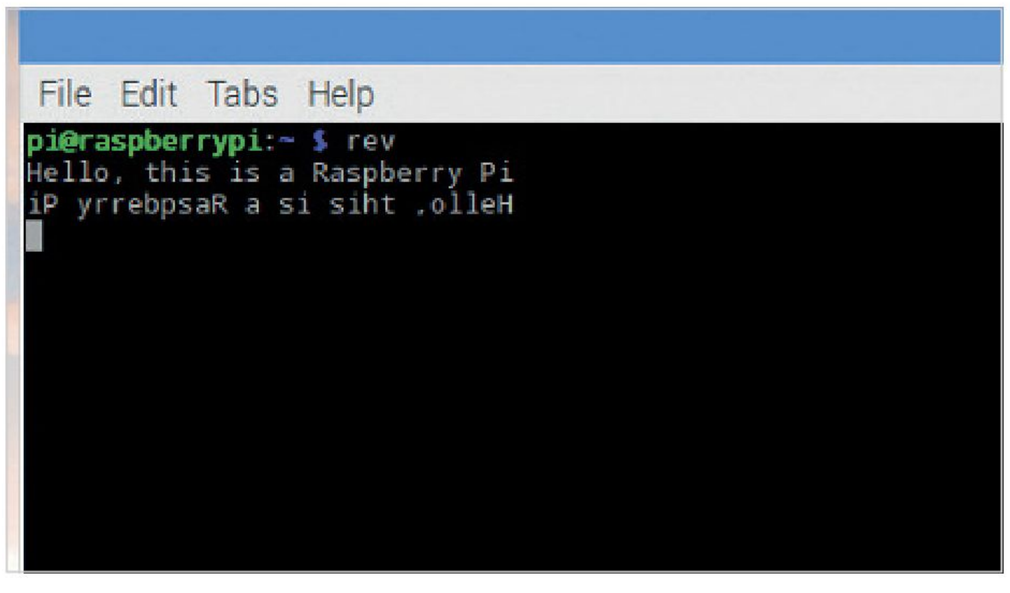
STEP 3

If you've ever fancied having the computer read a random fortune out to you then you're in luck. Raspbian requires you to install the Terminal app, Fortune, first. Enter: `sudo apt-get install fortune`, then simply enter: `fortune`, into the Terminal to see what comes up.



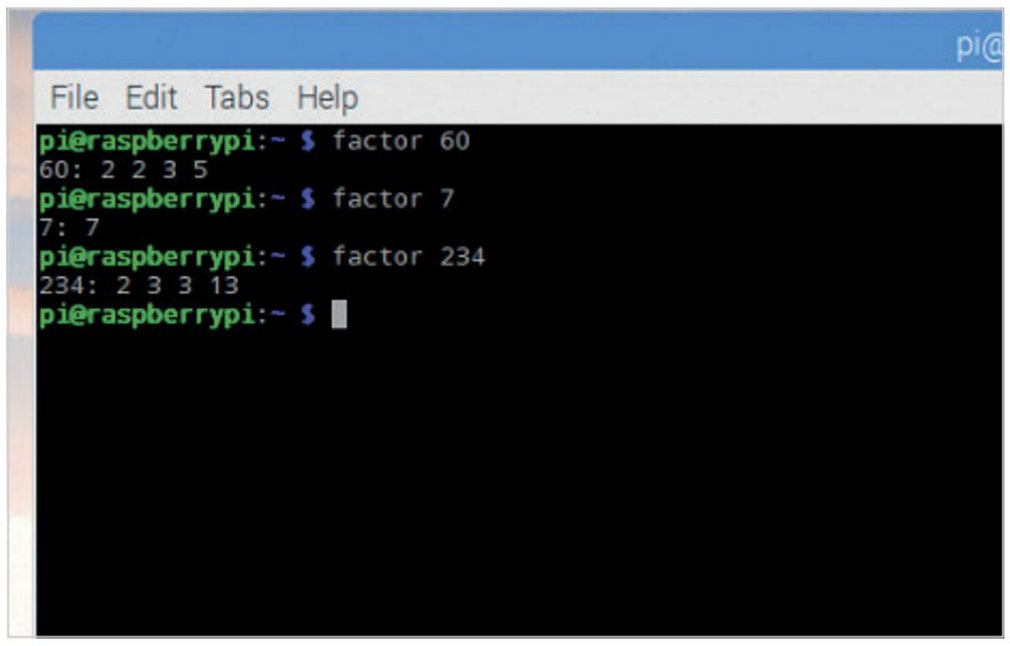
STEP 4

The rev command is certainly interesting, and at first what seems a quite useless addition to the OS can, however, be used to create some seemingly unbreakable passwords. Enter: `rev` and then type some text. Then press Enter and everything you typed in is reversed. Press Ctrl+C to exit.



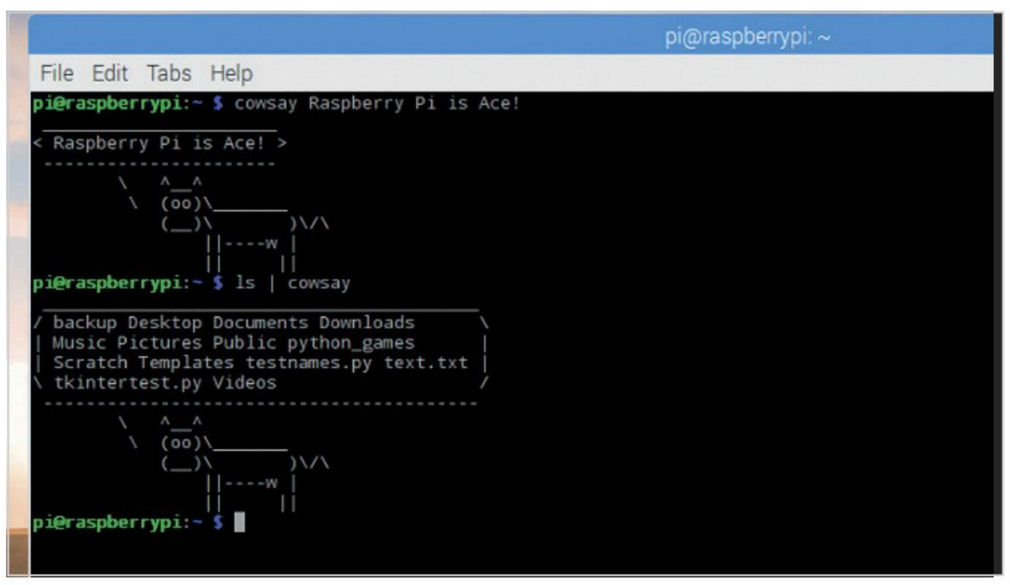
STEP 5

If you're stuck trying to work out all the possible factors for any particular number, simply enter: `factor` followed by the number. For example, factor 7 doesn't offer much output, whereas factor 60 displays more.



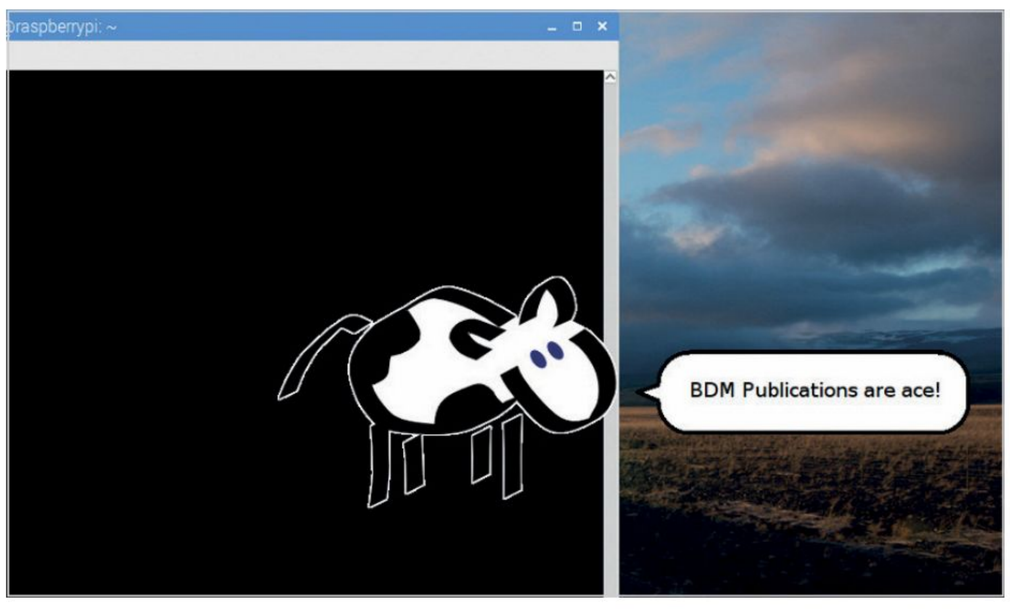
STEP 6

There's a fine line between the rather cool and the really quite weird. Having an ASCII cow repeat text to you could potentially fall into the latter. Start by installing **cowsay**: `sudo apt-get install cowsay`, then enter: `cowsay Raspberry Pi is Ace!`. In fact, you can even output the ls command through the cow, by entering: `ls | cowsay`.



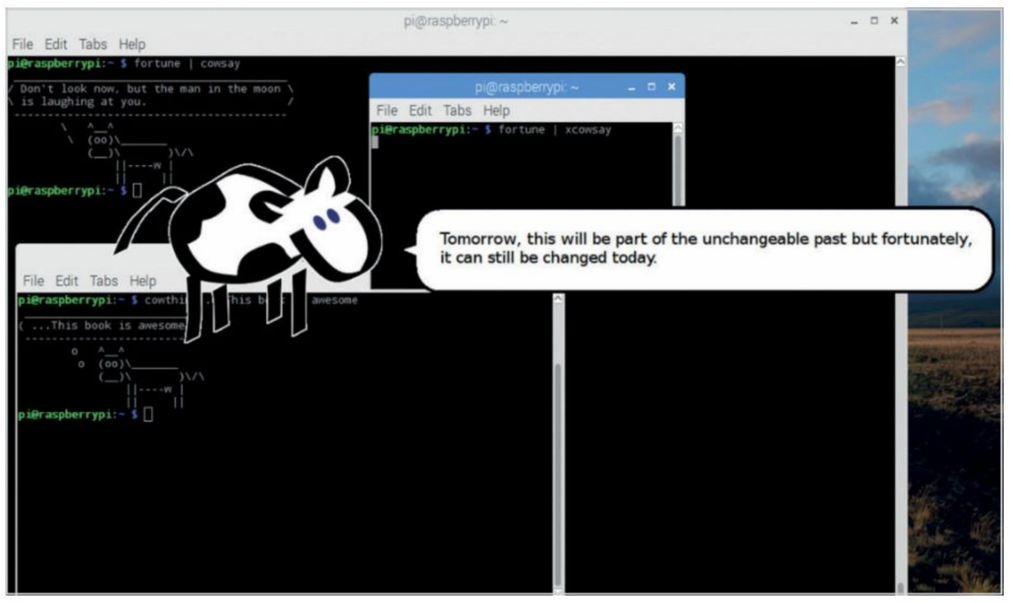
STEP 7

To further the cow element, there's even a graphical, i.e. non-Terminal, cow available. Install it with: `sudo apt-get install xcowsay`, then when it's installed enter something similar to cowsay, such as: `xcowsay BDM Publications are ace!`.



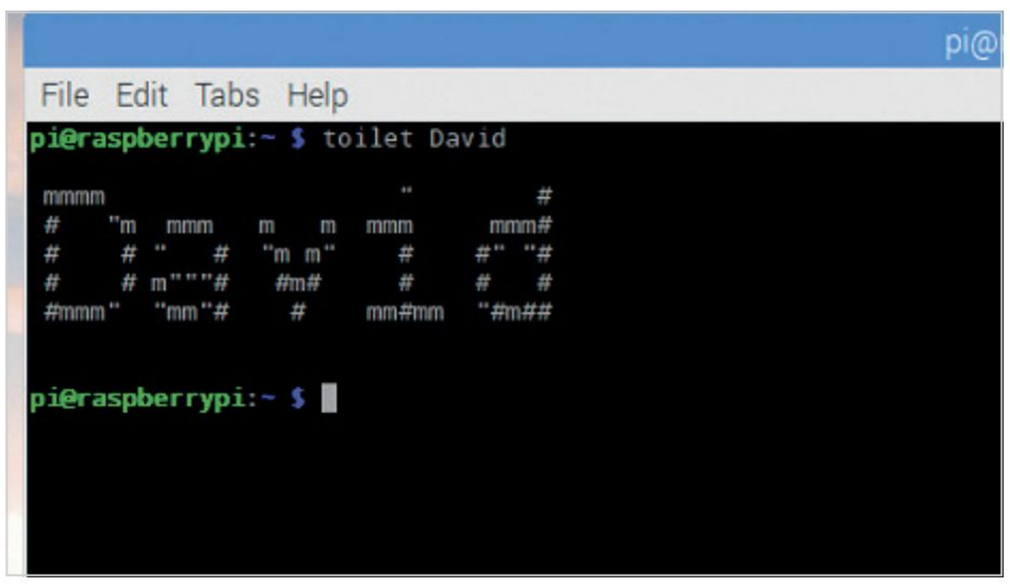
STEP 8

If you really want to expand the whole cow thing, for whatever reason, then pipe the fortune command through it, with: `fortune | cowsay`; and for the graphical cow equivalent: `fortune | xcowsay`. Plus, there's always cowthink. Try: c`owthink ...This book is awesome`.



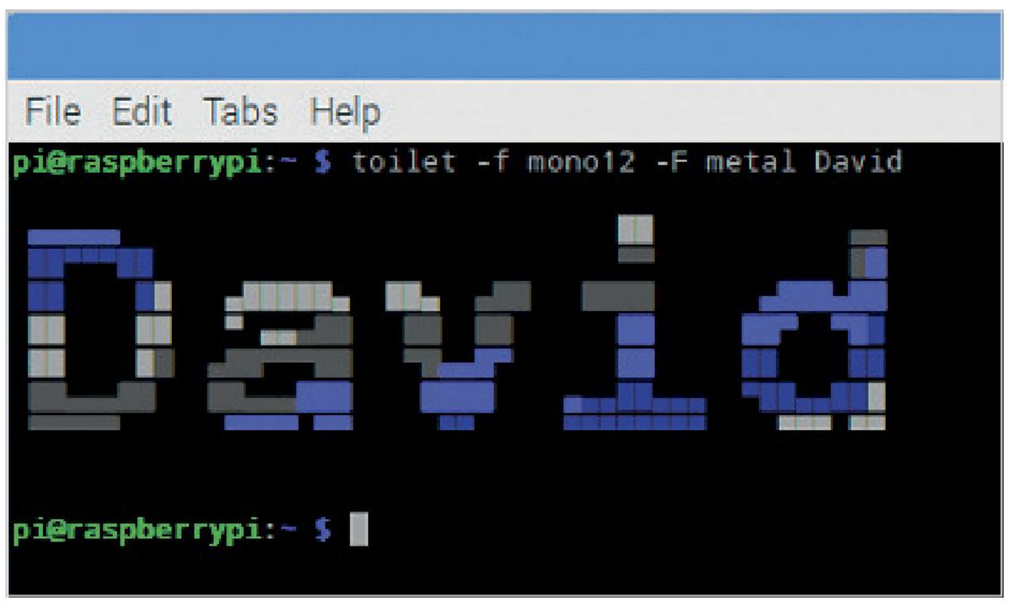
STEP 9

Admittedly, the command 'toilet' doesn't inspire much confidence. However, it's not as bad as it first sounds. Start by installing it with: `sudo apt-get install toilet`, then when it's installed, type something along the lines of: `toilet David`; or perhaps list the contents of the current folder through it, with: `ls | toilet`.



STEP 10

Expanding the toilet command, you can actually generate some decent looking graphics through it. For example, try this: `toilet -f mono12 -F metal David`. You can enter: `toilet --help`, for a list of the command line arguments to expand further.



More Fun Things to Do in the Terminal

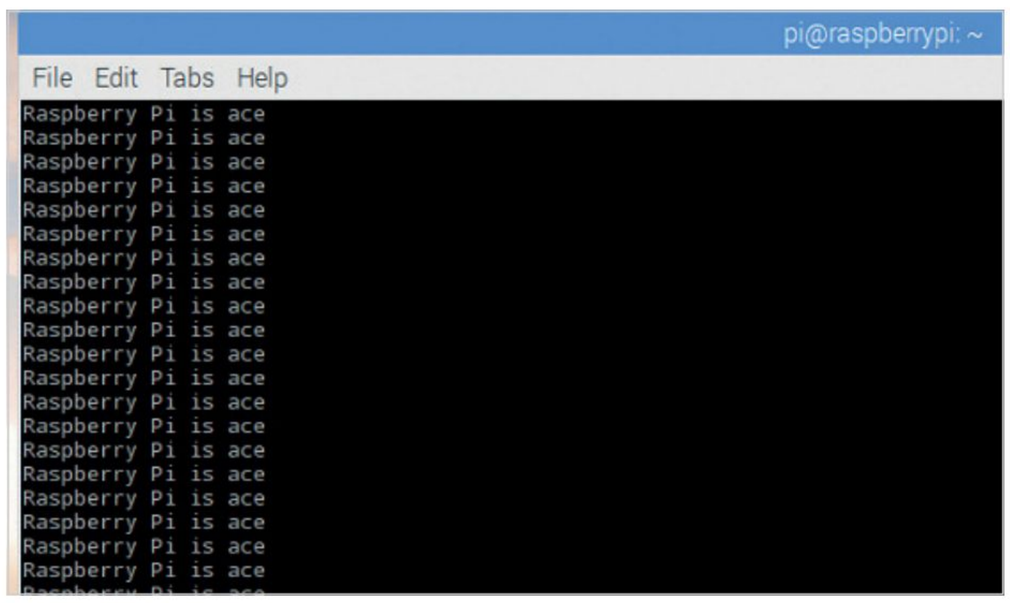
If the previous list of bizarre, and fun things to do in the Terminal has you wanting more, you're in luck. We've put together another batch of both useful, and not so useful, commands for you to try out.

MORE FUN, YAY

Since the Terminal session is already open, and your keyboard digits are nicely warmed up, here are another two pages of Terminal nonsense.

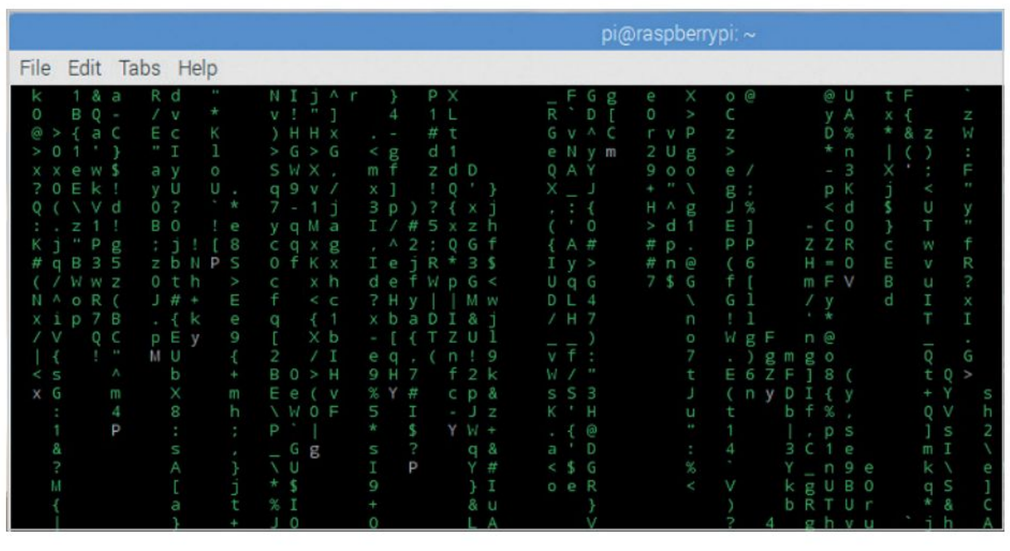
STEP 1

Remember the old ZX Spectrum days of computing, when you could type in 10 print "Hello", 20 goto 10 and Hello would list down the screen? Well, in Raspbian you can do the same. Simply enter: `yes` followed by some text, i.e. `yes Raspberry Pi` is ace. It keeps going until you press Ctrl+C.



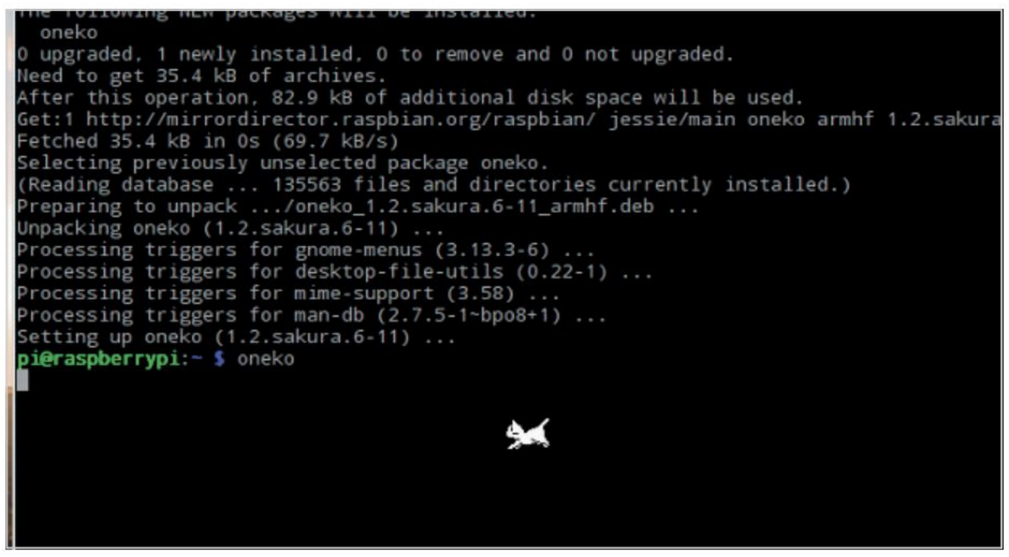
STEP 2

The Matrix was one of the most visually copied films ever released and there's even a version of the Matrix code available for Linux. Install it with: `sudo apt-get install cmatrix`. When it's done enter: cmatrix and follow the white rabbit, Neo. Unlike the real Matrix though, you can press Ctrl+C to exit.



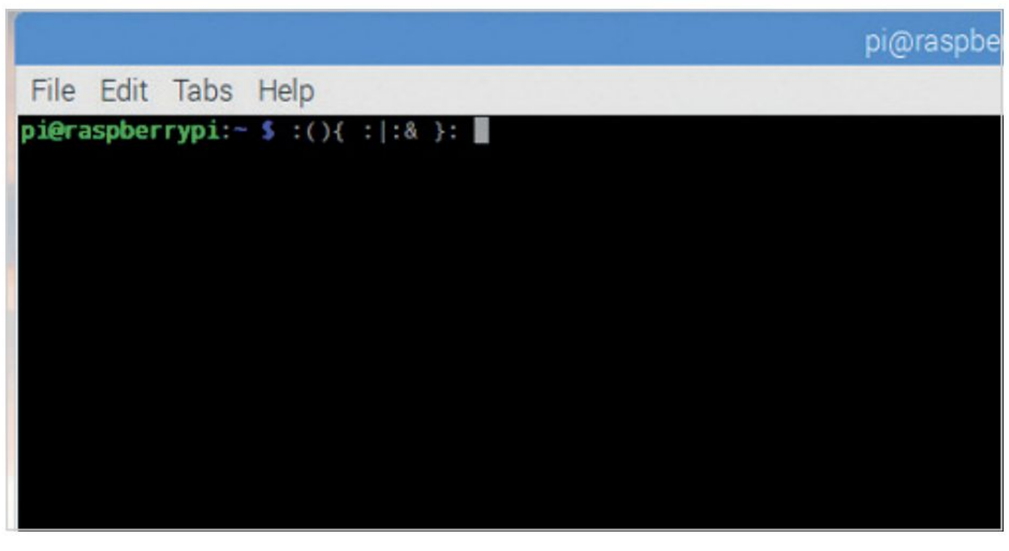
STEP 3

Having a little white cat chase your mouse pointer around the desktop may sound like a terrible waste of time. Oddly though, it isn't. Enter: `sudo apt-get install oneko`, then type: `oneko` to have the cat appear. Move your 'mouse' cursor around the screen and the cat chases it. Use Ctrl+C to exit the action.



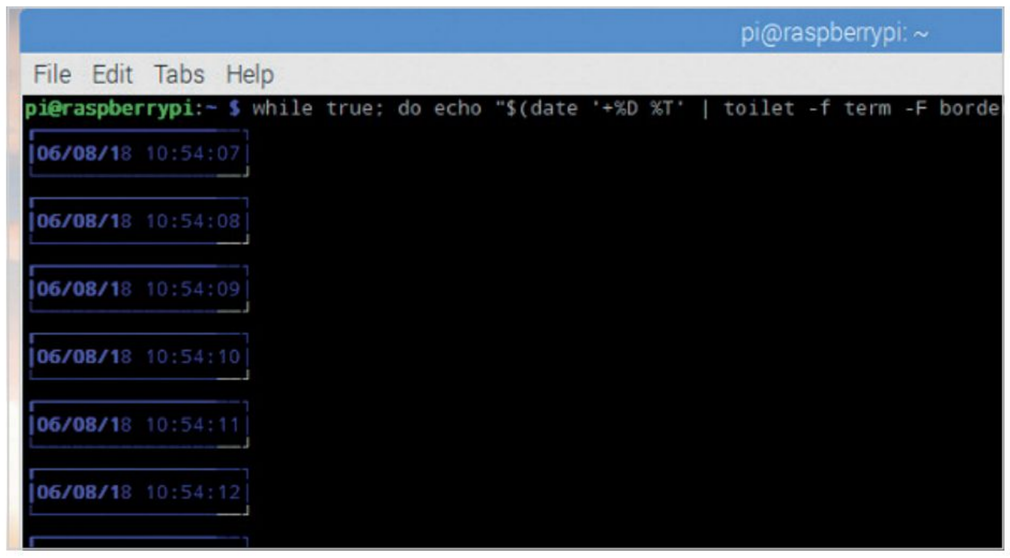
STEP 4

This entry is a little more serious than the previous. Fork Bomb is a command that continually replicates itself until it has used up all the available system resources, eventually causing your computer to crash. You don't have to try it, but it's interesting nonetheless. Simply enter: `:(){ :|:& };:` and be prepared to reboot.



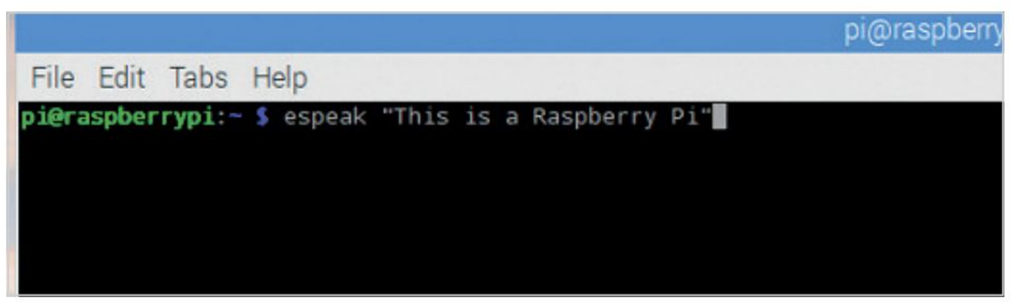
STEP 5

Stringing several commands and piping them through other commands is what makes scripting such a powerful element to an OS. For example, using the while command, together with toilet, can yield some impressive results. Enter: `while true; do echo "$(date '+%D %T' | toilet -f term -F border --metal)"; sleep 1; done`.



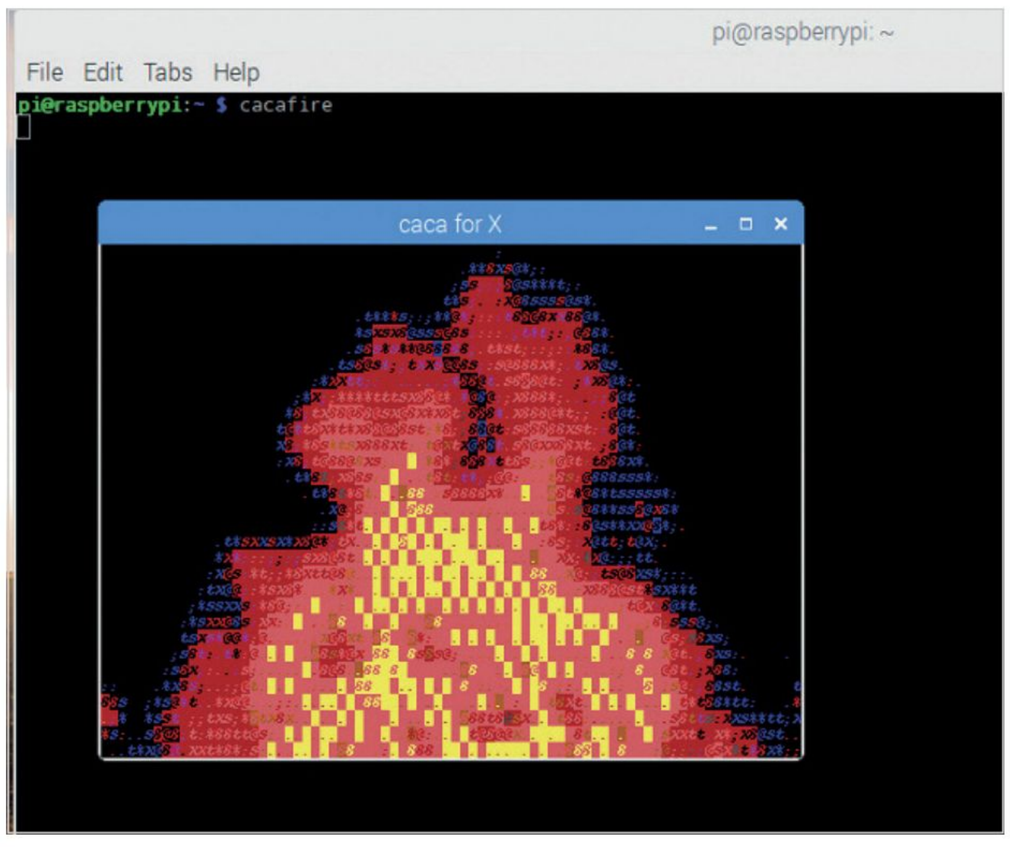
STEP 6

Talking computers were the craze of the '80s. To re-live the fun enter: `sudo apt-get install espeak`, then: `espeak "This is a Raspberry Pi"` to have the computer repeat the text inside the quotes to you. Make sure your volume is turned up and try the following: `ls > folders.txt && espeak -f folders.txt`. This gets Raspbian to read back the contents of the ls command.



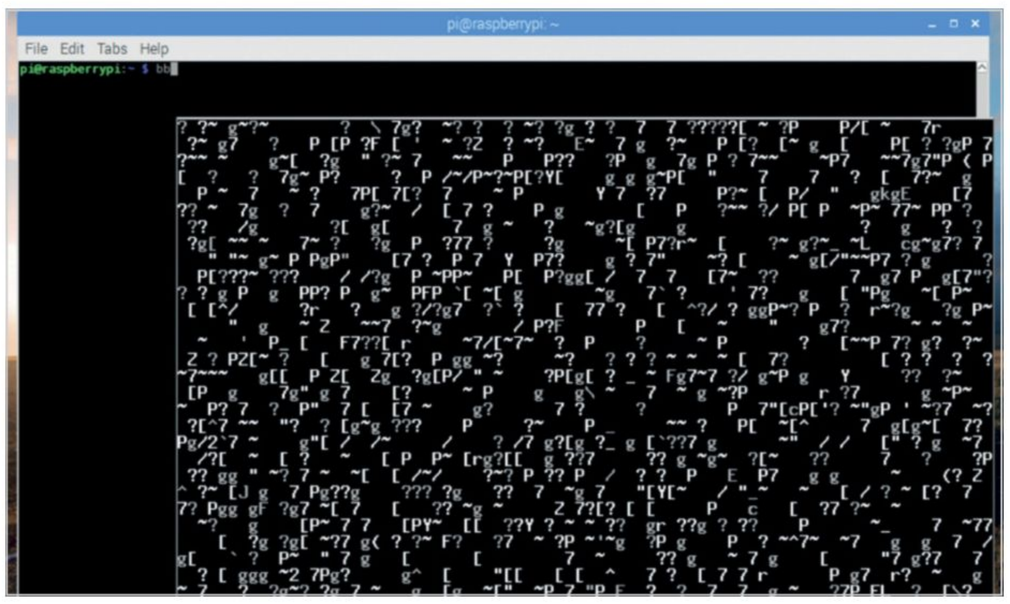
STEP 7

A roaring ASCII fire isn't the most useful command to have at your disposal but it's fun. Install it with: `sudo apt-get install libaa-bin`, then use: `aafire`. It's not exactly warming but you get the idea. To expand the above, enter: `sudo apt-get install bb caca-utils`, then: `cacafire`.



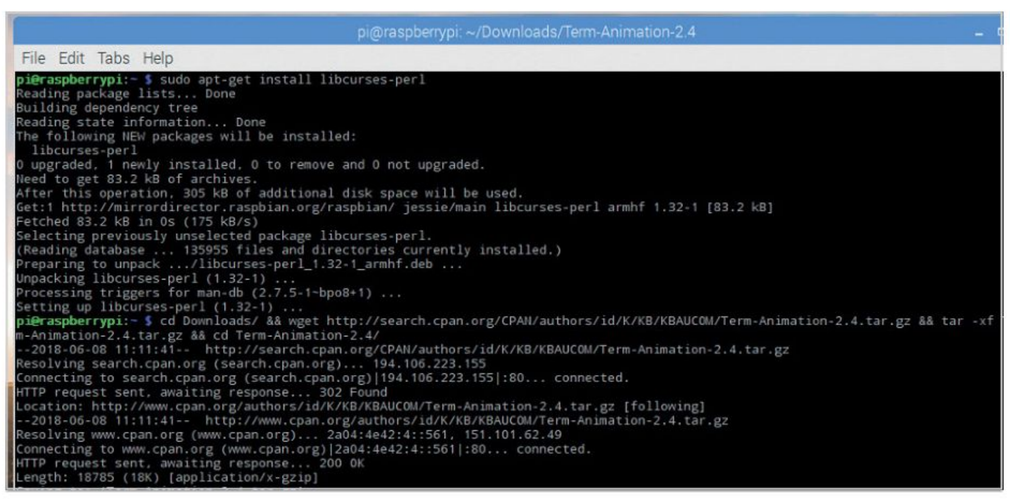
STEP 8

Used as a music demo from the old Amiga and DOS days, the bb command evokes memories of three and a half inch floppies crammed with all manner of demo scene goodies. You've already installed bb from the previous step, so just enter: `bb`. Follow the onscreen instructions and turn up your volume.



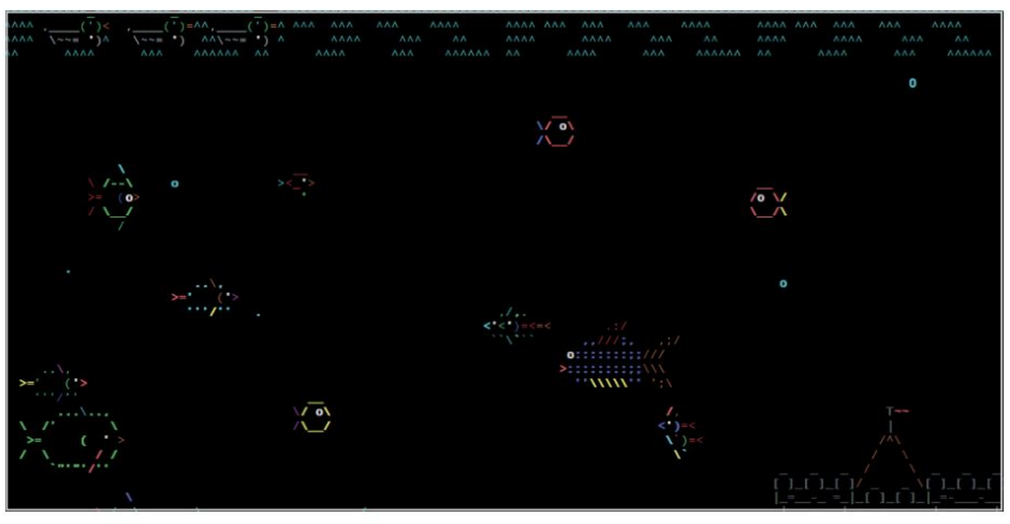
STEP 9

This entry is in two parts. First you need to get hold of the necessary packages: `sudo apt-get install libcurses-perl`. When that's done enter: `cd Downloads/ && wget http://search.cpan.org/CPAN/authors/id/K/KB/KBAUCOM/Term-Animation-2.4.tar.gz && tar -xf Term-Animation-2.4.tar.gz && cd Term-Animation-2.4/. Followed by: perl Makefile.PL && make && make test && sudo make install`.



STEP 10

With that little lot completed, onto the next part. Enter: `cd .. && wget http://www.robobunny.com/projects/asciiquarium/asciiquarium.tar.gz && tar -xf asciiquarium.tar.gz && cd asciiquarium_1.1/ && chmod +x asciiquarium`. Providing all went well, enter: `./asciiquarium` and enjoy your very own ASCII-based aquarium.



Linux Tips and Tricks

The Linux Terminal, you'll no doubt agree, is an exceptional environment and with a few extra apps installed along with a smidgen of command knowledge, incredible and often quite strange things can be accomplished.

