



Coding & Programming

Linux

Tricks and Tips

Advanced
Tutorials
& Guides

Master
Your
Code!

Expert
Pro Tips &
Hacks

Next Level
Secrets
& Fixes

Over **430** Secrets & Hacks

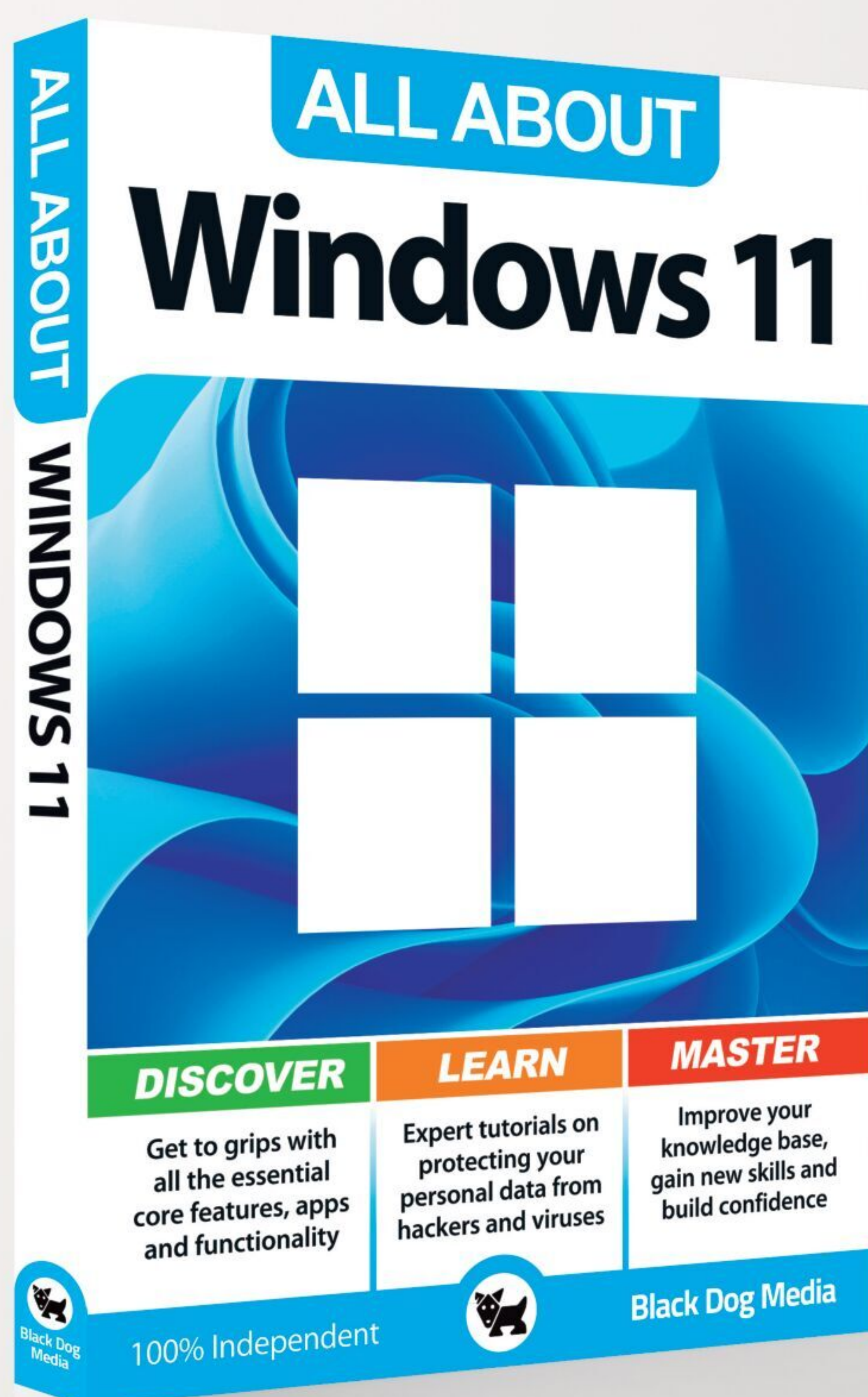
Discover Windows 11. Learn New Features. Master Your Desktop.



Black Dog Media

www.bdmpublications.com

NEW
PRINT
EDITION



**Over 250 Pages of
Expert Tutorials,
Guides & Tips!**

FEATURED INSIDE:

- Navigating the Start Menu
- Connecting to the Internet
- How to personalise Windows 11
- Web browsing with Edge
- Using OneDrive Cloud Storage
- Video chatting with Skype
- Email, Social Media & Messaging
- Improving Windows 11 security
- Wi-fi and Personal wi-fi hotspots
- Speeding up your computer
- Troubleshooting & User Advice and much more...

Click the link to **BUY NOW** from **amazon**

<https://amzn.to/3EZFHKC>

Available in all good bookshops



Coding & Programming

Linux

Tricks and Tips

Welcome to the next level of understanding and expansion of your user experience!

For some it is enough to master the basics of your new device, software or hobby. Yet for many, like you, that's just the start! Advancing your skill set is the goal of all users of consumer technology and our team of long term industry experts will help you achieve exactly that. Over this extensive series of titles we will be looking in greater depth at how you make the absolute most from the latest consumer electronics, software, hobbies and trends! We will guide you step-by-step through using all the advanced aspects of the technology that you may have been previously apprehensive at attempting. Let our expert guide help you build your understanding of technology and gain the skills to take you from a confident user to an experienced expert.

*Over the page
our journey continues,
and we will be with you
at every stage to advise,
inform and ultimately
inspire you to
go further.*



Contents

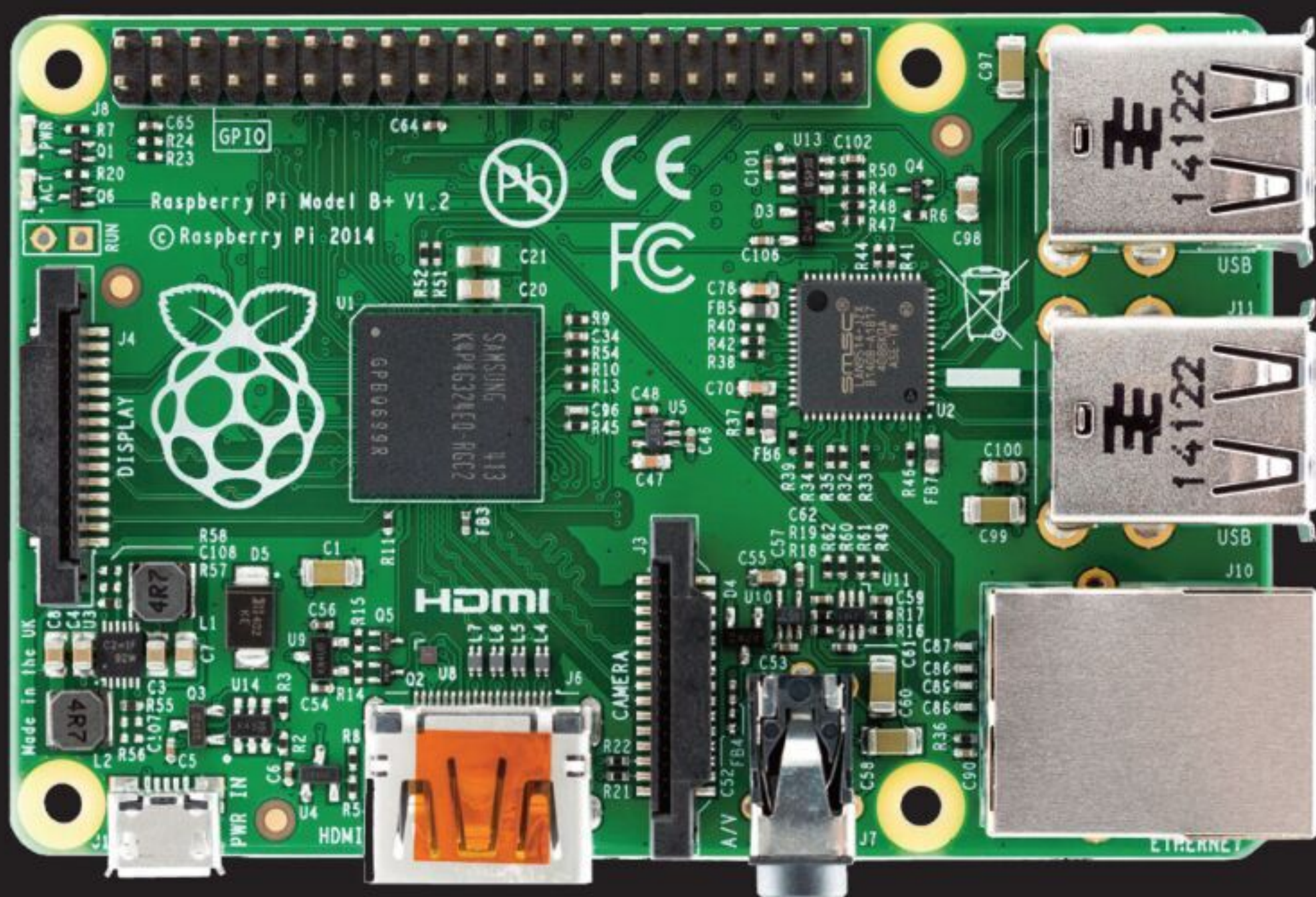
BDM's Code Portal
60+ Python programs
21,500+ lines of code
 Master Python with the help of our fantastic Code Portal, featuring code for games, tools and more.
 Visit: <https://bdmpublications.com/code-portal>, and log in to get access!

6 Python on Linux

- 8 Why Python?
- 10 How to Set Up Python in Linux
- 12 Starting Python for the First Time
- 14 Your First Code
- 16 Saving and Executing Your Code
- 18 Executing Code from the Terminal
- 20 Did you Know...Space Invaders
- 22 Numbers and Expressions
- 24 Using Comments
- 26 Working with Variables
- 28 User Input
- 30 Creating Functions
- 32 Conditions and Loops
- 34 Python Modules
- 36 Did you Know...Debugging

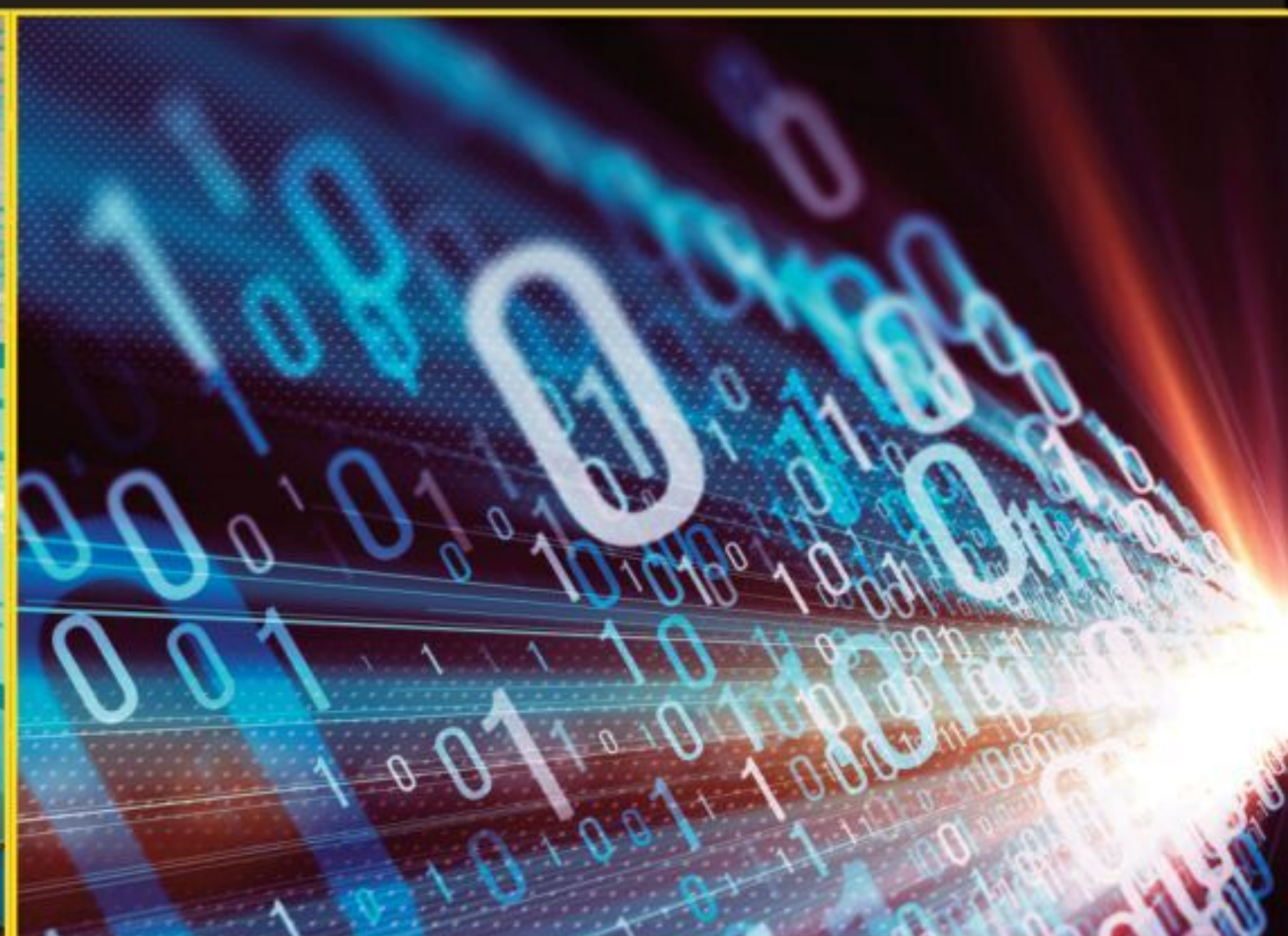
38 C++ on Linux

- 40 Why C++?
- 42 Your First C++ Program
- 44 Structure of a C++ Program
- 46 Compile and Execute
- 48 Did you Know...Virus!
- 50 Using Comments
- 52 Variables
- 54 Data Types
- 56 Strings
- 58 C++ Maths
- 60 User Interaction
- 62 Did you Know...The Hobbit
- 64 Common Coding Mistakes





“There’s more to Linux than simply being a free to use operating system. Its unique configuration allows the user to customise and personalise the OS into any form they wish. A Linux user can change their OS look and feel from one day to the next, install thousands of freely available apps and programs and take back control of their computer.”





“Python is an experiment in how much freedom programmers need. Too much freedom and nobody can read another’s code; too little and expressiveness is endangered.”

– Guido van Rossum (Developer of the Python programming language)



Python on Linux

Python is a fantastic programming language, combining ease of use with a generous helping of power to allow the user to create both minor utilities or performance-heavy computational tasks.

However, there's more to Python than simply being another programming language. It has a vibrant and lively community behind it that shares knowledge, code and project ideas, as well as bug fixes for future releases.

We've used a Raspberry Pi for this section of the book, as it's Linux based and one of the best coding platforms available. The Pi's features and functions make it the perfect Python programming partner, so let's get started.



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.

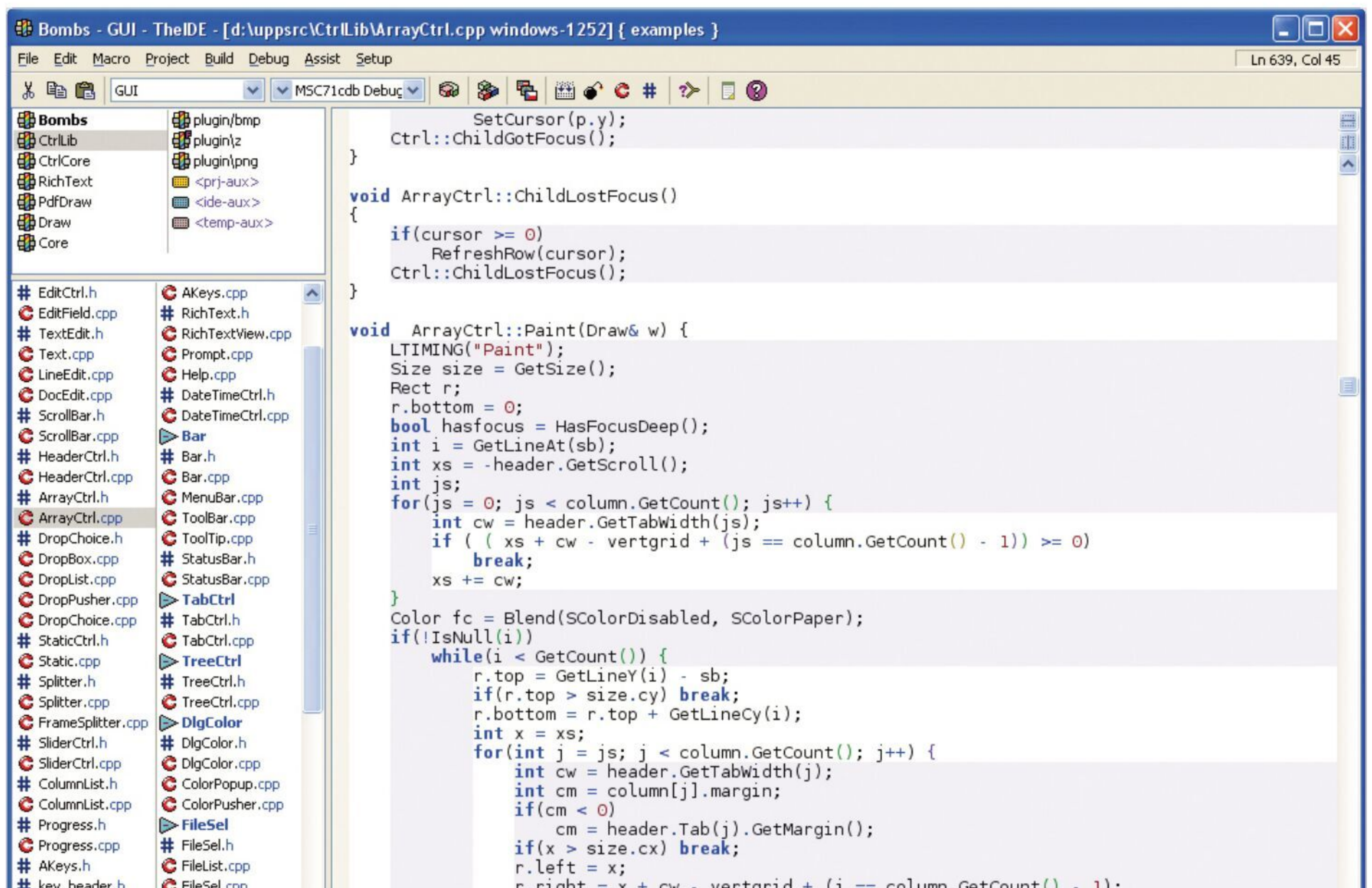
These pioneers of programming carved their way into a new frontier, forging small routines that enabled the letter 'A' to scroll across the screen. It may not sound terribly exciting to a generation that's used to ultra high-definition graphics and open world, multi-player online gaming. However, forty-something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

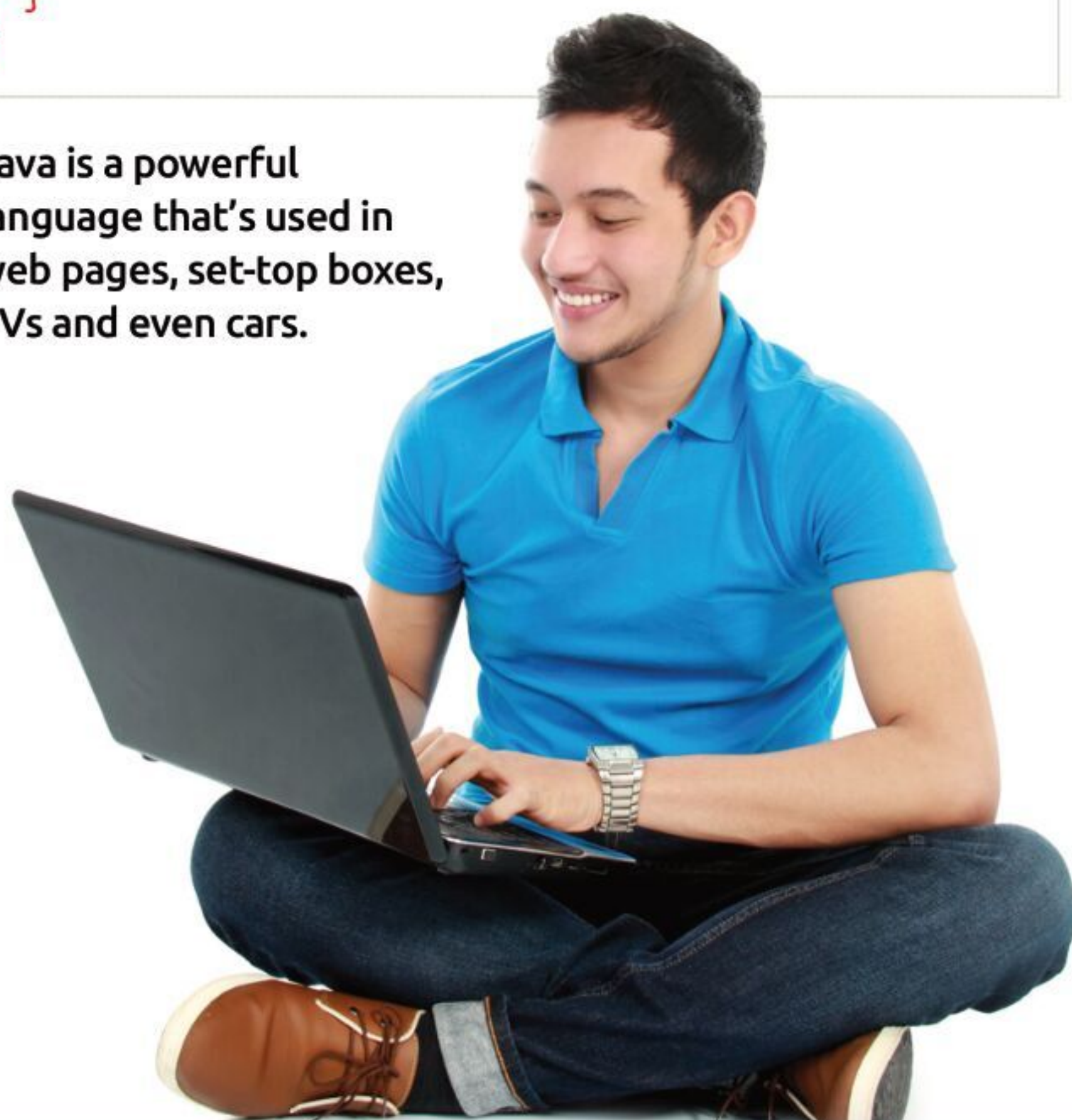
Python is what's known as a High-Level Language, in that it 'talks' to the hardware and operating system using a variety of arrays, variables, objects, arithmetic, subroutines, loops and countless more interactions. Whilst it's not as streamlined as a Low-Level Language, which can deal directly with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

```

1 //file: Invoke.java
2 import java.lang.reflect.*;
3
4 class Invoke {
5     public static void main( string [] args ) {
6         try {
7             Class c = Class.forName( args[0] );
8             Method m = c.getMethod( args[1], new Class
9                 [] { } );
10            Object ret = m.invoke( null, null );
11            System.out.println(
12                "Invoked static method: " + args[1]
13                + " of class: " + args[0]
14                + " with no args\nResults: " + ret );
15        } catch ( ClassNotFoundException e ) {
16            // Class.forName( ) can't find the class
17        } catch ( NoSuchMethodException e2 ) {
18            // that method doesn't exist
19        } catch ( IllegalAccessException e3 ) {
20            // we don't have permission to invoke that
21            // method
22        } catch ( InvocationTargetException e4 ) {
23            // an exception occurred while invoking that
24            // method
25            System.out.println(
26                "Method threw an: " + e4.
                getTargetException( ) );
27        }
28    }
29 }

```

 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
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.



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 preinstalled. 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 we're sticking with Linux Mint for this particular tutorial.

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.

```
david@david-Mint: ~  
File Edit View Search Terminal Help  
david@david-Mint:~$
```

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

```
david@david-Mint: ~  
File Edit View Search Terminal Help  
david@david-Mint:~$ python --version  
Python 2.7.15rc1  
david@david-Mint:~$ python3 --version  
Python 3.6.7  
david@david-Mint:~$
```

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, so enter: `sudo apt-get update && sudo apt-get upgrade` to update the system.

```
david@david-Mint: ~  
File Edit View Search Terminal Help  
david@david-Mint:~$ python --version  
Python 2.7.15rc1  
david@david-Mint:~$ python3 --version  
Python 3.6.7  
david@david-Mint:~$ sudo apt-get update && sudo apt-get upgrade  
[sudo] password for david:
```

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 and further updates. Often they're unnecessary but can contain vital new elements.

```
4 1.1.3-Subuntu0.2 [359 kB]  
Get:2 http://archive.ubuntu.com/ubuntu bionic-updates/main amd64 libasound2-data  
all 1.1.3-5ubuntu0.2 [36.5 kB]  
Get:3 http://archive.ubuntu.com/ubuntu bionic-updates/main amd64 linux-libc-dev  
amd64 4.15.0-44.47 [1,013 kB]  
Fetched 1,409 kB in 0s (3,023 kB/s)  
(Reading database ... 290768 files and directories currently installed.)  
Preparing to unpack .../libasound2_1.1.3-5ubuntu0.2_amd64.deb ...  
Unpacking libasound2:amd64 (1.1.3-5ubuntu0.2) over (1.1.3-5ubuntu0.1) ...  
Preparing to unpack .../libasound2-data_1.1.3-5ubuntu0.2_all.deb ...  
Unpacking libasound2-data (1.1.3-5ubuntu0.2) over (1.1.3-5ubuntu0.1) ...  
Preparing to unpack .../linux-libc-dev_4.15.0-44.47_amd64.deb ...  
Unpacking linux-libc-dev:amd64 (4.15.0-44.47) over (4.15.0-43.46) ...  
Setting up libasound2-data (1.1.3-5ubuntu0.2) ...  
Setting up linux-libc-dev:amd64 (4.15.0-44.47) ...  
Setting up libasound2:amd64 (1.1.3-5ubuntu0.2) ...  
Processing triggers for libc-bin (2.27-3ubuntu1) ...  
david@david-Mint:~$ python3 --version  
Python 3.6.7  
david@david-Mint:~$
```

**STEP 5**

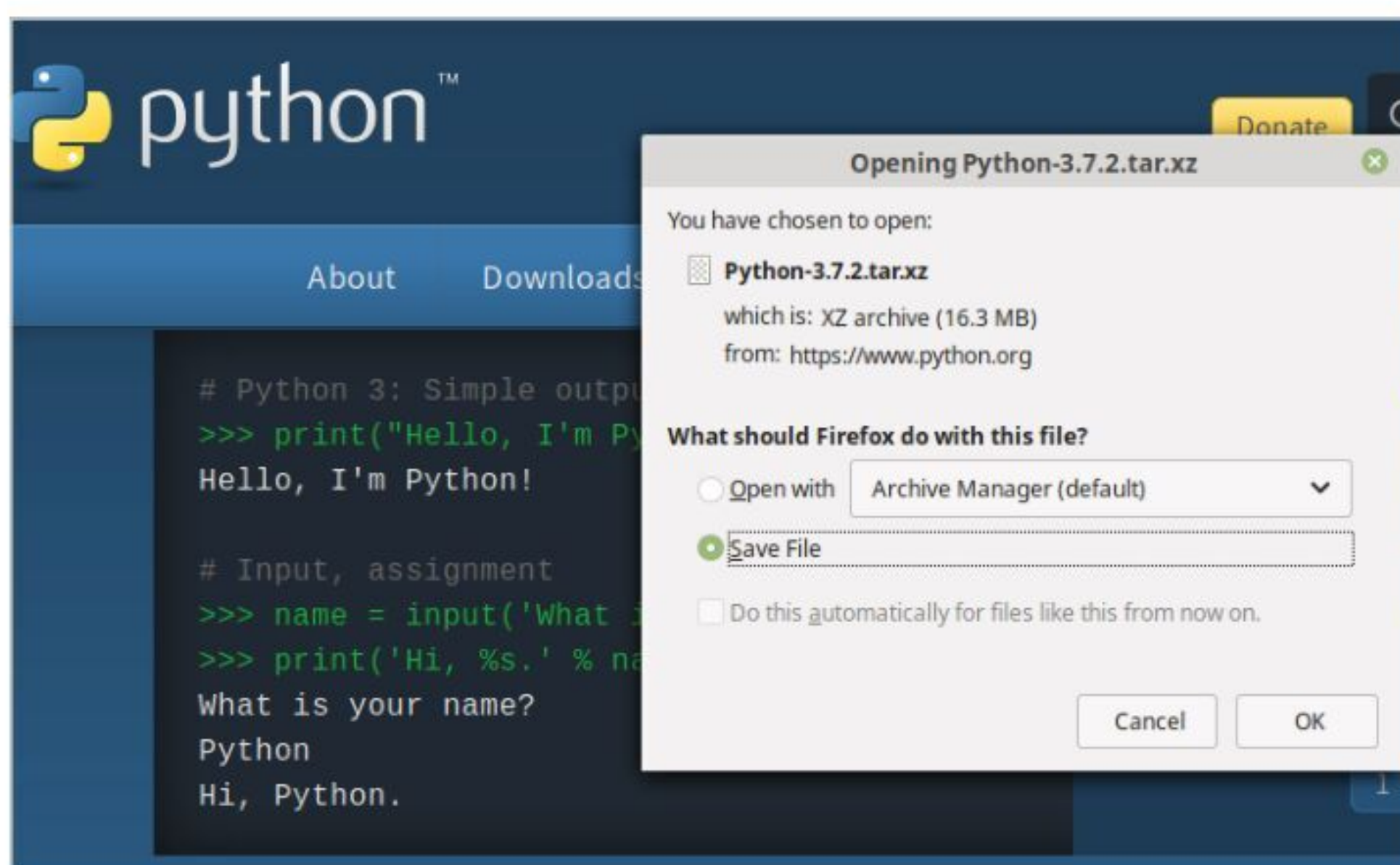
However, if you want the latest, cutting edge version, you need to build Python from source. Start by entering these commands into the Terminal:

```
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
```

```
david@david-Mint: ~
File Edit View Search Terminal Help
david@david-Mint:~$ sudo apt-get install build-essential checkinstall
Reading package lists... Done
Building dependency tree
Reading state information... Done
build-essential is already the newest version (12.4ubuntu1).
The following NEW packages will be installed
  checkinstall
0 to upgrade, 1 to newly install, 0 to remove and 3 not to upgrade.
Need to get 97.1 kB of archives.
After this operation, 438 kB of additional disk space will be used.
Do you want to continue? [Y/n] y
```

STEP 6

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

**STEP 7**

In the Terminal, go 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` replacing 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/bytestrarrayobject.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 installed latest 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 need to enter the following command into the Terminal:

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

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

```
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 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`

When 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]
```



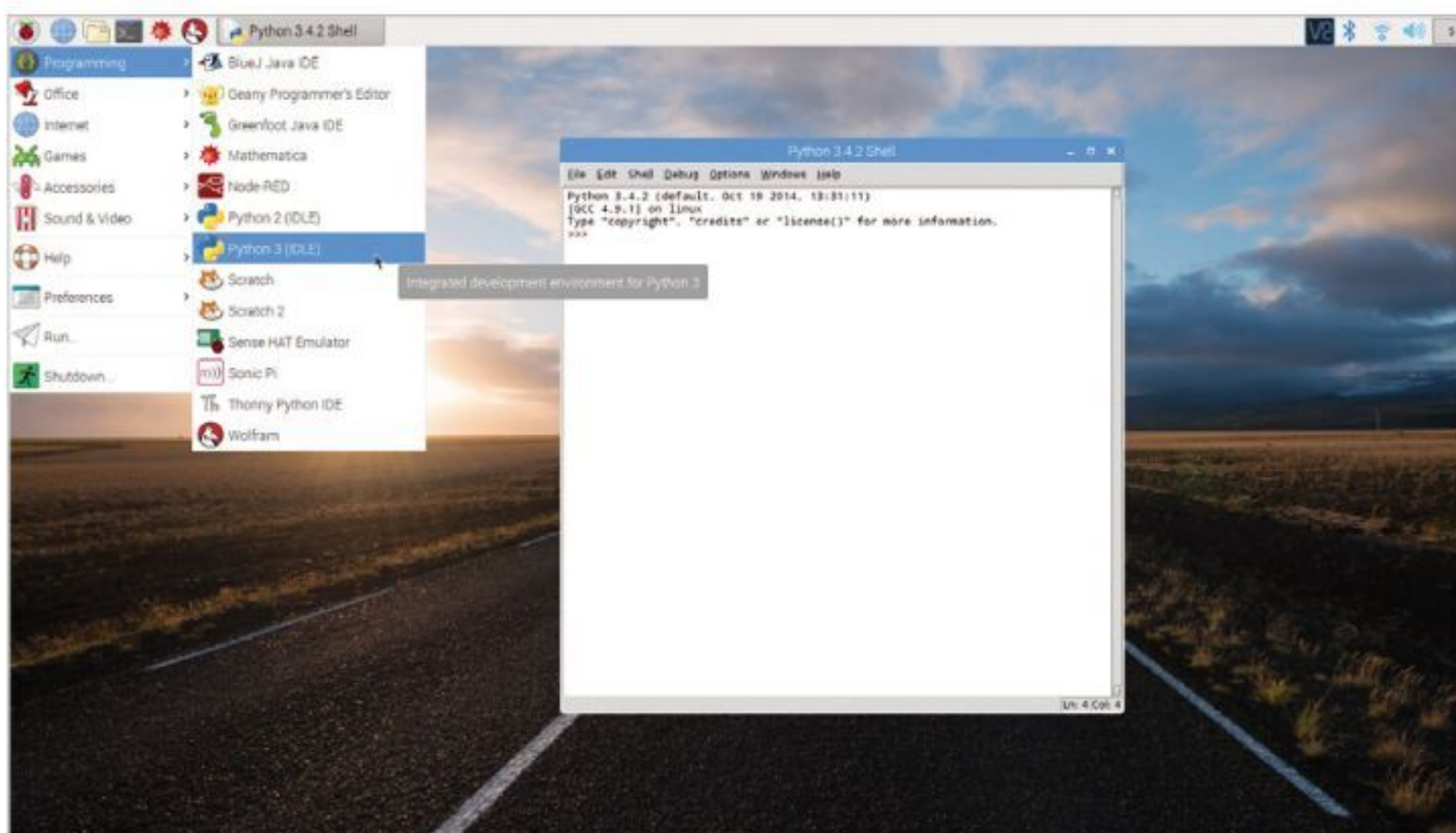
Starting Python for the First Time

If you're using the new Raspberry Pi together with its latest release of the Pi OS, then you will need to manually install the Python IDLE. This is due to the Pi team removing the core Python IDLE in favour of their own coding text editor.

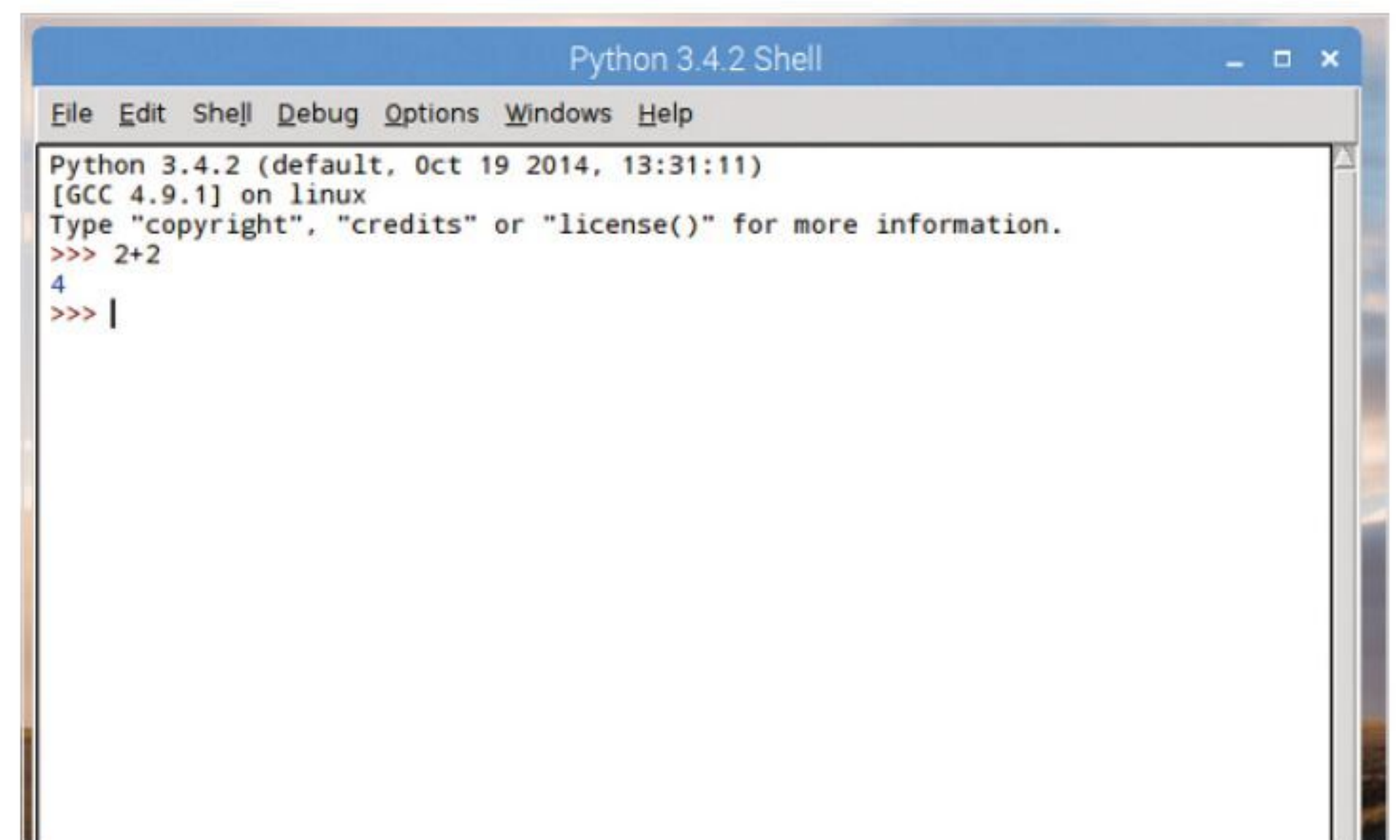
STARTING PYTHON

For those using the Pi4 and new Raspbian, drop into a Terminal and enter: `sudo apt-get install idle3`. Older versions of Raspbian already have the official Python IDLE pre-installed.

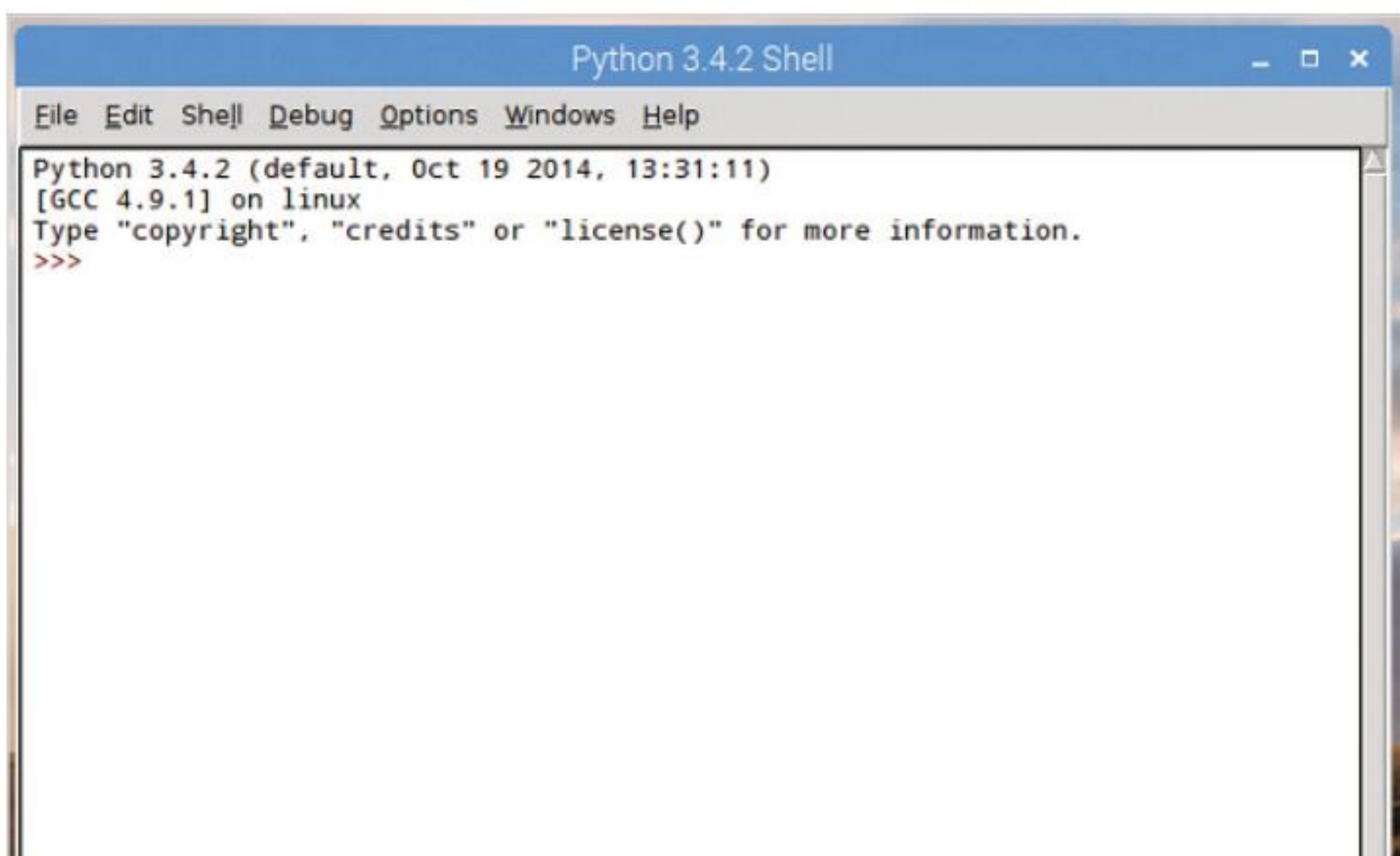
STEP 1 With the Raspbian desktop loaded, click on the Menu button followed by Programming > Python 3 (IDLE). This opens the Python 3 Shell. Windows and Mac users can find the Python 3 IDLE Shell from within the Windows Start button menu and via Finder.



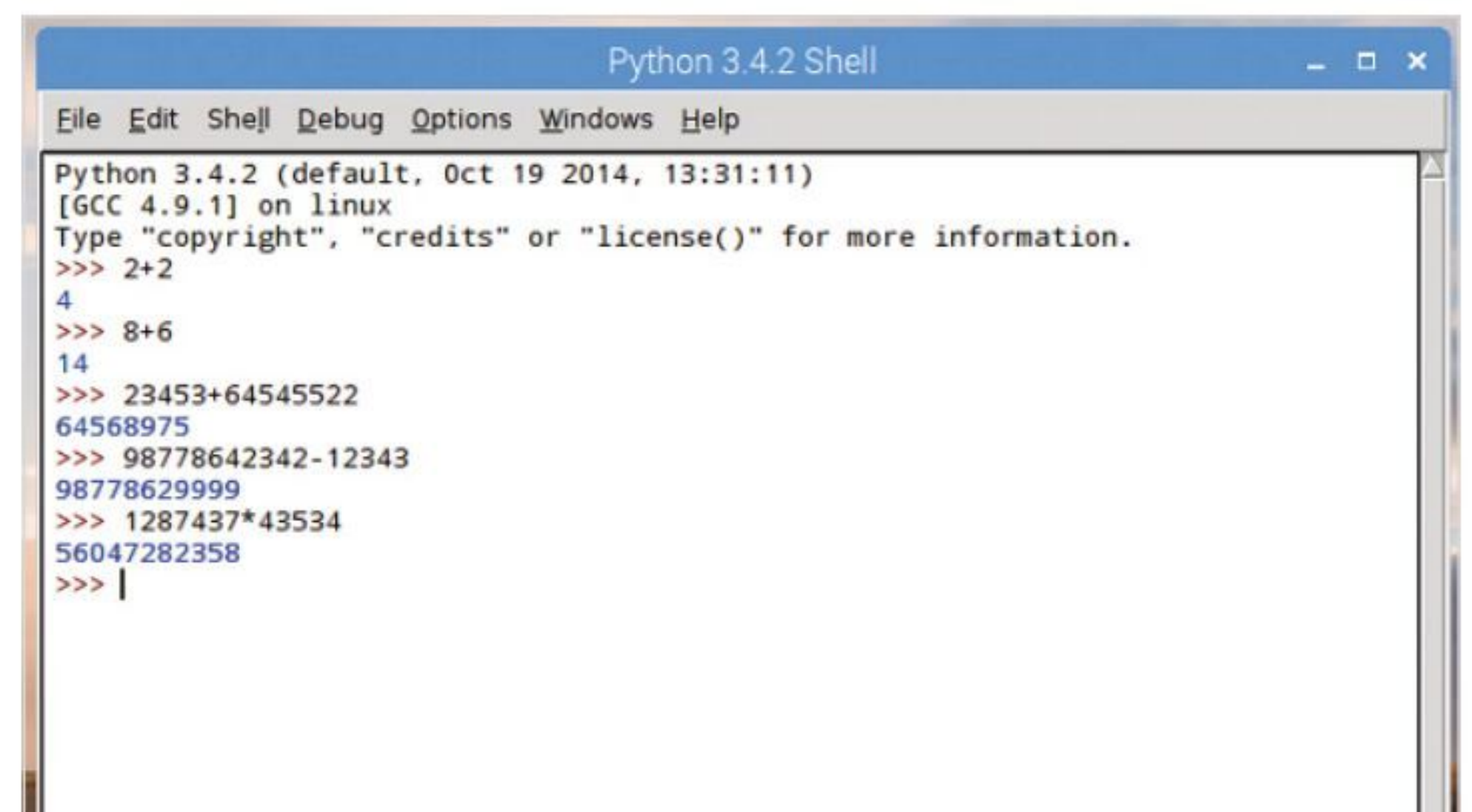
STEP 3 For example, in the Shell enter: `2+2`. After pressing Enter, the next line displays the answer: 4. Basically, Python has taken the 'code' and produced the relevant output.



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



STEP 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.



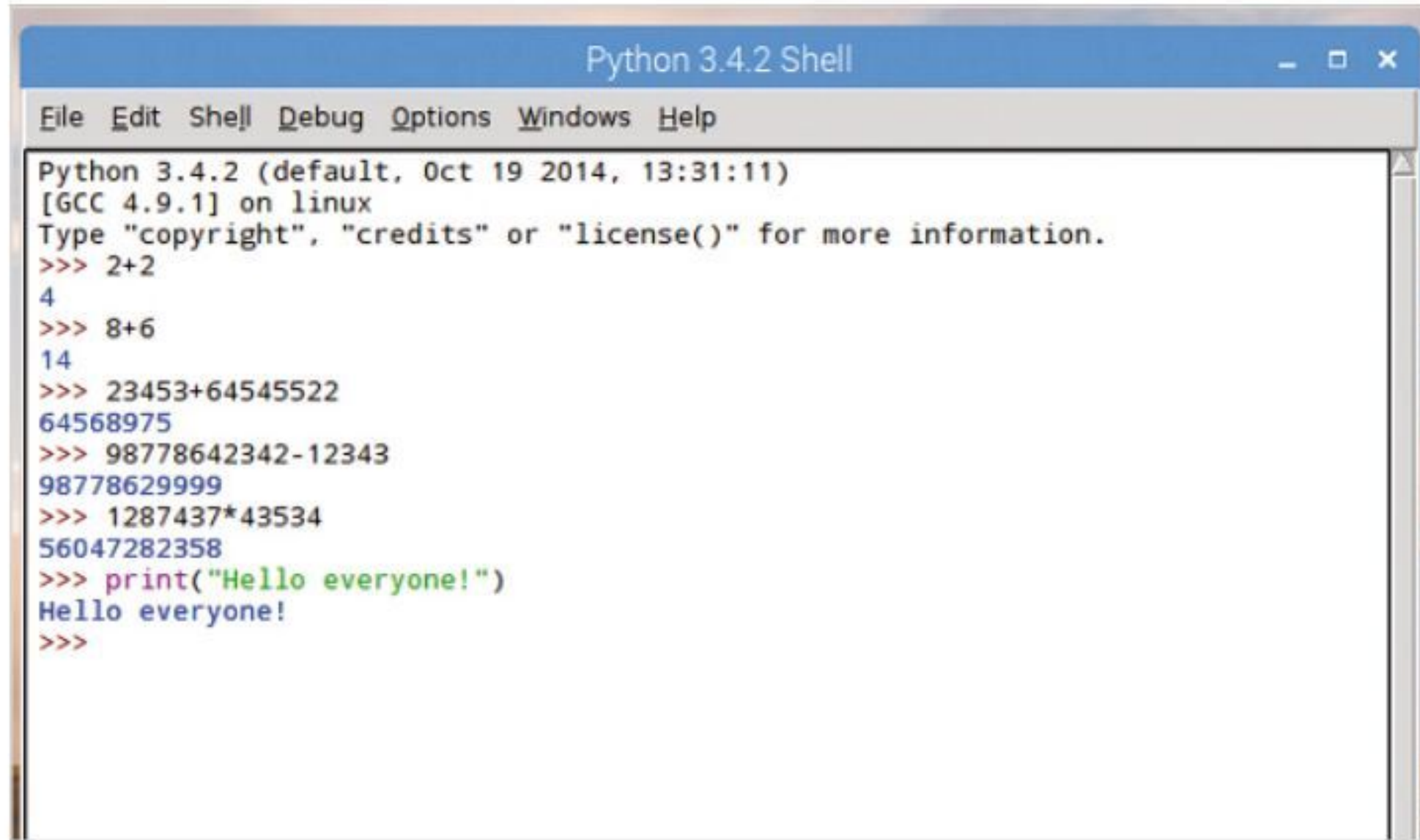


STEP 5

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 brackets as well as quote marks in order to output content to the screen, in this case the 'Hello everyone!' bit.

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

STEP 7

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

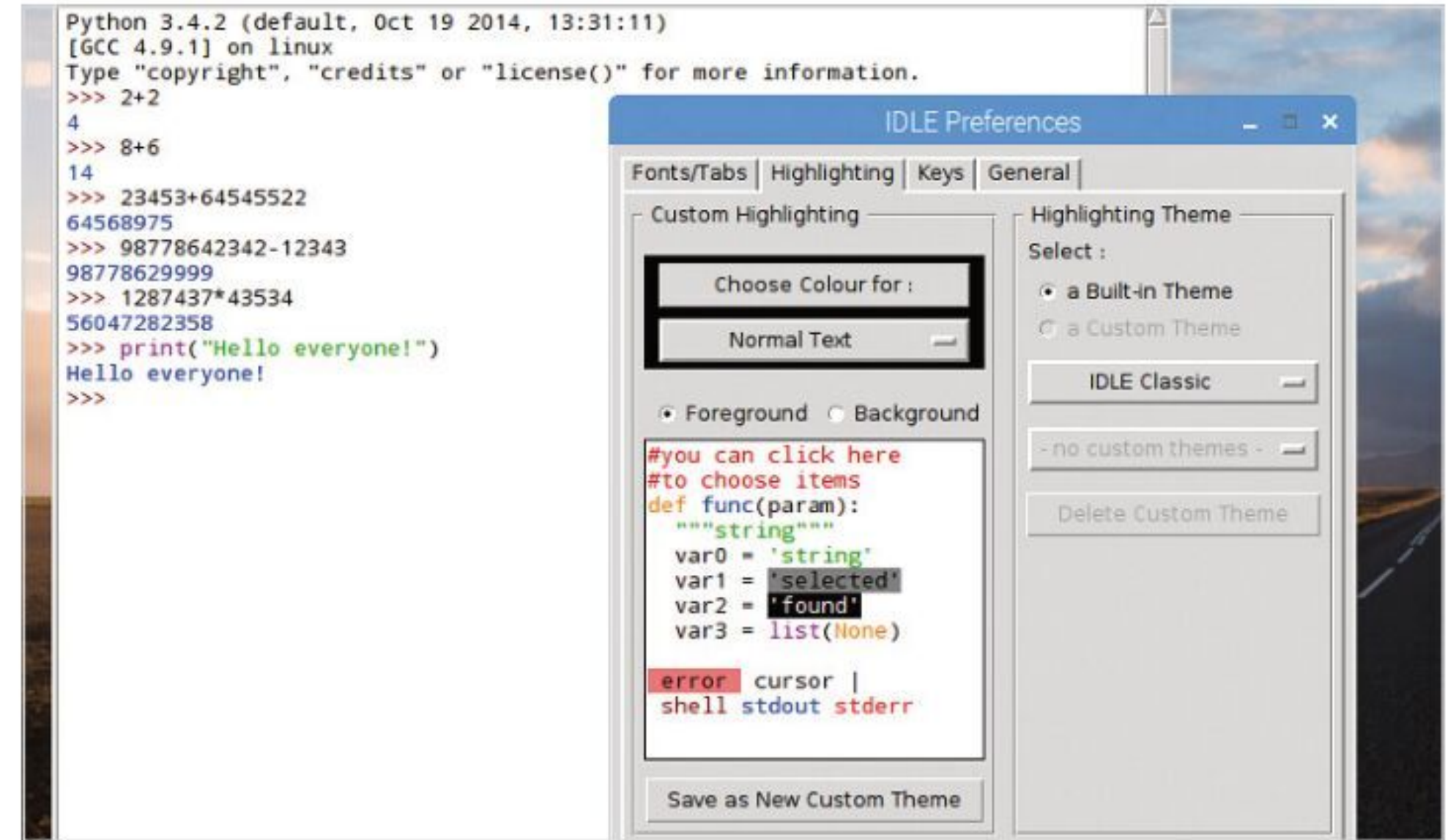
- Black – Data and Variables
- Green – Strings
- Purple – Functions
- Orange – Commands
- Blue – User Functions
- Dark Red – Comments
- Light Red – Error Messages

IDLE Colour Coding

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

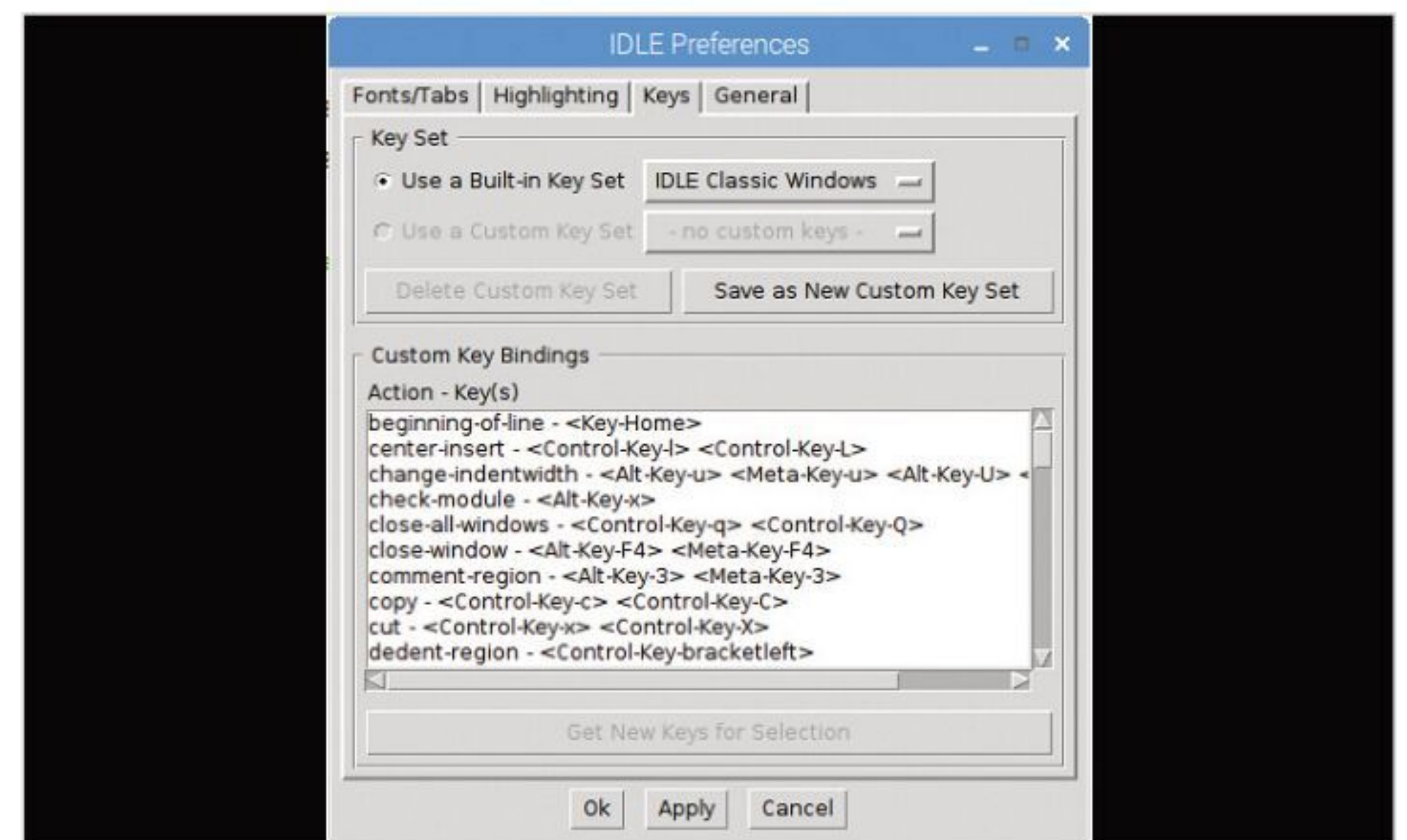
STEP 8

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



STEP 9

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



STEP 10

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

25.5. IDLE

Source code: [Lib/idlelib](#)

IDLE is Python's Integrated Development and Learning Environment.

IDLE has the following features:

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

25.5.1. Menus

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

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

25.5.1.1. File menu (Shell and Editor)

New File
Create a new file editing window.

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

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

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

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



Your First Code

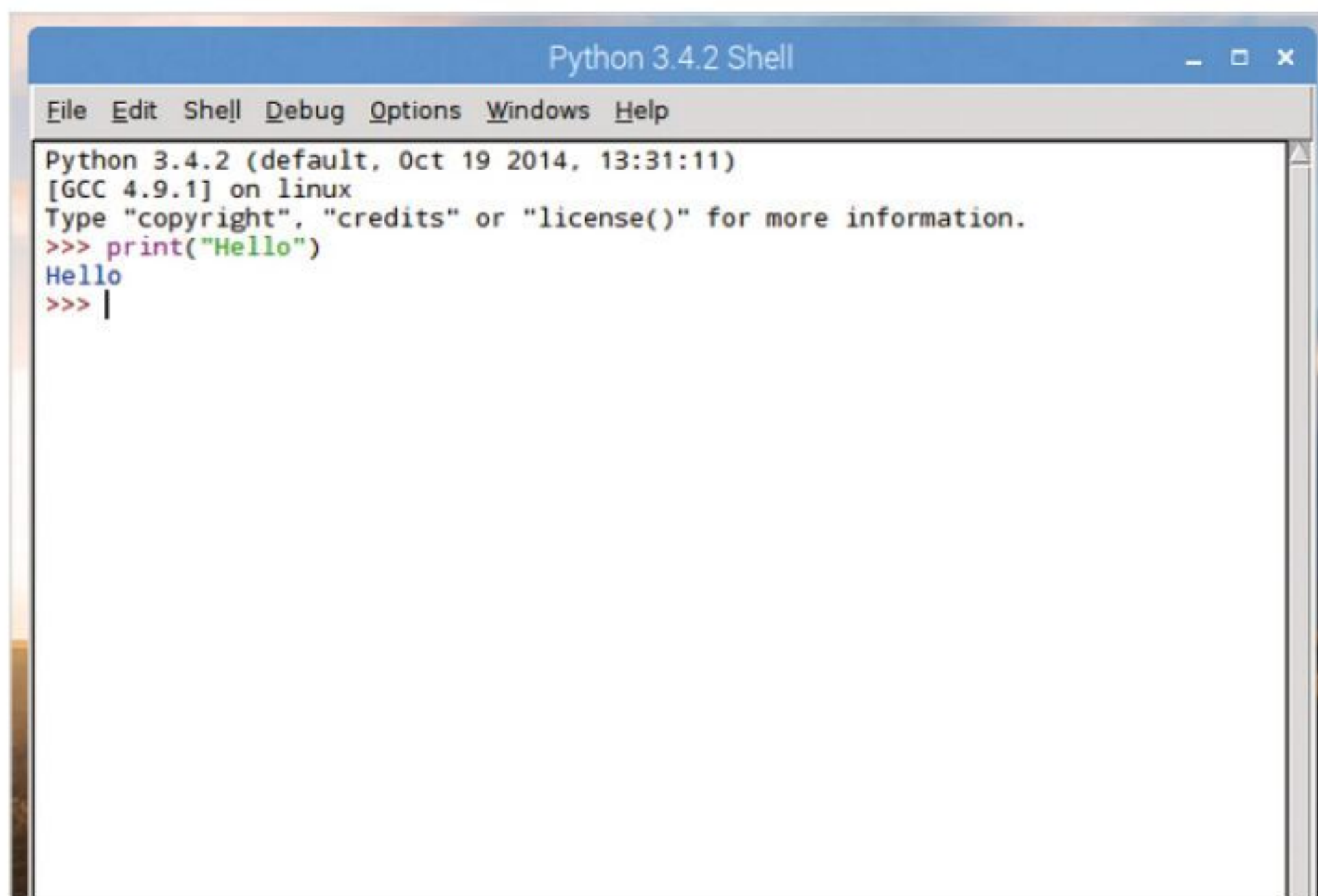
Essentially, you've already written your first piece of code with the 'print("Hello everyone!")' function from the previous tutorial. However, let's expand that and look at entering your code and playing around with some other Python examples.

PLAYING WITH PYTHON

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

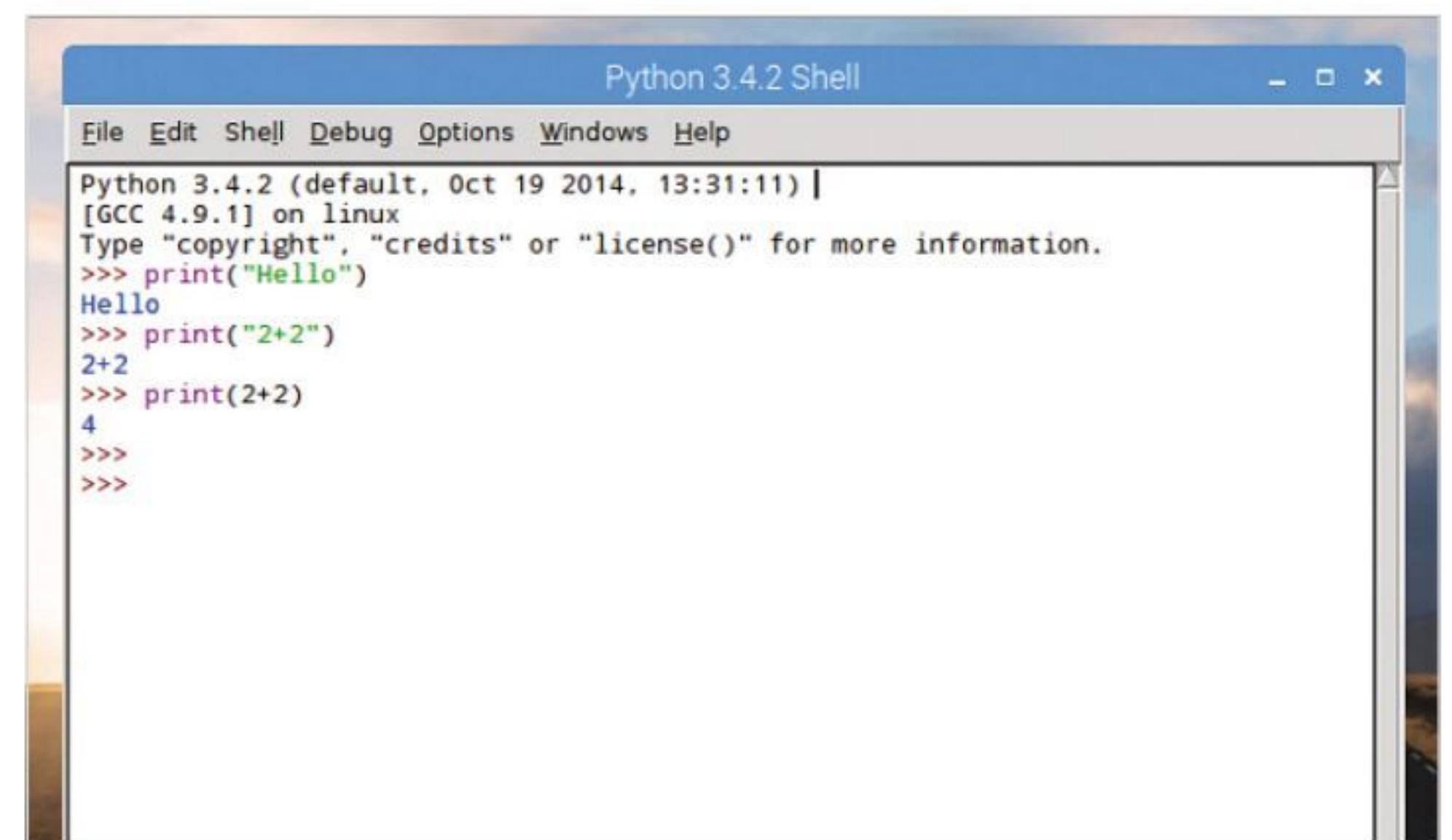
STEP 1 If you've closed Python 3 IDLE, reopen it in whichever operating system version you prefer. In the Shell, enter the familiar following:

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



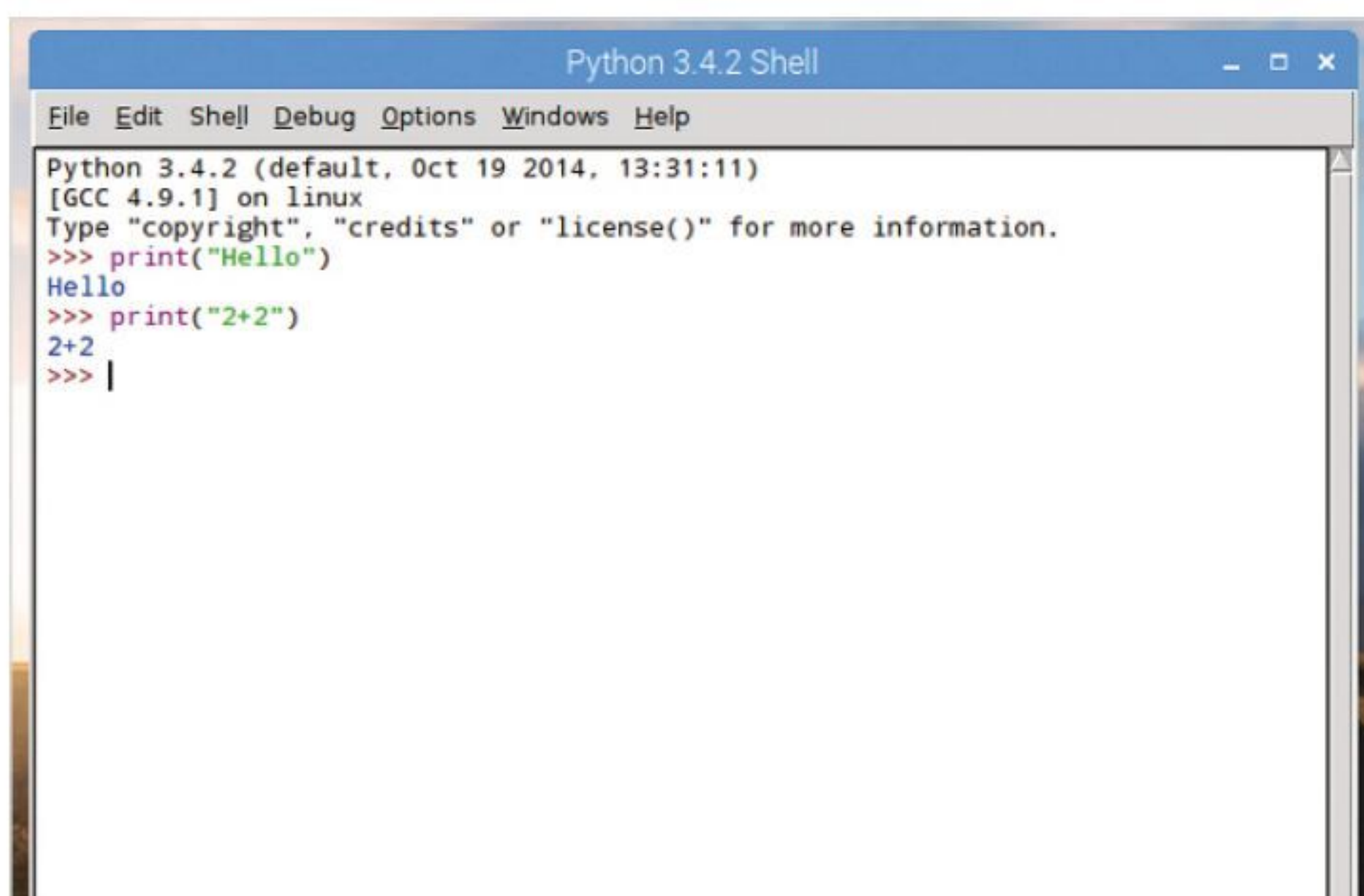
STEP 3 You can see that instead of the number 4, the output is the 2+2 you asked to be printed to the screen. The quotation marks are defining what's being outputted to the IDLE Shell; to print the total of 2+2 you need to remove the quotes:

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



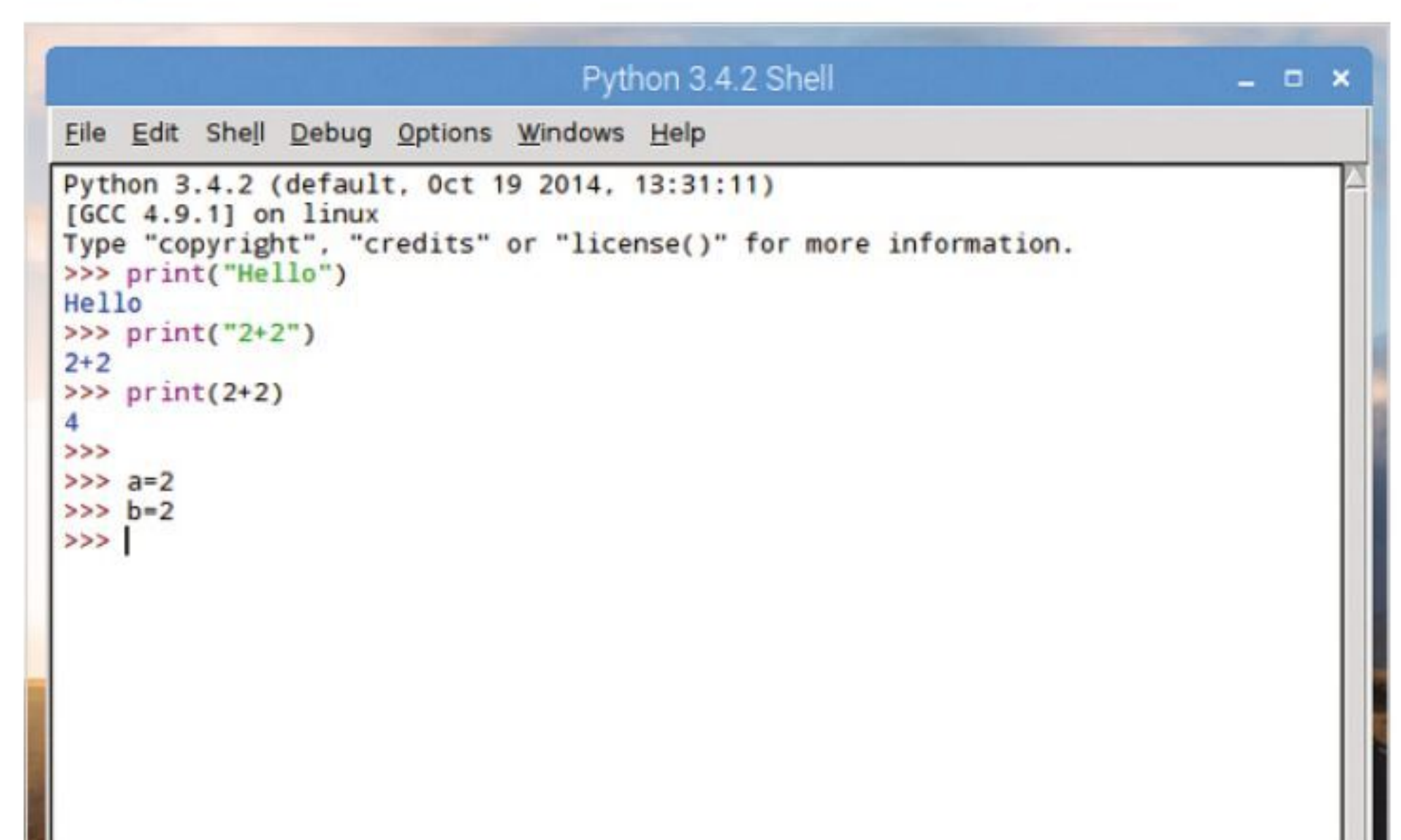
STEP 2 Just as predicted, the word Hello appears in the Shell as blue text, indicating output from a string. It's fairly straightforward and doesn't require too much explanation. Now try:

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



STEP 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)
```

```
Python 3.4.2 Shell
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> print("Hello")
Hello
>>> print("2+2")
2+2
>>> print(2+2)
4
>>>
>>> a=2
>>> b=2
>>> print(a)
2
>>> print(b)
2
>>> |
```

STEP 6

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

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

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

```
Python 3.4.2 Shell
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> print("Hello")
Hello
>>> print("2+2")
2+2
>>> print(2+2)
4
>>>
>>> a=2
>>> b=2
>>> print(a)
2
>>> print(b)
2
>>> print(a+b)
4
>>> |
```

STEP 7

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

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

```
Python 3.4.2 Shell
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> print("Hello")
Hello
>>> print("2+2")
2+2
>>> print(2+2)
4
>>>
>>> a=2
>>> b=2
>>> print(a)
2
>>> print(b)
2
>>> print(a+b)
4
>>>
>>> name="David"
>>> print(name)
David
>>> |
```

STEP 8

Now let's add a surname:

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

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

```
Python 3.4.2 Shell
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> name="David"
>>> print(name)
David
>>> surname="Hayward"
>>> print(surname)
Hayward
>>> |
```

STEP 9

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

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

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

```
Python 3.4.2 Shell
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> name="David"
>>> print(name)
David
>>> surname="Hayward"
>>> print(surname)
Hayward
>>> print(name+surname)
DavidHayward
>>> |
```

STEP 10

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

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

Alternatively, you can add the space yourself:

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

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