TAKING COMMAND

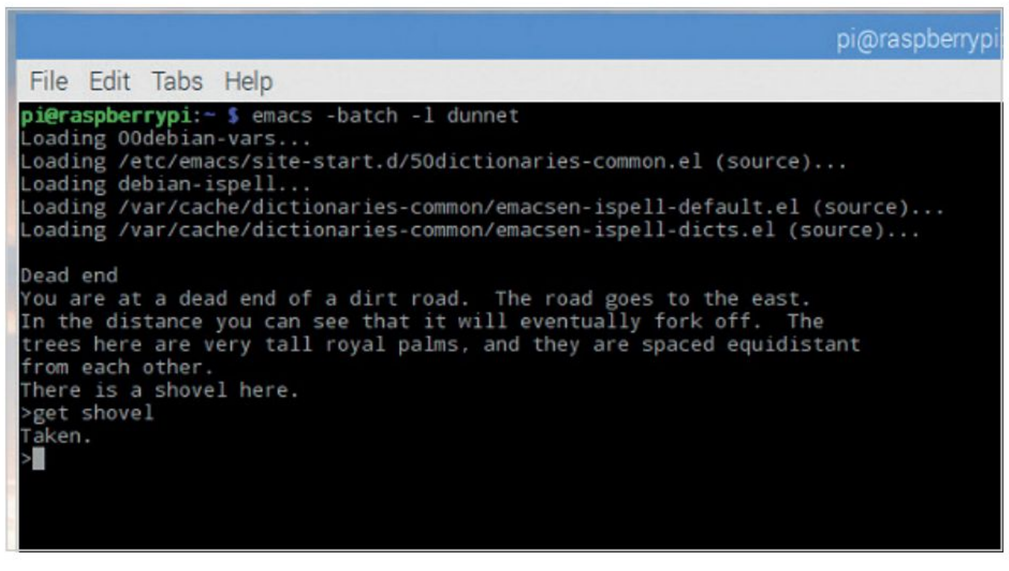
There are countless Linux tips, secrets, hacks and tricks out there. Some are very old, originating from Linux's Unix heritage, while others are recent additions to Linux lore. Here are our ten favourite tips and tricks.

EASTER EGGS

Emacs text editor, is a great piece of software but did you know it also contains a hidden Easter Egg? With Emacs installed (**sudo apt-get install emacs24**), drop to a Terminal session and enter:

```
emacs -batch -l dunnet
```

Dunnet is a text adventure written by Ron Schnell in 1982, and hidden in Emacs since 1994.



TERMINAL BROWSING

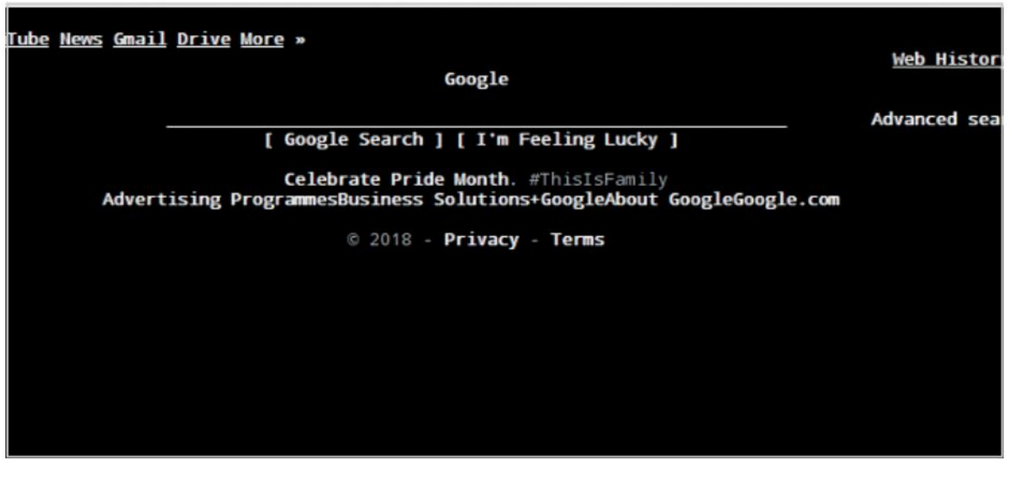
Ever fancied being able to browse the Internet from the Terminal? While not particularly useful, it is a fascinating thing to behold. To do so, enter:

```
sudo apt-get install elinks
```

Then:

```
elinks
```

Enter the website you want to visit.



MOON BUGGY

Based on the classic 1982 arcade game, Moon Patrol, Moon Buggy appeared on home computers in 1985 amid much praise. It's a cracking Atari game available in the Linux Terminal by entering:

```
sudo apt-get install moon-buggy
```

Then:

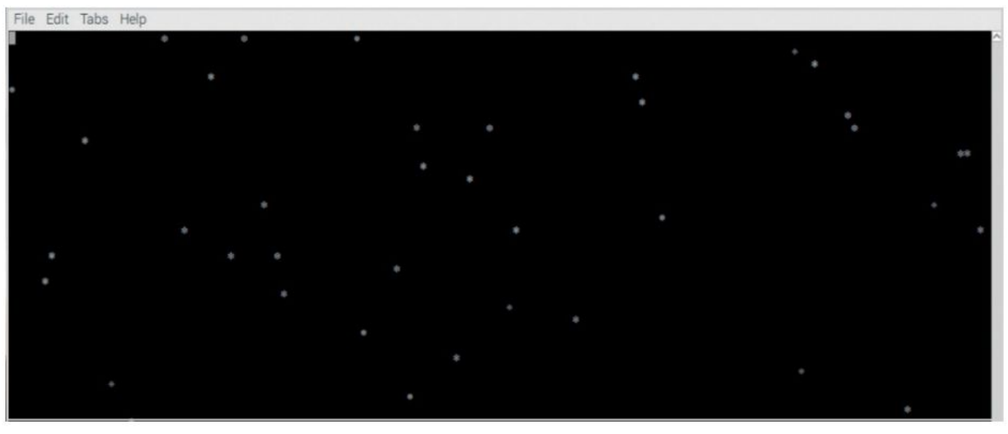
```
moon-buggy
```

Enjoy.

LET IT SNOW

Snowing in the Terminal console isn't something you come across every day. If you're interested, however, enter:

```
wget
https://gist.githubusercontent.com/sontek/1505483/raw/7d024716ea57e69fb52632fee09f42753361c4a2/snowjob.sh
chmod +x snowjob.sh
./snowjob.sh
```

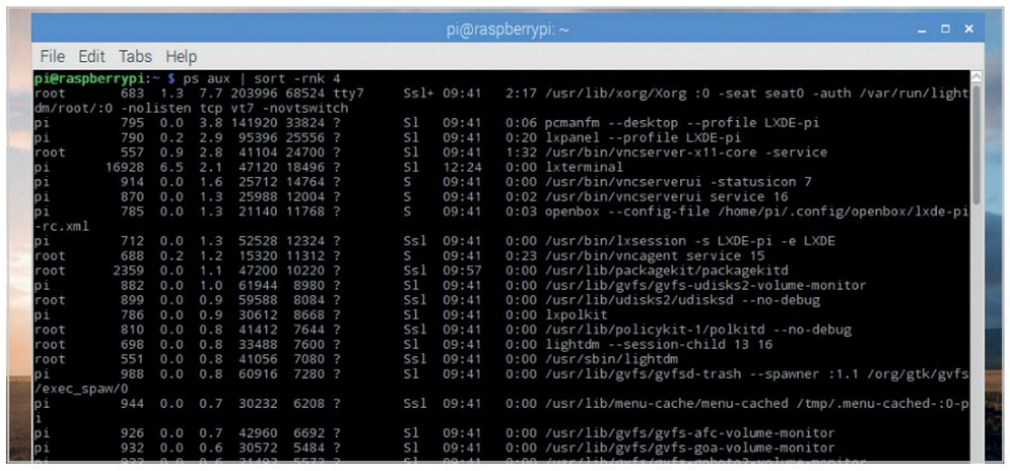


MEMORY HOGS

Memory Hogs – If you need to see which apps are consuming the most memory on your Raspberry Pi, simple enter:

```
ps aux | sort -rnk 4
```

This sorts the output by system memory use.

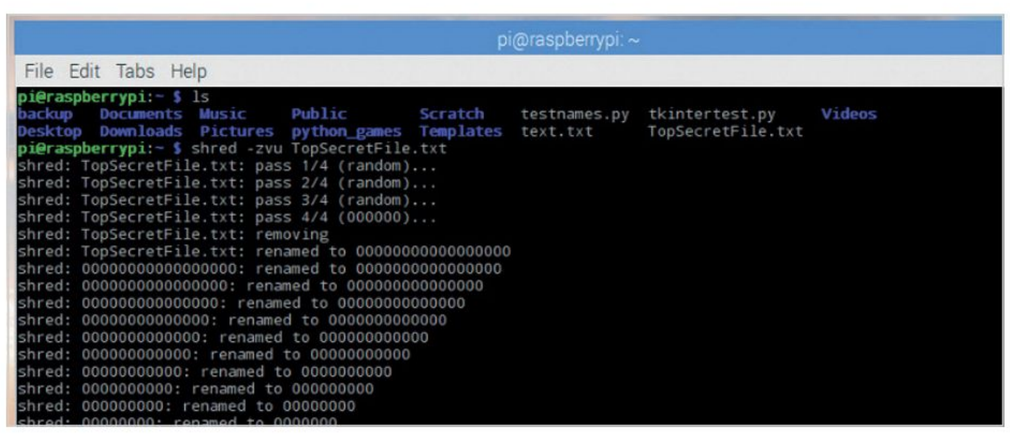


SHREDDER

When you delete a file, there's still a chance of someone with the right software being able to retrieve it. However, files can be securely and permanently deleted using Shred:

```
shred -zvu NAMEOFFILE.txt
```

Replace NAMEOFFILE with the name of the file to delete.

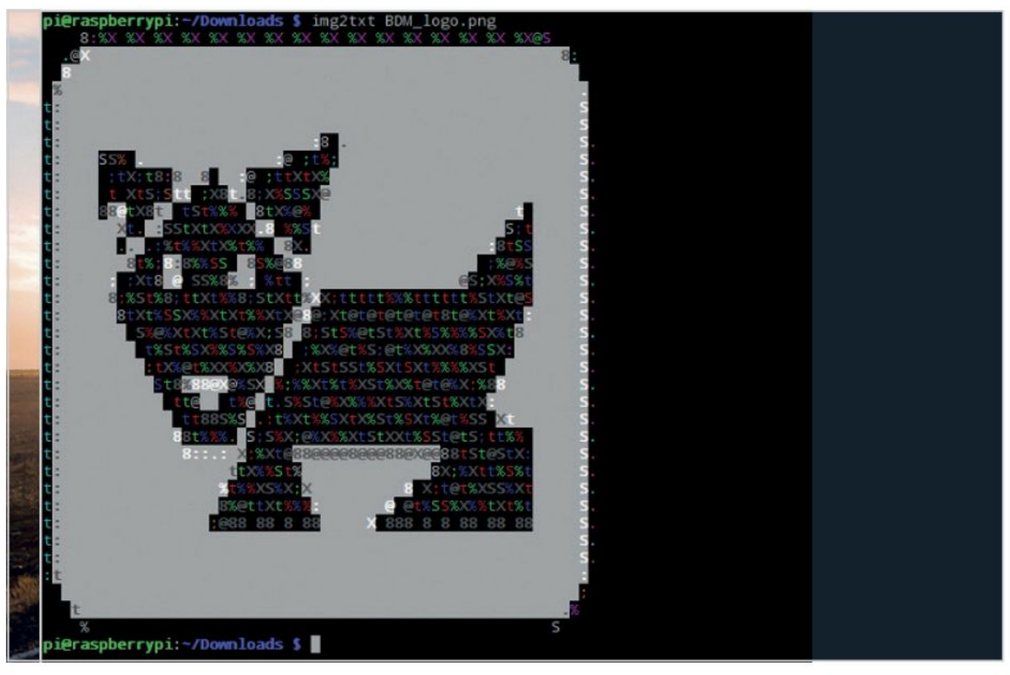


ASCII ART

ASCII art can be quite striking when applied to some images. However, it's often difficult to get just right. You can create some great ASCII art from the images you already have on the Raspberry Pi by using img2txt:

```
img2txt NAMEOFIMAGEFILE.png
```

Replace NAMEOFIMAGEFILE with the actual name of the image file on your system.

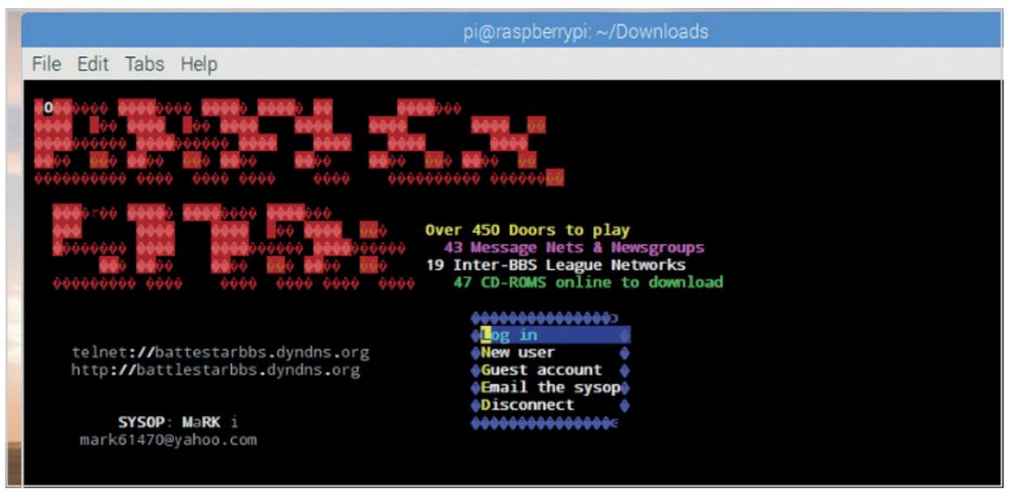


BBS

Back in the days of dial-up connections, the online world was made up of Bulletin Board Systems. These remote servers provided hangouts for users to chat, swap code, play games and more. Using Telnet in Linux, you can still connect to some active BBSes:

```
telnet battlestarbbs.dyndns.org
```

There are countless operational BBSes available; check out **www.telnetbbsguide.com/bbs/list/detail/** for more.

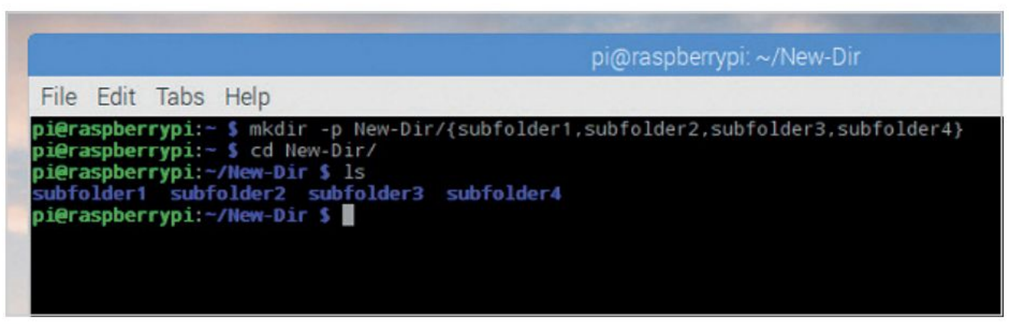


DIRECTORY TREES

If you want to create an entire directory (or folder) tree with a single command, you can use:

```
mkdir -p New-Dir/
{subfolder1,subfolder2,subfolder3,subfolder4}
```

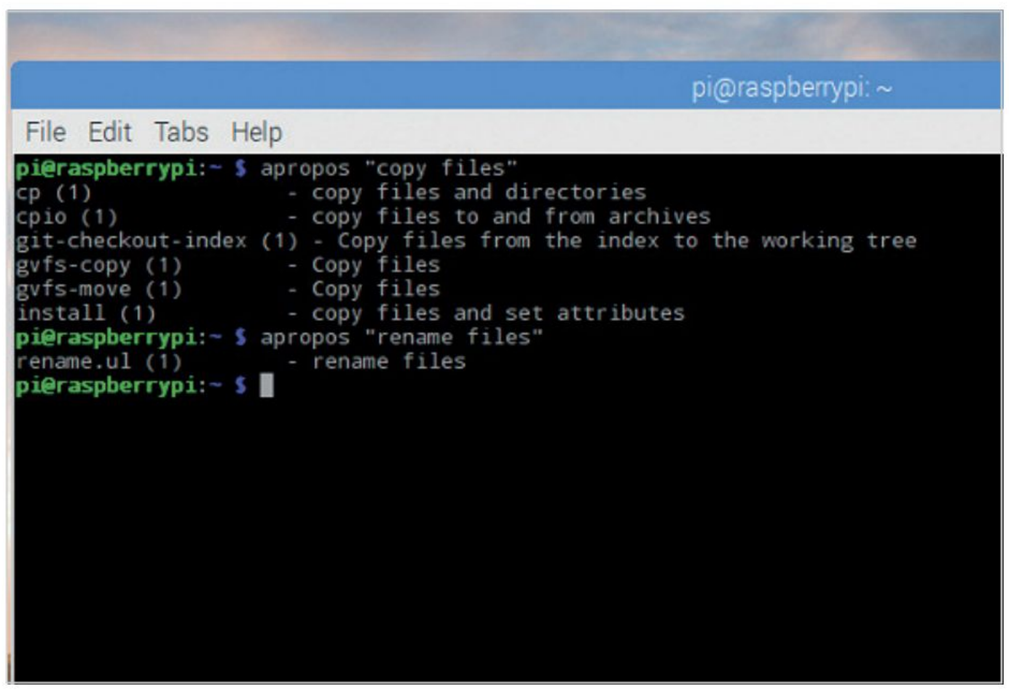
This creates a New-Dir with four sub folders within.



FORGOTTEN COMMANDS

It's not easy trying to remember all the available Linux commands. Thankfully, you can use apropos to help. Simply use it with a description of the command:

```
apropos "copy files"
apropos "rename files"
```



Command Line Quick Reference

When you start using Linux full time, you will quickly realise that the graphical interfaces of Ubuntu, Mint, etc. are great for many tasks but not great for all tasks. Understanding how to use the command line not only builds your understanding of Linux but also improves your knowledge of coding and programming in general. Our command line quick reference guide is designed to help you master Linux quicker.

TOP 10 COMMANDS

These may not be the most common commands used by everyone but they will certainly feature frequently for many users of Linux and the command line.

cd

The `cd` command is one of the commands you will use the most at the command line in Linux. It allows you to change your working directory. You use it to move around within the hierarchy of your file system. You can also use chdir.

ls

The `ls` command shows you the files in your current directory. Used with certain options, it lets you see file sizes, when files where created and file permissions. For example, `ls ~` shows you the files that are in your home directory.

cp

The `cp` command is used to make copies of files and directories. For example, `cp file sub` makes an exact copy of the file whose name you entered and names the copy sub but the first file will still exist with its original name.

pwd

The `pwd` command prints the full pathname of the current working directory (pwd stands for "print working directory"). Note that the GNOME terminal also displays this information in the title bar of its window.

clear

The `clear` command clears your screen if this is possible. It looks in the environment for the terminal type and then in the terminfo database to figure out how to clear the screen. This is equivalent to typing Control-L when using the bash shell.

mv

The `mv` command moves a file to a different location or renames a file. For example `mv file sub` renames the original file to `sub. mv sub ~/Desktop` moves the file 'sub' to your desktop directory but does not rename it. You must specify a new filename to rename a file.

chown

The `chown` command changes the user and/or group ownership of each given file. If only an owner (a user name or numeric user ID) is given, that user is made the owner of each given file, and the files' group is not changed.

cmod

The `chmod` command changes the permissions on the files listed. Permissions are based on a fairly simple model. You can set permissions for user, group and world and you can set whether each can read, write and or execute the file.

rm

The `rm` command removes (deletes) files or directories. The removal process unlinks a filename in a filesystem from data on the storage device and marks that space as usable by future writes. In other words, removing files increases the amount of available space on your disk.

mkdir

Short for "make directory", `mkdir` is used to create directories on a file system, if the specified directory does not already exist. For example, `mkdir` work creates a work directory. More than one directory may be specified when calling mkdir.

USEFUL HELP/INFO COMMANDS

The following commands are useful for when you are trying to learn more about the system or program you are working with in Linux. You might not need them every day, but when you do, they will be invaluable.

free

The `free` command displays the total amount of free and used physical and swap memory in the system. For example, `free -m` gives the information using megabytes.

df

The `df` command displays filesystem disk space usage for all partitions. The command `df-h` is probably the most useful (the -h means human-readable).

top

The `top` program provides a dynamic real-time view of a running system. It can display system summary information, as well as a list of processes.

uname-a

The `uname` command with the `-a` option prints all system information, including machine name, kernel name, version and a few other details.

ps

The `ps` command allows you to view all the processes running on the machine. Every operating system's version of ps is slightly different but all do the same thing.

grep

The `grep` command allows you to search inside a number of files for a particular search pattern and then print matching lines. An example would be: `grep blah file`.

sed

The `sed` command opens a stream editor. A stream editor is used to perform text transformations on an input stream: a file or input from a pipeline.

adduser

The `adduser` command adds a new user to the system. Similarly, the addgroup command adds a new group to the system.

deluser

The `deluser` command removes a user from the system. To remove the user's files and home directory, you need to add the `-remove-home` option.

delgroup

The `delgroup` command removes a group from the system. You cannot remove a group that is the primary group of any users.

man man

The `man man` command brings up the manual entry for the man command, which is a great place to start when using it.

man intro

The `man intro` command is especially useful. It displays the Introduction to User Commands, which is a well written, fairly brief introduction to the Linux command line.

A-Z of Linux Commands

There are literally thousands of Linux commands, so while this is not a complete A-Z, it does contain many of the commands you will most likely need. You will probably find that you end up using a smaller set of commands over and over again but having an overall knowledge is still very useful.

A

`adduser`	Add a new user
`arch`	Print machine architecture
`awk`	Find and replace text within file(s)

B

`bc`	An arbitrary precision calculator language

C

`cat`	Concatenate files and print on the standard output
`chdir`	Change working directory
`chgrp`	Change the group ownership of files
`chroot`	Change root directory
`cksum`	Print CRC checksum and byte counts
`cmp`	Compare two files
`comm`	Compare two sorted files line by line
`cp`	Copy one or more files to another location
`crontab`	Schedule a command to run at a later time
`csplit`	Split a file into context-determined pieces
`cut`	Divide a file into several parts

D

`date`	Display or change the date & time
`dc`	Desk calculator
`dd`	Data Dump, convert and copy a file
`diff`	Display the differences between two files
`dirname`	Convert a full path name to just a path
`du`	Estimate file space usage

E

`echo`	Display message on screen
`ed`	A line oriented text editor (edlin)
`egrep`	Search file(s) for lines that match an extended expression
`env`	Display, set or remove environment variables
`expand`	Convert tabs to spaces
`expr`	Evaluate expressions

F

`factor`	Print prime factors
`fdisk`	Partition table manipulator for Linux
`fgrep`	Search file(s) for lines that match a fixed string
`find`	Search for files that meet a desired criteria
`fmt`	Reformat paragraph text
`fold`	Wrap text to fit a specified width
`format`	Format disks or tapes
`fsck`	Filesystem consistency check and repair

G

`gawk`	Find and Replace text within file(s)
`grep`	Search file(s) for lines that match a given pattern
`groups`	Print group names a user is in
`gzip`	Compress or decompress named file(s)

H

`head`	Output the first part of file(s)
`hostname`	Print or set system name

I

`id`	Print user and group ids
`info`	Help info
`install`	Copy files and set attributes

J

`join`	Join lines on a common field

K

`kill`	Stop a process from running

L

`less`	Display output one screen at a time
`ln`	Make links between files
`locate`	Find files

`logname`	Print current login name
`lpc`	Line printer control program
`lpr`	Off line print
`lprm`	Remove jobs from the print queue

M

`man`	See Help manual
`mkdir`	Create new folder(s)
`mkfifo`	Make FIFOs (named pipes)
`mknod`	Make block or character special files
`more`	Display output one screen at a time
`mount`	Mount a file system

N

`nice`	Set the priority of a command or job
`nl`	Number lines and write files
`nohup`	Run a command immune to hangups

P

`passwd`	Modify a user password
`paste`	Merge lines of files
`pathchk`	Check file name portability
`pr`	Convert text files for printing
`printcap`	Printer capability database
`printenv`	Print environment variables
`printf`	Format and print data

Q

`quota`	Display disk usage and limits
`quotacheck`	Scan a file system for disk usage
`quotactl`	Set disk quotas

R

`ram`	Ram disk device
`rcp`	Copy files between two machines
`rm`	Remove files
`rmdir`	Remove folder(s)
`rpm`	Remote Package Manager
`rsync`	Remote file copy (synchronise file trees)

S

`screen`	Terminal window manager
`sdiff`	Merge two files interactively
`select`	Accept keyboard input
`seq`	Print numeric sequences
`shutdown`	Shutdown or restart Linux
`sleep`	Delay for a specified time
`sort`	Sort text files
`split`	Split a file into fixed-size pieces
`SSH`	Connects to a remote host computer as a specified user, using secure encrypted protocols.
`su`	Substitute user identity
`sudo`	Execute a command as another user, primarily as the Root level, administrator user.
`sum`	Print a checksum for a file
`symlink`	Make a new name for a file
`sync`	Synchronise data on disk with memory

T

`tac`	Concatenate and write files in reverse
`tail`	Output the last part of files
`tar`	Tape Archiver
`tee`	Redirect output to multiple files
`test`	Evaluate a conditional expression
`time`	Measure Program Resource Use
`touch`	Change file timestamps
`top`	List processes running on the system
`traceroute`	Trace Route to Host
`tr`	Translate, squeeze and or delete characters
`tsort`	Topological sort

U

`umount`	Unmount a device
`unexpand`	Convert spaces to tabs
`uniq`	Uniquify files
`units`	Convert units from one scale to another
`unshar`	Unpack shell archive scripts
`useradd`	Create new user account
`usermod`	Modify user account
`users`	List users currently logged in

V

`vdir`	Verbosely list directory contents (`ls -l -b')

W

`watch`	Execute or display a program periodically
`wc`	Print byte, word, and line counts
`whereis`	Report all known instances of a command
`which`	Locate a program file in the user's path
`who`	Print all usernames currently logged in
`whoami`	Print the current user id and name

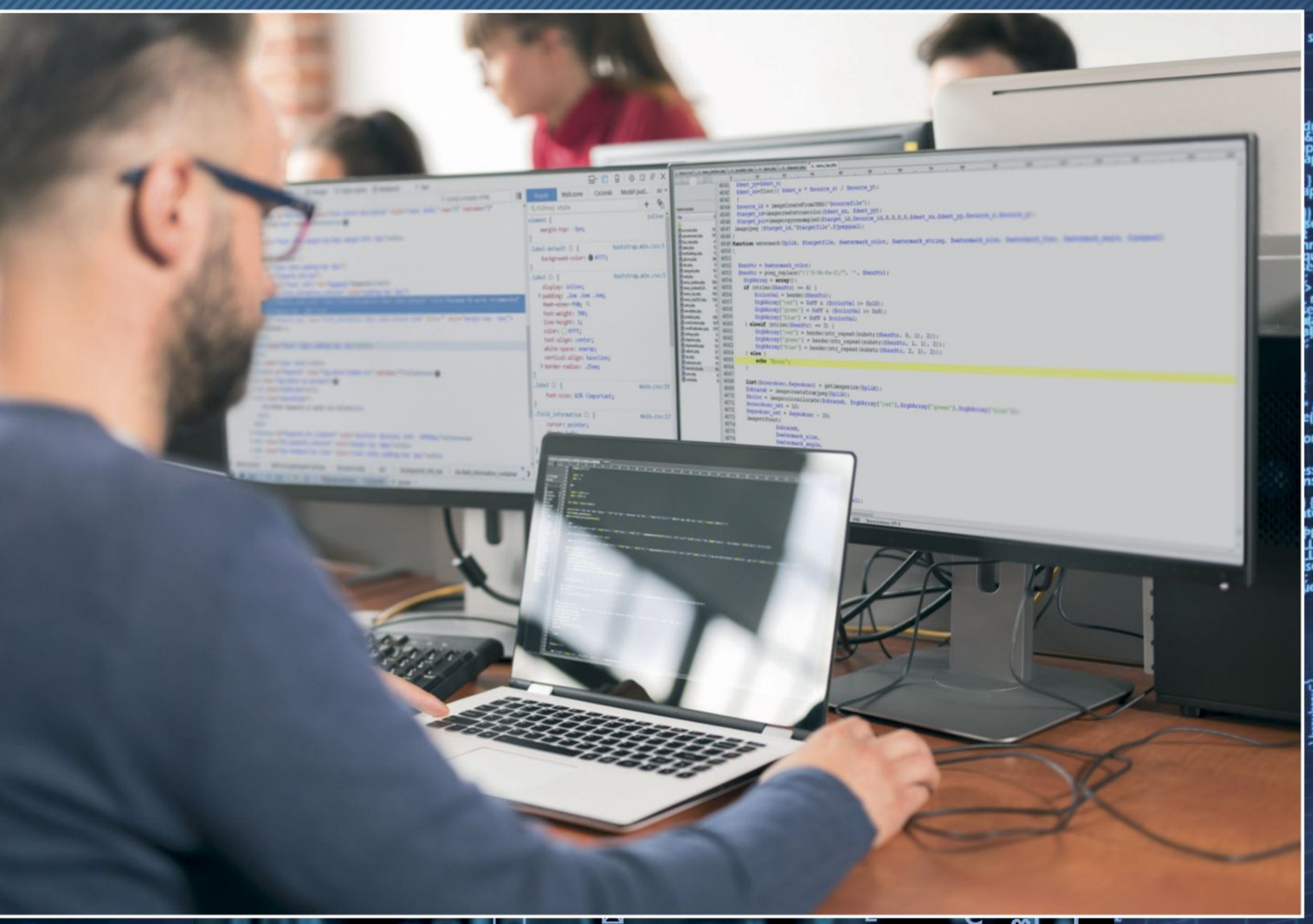
X

`xargs`	Execute utility, passing constructed argument list(s)

Y

`yes`	Print a string until interrupted

Code Repository



SHARE YOUR CODE!

The code listed within this section can be downloaded as a Python file, so you don't have to type it out. Simply visit www.bdmpublications.com/code-portal, sign up for access to the portal and the code is available as a compressed file for you to download and execute.

Maybe you've written something amazing and want to show it off; if so, why not send it in and we can add it to the Code Portal as well as mention it via our social media accounts.

Tell us what the code does, how it works (don't forget to include comments in the code) and what platform to run it on.

Send it in to: enquiries:bdmpublications.com. We look forward to seeing what you've done.

Code Repository

We've included a unique Python code repository for you to freely use in your own programs. There's everything you can think of in here to help you create a superb piece of programming.

We've got code that covers number guessing games, random number generators, Google search code, game code, animation code, graphics code, text adventure code, and even code that plays music stored on your computer.

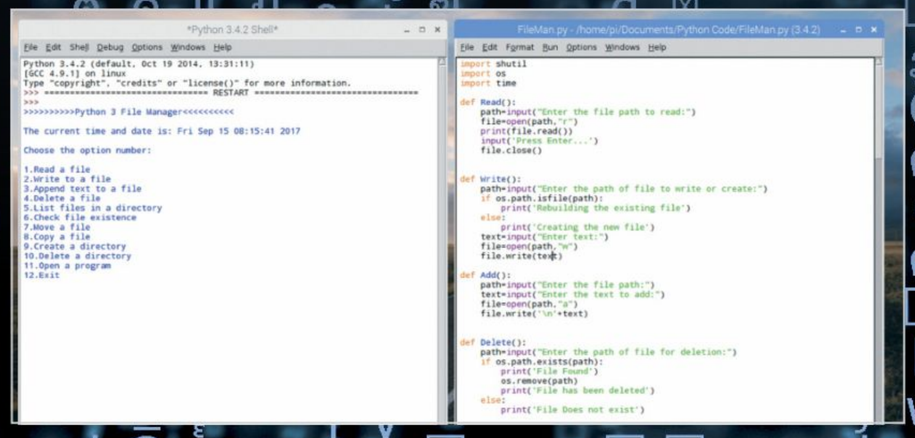
This is an excellent resource that you won't find in any other Python book. So use it, take it apart, adapt it to your own programs, and see what you can create.

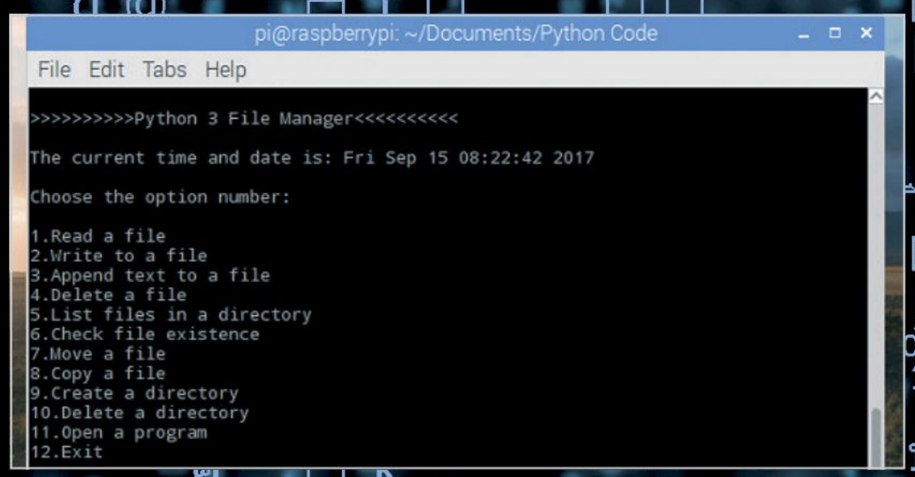
150	Python File Manager
152	Number Guessing Game
154	Polygon Circles
155	Random Number Generator
156	Random Password Generator
157	Keyboard Drawing Script
158	Pygame Text Examples
159	Google Search Script
160	Text to Binary Convertor
162	Text Adventure Script
164	Mouse Controlled Turtle
165	Python Alarm Clock
166	Vertically Scrolling Text
168	Python Digital Clock
170	Python Scrolling Ticker Script
171	Simple Python Calculator
172	Playing Music with the Winsound Module
174	Hangman Game Script

Code Repository

Python File Manager

This file manager program displays a list of options that allow you to read a file, write to a file, append to a file, delete a file, list the contents of a directory and much more. It's remarkably easy to edit and insert into your own code, or add to.





1 This part of the code imports the necessary modules. The OS and Subprocess modules deal with the operating system elements of the program.

2 Each def XXX() functions store the code for each of the menu's options. Once the code within the function is complete, the code returns to the main menu for another option.

3 This is part of the code that checks to see what OS the user is running. In Windows the CLS command clears the screen, whereas in Linux and macOS, the Clear command wipes the screen. If the code tries to run CLS when being used in Linux or macOS, an error occurs, which then prompts it to run the Clear command instead.