```
Python 3.4.2 Shell
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> name="David"
>>> print(name)
David
>>> surname="Hayward"
>>> print(surname)
Hayward
>>> print(name+surname)
DavidHayward
>>> print(name, surname)
David Hayward
>>> print(name+" "+surname)
David Hayward
>>> |
```



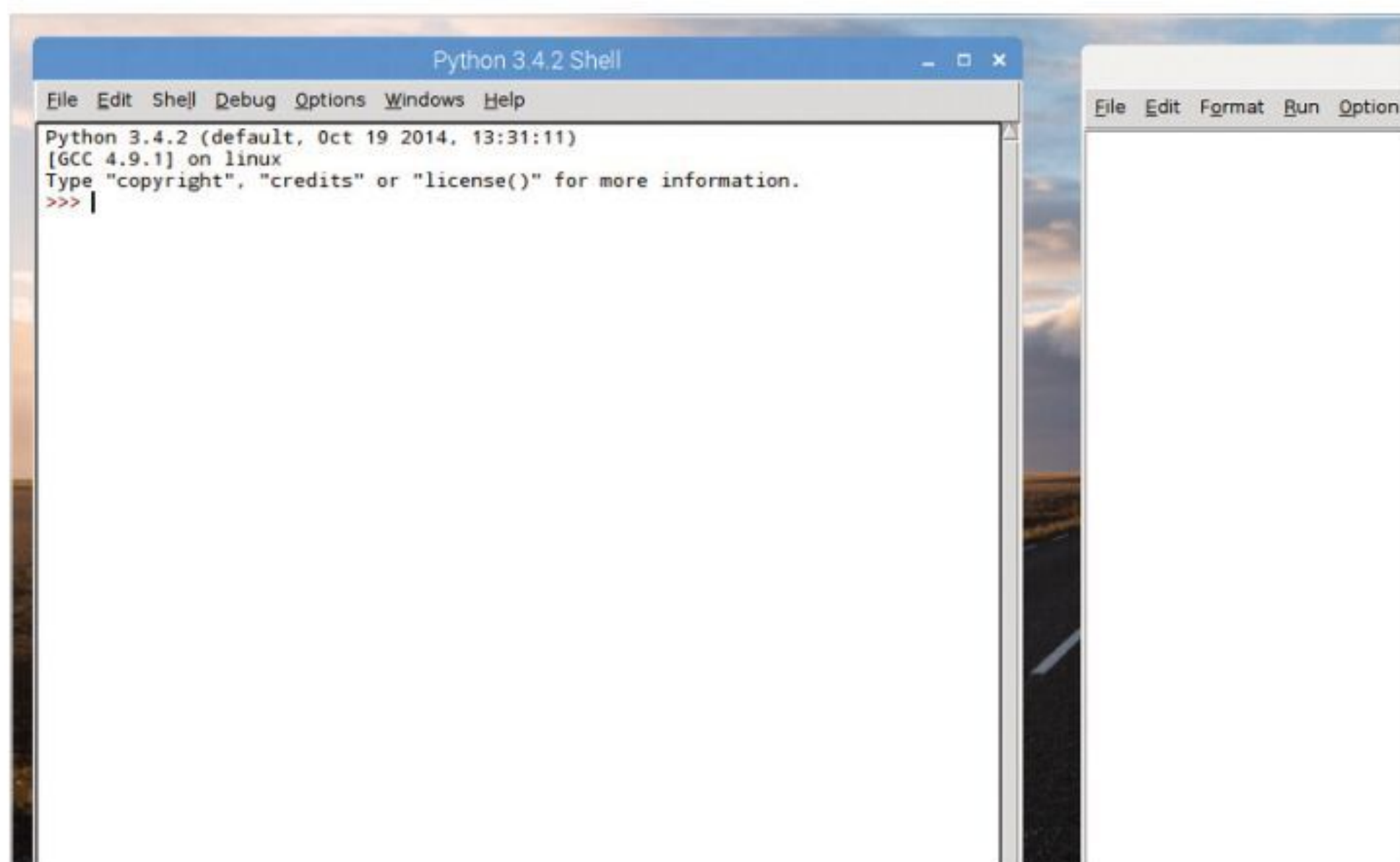
Saving and Executing Your Code

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

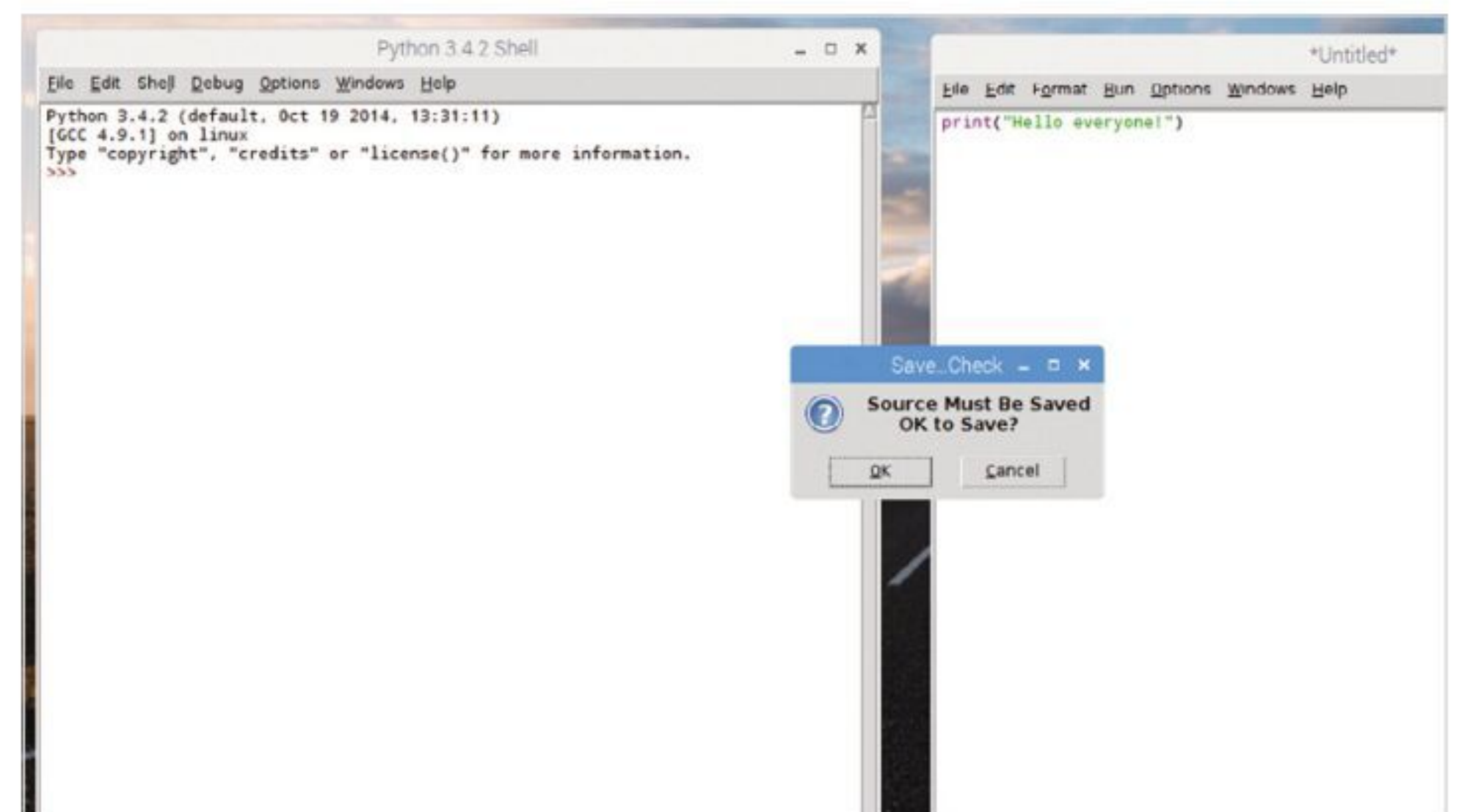
EDITING CODE

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

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



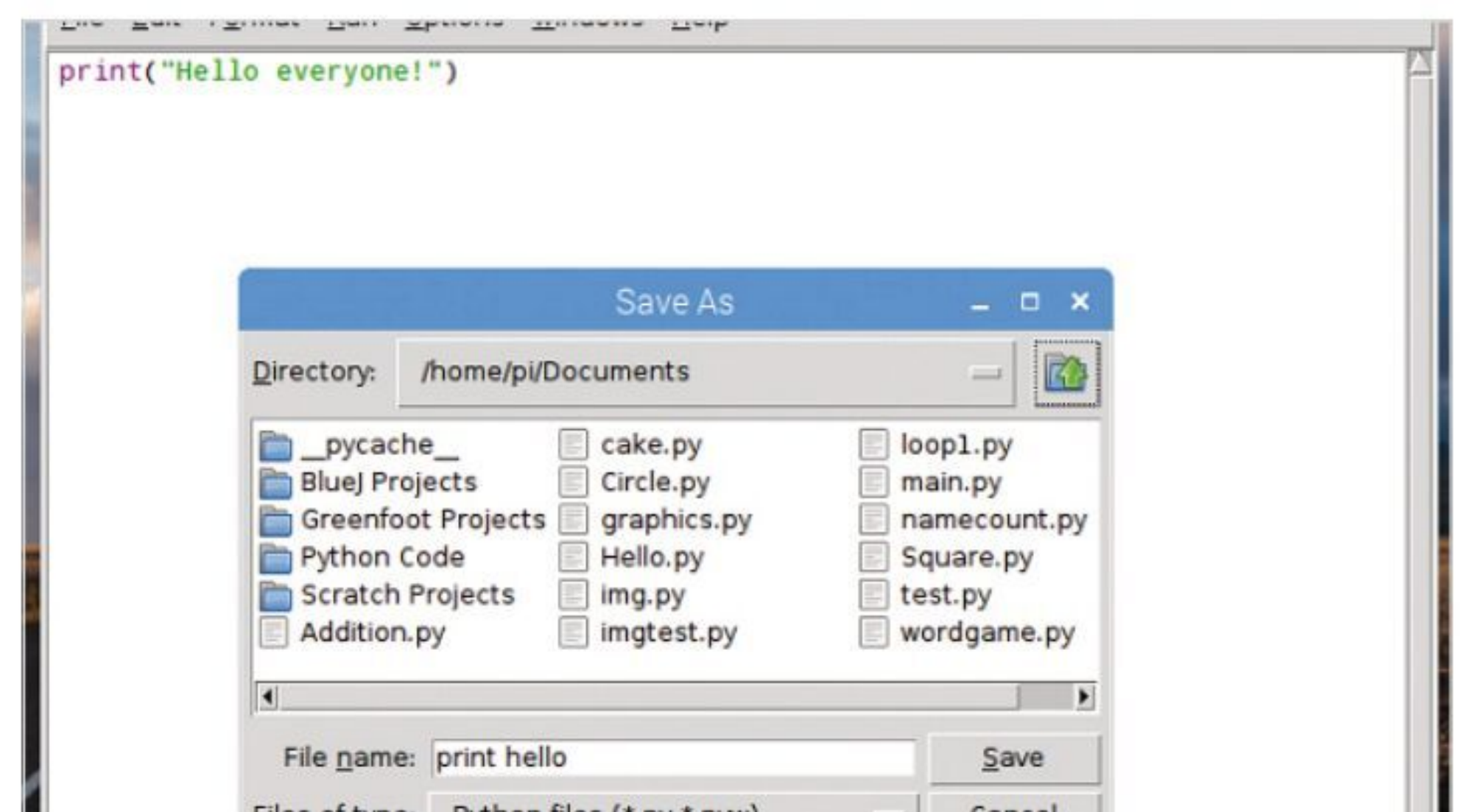
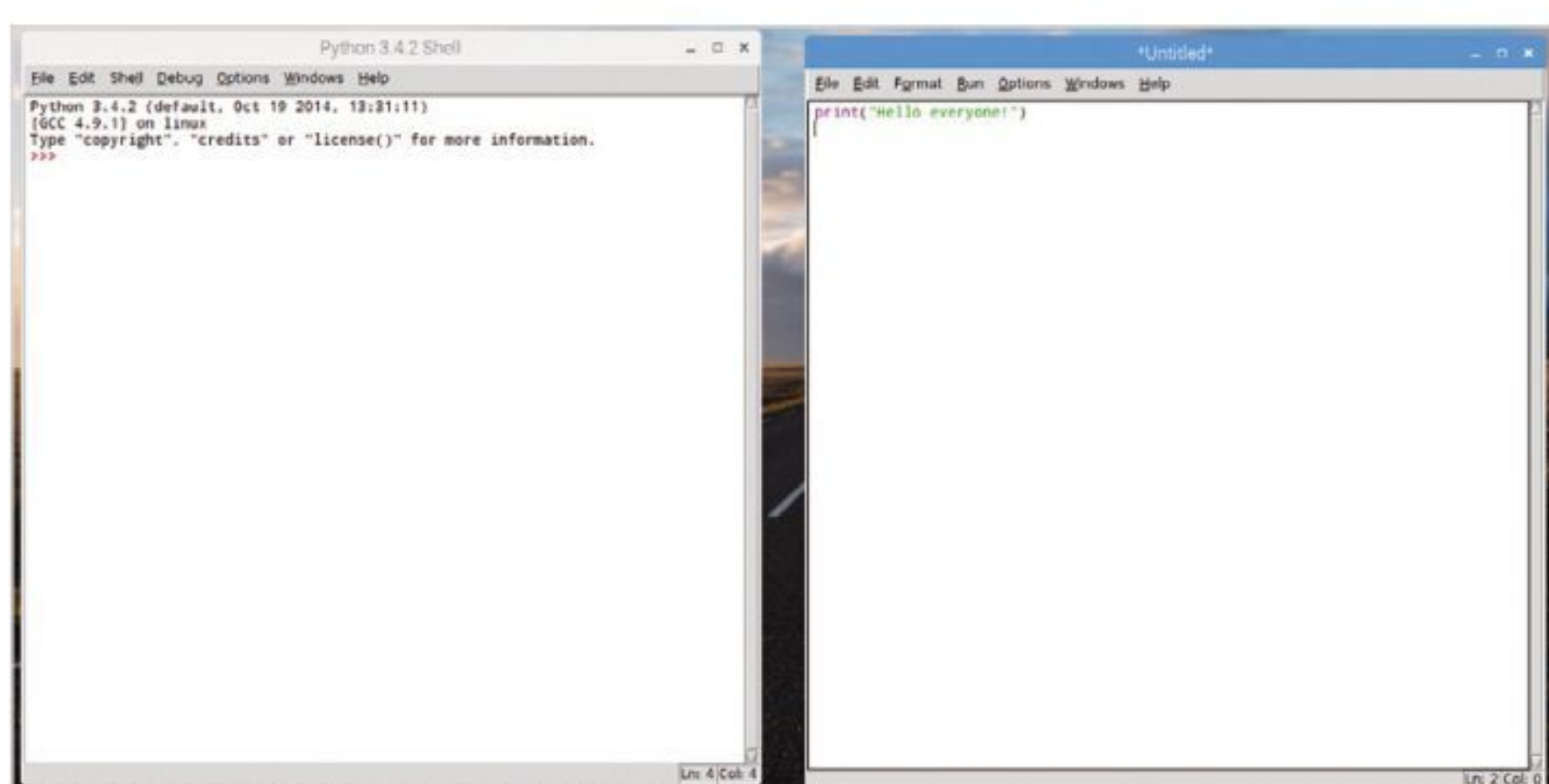
STEP 3 You can see that the same colour coding is in place in the IDLE Editor as it is in the Shell, enabling you to better understand what's going on with your code. However, to execute the code you need to first save it. Press F5 and you get a Save...Check box open.



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

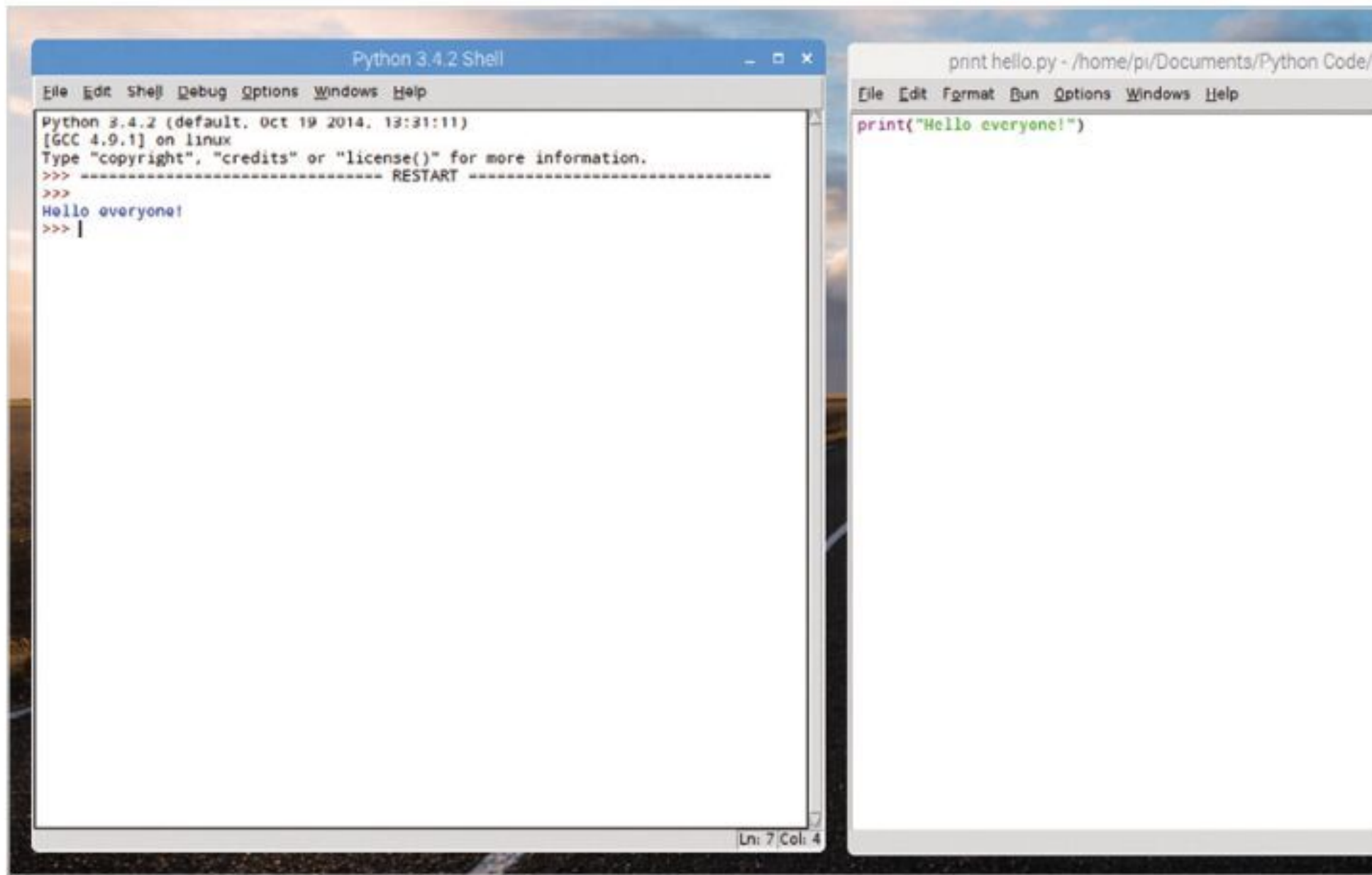
```
print("Hello everyone!")
```

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





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



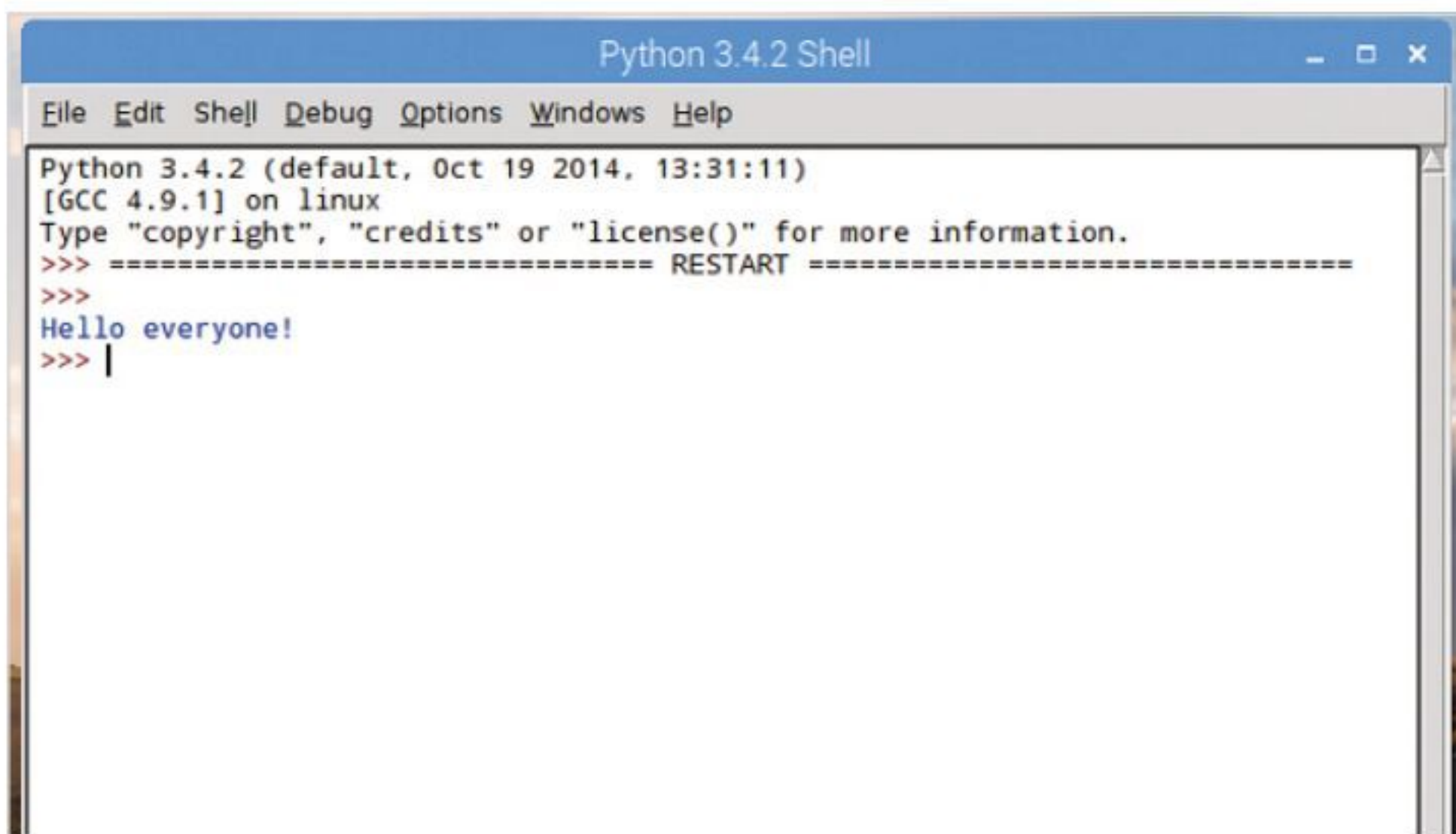
STEP 8 Let's extend the code and enter a few examples from the previous tutorial:

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

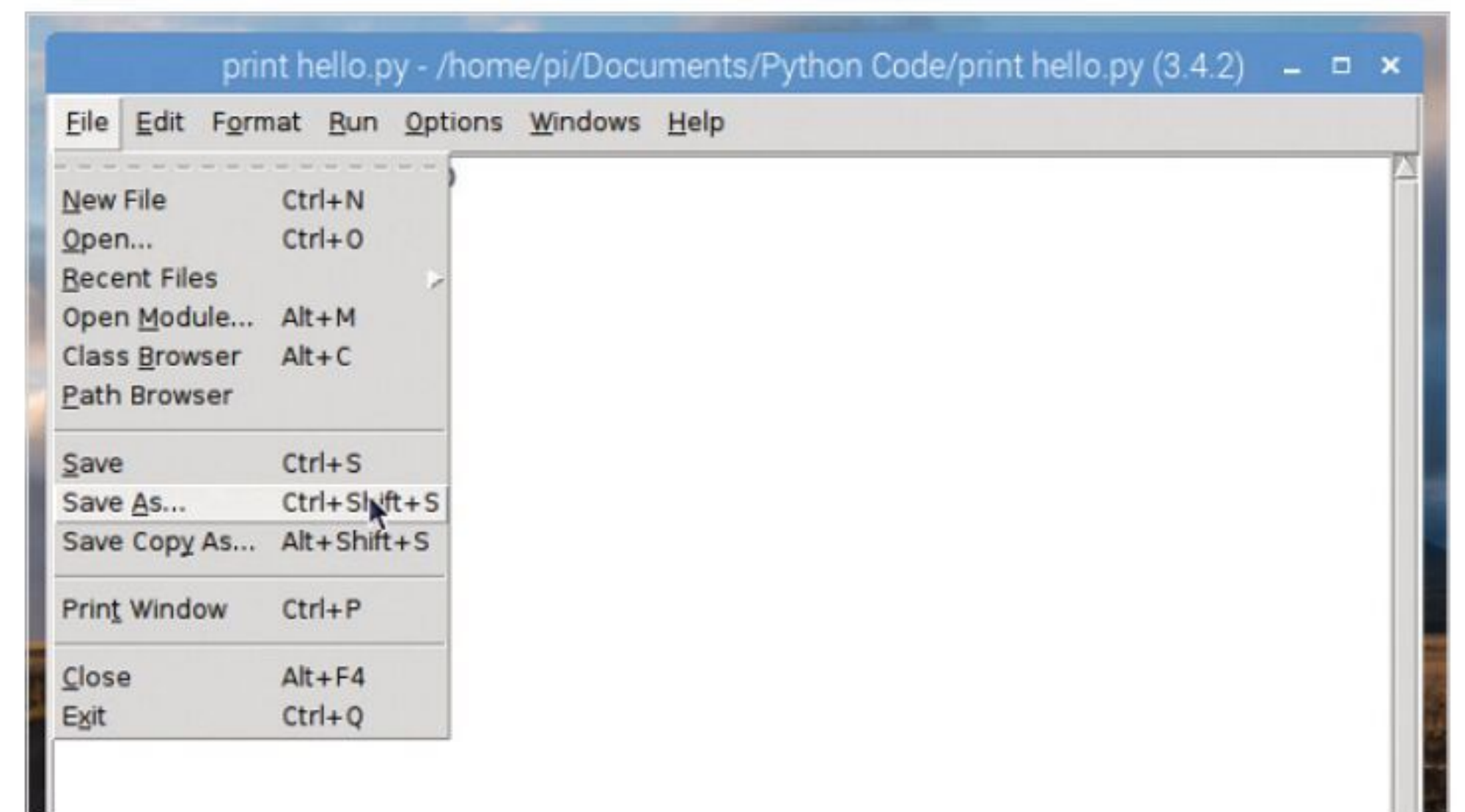
If you press F5 now you'll be asked to save the file, again, as it's been modified from before.



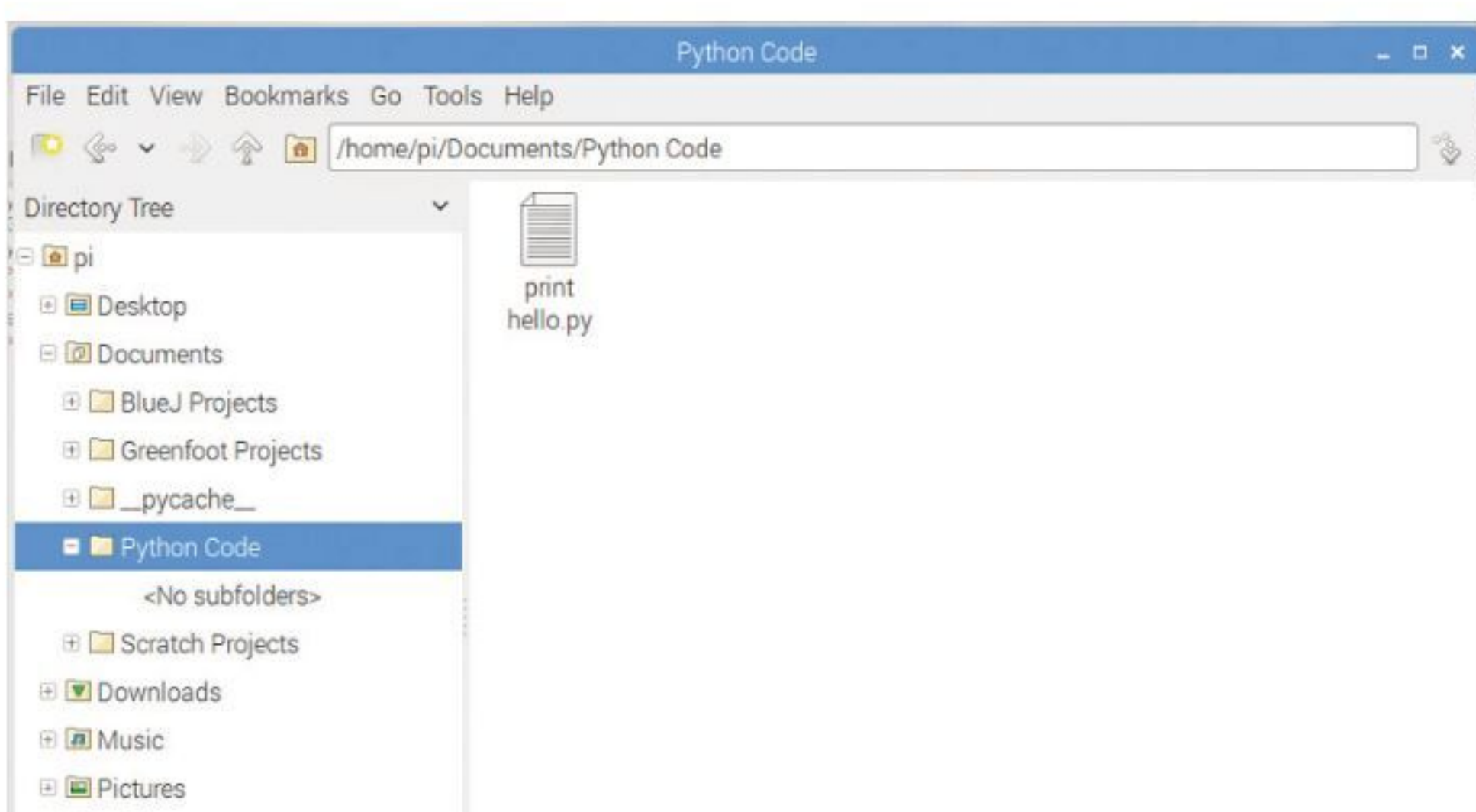
STEP 6 This is how the vast majority of your Python code will be conducted. Enter it into the Editor, hit F5, save the code and look at the output in the Shell. Sometimes things will differ, depending on whether you've requested a separate window, but essentially that's the process. It's the process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



STEP 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.



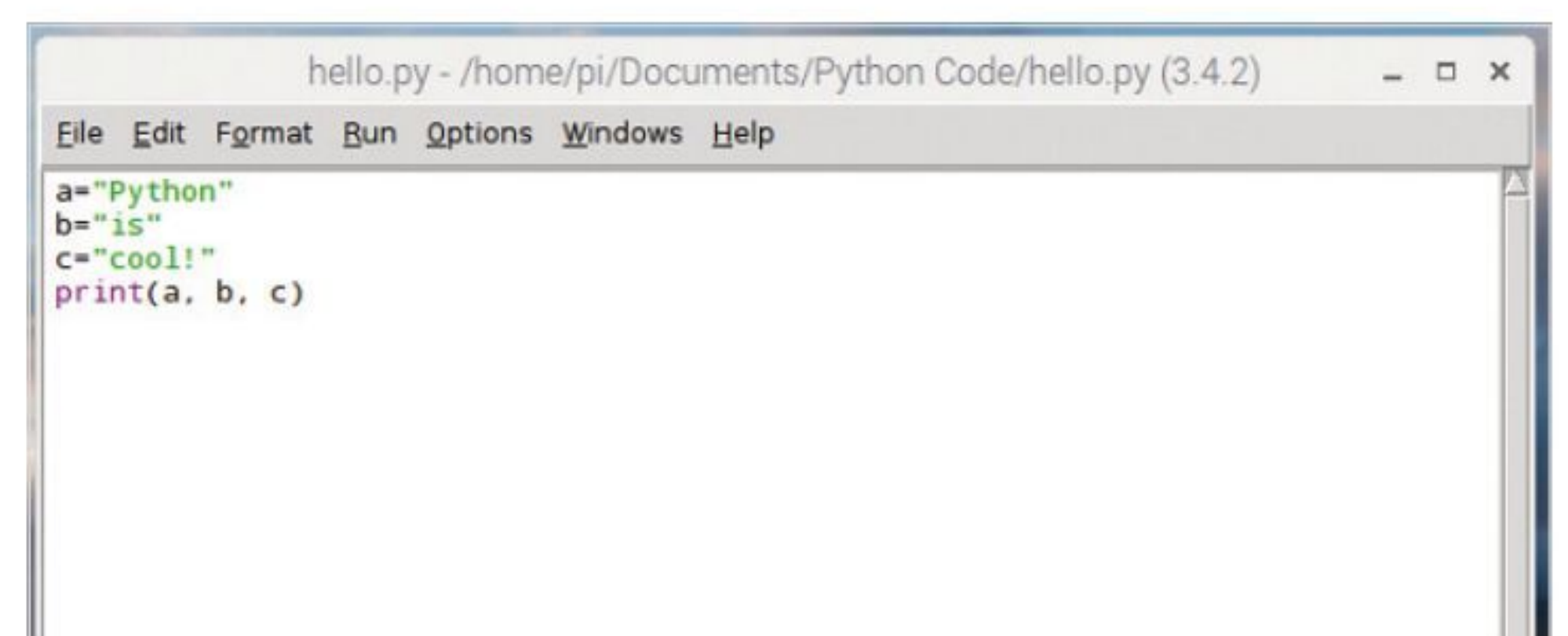
STEP 7 If you open the file location of the saved Python code, you can see that it ends in a .py extension. This is the default Python file name. Any code you create will be whatever.py and any code downloaded from the many Internet Python resource sites will be .py. Just ensure that the code is written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



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

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

You will use this code in the next tutorial.





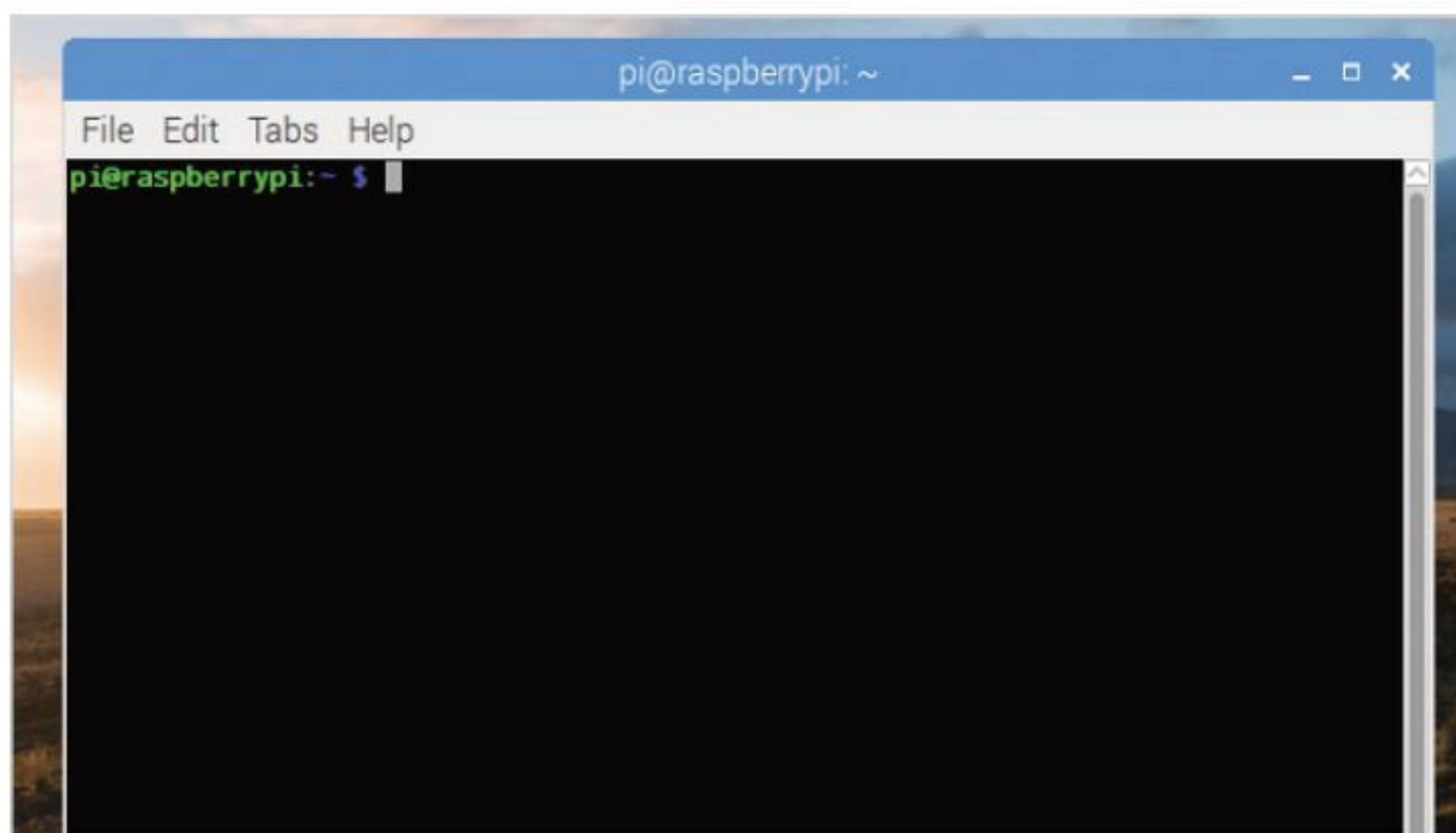
Executing Code from the Terminal

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

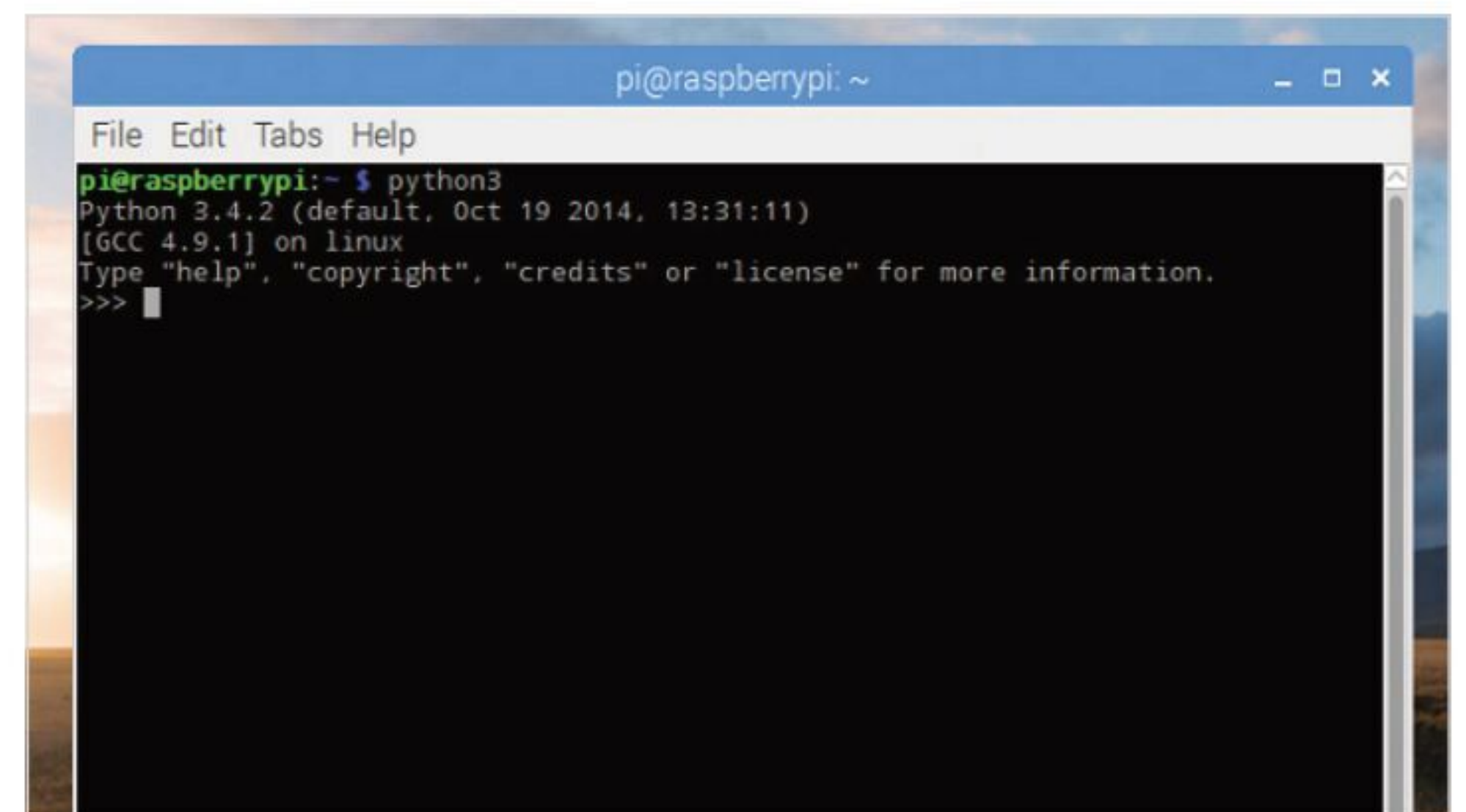
COMMAND THE CODE

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

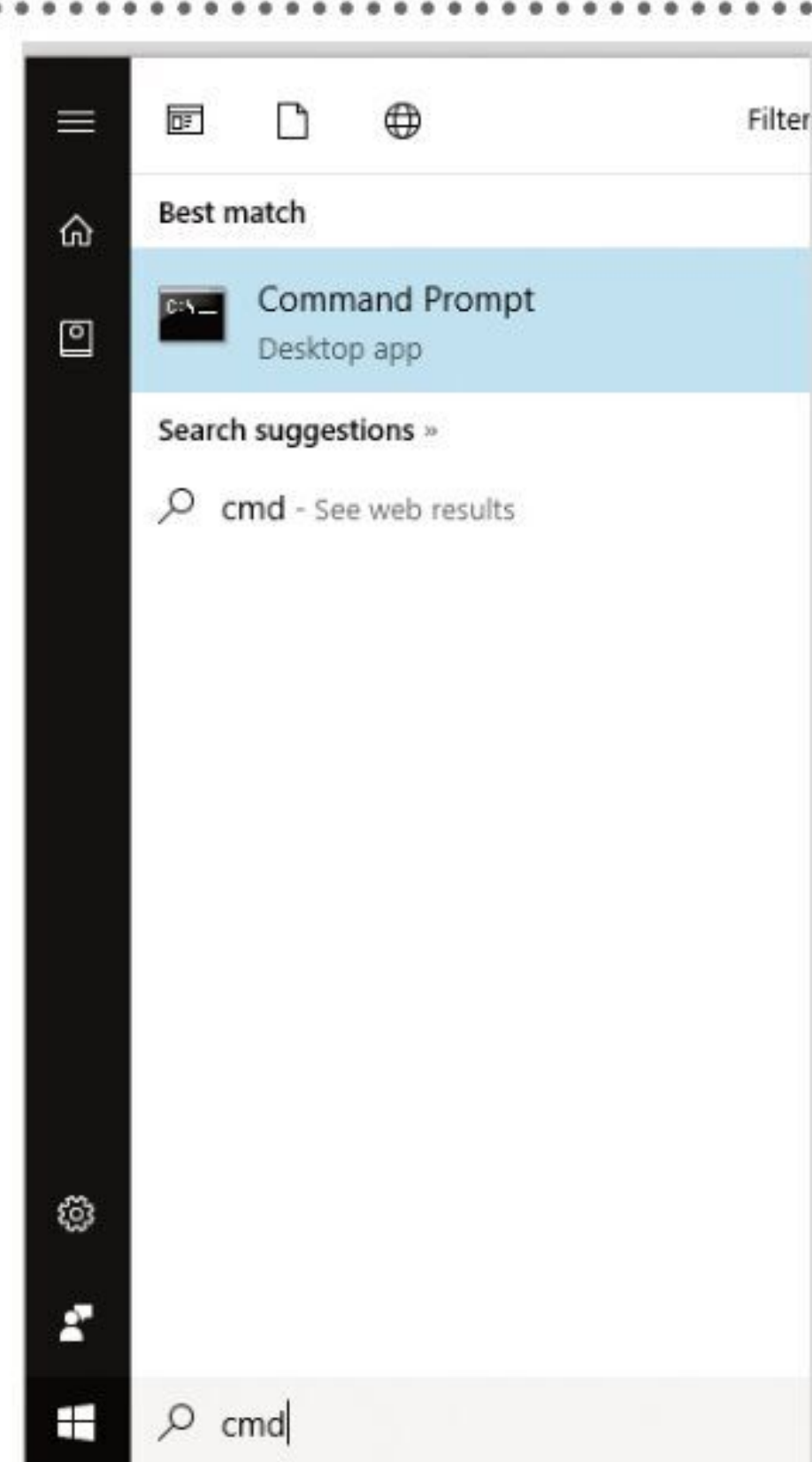
STEP 1 Python, in Linux, comes with two possible ways of executing code via the command line. One of the ways is with Python 2, whilst the other uses the Python 3 libraries and so on. First though, drop into the command line or Terminal on your operating system.



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



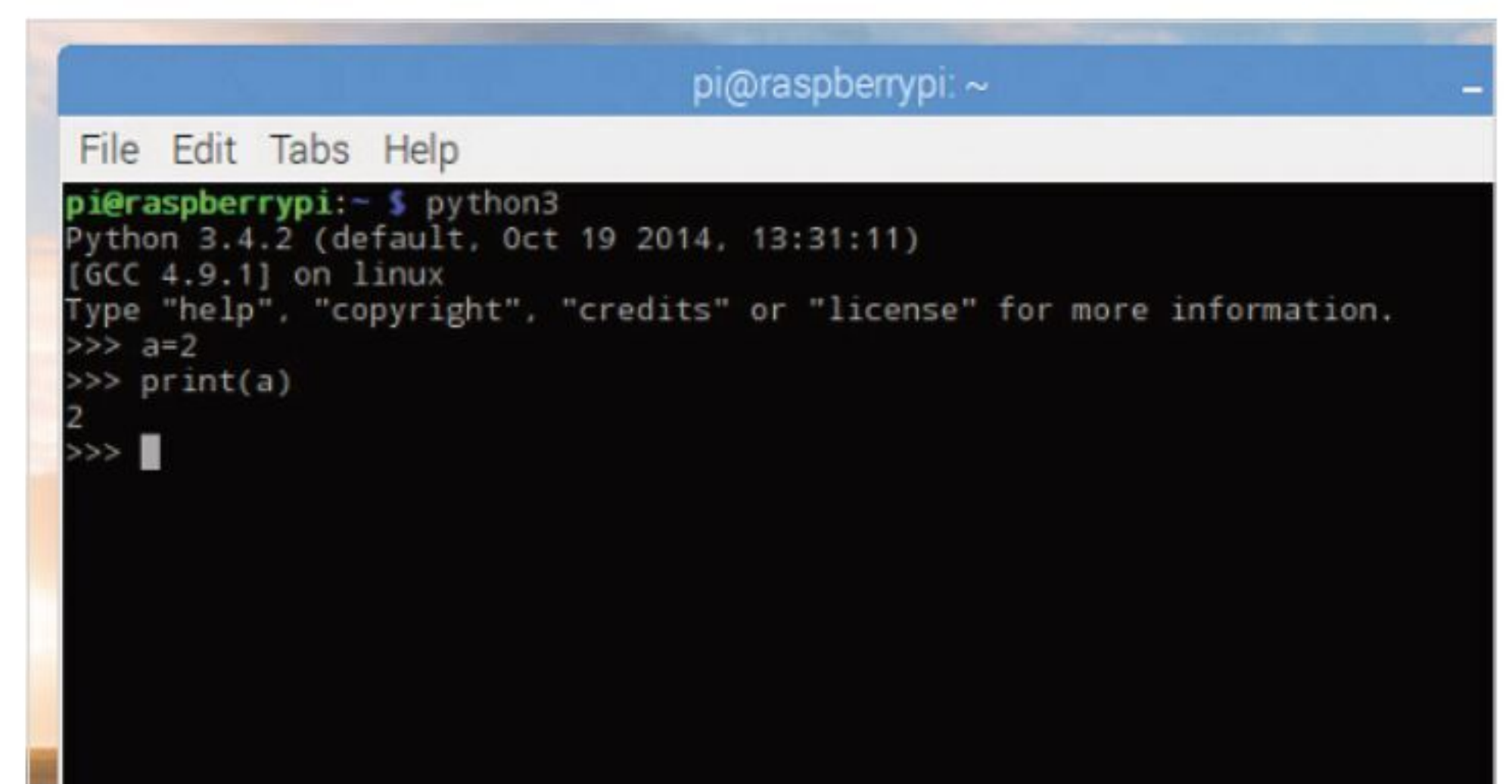
STEP 2 Just as before, we're using a Raspberry Pi: Windows users will need to click the Start button and search for CMD, then click the Command Line returned search; and macOS users can get access to their command line by clicking Go > Utilities > Terminal.



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

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

You can see that it works exactly the same.





STEP 5

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

```

pi@raspberrypi: ~/Documents/Python Code
File Edit Tabs Help
pi@raspberrypi:~$ python3
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> a=2
>>> print(a)
2
>>> exit()
pi@raspberrypi:~$ cd Documents/
pi@raspberrypi:~/Documents$ cd Python\ Code/
pi@raspberrypi:~/Documents/Python Code$ ls
hello.py print hello.py
pi@raspberrypi:~/Documents/Python Code$
    
```

STEP 6

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

```
python3 hello.py
```

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

```

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

```

pi@raspberrypi:~$ python3
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> a=2
>>> print(a)
2
>>> exit()
pi@raspberrypi:~$ cd Documents/
pi@raspberrypi:~/Documents$ cd Python\ Code/
pi@raspberrypi:~/Documents/Python Code$ ls
hello.py print hello.py
pi@raspberrypi:~/Documents/Python Code$ python3 hello.py
Python is cool!
pi@raspberrypi:~/Documents/Python Code$
    
```

STEP 7

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

```
python hello.py
```

```

pi@raspberrypi:~/Documents/Python Code
File Edit Tabs Help
pi@raspberrypi:~$ python3
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> a=2
>>> print(a)
2
>>> exit()
pi@raspberrypi:~$ cd Documents/
pi@raspberrypi:~/Documents$ cd Python\ Code/
pi@raspberrypi:~/Documents/Python Code$ ls
hello.py print hello.py
pi@raspberrypi:~/Documents/Python Code$ python3 hello.py
Python is cool!
pi@raspberrypi:~/Documents/Python Code$ python hello.py
('Python', 'is', 'cool!')
pi@raspberrypi:~/Documents/Python Code$
    
```

STEP 8

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

```

C:\Users\david\Documents\Python\hello.py - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
hello.py
1 a="Python"
2 b="is"
3 c="cool!"
4 print(a, b, c)
5
    
```

STEP 9

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

```

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

```

C:\Users\david\Documents\Python\hello.py - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
hello.py
1 a="Python"
2 b="is"
3 c="cool!"
4 print(a, b, c)
5 name=input("What is your name? ")
6 print("Hello,", name)
7
    
```

STEP 10

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

```
python3 hello.py
```

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

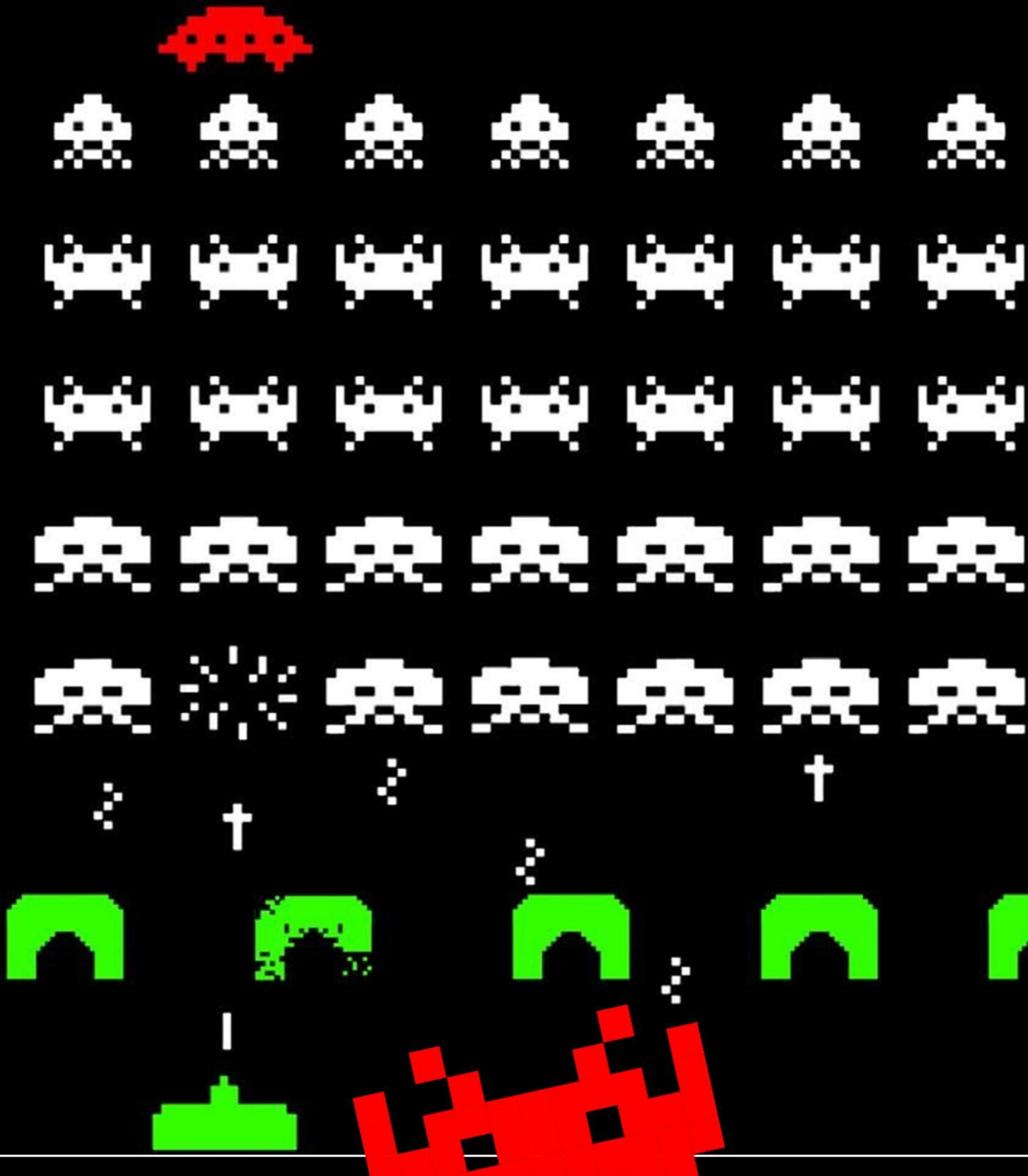
```

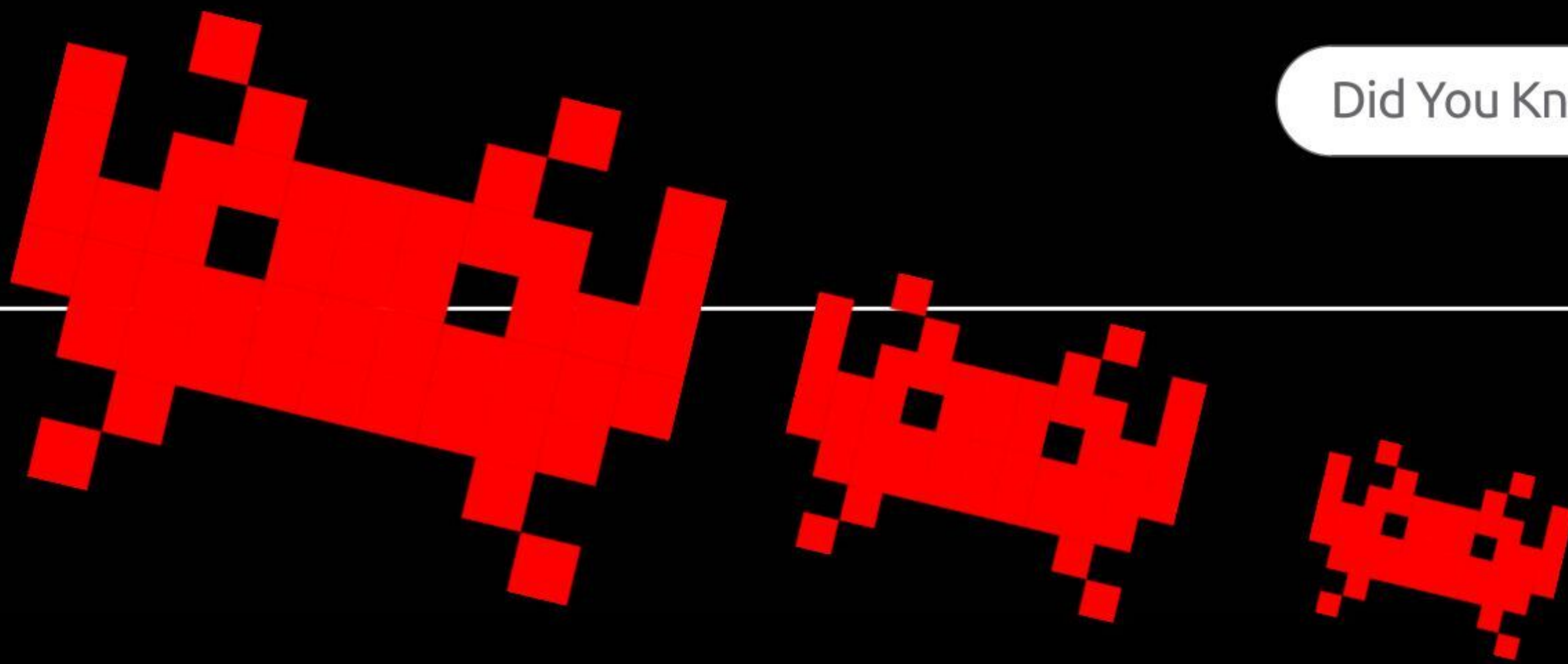
pi@raspberrypi:~/Documents/Python Code
File Edit Tabs Help
pi@raspberrypi:~/Documents/Python Code$ python3 hello.py
Python is cool!
what is your name? David
Hello, David
pi@raspberrypi:~/Documents/Python Code$
    