4 These are the options, from 1 to 12. Each executes the appropriate function when the relevant number is entered.

FILEMAN.PY

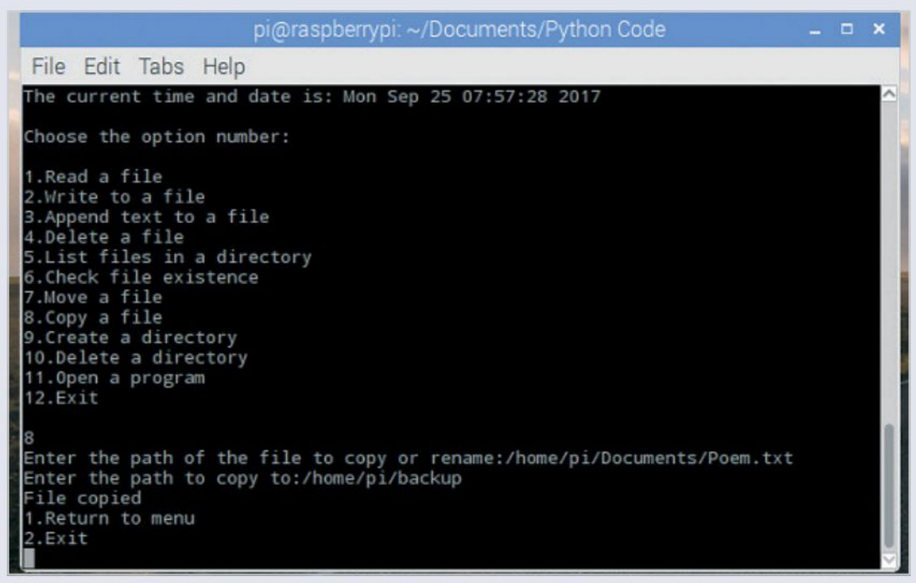
Copy the code below into a New > File and save it as FileMan.py. Once executed it will display the program title, along with the current time and date and the available options.

```
import shutil
import os
import time
import subprocess

def Read():
   path=input("Enter the file path to read:")
   file=open(path,"r")
   print(file.read())
   input('Press Enter...')
   file.close()

def Write():
   path=input("Enter the path of file to write or create:")
   if os.path.isfile(path):
      print('Rebuilding the existing file')
   else:
      print('Creating the new file')
   text=input("Enter text:")
   file=open(path,"w")
   file.write(text)

def Add():
   path=input("Enter the file path:")
   text=input("Enter the text to add:")
   file=open(path,"a")
   file.write('\n'+text)

def Delete():
   path=input("Enter the path of file for deletion:")
   if os.path.exists(path):
      print('File Found')
      os.remove(path)
      print('File has been deleted')
   else:
      print('File Does not exist')

def Dirlist():
   path=input("Enter the Directory path to display:")
   sortlist=sorted(os.listdir(path))
   i=0
   while(i<len(sortlist)):
      print(sortlist[i]+'\n')
      i+=1

def Check():
   fp=int(input('Check existence of \n1.File \n2.
   Directory\n'))
      if fp==1:
         path=input("Enter the file path:")
         os.path.isfile(path)
```

```
      if os.path.isfile(path)==True:
         print('File Found')
      else:
         print('File not found')
   if fp==2:
      path=input("Enter the directory path:")
      os.path.isdir(path)
      if os.path.isdir(path)==False:
         print('Directory Found')
      else:
         print('Directory Not Found')

def Move():
   path1=input('Enter the source path of file to move:')
   mr=int(input('1.Rename \n2.Move \n'))
   if mr==1:
      path2=input('Enter the destination path and file name:')
      shutil.move(path1,path2)
      print('File renamed')
   if mr==2:
      path2=input('Enter the path to move:')
      shutil.move(path1,path2)
      print('File moved')

def Copy():
   path1=input('Enter the path of the file to copy or rename:')
   path2=input('Enter the path to copy to:')
   shutil.copy(path1,path2)
   print('File copied')

def Makedir():
   path=input("Enter the directory name with path to make
   \neg. C:\\Hello\\Newdir \nWhere Newdir is new
   directory:")
   os.makedirs(path)
   print('Directory Created')

def Removedir():
   path=input('Enter the path of Directory:')
   treedir=int(input('1.Deleted Directory \n2.Delete
   Directory Tree \n3.Exit \n'))
   if treedir==1:
      os.rmdir(path)
   if treedir==2:
      shutil.rmtree(path)
      print('Directory Deleted')
   if treedir==3:
      exit()

def Openfile():
   path=input('Enter the path of program:')
   try:
      os.startfile(path)
   except:
      print('File not found')

run=1
while(run==1):
   try:
      os.system('clear')
   except OSError:
      os.system('cls')
   print('\n>>>>>>>>>>Python 3 File Manager<<<<<<<<<<\n')
   print('The current time and date is:',time.asctime())
   print('\nChoose the option number: \n')
   dec=int(input('''1.Read a file
2.Write to a file
3.Append text to a file
4.Delete a file
5.List files in a directory
6.Check file existence
7.Move a file
8.Copy a file
9.Create a directory
10.Delete a directory
11.Open a program
12.Exit

'''))
   if dec==1:
      Read()
   if dec==2:
      Write()
   if dec==3:
      Add()
   if dec==4:
      Delete()
   if dec==5:
      Dirlist()
   if dec==6:
      Check()
   if dec==7:
      Move()
   if dec==8:
      Copy()
   if dec==9:
      Makedir()
   if dec==10:
      Removedir()
   if dec==11:
      Openfile()
   if dec==12:
      exit()
   run=int(input("1.Return to menu\n2.Exit \n"))
   if run==2:
      exit()
```

3

4



Imports

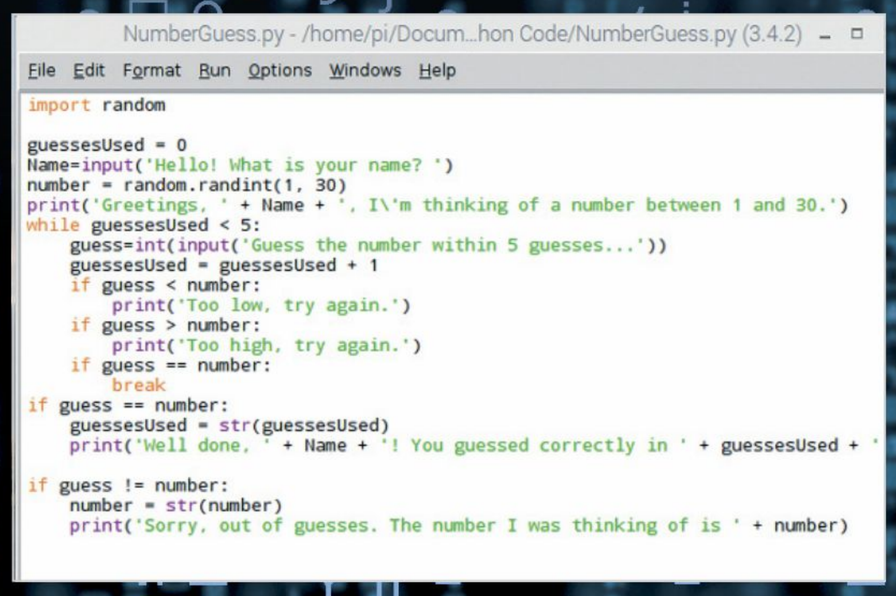
There are three modules to import here: Shutil, OS and Time. The first two deal with the operating system and file management and manipulation; and the Time module simply displays the current time and date.

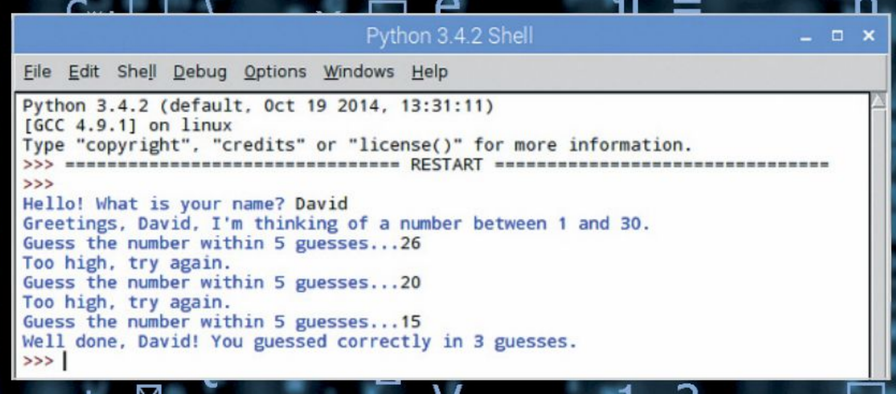
Note how we've included a try and except block to check if the user is running the code on a Linux system or Windows. Windows uses CLS to clear the screen, while Linux uses clear. The try block should work well enough but it's a point of possible improvement depending on your own system.

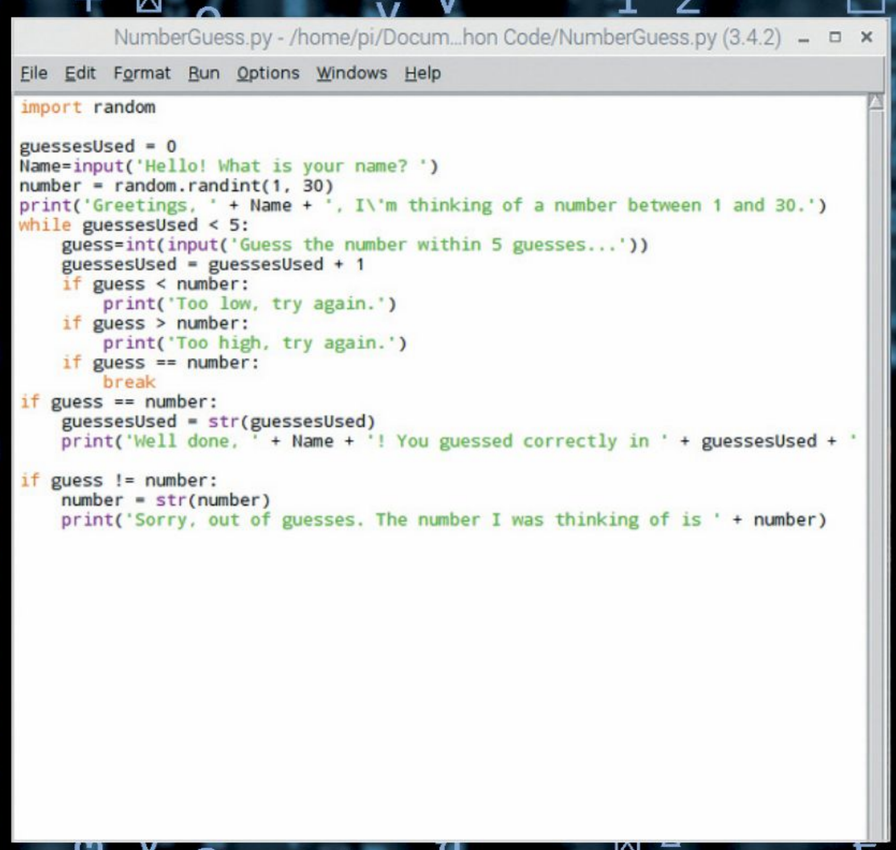
Code Repository

Number Guessing Game

This is a simple little piece of code but it makes good use of the Random module, print and input, and a while loop. The number of guesses can be increased from 5 and the random number range can easily be altered too.







NUMBERGUESS.PY

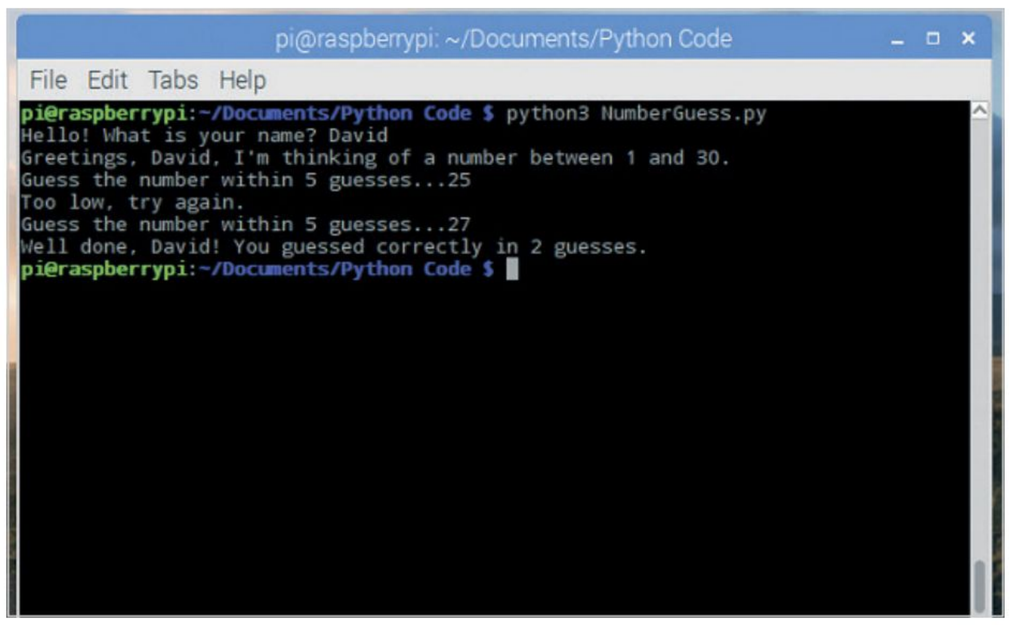
Copy the code and see if you can beat the computer within five guesses. It's an interesting bit of code that can be quite handy when your implementing a combination of the Random module alongside a while loop.

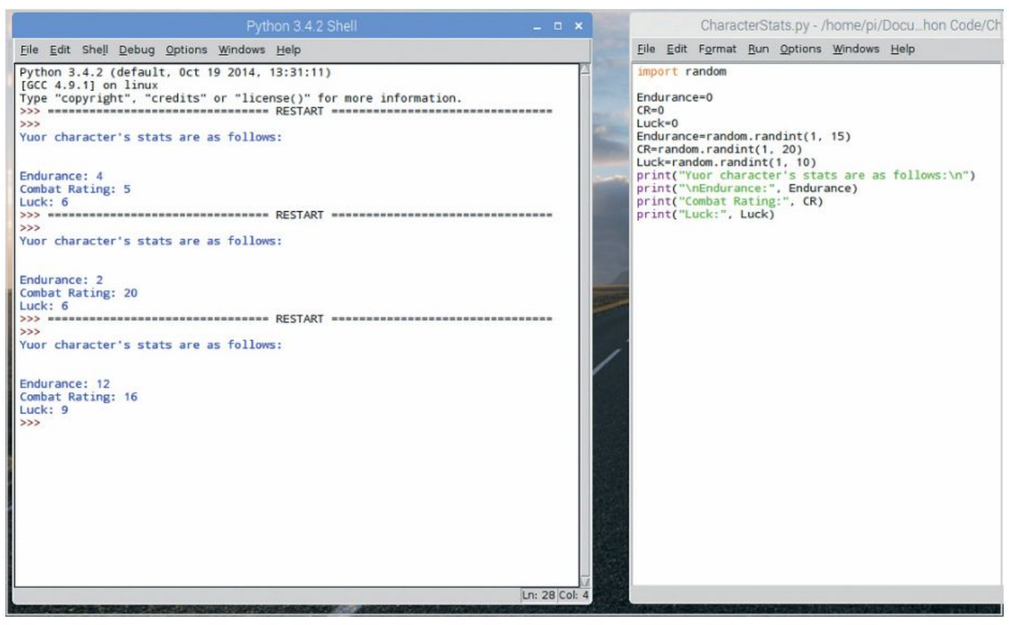
```
import random

guessesUsed = 0
Name=input('Hello! What is your name? ')
number = random.randint(1, 30)
print('Greetings, ' + Name + ', I\'m thinking of a
number between 1 and 30.')
while guessesUsed < 5:
   guess=int(input('Guess the number within 5 guesses...'))
   guessesUsed = guessesUsed + 1
   if guess < number:
      print('Too low, try again.')
   if guess > number:
      print('Too high, try again.')
   if guess == number:
      break
if guess == number:
   guessesUsed = str(guessesUsed)
   print('Well done, ' + Name + '! You guessed
   correctly in ' + guessesUsed + ' guesses.')

if guess != number:
   number = str(number)
   print('Sorry, out of guesses. The number I was
   thinking of is ' + number)
```

1 Although this is a reasonably easy to follow program, there are some elements to the code that are worth pointing out. To begin with, you need to import the Random module, as you're using random numbers within the code.

2 This section of the code creates the variables for the number of guesses used, along with the name of the player, and also sets up the random number between 1 and 30. If you want a wider range of random number selection, then increase the **number=random.randint(1, 30)** end value of 30; don't make it too high though or the player will never be able to guess it. If the player guesses too low or too high, they are given the appropriate output and asked to try again, while the number of guesses is less than five. You can also increase the number of guesses from 5 by altering the **while guessesUsed < 5:** value.

3 If the player guessed the correct number then they are given a 'well done' output, along with how many guesses they used up. If the player runs out of guesses, then the game over output is displayed instead, along with revealing the number the computer was thinking of. Remember, if you do alter the values of the random number chosen by the computer, or the number of guesses the player can take, then along with the variable values, you also need to amend the instructions given in the print statements at the start of the code.





Code Improvements

Since this is such as simple script to apply to a situation, there's plenty of room to mess around with it and make it more interesting. Perhaps you can include an option to take score, the best out of three rounds. Maybe an elaborate way to congratulate the player for getting a 'hole in one' correct guess on their first try.

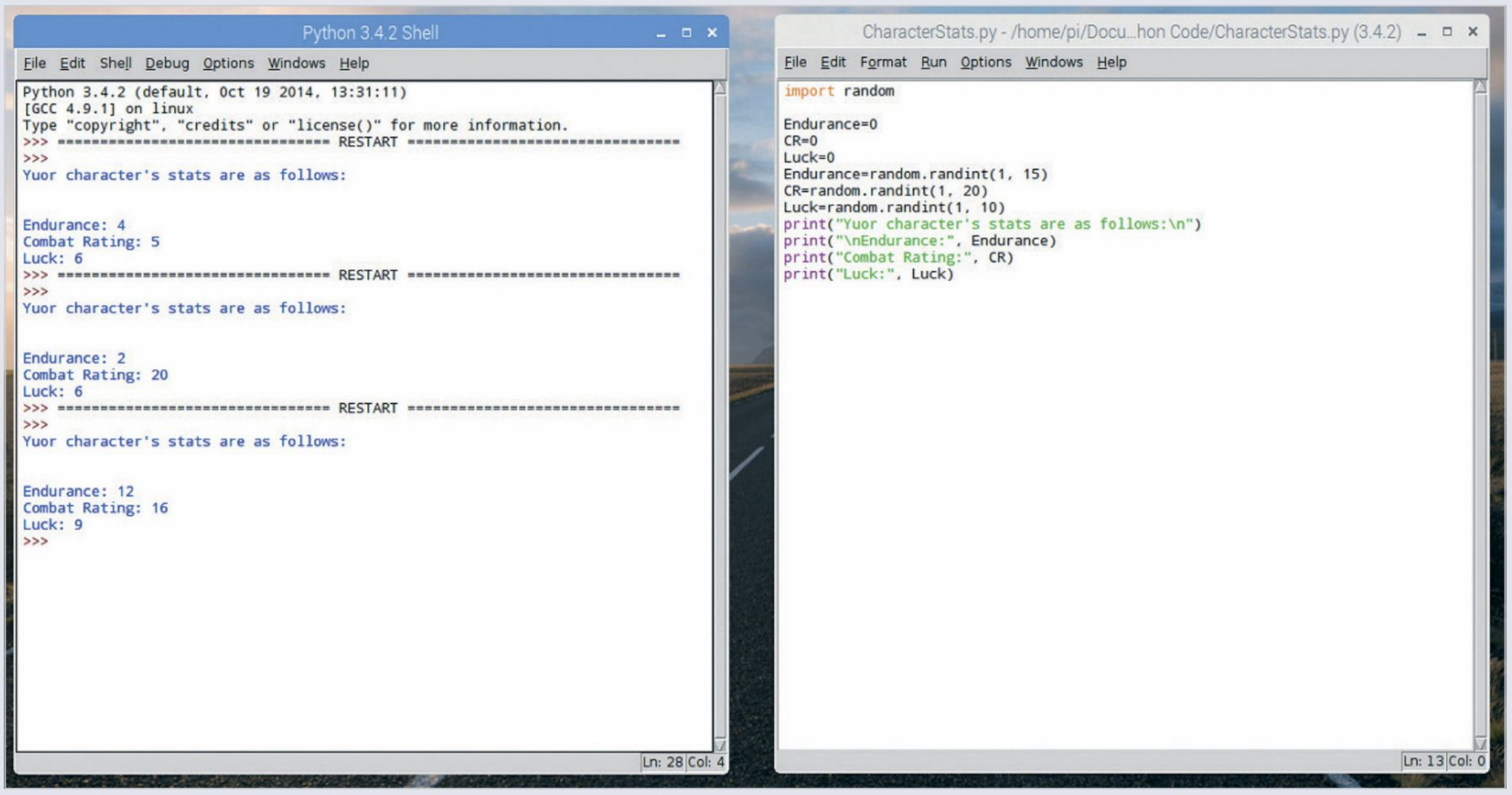
Moreover, the number guessing game code does offer some room for implementing into your code in a different manner. What we mean by this is, the code can be used to retrieve a random number between a range, which in turn can give you the start of a character creation defined function within an adventure game.

Imagine the start of a text adventure written in Python, where the player names their character. The next step is to roll the virtual random dice to decide what that character's combat rating, strength, endurance and luck values are. These can then be carried forward into the game under a set of variables that can be reduced or increased depending on the circumstances the player's character ends up in.

For example, as per the screenshot provided, you could use something along the lines of:

```
Endurance=0
CR=0
Luck=0
Endurance = random.randint(1, 15)
CR = random.randint(1, 20)
Luck = random.randint(1, 10)
Print("Your character's stats are as follows:\n")
Print("Endurance:", Endurance)
Print("Combat Rating:", CR)
Print("Luck:", Luck)
```

The player can then decide to either stick with their roll or try again for the hope of better values being picked. There's ample ways in which to implement this code into a basic adventure game.



Code Repository

Polygon Circles

Here's a fun, and mathematical, look at making a circle from straight lines. Using the Math module, in particular sin, cos and Pi, this code will draw a series of straight lines using the Turtle module. The end result is quite remarkable and has plenty of scope for further exploration.

POLYGONCIRCLES.PY

There's lots of mathematics used here along with some intricate coordinate manipulation with the Turtle module. Enter the code and execute it to see how it turns out.

```
from turtle import*
from math import sin, cos, pi
r=200
inc=2*pi/100
t=0;n=1.5
for i in range(100):
   x1=r*sin(t);  y1=r*cos(t)
   x2=r*sin(t+n);y2=r*cos(t+n)
   penup();  goto(x1,y1)
   pendown();goto(x2,y2)
   t+=inc
```

Graphical Enhancements

There are several ways in which you can improve this code to make it more interesting. You can insert colours, perhaps a different colour for every line. You can display a message inside the circle and have the Turtle draw around it. Let your imagination run wild on this one.

Turtle's graphics can take a while to map out and draw, depending on how big and how intricate an image it is you're designing. Whilst the effect can be quite stunning, it is limited by the amount of time it takes to display an image. Therefore it's worth seeing if the function turtle.speed() will quicken things up.

Turtle.speed() comes in various values:

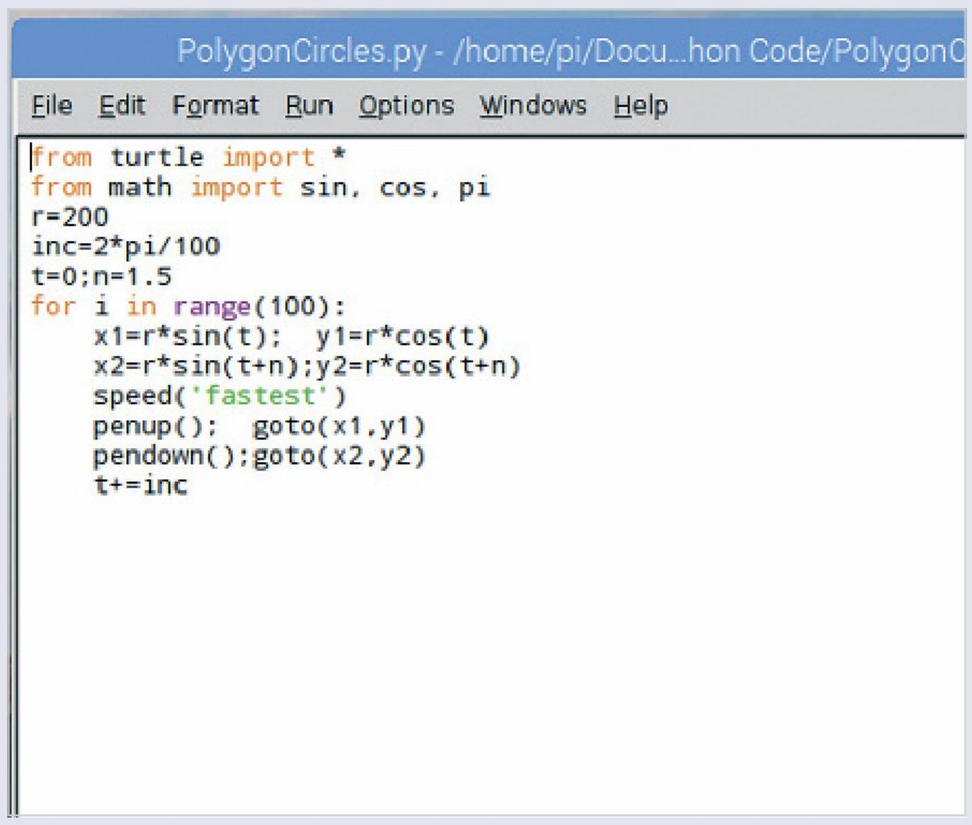
```
slowest
slow
normal
fast
fastest
```

You can experiment with the various speeds by adding the function in the for loop, just before the penup line.

For example:

```
for i in range(100):
   x1=r*sin(t);  y1=r*cos(t)
   x2=r*sin(t+n);y2=r*cos(t+n)
   speed('fastest')
   penup();  goto(x1,y1)
   pendown();goto(x2,y2)
   t+=inc
```

This will run through the code at the 'fastest' speed possible for the Turtle. It certainly makes a difference and is worth considering if you're drawing Turtle images for games or presentations.



Random Number Generator

User input and the ability to manipulate that input are important elements with any programming language. It's what separates a good program from a great program, one that allows the user to interact and see the results of that interaction.

RNDNUMGEN.PY

It might be simple but this little piece of code will ask the user for two sets of numbers, a start and a finish. The code will then pluck out a random number between the two sets and display it.

```
from random import *

print("\n>>>>>>>>>>Random Number Generator<<<<<<<<<<\n")
nmb1=int(input("Enter the start number: "))
nmb2=int(input("Enter the last number: "))

x = randint(nmb1, nmb2)
print("\nThe random number between",nmb1,"and",nmb2,"is:\n")
print(x)
```

More Input

While an easy code to follow, it could be more interesting if you prompt the user for more input. Perhaps you can provide them with addition, subtraction, multiplication elements with their numbers. If you're feeling clever, see if you can pass the code through a Tkinter window or even the Ticker window that's available on Page 128.

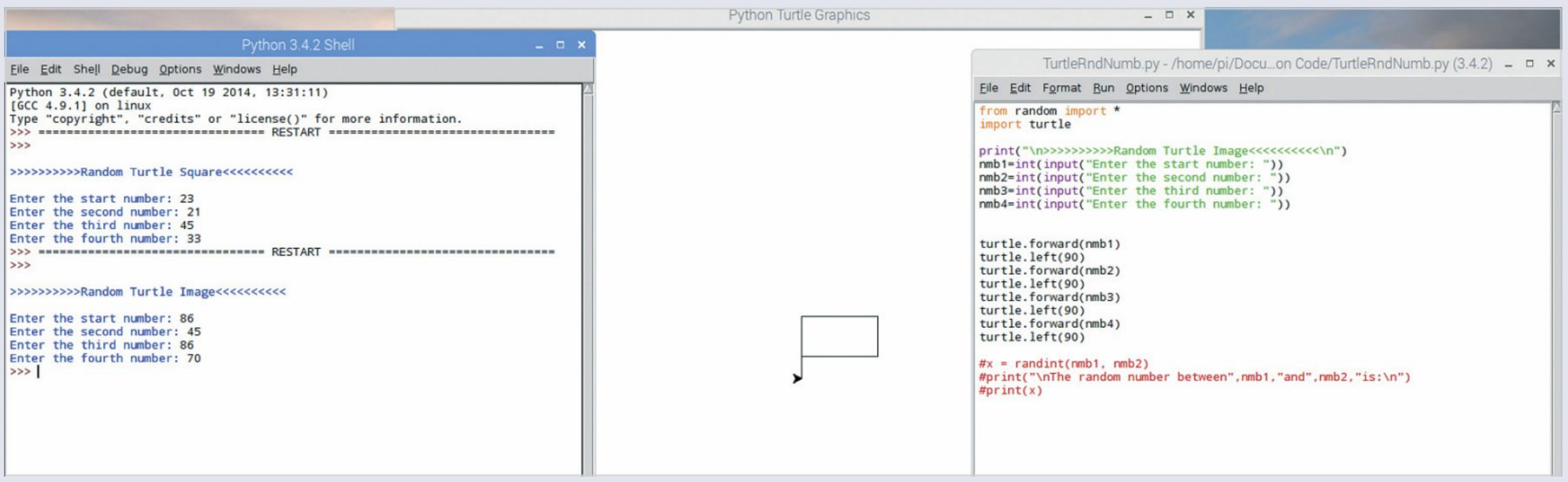
Furthermore, the core of the code can be used in a text adventure game, where the character fights something and their health, along with the enemy's, is reduced by a random number. This can be mixed with the previous code from Page 90's Number Guessing Game, where we defined the stats for the adventure game's character.

You can also introduce the Turtle module into the code and perhaps set some defined rules for drawing a shape, object or something based on a user inputted random value from a range of numbers. It takes a little working out but the effect is certainly really interesting.

For example, the code could be edited to this:

```
from random import *
import turtle

print("\n>>>>>>>>>>Random Turtle Image<<<<<<<<<<\n")
nmb1=int(input("Enter the start number: "))
nmb2=int(input("Enter the second number: "))
nmb3=int(input("Enter the third number: "))
nmb4=int(input("Enter the fourth number: "))


turtle.forward(nmb1)
turtle.left(90)
turtle.forward(nmb2)
turtle.left(90)
turtle.forward(nmb3)
turtle.left(90)
turtle.forward(nmb4)
turtle.left(90)
```

Whilst it's a little rough around the edges, you can easily make it more suitable.



Code Repository

Random Password Generator

We're always being told that our passwords aren't secure enough; well here's a solution for you to implement into your own future programs. The random password generator code below will create a 12-letter string of words (both cases) and numbers each time it's executed.

RNDPASSWORD.PY

Copy the code and run it; each time you'll get a random string of characters that can easily be used as a secure password which will be incredibly difficult for a password cracker to hack.

```
import string
import random

def randompassword():
   chars=string.ascii _ uppercase + string.ascii _
   lowercase + string.digits
   size= 8
   return ''.join(random.choice(chars) for x in
   range(size,20))

print(randompassword())
```

Secure Passwords

There's plenty you can do to modify this code and improve it further. For one, you can increase the number of characters the generated password displays and perhaps you can include special characters too, such as signs and symbols. Then, you can output the chosen password to a file, then securely compress it using the previous random number generator as a file password and send it to a user for their new password.

An interesting aspect to this code is the ability to introduce a loop and print any number of random passwords. Let's assume you have a list of 50 users for a company and you're in charge of generating a random password for them each month.

```
Python 3.4.2 Shell
File  Edit  Shell  Debug  Options  Windows  Help
>>> ================================ RESTART ================================
>>>
P9nLS9MFXLIbCq1z
QfnQRjt5qf8pjdPT
mfDGBK1Kcv0LilHR
Ri967mcVqChvoHdv
RoLZWVDNlqiCXTlK
znsSBooDsDL4TcVD
KvHKy616fIJ5dxHE
SPSk77QPZnE20Cm7
8DWHFcubPOXMi3lI
UuCR5GhxFL4fWP50
BCyVkma09Qrp5MKc
C2X7add0CsXGx0at
O5FvZl5oCHApT7Bx
WYBbbzy3nPqHyTWb
2PfTnUv3fzgNbBqH
H820ULLPkXbElL2u
y57cKCE7IWXBkNNe
t0DdzlQDuSWYCSga
225Hpidc1tdLXP4v
TnJApwX0uiMlL6EC
MWmA0WGHE8pqI4fe
9zYtByYRgs0zS2d1
2N0xyped208a5ME8
Fjd2145MIhwgKNTF
rTYN44th0xP0KJz0
l3hM40ZqMcGs6L76
pntySQlVTRJfC7kN
iazbaL4SK1Yz9cSc
s1hzoLLqG2I2kx2z
TiLtPlJCzXcdrYZ8
onSV3wL0Qni5KPXI
```

Adding a loop to print a password fifty times is extremely easy, for example:

```
import string
import random


def randompassword():
   chars=string.ascii _ uppercase + string.ascii _
   lowercase + string.digits
   size= 4
   return ''.join(random.choice(chars) for x in
   range(size,20))

n=0
while n<50:
   print(randompassword())
   n=n+1
```

This will output fifty random passwords based on the previous random selection of characters.

```
RndPasswordLoop.py - /home/pi/D... Code/RndPasswordLoop.py (3.4.2)
File  Edit  Format  Run  Options  Windows  Help
import string
import random


def randompassword():
    chars=string.ascii_uppercase + string.ascii_lowercase + string.digits
    size= 4
    return ''.join(random.choice(chars) for x in range(size,20))

n=0
while n<50:
    print(randompassword())
    n=n+1
```

Keyboard Drawing Script

The Turtle module is an excellent resource for the Python programmer. However, what makes it more interesting, is its ability to enable the user to control the turtle on the screen. This piece of code does exactly that, allowing the user to unleash their inner artist.

KEYBDRAW.PY

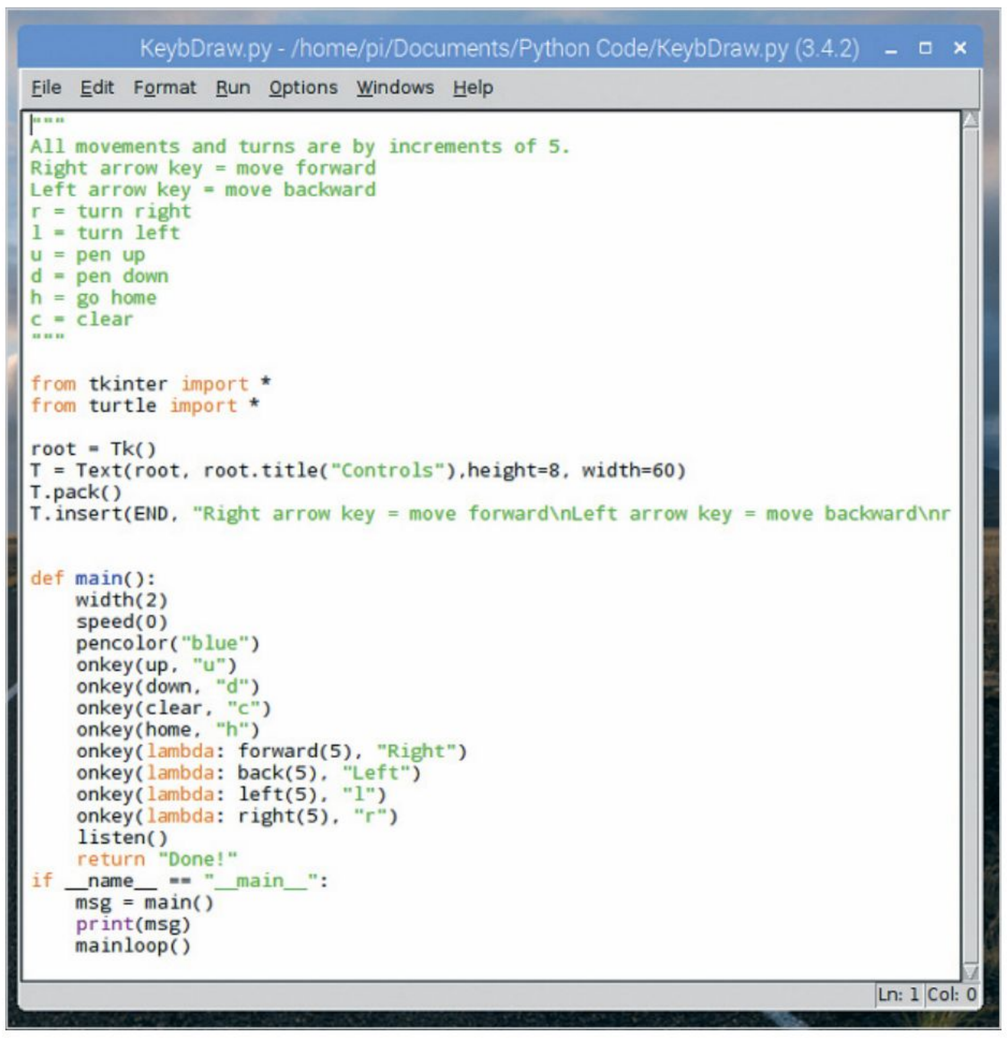
There are two modules in this script: Turtle and Tkinter. The Turtle module is the main display, where the user controls the drawing, whereas Tkinter simply displays the user controls.

```
"""
All movements and turns are by increments of 5.
Right arrow key = move forward
Left arrow key = move backward
r = turn right
l = turn left
u = pen up
d = pen down
h = go home
c = clear
"""

from tkinter import *
from turtle import *

root = Tk()
T = Text(root, root title("Controls"),height=8, width=60)
T.pack()
T.insert(END, "Right arrow key = move forward\
nLeft arrow key = move backward\nr = turn right\nl
= turn left\nu = pen up\nd = pen down\nh = go home\
nc = clear")


def main():
    width(2)
    speed(0)
    pencolor("blue")
    onkey(up, "u")
    onkey(down, "d")
    onkey(clear, "c")
    onkey(home, "h")
    onkey(lambda: forward(5), "Right")
    onkey(lambda: back(5), "Left")
    onkey(lambda: left(5), "l")
    onkey(lambda: right(5), "r")
    listen()
    return "Done!"
if __name__ == "__main__":
    msg = main()
    print(msg)
    mainloop()
```

1

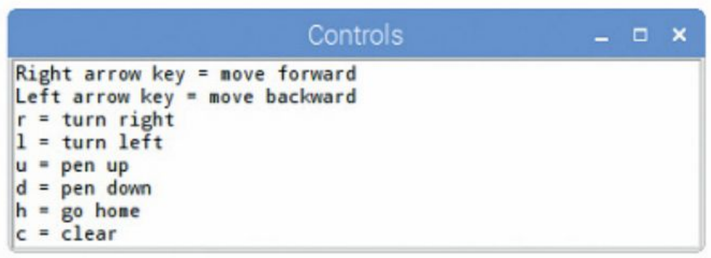
2

3



Artwork

Just as with all code, there's always room for improvement somewhere. Here you could change the colours or ask the user which colour they want to start with and then include a key in the controls to change the pen colour whilst drawing. There's also room to increase or decrease the speed of the pen, again that could be a user-defined speed. You can also expand the controls thoroughly to include a lot more detail and options.



1 An initial set of comments to display the controls, which can also be outputted as a separate tkinter window if you want, and importing the necessary modules: tkinter and turtle.

2 Setting up the separate tkinter window displaying the controls and creating a function to define the width, speed and colour of the line being drawn; also setting up the onkey functions for the controls of the pen.

3 A neat way to finalise the code and loop it so you can continue drawing with the pen.

Code Repository

Pygame Text Examples

There's a lot you can do using the Pygame module that's not related to displaying graphics. The module contains many functions which can manipulate text, such as flipping it, displaying it sideways, upside down and even rotating it with animations.

TXTROT.PY

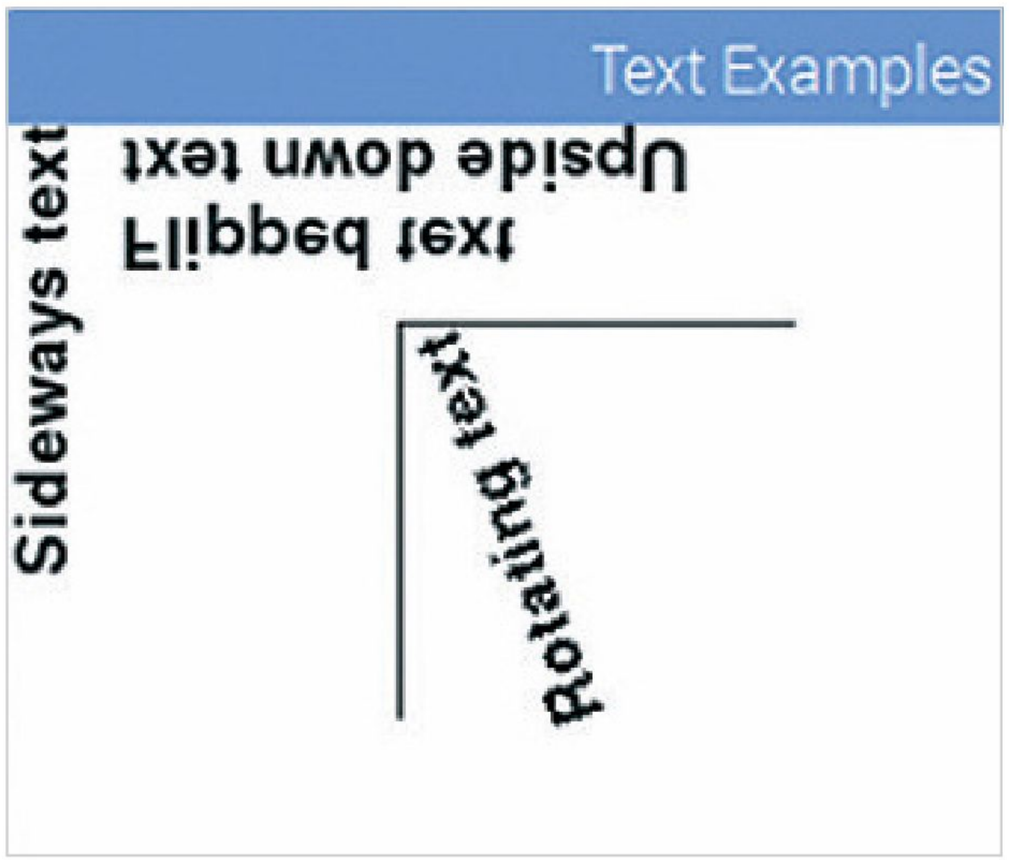
Here we've introduced several examples of displaying text within the Pygame module. Each is easily recognised within the code, so you can pull it out and use it in your code.

```
import pygame
pygame.init()

BLACK = (0, 0, 0)
WHITE = (255, 255, 255)
BLUE = (0, 0, 255)
GREEN = (0, 255, 0)
RED = (255, 0, 0)

PI = 3.141592653

size = (400, 500)
screen = pygame.display.set_mode(size)

pygame.display.set_caption("Text Examples")

done = False
clock = pygame.time.Clock()

text_rotate_degrees = 0

while not done:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            done = True

    screen.fill(WHITE)

    pygame.draw.line(screen, BLACK, [100,50],
    [200, 50])
    pygame.draw.line(screen, BLACK, [100,50],
    [100, 150])

    font = pygame.font.SysFont('Calibri', 25,
    True, False)
    text = font.render("Sideways text", True, BLACK)
    text = pygame.transform.rotate(text, 90)
    screen.blit(text, [0, 0])

    text = font.render("Upside down text", True, BLACK)
    text = pygame.transform.rotate(text, 180)
    screen.blit(text, [30, 0])

    text = font.render("Flipped text", True, BLACK)
    text = pygame.transform.flip(text, False, True)
    screen.blit(text, [30, 20])

    text = font.render("Rotating text", True, BLACK)
    text = pygame.transform.rotate(text, text_
    rotate_degrees)
    text_rotate_degrees += 1
    screen.blit(text, [100, 50])

    pygame.display.flip()

    clock.tick(60)

pygame.quit()
```

The Joy Of Text

Here, you can see that we've started by defining the colours but left the text black throughout the rest of the code; then, also left the display window with a white background. The code itself is fairly easy to follow and you can improve it by using different colours, changing the font for each text example, and its size too.

Try using some of the text examples individually in your code as an introduction to your program perhaps. Either way you use it, it will help make it stand out a little more than the standard Python code you will likely come across during your time as a programmer.



Google Search Script

Using the OS and Urllib modules, this small snippet of code will display a window for the user's input, then pass the input to a Google search in their browser. It's really quite a handy script and one that's easily introduced into your own code.

GOOGLESEARCH.PY

You will need to pip install the Urllib module, unless it's already installed. It also uses Zenity, which is a Linux-based (GNOME) tool for creating dialog boxes.

```
import os
import urllib.parse
google = os.popen('zenity --entry --text="Enter your
Google search: " --title="Google Search"').read()
google = urllib.parse.quote(google)
os.system('chromium-browser http://www.google.com/
search?q=%s' % (google))
```

Searching For More

Here we've used the Zenity command to create the dialog box, which as we mentioned is only available to Linux machines (such as Ubuntu, Raspberry Pi, Linux Mint and so on). If you want to execute it in Windows you have a couple of possibilities: you can find a Windows version of Zenity and pass the user's query through it; or you can create a Tkinter dialog box to pass the information.

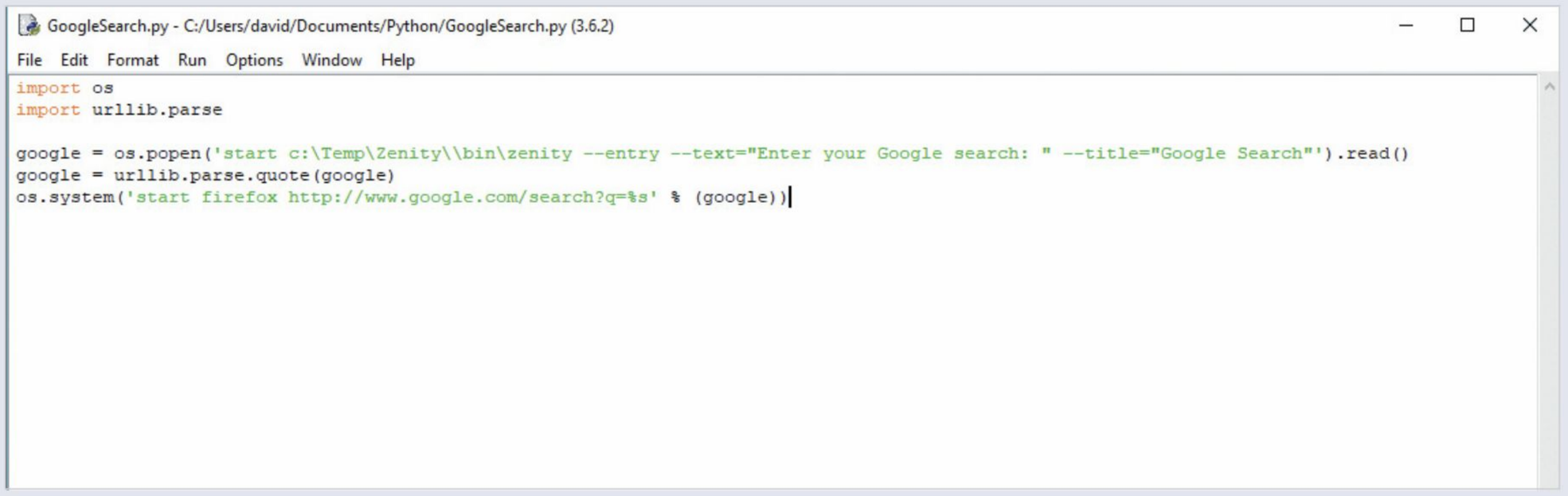
You can see that this particular code uses the Chromium browser which comes preinstalled on the Raspberry Pi, and some versions of Linux. To use your favourite browser in Windows, for example, you will need to change the command in the last line of the code to read Firefox, or whatever you use, together with the Start command. So essentially, one of the following:

```
os.system('start firefox http://www.google.com/
search?q=%s' % (google))

os.system('start chrome http://www.google.com/
search?q=%s' % (google))

os.system('start iexplore http://www.google.com/
search?q=%s' % (google))
```

The last two for Chrome and Internet Explorer respectively.

There is a Zenity for Windows project available on GitHub at www.github.com/kvaps/zenity-windows. It's a good working version but you do need to install it to a folder on your system where you won't require administrator access to be able to run the Zenity program. When you have Zenity installed, you can modify the Windows version of this code in Python to read:

```
import os
import urllib.parse

google = os.popen('start c:\Temp\Zenity\\bin\
zenity --entry --text="Enter your Google search: "
--title="Google Search"').read()
google = urllib.parse.quote(google)
os.system('start firefox http://www.google.com/
search?q=%s' % (google))
```



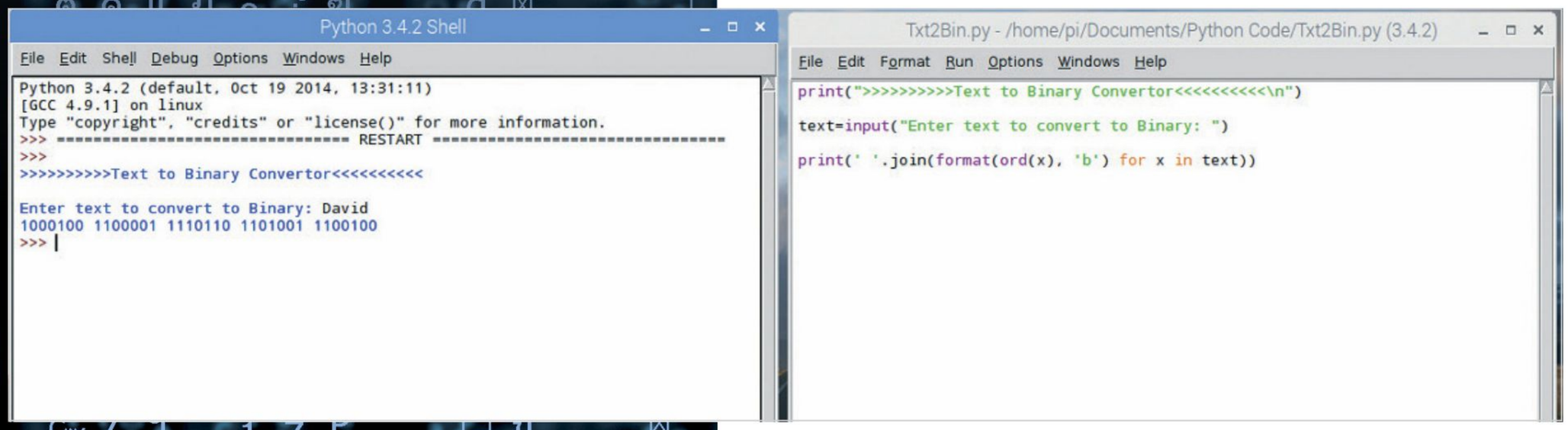
Text to Binary Convertor

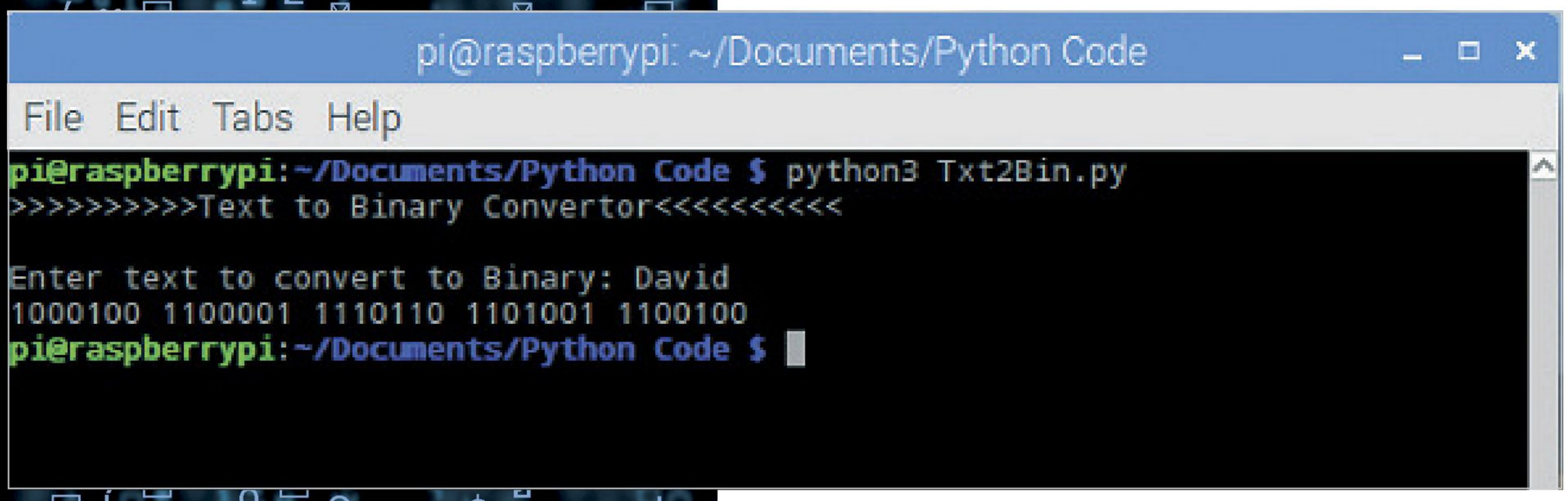
While it may not seem too exciting, this text to binary convertor is actually quite good fun. It also only uses two lines of code, so it's extremely easy to insert into your own script.

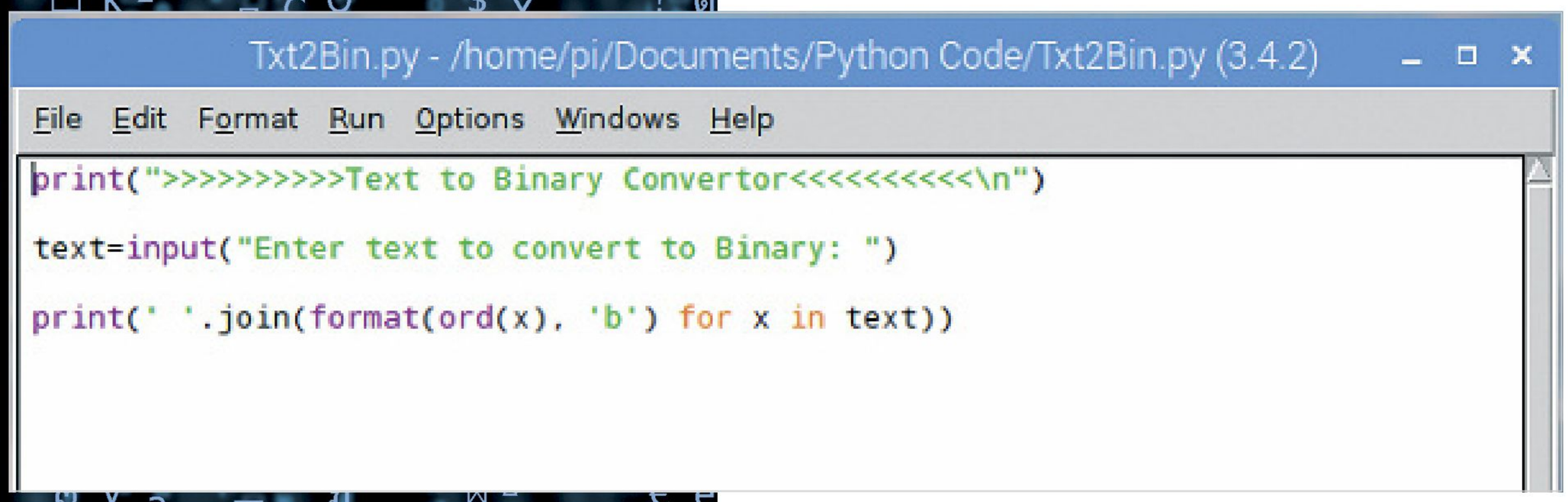
TXT2BIN.PY

Naturally we're using the format function to convert the user's entered text string into its binary equivalent. If you want to check its accuracy, you can plug the binary into an online convertor.

```
text=input("Enter text to convert to Binary: ")

print(' '.join(format(ord(x), 'b') for x in text))
```







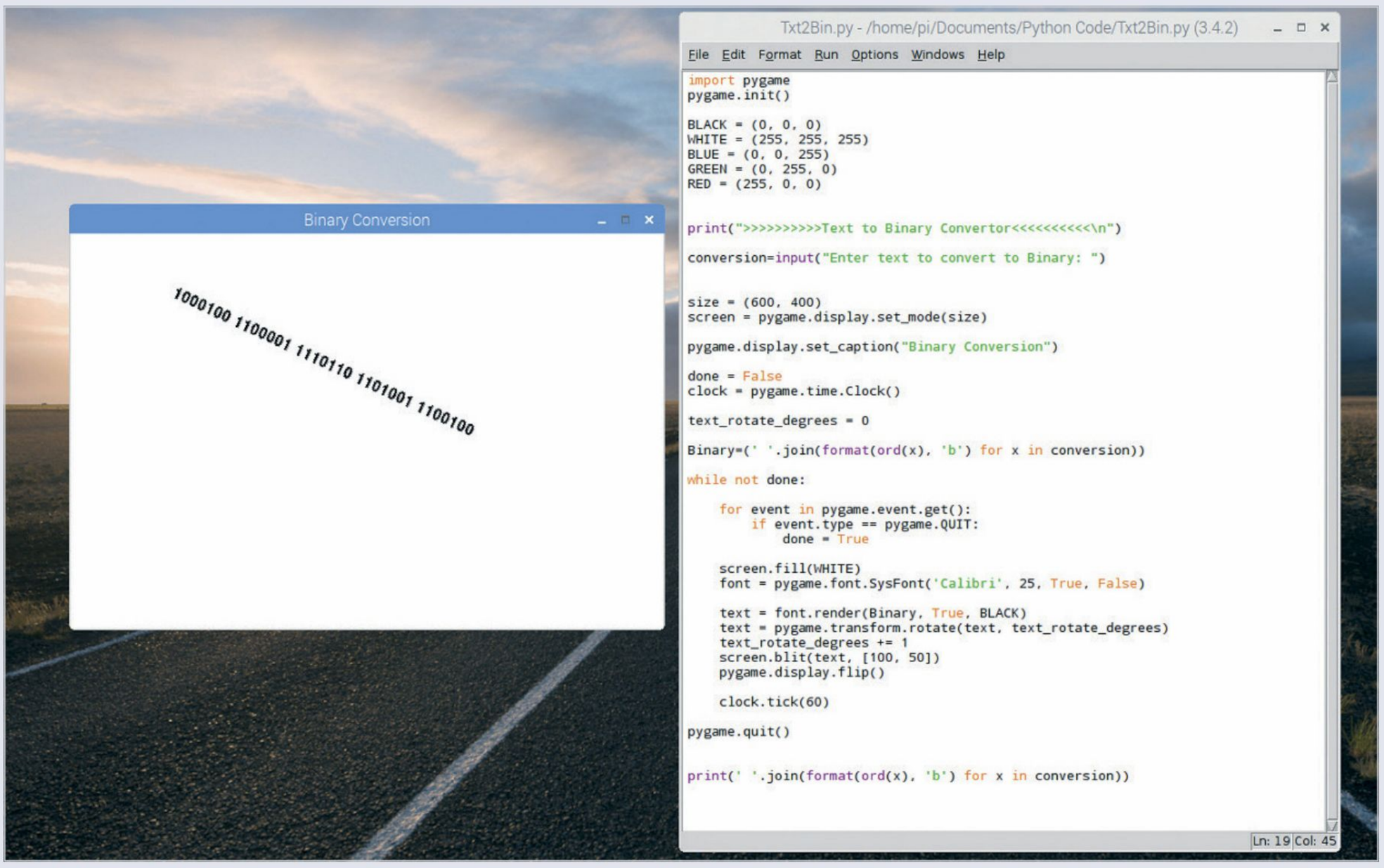
1000010 1101001 1101110 1100001 1110010 1111001

The text to binary convertor does offer some room for improvement and enhancement. There are many uses: it could be utilised in a password or secret word script, as part of an adventure game or just a novel way to display someone's name.

With regards to improvements, you could display the binary conversion in a Pygame window, using the animated text options from page 100. You could also ask the user if they wanted to have another go, or even ask if they wanted the binary output to be saved to a file.

With regards to rendering the outputted binary conversion to a Pygame window, complete with rotating text, you can use:

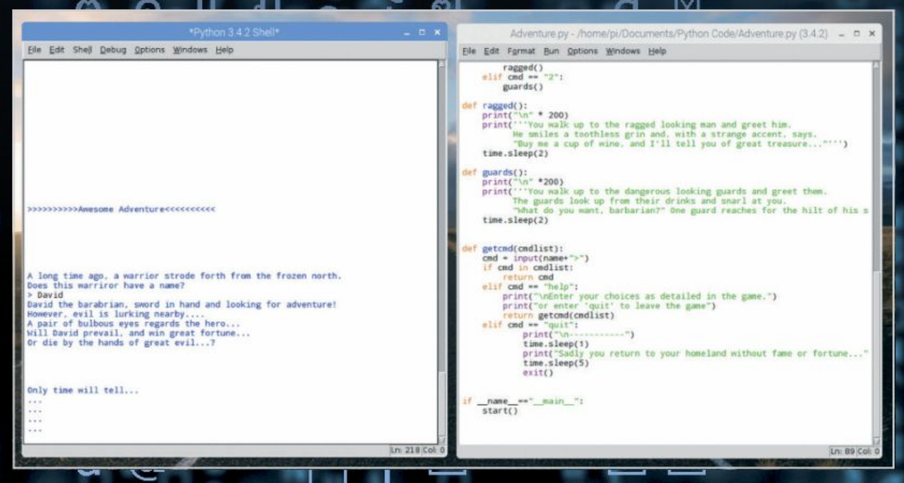
```
import pygame
pygame.init()

BLACK = (0, 0, 0)
WHITE = (255, 255, 255)
BLUE = (0, 0, 255)
GREEN = (0, 255, 0)
RED = (255, 0, 0)

print(">>>>>>>>>>Text to Binary Convertor<<<<<<<<<<\n")

conversion=input("Enter text to convert to Binary: ")

size = (600, 400)
screen = pygame.display.set _ mode(size)
pygame.display.set _ caption("Binary Conversion")

done = False
clock = pygame.time.Clock()

text _ rotate _ degrees = 0

Binary=(' '.join(format(ord(x), 'b') for x
   in conversion))

while not done:

   for event in pygame.event.get():
      if event.type == pygame.QUIT:
         done = True

   screen.fill(WHITE)
   font = pygame.font.SysFont('Calibri', 25, True, False)

   text = font.render(Binary, True, BLACK)
   text = pygame.transform.rotate(text, text _
   rotate _ degrees)
   text _ rotate _ degrees += 1
   screen.blit(text, [100, 50])
   pygame.display.flip()

   clock.tick(60)

pygame.quit()

print(' '.join(format(ord(x), 'b') for x in conversion))
```

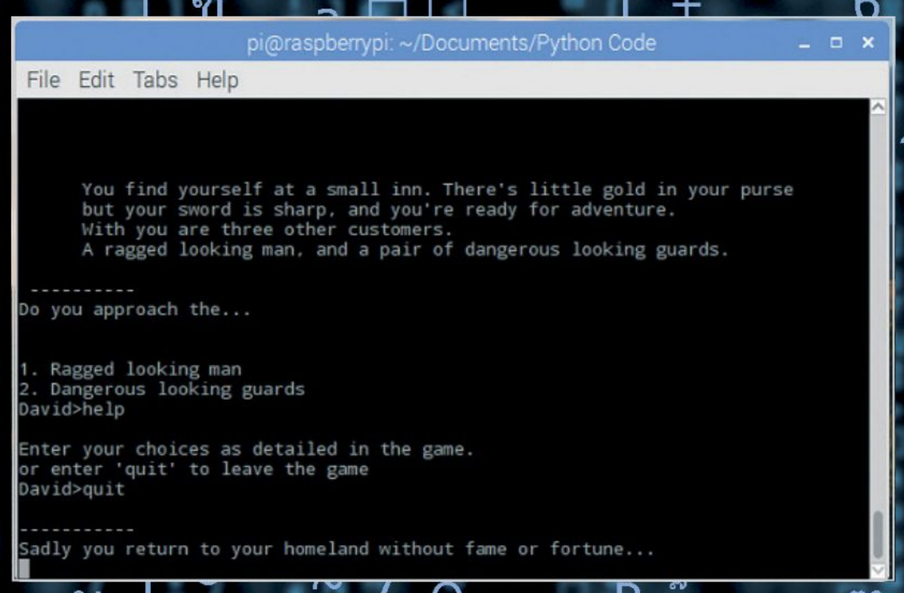


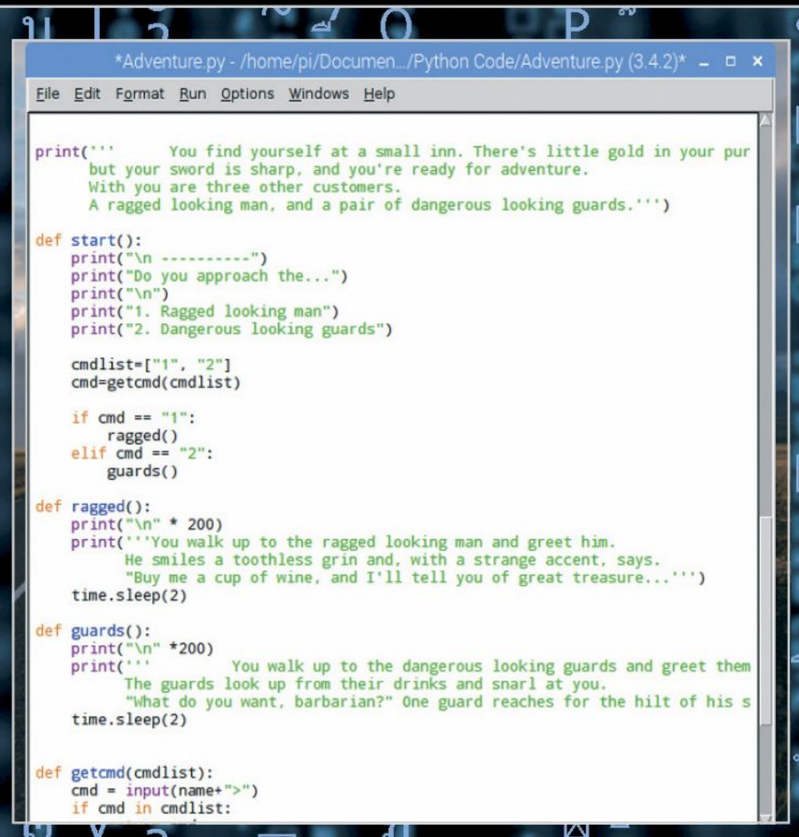
Code Repository

Text Adventure Script

Text adventures are an excellent way to build your Python coding skills and have some fun at the same time. This example that we created will start you on the path to making a classic text adventure; where it will end is up to you.







ADVENTURE.PY

The Adventure game uses just the Time module to begin with, creating pauses between print functions. There's a help system in place to expand upon, as well as the story itself.

```
import time

print("\n" * 200)
print(">>>>>>>>>>Awesome Adventure<<<<<<<<<<\n")
print("\n" * 3)
time.sleep(3)
print("\nA long time ago, a warrior strode forth from
the frozen north.")
time.sleep(1)
print("Does this warrior have a name?")
name=input("> ")
print(name, "the barbarian, sword in hand and looking
for adventure!")
time.sleep(1)
print("However, evil is lurking nearby....")
time.sleep(1)
print("A pair of bulbous eyes regards the hero...")
time.sleep(1)
print("Will", name, "prevail, and win great fortune...")
time.sleep(1)
print("Or die by the hands of great evil...?")
time.sleep(1)
print("\n" *3)
print("Only time will tell...")
time.sleep(1)
print('...')
time.sleep(1)
print('...')
time.sleep(1)
print('...')
time.sleep(1)
print('...')
time.sleep(5)
print("\n" *200)

print('''      You find yourself at a small inn. There's
   little gold in your purse but your sword is sharp,
   and you're ready for adventure.
   With you are three other customers.
   A ragged looking man, and a pair of dangerous
   looking guards.''')

def start():
   print("\n ----------")
   print("Do you approach the...")
   print("\n")
   print("1. Ragged looking man")
   print("2. Dangerous looking guards")

   cmdlist=["1", "2"]
   cmd=getcmd(cmdlist)
```

```
    if cmd == "1":
        ragged()
    elif cmd == "2":
        guards()

def ragged():
    print("\n" * 200)
    print('''You walk up to the ragged looking man and
    greet him.
        He smiles a toothless grin and, with a strange
        accent, says.
        "Buy me a cup of wine, and I'll tell you of
        great treasure...''')
    time.sleep(2)

def guards():
    print("\n" *200)
    print('''You walk up to the dangerous looking guards
    and greet them.
        The guards look up from their drinks and
        snarl at you.
        "What do you want, barbarian?" One guard reaches
        for the hilt of his sword...''')
    time.sleep(2)
```

```
def getcmd(cmdlist):
    cmd = input(name+">")
    if cmd in cmdlist:
        return cmd
    elif cmd == "help":
        print("\nEnter your choices as detailed in
        the game.")
        print("or enter 'quit' to leave the game")
        return getcmd(cmdlist)
    elif cmd == "quit":
        print("\n----------")
        time.sleep(1)
        print("Sadly you return to your homeland without
        fame or fortune...")
        time.sleep(5)
        exit()


if _ _ name _ _ ==" _ _ main _ _ ":
    start()
```

Adventure Time

This, as you can see, is just the beginning of the adventure and takes up a fair few lines of code. When you expand it, and weave the story along, you'll find that you can repeat certain instances such as a chance meeting with an enemy or the like.

We've created each of the two encounters as a defined set of functions, along with a list of possible choices under the cmdlist list, and cmd variable, of which is also a defined function. Expanding on this is quite easy, just map out each encounter and choice and create a defined function around it. Providing the user doesn't enter quit into the adventure, they can keep playing.

There's also room in the adventure for a set of variables designed for combat, luck, health, endurance and even an inventory or amount of gold earned. Each successful combat situation can reduce the main character's health but increase their combat skills or endurance. Plus, they could loot the body and gain gold, or earn gold through quests.

Finally, how about introducing the Random module. This will enable you to include an element of chance in the game. For example, in combat, when you strike an enemy you will do a random amount of damage as will they. You could even work out the maths behind improving the chance of a better hit based on your or your opponent's combat skills, current health, strength and endurance. You could create a game of dice in the inn, to see if you win or lose gold (again, improve the chances of winning by working out your luck factor into the equation).

Needless to say, your text adventure can grow exponentially and prove to be a work of wonder. Good luck, and have fun with your adventure.

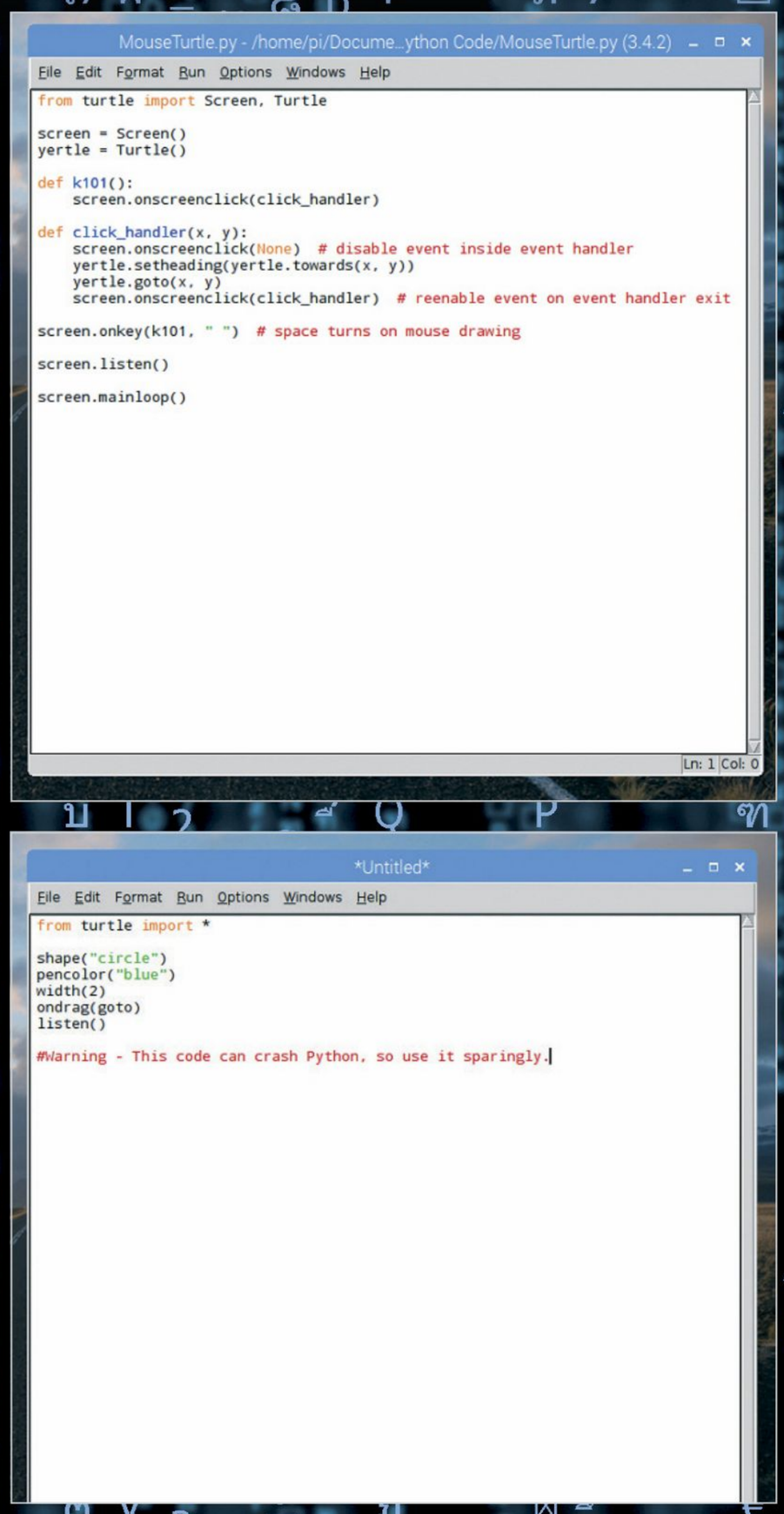
Adventure.py - /home/pi/Documents/Python Code/Adventure.py (3.4.2)

File Edit Format Run Options Windows Help

```
print("\n" *200)

CR=0
Strength=0
Health=0
Luck=0

print("The mountains of the north make for a hard life.")
print("Press Enter to roll the dice and see how strong", name, "is:")
input()
Strength=random.randint(1,20)
print(name, "has a Strength value of:", Strength)
print("\nIt's a hard life indeed, and all northeners are born warriors.")
print("Press Enter to roll the dice and see the Combat Rating for", name+".")
input()
CR=random.randint(1, 30)
print(name, "has a Combat Rating of:", CR)
print("\nYour Health is the total of your Strength and Combat Rating.")
print("Press Enter to see", name+"'s", "Health value.")
input()
Health=Strength+CR
print(name, "has a Health value of:", Health)
print("\nEveryone needs a certain amount of luck to survive.")
print("Press Enter to roll the dice and see how lucky", name, "is.")
input()
Luck=random.randint(1, 15)
if Luck > 13:
    print(name, "is luck indeed, and has a Luck value of:", Luck)
else:
    print(name, "has a Luck value of:", Luck)
time.sleep(5)
print("\n" *200)
print("Here's your character stats:\n")
print(name)
print("\nCombat Rating =", CR)
print("Strength =", Strength)
print("Health =", Health)
print("Luck =", Luck)
print("\n" *5)
print("Press Enter to start your adventure...")
input()
print("\n" *200)


print('''      You find yourself at a small inn. There's little gold in your purse
      but your sword is sharp, and you're ready for adventure.
      With you are three other customers.
      A ragged looking man, and a pair of dangerous looking guards.''')

def start():
    print("\n ----------")
    print("Do you approach the...")
    print("\n")
    print("1. Ragged looking man")
    print("2. Dangerous looking guards")
```

Mouse Controlled Turtle

We've already seen the Turtle module being controlled by the user via the keyboard but now we thought we'd see how the user can use their mouse as a drawing tool within Python. We have two possible code examples here, pick which works best for you.



MOUSETURTLE.PY

The first piece of code presents the standard Turtle window. Press Space and then click anywhere on the screen for the Turtle to draw to the mouse pointer. The second allows you to click the Turtle and drag it around the screen; but be warned, it can crash Python.

1st Code Example:

```
from turtle import Screen, Turtle

screen = Screen()
yertle = Turtle()

def k101():
   screen.onscreenclick(click _ handler)

def click _ handler(x, y):
   screen.onscreenclick(None)  # disable event inside
   event handler
   yertle.setheading(yertle.towards(x, y))
   yertle.goto(x, y)
   screen.onscreenclick(click _ handler)  # reenable
   event on event handler exit

screen.onkey(k101, " ")  # space turns on mouse drawing

screen.listen()

screen.mainloop()
```

2nd Code Example:

```
from turtle import *
shape("circle")
pencolor("blue")
width(2)
ondrag(goto)
listen()
```

Ninja TurtleMouse

This code utilises some interesting skills. Obviously it will stretch your Python Turtle skills to come up with any improvements, which is great, but it could make for a nice piece of code to insert into something a young child will use. Therefore it can be a fantastic project for a younger person to get their teeth into; or perhaps even as part of a game where the main character is tasked to draw a skull and crossbones or something similar.

Python Alarm Clock

Ever taken a quick break from working at the computer, then suddenly realised many minutes later that you've spent all that time on Facebook? Introducing the Python alarm clock code, where you can drop into the command prompt and tell the code how many minutes until the alarm goes off.