```



Space Invaders

SCORE 1,337





LIVES



DID YOU KNOW...

that when Space Invaders creator, Tomohiro Nishikado, was coding Space Invaders, he had to create his own hardware and development tools to write the game. Furthermore, he then discovered that as a result of the coding and the custom hardware, the fewer Invaders on-screen the faster the processor was able to render them. Thus, as you kill more Invaders, the faster they become. So rather than design the game to compensate for the speed increase, he decided to keep it as a challenging gameplay mechanism.

It was pure accident that the Invaders got faster the less of them there were.



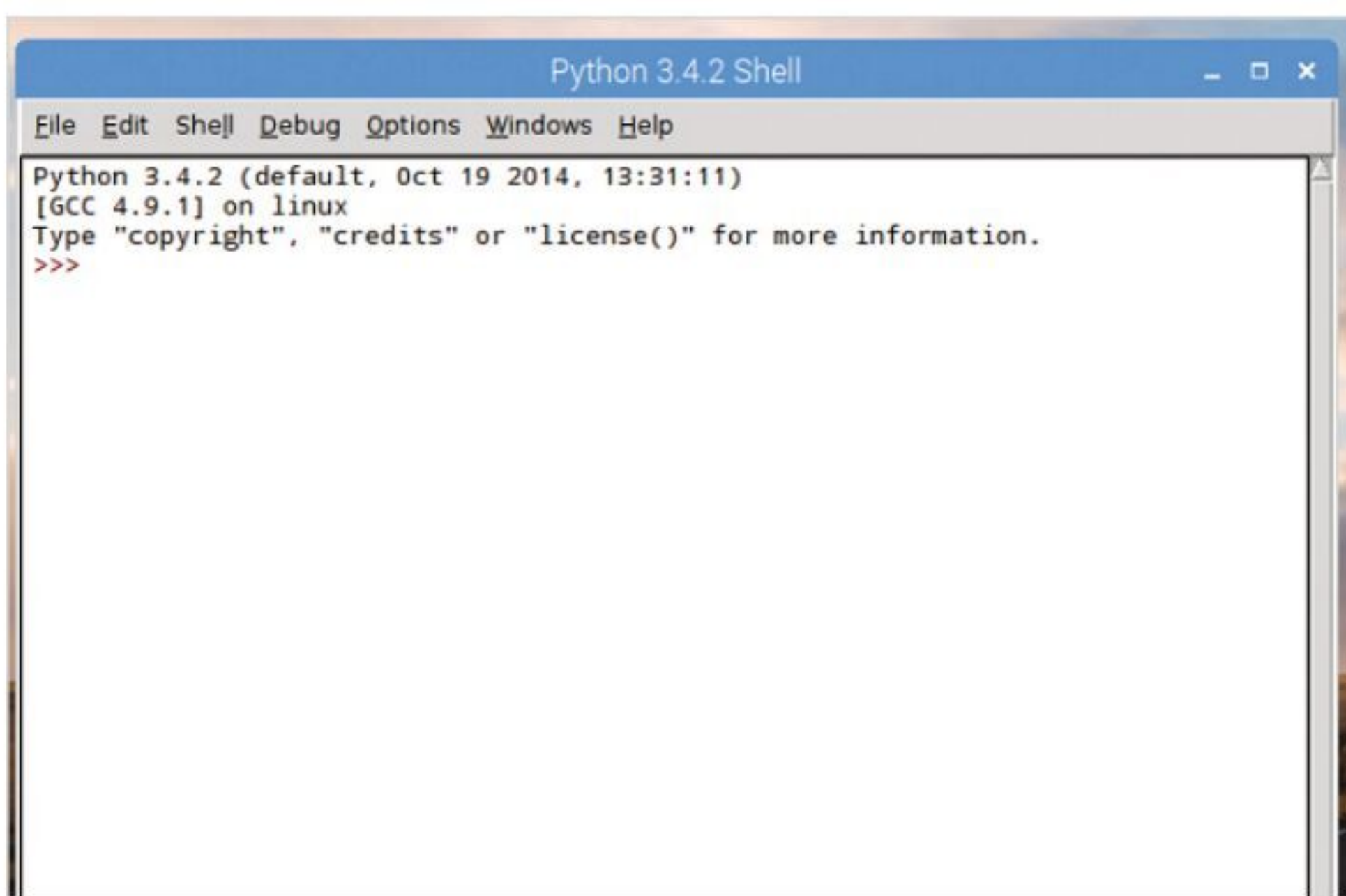
Numbers and Expressions

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

IT'S ALL MATHS, MAN

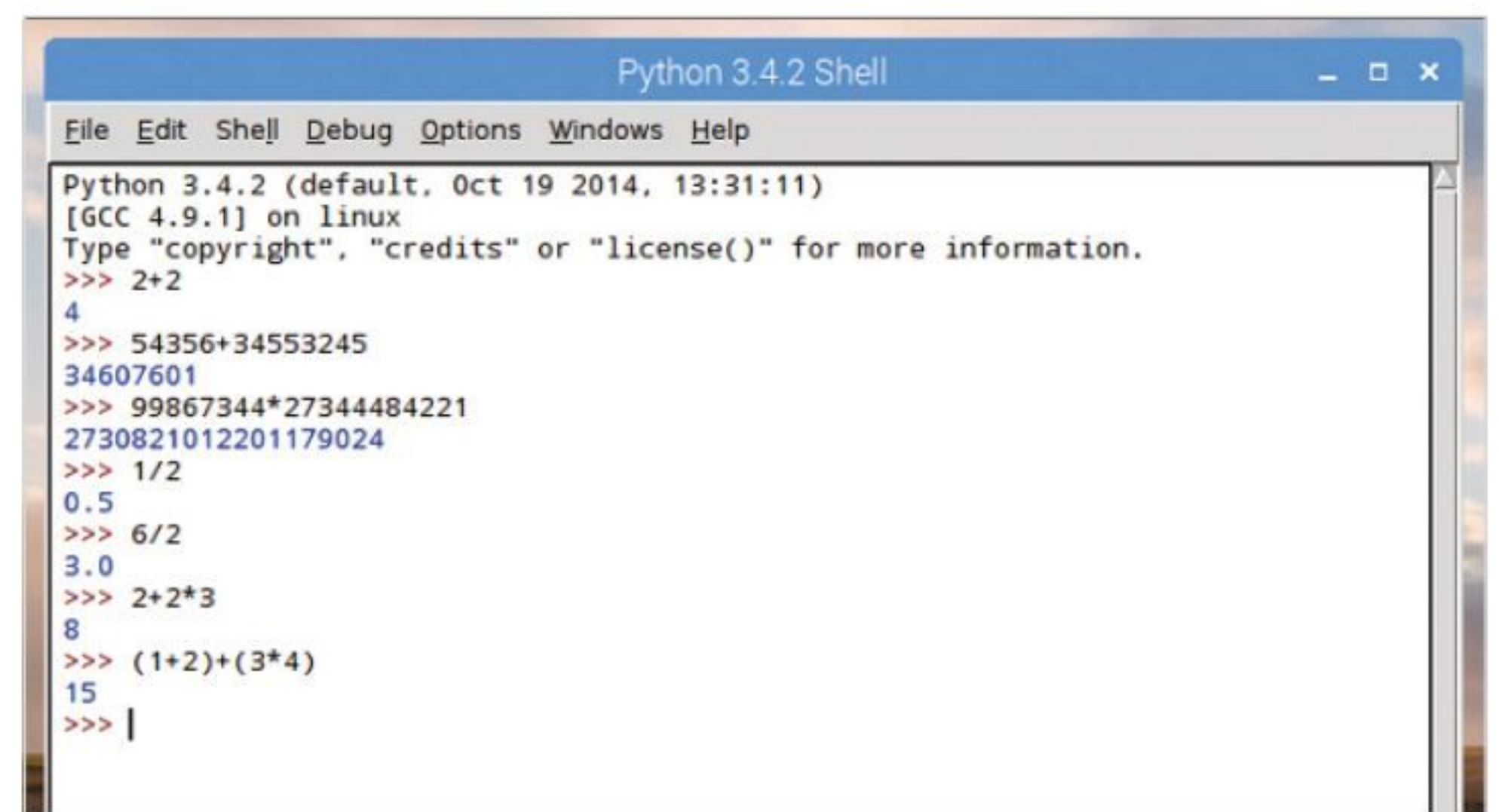
You can get some really impressive results with the mathematical powers of Python; as with most, if not all, programming languages, maths is the driving force behind the code.

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



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

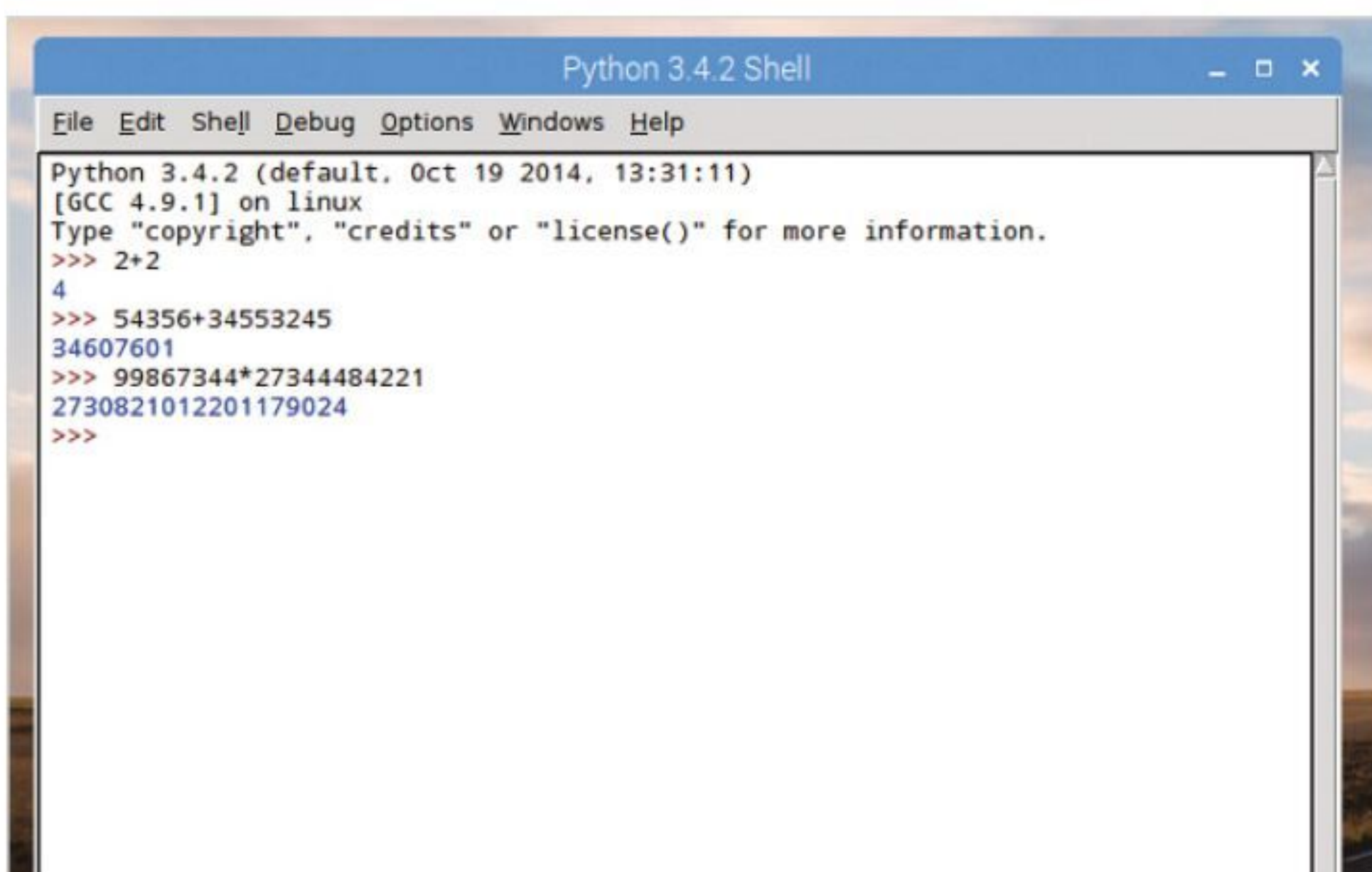
1/2
6/2
2+2*3
(1+2)+(3*4)



STEP 2 In the Shell enter the following:

2+2
54356+34553245
99867344*27344484221

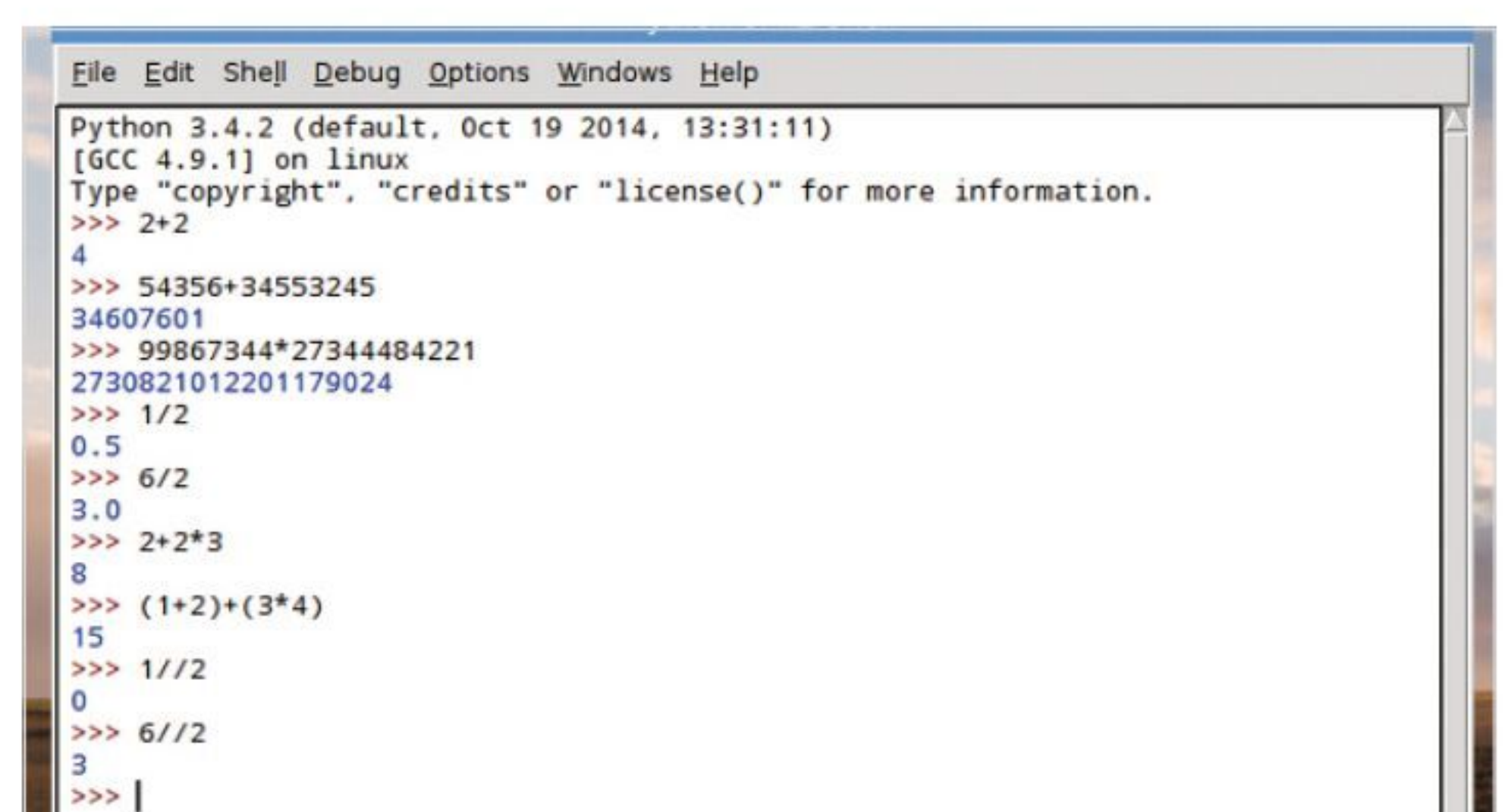
You can see that Python can handle some quite large numbers.



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

1//2
6//2

And so on.





STEP 5

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

`10/3`

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

`10%3`

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

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

STEP 6

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

`2**3`

`10**10`

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

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

STEP 7

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

`bin(3)`

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

STEP 8

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

`format(3, 'b')`

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

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

STEP 9

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

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

STEP 10

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



Using Comments

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

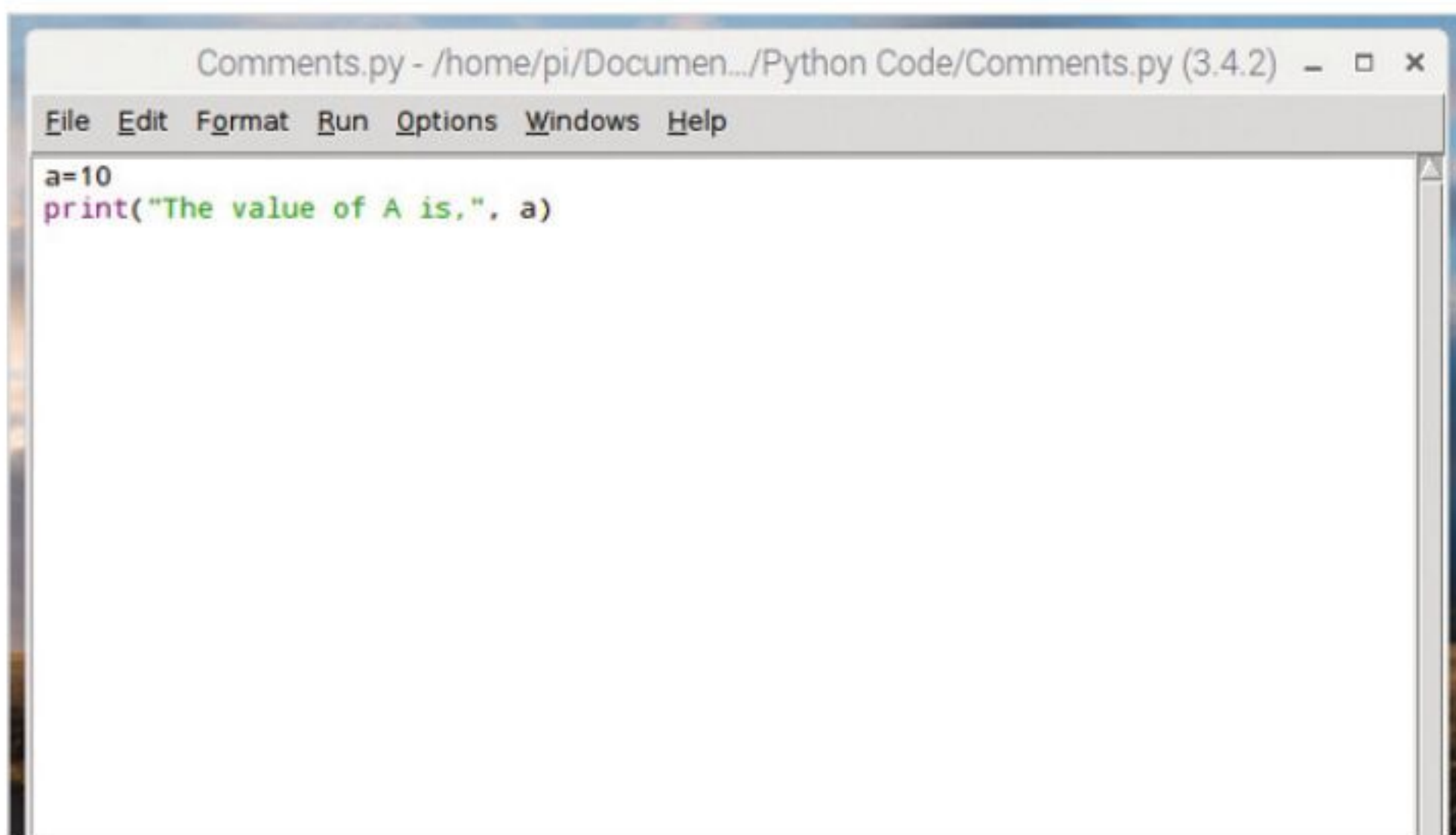
#COMMENTS!

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

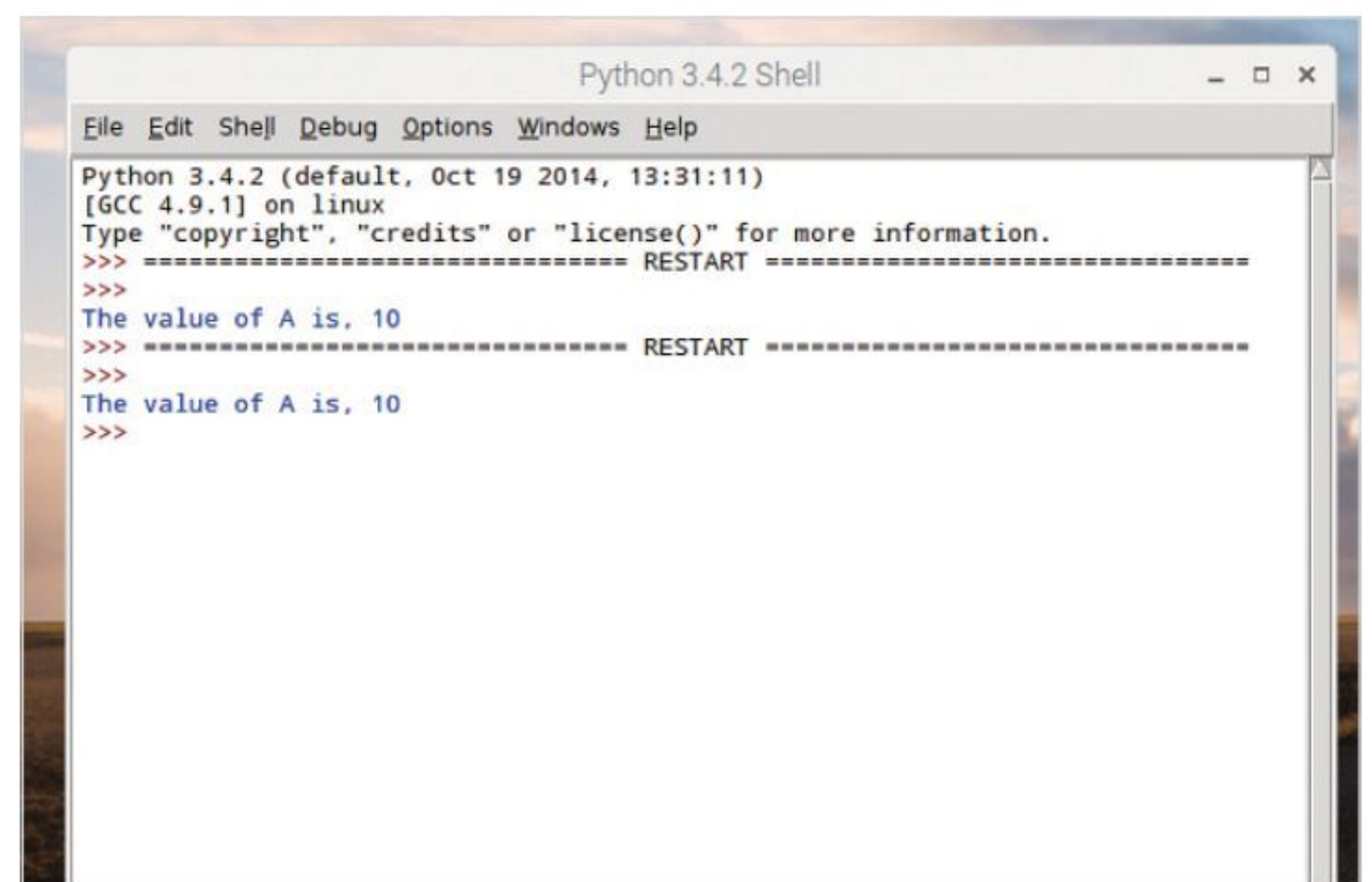
STEP 1 Start by creating a new instance of the IDLE Editor (File > New File) and create a simple variable and print command:

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

Save the file and execute the code.

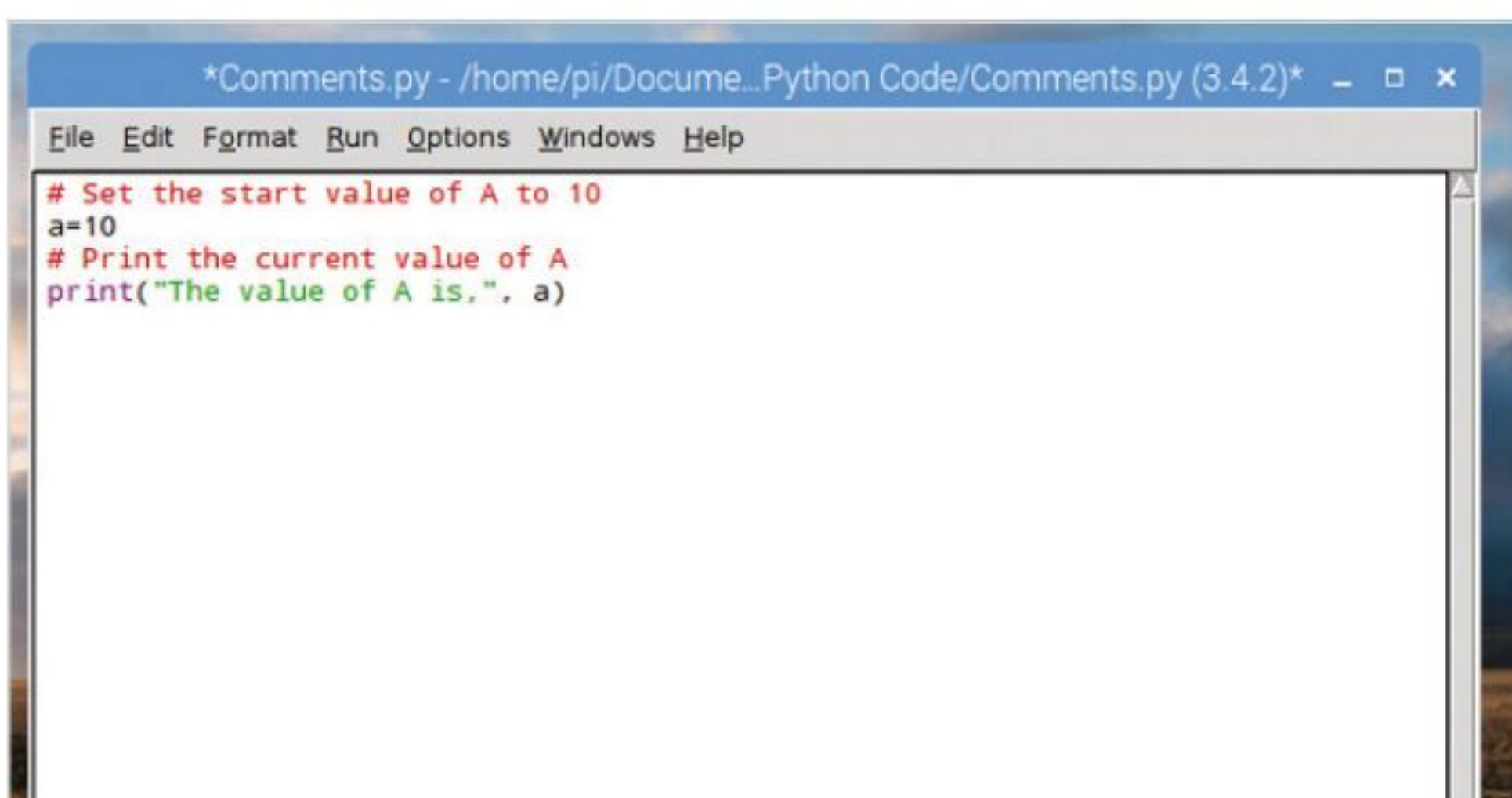


STEP 3 Resave the code and execute it. You can see that the output in the IDLE Shell is still the same as before, despite the extra lines being added. Simply put, the hash symbol (#) denotes a line of text the programmer can insert to inform them, and others, of what's going on without the user being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



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

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



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

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



**STEP 5**

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

```
Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> ----- RESTART -----
>>>
The value of A is, 10
>>> ----- RESTART -----
>>>
The value of A is, 10
>>> ----- RESTART -----
>>>
The value of A is, 10
You've just lost a life!
You now have 9 lives left!
>>>
```

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!")
```

```
File Edit Format Run Options Windows Help
# Set the start value of A to 10
a=10
# Print the current value of A
print("The value of A is,", a)
# Player lost a life!
a=a-1
# Inform player, and display current value of A (lives)
print("You've just lost a life!")
print("You now have", a, "lives left!")
```

STEP 7

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

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

```
*Comments.py - /home/pi/Documents/Python Code/Comments.py (3.4.2)*
File Edit Format Run Options Windows Help
# 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.
# 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 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)
```

```
Comments.py - /home/pi/Documents/Python Code/Comments.py (3.4.2)
File Edit Format Run Options Windows Help
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)
```

```
*Comments.py - /home/pi/Documents/Python
File Edit Format Run Options Windows Help
# 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 10

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

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

```
*Comments.py - /home/pi/Documents/Python Code/Comments.py (3.4.2)
File Edit Format Run Options Windows Help
'''
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.
'''
# 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!")
```



Working with Variables

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

VARIOUS VARIABLES

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

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

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

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

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

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

Then output the content of the new variable as:

```
print (character)
```

Numbers are stored as different variables:

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

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

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

```
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> name="David Hayward"
>>> print (name)
David Hayward
>>> type (name)
<class 'str'>
>>> title="Descended from Vikings"
>>> print (name + ": " + title)
David Hayward: Descended from Vikings
>>> character=name + ": " + title
>>> print (character)
David Hayward: Descended from Vikings
>>> age=44
>>> type (age)
<class 'int'>
>>> print (name+age)
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    print (name+age)
TypeError: Can't convert 'int' object to str implicitly
>>> |
```

STEP 6

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

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

or you can use:

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

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

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

STEP 7

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

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

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

```
Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> age= input ("How old are you? ")
How old are you? 44
>>> 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, as `age + 10` won't work due to being a string variable and an integer. Instead, you need to enter:

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

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

```
Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> age= input ("How old are you? ")
How old are you? 44
>>> type(age)
<class 'str'>
>>> age + 10
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    age + 10
TypeError: Can't convert 'int' object to str implicitly
>>> int(age) + 10
54
>>> |
```

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 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> 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 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> shirt=19.99
>>> type(shirt)
<class 'float'>
>>> int(shirt)
19
>>> |
```



User Input

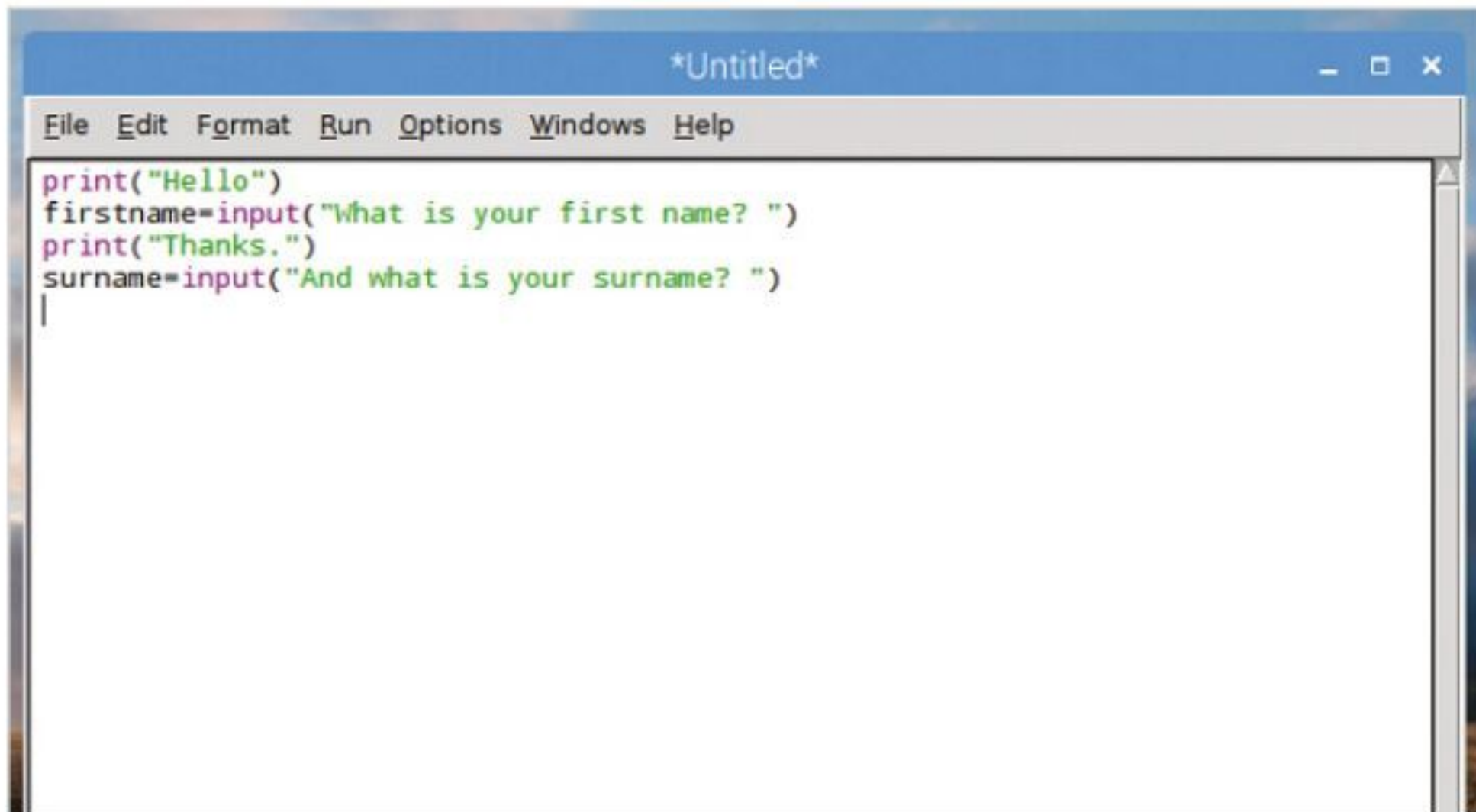
We've seen some basic user interaction with the code from a few of the examples earlier, so now would be a good time to focus solely on how you would get information from the user then store and present it.

USER FRIENDLY

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

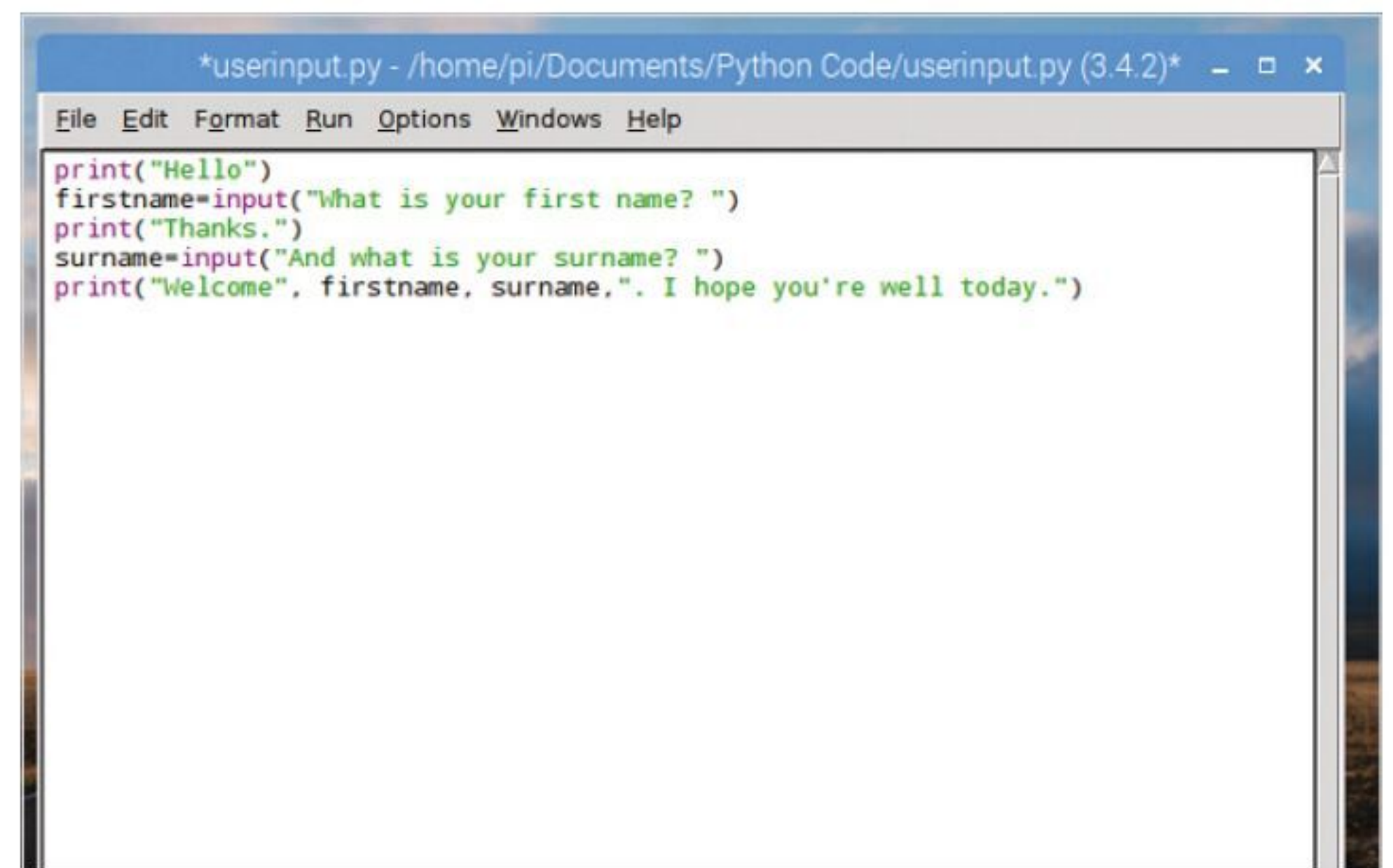
STEP 1 If it's not already, open the Python 3 IDLE Shell, and start a New File in the Editor. Let's begin with something really simple, enter:

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

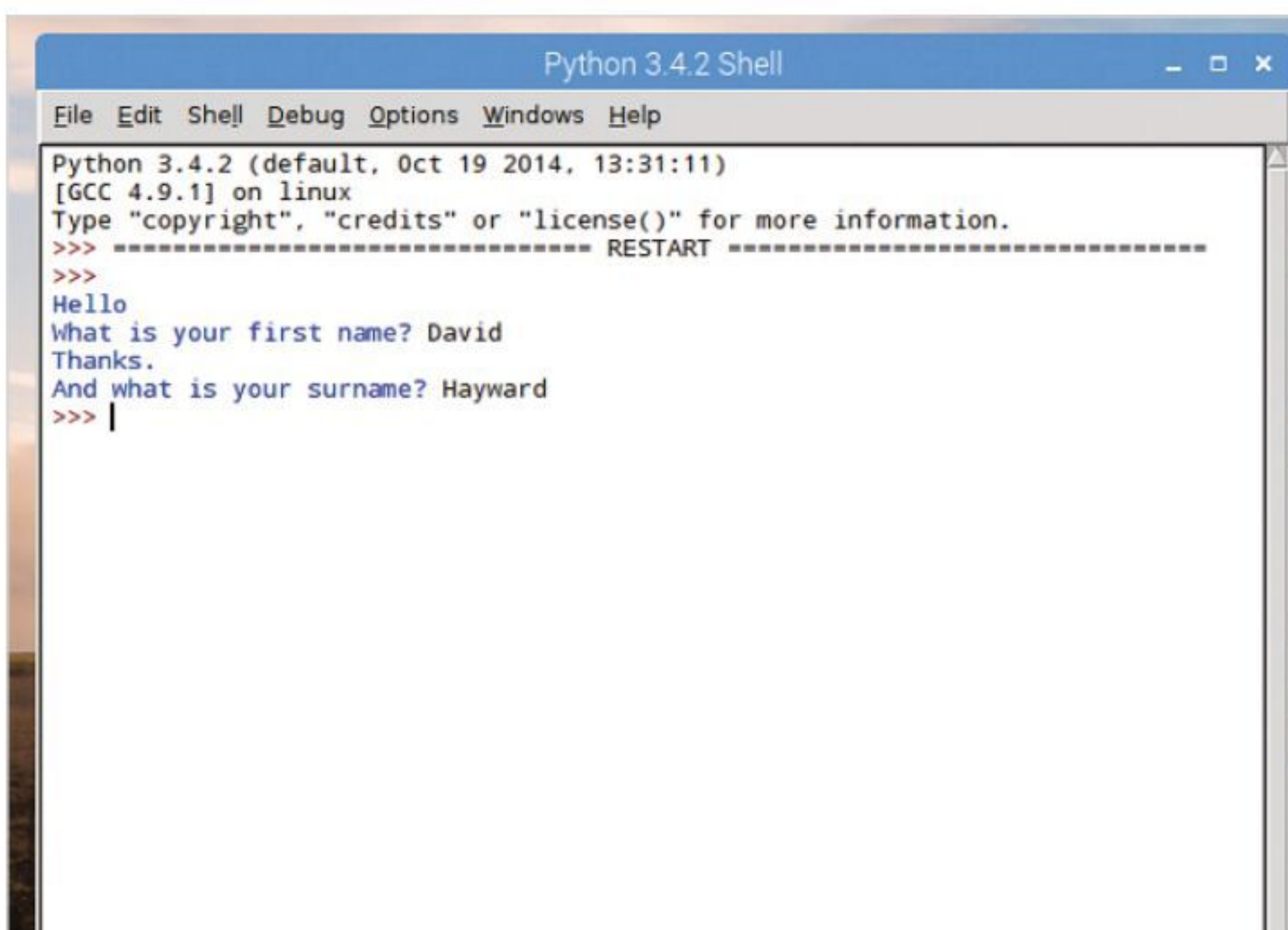


STEP 3 Now that we have the user's name stored in a couple of variables we can call them up whenever we want:

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

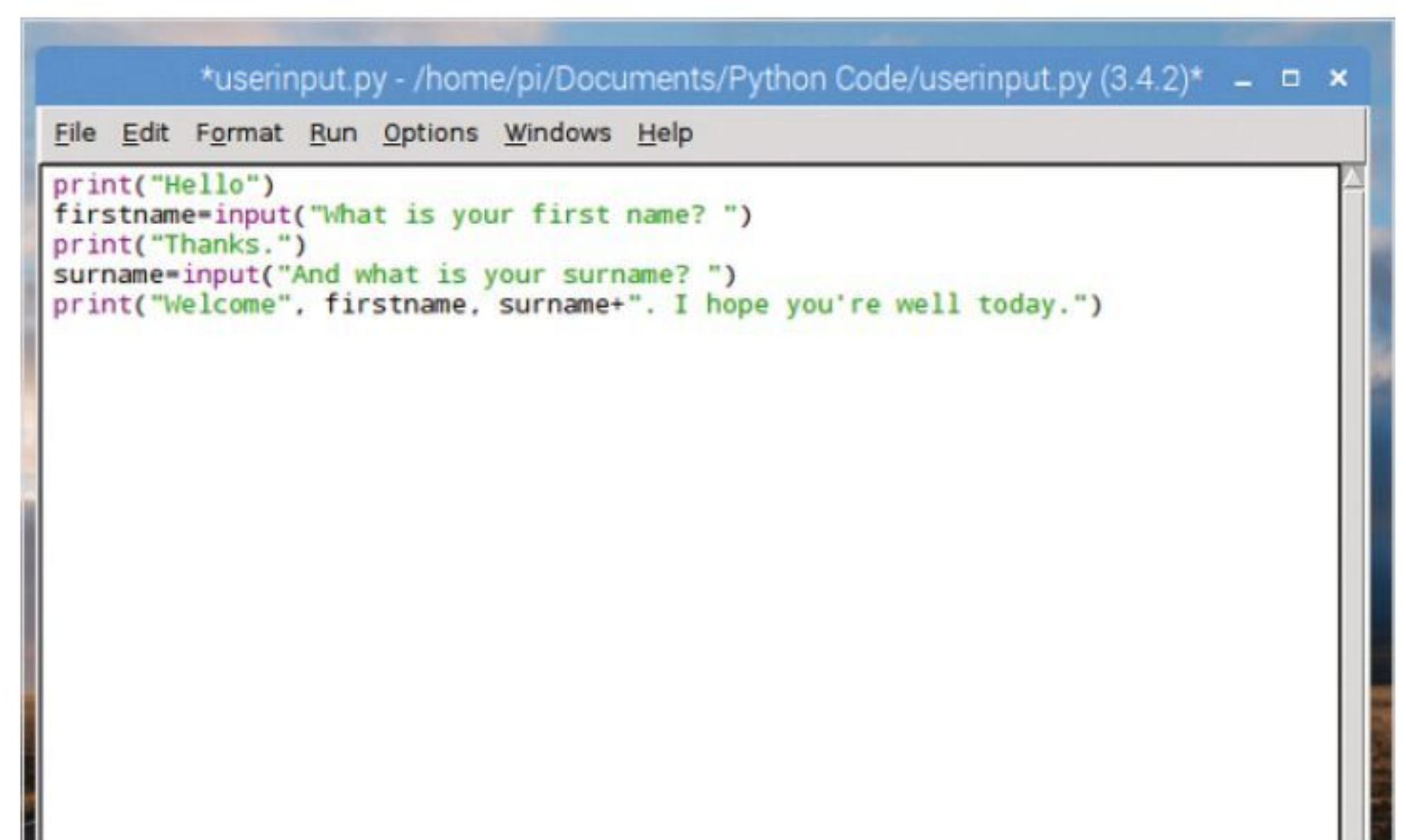


STEP 2 Save and execute the code, and as you already no doubt suspected, in the IDLE Shell the program will ask for your first name, storing it as the variable `firstname`, followed by your surname; also stored in its own variable (`surname`).



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

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



**STEP 5**

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

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

STEP 8

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

STEP 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 9

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

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

STEP 7

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

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

STEP 10

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

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

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



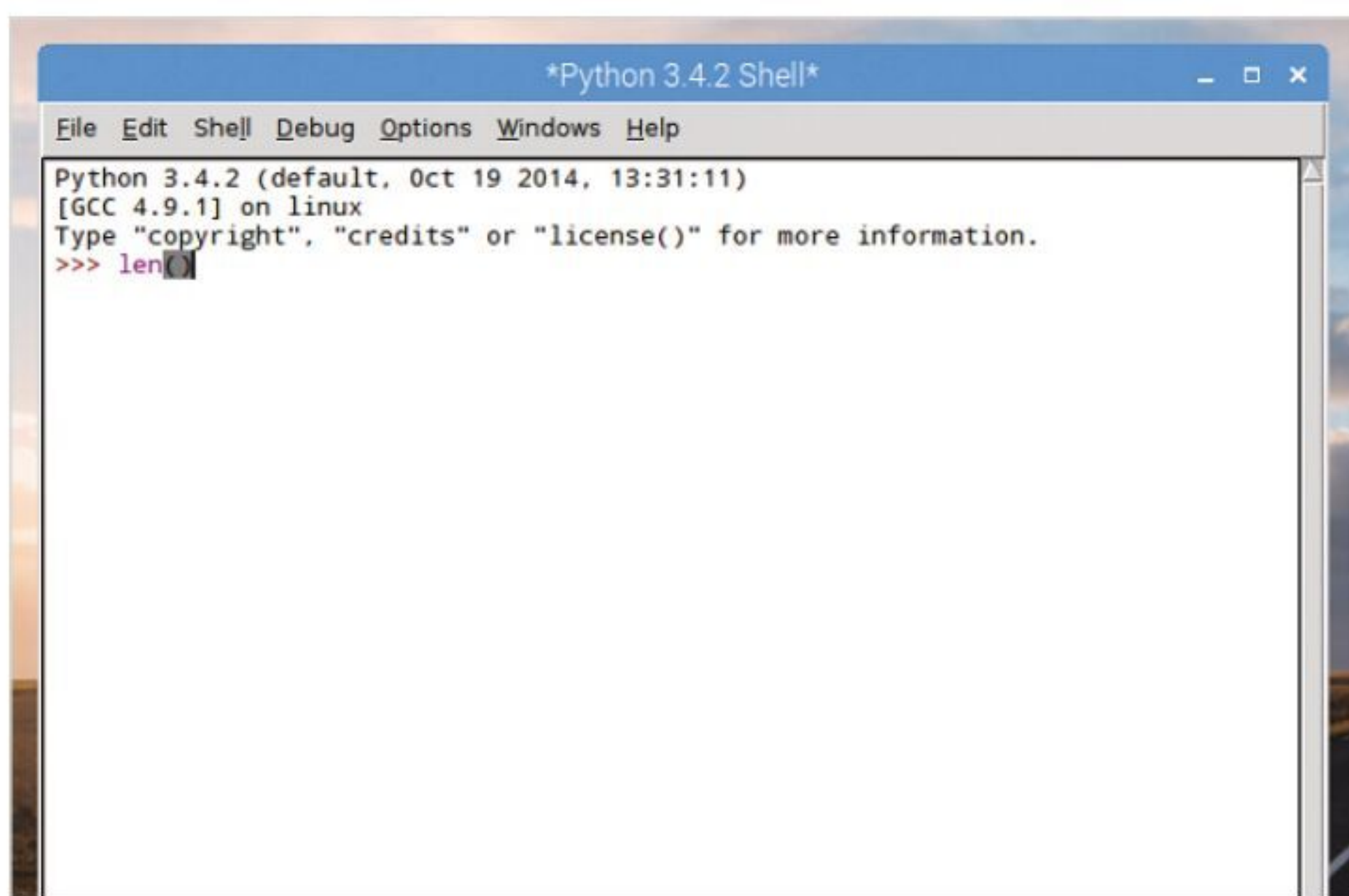
Creating Functions

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

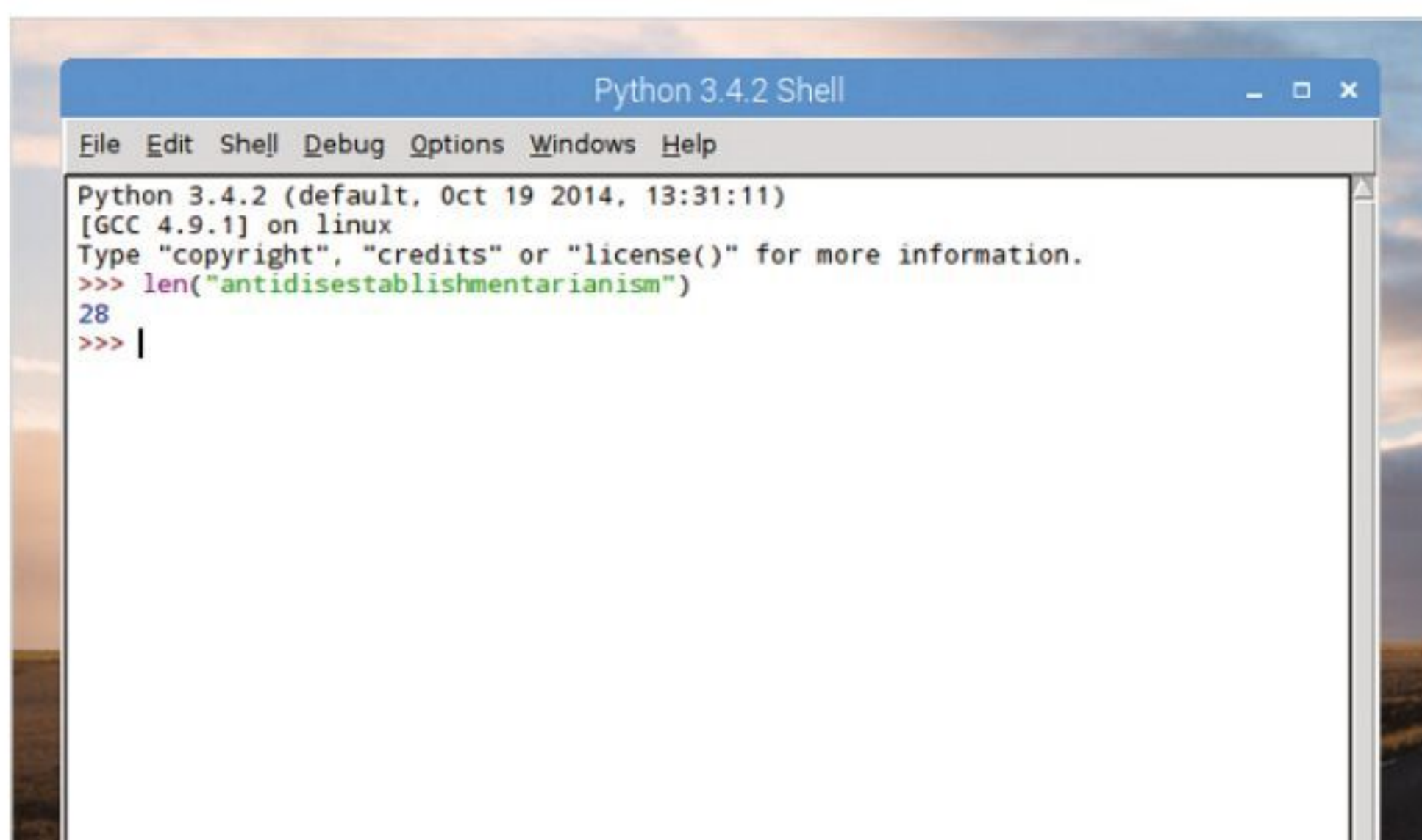
FUNKY FUNCTIONS

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

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



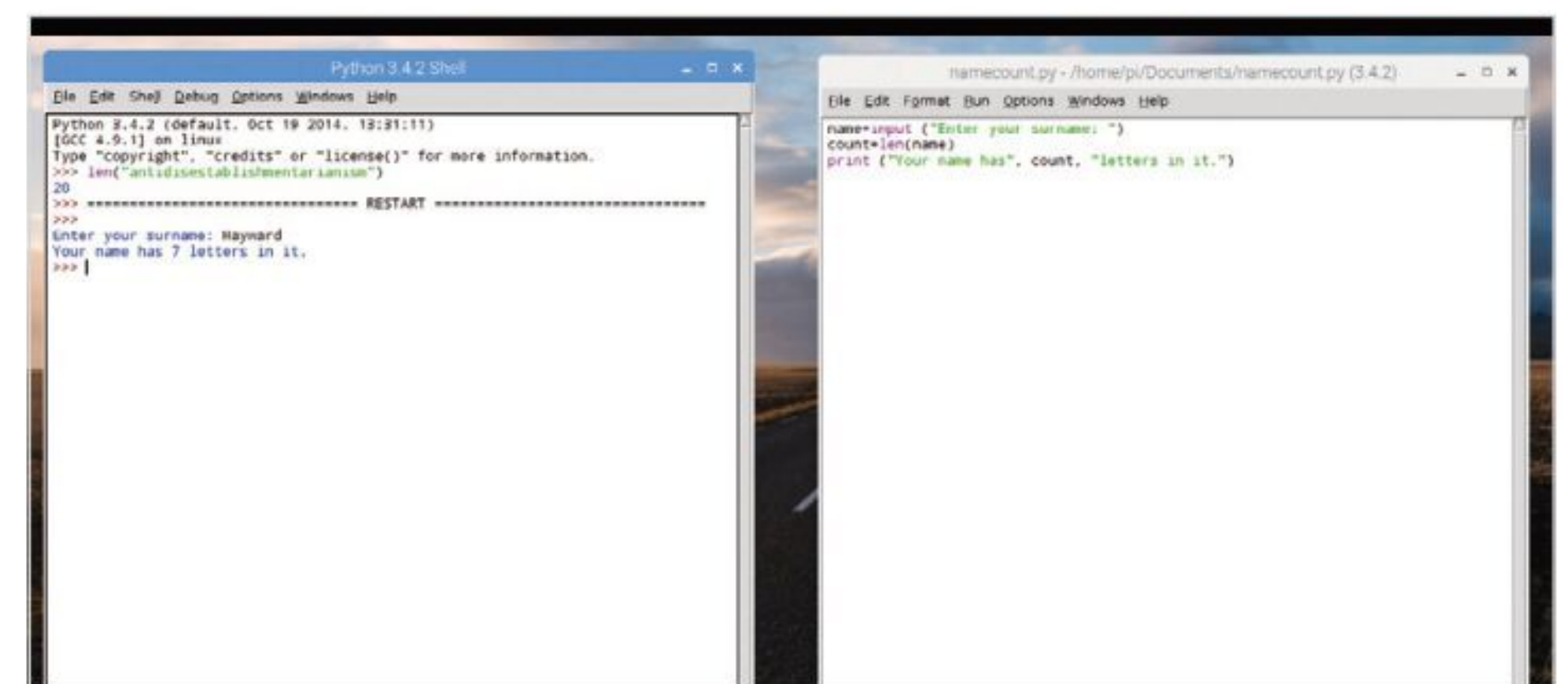
STEP 2 A function takes data, usually a variable, works on it depending on what the function is programmed to do and returns the end value. The data being worked on goes inside the brackets, so if you wanted to know how many letters are in the word **antidisestablishmentarianism**, then you'd enter: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



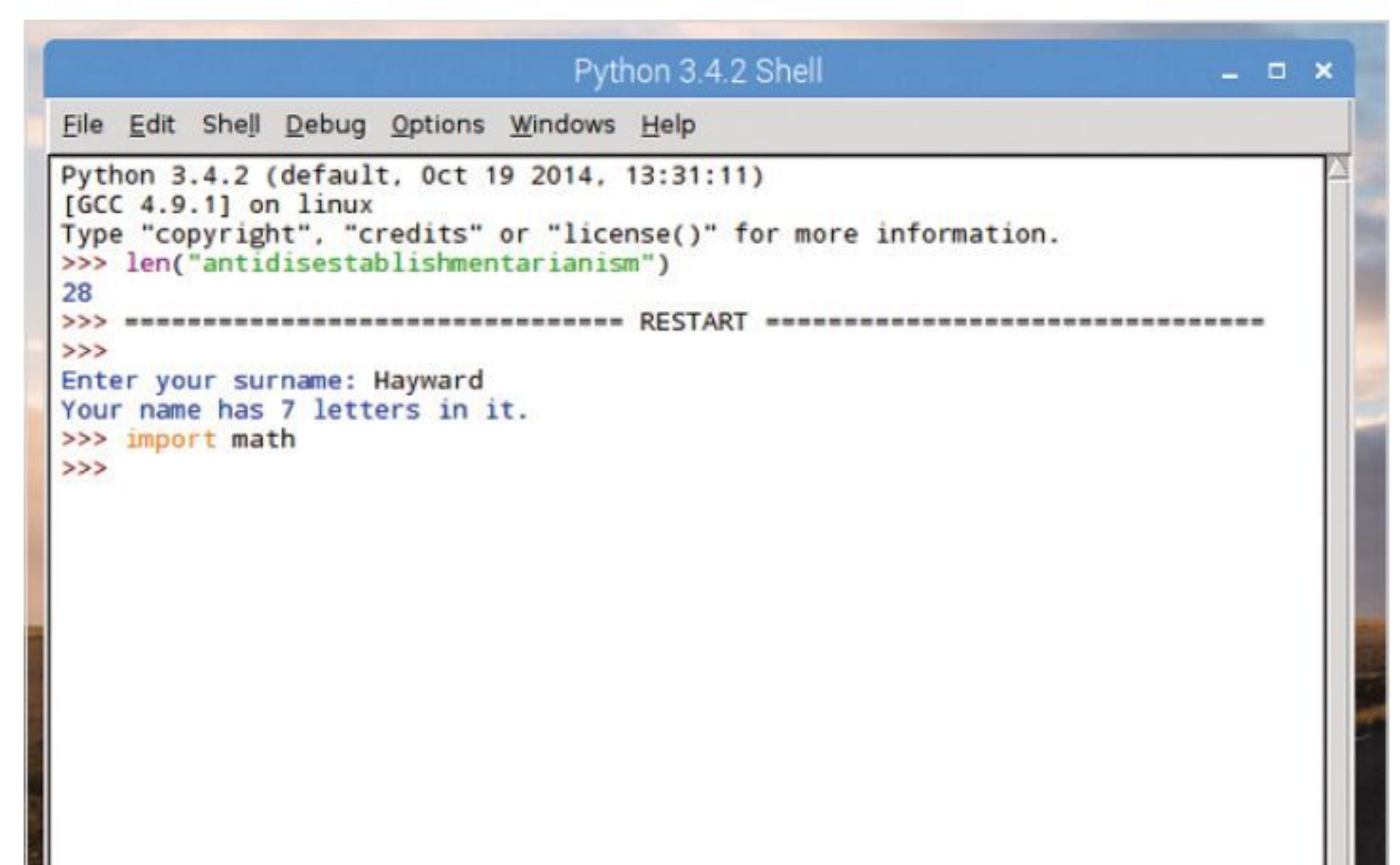
STEP 3 You can pass variables through functions in much the same manner. Let's assume you want the number of letters in a person's surname, you could use the following code (enter the text editor for this example):

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

Press F5 and save the code to execute it.



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

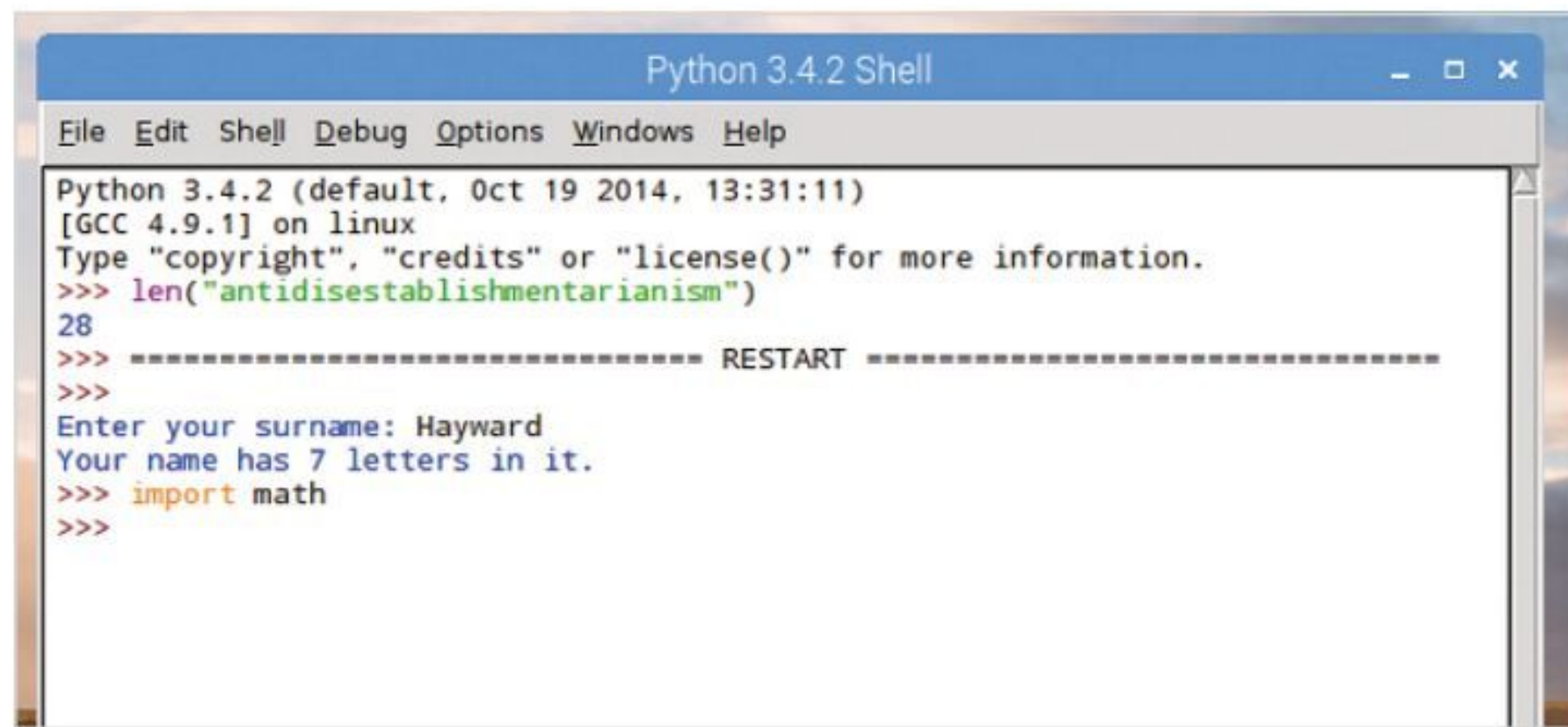




STEP 5 Additional functions can be added to Python through modules. Python has a vast range of modules available that can cover numerous programming duties. They add functions and can be imported as and when required. For example, to use advanced Mathematics functions enter:

```
import math
```

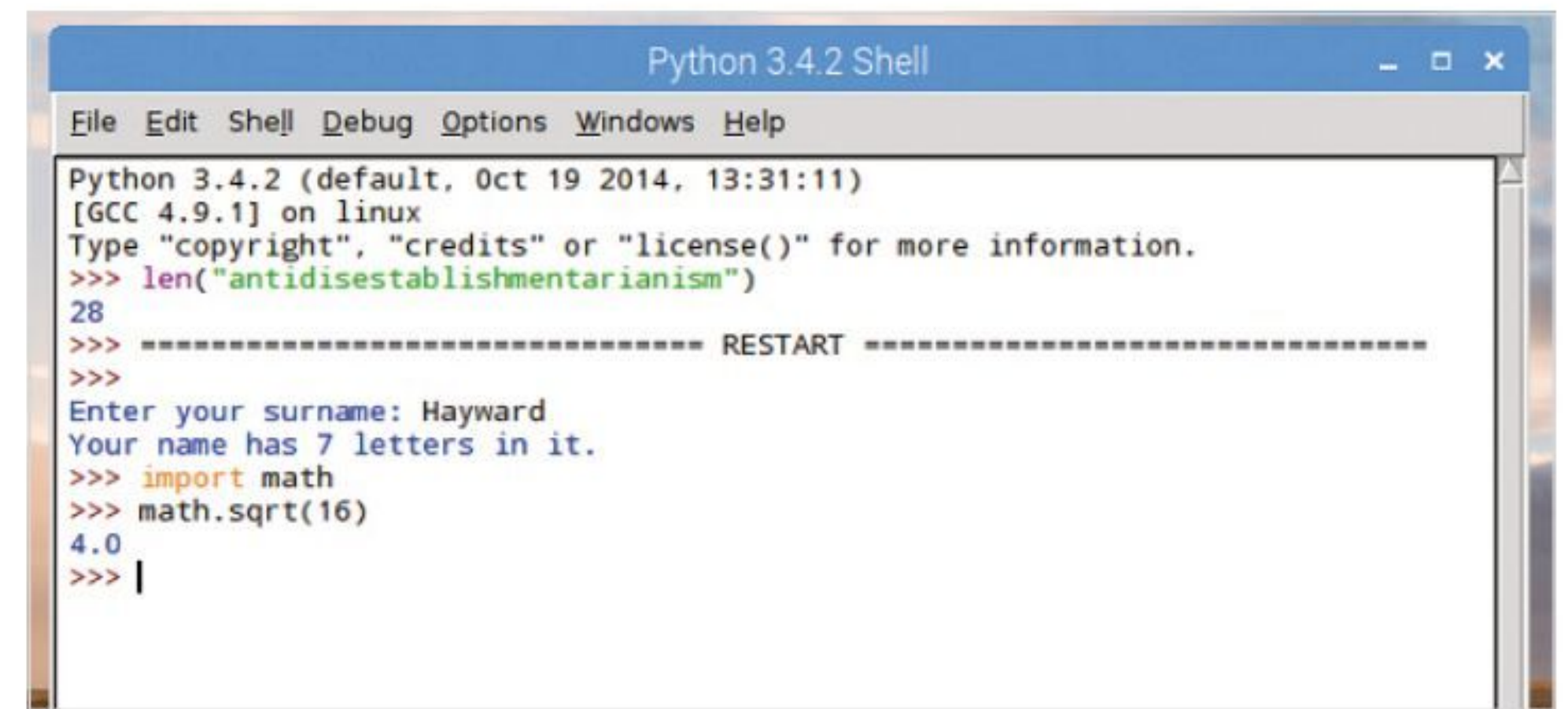
Once entered, you have access to all the Math module functions.



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

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

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



FORGING FUNCTIONS

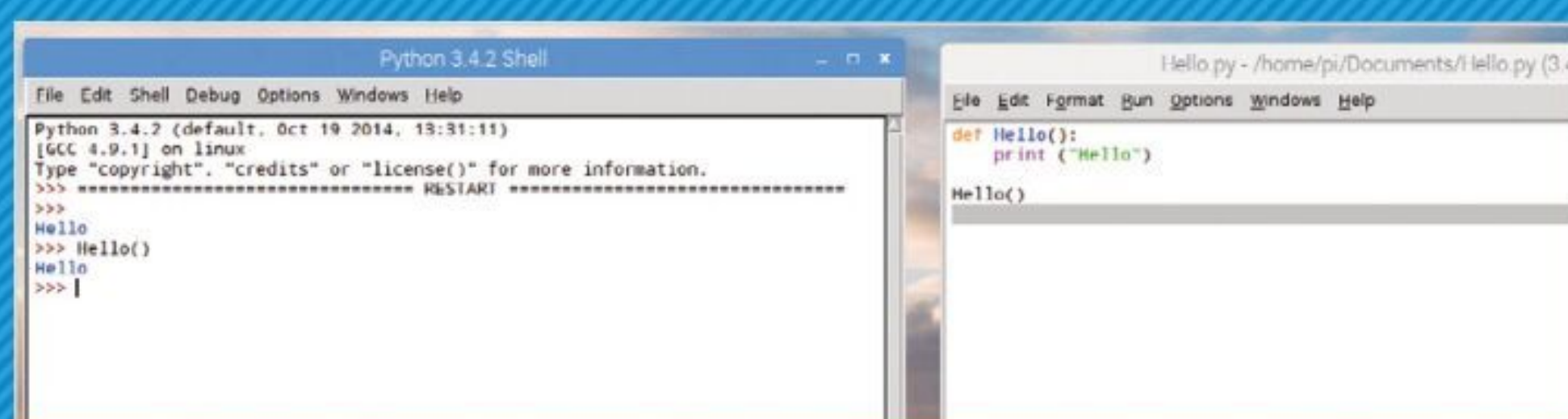
There are many different functions you can import created by other Python programmers and you will undoubtedly come across some excellent examples in the future; you can also create your own with the def command.

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

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

```
Hello()
```

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

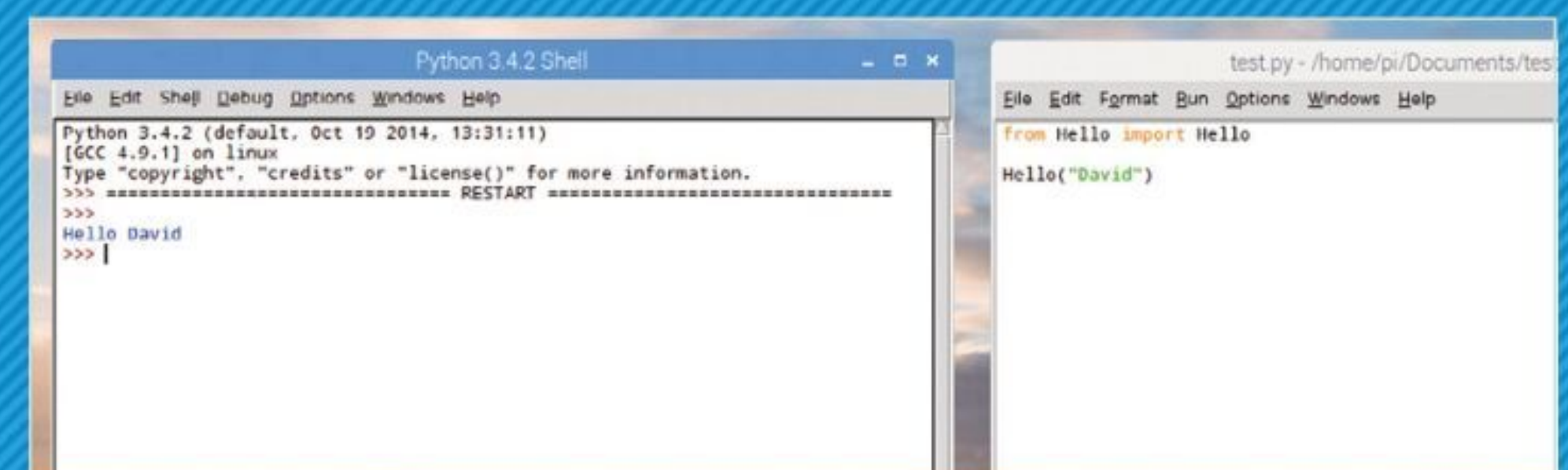


STEP 3 To modify it further, delete the Hello("David") line, the last line in the script and press Ctrl+S to save the new script. Close the Editor and create a new file (File > New File). Enter the following:

```
from Hello import Hello
```

```
Hello("David")
```

Press F5 to save and execute the code.

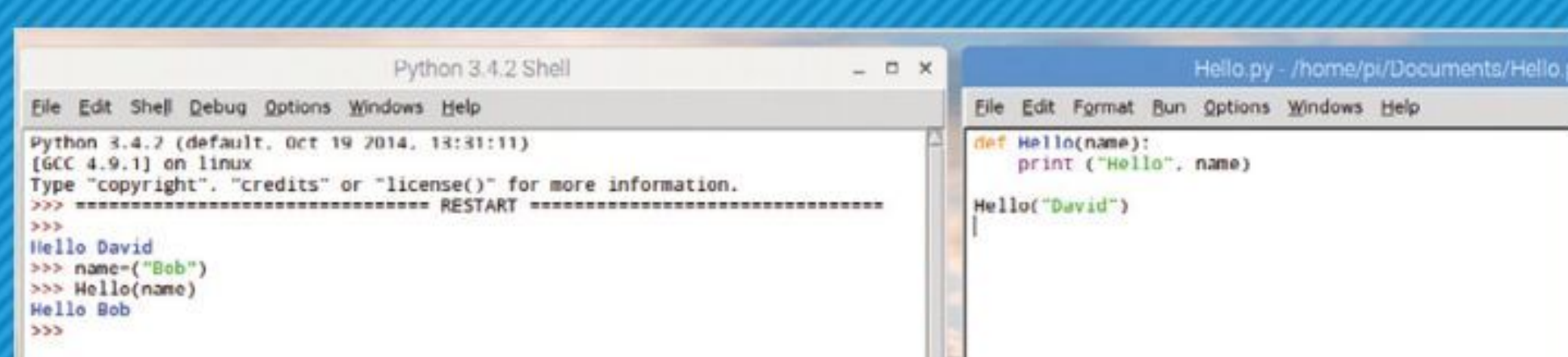


STEP 2 Let's now expand the function to accept a variable, the user's name for example. Edit your script to read:

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

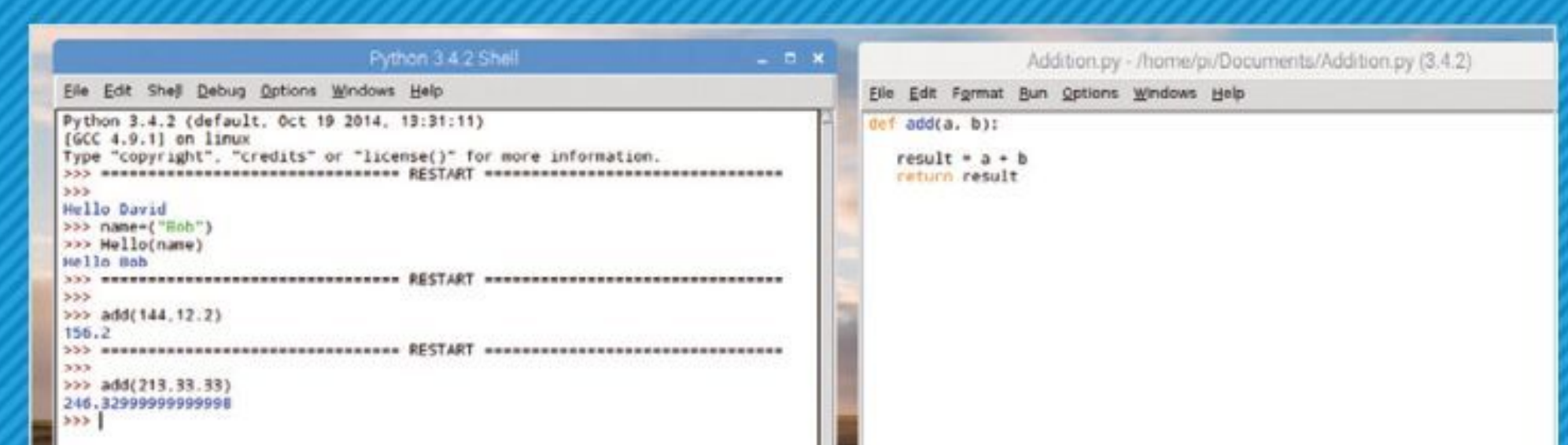
```
Hello("David")
```

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



STEP 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

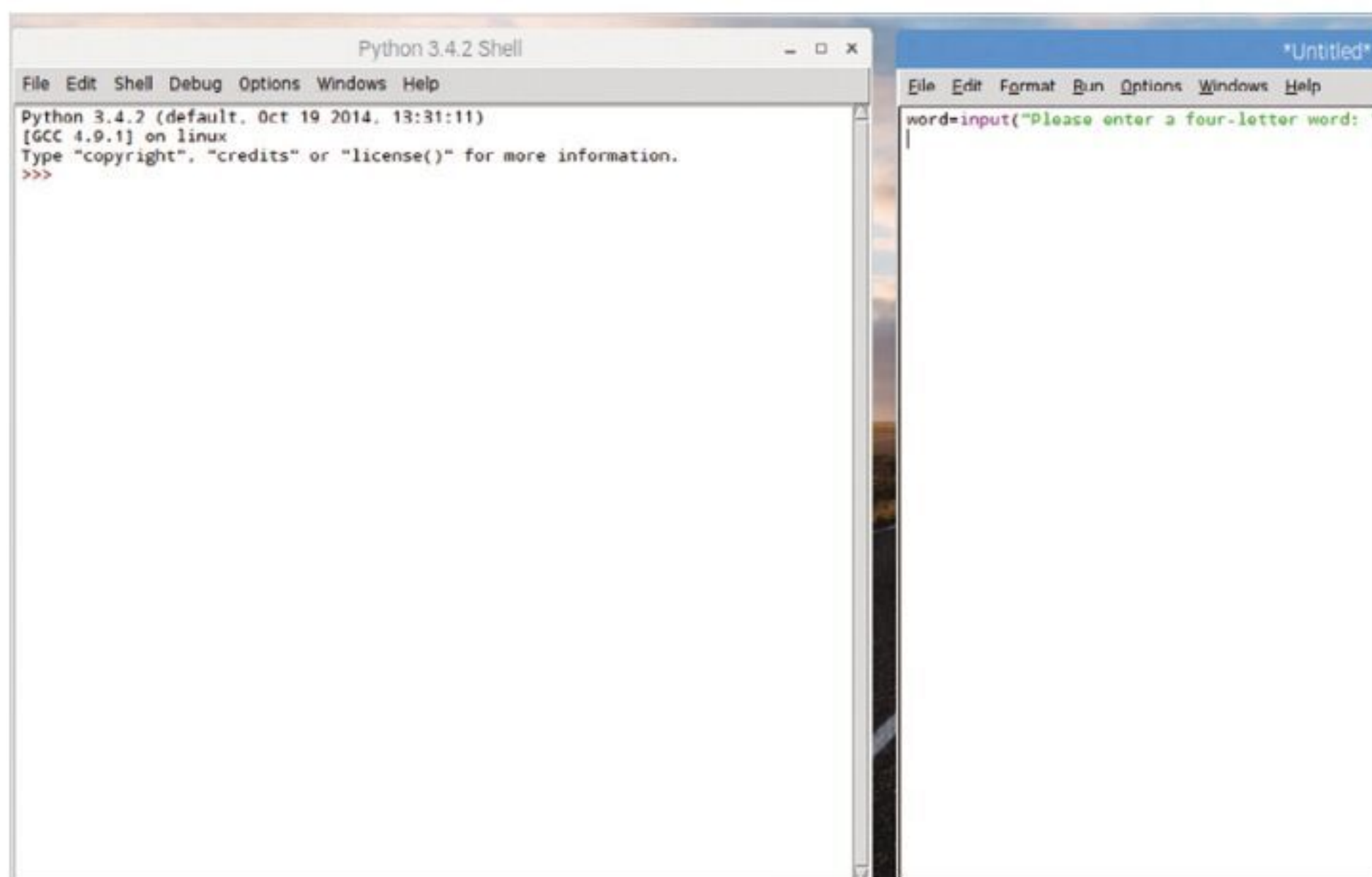
Conditions and loops are what make a program interesting; they can be simple or rather complex. How you use them depends greatly on what the program is trying to achieve; they could be the number of lives left in a game or just displaying a countdown.

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.

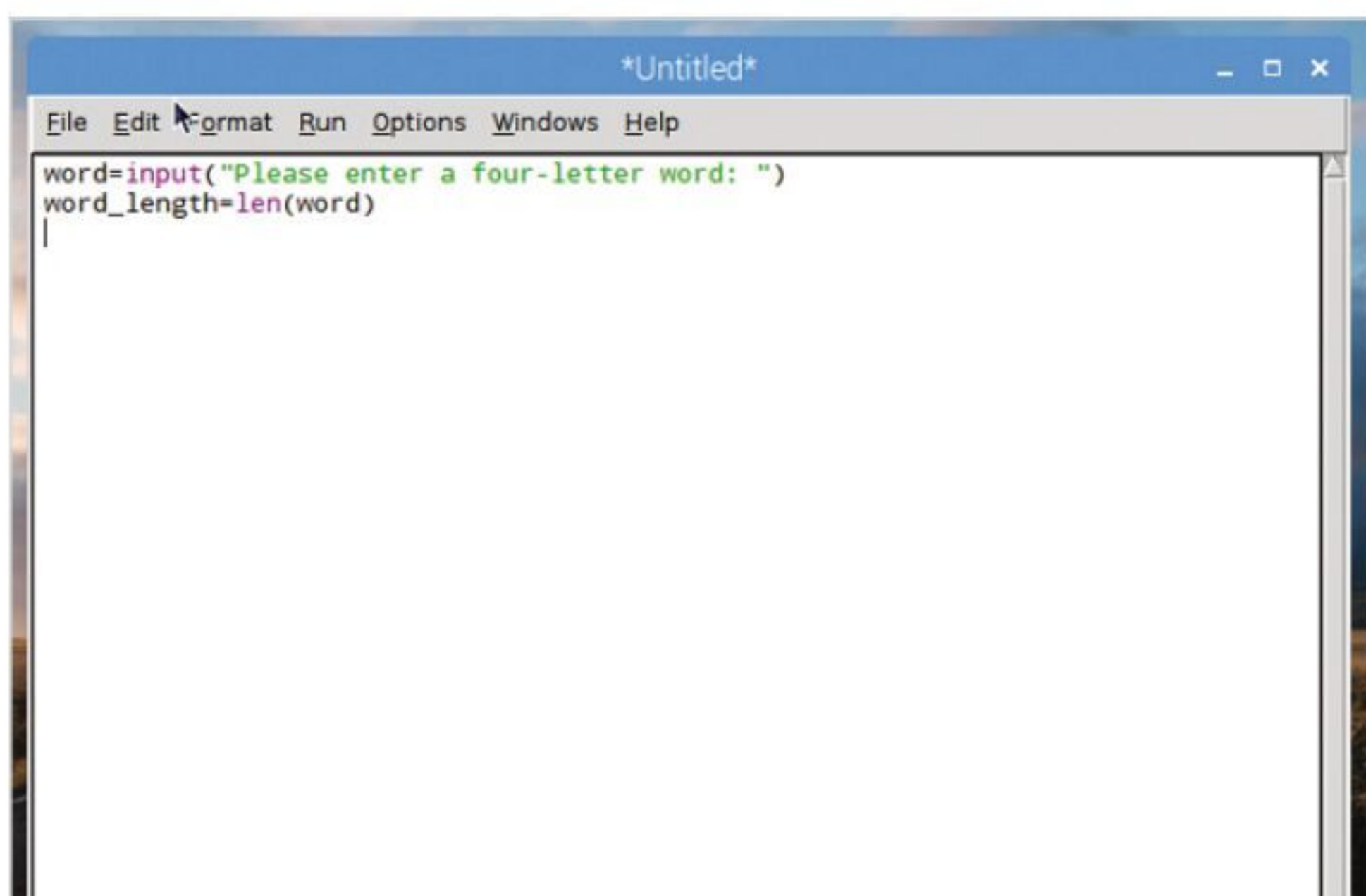
STEP 1 Let's create a new Python program that will ask the user to input a word, then check it to see if it's a four-letter word or not. Start with File > New File, and begin with the input variable:

```
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:

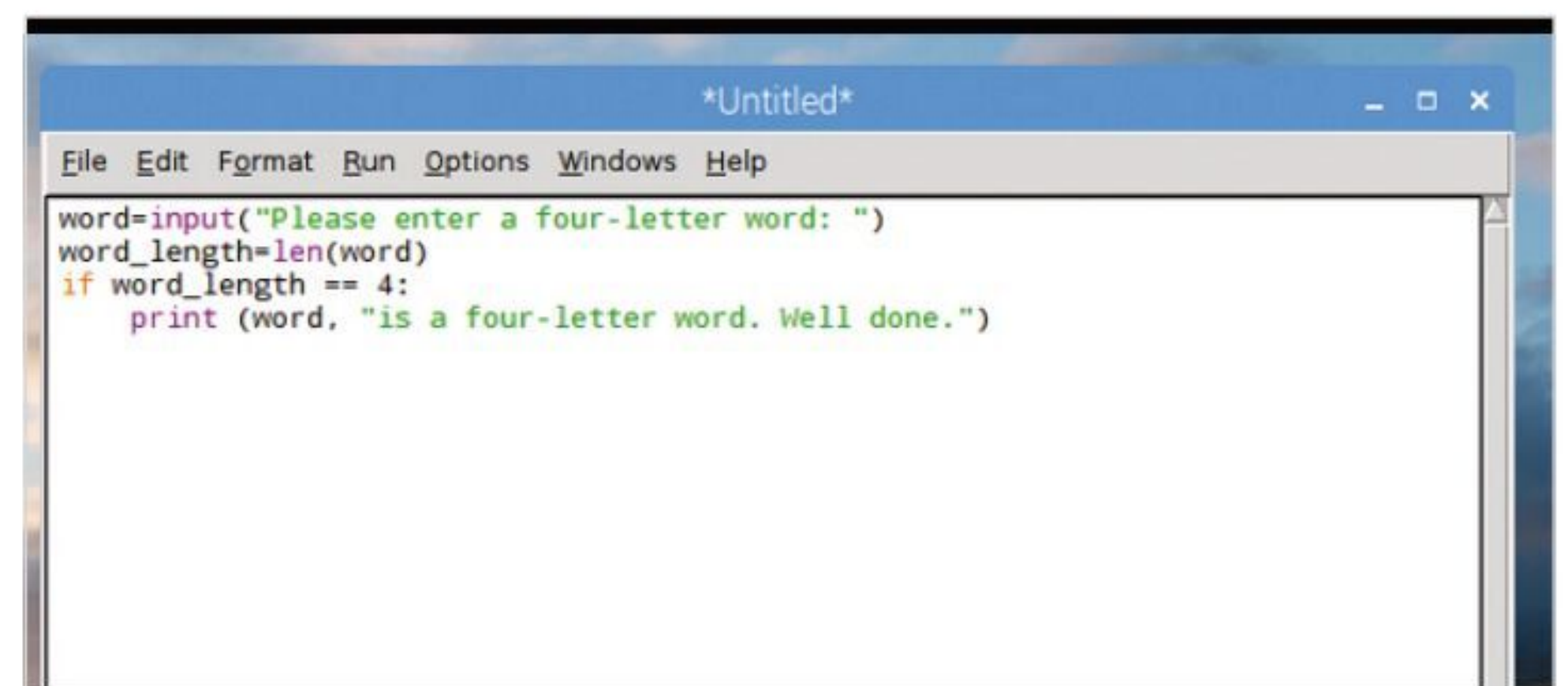
```
word=input("Please enter a four-letter word: ")
word_length=len(word)
```



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

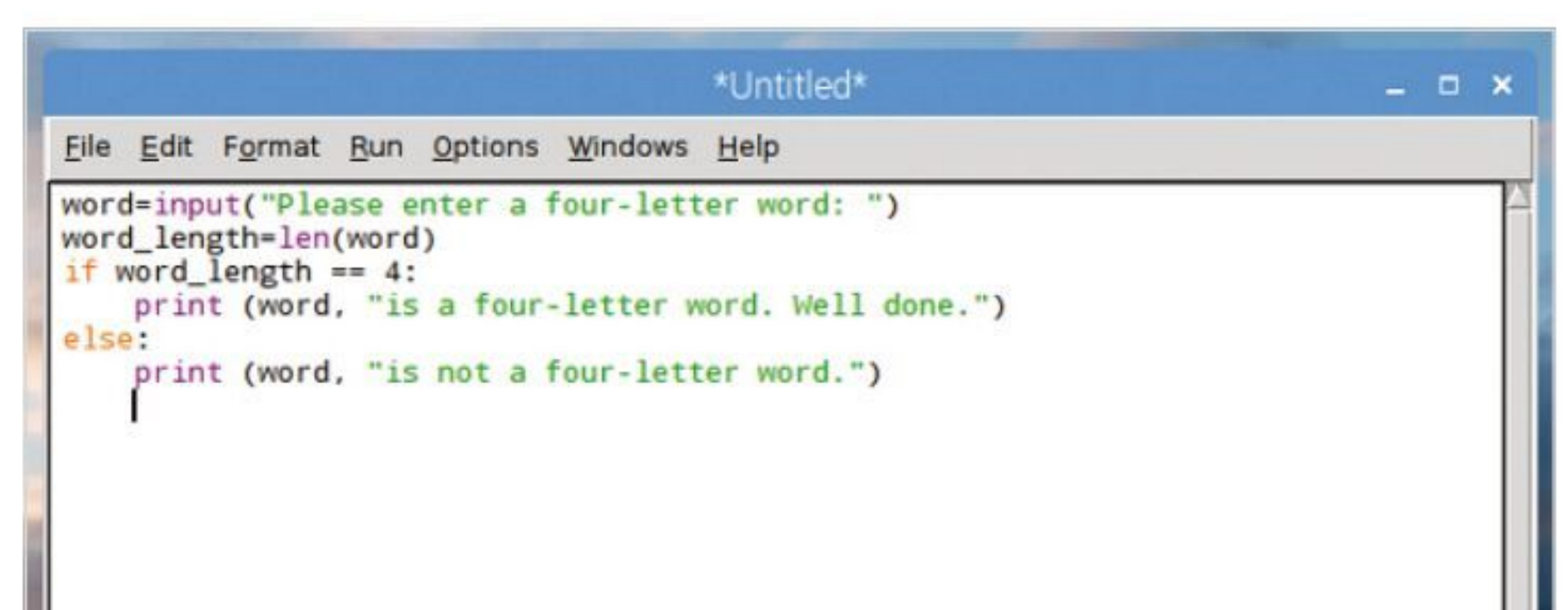
```
word=input("Please enter a four-letter word: ")
word_length=len(word)
if word_length == 4:
    print (word, "is a four-letter word. Well done.")
```

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



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

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





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

```

Python 3.4.2 Shell
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>>
Please enter a four-letter word: word
Word is a four-letter word. Well done.
>>>
Please enter a four-letter word: Frost
Frost is not a four-letter word.
>>>

wordgame.py - /home/pi/Documents/wordgam
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 6 Now expand the code to include another conditions. Eventually, it could become quite complex. We've added a condition for three-letter words:

```

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

```

Python 3.4.2 Shell
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>>
Please enter a four-letter word: word
Word is a four-letter word. Well done.
>>>
Please enter a four-letter word: Frost
Frost is not a four-letter word.
>>>
Please enter a four-letter word: Egg
Egg is a three-letter word. Try again.
>>>

wordgame.py - /home/pi/Documents/wordgame.py (3.4.2)
word=input("Please enter a four letter word: ")
word_length=len(word)
if word_length == 4:
    print (word, "is a four-letter word. Well done.")
elif word_length == 3:
    print (word, "is a three-letter word. Try again.")
else:
    print (word, "is not a four-letter word.")
    
```

LOOPS

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

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

```

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

```

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

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

```

Python 3.4.2 Shell
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>>
1
2
3
4
5
6
7
8
9
>>>

loop1.py - /home/pi/D
x=1
while x<10:
    print (x)
    x=x+1
    
```

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)
    
```

```

Python 3.4.2 Shell
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>>
1
2
3
4
5
6
7
8
9
>>>
cat
dog
unicorn
>>>

loop1.py - /home/pi/Documents/loop
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 3.4.2 Shell
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>>
1
2
3
4
5
6
7
8
9
>>>

loop1.py - /home/pi/Documen
for x in range (1, 10):
    print (x)
    
```



Python 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 the Pi has most of them pre-installed; as do many Linux distros.

MASTERING MODULES

In this instance we're using Windows 10 to demonstrate installing modules, since they're already there in Pi OS. The same steps apply for those Linux distro that don't feature any modules, so don't worry about cross platform compatibility.

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`.

```
Python 3.6.2 Shell
File Edit Shell Debug Options Window Help
Python 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:14:34) [MSC v.1900 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>> import math
>>>
```

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

```
Python 3.6.2 Shell
File Edit Shell Debug Options Window Help
Python 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:14:34) [MSC v.1900 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>> import math
>>> import pygame
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    import pygame
ModuleNotFoundError: No module named '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`

```
Command Prompt
C:\Users\David>pip install pygame
```

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

```
Administrator: Command Prompt
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.
C:\WINDOWS\system32>pip install pygame
Collecting pygame
  Using cached pygame-1.9.3-cp36-cp36m-win32.whl
Installing collected packages: pygame
Successfully installed pygame-1.9.3
C:\WINDOWS\system32>
```

**STEP 5**

Close the command prompt or Terminal and relaunch the IDLE Shell. When you now enter:

`import pygame`, the module will be imported into the code without any problems. You'll find that most code downloaded or copied from the Internet will contain a module, mainstream or unique, these are usually the source of errors in execution due to them being missing.

```
Python 3.6.2 Shell
File Edit Shell Debug Options Window Help
Python 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:14:34) [MSC v.1900 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>> import pygame
>>>
```

STEP 6

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

```
import random
```

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

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

```
*Untitled*
File Edit Format Run Options Window Help
import random

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

STEP 7

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

```
import random
```

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

```
Python 3.6.2 Shell
File Edit Shell Debug Options Window Help
>>>
***** RESTART: C:/Users/david/Documents/Python/Rnd Number.py *****
24
21
9
17
22
5
8
5
10
13
>>>
***** RESTART: C:/Users/david/Documents/Python/Rnd Number.py *****
26
11
17
65
37
22
37
35
89
54
42
48
94
28
```

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)
```

```
Rnd Number.py - C:/Users/david/Documents/Python/Rnd Number.py (3.6.2)
File Edit Format Run Options Window Help
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))
```

```
Rnd Number.py - C:/Users/david/Documents/Python/Rnd Number.py (3.6.2)
File Edit Format Run Options Window Help
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.

```
*Rnd Number.py - C:/Users/david/Documents/Python/Rnd Number.py (3.6.2)*
File Edit Format Run Options Window Help
import math as m

print(m.pi)
```



Admiral Grace Hopper, debugging computers since the '40s



Debugging

DID YOU KNOW...

that 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 stopping the system from working. Removal of the moth was hence called debugging.



“The most important single aspect of software development is to be clear about what you are trying to build.”

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



C++ on Linux

C++ is an excellent, high-level programming language that's used in a multitude of technologies. Everything from your favourite mobile app, console and PC game to entire operating systems are developed with C++ at the core, together with a collection of software development kits and custom libraries.

C++ is the driving force behind most of what you use on a daily basis, which makes it a complex and extraordinarily powerful language to get to grips with. It's the code behind Linux itself, as well as most of the behind the scenes drivers, library files and control elements of most, if not all, operating systems.

This section shows you how to start coding in C++ on Linux using a Raspberry Pi. From there, you can begin to create great things.



Why C++?

C++ is one of the most popular programming languages available today. Originally called C with Classes, the language was renamed C++ in 1983. It's an extension of the original C language and is a general purpose object-oriented (OOP) environment.

C EVERYTHING

Due to how complex the language can be, and its power and performance, C++ is often used to develop games, programs, device drivers and even entire operating systems.

Dating back to 1979, the start of the golden era of home computing, C++, or rather C with Classes, was the brainchild of Danish computer scientist Bjarne Stroustrup while working on his PhD thesis. Stroustrup's plan was to further the original C language, which was widely used since the early seventies.

C++ proved to be popular among the developers of the '80s, since it was a much easier environment to get to grips with and more importantly, it was 99% compatible with the original C language. This meant that it could be used beyond the mainstream

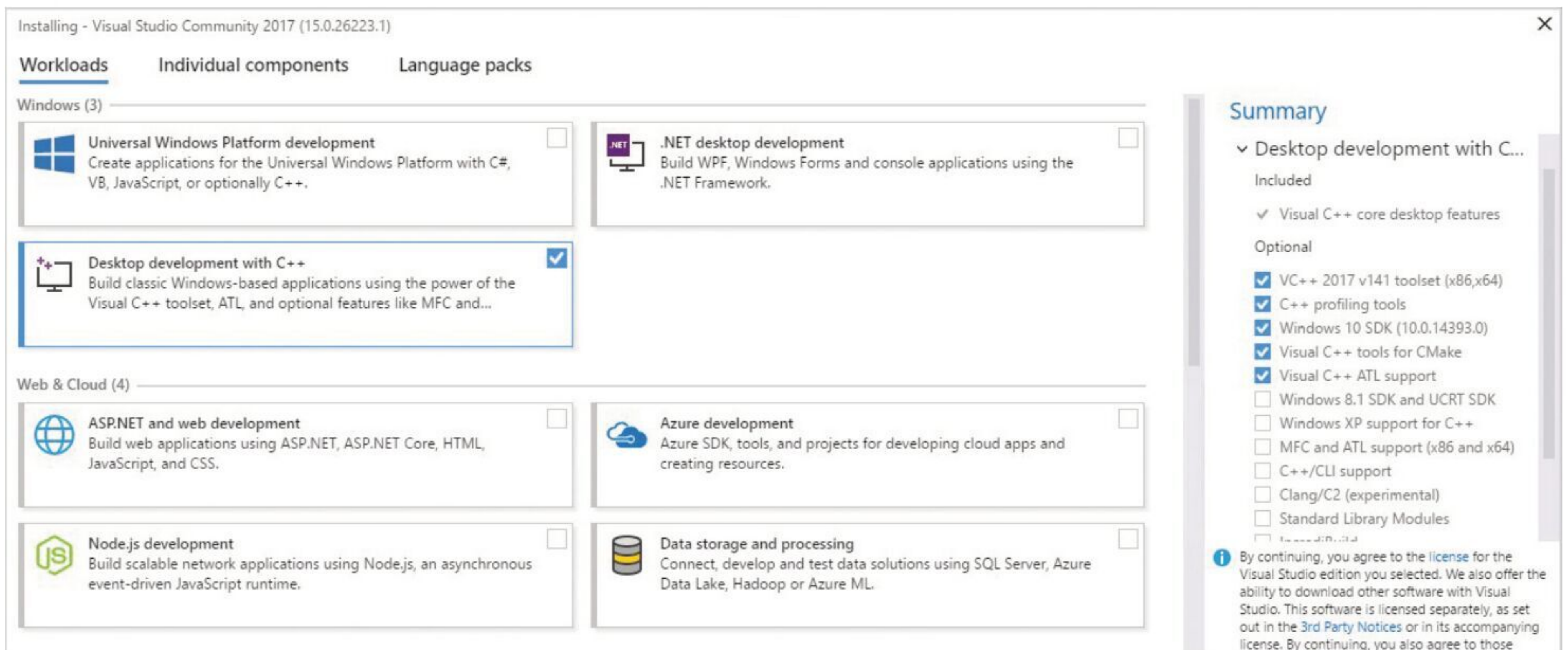
computing labs and by regular people who didn't have access to the mainframes and large computing data centres.

C++'s impact in the digital world is immense. Many of the programs, applications, games and even operating systems are coded using C++. For example, all of Adobe's major applications, such as Photoshop, InDesign and so on, are developed in C++. You will find that the browser you surf the Internet with is written in C++, as well as Windows 10, Microsoft Office and the backbone to Google's search engine. Apple's macOS is written largely in C++ (with some



C++ code is much faster than that of Python.