ALARMCLOCK.PY

This code is designed for use in the command prompt, be that Windows, Linux or macOS. There are some instructions on how to use it in the main print section but essentially it's: python3 AlarmClock.py 10 (to go off in ten minutes).

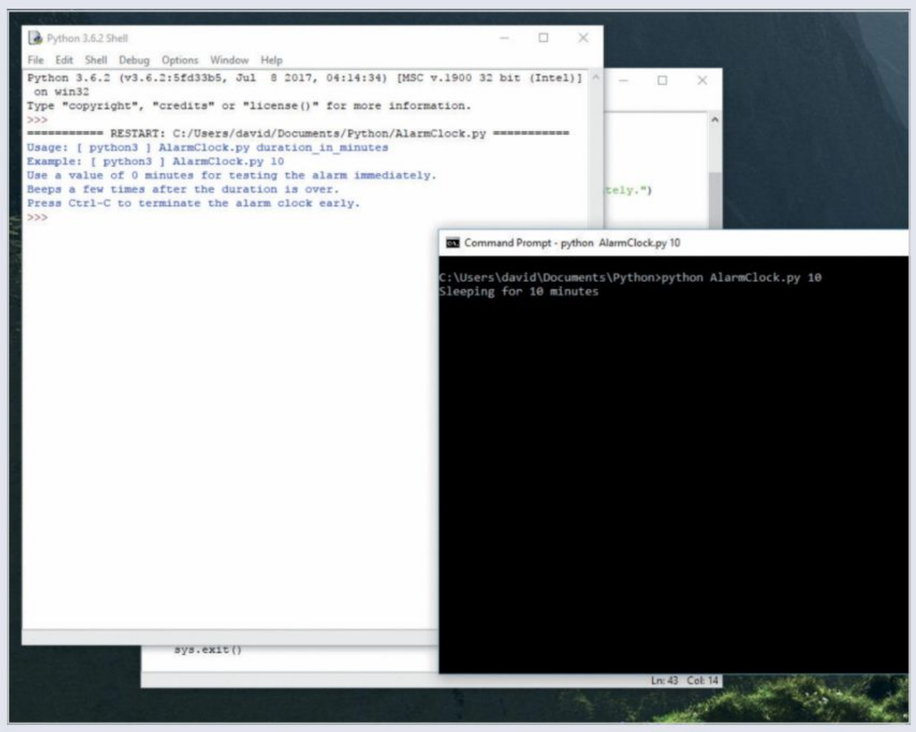
```
import sys
import string
from time import sleep

sa = sys.argv
lsa = len(sys.argv)
if lsa != 2:
   print ("Usage: [ python3 ] AlarmClock.py duration _
   in _ minutes")
   print ("Example: [ python3 ] AlarmClock.py 10")
   print ("Use a value of 0 minutes for testing the
   alarm immediately.")
   print ("Beeps a few times after the duration is over.")
   print ("Press Ctrl-C to terminate the alarm
   clock early.")
   sys.exit(1)

try:
   minutes = int(sa[1])
except ValueError:
   print ("Invalid numeric value (%s) for minutes" % sa[1])
   print ("Should be an integer >= 0")
   sys.exit(1)

if minutes < 0:
   print ("Invalid value for minutes, should be >= 0")
   sys.exit(1)

seconds = minutes * 60

if minutes == 1:
   unit _ word = " minute"
else:
   unit _ word = " minutes"
```

```
try:
   if minutes > 0:
      print ("Sleeping for " + str(minutes) + unit _ word)
      sleep(seconds)
   print ("Wake up")
   for i in range(5):
      print (chr(7)),
      sleep(1)
except KeyboardInterrupt:
   print ("Interrupted by user")
   sys.exit(1)
```

Wakey Wakey

There's some good use of try and except blocks here, alongside some other useful loops that can help you get a firmer understanding of how they work in Python. The code itself can be used in a variety of ways: in a game where something happens after a set amount of time or simply as a handy desktop alarm clock for your tea break.

Linux users, try making the alarm clock code into an alias, so you can run a simple command to execute it. Then, why not integrate a user input at the beginning to ask the user for the length of time they want until the alarm goes off, rather than having to include it in the command line.



Windows users, if Python 3 is the only version installed on your system then you will need to execute the code without adding the 3 to the end of the Python command. For example:

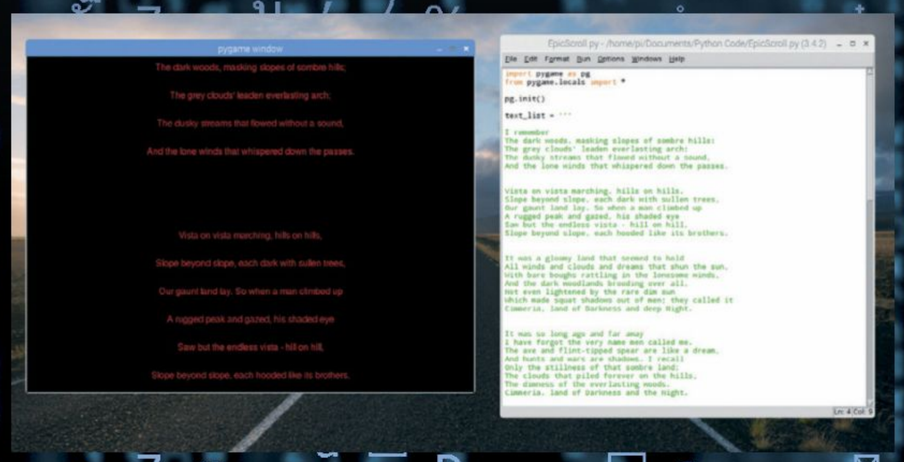
```
python AlarmClock.py 10
```

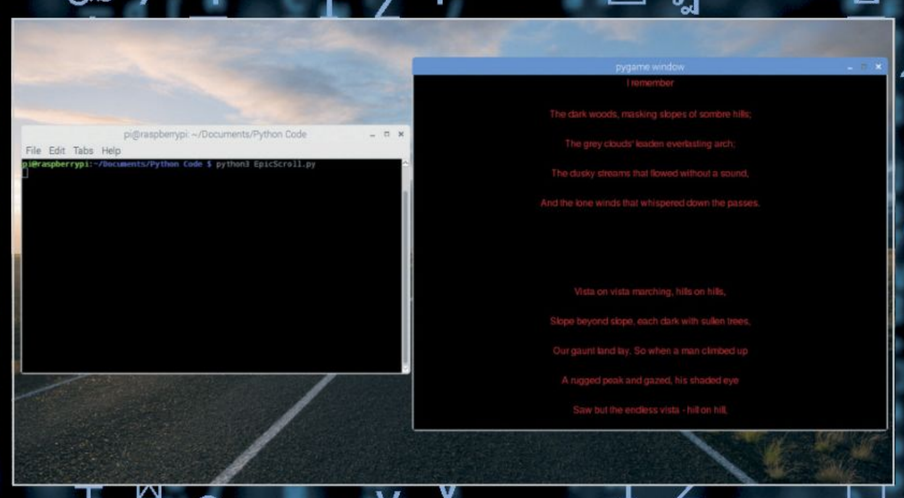
Again, you could easily incorporate this into a Windows batch file and even set a schedule to activate the alarm at certain times of the day.

Code Repository

Vertically Scrolling Text

What's not to like about vertically scrolling text? Its uses are many: the beginning of a game or introduction to something epic, like the beginning of every Star Wars movie; a list of credits at the end of something, such as a Python presentation. The list goes on.





EPICSCROLL.PY

We've used the poem Cimmeria by Robert E. Howard for the code's scrolling text, along with a dramatic black background and red text. We think you'll agree, it's quite epic.

```
import pygame as pg
from pygame.locals import *

pg.init()

text_list = '''

I remember
The dark woods, masking slopes of sombre hills;
The grey clouds' leaden everlasting arch;
The dusky streams that flowed without a sound,
And the lone winds that whispered down the passes.


Vista on vista marching, hills on hills,
Slope beyond slope, each dark with sullen trees,
Our gaunt land lay. So when a man climbed up
A rugged peak and gazed, his shaded eye
Saw but the endless vista - hill on hill,
Slope beyond slope, each hooded like its brothers.


It was a gloomy land that seemed to hold
All winds and clouds and dreams that shun the sun,
With bare boughs rattling in the lonesome winds,
And the dark woodlands brooding over all,
Not even lightened by the rare dim sun
Which made squat shadows out of men; they called it
Cimmeria, land of Darkness and deep Night.


It was so long ago and far away
I have forgot the very name men called me.
The axe and flint-tipped spear are like a dream,
And hunts and wars are shadows. I recall
Only the stillness of that sombre land;
The clouds that piled forever on the hills,
The dimness of the everlasting woods.
Cimmeria, land of Darkness and the Night.


Oh, soul of mine, born out of shadowed hills,
To clouds and winds and ghosts that shun the sun,
How many deaths shall serve to break at last
This heritage which wraps me in the grey
Apparel of ghosts? I search my heart and find
Cimmeria, land of Darkness and the Night!

'''.split('\n')
```

```
class Credits:
   def __init__(self, screen_rect, lst):
      self.srect = screen_rect
      self.lst = lst
      self.size = 16
      self.color = (255,0,0)
      self.buff_centery = self.srect.height/2 + 5
      self.buff_lines = 50
      self.timer = 0.0
      self.delay = 0
      self.make_surfaces()


   def make_text(self,message):
      font = pg.font.SysFont('Arial', self.size)
      text = font.render(message,True,self.color)
      rect = text.get_rect(center = (self.srect.
      centerx, self.srect.centery + self.buff_centery) )
      return text,rect

   def make_surfaces(self):
      self.text = []
      for i, line in enumerate(self.lst):
         l = self.make_text(line)
         l[1].y += i*self.buff_lines
         self.text.append(l)

   def update(self):
      if pg.time.get_ticks()-self.timer > self.delay:
         self.timer = pg.time.get_ticks()
         for text, rect in self.text:
            rect.y -= 1

   def render(self, surf):
      for text, rect in self.text:
         surf.blit(text, rect)

screen = pg.display.set_mode((800,600))
screen_rect = screen.get_rect()
clock = pg.time.Clock()
running=True
cred = Credits(screen_rect, text_list)


while running:
   for event in pg.event.get():
      if event.type == QUIT:
         running = False
   screen.fill((0,0,0))
   cred.update()
   cred.render(screen)
   pg.display.update()
   clock.tick(60)
```

A Long Time Ago...

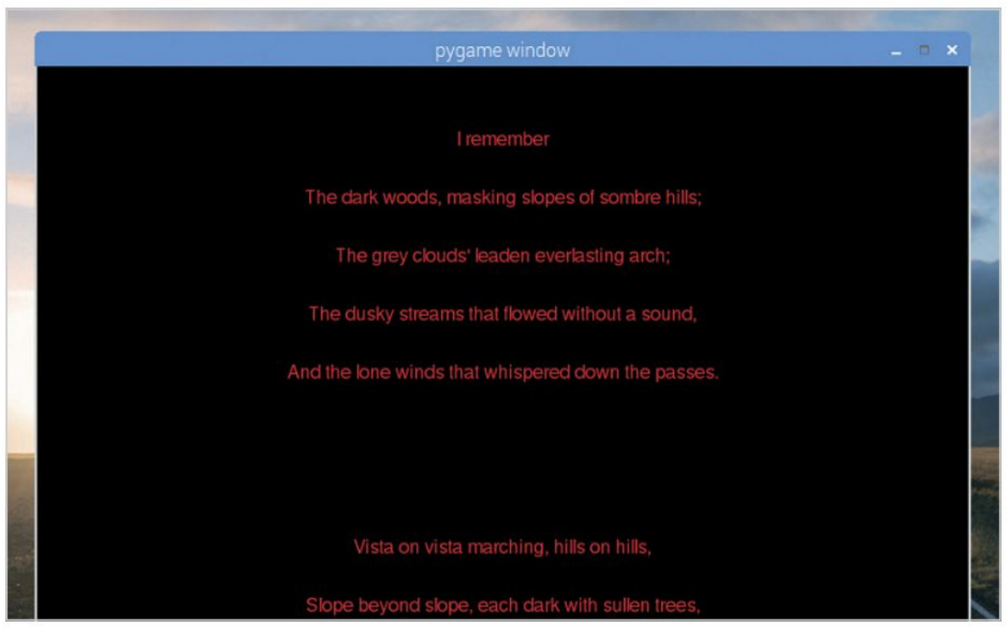
The obvious main point of enhancement is the actual text itself. Replace it with a list of credits, or an equally epic opening storyline to your Python game, and it will certainly hit the mark with whoever plays it. Don't forget to change the screen resolution if needed; we're currently running it at 800 x 600.

EpicScroll.py - /home/pi/Documents/Python Code/EpicScroll.py (3.4.2)

File Edit Format Run Options Windows Help

```
'''.split('\n')

class Credits:
    def __init__(self, screen_rect, lst):
        self.srect = screen_rect
        self.lst = lst
        self.size = 16
        self.color = (255,0,0)
        self.buff_centery = self.srect.height/2 + 5
        self.buff_lines = 50
        self.timer = 0.0
        self.delay = 0
        self.make_surfaces()


    def make_text(self,message):
        font = pg.font.SysFont('Arial', self.size)
        text = font.render(message,True,self.color)
        rect = text.get_rect(center = (self.srect.centerx, self.srect.centery +
        return text,rect

    def make_surfaces(self):
        self.text = []
        for i, line in enumerate(self.lst):
            l = self.make_text(line)
            l[1].y += i*self.buff_lines
            self.text.append(l)

    def update(self):
        if pg.time.get_ticks()-self.timer > self.delay:
            self.timer = pg.time.get_ticks()
            for text, rect in self.text:
                rect.y -= 1

    def render(self, surf):
        for text, rect in self.text:
            surf.blit(text, rect)

screen = pg.display.set_mode((800,600))
screen_rect = screen.get_rect()
clock = pg.time.Clock()
running=True
cred = Credits(screen_rect, text_list)


while running:
    for event in pg.event.get():
        if event.type == QUIT:
            running = False
    screen.fill((0,0,0))
    cred.update()
    cred.render(screen)
    pg.display.update()
    clock.tick(60)
```

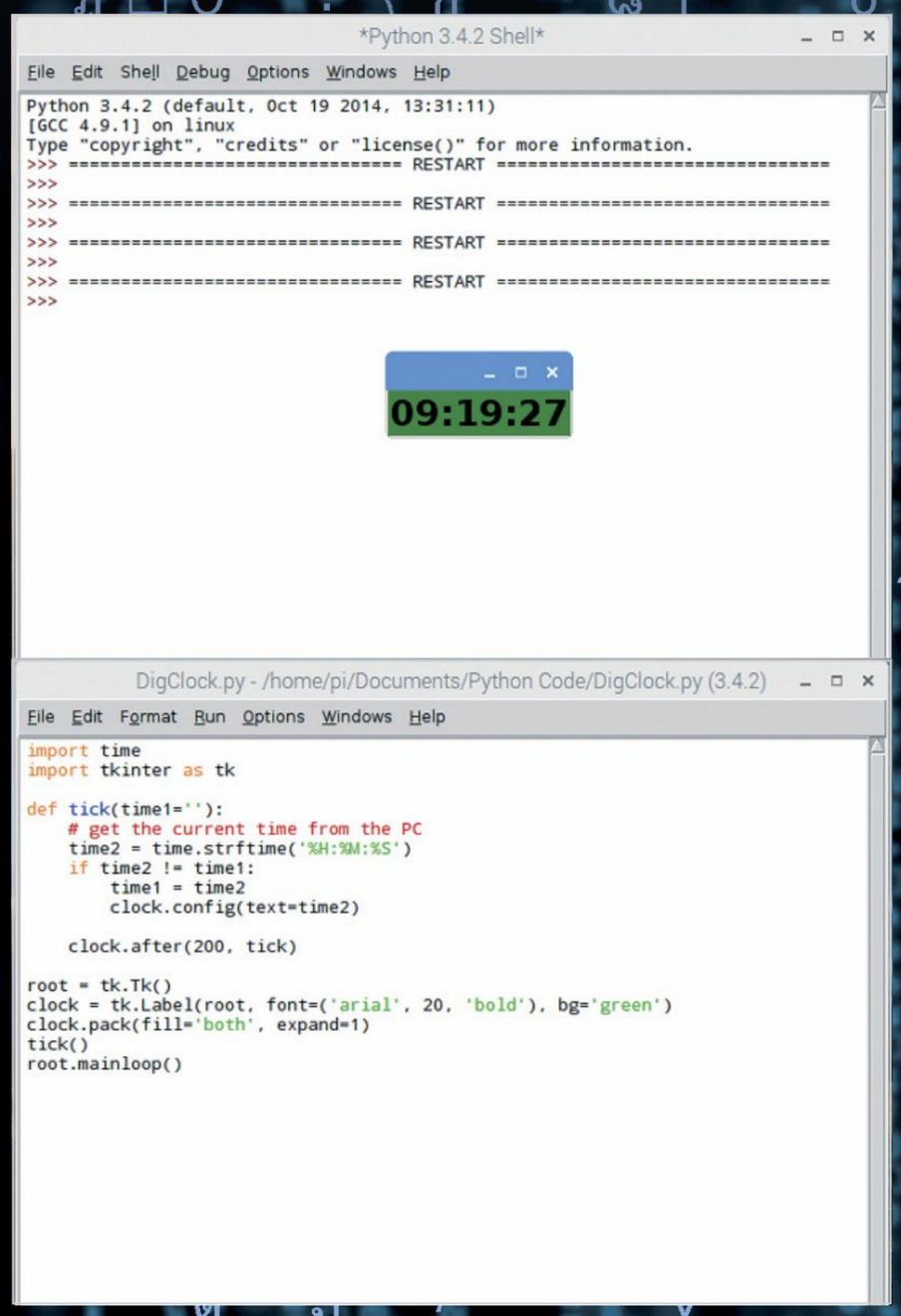
Ln: 1 Col: 0



Code Repository

Python Digital Clock

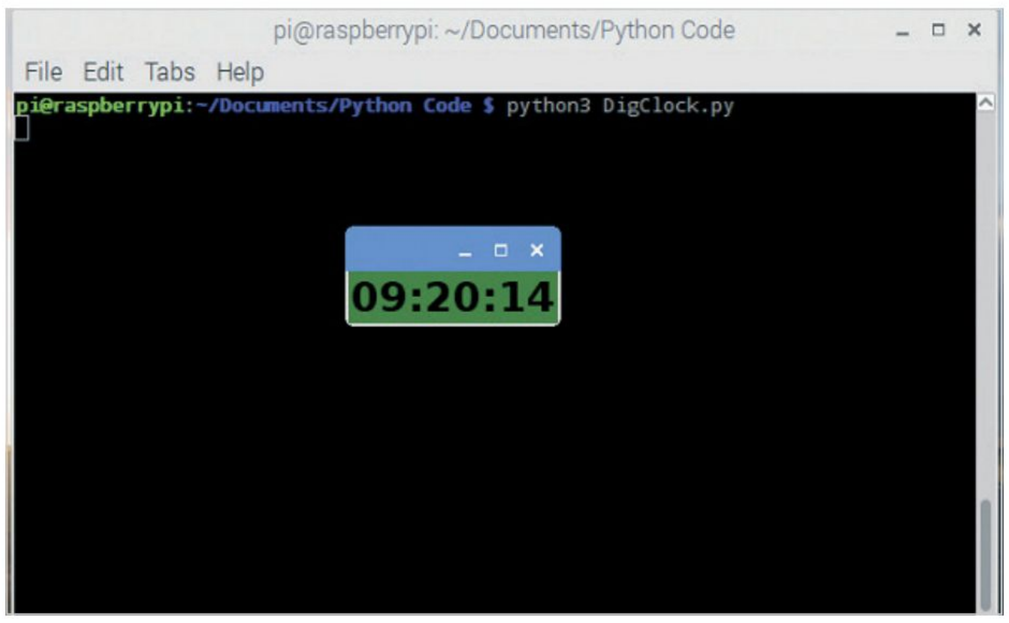
There is already a clock displayed on the desktop of most operating systems but it's always handy to have one on top of the currently open window. To that end, why not create a Python digital clock that can be a companion desktop widget for you.

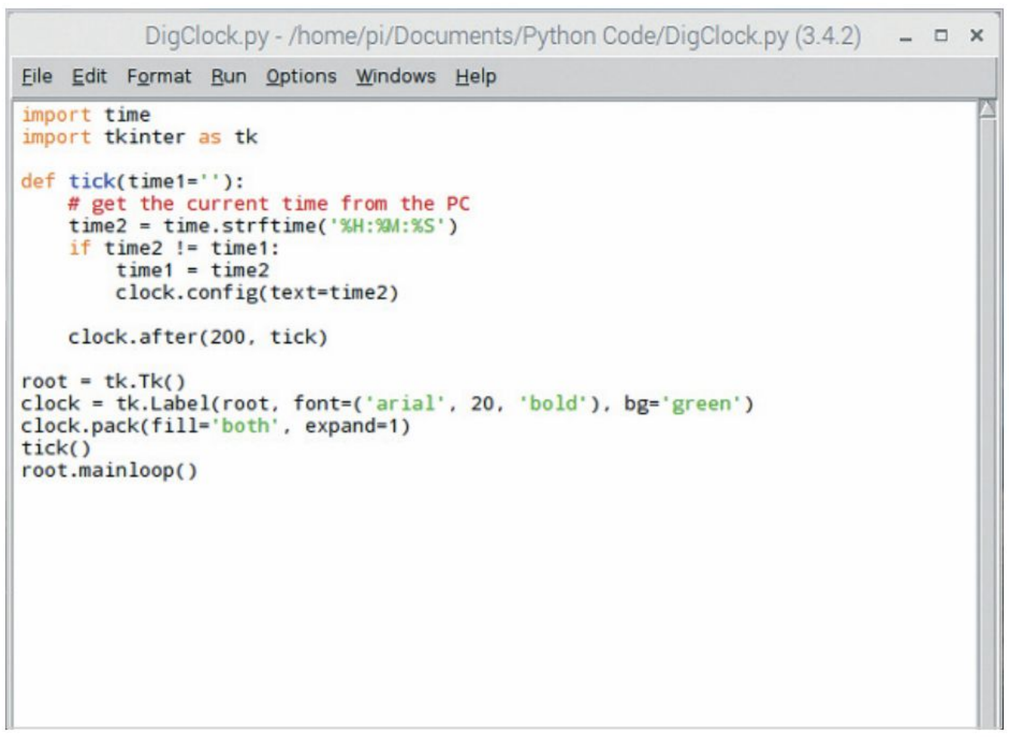


DIGCLOCK.PY

This is a surprisingly handy little script and one that we've used in the past instead of relying on a watch or even the clock in the system tray of the operating system.

```
import time
import tkinter as tk

def tick(time1=''):
   # get the current time from the PC
   time2 = time.strftime('%H:%M:%S')
   if time2 != time1:
      time1 = time2
      clock.config(text=time2)

   clock.after(200, tick)

root = tk.Tk()
clock = tk.Label(root, font=('arial', 20, 'bold'),
bg='green')
clock.pack(fill='both', expand=1)
tick()
root.mainloop()
```



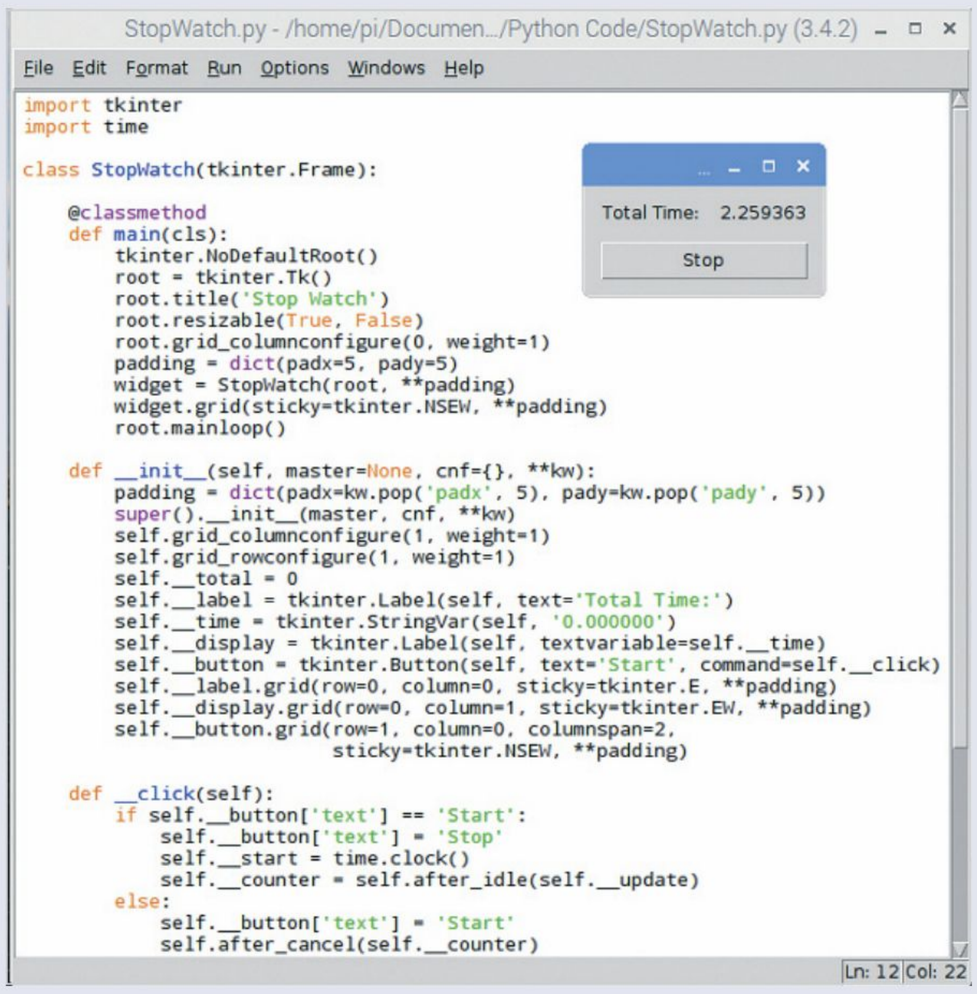


Tick Tock

This is a piece of code we've used many times in the past to keep track of time while working on multiple monitors and with just a quick glance to where we've placed it on the screen.

The Tkinter box can be moved around without affecting the time, maximised or closed by the user at will. We haven't given the Tkinter clock window a title, so you can add to that easily enough by snipping the code from other examples in this book.

Another area of improvement is to include this code when Windows or Linux starts, so it automatically pops up on the desktop. See also, if you're able to improve its functionality by including different time zones: Rome, Paris, London, New York, Moscow and so on.



Another example, expanding on the original code, could be a digital stopwatch. For that you could use the following:

```
import tkinter
import time

class StopWatch(tkinter.Frame):

    @classmethod
    def main(cls):
        tkinter.NoDefaultRoot()
        root = tkinter.Tk()
        root.title('Stop Watch')
        root.resizable(True, False)
        root.grid _ columnconfigure(0, weight=1)
        padding = dict(padx=5, pady=5)
        widget = StopWatch(root, **padding)
        widget.grid(sticky=tkinter.NSEW, **padding)
        root.mainloop()

    def _ _ init _ _ (self, master=None, cnf={}, **kw):
        padding = dict(padx=kw.pop('padx', 5), pady=kw.
        pop('pady', 5))
        super(). _ _ init _ _ (master, cnf, **kw)
        self.grid _ columnconfigure(1, weight=1)
        self.grid _ rowconfigure(1, weight=1)
        self. _ _ total = 0
        self. _ _ label = tkinter.Label(self,
        text='Total Time:')
        self. _ _ time = tkinter.StringVar(self,
            '0.000000')
        self. _ _ display = tkinter.Label(self,
        textvariable=self. _ _ time)
        self. _ _ button = tkinter.Button(self,
        text='Start', command=self. _ _ click)
        self. _ _ label.grid(row=0, column=0,
        sticky=tkinter.E, **padding)
        self. _ _ display.grid(row=0, column=1,
        sticky=tkinter.EW, **padding)
        self. _ _ button.grid(row=1, column=0,
        columnspan=2,sticky=tkinter.NSEW, **padding)

    def _ _ click(self):
        if self. _ _ button['text'] == 'Start':
            self. _ _ button['text'] = 'Stop'
            self. _ _ start = time.clock()
            self. _ _ counter = self.after _ idle(self. _ _ update)
        else:
            self. _ _ button['text'] = 'Start'
            self.after _ cancel(self. _ _ counter)

    def _ _ update(self):
        now = time.clock()
        diff = now - self. _ _ start
        self. _ _ start = now
        self. _ _ total += diff
        self. _ _ time.set('{:.6f}'.format(self. _ _ total))
        self. _ _ counter = self.after _ idle(self. _ _ update)

if _ _ name _ _ == ' _ _ main _ _ ':
    StopWatch.main()
```

Code Repository

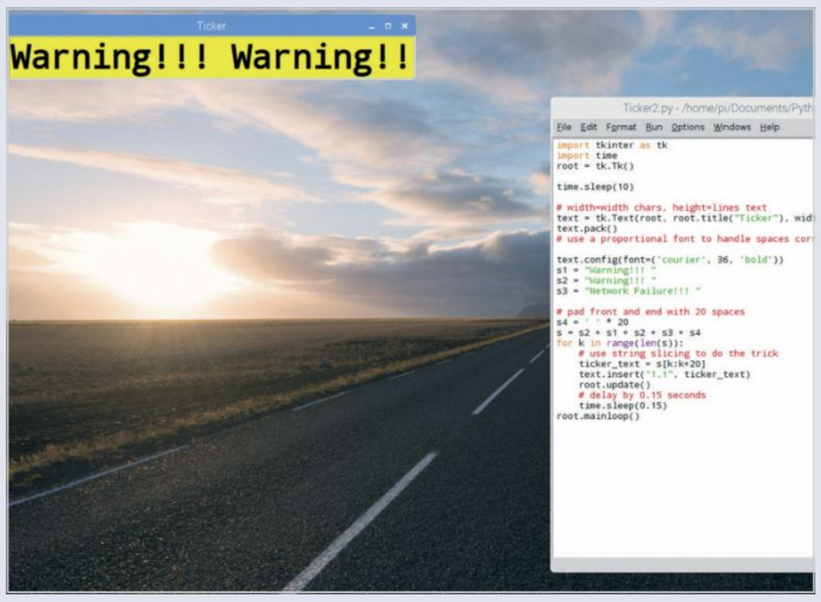
Python Scrolling Ticker Script

You may be surprised to hear that one of the snippets of code we're often asked for is some form of scrolling ticker. Whilst we've covered various forms of scrolling text previously, the ticker is something that seems to keep cropping up. So, here it is.

Ticker Time

The obvious improvements to the Ticker code lie in the speed of the text and what the text will display. Otherwise you can change the background colour of the ticker window, the font and the font colour, along with the geometry of the Tkinter window if you want to.

Yet another interesting element that could be introduced is one of the many text to Speech modules available for Python 3. You could pip install one, import it, then as the ticker displays the text, the text to speech function will read out the variable at the same time, since the entire text is stored in the variable labelled 's'.



The ticker example can be used for system warnings, perhaps something that will display across your work or home network detailing the shutting down of a server over the weekend for maintenance; or even just to inform everyone as to what's happening. We're sure you will come up with some good uses for it.

TICKER.PY

We're using Tkinter here along with the Time module to determine the speed the text is displayed across the window.

```
import time
import tkinter as tk

root = tk.Tk()
canvas = tk.Canvas(root, root.title("Ticker Code"),
height=80, width=600, bg="yellow")
canvas.pack()
font = ('courier', 48, 'bold')
text _ width = 15

#Text blocks insert here....

s1 = "This is a scrolling ticker example. As you
can see, it's quite long but can be a lot longer if
necessary... "
s2 = "We can even extend the length of the ticker
message by including more variables... "
s3 = "The variables are within the s-values in
the code. "
s4 = "Don't forget to concatenate them all before the
For loop, and rename the 'spacer' s-variable too."

# pad front and end of text with spaces
s5 = ' ' * text _ width
# concatenate it all
s = s5 + s1 + s2 + s3 + s4 + s5
x = 1
y = 2
text = canvas.create _ text(x, y, anchor='nw', text=s,
font=font)
dx = 1
dy = 0  # use horizontal movement only

# the pixel value depends on dx, font and length of text
pixels = 9000

for p in range(pixels):
   # move text object by increments dx, dy
   # -dx --> right to left
   canvas.move(text, -dx, dy)
   canvas.update()
   # shorter delay --> faster movement
   time.sleep(0.005)
   #print(k)  # test, helps with pixel value

root.mainloop()
```

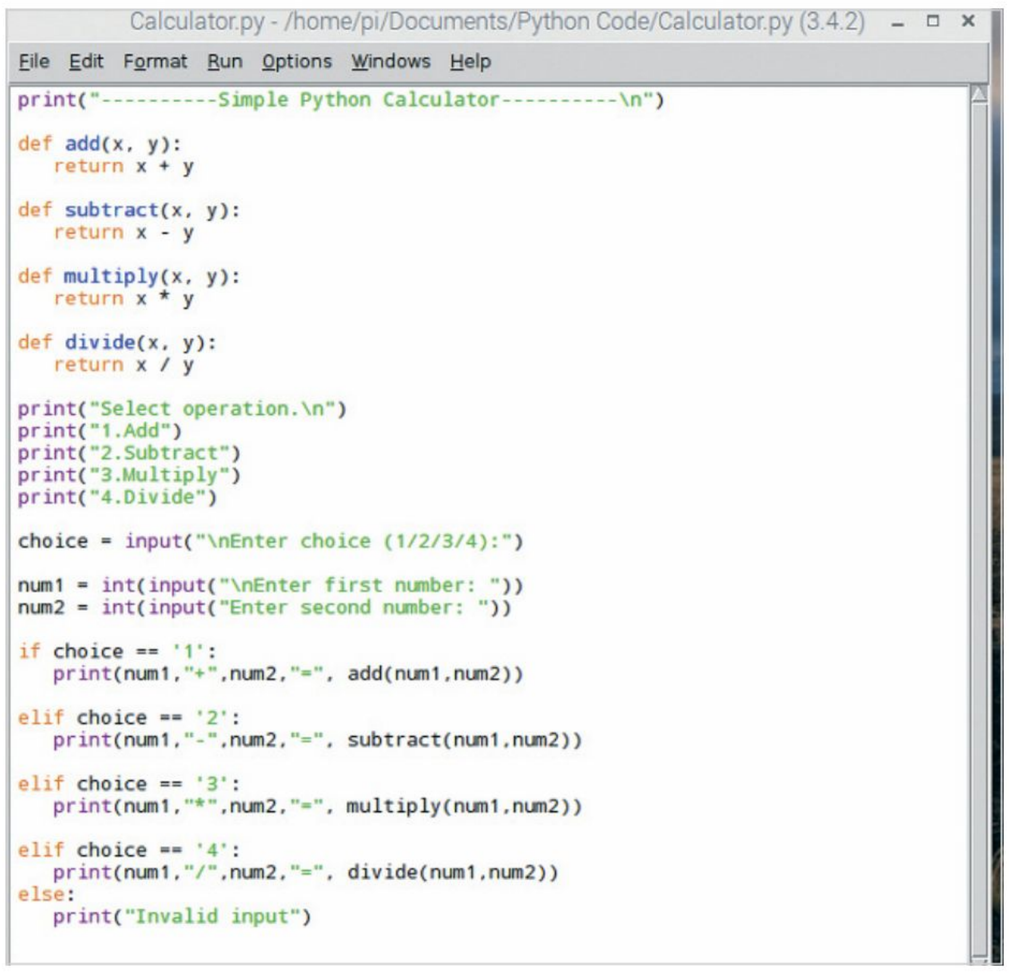
Simple Python Calculator

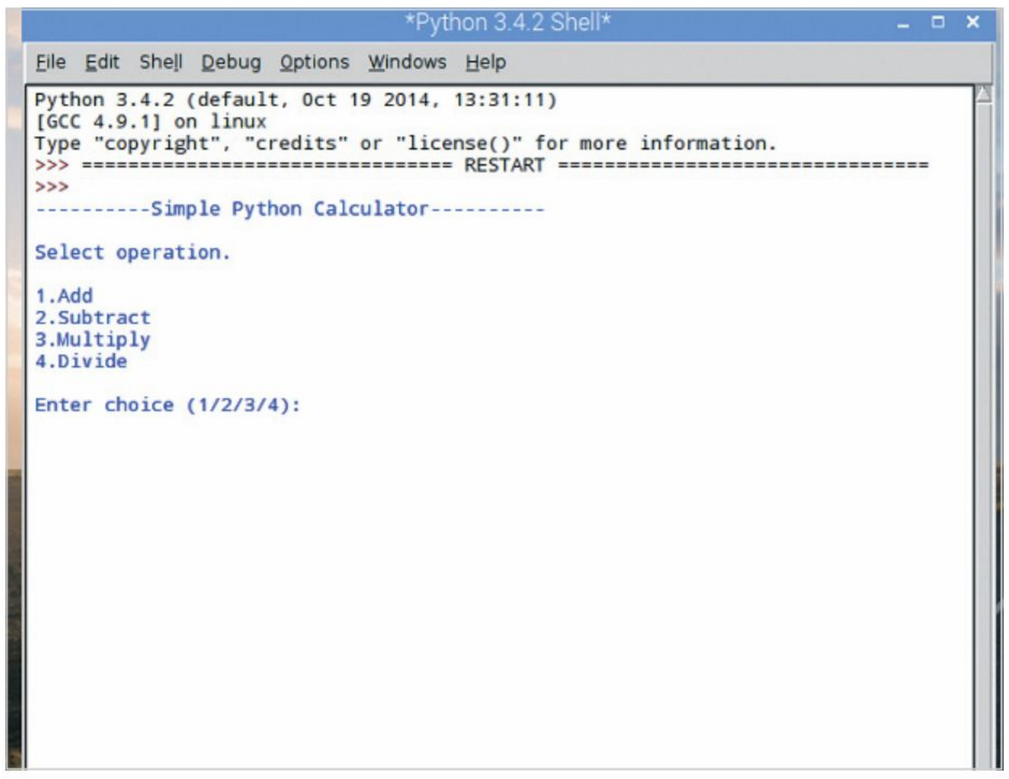
Sometimes the simplest code can be the most effective. Take for example, this Simple Python Calculator script. It's based on the Create Your Own Modules section seen earlier but doesn't utilise any external modules.

CALCULATOR.PY

We created some function definitions to begin with, then lead on to the user menu and inputs. It's an easy piece of code to follow and as such can also be expanded well too.

```
print("----------Simple Python Calculator----------\n")

def add(x, y):
   return x + y

def subtract(x, y):
   return x - y

def multiply(x, y):
   return x * y

def divide(x, y):
   return x / y

print("Select operation.\n")
print("1.Add")
print("2.Subtract")
print("3.Multiply")
print("4.Divide")

choice = input("\nEnter choice (1/2/3/4):")

num1 = int(input("\nEnter first number: "))
num2 = int(input("Enter second number: "))

if choice == '1':
   print(num1,"+",num2,"=", add(num1,num2))

elif choice == '2':
   print(num1,"-",num2,"=", subtract(num1,num2))

elif choice == '3':
   print(num1,"*",num2,"=", multiply(num1,num2))

elif choice == '4':
   print(num1,"/",num2,"=", divide(num1,num2))
else:
   print("Invalid input")
```





Improved Calculations

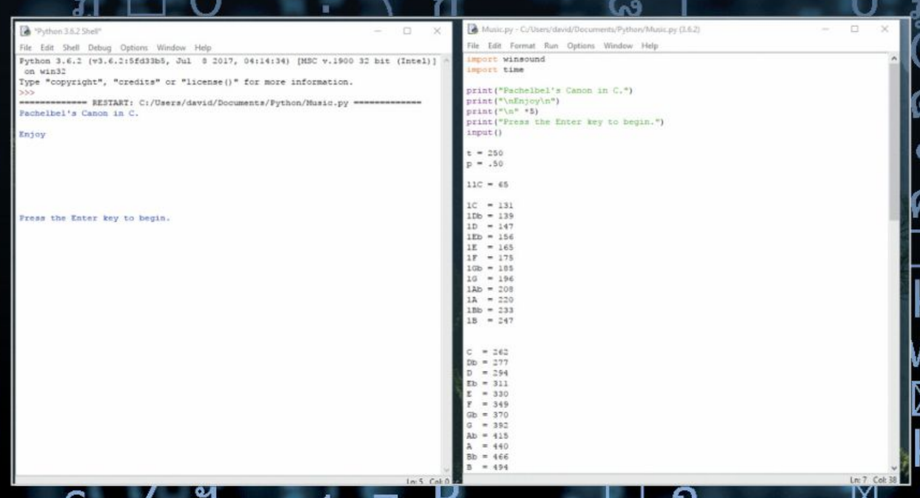
The obvious contender for improvement here is using the Create Your Own Modules route and extracting the function definitions as a module. You can then call the module and focus on the body of the code.

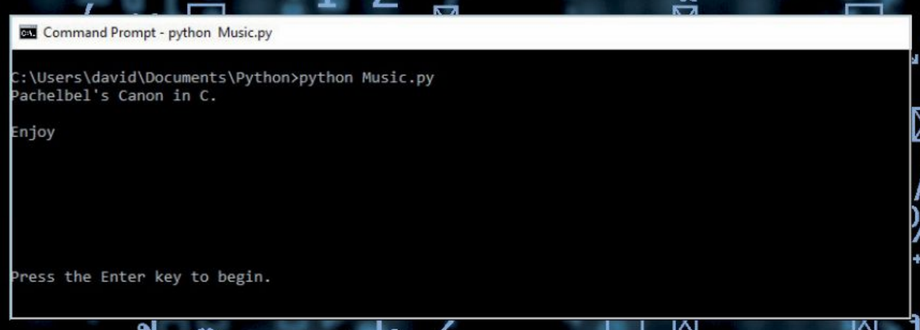
The other area of improvement is code itself. Where there's just a single shot at making a calculation, you could encase it in a while loop, so once a value is presented the user is sent back to the main menu. Perhaps, improvement to the Invalid Input section is worth looking into as well.

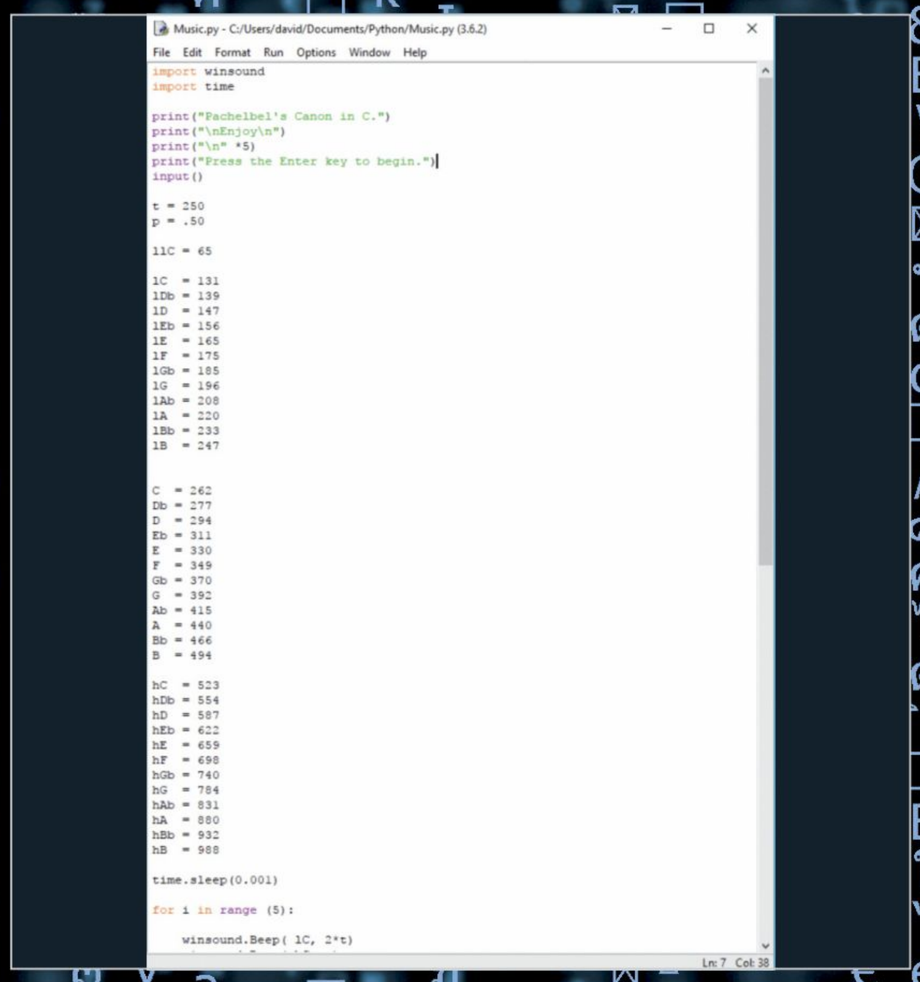
Code Repository

Playing Music with the Winsound Module

Of course, instead of playing an existing MP3, you can always make your own music. The code below will play out Pachelbel's Canon in D, no less.







MUSIC.PY

The code utilises both the Time and Winsound modules, defining the tone and pitch and inserting small pauses of .5 of a second.

```
import winsound
import time

t = 250
p = .50
```

1

```
llC = 65

lC  = 131
lDb = 139
lD  = 147
lEb = 156
lE  = 165
lF  = 175
lGb = 185
lG  = 196
lAb = 208
lA  = 220
lBb = 233
lB  = 247

C  = 262
Db = 277
D  = 294
Eb = 311
E  = 330
F  = 349
Gb = 370
G  = 392
Ab = 415
A  = 440
Bb = 466
B  = 494

hC  = 523
hDb = 554
hD  = 587
hEb = 622
hE  = 659
hF  = 698
hGb = 740
hG  = 784
hAb = 831
hA  = 880
hBb = 932
hB  = 988

time.sleep(0.001)
```

2

```
for i in range (5):

   winsound.Beep( lC, 2*t)
   winsound.Beep( hC, t)
   winsound.Beep( hE, t)
   winsound.Beep( hG, t)
   time.sleep(p)

   winsound.Beep( lG, 2*t)
   winsound.Beep( G, t)
   winsound.Beep( B, t)
   winsound.Beep( hD, t)
   time.sleep(p)

   winsound.Beep( lA, 2*t)
   winsound.Beep( A, t)
   winsound.Beep( hC, t)
   winsound.Beep( hE, t)
   time.sleep(p)

   winsound.Beep( lE, 2*t)
   winsound.Beep( E, t)
   winsound.Beep( G, t)
   winsound.Beep( B, t)
   time.sleep(p)

   winsound.Beep( lF, 2*t)
   winsound.Beep( F, t)
   winsound.Beep( A, t)
   winsound.Beep( hC, t)
   time.sleep(p)

   winsound.Beep( llC, 2*t)
   winsound.Beep( C, t)
   winsound.Beep( E, t)
   winsound.Beep( G, t)
   time.sleep(p)

   winsound.Beep( lF, 2*t)
   winsound.Beep( F, t)
   winsound.Beep( A, t)
   winsound.Beep( hC, t)
   time.sleep(p)

   winsound.Beep( lG, 2*t)
   winsound.Beep( G, t)
   winsound.Beep( B, t)
   winsound.Beep( hD, t)
   time.sleep(p)
```

3

1 The start of the code imports the Winsound and Tie modules; remember, this is a Windows-only Python script. The variable t is setting the duration, while p equals .5, which you can use for the time.sleep function.

2 These variables set the frequencies, with the corresponding numbers, which can be used in the next section of the code.

3 Winsound.beep requires a frequency and duration within the brackets. The frequencies come from the large set of variables called in the second section of the code and the duration is through the t variable set at the start of the code. There's a half-second, using the variable p, pause between blocks of winsound.beep statements.

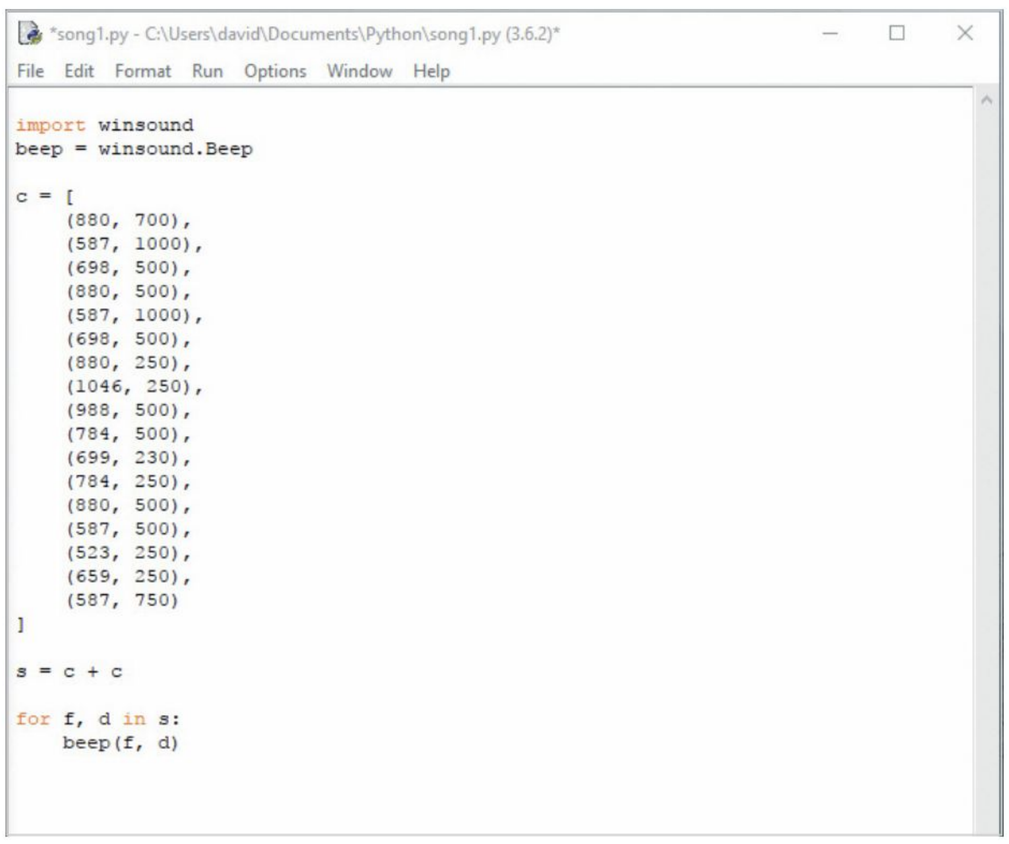
Sweet Music

Obviously the Winsound module is a Windows-only set of functions for Python. Open your IDLE in Windows and copy the code in. Press F5 to save and execute, then press the Enter key, as instructed in the code, to start the music.

Naturally you can swap out the winsound.Beep frequency and durations to suit your own particular music; or you can leave it as is and enjoy. Perhaps play around with the various methods to make other music.

For example, players of the Nintendo classic game, The Legend of Zelda: Ocarina of Time, can enjoy the game's titular musical intro by entering:

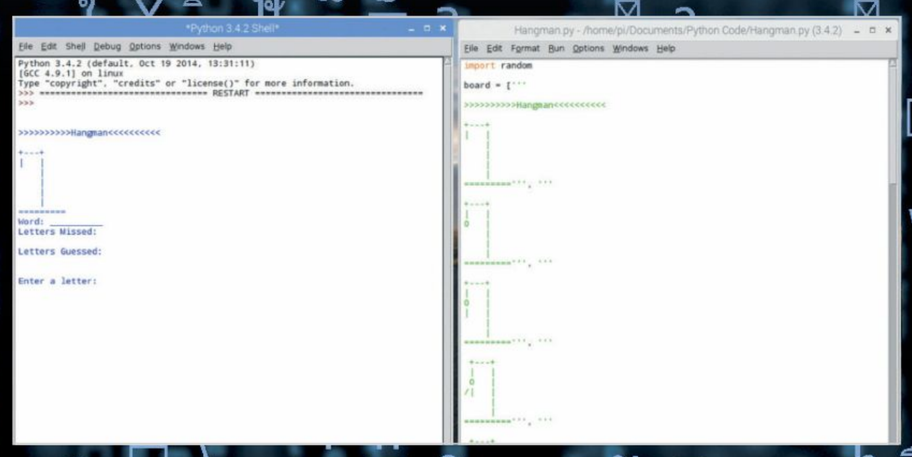
```
import winsound
beep = winsound.Beep

c = [
   (880, 700),
   (587, 1000),
   (698, 500),
   (880, 500),
   (587, 1000),
   (698, 500),
   (880, 250),
   (1046, 250),
   (988, 500),
   (784, 500),
   (699, 230),
   (784, 250),
   (880, 500),
   (587, 500),
   (523, 250),
   (659, 250),
   (587, 750)
]

s = c + c

for f, d in s:
   beep(f, d)
```

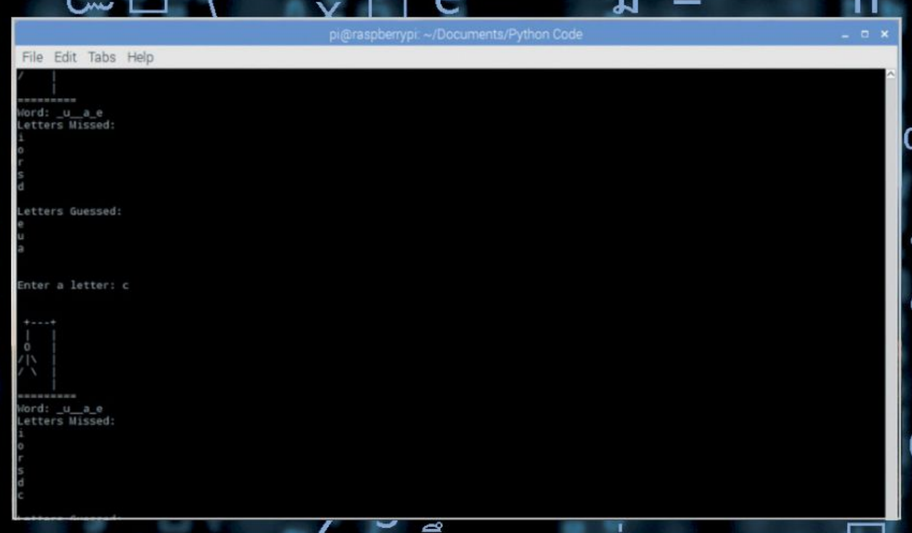


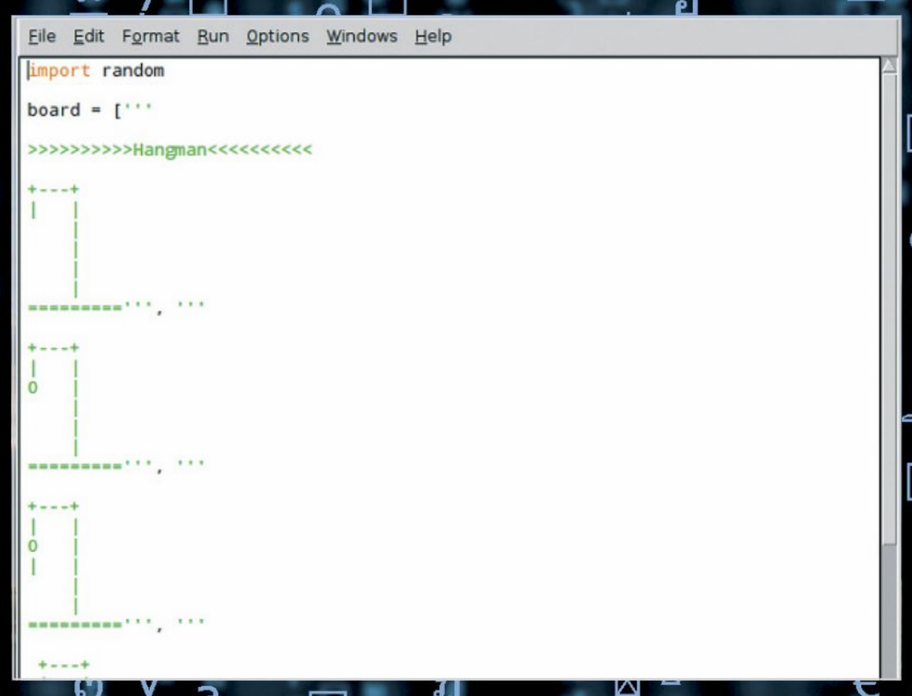
Code Repository

Hangman Game Script

Hangman is a great game to program into Python. It can be extremely complex, displaying graphics, the number of guesses left in the secret word, a huge bank of available words picked at random and countless other elements. It can also be quite simple. Here we have a mix between the two.







HANGMAN.PY

We've made a Hangman game board (the gallows) out of characters that can be displayed in the IDLE Shell, along with a huge bank of words to randomly choose from.

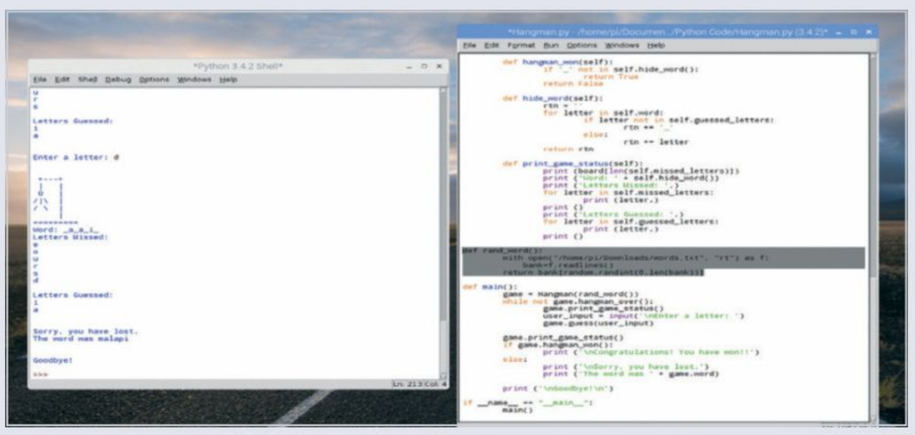
```
import random

board = ['''

>>>>>>>>>>Hangman<<<<<<<<<<

+---+
|   |
    |
    |
    |
    |
=========''', '''

+---+
|   |
O   |
    |
    |
    |
=========''', '''

+---+
|   |
O   |
|   |
    |
    |
=========''', '''

 +---+
 |   |
 O   |
/|   |
     |
     |
=========''', '''

 +---+
 |   |
 O   |
/|\  |
     |
     |
=========''', '''

 +---+
 |   |
 O   |
/|\  |
/    |

=========''', '''

 +---+
 |   |
 O   |
/|\  |
/ \  |

=========''']
```

```
        |
=========''', '''

 +---+
 |     |
 O     |
/|\    |
/ \    |
        |
=========''']


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

   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 ()

def rand_word():
   bank = 'ability about above absolute accessible
   accommodation accounting beautiful bookstore
   calculator clever engaged engineer enough
   handsome refrigerator opposite socks interested
   strawberry backgammon anniversary confused
   dangerous entertainment exhausted impossible
   overweight temperature vacation scissors
   accommodation appointment decrease development
   earthquake environment brand environment necessary
   luggage responsible ambassador circumstance
   congratulate frequent'.split()
   return bank[random.randint(0,len(bank))]

def main():
   game = Hangman(rand_word())
   while not game.hangman_over():
      game.print_game_status()
      user_input = input('\nEnter a letter: ')
      game.guess(user_input)

   game.print_game_status()
   if game.hangman_won():
      print ('\nCongratulations! You have won!!')
   else:
      print ('\nSorry, you have lost.')
      print ('The word was ' + game.word)

   print ('\nGoodbye!\n')

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

QUIT()

Since this is the last example in our Python code repository, we thought we'd go out with a bang and feature the hangman gallows being drawn with each incorrect guess of the word. Don't worry if it looks misaligned in the text here, this is merely due to the differences between using the Python IDLE editor and pasting the code into a word processor (which formats things differently).

There's plenty you can do to improve, enhance and expand on what we've presented here. You can include a routine that returns an error if the user enters a number or character. You can include extra points for someone who guesses the entire word in one go rather than one letter at a time and you could perhaps add Chopin's Funeral March should you lose the game; or something celebratory if you win.



Consider replacing the bank of words too. They're found under the bank list, and could easily be swapped out for something more difficult. If you download www.github.com/dwyl/english-words you can find a text document with over 466,000 words. Perhaps you could swap the words in the bank to instead read the contents of the text file:

```
def rand_word():
   with open("/home/pi/Downloads/words.txt", "rt") as f:
      bank=f.readlines()
   return bank[random.randint(0,len(bank))]
```

Coding Projects & Tips

We're going to finish off by looking at some interesting coding elements that you can incorporate into your own unique projects and code. In this section you'll find code for creating a loading screen for your content, text animations using ASCII art, clever ways in which you can include animated sections at the command prompt, and even code that can track the International Space Station in realtime, while displaying both the astronauts on board and its current latitude and longitude.

There are also pages of common, and language specific, mistakes to help you avoid, and create, error-free and effective code in the future.

178 Creating a Loading Screen

180 Text Animations

182 Tracking the ISS with Python

186 Using Text Files for Animation

188 Passing Variables to Python

190 Python Beginner's Mistakes

192 Glossary of Python Terms

Creating a Loading Screen

If you're looking to add a little something extra to your Python code, then consider including a loading screen. The loading screen is a short introduction, or piece of art, that appears before the main part of your code.

LOAD""

Back in the 80s, in the 8-bit home computing era, loading screens were often used to display the cover of a game as it loaded from tape. The image would load itself in, usually one-line-at-a-time, then proceed to colour itself in while the loading raster bars danced around in the borders of the screen.

Loading screens were a part of the package, and the buy-in for the whole game as an experience. Some loading screens featured animations, or a countdown for time remaining as the game loads, while others even went so far as to include some kind of playable game. The point being: a loading screen is not just an artistic part of computing history, but an introduction to the program that's about to be run.

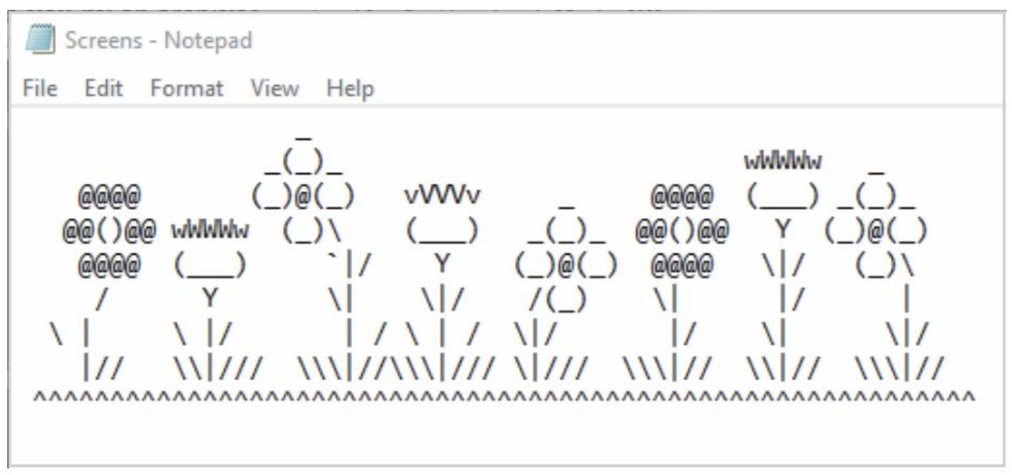
While these days loading screens may no longer be with us, in terms of modern gaming we can still include them in our own Python content. Either just for fun, or to add a little retro-themed spice to the mix.

SCREEN$

Creating a loading screen in Python is remarkably easy. You have several options to hand: you can create a tkinter window and display an image, followed by a brief pause, before starting your main code, or you could opt for a console-based ASCII art version, that loads one-line-at-a-time. Let's look at the latter and see how it works.

First you'll need some ASCII art, you can look up plenty of examples online, or use an image to ASCII Art converter to convert any images you have to ASCII. When you have your ASCII art, drop it into a newly created folder inside a normal text file.

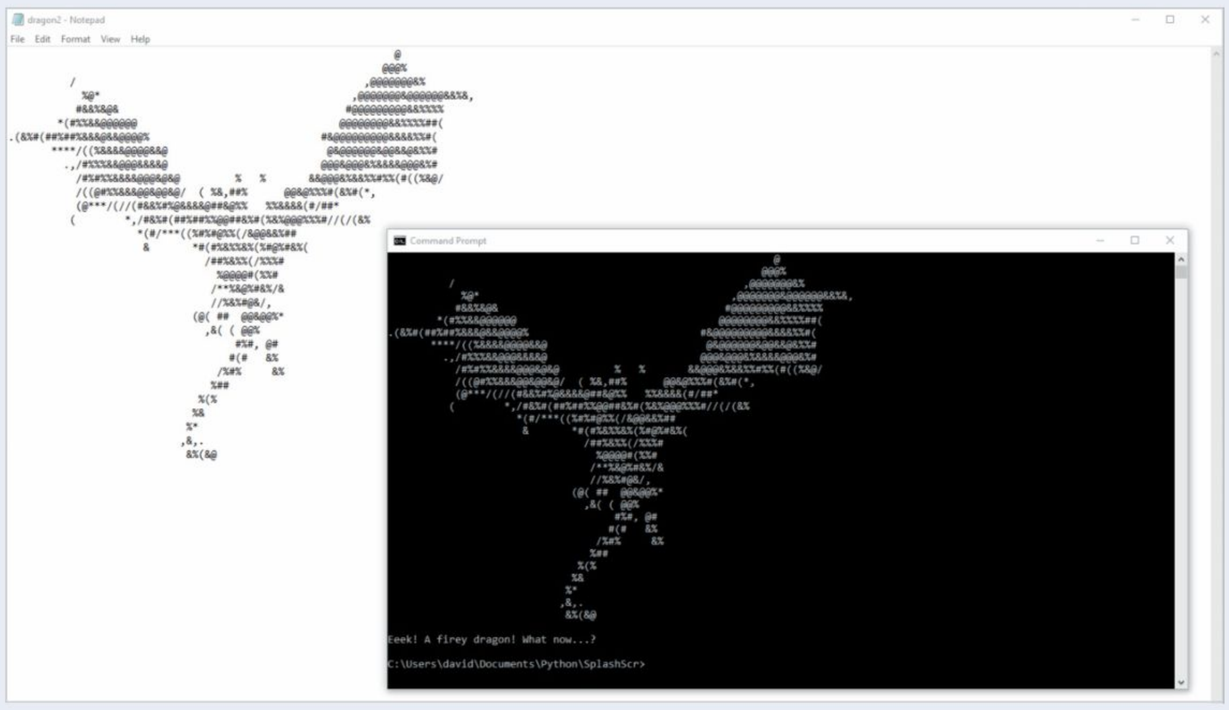
Save the file, call it screens.txt for now.



ADVENTURE TIME

A good example of using loading screen ASCII art, text images, is when coding a text adventure. Once you've established your story, created the characters, events and so on, you could easily incorporate some excellently designed ASCII art to your game.

Imagine coming across a dragon, in game, and displaying its representation as ASCII. You can then load up the image lines one-by-one, and continue with the rest of the adventure code. It's certainly worth having a play around with and it'll definitely add a little something else extra.



THE CODE

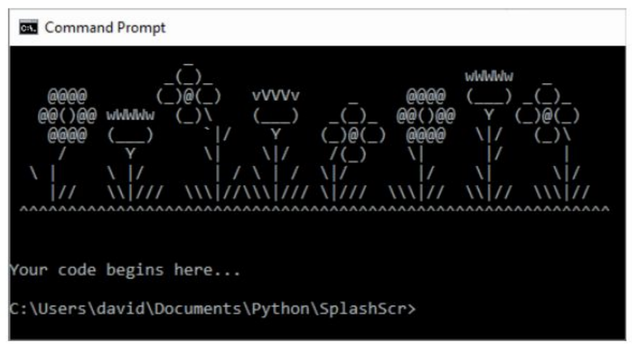
Launch Python and enter the following code to a New File:

```
import os
import time

def loading _ screen(seconds):
    screens=open("screens.txt", 'r')
    for lines in screens:
        print(lines, end='')
        time.sleep(seconds)
    screens.close()


#Main Code Start
os.system('cls' if os.name == 'nt' else 'clear')
loading _ screen(.5)

print ("\nYour code begins here...")
```

The code is quite simple: import the OS and Time modules and then create a Python function called loading_screen with a (seconds) option. Within the function, open the text file with the ASCII art as read-only and create a For loop that'll read the text file one-line-at-a-time. Next, print the lines – incidentally, the **lines, end=''** element will strip the newline from the text document, without it you'll end up with a double-line spaced display. Include the timing in seconds and close the text file buffer.

The final part of the code, #Main Code Start, is where you'll clear the screen (CLS for Windows, Clear for Linux and macOS) and call the function, together with the output number of seconds to wait for each line to be written to the screen – in this case, .5 of a second.

Save the code as screens.py, drop into a Command Prompt or Terminal and execute it. The screen will clear and your ASCII art inside the text file will load in line-by-line, creating a loading screen effect.




```
Loading...
```

Another favourite introduction screen is that of a simple loading animation, where the word loading is displayed followed by some characters and a percentage of the program loaded. While it may not take long for your Python code to load, the effect can be interesting.

Create a New File in Python and enter the following code:

```
import os
import time

def loading _ bar(seconds):
  for loading in range(0,seconds+1):
    percent = (loading * 100) // seconds
    print("\n")
    print("Loading...")
    print("<" + ("-" * loading) + (" " * (seconds
    loading)) + "> " + str(percent) + "%")
    print("\n")
    time.sleep(1)
    os.system('cls' if os.name == 'nt' else 'clear')


#Main Code Start
loading _ bar(10)

print ("\nYour code begins here...")
```

The code works in much the same way as the previous except, instead of reading from a text file, it's running through a for loop that prints Loading… followed by an animation of sorts, along with a percentage counter; clearing the screen every second and displaying the new results. It's simple, yes, but quite effective in its design.

COMBINING THE TWO

How about combining the two elements we've looked at? Let's begin with a Loading… progress bar, followed by the loading screen. After that, you can include your own code and continue your program. Here's the code:

```
import os
import time

def loading _ bar(seconds):
  for loading in range(0,seconds+1):
    percent = (loading * 100) // seconds
    print("\n")
    print("Loading...")
    print("<" + ("-" * loading) + (" " * (seconds-
    loading)) + "> " + str(percent) + "%")
    print("\n")
    time.sleep(1)
    os.system('cls' if os.name == 'nt' else 'clear')


def loading _ screen(seconds):
    screens=open("screens.txt", 'r')
    for lines in screens:
        print(lines, end='')
        time.sleep(seconds)
    screens.close()


#Main Code Start
loading _ bar(10)
os.system('cls' if os.name == 'nt' else 'clear')
loading _ screen(.5)


print ("\nYour code begins here...")
```

You can of course pull those functions up wherever and whenever you want in your code and they'll display, as they should, at the beginning. Remember to have the ASCII text file in the same folder as the Python code, or, at the **screens=open("screens.txt", 'r')** part of the code, point to where the text file is located.

Text Animations

There's a remarkable amount you can do with some simple text and a little Python know-how. Combining what you've already learned, we can create some interesting animation effects from the command line.

THE FINAL COUNTDOWN

Let's begin with some example code that will display a large countdown from ten, then clear the screen and display a message. The code itself is quite simple, but lengthy. You will need to start by importing the OS and Time modules, then start creating functions that display the numbers:

```
def one():
    print("""
  1111111
 1::::::1
1:::::::1
111::::::1
   1::::1
   1::::1
   1::::1
   1::::1
   1::::1
   1::::1
   1::::1
   1::::1
111::::::111
1::::::::::1
1::::::::::1
111111111111
  """)
```

```
def two():
    print("""
 222222222222222
2::::::::::::::22
2::::::222222:::::2
2222222      2:::::2
             2:::::2
             2:::::2
          2222::::2
     22222::::::22
   22::::::::222
 2:::::22222
2:::::2
2:::::2
2:::::2        222222
2::::::2222222:::::2
2::::::::::::::::::2
22222222222222222222
  """)
```

```
def three():
    print("""
 333333333333333
3::::::::::::::33
3::::::33333::::::3
3333333      3:::::3
             3:::::3
             3:::::3
     33333333:::::3
     3:::::::::::3
     33333333:::::3
             3:::::3
             3:::::3
             3:::::3
3333333      3:::::3
3::::::33333::::::3
3::::::::::::::::33
 333333333333333
  """)
```

and so on to 10. It'll take some time, but it's worth it in the end. Of course, you can always take a different approach and design the numbers yourself.

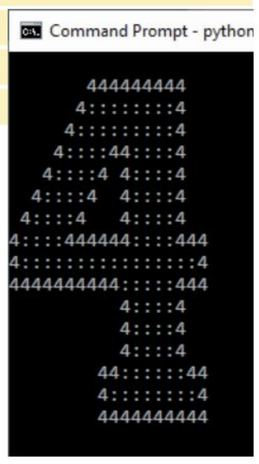
The next step of the process is to initialise the code settings and start the countdown:

```
#Initialise settings
start = 10
message = "> BLAST OFF!! <"

#Start the countdown
for counter in range(start, 0, -1):
    if counter == 10:
        ten()
    elif counter == 9:
        nine()
    elif counter == 8:
        eight()
    elif counter == 7:
        seven()
    elif counter == 6:
        six()
    elif counter == 5:
        five()
    elif counter == 4:
        four()
    elif counter == 3:
        three()
    elif counter == 2:
        two()
    elif counter == 1:
        one()
    time.sleep(1)
    os.system('cls' if os.name == 'nt' else 'clear')
```

And finally, we can add a display for the message:

```
#Display the message
print("v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v")
print("<                                >")
print(message)
print("<                                >")
print("v^v^v^v^v^v^v^v^v^v^v^v^v^v^v^v")
print("\n\n\n")
```

The code in its entirety can be viewed from within our Code Repository: **https://bdmpublications.com/ code-portal**, where you're free to copy it to your own Python IDLE and use it as you see fit. The end effect is quite good and it'll be worth adding to your own games, or presentations, in Python.



To extend the code, or make it easier to use, you can always create the number functions in their own Python code, call it Count.py for example, then import Count at the beginning of a new Python file called Countdown.py, along with the OS and Time modules:

```
import os
import time
import count
```

From there, you will need to specify the imported code in the Countdown section:

```
#Start the countdown
for counter in range(start, 0, -1):
    if counter == 10:
        count.ten()
    elif counter == 9:
        count.nine()
    elif counter == 8:
        count.eight()
    elif counter == 7:
       count.seven()
    elif counter == 6:
        count.six()
    elif counter == 5:
        count.five()
    elif counter == 4:
        count.four()
    elif counter == 3:
        count.three()
    elif counter == 2:
        count.two()
    elif counter == 1:
        count.one()
```

This will pull the functions from the imported Count.py and print them to the screen.

BLAST OFF!!

Building on the previous countdown example, we can create an animated rocket that'll launch after the Blast Off!! message has been printed. The code for this would look something like this:

```
def Rocket():
  distanceFromTop = 20
  while True:
    os.system('cls' if os.name == 'nt' else 'clear')
    print("\n" * distanceFromTop)
    print("          /\        ")
    print("          ||        ")
    print("          ||        ")
    print("         /||\       ")
    time.sleep(0.2)
    os.system('cls' if os.name == 'nt' else 'clear')
    distanceFromTop -= 1
    if distanceFromTop <0:
      distanceFromTop = 20