```
1  #include<iostream>
2  using namespace std;
3  void main()
4  {char ch;
5  cout<<"Enter a charater to check it is vowel or not";
6  cin>>ch;
7      switch(ch)
8      {
9          case 'a': case 'A':
10         cout<<ch<<" is a Vowel";
11         break;
12         case 'e': case 'E':
13         cout<<ch<<" is a Vowel";
14         break;
15         case 'i': case 'I':
16         cout<<ch<<" is a Vowel";
17         break;
18         case 'o': case 'O':
19         cout<<ch<<" is a Vowel";
20         break;
21         case 'u': case 'U':
22         cout<<ch<<" is a Vowel";
23         break;
24         default:
```

 **Microsoft's Visual Studio is a great, free environment to learn C++ in.**

other languages mixed in depending on the function) and the likes of NASA, SpaceX and even CERN use C++ for various applications, programs, controls and umpteen other computing tasks.

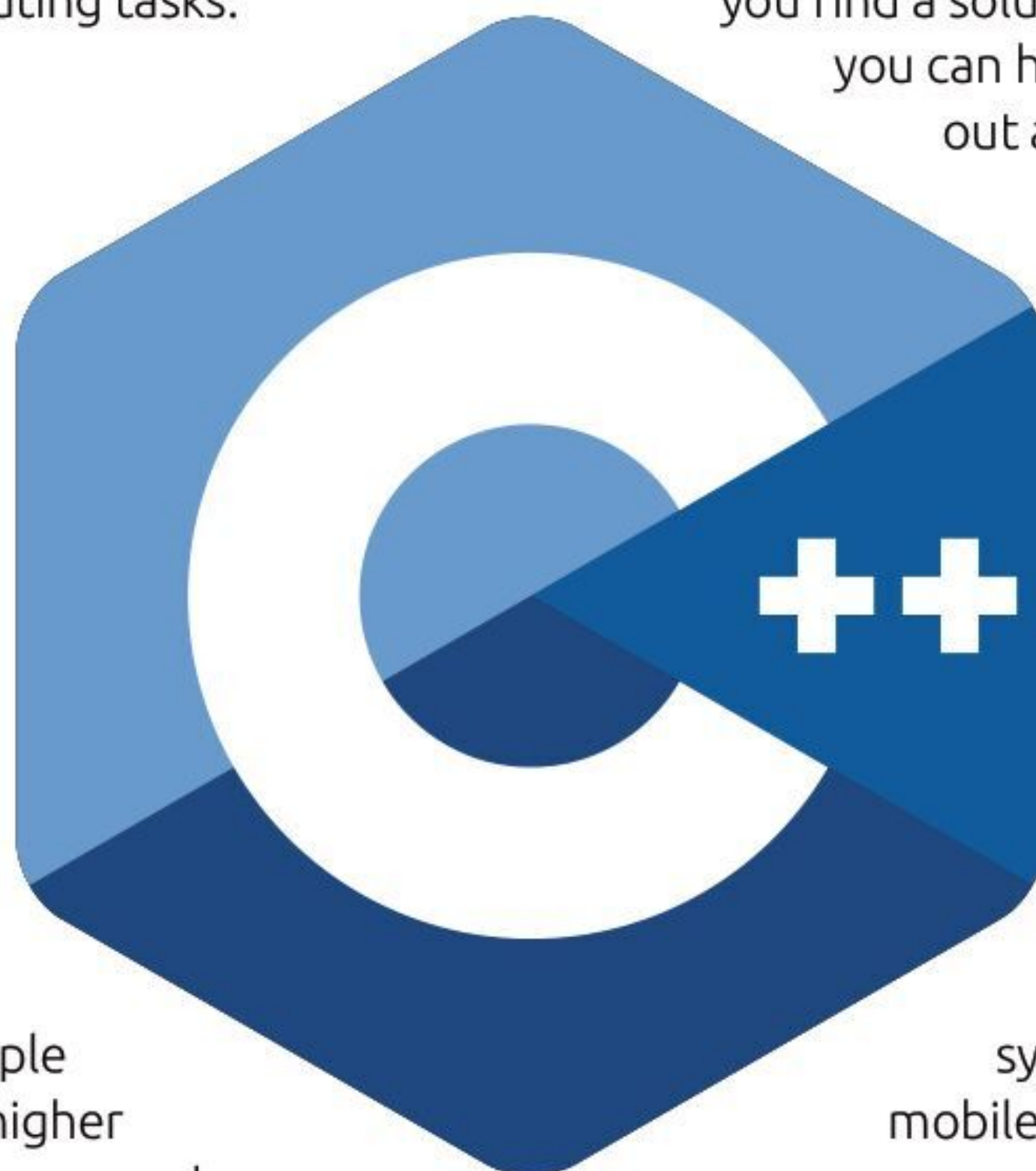
C++ is also extremely efficient and performs well across the board as well as being an easier addition to the core C language. This higher level of performance over other languages, such as Python, BASIC and such, makes it an ideal development environment for modern computing, hence the aforementioned companies using it so widely.

While Python is a great programming language to learn, C++ puts the developer in a much wider world of coding. By mastering C++, you can find yourself developing code for the likes of Microsoft, Apple and so on. Generally, C++ developers enjoy a higher salary than programmers of some other languages and due to its versatility, the C++ programmer can move between jobs and companies without the need to relearn anything specific. However, Python is an easier language to begin with. If you're completely new to programming then we would recommend you

begin with Python and spend some time getting to grips with programming structure and the many ways and means in which you find a solution to a problem through programming. Once you can happily power up your computer and whip out a Python program with one hand tied behind your back, then move on to C++. Of course, there's nothing stopping you from jumping straight into C++; if you feel up to the task, go for it.

Getting to use C++ is as easy as Python, all you need is the right set of tools in which to communicate with the computer in C++ and you can start your journey. A C++ IDE is free of charge, even the immensely powerful Visual Studio from Microsoft is freely available to download and use. You can get into C++ from any operating system, be it macOS, Linux, Windows or even mobile platforms.

Just like Python, to answer the question of Why C++ is the answer is because it's fast, efficient and developed by most of the applications you regularly use. It's cutting edge and a fantastic language to master.



 **Indeed, the operating system you're using is written in C++.**





Your First C++ Program

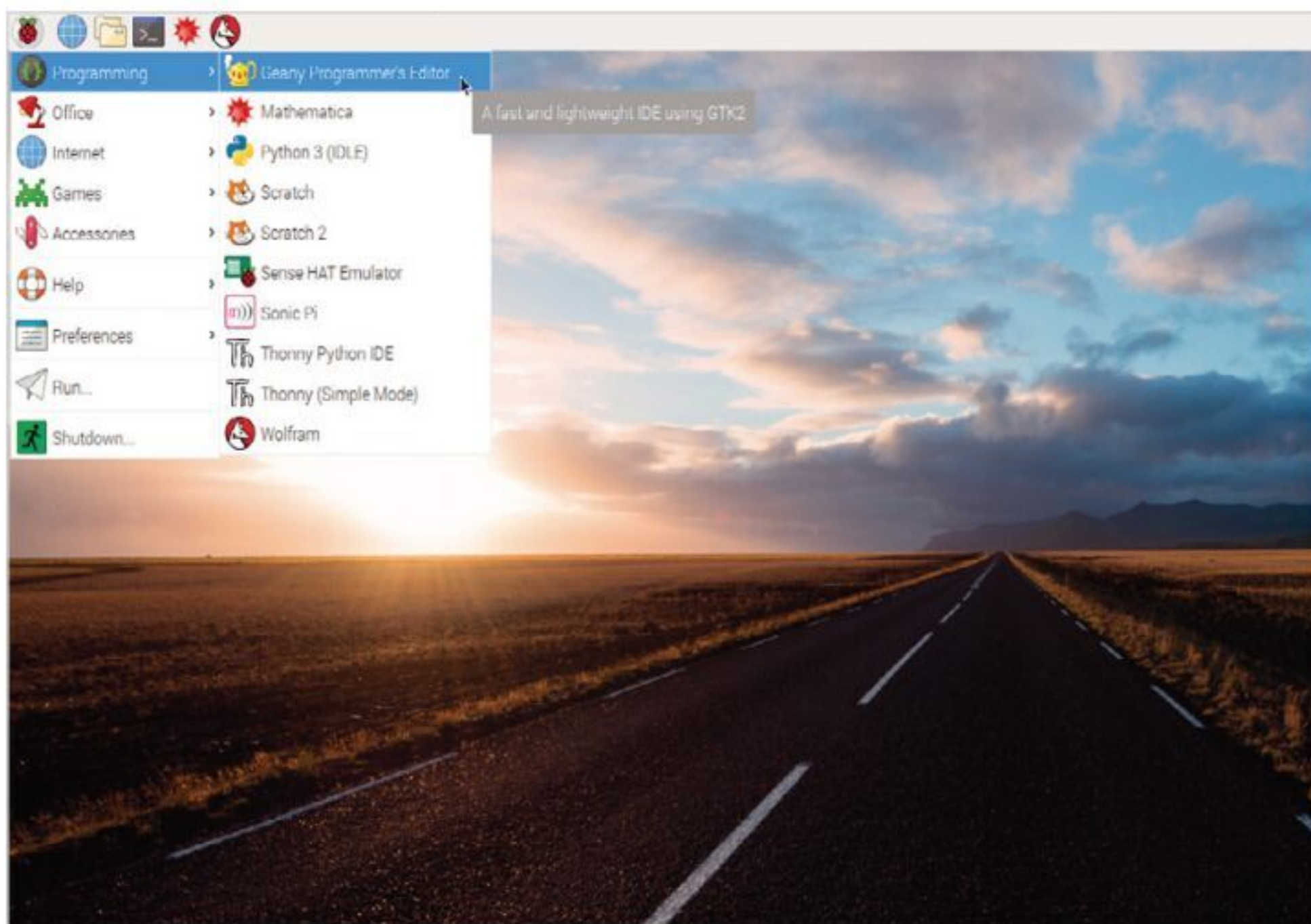
The Raspberry Pi makes for a great coding base on which to learn C++. You won't need any complicated, third-party IDEs or to install any extra components. Everything you need to get started is already built into the Pi.

HELLO, WORLD!

It's traditional in programming for the first code to be entered to output the words 'Hello, World!' to the screen. Interestingly, this dates back to 1968 using a language called BCPL.

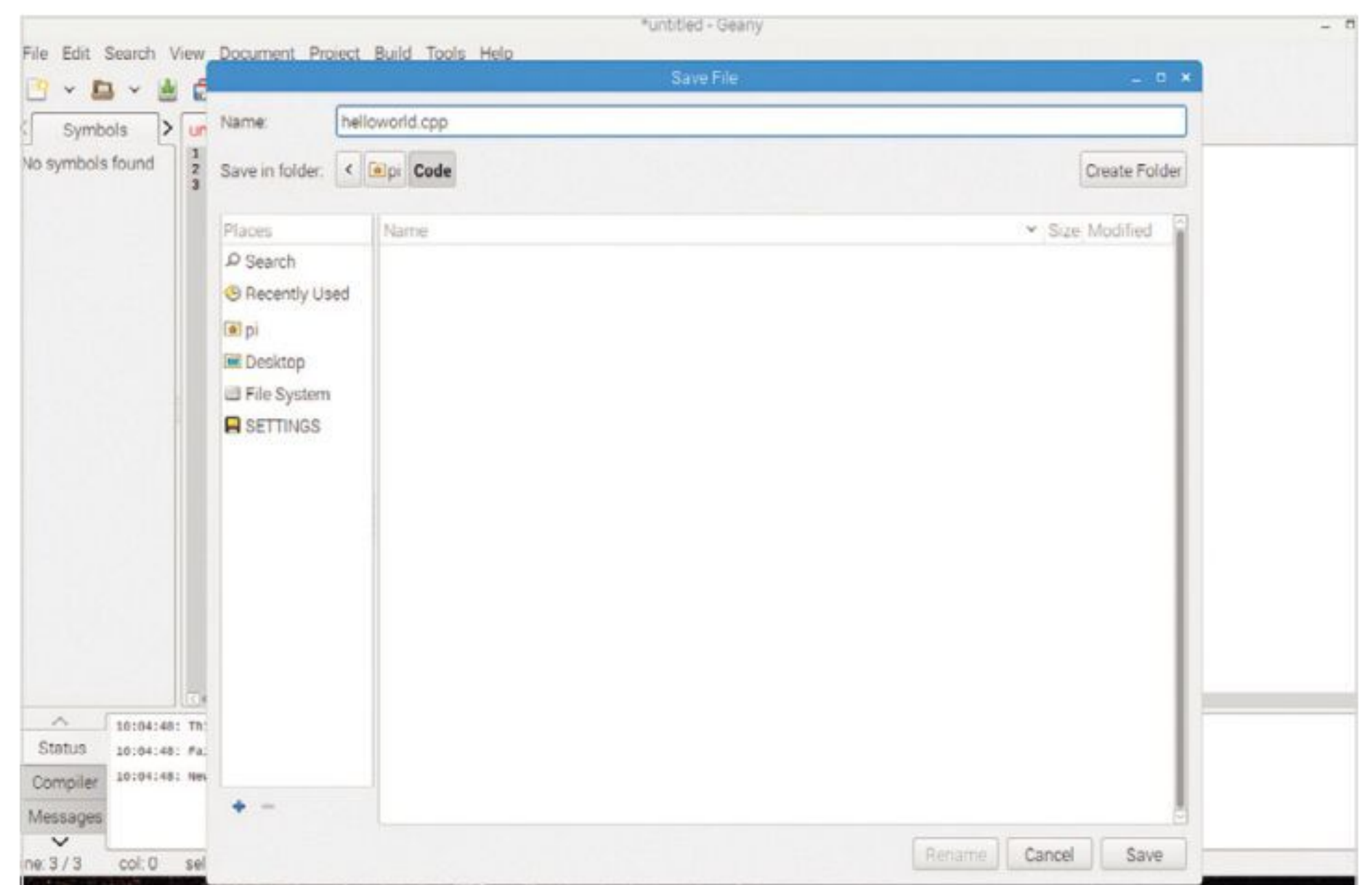
STEP 1

You can use the included Geany Programmer's Editor to write, compile and execute your C++ code on the Raspberry Pi. You can find Geany as the first entry in the Programming section from the main Raspberry Pi menu. Click the Geany icon to open and start coding.



STEP 3

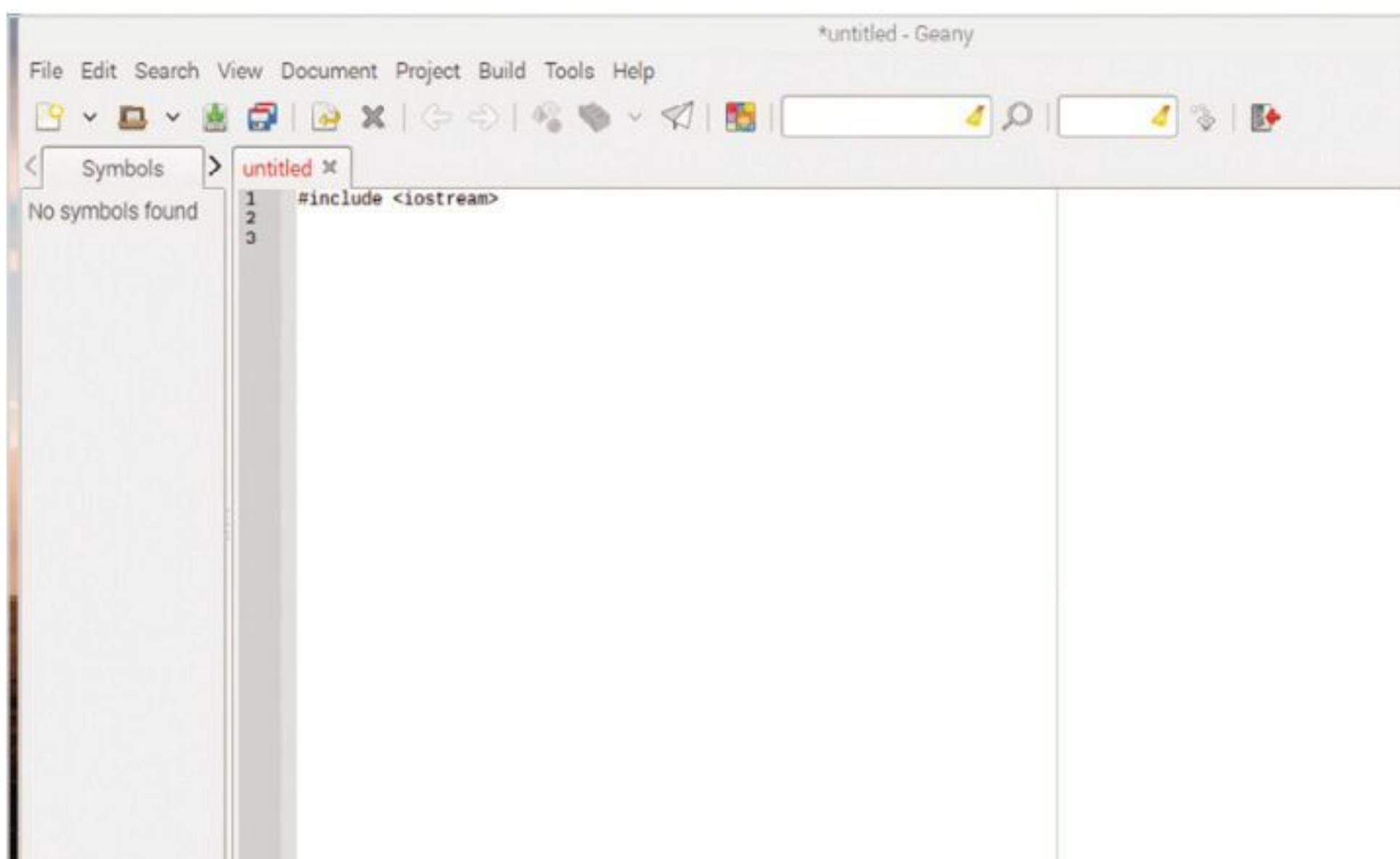
It doesn't look like much at the moment but you can get into what the C++ header commands mean in due course. Geany uses syntax highlighting, meaning it colour-codes the code depending on the language. To activate this, save the code by clicking on File > Save As. Now name the code helloworld.cpp and click the Save button.



STEP 2

The Geany editor is quite powerful while still being easy to use. All the code you enter shows in the main window and is numbered to help you keep track of your code. For now, in line 1, enter:

```
#include <iostream>
```

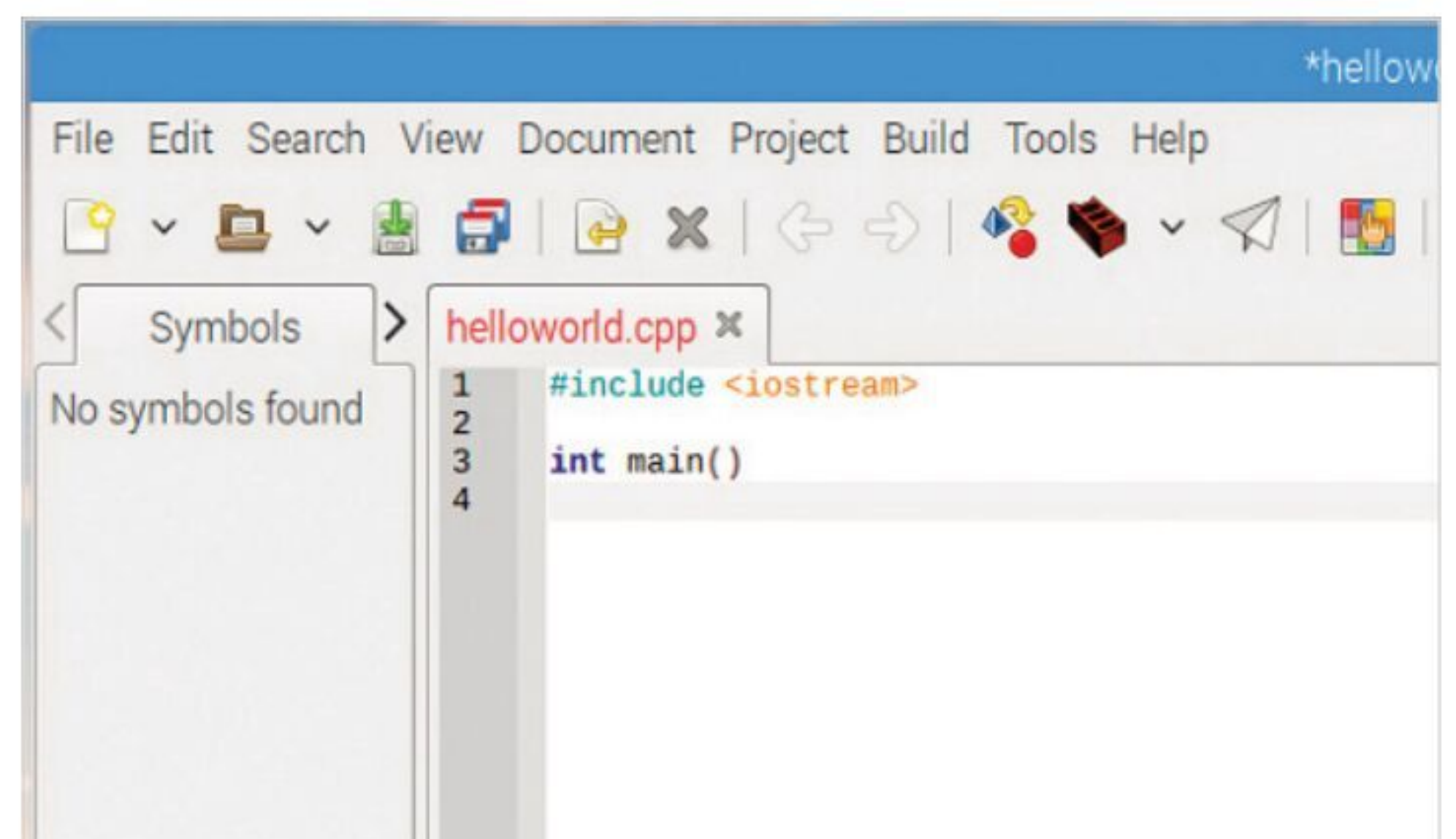


STEP 4

The colour coding helps a lot when your coding. Now press Enter and start a new command on line 3. Enter the following:

```
int main()
```

Note: there's no space between the brackets.

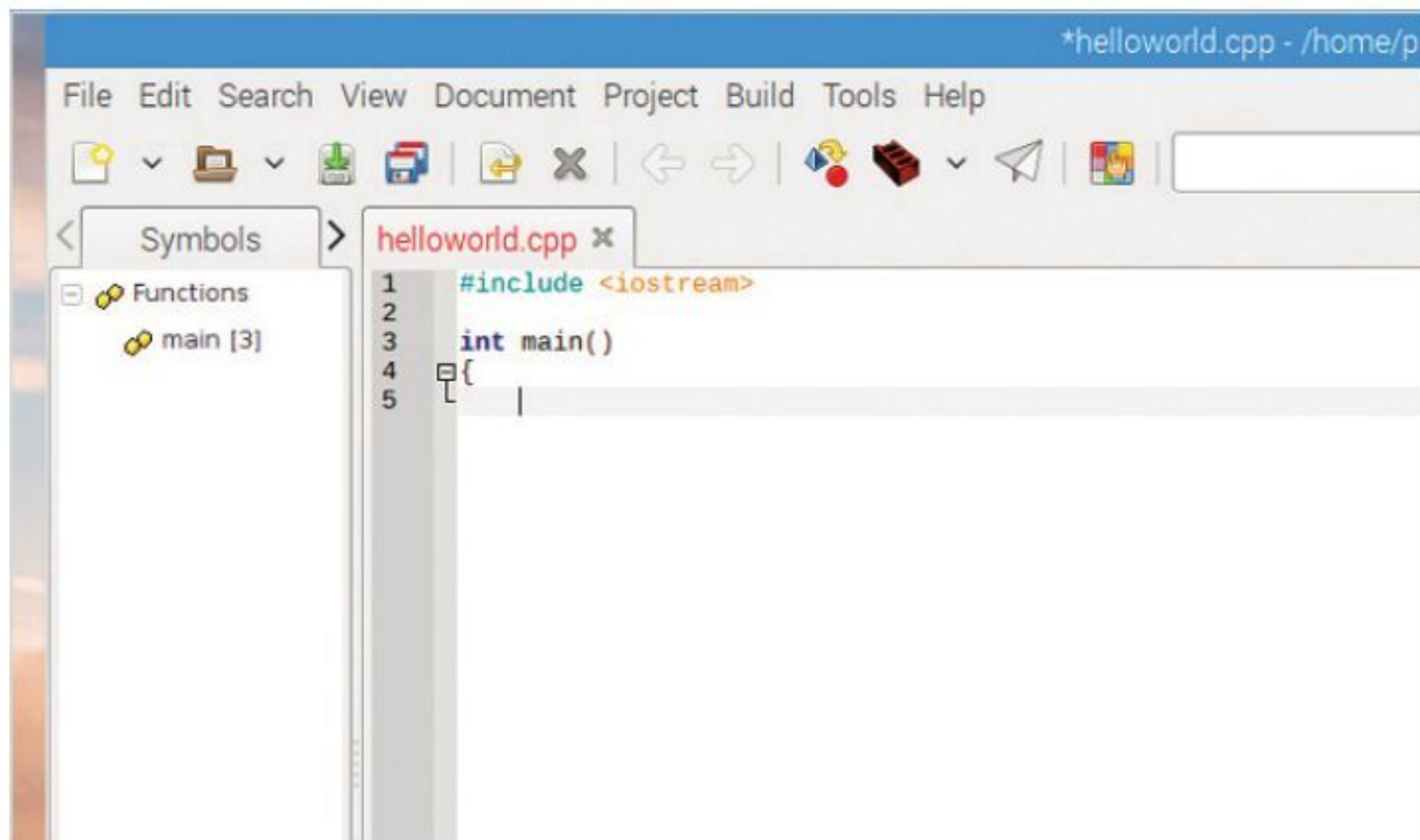




STEP 5 On the next line, below `int main()`, enter a curly bracket:

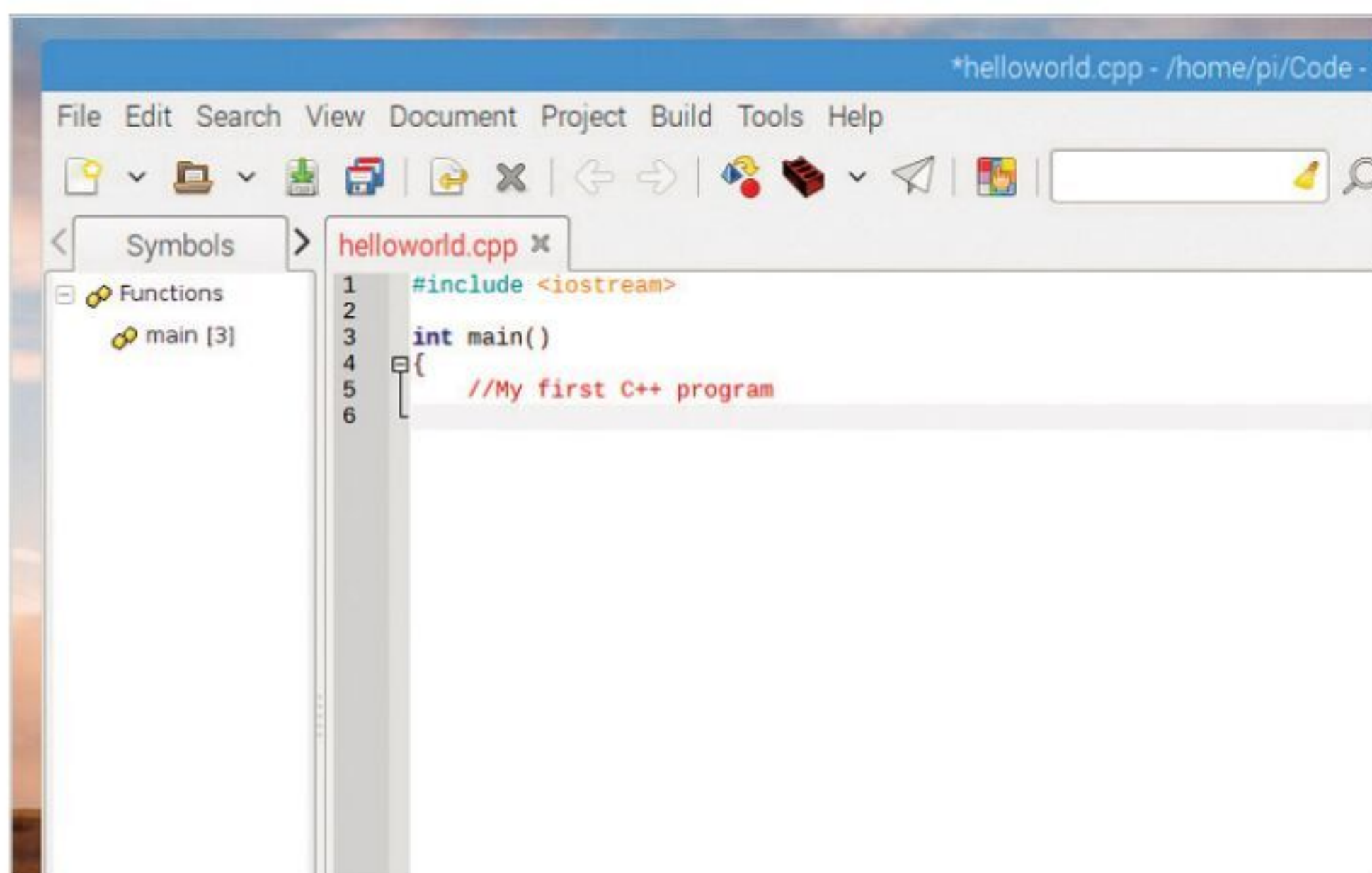
```
{
```

This can be done by pressing Shift and the key to the right of P on an English UK keyboard layout.



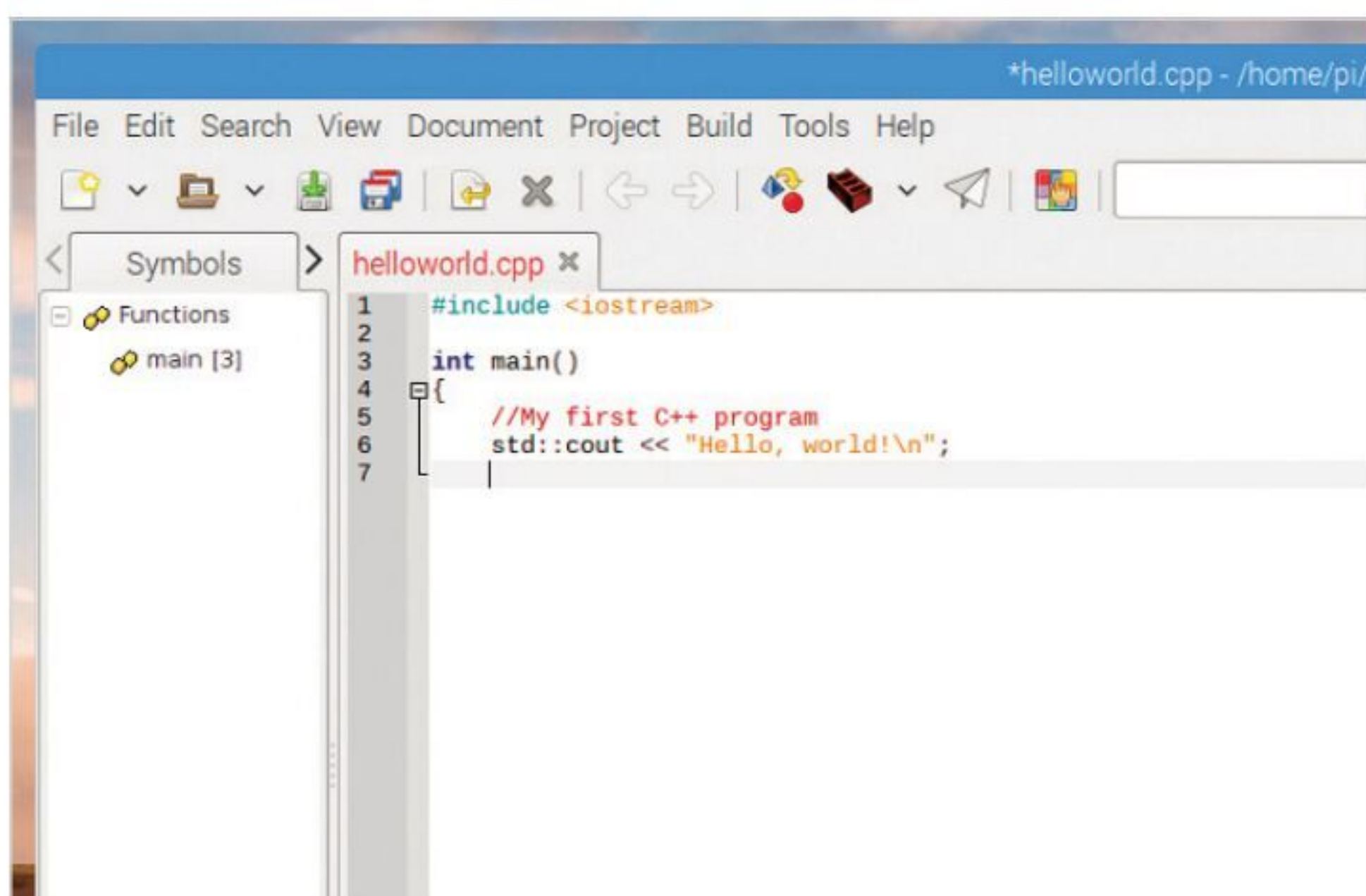
STEP 6 Notice that Geany has automatically created a slight indent. This is due to the structure of C++ and it's where the meat of the code is entered. Now enter:

```
//My first C++ program
```



STEP 7 Note again the colour coding change. Press Return at the end of the previous step's line and then enter:

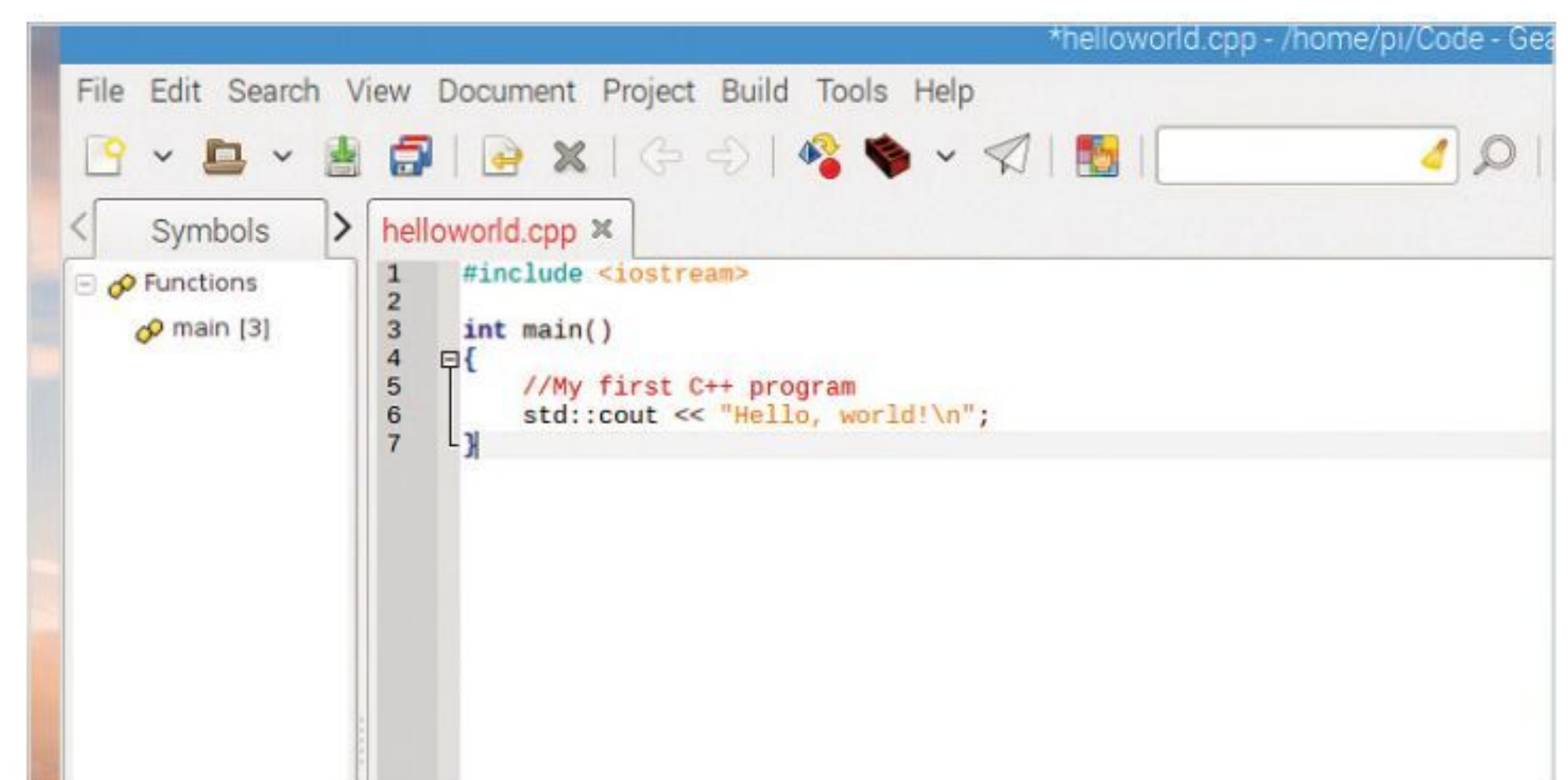
```
std::cout << "Hello, world!\n";
```



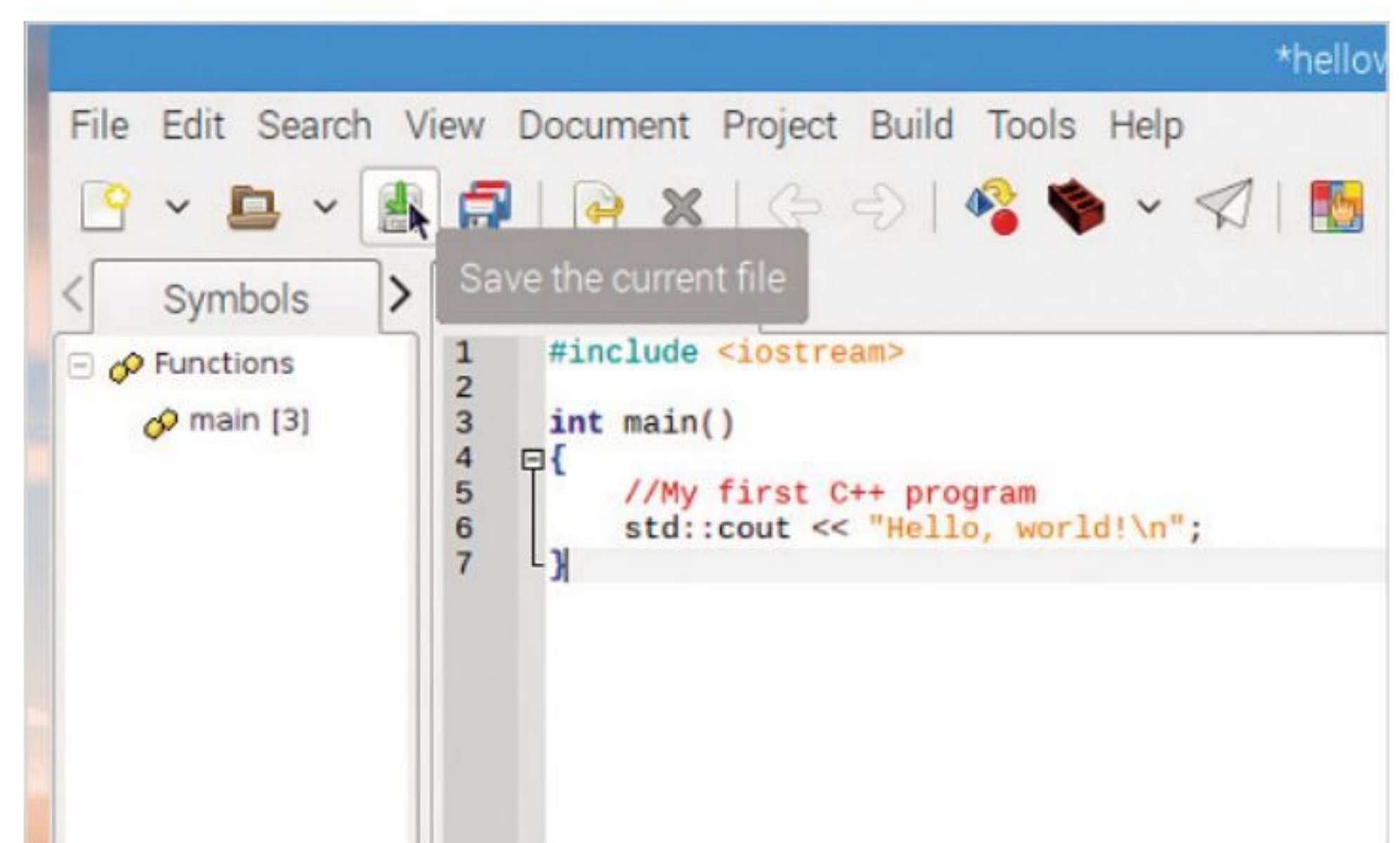
STEP 8 Now you need to close off the code, by inserting the opposite, closing, curly bracket. On line 7 enter:

```
}
```

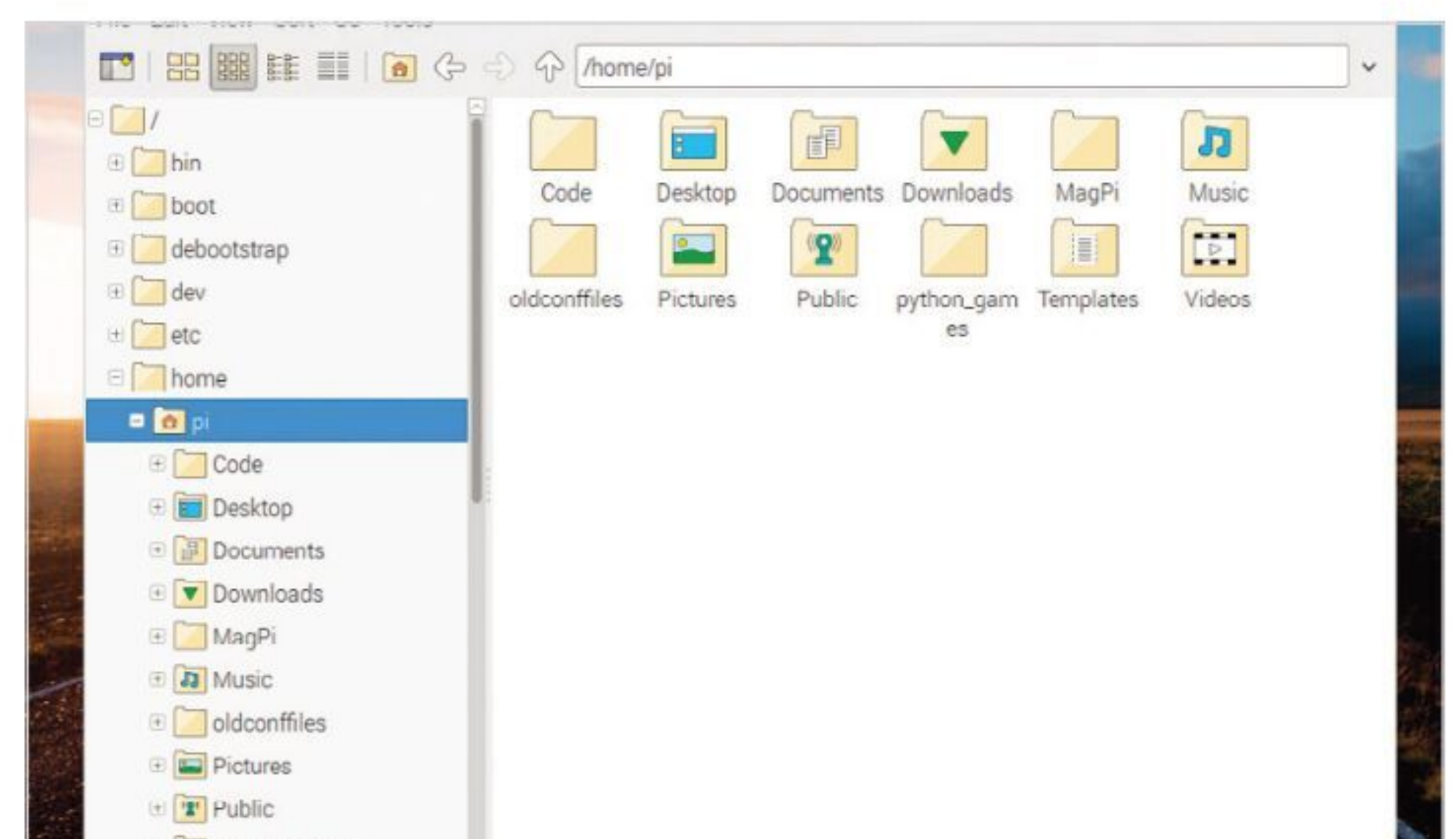
Note that you won't need to backspace to the start of the line, removing the indent, just by entering the curly bracket Geany auto-jumps to the correct part of the line, thus ending the code block.



STEP 9 Essentially, these are your first steps into the world of coding in C++. We will look at the structure of the code in more detail on the next couple of pages; for now though, click on the Save button as represented by a filing cabinet with a green downward pointing arrow.



STEP 10 If you haven't done so already, it's always advisable to create a blank folder where you can save all your code. You can have one each for Python and C++ or simply a single folder called Code, whichever you prefer. It's not a necessity, just good housekeeping.





Structure of a C++ Program

C++ has a very defined structure and way of doing things. Miss something out, even as small as a semicolon, and your entire program will fail to be compiled and executed. Many a professional programmer has fallen foul of sloppy structure.

#INCLUDE <C++ STRUCTURE>

Learning the basics of programming, you begin to understand the structure of a program. The commands may be different from one language to the next but you can start to see how the code works.

C++

C++ was invented by Danish student Bjarne Stroustrup in 1979, as a part of his Ph.D. thesis. Initially C++ was called C with Classes, which added features to the already popular C programming language, while making it a more user-friendly environment through a new structure.



Bjarne Stroustrup, inventor of C++.

INT MAIN()

int main() initiates the declaration of a function, which is a group of code statements under the name 'main'. All C++ code begins at the main function, regardless of where it actually lies within the code.

```
helloworld.cpp x
1  #include <iostream>
2
3  int main()
4
5
```

#INCLUDE

The structure of a C++ program is quite precise. Every C++ code begins with a directive: #include <>. The directive instructs the pre-processor to include a section of the standard C++ code. For example: #include <iostream> includes the iostream header to support input/output operations.

```
helloworld.cpp x
1  #include <iostream>
2
3
4
```

BRACES

The open brace (curly brackets) is something that you may not have come across before, especially if you're used to Python. The open brace indicates the beginning of the main function and contains all the code that belongs to that function.

```
helloworld.cpp x
1  #include <iostream>
2
3  int main()
4  {
5
6
7  }
8
```

**COMMENTS**

Lines that begin with a double slash are comments. This means they won't be executed in the code and are ignored by the compiler. Comments are designed to help you, or another programmer looking at your code, and explain what's going on. There are two types of comment: `/*` covers multiple line comments and `//` a single line.

```
helloworld.cpp x
1  #include <iostream>
2
3  int main()
4  {
5      //My first C++ program
6
7  }
8
```

<<

The two chevrons used here are insertion operators. This means that whatever follows the chevrons is to be inserted into the `std::cout` statement. In this case they're the words 'Hello, world' which are to be displayed on the screen when you compile and execute the code.

```
helloworld.cpp x
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      //My first C++ program
7      cout << |
8  }
9
```

STD

While `std` stands for something quite different, in C++ it means Standard. It's part of the Standard Namespace in C++, which covers a number of different statements and commands. You can leave the `std::` part out of the code but it must be declared at the start with: `using namespace std;` not both. For example:

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

```
helloworld.cpp x
1  #include <iostream>
2
3  int main()
4  {
5      //My first C++ program
6      std::cout << "Hello, world!\n";
7  }
8
```

OUTPUTS

Leading on, the "Hello, world!" part is what you want to appear on the screen when the code is executed. You can enter whatever you like, as long as it's inside the quotation marks. Sometimes brackets are needed, depending on the compiler. The `\n` part indicates a new line is to be inserted.

```
//My first C++ program
cout << "Hello, world!\n"
```

cout

In this example we're using `cout`, which is a part of the Standard Namespace, hence why it's there, as you're asking C++ to use it from that particular namespace. `cout` means Character OUTPUT, which displays, or prints, something to the screen. If you leave `std::` out you have to declare it at the start of the code, as mentioned previously.

```
helloworld.cpp x
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      //My first C++ program
7      cout |
8  }
9
```

; AND }

Finally you can see that lines within a function code block (except comments) end with a semicolon. This marks the end of the statement; all statements in C++ must have one at the end or the compiler fails to build the code. The very last line has the closing brace to indicate the end of the main function.