#Launch Rocket
Rocket()
```

Here we've created a new function called Rocket, which produces the effect of an ASCII-like rocket taking off and scrolling upwards; using the distanceFromTop variable.

To use this, add it to the end of the previous countdown code and at the end of the Blast Off!! message, add the following lines:

```
print("\n\n\n\n")
input("Press Enter to launch rocket...")
```

This will allow your message to be displayed and then, when the user has hit the Enter button, the rocket will launch.

Again, the code in its entirety can be found in the Code Repository at: **https://bdmpublications.com/ code-portal**.

ROLLING DIE

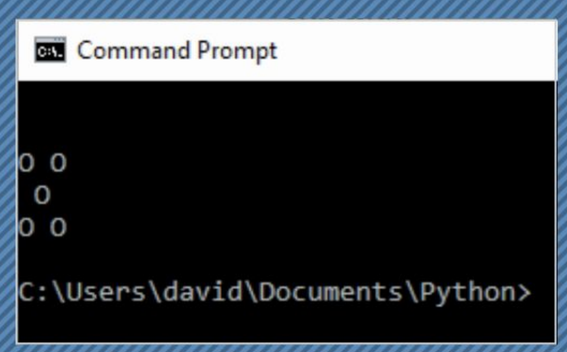
Aside from the rocket animation, together with its countdown, another fun bit of text-based animation is that of a rolling dice.

A rolling dice can be a great animation to include in an adventure game, where the player rolls to see what their score is compared to that of an enemy. The highest roller wins the round and the losers' health drops as a result. It's an age-old combat sequence, used mainly in the Dungeon and Dragons board games and Fighting Fantasy novels, but it works well.

The code you'll need to animate a dice roll is:

```
import os
import time
from random import randint

die     = ["    \n O \n    "]    #1
die.append("   O\n    \nO   ")    #2
die.append("O   \n O \n   O")    #3
die.append("O O\n    \nO O")    #4
die.append("O O\n O \nO O")    #5
die.append("O O\nO O\nO O")    #6

def dice():
  for roll in range(0,15):
    os.system('cls' if os.name == 'nt' else 'clear')
    print("\n")
    number = randint(0,5)
    print(die[number])
    time.sleep(0.2)

#Main Code Begins
dice()
```

You may need to tweak the O entries, to line up the dots on the virtual dice. Once it's done, though, you'll be able to add this function to your adventure game code and call it up whenever your character, or the situation, requires some element of luck, combat or chance roll of the dice.



Tracking the ISS with Python

Of the many amazing human achievements over the past decade or so, the International Space Station tops the bill. This incredible collaboration between nations sees vital experiments carried out in space as well as observations of our own planet.

TO BOLDLY GO...

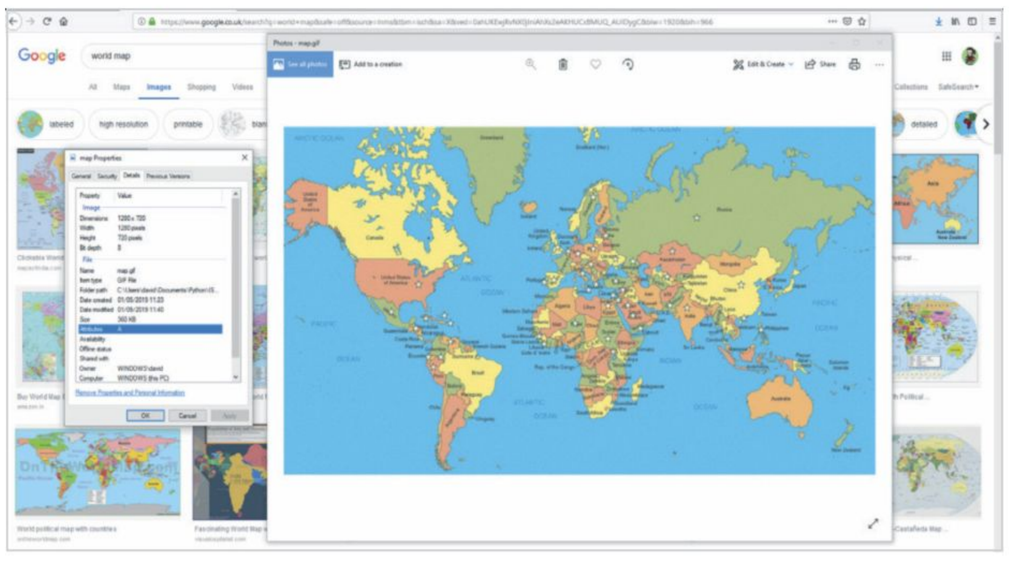
Indeed, the ISS is something most of us consider as a worthy example of what can happen when we work together. NASA, among other agencies, uses a wealth of Python code onboard the ISS to help automate routines, as well as act as an in-between link to translate code from one language to another, and then into a human-readable format. If you're considering a career in space, then learning Python is a must-have skill.

While we're not able to fill you in on all the details, regarding the code the ISS utilises, we can look at a fun Python script that will track the ISS, display the number of astronauts on board, update the station's current latitude and longitude every five seconds, while also displaying your current latitude and longitude.

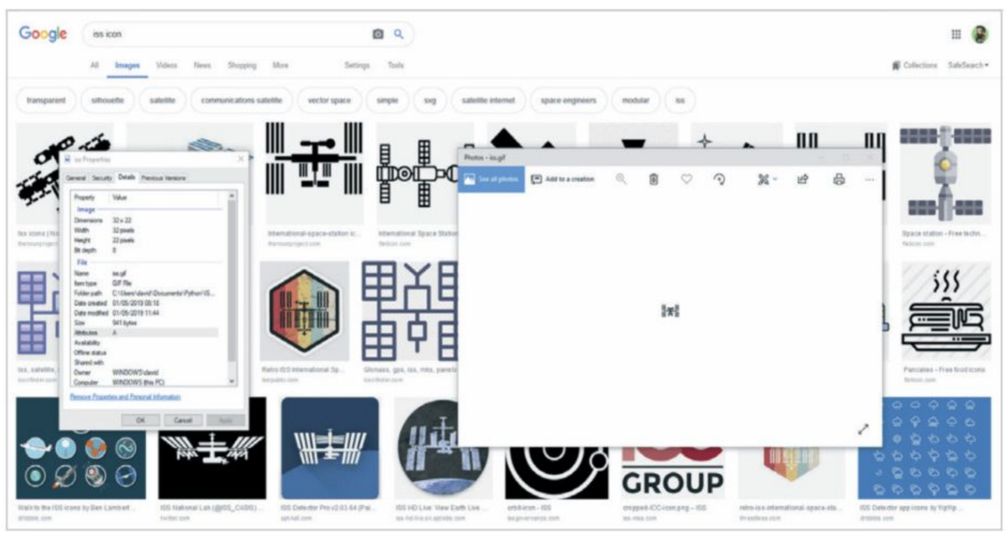
Displaying all that information in a single screen can become a little cluttered, so in this instance we're going to look at spreading all those details over three screens: a text document window -displaying the astronauts and your current latitude and longitude, a command line (or Terminal window) - displaying the continually updating latitude and longitude of the ISS as it orbits Earth, and a final, large window displaying a map of the world, together with an icon representing the ISS, that's updated as it orbits. Interested? Read on.

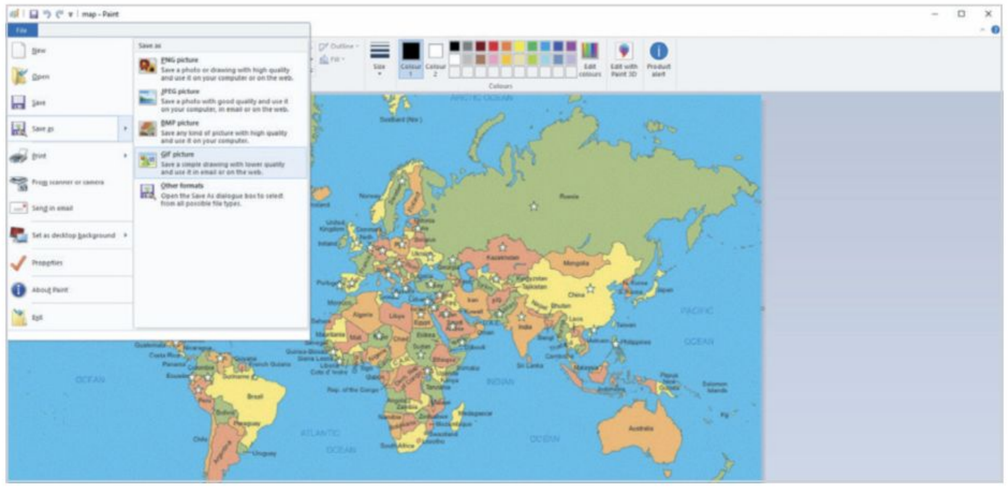
THE GRAPHICS

Firstly, we need to get hold of a map of the world, and an image of the ISS to use as an icon, that will be updated according to the position of the space station as it travels over the surface. A quick Google of World Map will help you out here. Look for one that's reasonably large, the one we used for this example was 1280 x 700, and one that has the names of the countries – if you're using this with young people, to help with putting shapes of countries to names.

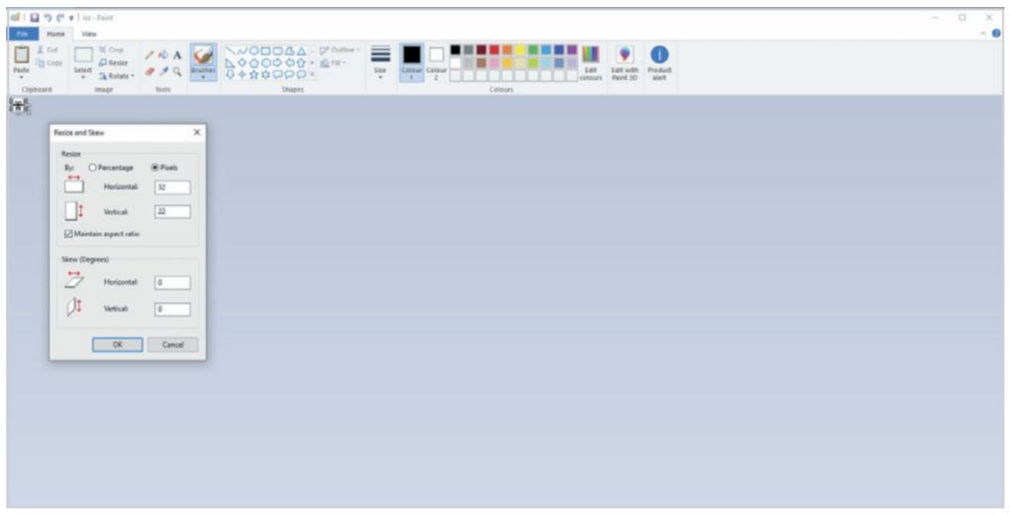


Next, look for an ISS icon. As this is going to be a graphical representation of the location of the ISS, we need the image to be reasonably small so it doesn't drown out the locations on the map, but also prominent enough to see when the map is loaded. We opted for an image that's 32 x 22 pixels in size. Don't worry too much if you're not able to find one that small, you can always resize it in an image-editing app such as Paint or GIMP.





As we're going to be using Turtle, a component of tkinter, the downloaded images will need to be converted to GIF, since this is the default and recommended image format. You can easily look up a converter online, but using Paint in Windows 10, or GIMP, which is cross-platform, will suffice.

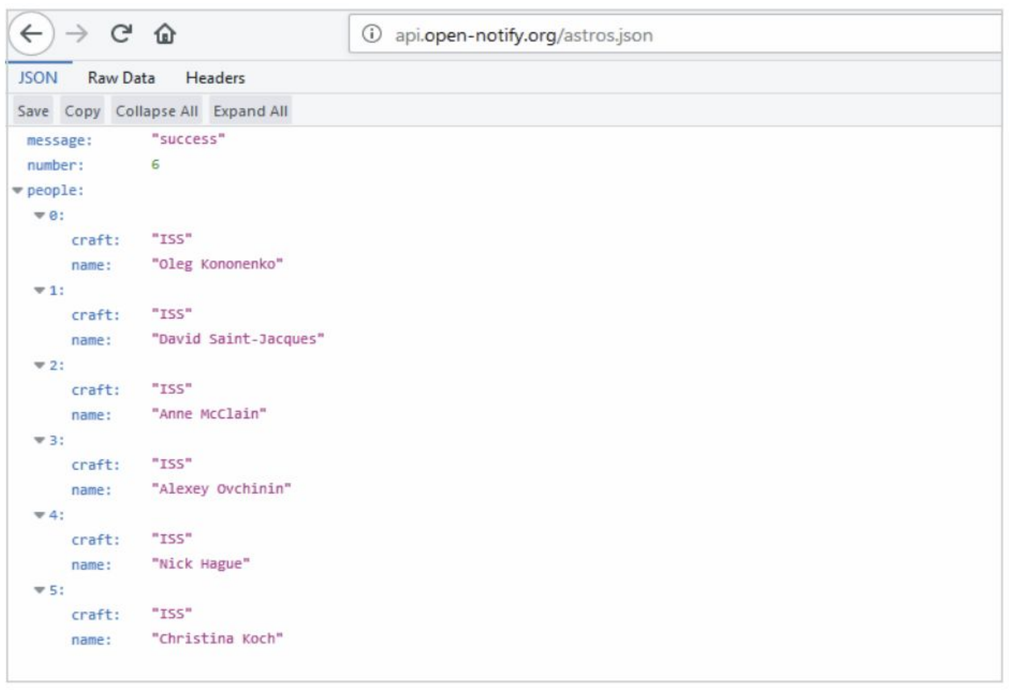


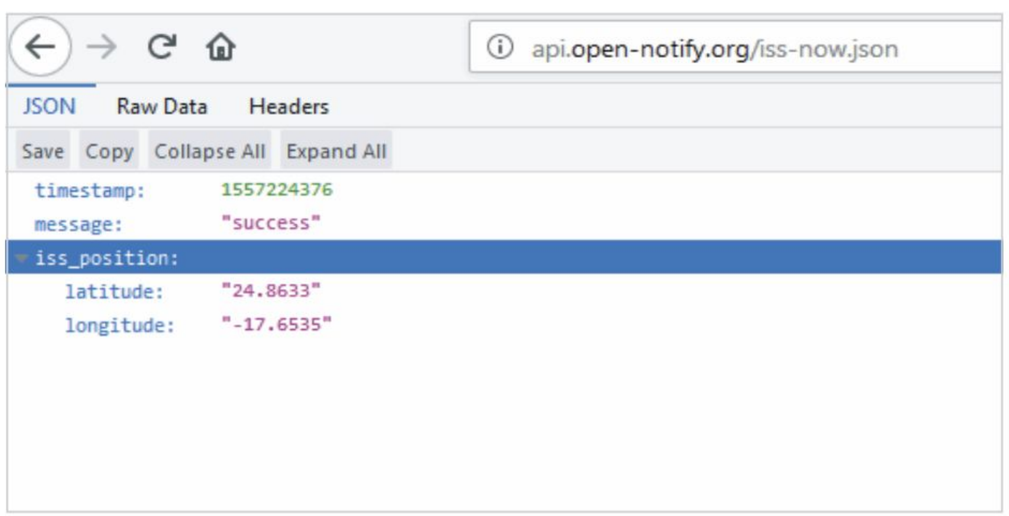
Simply load the image up in your image editor app and choose Save As, call them map and iss respectively, and click GIF as the image format. Remember to also resize the ISS image before saving it as a GIF.

THE CODE

The code we're using here utilises an open source API (Application Programming Interface) to retrieve real-time data online regarding the status of the ISS. An API enables applications to communicate with one another by providing the raw data that a programmer can pull out and interact with in their own code. In this case, the API in question is a web-based collection of raw data that's stored in a JSON (JavaScript Object Notation) format – an accessible and easy to use data-interchange interface.

In order for Python to interact with the JSON provided data, it needs to use the urllib.request and json modules. We're also going to be using a new module called geocoder, which will pull up a users' current latitude and longitude based on their IP address. The two JSON API's can be located at: **http://api.open-notify.org/astros.json**, and, **http://api.open-notify.org/iss-now.json**. One of which contains the data regarding the astronauts onboard the ISS, and the other contains the data regarding the current location of the ISS in longitude and latitude.





Let's begin by breaking the code into bite-sized chunks:

```
import json, turtle, urllib.request, time, webbrowser
import geocoder # need to pip install geocoder for your
lat/long to work.
```

First, we need to import the modules used in the code: json, turtle, urlib.request, time and webbrowser. The json and urlib.request modules deal with the data being pulled from the APIs, turtle will display the graphics, and time you already know. The webbrowser module will open text files in the default text file application – despite what its name may suggest. The geocoder module is a new element, and you will need to install it with: **pip install geocoder**. Geocoder can retrieve a users' location based on their IP address, as each ISP will have a geo-specific IP.

```
#Retrieve the names of all the astronauts currently on
board the ISS, and own lat/long - write to a file and display
url = "http://api.open-notify.org/astros.json"
response = urllib.request.urlopen(url)
result = json.loads(response.read())
a=open("iss.txt","w")
a.write("There are currently " + str(result["number"]) + "
astronauts on the ISS:\n\n")

people = result["people"]

for p in people:
  a.write(p["name"] + " - on board" + "\n")

g=geocoder.ip('me') # need to pip install geocoder, and
import as in the headers above
a.write("\nYour current Lat/Long is: " + str(g.latlng)) #
prints your current lat/long in the text file.
a.close()
webbrowser.open("iss.txt")
```

This section will use the json and urllib.request modules to pull the data from the API that contains the names of the astronauts onboard the ISS. It then creates a new text file called iss.txt, where it'll write 'There are currently X astronauts on the ISS…' and list them by name. The second part of this section will then use the geocoder module to retrieve your current latitude and longitude, based on your IP address, and also write that information to the end of the text file that contains the names of the astronauts. The last line, webbrowser.open("iss.txt") will use the webbroser module to open and display the newly written text file in the system's default text file reading app.

```
#Setup world map in Turtle
screen = turtle.Screen()
screen.setup(1280, 720)
screen.setworldcoordinates(-180, -90, 180, 90)
#Load the world map image
screen.bgpic("map.gif")

screen.register _ shape("iss.gif")
iss = turtle.Turtle()
iss.shape("iss.gif")
iss.setheading(45)
iss.penup()
```

This section of the code sets up the graphical window containing the world map and the ISS icon. To begin we set up the turtle screen, using the resolution of the world map image we downloaded at the start of this tutorial (1280 x 720). The screen.setworldcoordinates syntax will mark the boundaries of the screen, creating the x and y coordinates of

the four corners of the canvas, so that the ISS icon can freely travel across the map of the world and wrap itself back to the opposite side when it reaches an edge. The ISS icon is set as the turtle pen shape, giving it an angle of 45 degrees when moving.

```
while True:
  #Load the current status of the ISS in real-time
  url = "http://api.open-notify.org/iss-now.json"
  response = urllib.request.urlopen(url)
  result = json.loads(response.read())

  #Extract the ISS location
  location = result["iss _ position"]
  lat = location["latitude"]
  lon = location["longitude"]

  #Output Latitude and Longitude to the console
  lat = float(lat)
  lon = float(lon)
  print("\nLatitude: " + str(lat))
  print("Longitude: " + str(lon))

  #Update the ISS location on the map
  iss.goto(lon, lat)
  #refresh every 5 seconds
  time.sleep(5)
```

Now for the final part of the code: Here we collect the location data from the ISS status API, pulling out the latitude and longitude elements of the JSON file. The code then prints the latitude and longitude data to the console/Terminal, transferring the data from a float to a string in order to print it correctly. The last section will use the latitude and longitude as variables lat and lon, to update the ISS icon on the map every five seconds.

Here's the code in its entirety:

```
import json, turtle, urllib.request, time, webbrowser
import geocoder # need to pip install geocoder for your
lat/long to work.

#Retrieve the names of all the astronauts currently on
board the ISS, and own lat/long - write to a file and display
url = "http://api.open-notify.org/astros.json"
response = urllib.request.urlopen(url)
result = json.loads(response.read())
a=open("iss.txt","w")
a.write("There are currently " + str(result["number"]) + "
astronauts on the ISS:\n\n")

people = result["people"]

for p in people:
  a.write(p["name"] + " - on board" + "\n")

g=geocoder.ip('me') # need to pip install geocoder, and
import
as in the headers above
a.write("\nYour current Lat/Long is: " + str(g.latlng)) #
prints
your current lat/long in the text file.
a.close()
webbrowser.open("iss.txt")

#Setup world map in Turtle
screen = turtle.Screen()
screen.setup(1280, 720)
screen.setworldcoordinates(-180, -90, 180, 90)
#Load the world map image
screen.bgpic("map.gif")
```

```
while True:
  #Load the current status of the ISS in real-time
  url = "http://api.open-notify.org/iss-now.json"
  response = urllib.request.urlopen(url)
  result = json.loads(response.read())

  #Extract the ISS location
  location = result["iss_position"]
  lat = location["latitude"]
  lon = location["longitude"]

  #Output Latitude and Longitude to the console
  lat = float(lat)
  lon = float(lon)
  print("\nLatitude: " + str(lat))
  print("Longitude: " + str(lon))

  #Update the ISS location on the map
  iss.goto(lon, lat)
  #refresh every 5 seconds
  time.sleep(5)
```

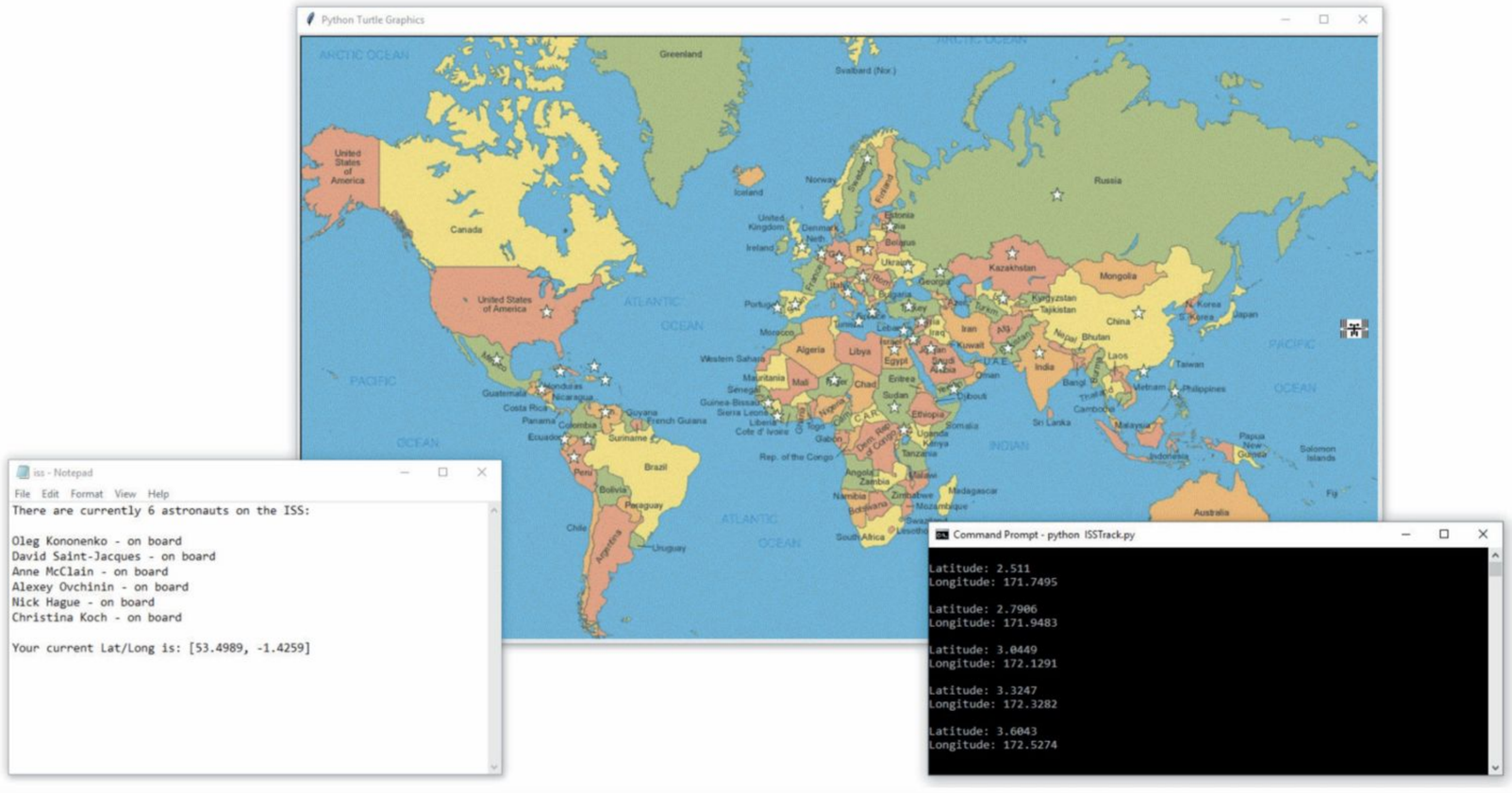
```
screen.register _ shape("iss.gif")
iss = turtle.Turtle()
iss.shape("iss.gif")
iss.setheading(45)
iss.penup()

while True:
  #Load the current status of the ISS in real-time
  url = "http://api.open-notify.org/iss-now.json"
  response = urllib.request.urlopen(url)
  result = json.loads(response.read())

  #Extract the ISS location
  location = result["iss _ position"]
  lat = location["latitude"]
  lon = location["longitude"]

  #Output Latitude and Longitude to the console
  lat = float(lat)
  lon = float(lon)
  print("\nLatitude: " + str(lat))
  print("Longitude: " + str(lon))
```

Create a new folder in your system, called ISSTrack (for example), and place the two graphics, as well as the Python code itself.

RUNNING THE CODE

The code is best executed from the command line or Terminal. Clear your desktop, enter your command line and navigate to where you've saved the code plus the two graphics. Launch the code with either: python ISSTrack.py or Python3 ISSTrack.py (depending on your system, and what you've called the Python code).

The code will launch two extra windows together with the command line window you already have open. One will be the text file containing the named astronauts along with your current latitude and longitude, and the other will be the world map with the ISS icon located wherever the ISS is currently orbiting. The command line window will start scrolling through the changing latitude and longitude of the ISS.



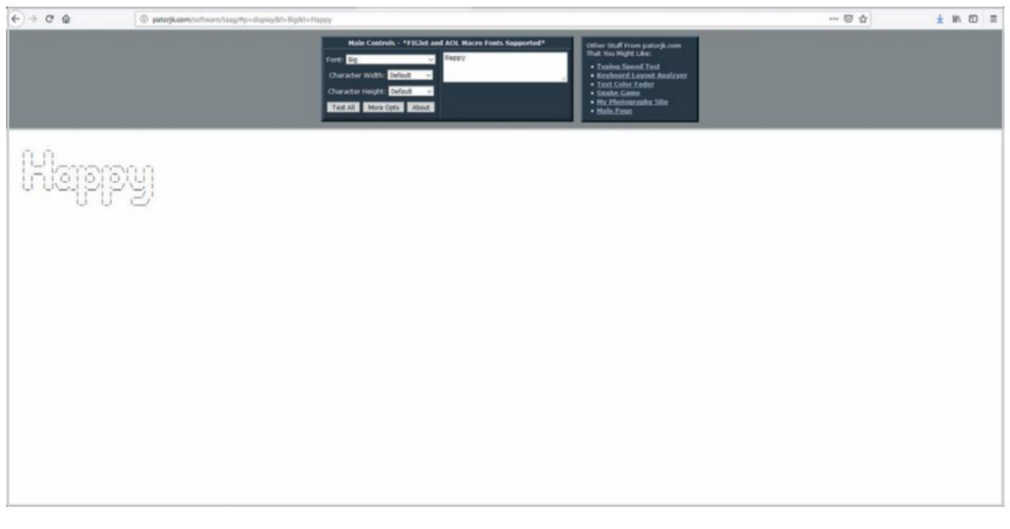
Using Text Files for Animation

Animation in Python can be handled with the likes of the Tkinter and Pygame modules, however, there's more than one way to achieve a decent end result. Using some clever, text file reading code, we can create command-line animations.

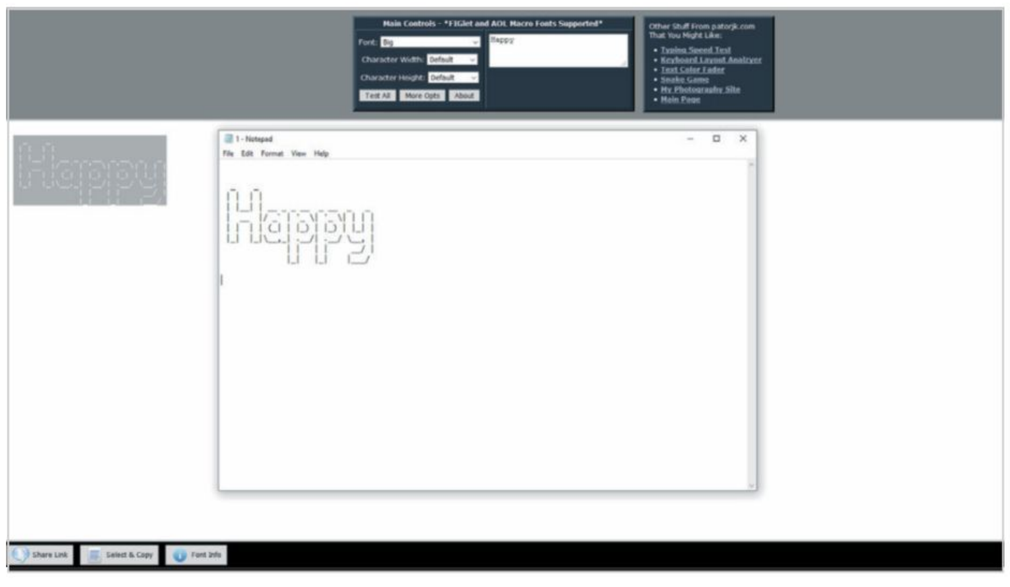
ASCII ANIMATION

Let's assume you wanted to create an animated ASCII Happy Birthday Python script, with the words Happy and Birthday alternating in appearance. Here's how it's done.

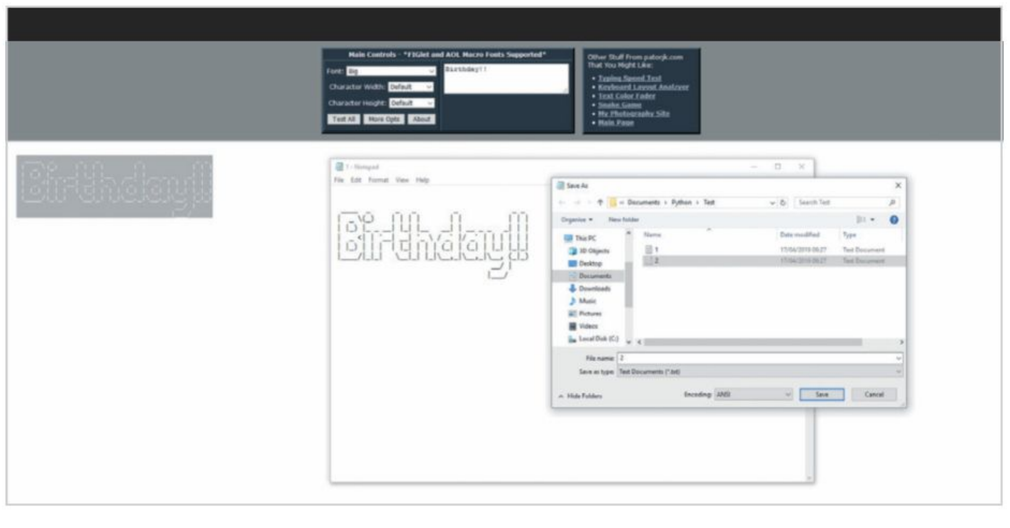
STEP 1 First we need to create some ASCII-like text, head over to **http://patorjk.com/software/taag**. This is an online Text to ASCII generator, created by Patrick Gillespie. Start by entering **Happy** into the text box, the result will be displayed in the main window. You can change the font with the drop-down menu, to the side of the text box; we've opted for **Big**.



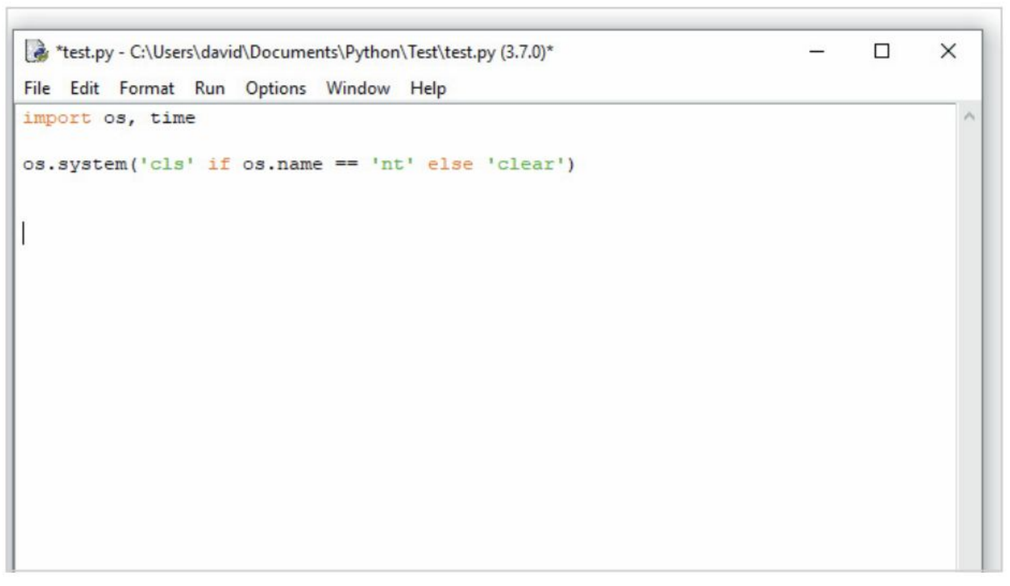
STEP 2 Now create a folder in your Python code directory on your computer (call it Test, for now), and open either Notepad for a Windows 10 computer, or, if you're using Linux, then the currently installed text editor. Click on the **Select & Copy** button at the bottom of the ASCII Generator webpage, and paste the contents into the text editor.



STEP 3 Save the text file as **1.txt** (you can call it what you like, but now for ease of use 1.txt will suffice). Save the file in the newly created Test folder. When it's saved, do exactly the same for the word Birthday. You can select a new font from the ASCII Generator, or add extra characters and when you're ready, save the file as **2.txt**.



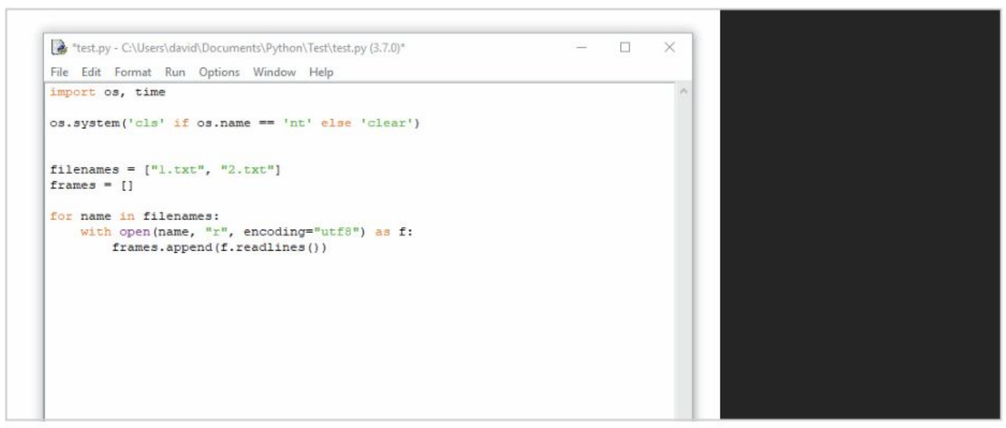
STEP 4 Open up Python and create a **New File**. We're going to need to import the OS and Time modules for this example, followed by a line to clear the screen of any content. If you're using Windows, then you'll use the CLS command, whereas it's Clear for Linux. We can create a simple if/else statement to handle the command.



STEP 5

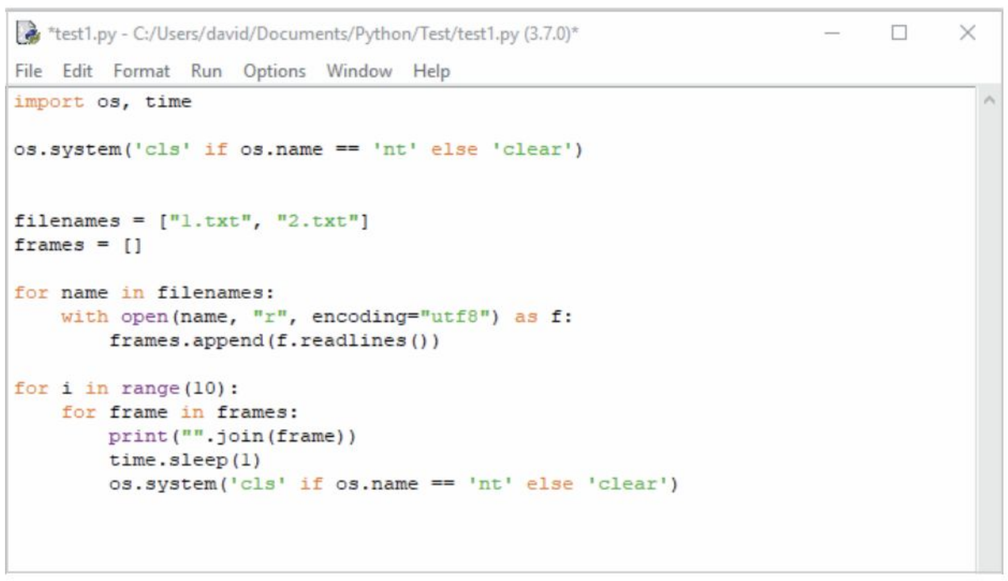
Next we need to create a list of the names of the text files we want to open, and then we need to open them for display in the Terminal.

```
filenames = [“1.txt”, “2.txt”]
frames = []

for name in filenames:
    with open(name, “r”, encoding=”utf8”) as f:
        frames.append(f.readlines())
```



STEP 6

We’ve used the UTF8 standard when opening the text files, as ASCII art as text, within a text file, often requires you to save the file as UTF compliant – due to the characters used. Now we can add a loop to display the files as 1.txt, then 2.txt, creating the illusion of animation while clearing the screen after each file is displayed.

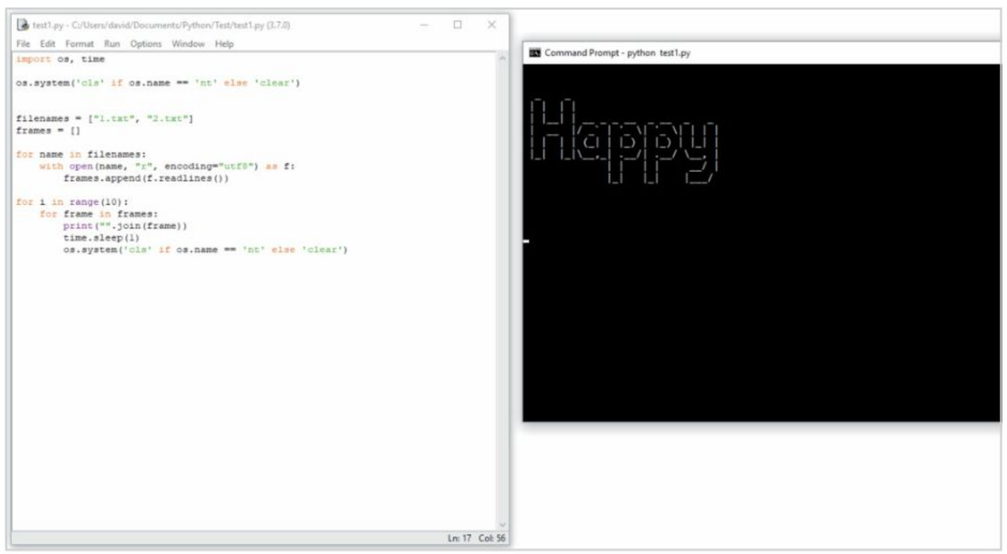


STEP 7

Save the Python code in the same folder as the text files and drop into a Terminal or Command Prompt. Navigate to the folder in question, and enter the command:

```
python NAME.py
```

Where NAME is whatever you called your saved Python code.



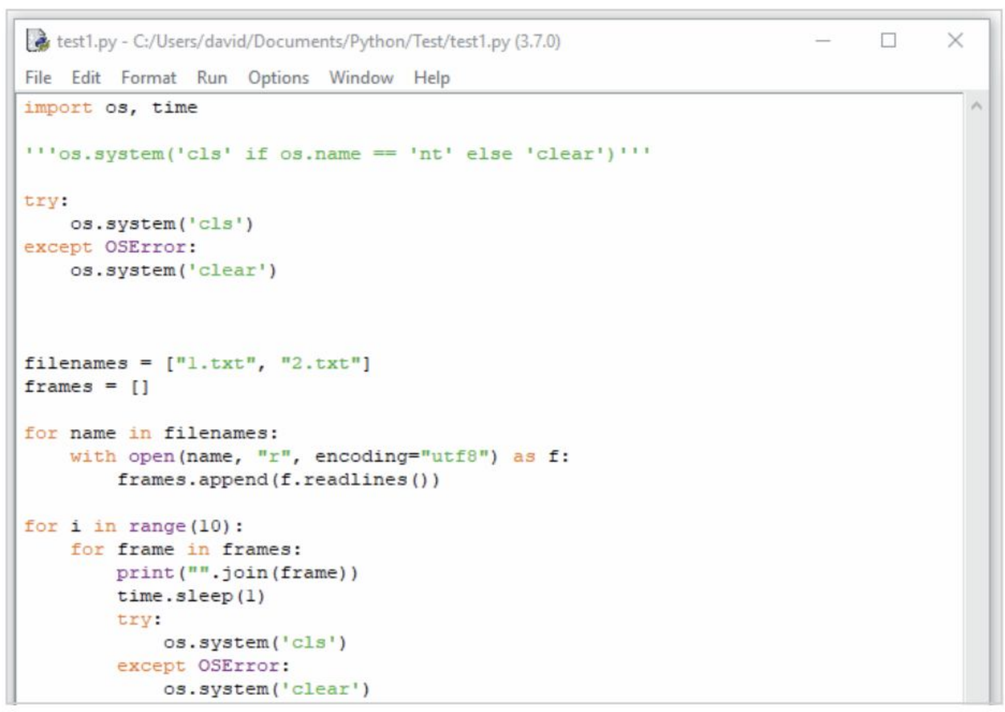
STEP 8

Here’s the code in full:

```
import os, time

os.system(‘cls’ if os.name == ‘nt’ else ‘clear’)

filenames = [“1.txt”, “2.txt”]
frames = []

for name in filenames:
    with open(name, “r”, encoding=”utf8”) as f:
        frames.append(f.readlines())

for i in range(10):
    for frame in frames:
        print(“”.join(frame))
        time.sleep(1)
        os.system(‘cls’ if os.name == ‘nt’ else
        ‘clear’)
```

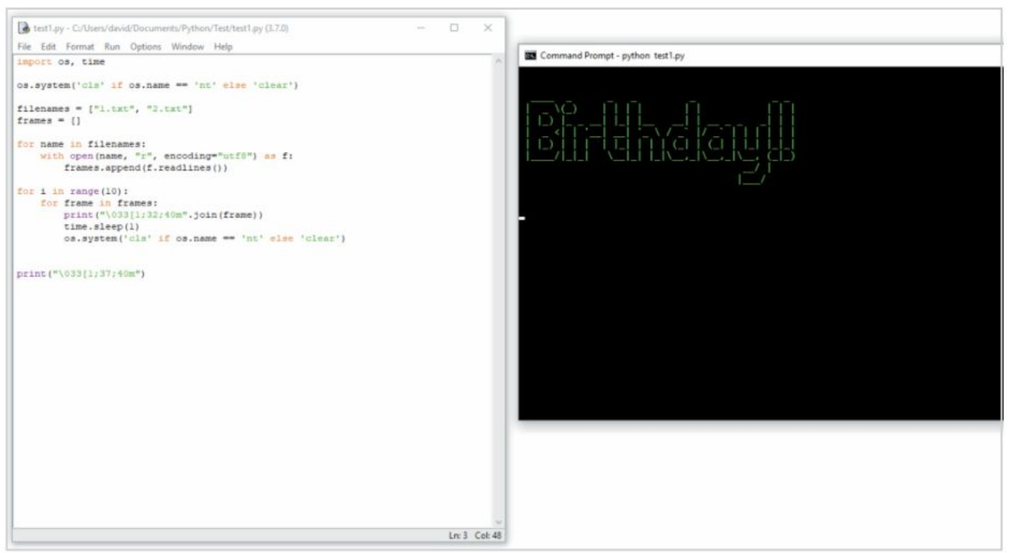
STEP 9

Note from the loop within the code, we’ve used the same CLS and Clear if/else statement as before. Again, if you’re running on Windows then the OS module will use the CLS command, ‘ELSE’ if you’re using Linux or a Mac, the Clear command will work correctly. If you want, you could use a Try/Except statement instead.



STEP 10

You can spice things up a little by adding system/Terminal colours. You’ll need to Google the system codes for the colours you want. The code in our example turns the Windows Command Line to green text on a black background, then changes it back to white on black at the end of the code. Either way, it’s a fun addition to your Python code.



Passing Variables to Python

Here's an interesting coding tutorial: how to pass a system variable to Python. By this we mean, if you create a variable within a Windows batch file, or Linux Bash script, then you're able to use that same variable from inside Python.

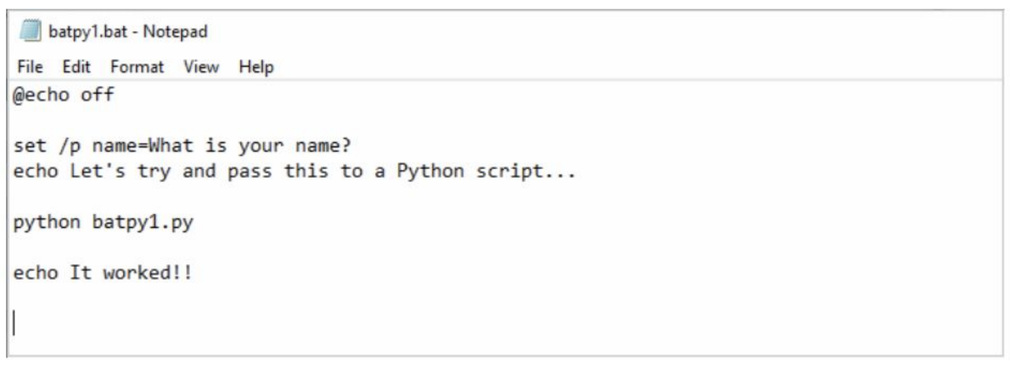
THE WINDOWS WAY

The two systems use moderately different ways to accomplish the same task, Windows is slightly easier (by a single command), and so we'll start there.

STEP 1

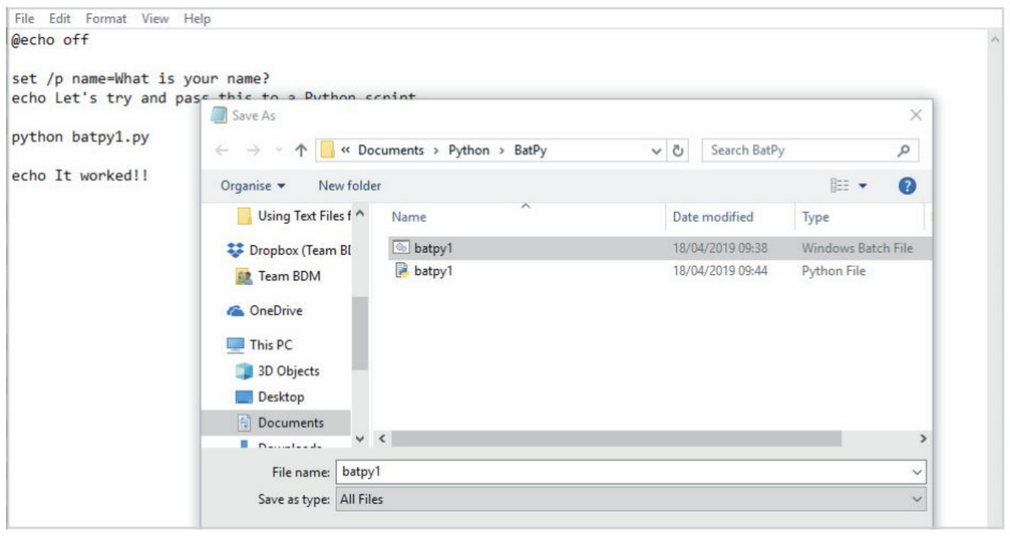
Let's begin by creating a sample folder within your Python directory; call it **batpy**, for example. Within the batpy folder, create a new text file and enter the following:

```
@echo off

set /p name=What is your name?
echo Let's try and pass this to a Python script...

python batpy1.py

echo It worked!!
```



STEP 2

We won't go into the intricacies of Windows batch files here. Suffice to say this code will ask for a user's name, store the name as a variable called name, display a message, then open the Python code we're about to create. We need to save the file as a batch file, click **File > Save As**, call it **batpy1.bat** and set **Save As Type** as **All Files**.



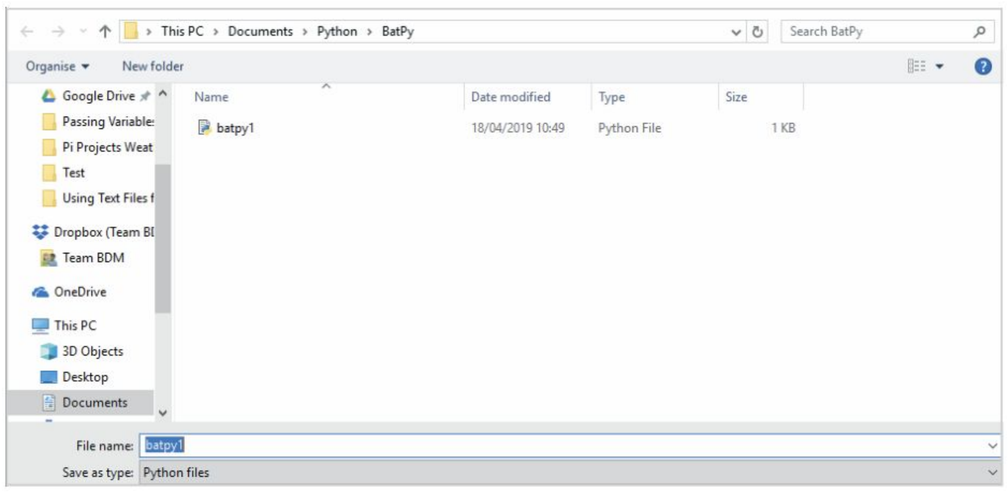
STEP 3

Now open Python, and create a New File. Enter the following code:

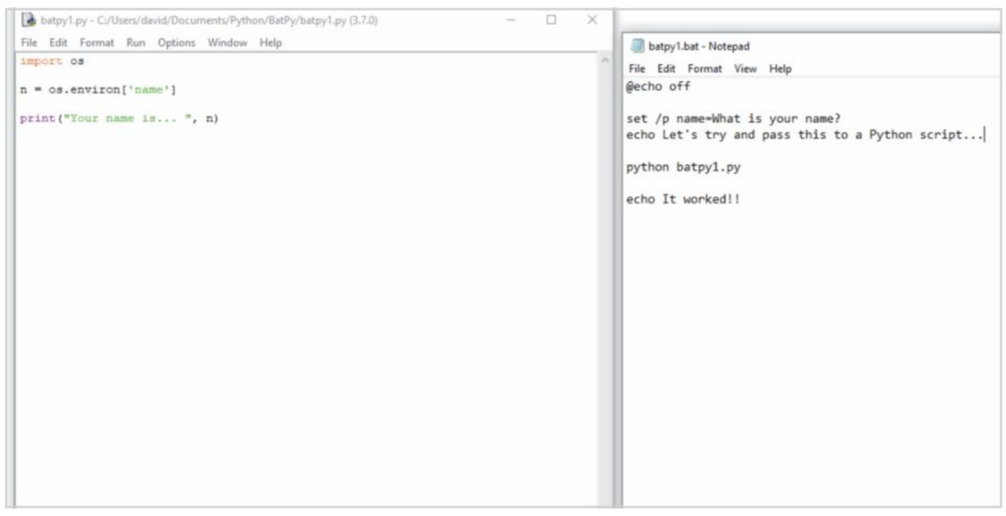
```
import os

n = os.environ['name']

print("Your name is... ", n)
```

STEP 4

Here we're using the OS module that will utilise the system variables. When we create a variable in a batch file (or Bash script), we're storing the contents of the variable as a system variable. In Python, we're passing **n** as the variable content of the command **os.environ**. Os.environ is calling the system variable 'name', which we created in the batch file. Save the code as **batpy1.py**.



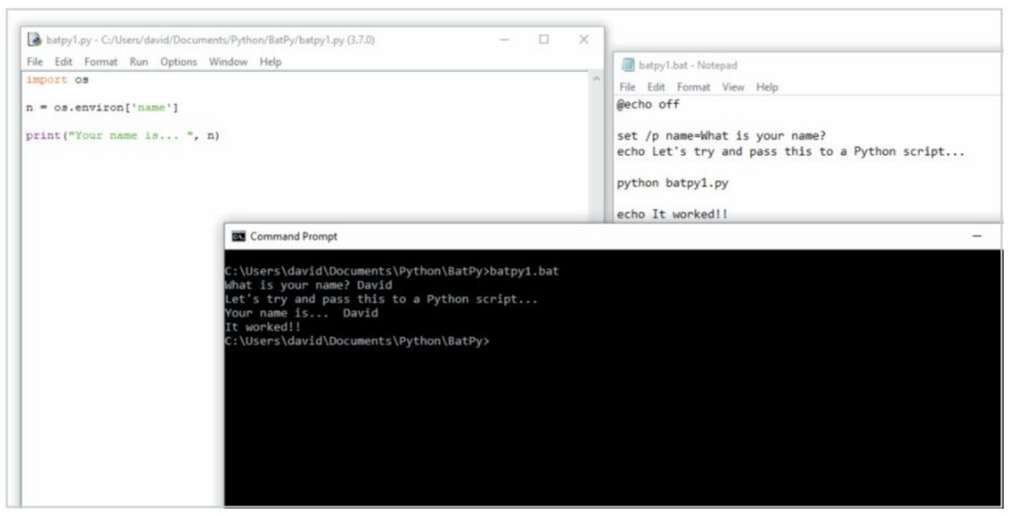
STEP 5 To recap: we have a batch file called **batpy1.bat**, which asks the user their name and stores the info as a system variable called **name**. It will then print a message to the screen, run the Python code, and finally print "It worked!!". The Python code, called **batpy1.py**, uses the OS module to call **os.environ['name']**, stored as n. It'll then print the value of n after a message.



STEP 8 Drop into a Windows command prompt, and navigate to the folder Batpy. To run a Windows batch file, simply enter the command:

batpy1.bat

Enter your name as instructed, and the reply will be passed to Python, then back to the batch file to end.

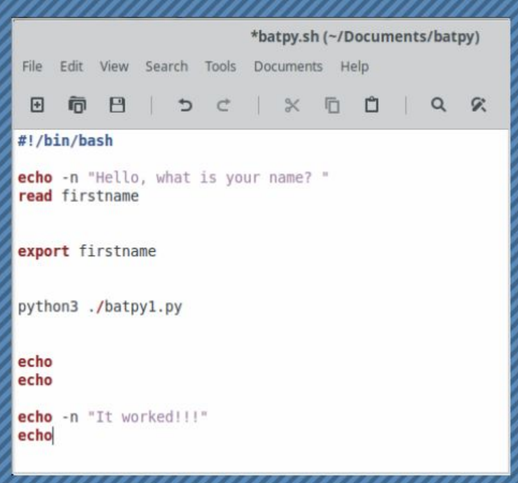


THE LINUX WAY

Linux's version of passing variables to Python is slightly different, just because it's Linux! It's easy, though, and here's how it's done.

STEP 1 Start by opening your favourite Linux text editor, and entering the following:

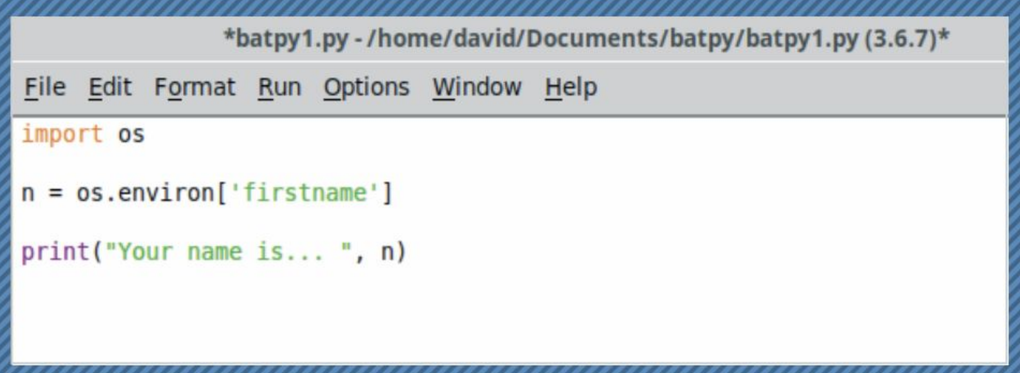
```
#!/bin/bash

echo -n "Hello, what is your name? "
read firstname

export firstname

python3 ./batpy1.py

echo
echo

echo -n "It worked!!!"
echo
```



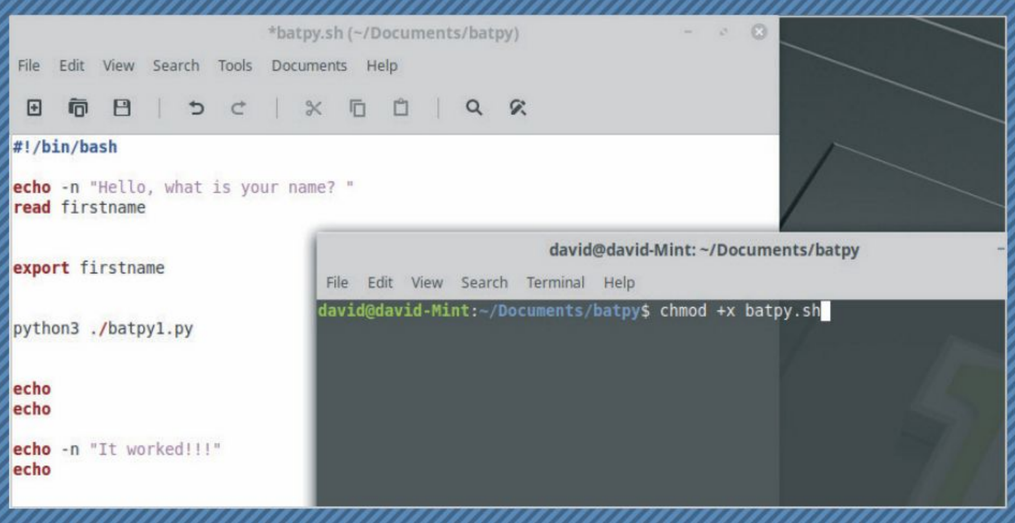
STEP 3 Now create a new Python file, and enter the following:

```
import os

n = os.environ['firstname']

print("Your name is... ", n)
```

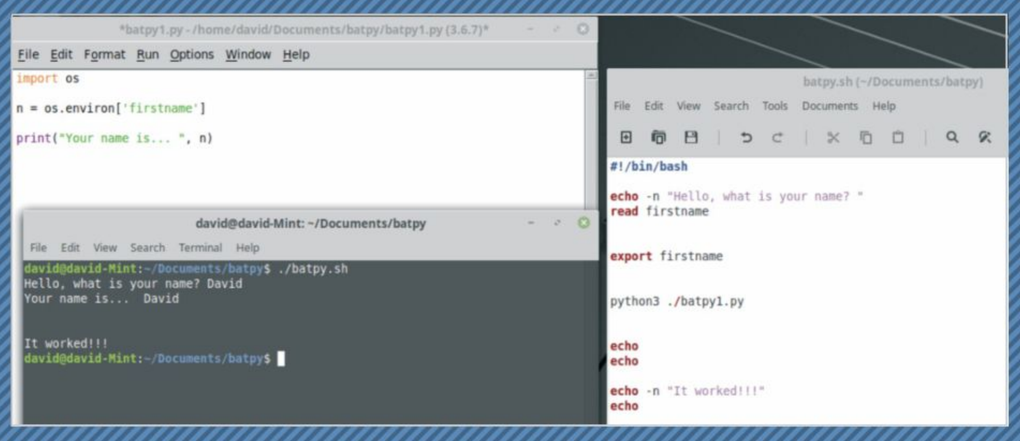
Save the Python code as **batpy1.py**.



STEP 2 Drop into a Terminal session and make a new directory called batpy (**mkdir batpy**). Save the Bash script as **batpy.sh**, and from the Terminal enable the script as executable with: `chmod +x batpy.sh`.



STEP 4 Back in the Terminal, enter: `./batpy.sh` to run the Bash script. As you can see, the results are the same as in Windows. The major differences are: making the Bash script executable, and adding the **export firstname** command to the Bash file. In Linux, you need to export system variables before they can be accessed by the subsystem, in this case the Python code.



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):
```

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.

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.

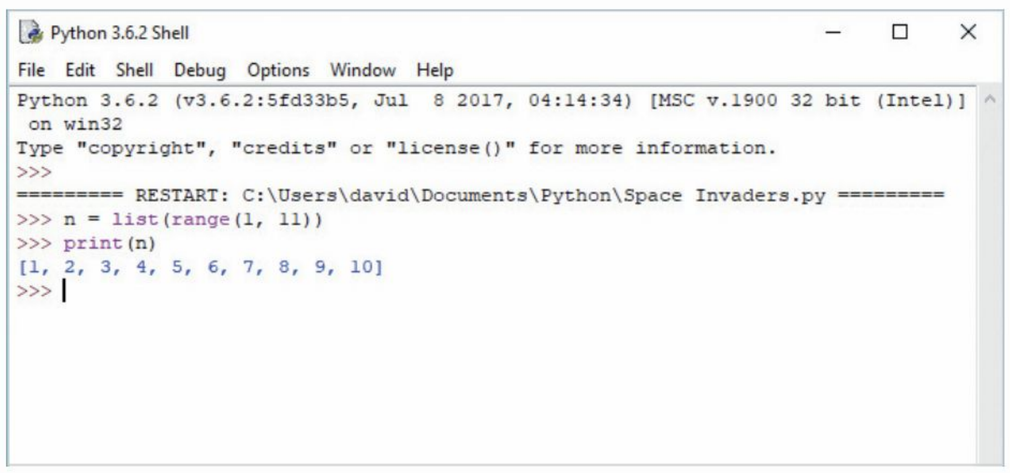
```
# set up pygame
pygame.init()
mainClock = pygame.time.Clock()

# 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
```

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:

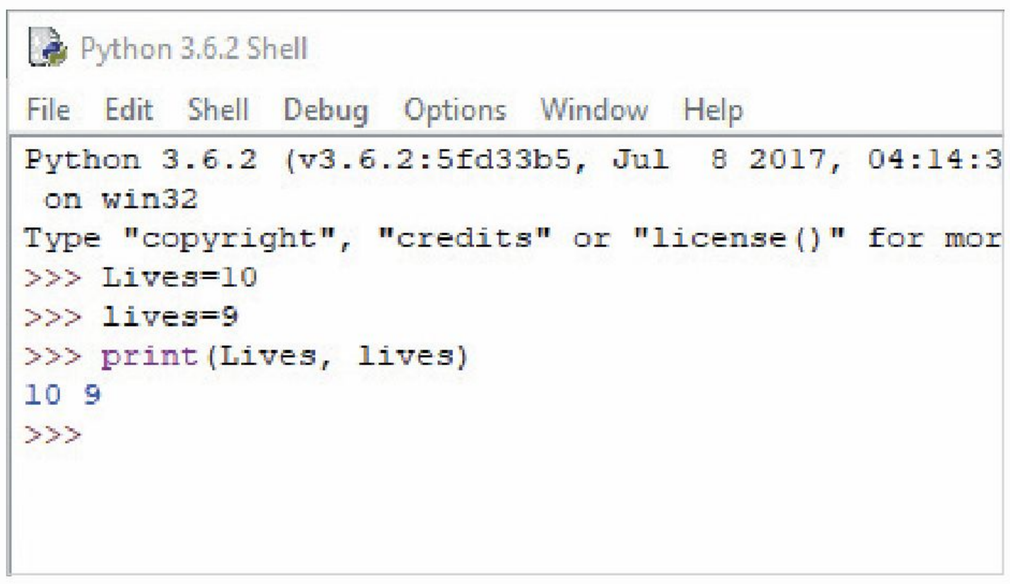
```
n = list(range(1, 11))
```

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')
```

Glossary of Python Terms

Just like most technology, Python contains many confusing words and acronyms. Here then, for your own sanity, is a handy glossary to help you keep on top of what's being said when the conversation turns to Python programming.

Argument
The detailed extra information used by Python to perform more detailed commands. Can also be used in the command prompt to specify a certain runtime event.

Block
Used to describe a section or sections of code that are grouped together.

Break
A command that can be used to exit a for or while loop. For example, if a key is pressed to quit the program, Break will exit the loop.

Class
A class provides a means of bundling data and functionality together. They are used to encapsulate variables and functions into a single entity.

Comments
A comment is a section of real world wording inserted by the programmer to help document what's going on in the code. They can be single line or multi-line and are defined by a # or '''.

Debian
A Linux-based distro or distribution that forms the Debian Project. This environment offers the user a friendly and stable GUI to interact with along with Terminal commands and other forms of system level administration.

Def
Used to define a function or method in Python.

Dictionaries
A dictionary in Python is a data structure that consists of key and value pairs.

Distro
Also Distribution, an operating system that uses the Linux Kernel as its core but offers something different in its presentation to the end user.

Editor
An individual program, or a part of the graphical version of Python, that enables the user to enter code ready for execution.

Exceptions
Used as a means of breaking from the normal flow of a code block in order to handle any potential errors or exceptional conditions within the program.

Expression
Essentially, Python code that produces a value of something.

Float
An immutable floating point number used in Python.

Function
Used in Python to define a sequence of statements that can be called or referenced at any time by the programmer.

GitHub
A web-based version control and collaboration portal designed for software developers to better manage source code.

Global Variable
A variable that is useable anywhere in the program.

Graphics
The use of visual interaction with a program, game or operating system. Designed to make it easier for the user to manage the program in question.

GUI
Graphical User Interface. The interface which most modern operating systems use to enable the user to interact with the core programming of the system. A friendly, easy to use graphical desktop environment.

High-Level Language
A programming language that's designed to be easy for people to read.

IDLE
Stands for Integrated Development Environment or Integrated Development and Learning Environment.

Immutable
Something that cannot be changed after it is created.

Import
Used in Python to include modules together with all the accompanying code, functions and variables they contain.

Indentation
Python uses indentation to delimit blocks of code. The indents are four spaces apart, and are often created automatically after a colon is used in the code.

Integer
A number data type that must be a whole number and not a decimal.

Interactive Shell
The Python Shell, which is displayed whenever you launch the graphical version of Python.

Kernel
The core of an operating system, which handles data processing, memory allocation, input and output, and processes information between the hardware and programs.

Linux
An open source operating system that's modelled on UNIX. Developed in 1991 by Finnish student Linus Torvalds.

Lists
A Python data type that contains collections of values, which can be of any type and can readily be modified.

Local Variable
A variable that's defined inside a function and is only useable inside that function.

Loop
A piece of code that repeats itself until a certain condition is met. Loops can encase the entire code or just sections of it.

Module
A Python file that contains various functions that can be used within another program to further extend the effectiveness of the code.

Operating System
Also OS. The program that's loaded into the computer after the initial boot sequence has completed. The OS manages all the other programs, graphical user interface (GUI), input and output and physical hardware interactions with the user.

Output
Data that is sent from the program to a screen, printer or other external peripheral.

PIP
Pip Installs Packages. A package management system used to install and manage modules and other software written in Python.

Print
A function used to display the output of something to the screen.

Prompt
The element of Python, or the Command Line, where the user enters their commands. In Python it's represented as >>> in the interactive shell.

Pygame
A Python module that's designed for writing games. It includes graphics and sound libraries and was first developed in October 2000.

Python
An awesome programming language that's easy to learn and use, whilst still being powerful enough to enjoy.

Random
A Python module that implements a pseudo-random character generator using the Mersenne Twister PRNG.

Range
A function that used to return a list of integers, defined by the arguments passed through it.

Root
The bottom level user account used by the system itself. Root is the overall system administrator and can go anywhere, and do anything, on the system.

Sets
Sets are a collection of unordered but unique data types.

Strings
Strings can store characters that can be modified. The contents of a string are alphanumerical and can be enclosed by either single or double quote marks.

Terminal
Also Console or Shell. The command line interface to the operating system, namely Linux, but also available in macOS. From there you can execute code and navigate the filesystem.

Tkinter
A Python module designed to interact with the graphical environment, specifically the tk-GUI (Tool Kit Graphical User Interface).

Try
A try block allows exceptions to be raised, so any errors can be caught and handled according to the programmer's instructions.

Tuples
An immutable Python data type that contains an ordered set of either letters or numbers.

UNIX
A multitasking, multiuser operating system designed in the '70s at the Bell Labs Research Centre. Written in C and assembly language

Variables
A data item that has been assigned a storage location in the computer's memory.

X
Also X11 or X-windows. The graphical desktop used in Linux-based systems, combining visual enhancements and tools to manage the core operating system.

Zen of Python
When you enter: `import this` into the IDLE, the Zen of Python is displayed.

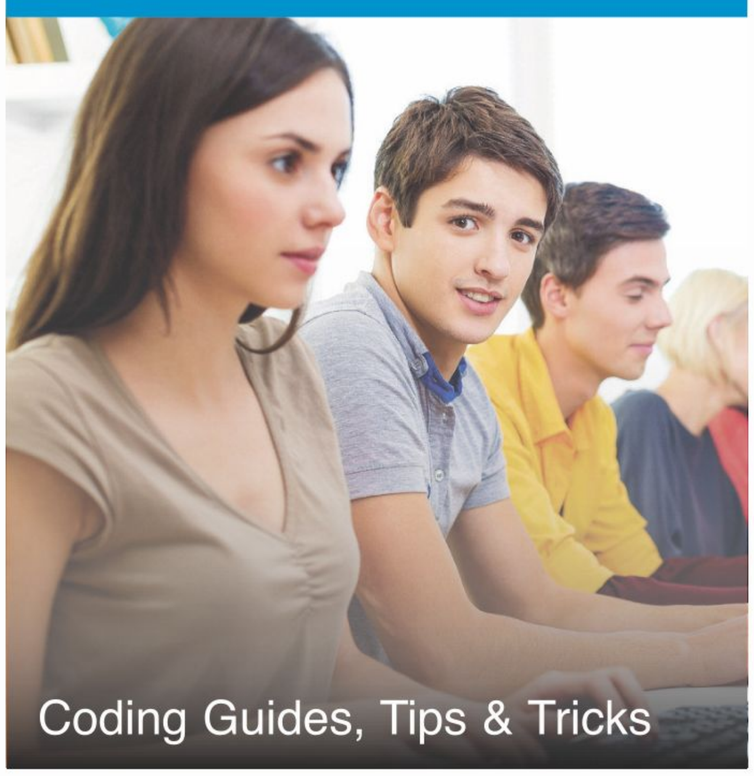
Black Dog Media

Master Your Tech From Beginner to Expert

To continue learning more about coding visit us at:

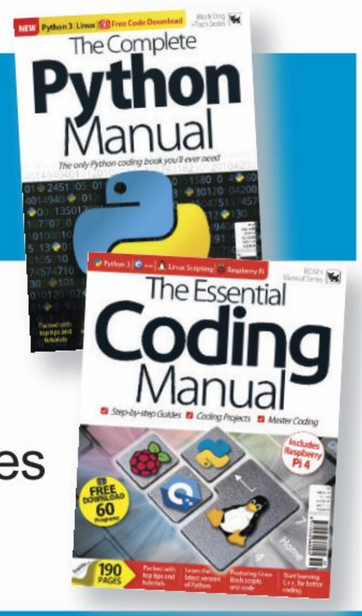
www.bdmpublications.com

FREE Tech Guides



Exclusive Offers on Tech Guidebooks

- Print & digital editions
- Featuring the latest updates
- Step-by-step tutorials & guides
- Created by BDM experts



ultimatephotoshop.com
Buy our Photoshop guides and download tutorial images for free!
Simply sign up and get creative.

PLUS Special Deals and Bonus Content

When you sign up to our monthly newsletter!

BDM's Manual Series – The Python Coding Manual
Volume 20 – ISSN 2056-6662
Published by: .....Black Dog Media Limited (BDM)
Editor: .....James Gale
Art Director & Production: ...Mark Ayshford
Production Manager: .....Karl Linstead
Design: .....Robin Drew, Lena Whitaker
Editorial:.....David Hayward
Sub Editor:.....Alison Drew

Printed and bound in Great Britain by: Acorn Web Offset Ltd
Newsstand distribution by: Seymour Distribution Limited
2, East Poultry Avenue, London EC1A 9PT
International distribution by: Pineapple Media Limited
www.pineapple-media.com
For all advertising and promotional opportunities contact:
enquiries@bdmpublications.com

Copyright © 2019 Black Dog Media. All rights reserved.

INTERNATIONAL LICENSING – Black Dog Media has many great publications and all are available for licensing worldwide. For more information go to: www.brucesawfordlicensing.com; email: bruce@brucesawfordlicensing.com; telephone: 0044 7831 567372

Editorial and design are the copyright © Papercut Limited and are reproduced under licence to Black Dog Media. 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, Black Dog Media 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 software industry, the publisher cannot guarantee that all tutorials will work on every version of Windows, macOS, or the Raspbian OS. It remains the purchaser's sole responsibility to determine the suitability of this book and its content for whatever purpose. 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.

BDM's Manual Series – The Python Coding Manual vol.20 is an independent publication and as such does not necessarily reflect the views or opinions of the producers contained within. This publication is not endorsed or associated in any way with Microsoft, The Linux Foundation, The Raspberry Pi Foundation, ARM Holding, Canonical Ltd, Python, Debian Project, Lenovo, Dell, Hewlett-Packard, Apple and Samsung or any associate or affiliate company. All copyrights, trademarks and registered trademarks for the respective companies are acknowledged. Relevant graphic imagery reproduced with courtesy of Lenovo, Hewlett- Packard, Dell, Samsung, Microsoft and Apple.

Additional images contained within this publication are reproduced under licence from Shutterstock.com.

Prices, international availability, ratings, titles and content are subject to change. All information was correct at time of print. Some content may have been previously published in other volumes or BDM titles. We advise potential buyers to check the suitability of contents prior to purchase.

Black Dog Media Limited (BDM)
Registered in England & Wales No: 5311511