```
helloworld.cpp x
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      //My first C++ program
7      cout << "Hello, world!\n";
8  }
9
```



Compile and Execute

You've created your first C++ program and you now understand the basics behind the structure of one. Let's get things moving and compile and execute, or run if you prefer, the program and see how it looks.

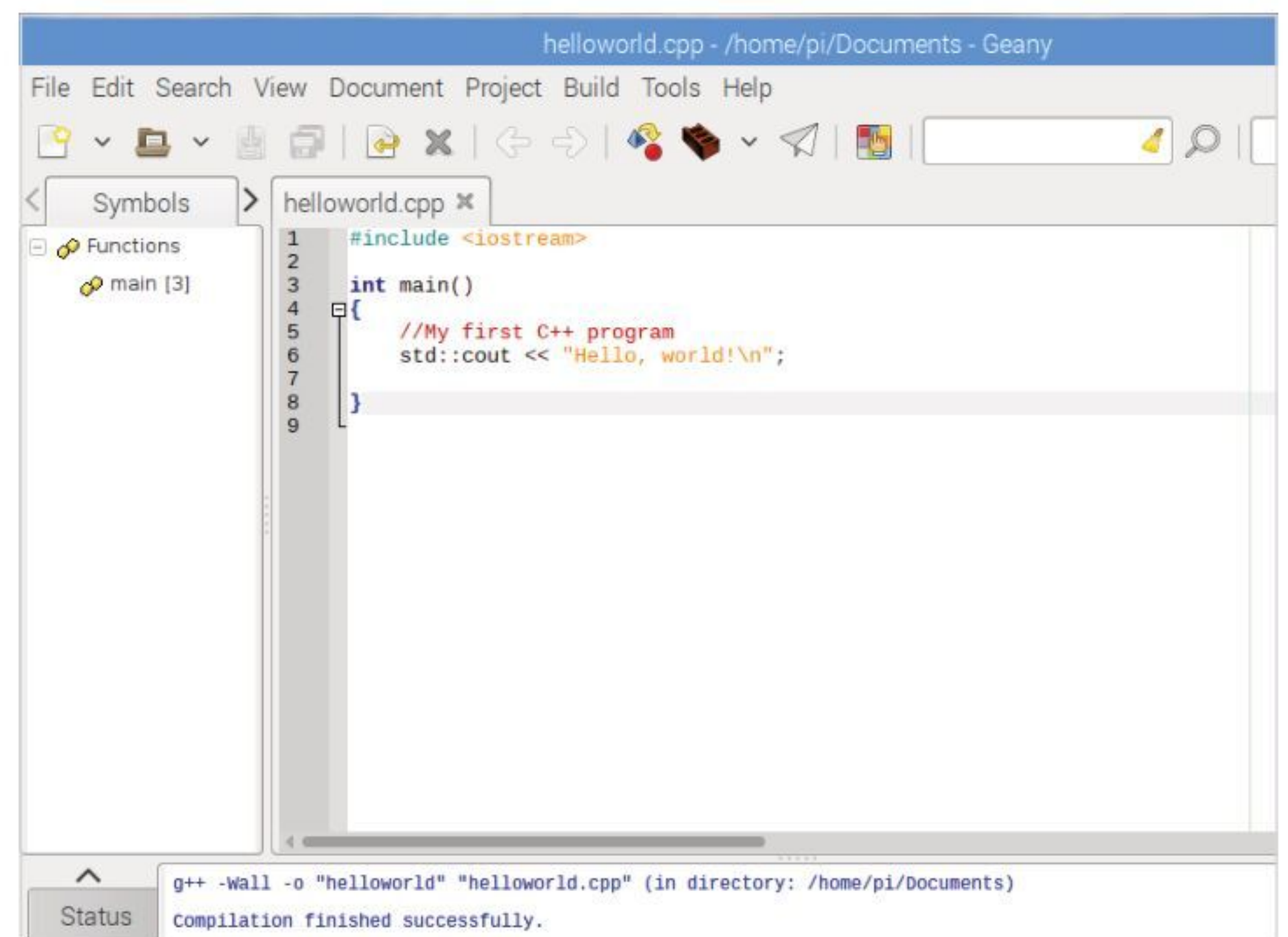
GREETINGS FROM C++

Compiling and executing code from within Geany is remarkably simple and just a matter of clicking a couple of icons and seeing the result. Here's how it's done.

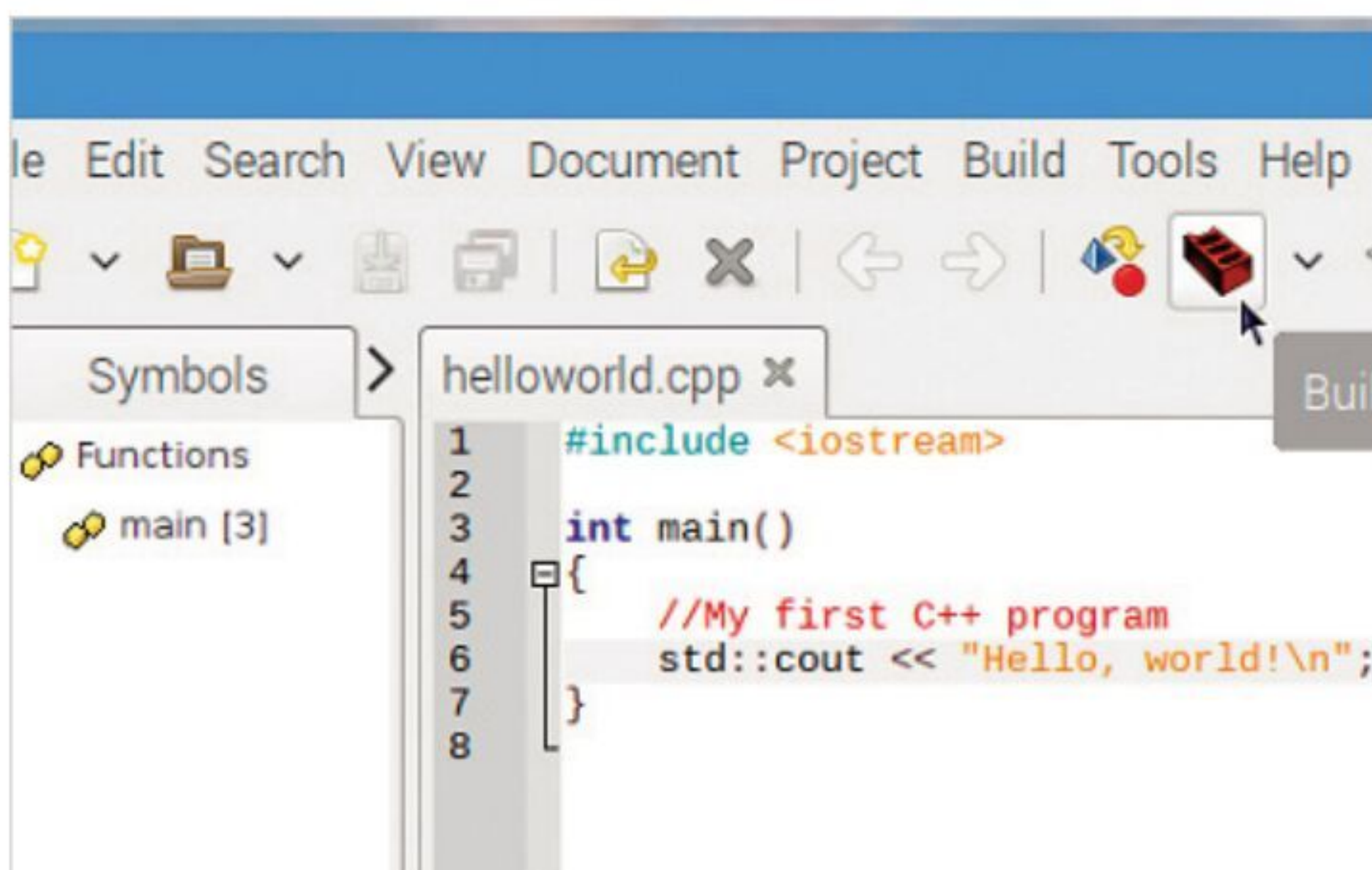
STEP 1 Open Geany, if you haven't already, and load up the previously saved Hello World code you created. Ensure that there are no visible errors, such as a missing semicolon at the end of the `std::cout` line.

```
helloworld.cpp x
1 #include <iostream>
2
3 int main()
4 {
5     //My first C++ program
6     std::cout << "Hello, world!\n";
7 }
8
```

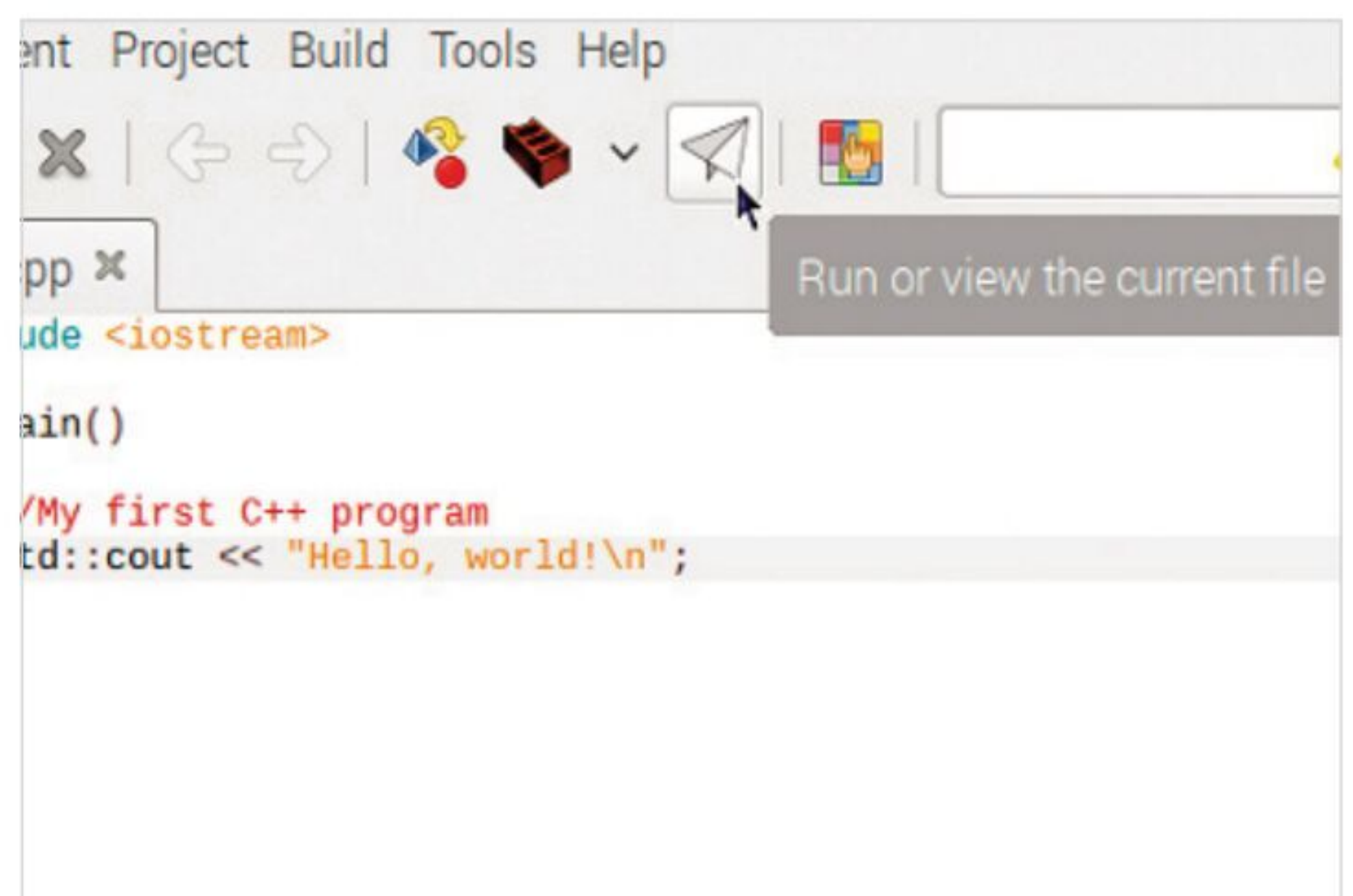
STEP 3 When you click the Build icon, the pane at the bottom of the Geany interface displays some results. If all is well, you will see a successful compilation, if not then Geany issues an error along with the line the error was found on.



STEP 2 If your code is ready, look to the menu bar along the top of the Geany window. Notice that midway along the line of icons there's a red brick icon. This is Build, which when clicked runs through the code, checking it against the C++ standards to see if there are any errors that stop it from running.



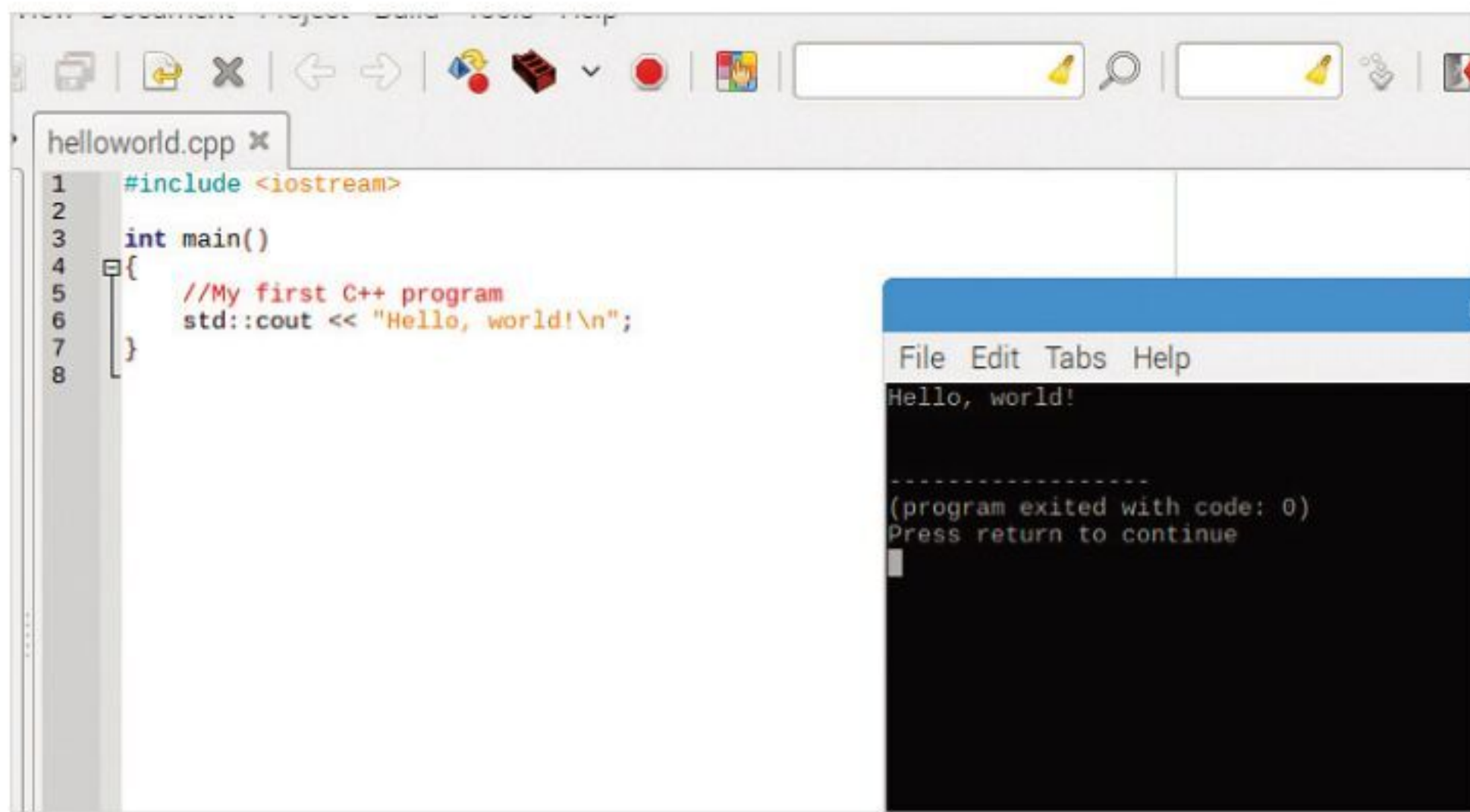
STEP 4 With the code successfully built, it's now time to execute it. Next to the Build icon there's a paper aeroplane icon, this is the Run icon. You can't run code that hasn't been previously built or has an error in place. Click the Run icon.





STEP 5

When you click Run, the C++ code executes and a simple Terminal window appears displaying the results of the code. In the case of this simple code, you can see the words: Hello, world! in the window; as you've already learned, the code is cout(ing) (outputting) the contents between the quotes in the std::cout line.



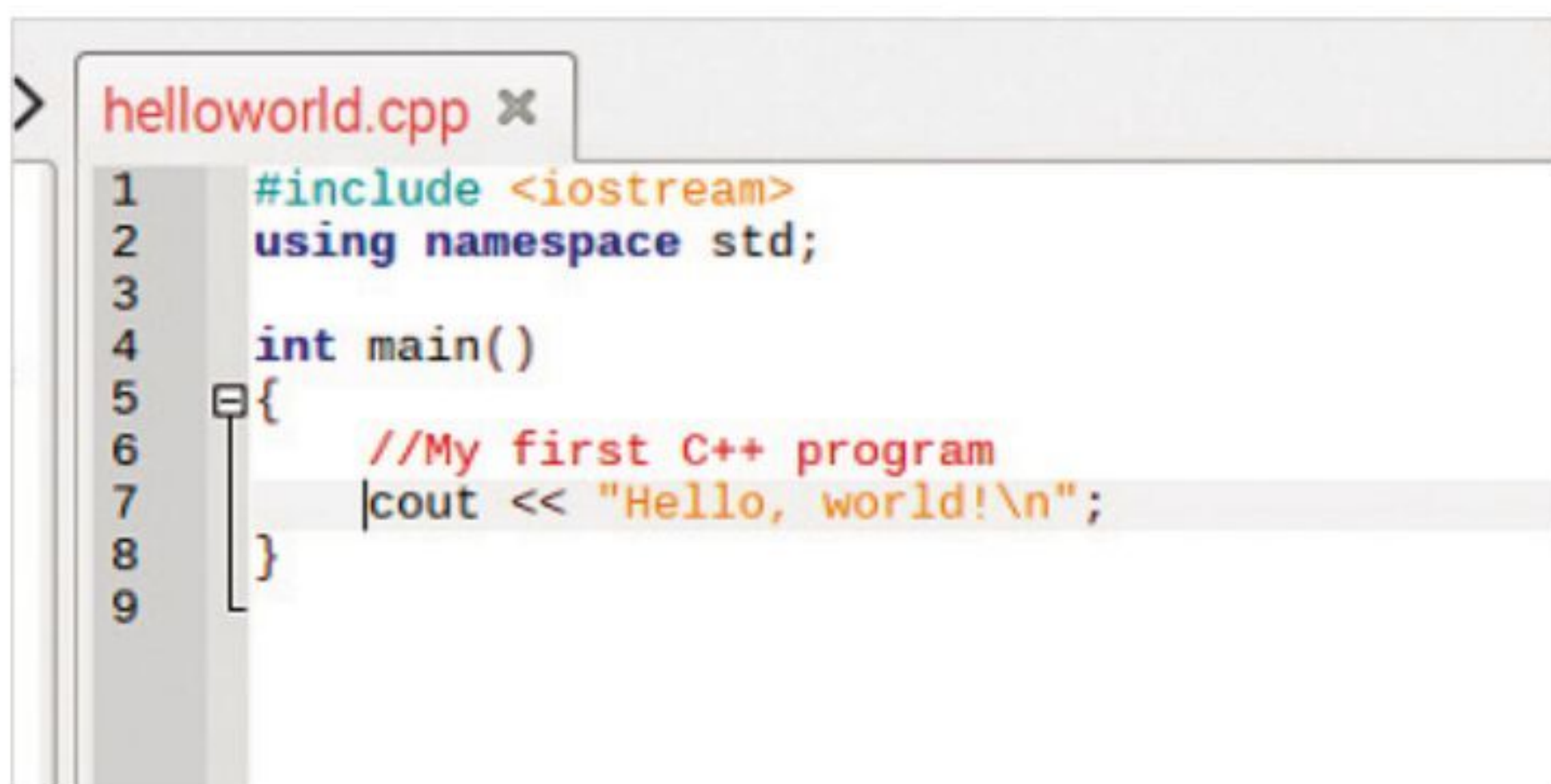
STEP 6

Pressing Return/Enter in the command line box closes it, returning you to Geany. Let's alter the code slightly. Under the #include line, enter:

```
using namespace std;
```

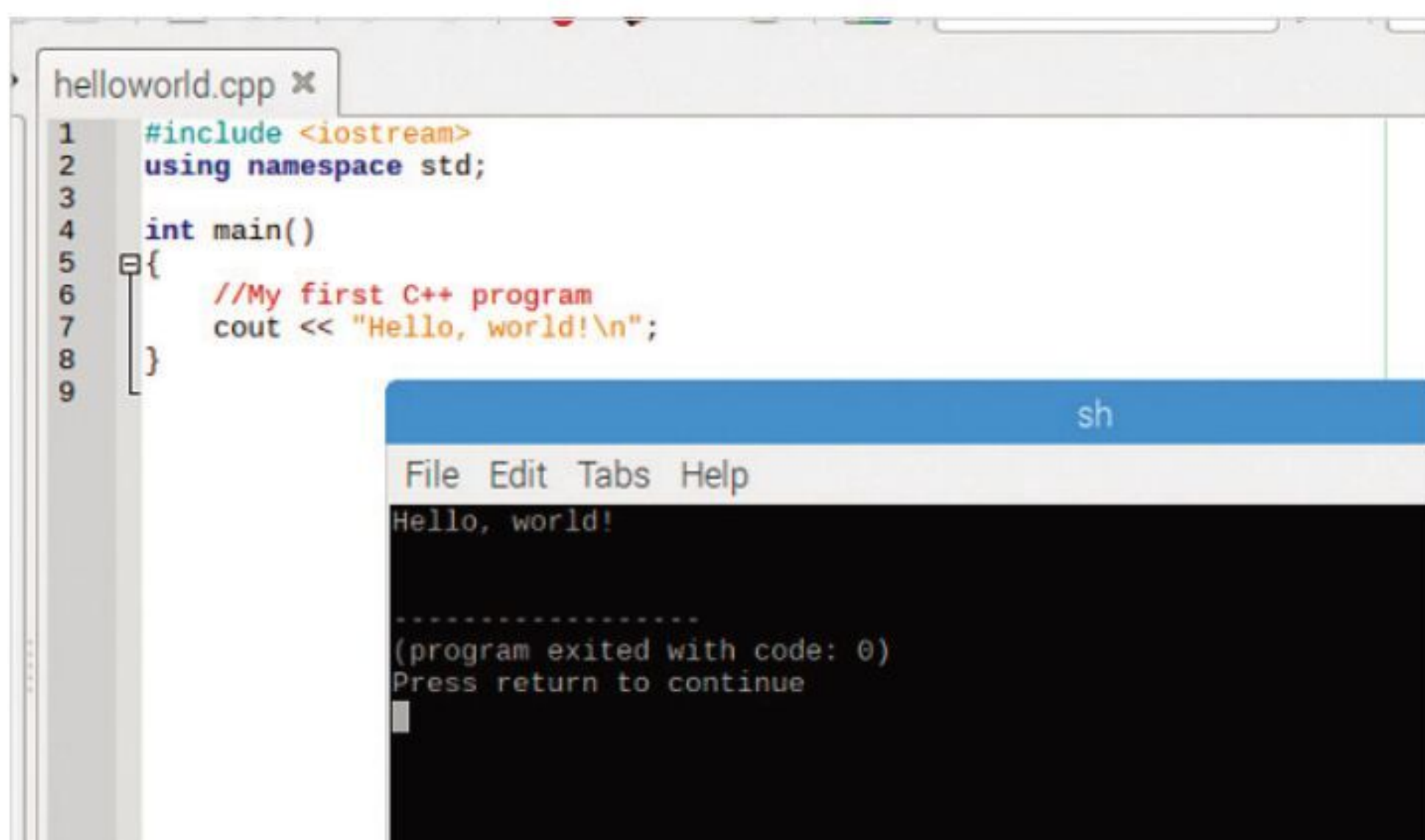
Then, delete the std:: part of the Cout line; like so:

```
cout << "Hello, world\n";
```



STEP 7

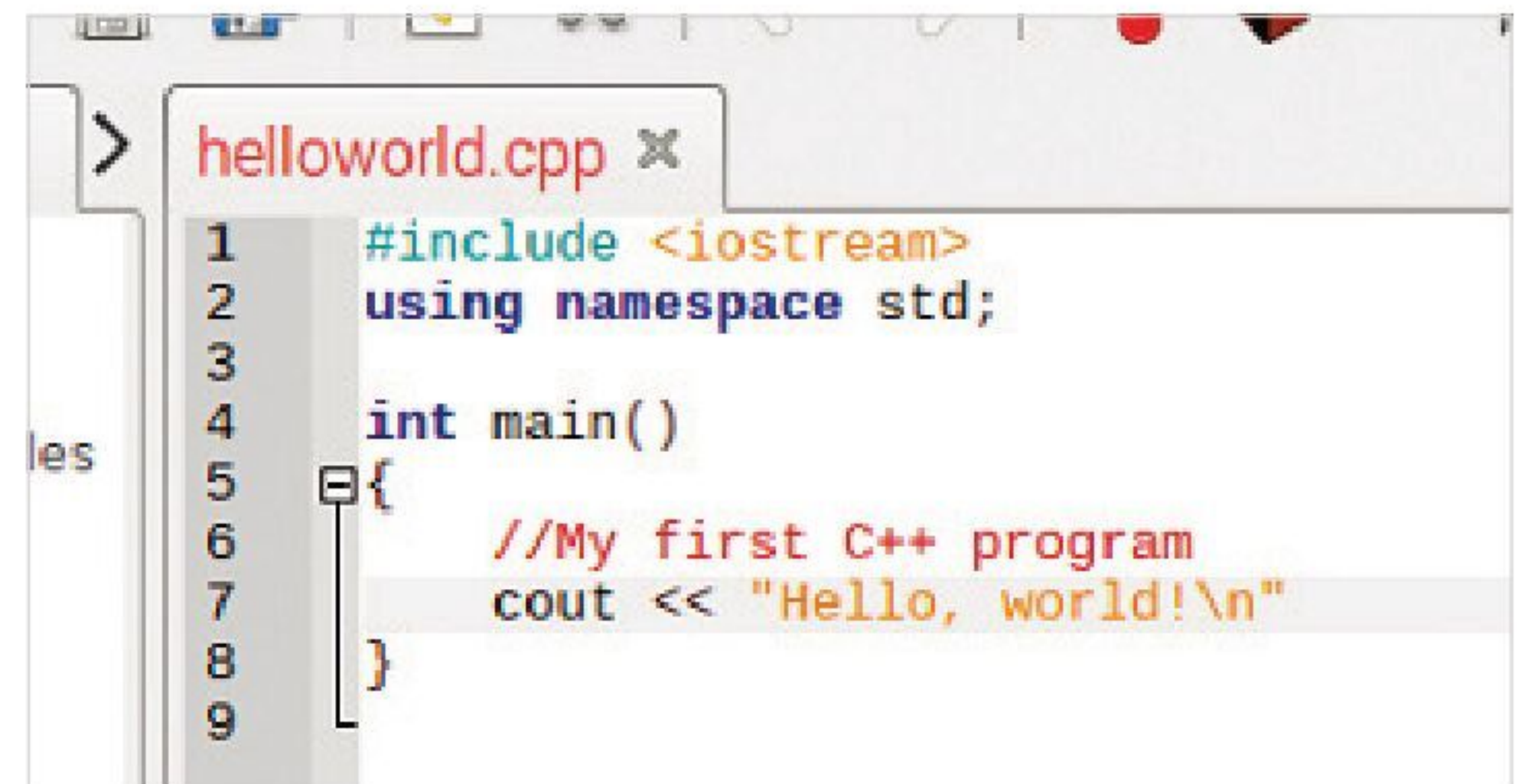
In order to apply the new changes you need to save the file, build it and execute it again. Go through the process as before, you don't need to change the name of the file unless you want to, of course, followed by clicking the Brick icon, then the Aeroplane icon.



STEP 8

Just as mentioned previously, you don't need to have std::cout if you've already declared the standard namespace at the start of the code. Now, let's create a deliberate mistake, to see what happens. Remove the semicolon from the cout line, so it reads:

```
cout << "Hello, world!\n"
```



STEP 9

Now try and build the code once more. This time, notice that the bottom panel is displaying an error, regarding the missing semicolon. Notice that error reads as being on line 8, the first character, the closed curly bracket. This is because there's an expected semicolon missing before the closed bracket.

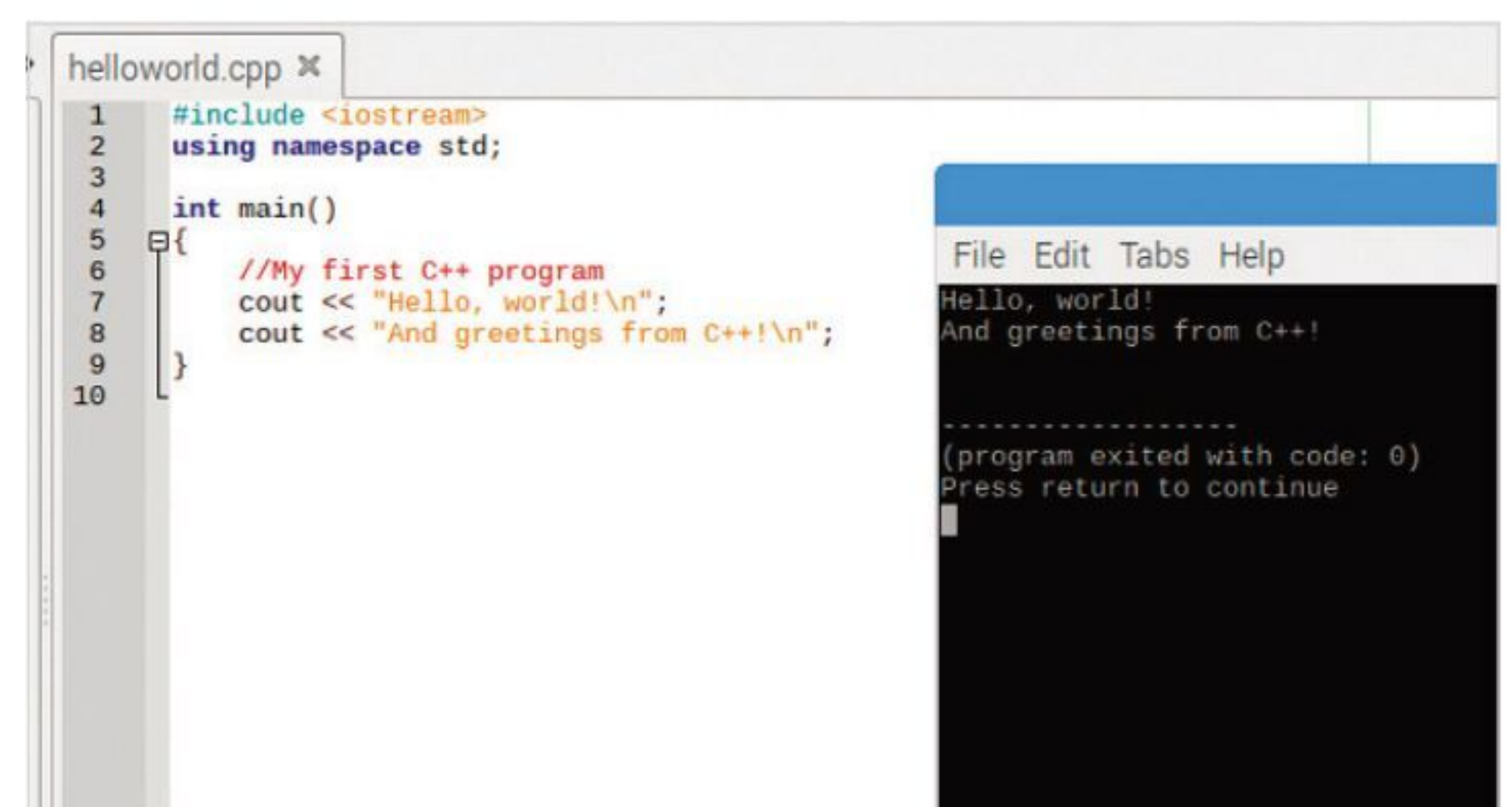


STEP 10

Replace the semicolon and under the cout line, enter a new line to your code:

```
cout << "And greetings from C++!\n";
```

The \n simply adds a new line under the last line of outputted text. Build and Run the code, to display your handiwork.





Virus!

DID YOU KNOW...

that one of the first programmed computer viruses that spread outside of a controlled system, i.e. into the wild, was called Elk Cloner and it was coded in 1982 by Rich Skrenta, a fifteen year old student.

Elk Cloner was programmed to infect the Apple DOS 3.3 operating system, and attached to a game. When the game was played fifty times the virus would be activated and instead of displaying the game the virus would blank the screen and display a poem:

Elk Cloner:
The program with a
personality

It will get on all
your disks
It will infiltrate your
chips
Yes, it's Cloner!

It will stick to you
like glue
It will modify RAM too
Send in the Cloner!





Using Comments

While comments may seem like a minor element to the many lines of code that combine to make a game, application or even an entire operating system, in actual fact they're probably one of the most important factors.

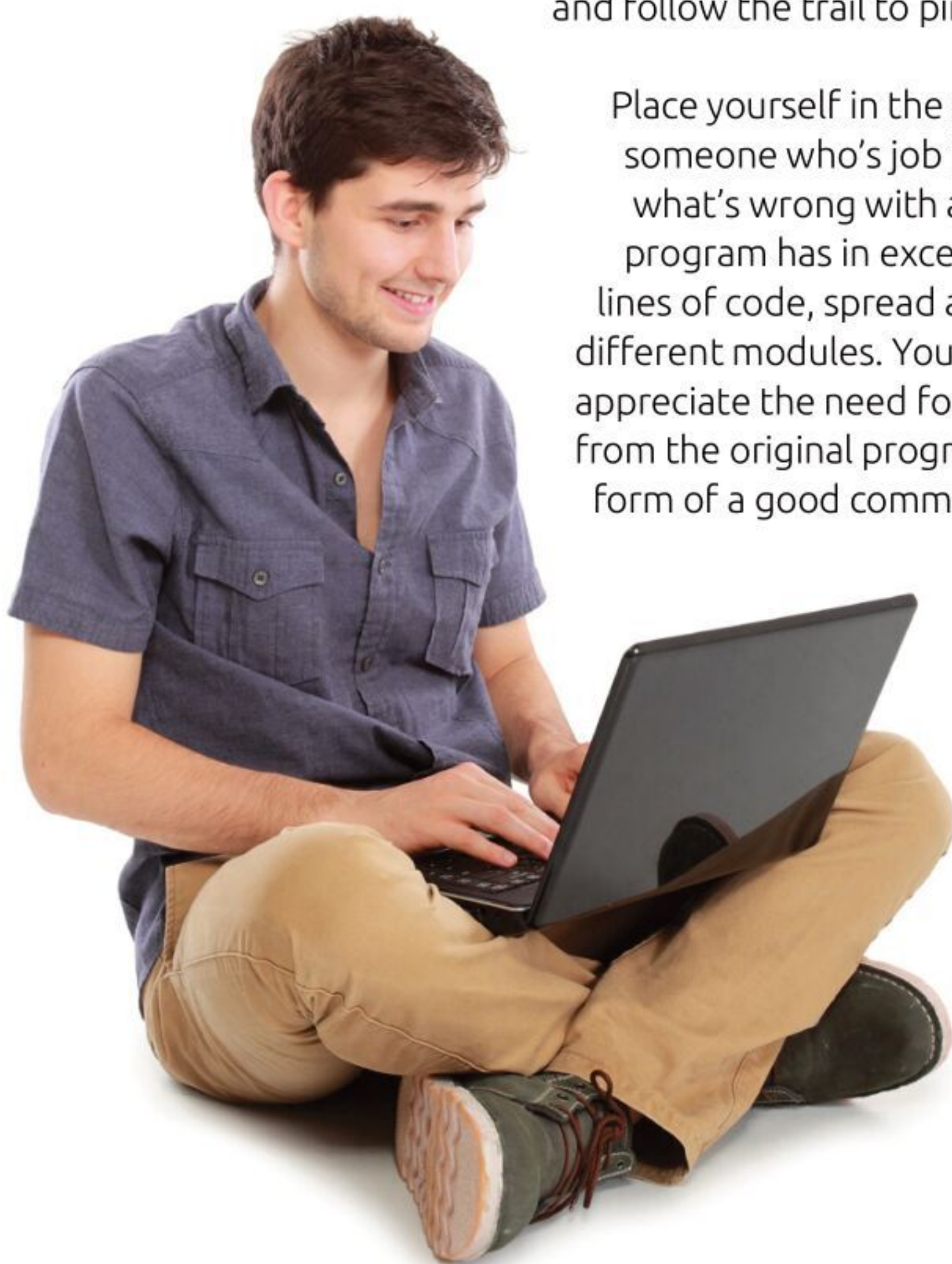
THE IMPORTANCE OF COMMENTING

Comments inside code are basically human readable descriptions that detail what the code is doing at that particular point. They don't sound especially important but code without comments is one of the many frustrating areas of programming, regardless of whether you're a professional or just starting out.

In short, all code should be commented in such a manner as to effectively describe the purpose of a line, section or individual elements. You should get in to the habit of commenting as much as possible, by imagining that someone who doesn't know anything about programming can pick up your code and understand what it's going to do simply by reading your comments.

In a professional environment, comments are vital to the success of the code and ultimately, the company or project. In an organisation, many programmers work in teams alongside engineers, other developers, hardware analysts and so on. If you're a part of the team that's writing a bespoke piece of software for the company, then your comments help save a lot of time should something go wrong and another team member has to pick up and follow the trail to pinpoint the issue.

Place yourself in the shoes of someone who's job it is to find out what's wrong with a program. The program has in excess of 800,000 lines of code, spread across several different modules. You can soon appreciate the need for a little help from the original programmers in the form of a good comment.



The best comments are always concise and link the code logically, detailing what happens when the program hits a line or section. You don't need to comment on every line. Something along the lines of: if x==0 doesn't require you to comment that if x equals zero then do something; that's going to be obvious to the reader. However, if x equalling zero is something that drastically changes the program for the user, such as, they've run out of lives, then it certainly needs to be commented on.

Even if the code is your own, you should write comments as if you were going to share it with others publicly. This way you can return to that code and always understand what it was you did or where it was you went wrong, or what worked brilliantly.

Comments are good practise and once you understand how to add a comment where needed, you soon do it as if it's second nature.

```

DEFB 26h,30h,32h,26h,30h,32h,0,0,32h,72h,73h,32h,72h,73h,32h
DEFB 60h,61h,32h,4Ch,4Dh,32h,4Ch,99h,32h,4Ch,4Dh,32h,4Ch,4Dh
DEFB 32h,4Ch,99h,32h,5Bh,5Ch,32h,56h,57h,32h,33h,0CDh,32h,33h
DEFB 34h,32h,33h,34h,32h,33h,0CDh,32h,40h,41h,32h,66h,67h,64h
DEFB 66h,67h,32h,72h,73h,64h,4Ch,4Dh,32h,56h,57h,32h,80h,0CBh
DEFB 19h,80h,0,19h,80h,81h,32h,80h,0CBh,0FFh

T858C:
DEFB 80h,72h,66h,60h,56h,66h,56h,56h,51h,60h,51h,51h,56h,66h
DEFB 56h,56h,80h,72h,66h,60h,56h,66h,56h,56h,51h,60h,51h,51h
DEFB 56h,56h,56h,56h,80h,72h,66h,60h,56h,66h,56h,56h,51h,60h
DEFB 51h,51h,56h,66h,56h,56h,80h,72h,66h,60h,56h,66h,56h,40h
DEFB 56h,66h,80h,66h,56h,56h,56h,56h

;
; Game restart point
;
START: XOR    A
LD     (SHEET),A
LD     (KEMP),A
LD     (DEMO),A
LD     (B845B),A
LD     (B8458),A
LD     A,2           ;Initial lives count
LD     (NOMEN),A
LD     HL,T845C
SET    0,(HL)
LD     HL,SCREEN
LD     DE,SCREEN+1
LD     BC,17FFh     ;Clear screen image
LD     (HL),0
LDIR
LD     HL,0A000h    ;Title screen bitmap
LD     DE,SCREEN
LD     BC,4096
LDIR
LD     HL,SCREEN + 800h + 1*32 + 29
LD     DE,MANDAT+64
LD     C,0
CALL  DRWFIX
LD     HL,0FC00h    ;Attributes for the last room
LD     DE,ATTR     ;(top third)
LD     BC,256
LDIR
LD     HL,09E00h    ;Attributes for title screen
LD     BC,512     ;(bottom two-thirds)
LDIR
LD     BC,31
DI
XOR    A
R8621: IN     E,(C)
OR     E
DJNZ  R8621    ;$-03
AND   20h
JR    NZ,R862F ;$+07
LD     A,1

```



C++ COMMENTS

Commenting in C++ involves using a double forward slash `/**` or a forward slash and an asterisk `/*`. You've already seen some brief examples but this is how they work.

STEP 1 Using the Hello World code as an example, you can easily comment on different sections of the code using the double forward slash:

```
//My first C++ program cout << "Hello, world!\n";
```

```
helloworld.cpp x
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      //My first C++ program
7      cout << "Hello, world!\n";
8      cout << "And greetings from C++!\n";
9  }
10
```

STEP 2 However, you can also add comments to the end of a line of code, to describe in a better way what's going on:

```
cout << "Hello, world!\n"; //This line outputs the words 'Hello, world!'. The \n denotes a new line.
```

Note, you don't have to put a semicolon at the end of a comment. This is because it's a line in the code that's ignored by the compiler.

```
helloworld.cpp x
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      //My first C++ program
7      cout << "Hello, world!\n"; //This line outputs the words 'Hello, world. The \n denotes a new line.
8      cout << "And greetings from C++!\n";
9  }
10
```

STEP 3 You can comment out several lines by using the forward slash and asterisk:

```
/* This comment can
   cover several lines
   without the need to add more slashes */
```

Just remember to finish the block comment with the opposite asterisk and forward slash.

```
helloworld.cpp x
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      //My first C++ program
7      cout << "Hello, world!\n"; //This line outputs the words 'Hello, world. The \n
8
9      /* This comment can
10     * cover several lines
11     * without the need to add more slashes */
12
13     cout << "And greetings from C++!\n";
14 }
15
16
```

STEP 4 Be careful when commenting, especially with block comments. It's very easy to forget to add the closing asterisk and forward slash and thus negate any code that falls inside the comment block.

```
helloworld.cpp x
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      //My first C++ program
7      cout << "Hello, world!\n"; //This line outputs the words 'Hello, w
8
9      /* This comment can
10     * cover several lines
11     * without the need to add more slashes
12
13     cout << "And greetings from C++!\n";
14 }
15
16
```

STEP 5 Obviously if you try and build and execute the code it errors out, complaining of a missing curly bracket `}` to finish off the block of code. If you've made the error a few times, then it can be time consuming to go back and rectify. Thankfully, the colour coding in Geany helps identify comments from code.

```
helloworld.cpp x
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      //My first C++ program
7      cout << "Hello, world!\n";
8
9      /* This comment can
10     * cover several lines
11     * without the need to add more slashes
12
13     cout << "And greetings from C++!\n";
14 }
15
16
```

STEP 6 With block comments, it's good practise in C++ to add an asterisk to each new line of the comment block. This also helps you to remember to close the comment block off before continuing with the code:

```
/* This comment can
 * cover several lines
 * without the need to add more slashes */
```

Thankfully, Geany does this automatically but be aware of it when using other IDEs.

```
/* This comment can
 * cover several lines
 * without the need to add more slashes */
```



Variables

Variables differ slightly when using C++ as opposed to Python. In Python, you can simply state that 'a' equals 10 and a variable is assigned. However, in C++ a variable has to be declared with its type before it can be used.

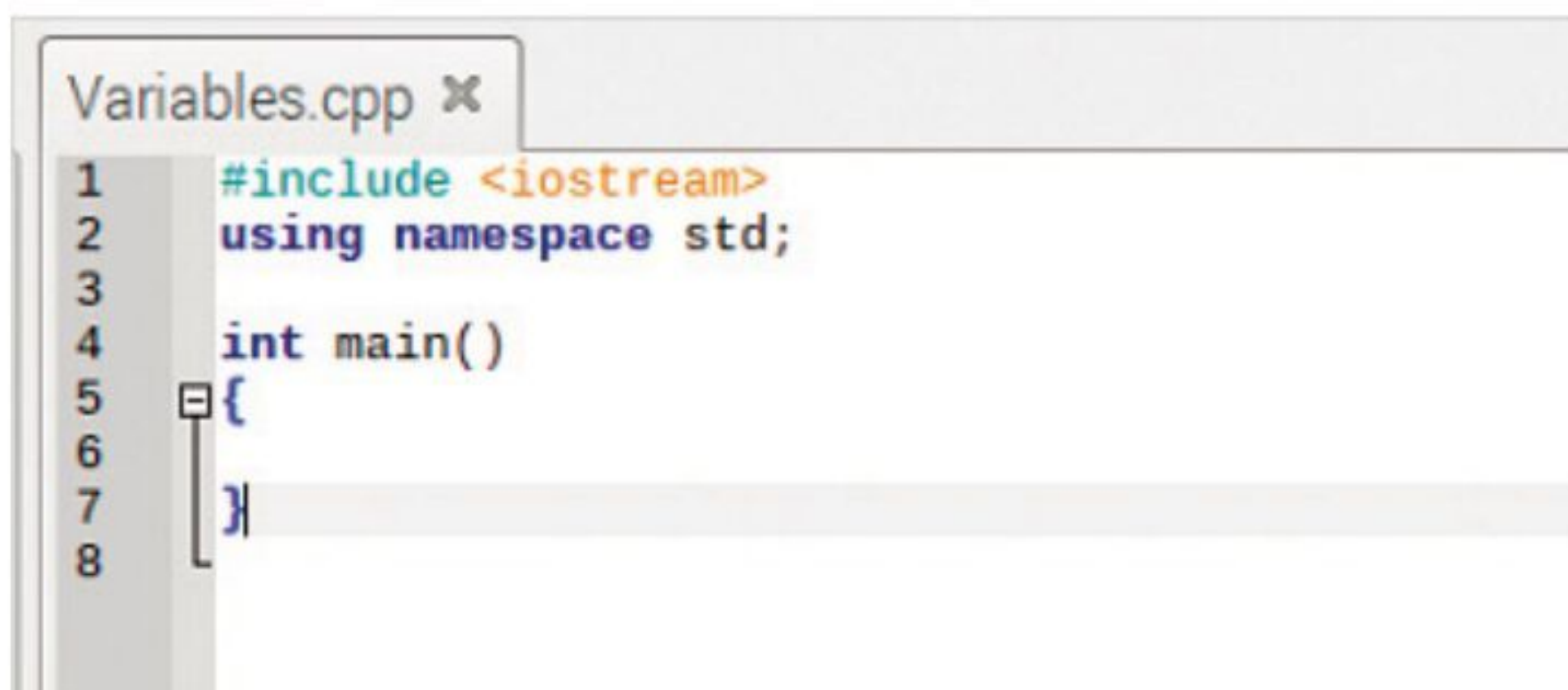
THE DECLARATION OF VARIABLES

You can declare a C++ variable by using statements within the code. There are several distinct types of variable you can declare. Here's how it works.

STEP 1 Open up a new, blank C++ file and enter the usual code headers:

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

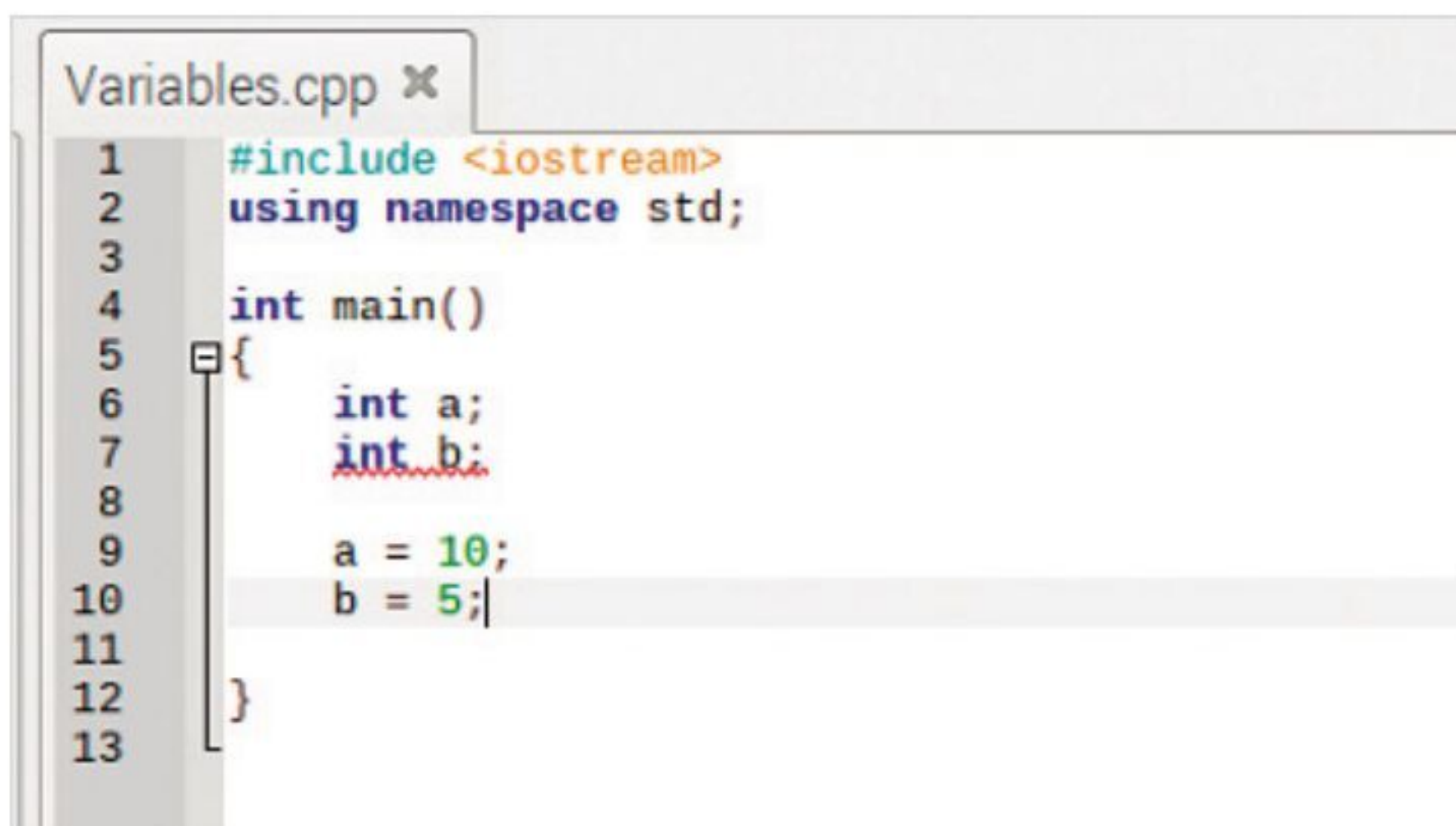
int main()
{
}
```



STEP 2 Start simple by creating two variables, a and b, with one having a value of 10 and the other 5. You can use the data type int to declare these variables. Within the curly brackets enter:

```
int a;
int b;

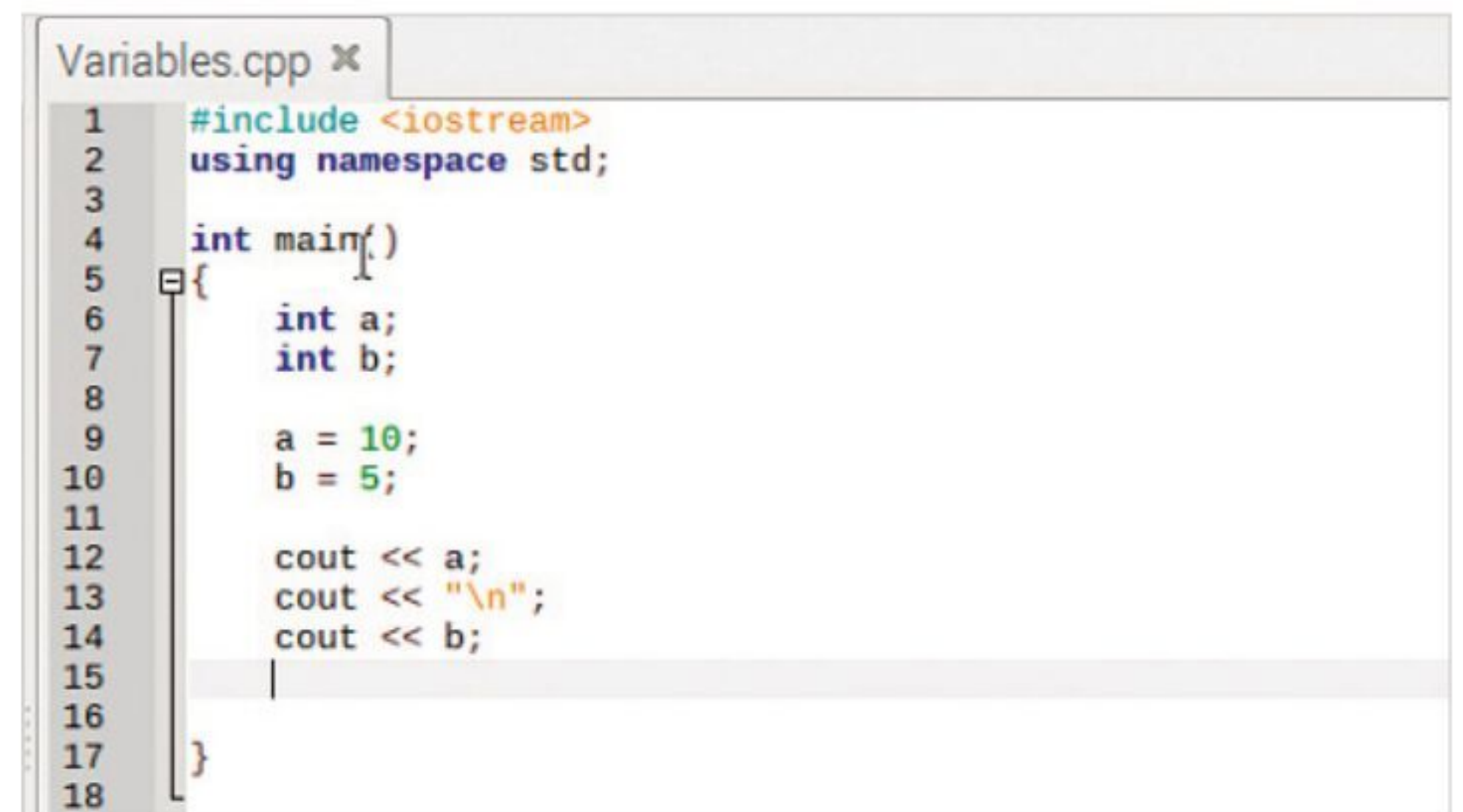
a = 10;
b = 5;
```



STEP 3 You can build and run the code but it won't do much, other than store the values 10 and 5 to the integers a and b. Ignore the warnings about variables set but not used, the code still works. To output the contents of the variables, add:

```
cout << a;
cout << "\n";
cout << b;
```

The cout << "\n"; part simply places a new line between the output of 10 and 5.

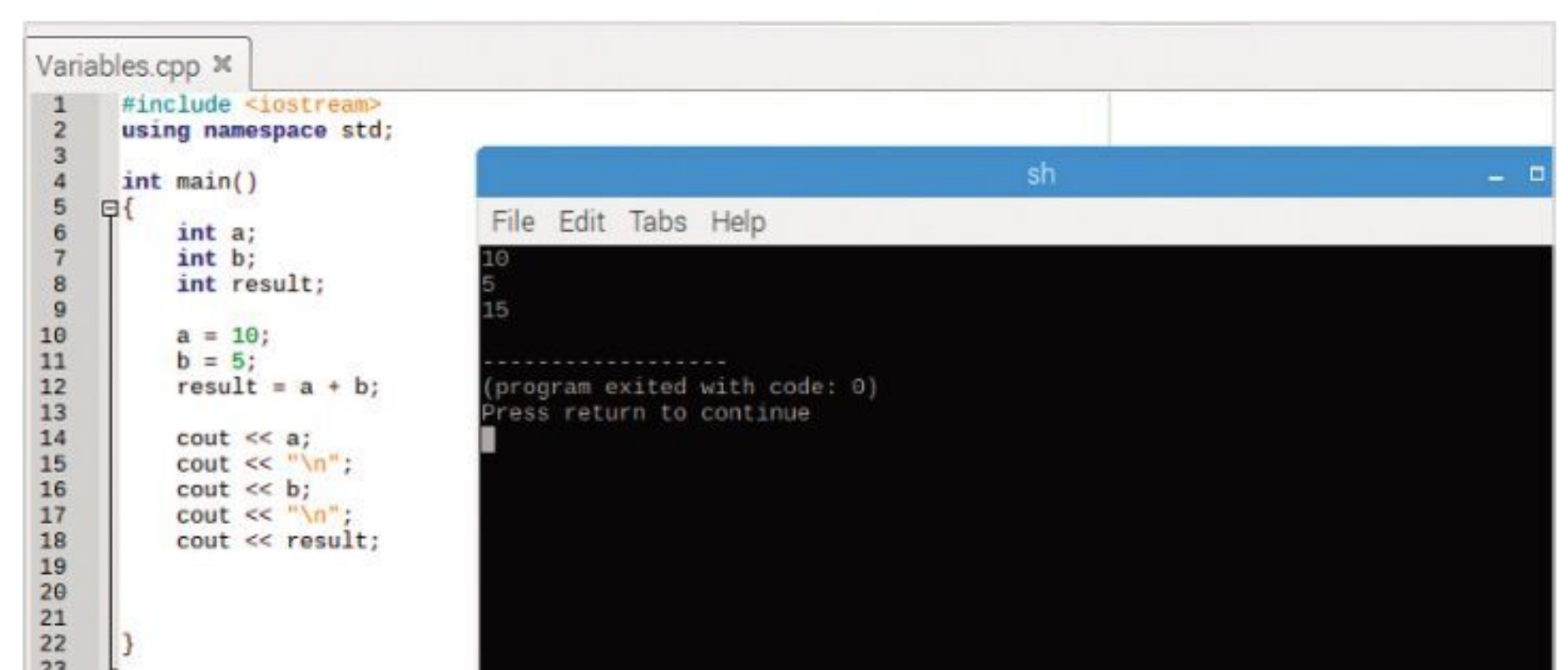


STEP 4 Naturally you can declare a new variable, call it result and output some simple arithmetic:

```
int result;

result = a + b;
cout << result;
```

Insert the above into the code as per the screenshot.



**STEP 5**

You can assign a value to a variable as soon as you declare it. The code you've typed in could look like this, instead:

```
int a = 10;
int b = 5;
int result = a + b;

cout << result;
```

STEP 6

Specific to C++, you can also use the following to assign values to a variable as soon as you declare them:

```
int a (10);
int b (5);
```

Then, from the C++ 2011 standard, using curly brackets:

```
int result {a+b};
```

STEP 7

You can create global variables, which are variables that are declared outside any function and used in any function within the entire code. What you've used so far are local variables: variables used inside the function. For example:

```
#include <iostream>
using namespace std;
int StartLives = 3;

int main ()
{
    StartLives = StartLives - 1;
    cout << StartLives;
```

STEP 8

The previous step creates the variable StartLives, which is a global variable. In a game, for example, a player's lives go up or down depending on how well or how bad they're doing. When the player restarts the game, the StartLives returns to its default state: 3. Here we've assigned 3 lives, then subtracted 1, leaving 2 lives left.

STEP 9

The modern C++ compiler is far more intelligent than most programmers give it credit. While there are numerous data types you can declare for variables, you can in fact use the auto feature:

```
#include <iostream>
using namespace std;
auto pi = 3.141593;

int main()
{
    double area, radius = 1.5;
    area = pi * radius * radius;
    cout << area;
```

STEP 10

Although we said to ignore the warnings in Step 3, there's a difference between a warning and an error. In this case, it's just a simple warning. The new data type, double, means double-precision floating point value, which makes the code more accurate. The result should be 7.06858. Essentially, you can often work with a warning but not an error.



Data Types

Variables, as you've seen, store information that the programmer can then later call up, and manipulate if required. Variables are simply reserved memory locations that store the values the programmer assigns, depending on the data type used.

THE VALUE OF DATA

There are many different data types available for the programmer in C++, such as an integer, floating point, Boolean, character and so on. It's widely accepted that there are seven basic data types, often called Primitive Built-in Types; however, you can create your own data types should the need ever arise within your code.

Type	Command
Integer	<code>int</code>
Floating Point	<code>float</code>
Character	<code>char</code>
Boolean	<code>bool</code>
Double Floating Point	<code>double</code>
Wide Character	<code>wchar_t</code>
No Value	<code>void</code>

These basic types can also be extended using the following modifiers: Long, Short, Signed and Unsigned. Basically this means the modifiers can expand the minimum and maximum range values for each data type. For example, the `int` data type has a default value range of -2147483648 to 2147483647, a fair value, you would agree.

Now, if you were to use one of the modifiers, the range alters:

- Unsigned int = 0 to 4294967295**
- Signed int = -2147483648 to 2147483647**
- Short int = -32768 to 32767**
- Unsigned Short int = 0 to 65,535**
- Signed Short int = -32768 to 32767**
- Long int = -2147483647 to 2147483647**
- Signed Long int = -2147483647 to 2147483647**
- Unsigned Long int = 0 to 4294967295**

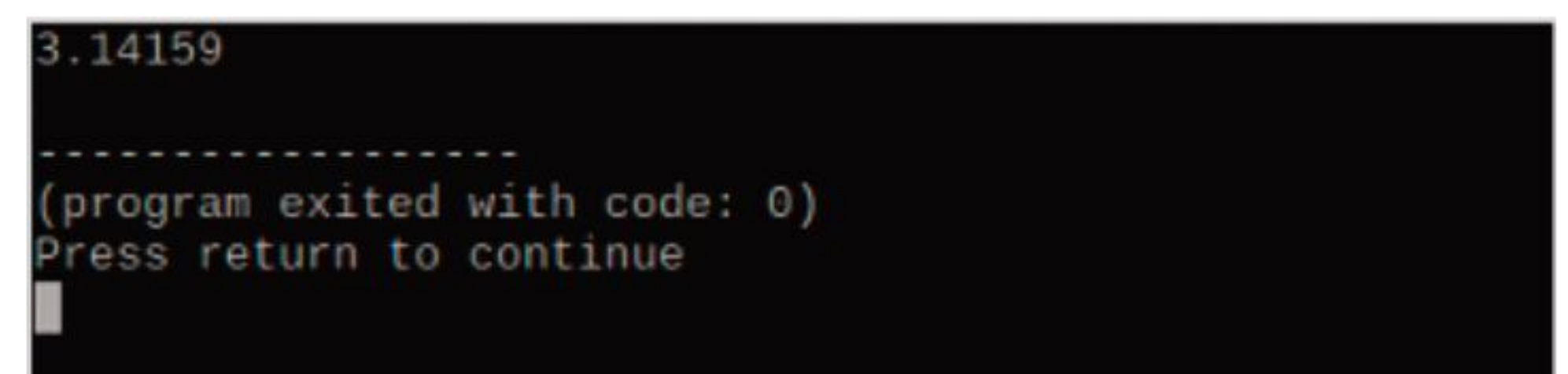
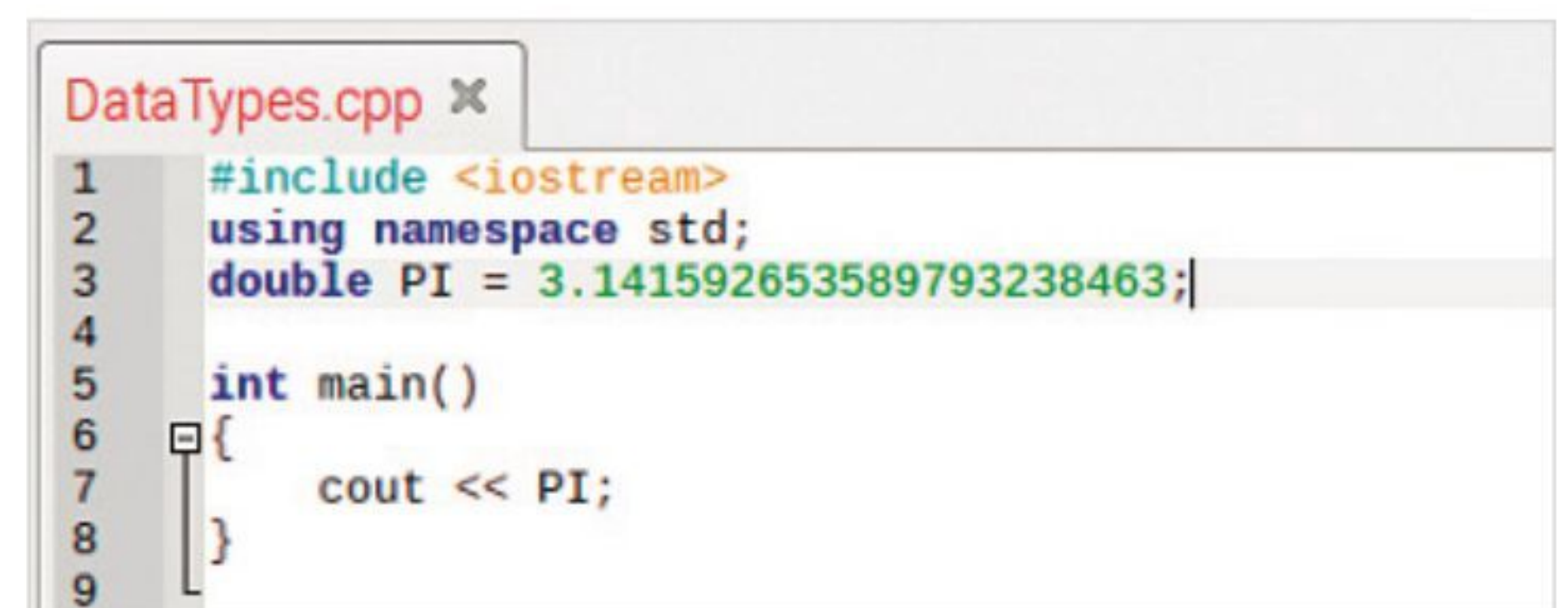
Naturally you can get away with using the basic type without the modifier, as there's plenty of range provided with each data type. However, it's considered good C++ programming practise to use the modifiers when possible.

There are issues when using the modifiers though. Double represents a double-floating point value, which you can use for incredibly accurate numbers but those numbers are only accurate up to the fifteenth decimal place. There's also the problem when displaying such numbers in C++ using the `cout` function, in that `cout` by default only outputs the first five decimal places. You can combat

that by adding a `cout.precision()` function and adding a value inside the brackets but even then you're still limited by the accuracy of the double data type. For example, try this code:

```
#include <iostream>
using namespace std;
double PI = 3.141592653589793238463;

int main()
{
    cout << PI;
}
```



Build and run the code and as you can see the output is only 3.14159, representing `cout`'s limitations in this example.

You can alter the code including the aforementioned `cout.precision` function, for greater accuracy. Take precision all the way up to 22 decimal places, with the following code:

```
#include <iostream>
using namespace std;
double PI = 3.141592653589793238463;

int main()
{
    cout.precision(22);
    cout << PI;
}
```



```
DataTypes.cpp
1 #include <iostream>
2 using namespace std;
3 double PI = 3.141592653589793238463;
4
5 int main()
6 {
7     cout.precision(22);
8     cout << PI;
9 }
10
```

```
3.141592653589793115998
-----
(program exited with code: 0)
Press return to continue
```

Again, build and run the code; as you can see from the command line window, the number represented by the variable PI is different to the number you've told C++ to use in the variable. The output reads the value of PI as 3.141592653589793115998, with the numbers going awry from the fifteenth decimal place.

Calculator Scientific

15.142857142857142857142857142857

DEG	HYP	F-E			
MC	MR	M+	M-	MS	M ⁺
x^2	x^y	sin	cos	tan	
$\sqrt{\quad}$	10^x	log	Exp	Mod	
\uparrow	CE	C	\leftarrow	\div	
π	7	8	9	\times	
n!	4	5	6	-	
\pm	1	2	3	+	
()	0	.	=	

This is mainly due to the conversion from binary in the compiler and that the IEEE 754 double precision standard occupies 64-bits of data, of which 52-bits are dedicated to the significant (the significant digits in a floating-point number) and roughly 3.5-bits are taken holding the values 0 to 9. If you divide 53 by 3.5, then you arrive at 15.142857 recurring, which is 15-digits of precision.

To be honest, if you're creating code that needs to be accurate to more than fifteen decimal places, then you wouldn't be using C++, you would use some scientific specific language with C++ as the connective tissue between the two languages.

You can create your own data types, using an alias-like system called typedef. For example:

```
#include <iostream>
using namespace std;
typedef int metres;

int main()
{
    metres distance;
    distance = 15;
    cout << "distance in metres is: " << distance;
}
}
```

```
DataTypes.cpp
1 #include <iostream>
2 using namespace std;
3 typedef int metres;
4
5 int main()
6 {
7     metres distance;
8     distance = 15;
9     cout << "Distance in metres is: " << distance;
10 }
11 }
12
```

```
sh
File Edit Tabs Help
Distance in metres is: 15
-----
(program exited with code: 0)
Press return to continue
```

This code when executed creates a new int data type called metres. Then, in the main code block, there's a new variable called distance, which is an integer; so you're telling the compiler that there's another name for int. We assigned the value 15 to distance and displayed the output: distance in metres is 15.

It might sound a little confusing to begin with but the more you use C++ and create your own code, the easier it becomes.



Strings

Strings are objects that represent and hold sequences of characters. For example, you could have a universal greeting in your code 'Welcome' and assign that as a string to be called up wherever you like in the program.

STRING THEORY

There are different ways in which you can create a string of characters, which historically are all carried over from the original C language and still supported by C++.

STEP 1 To create a string, you use the char function. Open a new C++ file and begin with the usual header:

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

int main ()
{
}
```

STEP 2 It's easy to confuse a string with an array. Here's an array, which can be terminated with a null character:

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

int main ()
{
    char greet[8] = {'W', 'e', 'l', 'c', 'o', 'm', 'e', '\0'};
    cout << greet << "\n";
}
```

STEP 3 Build and run the code and 'Welcome' appears on the screen. While this is perfectly fine, it's not a string. A string is a class, which defines objects that can be represented as a stream of characters and doesn't need to be terminated like an array. The code can therefore be represented as:

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

int main ()
{
    char greet[] = "Welcome";
    cout << greet << "\n";
}
```

STEP 4 In C++ there's also a string function, which works in much the same way. Using the greeting code again, you can enter:

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

int main ()
{
    string greet = "Welcome";
    cout << greet << "\n";
}
```


**STEP 5**

There are also many different operations that you can apply with the string function. For instance, to get the length of a string you can use:

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

int main ()
{
    string greet = "Welcome";
    cout << "The length of the string is: ";
    cout << greet.size() << "\n";
}
```

```
strings.cpp x
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      string greet = "Welcome";
7      cout << "The length of the string is: ";
8      cout << greet.size() << "\n";
9
10 }
11
```

STEP 6

You can see that we used `greet.size()` to output the length, the number of characters there are, of the contents of the string. Naturally, if you call your string something other than `greet`, then you need to change the command to reflect this. It's always `stringname.operation`. Build and run the code to see the results.

```
The length of the string is: 7
-----
(program exited with code: 0)
Press return to continue
```

STEP 7

You can of course add strings together, or rather combine them to form longer strings:

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

int main ()
{
    string greet1 = "Hello";
    string greet2 = ", world!";
    string greet3 = greet1 + greet2;

    cout << greet3 << "\n";
}
```

```
strings.cpp x
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      string greet1 = "Hello";
7      string greet2 = ", world!";
8      string greet3 = greet1 + greet2;
9
10     cout << greet3 << "\n";
11 }
12
13
```

STEP 8

Just as you might expect, you can mix in an integer and store something to do with the string. In this example, we created `int length`, which stores the result of `string.size()` and outputs it to the user:

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

int main ()
{
    int length;
    string greet1 = "Hello";
    string greet2 = ", world!";
    string greet3 = greet1 + greet2;

    length = greet3.size();

    cout << "The length of the combined strings
is: " << length << "\n";
}
```

STEP 9

Using the available operations that come with the string function, you can manipulate the contents of a string. For example, to remove characters from a string you could use:

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

int main ()
{
    string strg ("Here is a long sentence in a
string.");
    cout << strg << '\n';

    strg.erase (10,5);
    cout << strg << '\n';

    strg.erase (strg.begin()+8);
    cout << strg << '\n';

    strg.erase (strg.begin()+9, strg.end()-9);
    cout << strg << '\n';
}
```

STEP 10

It's worth spending some time playing around with the numbers, which are the character positions in the string. Occasionally, it can be hit and miss whether you get it right, so practice makes perfect. Take a look at the screenshot to see the result of the code.

```
sh
File Edit Tabs Help
Here is a long sentence in a string.
Here is a sentence in a string.
Here is sentence in a string.
Here is a string.
-----
(program exited with code: 0)
Press return to continue
```



C++ Maths

Programming is mathematical in nature and as you might expect, there's plenty of built-in scope for some quite intense maths. C++ has a lot to offer someone who's implementing mathematical models into their code. It can be extremely complex or relatively simple.

C++ = MC2

The basic mathematical symbols apply in C++ as they do in most other programming languages. However, by using the C++ Math Library, you can also calculate square roots, powers, trig and more.

STEP 1 C++'s mathematical operations follow the same patterns as those taught in school, in that multiplication and division take precedence over addition and subtraction. You can alter that though. For now, create a new file and enter:

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

int main ()
{
    float numbers = 100;

    numbers = numbers + 10; // This adds 10 to the
    initial 100

    cout << numbers << "\n";

    numbers = numbers - 10; // This subtracts 10
    from the new 110

    cout << numbers << "\n";
}
```

STEP 2 While simple, it does get the old maths muscle warmed up. Note that we used a float for the numbers variable. While you can happily use an integer, if you suddenly started to use decimals, you would need to change to a float or a double, depending on the accuracy needed. Run the code and see the results.

STEP 3 Multiplication and division can be applied as such:

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

int main ()
{
    float numbers = 100;

    numbers = numbers * 10; // This multiplies 100
    by 10

    cout << numbers << "\n";

    numbers = numbers / 10; // And this divides
    1000 by 10

    cout << numbers << "\n";
}
```

STEP 4 Again, execute the simple code and see the results. While not particularly interesting, it's an introduction to C++ maths. We used a float here, so you can play around with the code and multiply by decimal places, as well as divide, add and subtract.

**STEP 5**

The interesting maths content comes when you call upon the C++ Math Library. Within this header are dozens of mathematical functions along with further operations. Everything from computing cosine to arc tangent with two parameters, to the value of PI. You can call the header with:

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

int main ()
{
}
```

```
maths.cpp x
1 #include <iostream>
2 #include <cmath>
3 using namespace std;
4
5 int main()
6 {
7
8
9
10 }
11
```

STEP 6

Start by getting the square root of a number:

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

int main ()
{
    float number = 134;

    cout << "The square root of " << number << "
is: " << sqrt(number) << "\n";
}
```

```
maths.cpp x
1 #include <iostream>
2 #include <cmath>
3 using namespace std;
4
5 int main()
6 {
7     float number = 134;
8
9     cout << "The square root of " << number << " is: "
10
11 }
12
13
```

STEP 7

Here we created a new float called number and used the sqrt(number) function to display the square root of 134, the value of the variable, number. Build and run the code and your answer reads 11.5758.

```
The square root of 134 is: 11.5758
-----
(program exited with code: 0)
Press return to continue
```

STEP 8

Calculating powers of numbers can be done with:

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

int main ()
{
    float number = 12;

    cout << number << " to the power of 2 is " <<
pow(number, 2) << "\n";
    cout << number << " to the power of 3 is " <<
pow(number, 3) << "\n";
    cout << number << " to the power of 0.8 is "
<< pow(number, 0.8) << "\n";
}
```

```
maths.cpp x
1 #include <iostream>
2 #include <cmath>
3 using namespace std;
4
5 int main()
6 {
7     float number = 12;
8
9     cout << number << " to the power of 2 is " << pow(number, 2) << "\n";
10
11     cout << number << " to the power of 3 is " << pow(number, 3) << "\n";
12
13     cout << number << " to the power of 0.8 is " << pow(number, 0.8) << "\n";
14
15 }
```

STEP 9

Here we created a float called number with the value of 12 and the pow(variable, power) is where the calculation happens. Of course, you can calculate powers and square roots without using variables. For example, pow(12, 2) outputs the same value as the first cout line in the code.

```
12 to the power of 2 is 144
12 to the power of 3 is 1728
12 to the power of 0.8 is 7.30037
-----
(program exited with code: 0)
Press return to continue
```

STEP 10

The value of Pi is also stored in the cmath header library. It can be called up with the M_PI function. Enter cout << M_PI; into the code and you get 3.14159; or you can use it to calculate:

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

int main ()
{
    double area, radius = 1.5;

    area = M_PI * radius * radius;

    cout << area << "\n";
}
```

```
maths.cpp x
1 #include <iostream>
2 #include <cmath>
3 using namespace std;
4
5 int main()
6 {
7     double area, radius = 1.5;
8
9     area = M_PI * radius * radius;
10
11     cout << area << "\n";
12
13 }
14
15
```

```
sh
File Edit Tabs Help
7.06858
-----
(program exited with code: 0)
Press return to continue
```



User Interaction

There's nothing quite as satisfying as creating a program that responds to you. This basic user interaction is one of the most taught aspects of any language and with it you're able to do much more than simply greet the user by name.

HELLO, DAVE

You have already used `cout`, the standard output stream, throughout our code. Now you're going to be using `cin`, the standard input stream, to prompt a user response.

STEP 1 Anything that you want the user to input into the program needs to be stored somewhere in the system memory, so it can be retrieved and used. Therefore, any input must first be declared as a variable, so it's ready to be used by the user. Start by creating a blank C++ file with headers.

```
userinteraction.cpp x
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6
7
8
9
10 }
```

STEP 2 The data type of the variable must also match the type of input you want from the user. For example, to ask a user their age, you would use an integer like this:

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

int main ()
{
    int age;
    cout << "what is your age: ";
    cin >> age;

    cout << "\nYou are " << age << " years old.\n";
}
```

```
userinteraction.cpp x
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     int age;
7     cout << "what is your age: ";
8     cin >> age;
9
10    cout << "\nYou are " << age << " years old.\n";
11
12
13
14 }
```

STEP 3 The `cin` command works in the opposite way from the `cout` command. With the first `cout` line you're outputting 'What is your age' to the screen, as indicated with the chevrons. `Cin` uses opposite facing chevrons, indicating an input. The input is put into the integer `age` and called up in the second `cout` command. Build and run the code.

```
sh
File Edit Tabs Help
what is your age: 45
You are 45 years old.
-----
(program exited with code: 0)
Press return to continue
```

STEP 4 If you're asking a question, you need to store the input as a string; to ask the user their name, you would use:

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

int main ()
{
    string name;
    cout << "what is your name: ";
    cin >> name;

    cout << "\nHello, " << name << ". I hope you're well today?\n";
}
```

```
userinteraction.cpp x
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     string name;
7     cout << "what is your name: ";
8     cin >> name;
9
10    cout << "\nHello, " << name << ". I hope you're well today?\n";
11
12
13
14 }
```



STEP 5 The principal works the same as the previous code. The user's input, their name, is stored in a string, because it contains multiple characters, and retrieved in the second cout line. As long as the variable 'name' doesn't change, then you can recall it wherever you like in your code.

```
What is your name: David
Hello, David. I hope you're well today>
-----
(program exited with code: 0)
Press return to continue
```

STEP 6 You can chain input requests to the user but just make sure you have a valid variable to store the input to begin with. Let's assume you want the user to enter two whole numbers:

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

int main ()
{
    int num1, num2;

    cout << "Enter two whole numbers: ";
    cin >> num1 >> num2;

    cout << "you entered " << num1 << " and " <<
num2 << "\n";
}
```

```
userinteraction.cpp x
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     int num1, num2;
7
8     cout << "Enter two whole numbers: ";
9     cin >> num1 >> num2;
10
11     cout << "You entered " << num1 << " and " << num2 << "\n";
12
```

STEP 7 Likewise, inputted data can be manipulated once you have it stored in a variable. For instance, ask the user for two numbers and do some maths on them:

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

int main ()
{
    float num1, num2;

    cout << "Enter two numbers: \n";
    cin >> num1 >> num2;

    cout << num1 << " + " << num2 << " is: " <<
num1 + num2 << "\n";
}
```

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     int num1, num2;
7
8     cout << "Enter two whole numbers: ";
9     cin >> num1 >> num2;
10
11     cout << num1 << " + " << num2 << " is: " << num1 + num2 << "\n";
12
```

STEP 8 While cin works well for most input tasks, it does have a limitation. Cin always considers spaces as a terminator, so it's designed for just single words not multiple words. However, getline takes cin as the first argument and the variable as the second:

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

int main ()
{
    string mystr;
    cout << "Enter a sentence: \n";
    getline(cin, mystr);

    cout << "Your sentence is: " << mystr.size() <<
" characters long.\n";
}
```

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     string mystr;
7     cout << "Enter a sentence: \n";
8     getline(cin, mystr);
9
10    cout << "Your sentence is: " << mystr.size() << " characters long.\n";
11
12
```

STEP 9 Build and execute the code, then enter a sentence with spaces. When you're done the code reads the number of characters. If you remove the getline line and replace it with cin >> mystr and try again, the result displays the number of characters up to the first space.

```
Enter a sentence:
BDM Publications' Raspberry Pi: Tricks, Hacks & Fixes
Your sentence is: 53 charcters long.
```

```
-----
(program exited with code: 0)
Press return to continue
```

STEP 10 Getline is usually a command that new C++ programmers forget to include. The terminating white space is annoying when you can't figure out why your code isn't working. In short, it's best to use getline(cin, variable) in future:

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

int main ()
{
    string name;
    cout << "Enter your full name: \n";
    getline(cin, name);

    cout << "\nHello, " << name << ".\n";
}
```

```
userinteraction.cpp x
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     string name;
7     cout << "Enter your full name: \n";
8     getline(cin, name);
9
10    cout << "\nHello, " << name << ".\n";
11
```



The Hobbit

DID YOU KNOW...

released in 1982 for various platforms, Melbourne House's The Hobbit was a coding work of genius. Developed by Philip Mitchell and Dr. Veronika Megler, The Hobbit was uniquely coded using a parser called 'English' which allowed the player to enter full sentences, such as "Take sword from Gandalf, and kill goblin with it."

Furthermore, each object in the game had a specific size, weight and solidity; so if you sat on a stool, and someone picked up the stool or threw it, then your

character would be taken along with it. Also, in early versions of the game, the non-playing characters would often move around the map and capture and kill each other before the player had even reached that stage in the game. There was even a randomiser event in place with each start of the game, where any character could pick up a random object from any location. This meant that, potentially, it was possible for Thorin to pick you up and carry you for the entire adventure. Remarkable for a game that took up less than 38KB of memory.





Sinclair

48K ZX Spectrum

The Hobbit

THE HOBBIT is a super-program that is a milestone in computer software. You will face dangers, excitement and adventure in words and graphics. Meet all the characters from THE HOBBIT and talk to them in ordinary English! THE HOBBIT program brings you the future in an exciting and challenging fantasy!

MELBOURNE HOUSE



The Hobbit, a marvel in advanced coding from 1982.



Common Coding Mistakes

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

X=MISTAKE, PRINT Y

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



SMALL CHUNKS

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



EASY VARIABLES

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

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

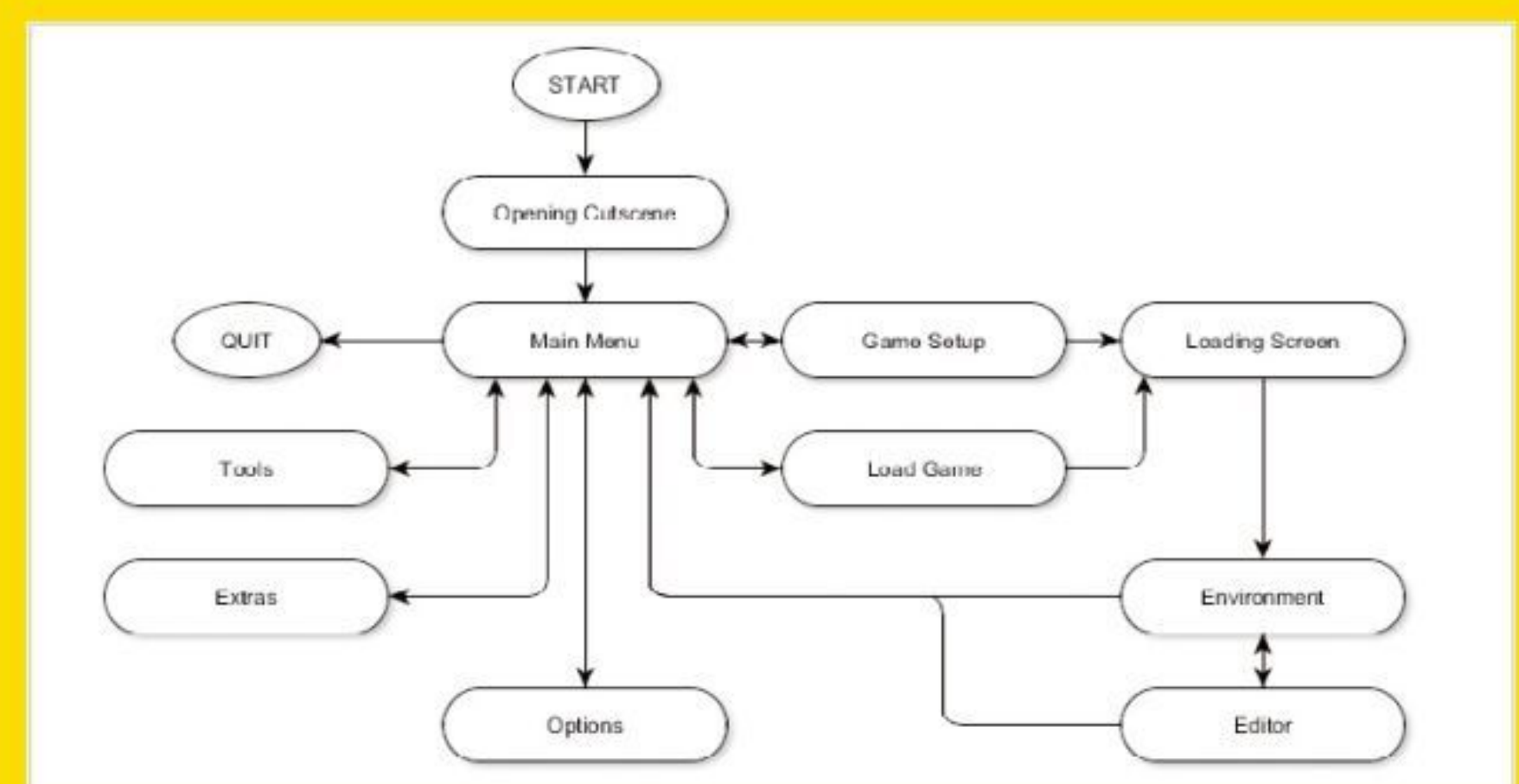
//COMMENTS

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

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

PLAN AHEAD

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





USER ERROR

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

```
Enter an integer number
aswdfdsf
You have entered wrong input
s
You have entered wrong input
!"£"!£!"
You have entered wrong input
sdfsd213213123
You have entered wrong input
123234234234234234
You have entered wrong input
12
the number is: 12

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

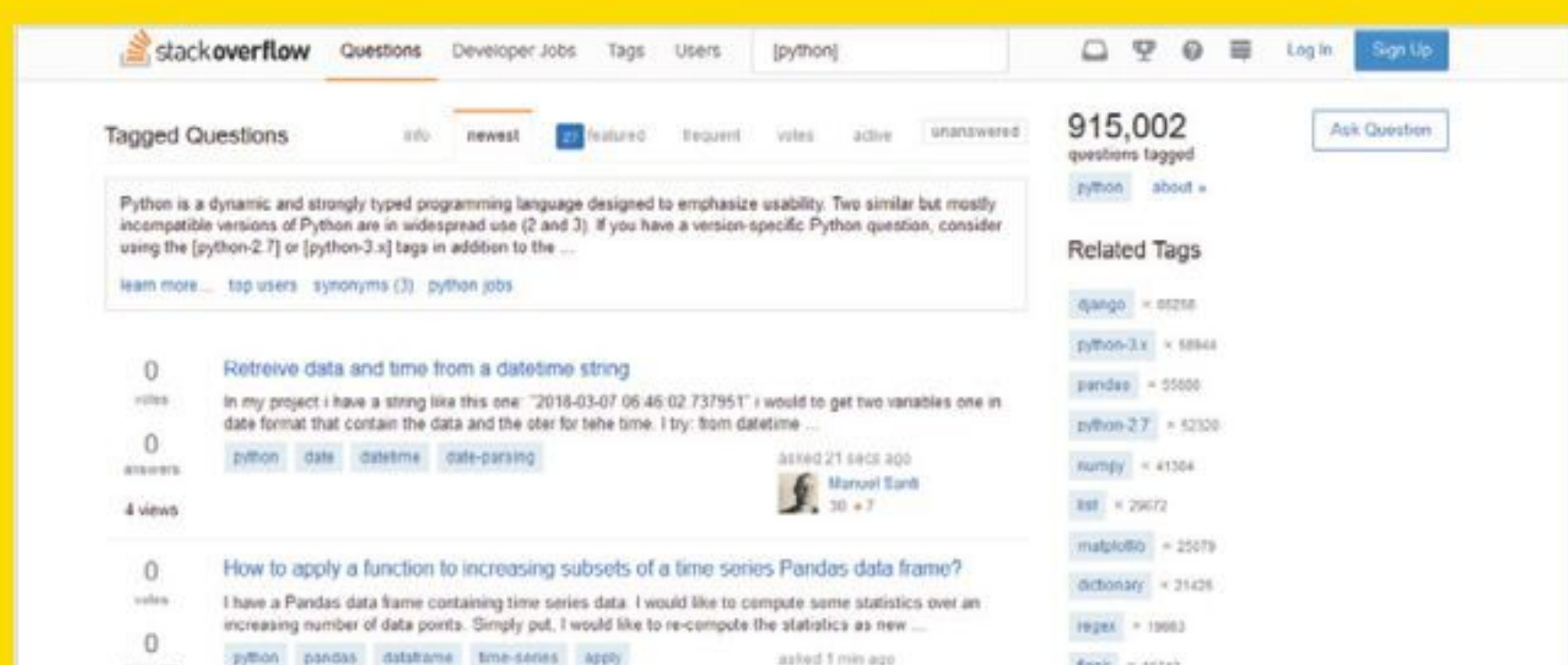
RE-INVENTING WHEELS

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



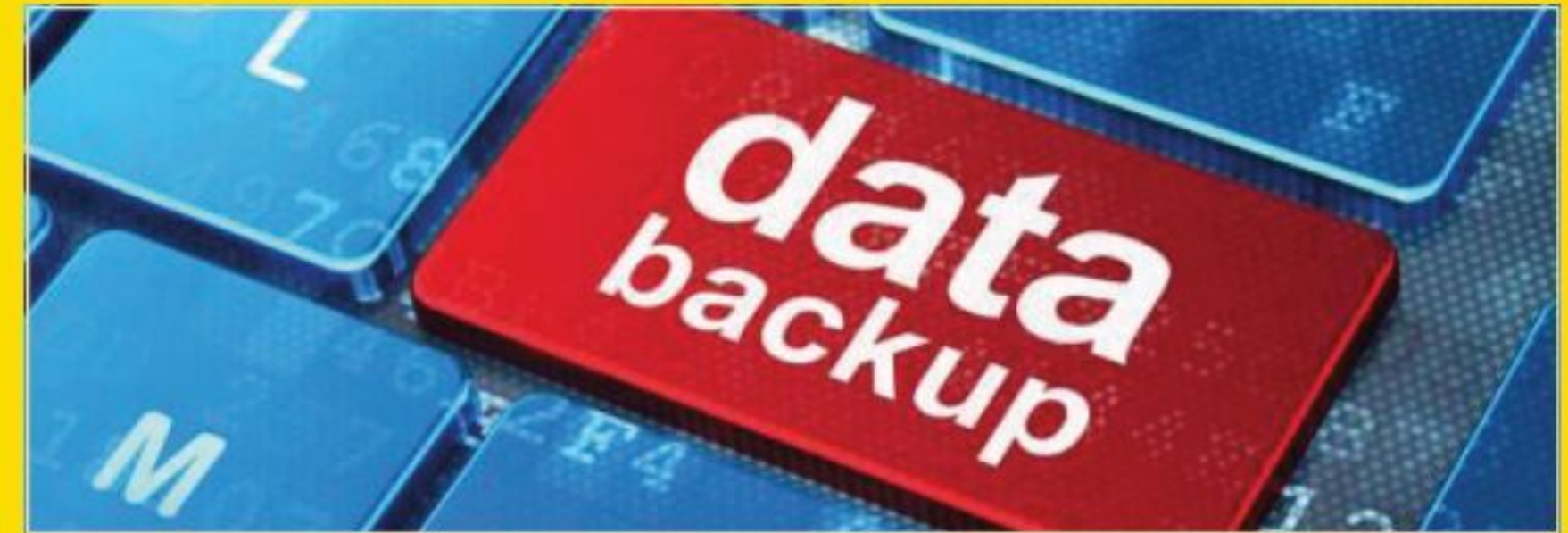
HELP!

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



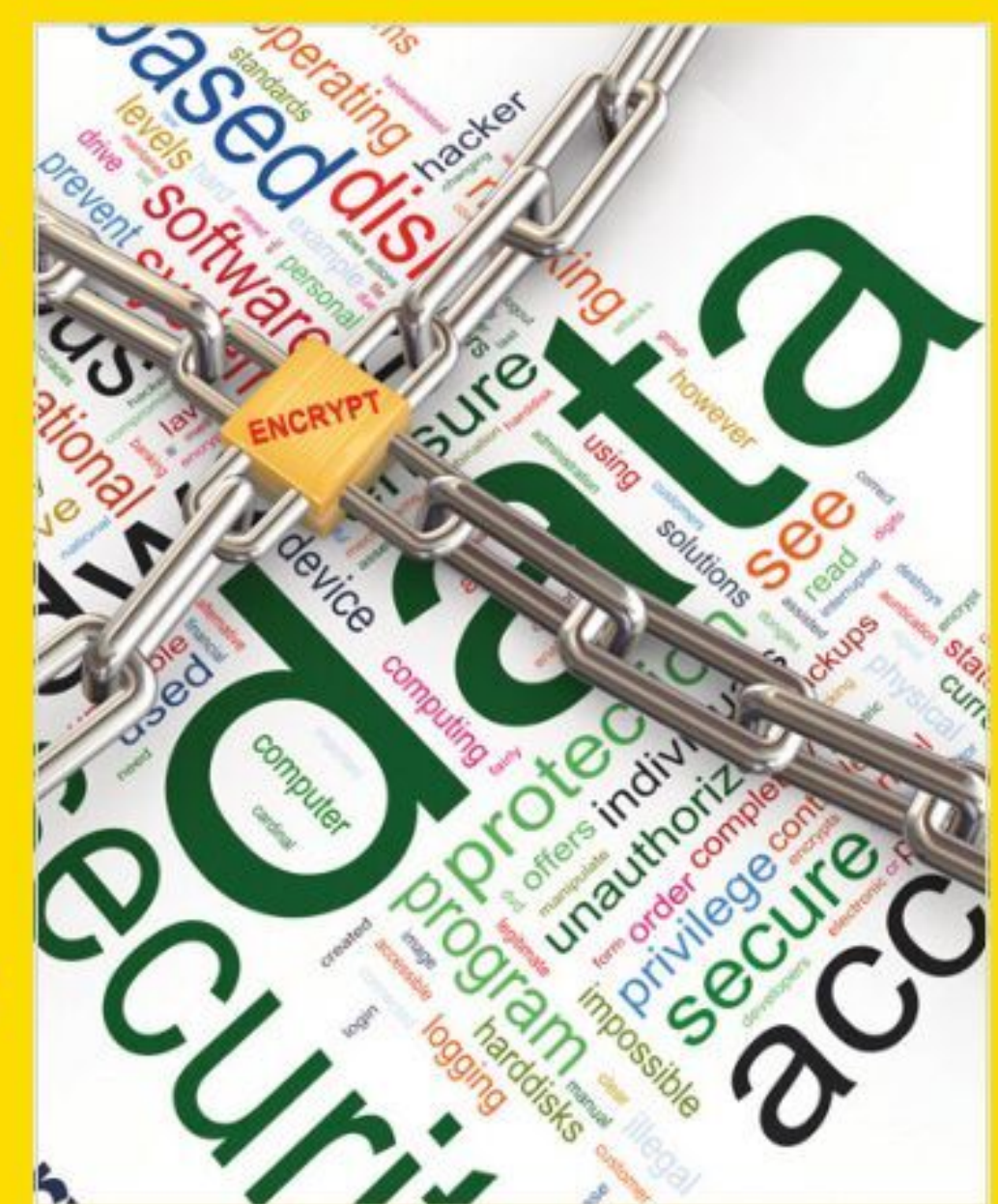
BACKUPS

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



SECURE DATA

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



MATHS

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

```
set terminal x11
set output

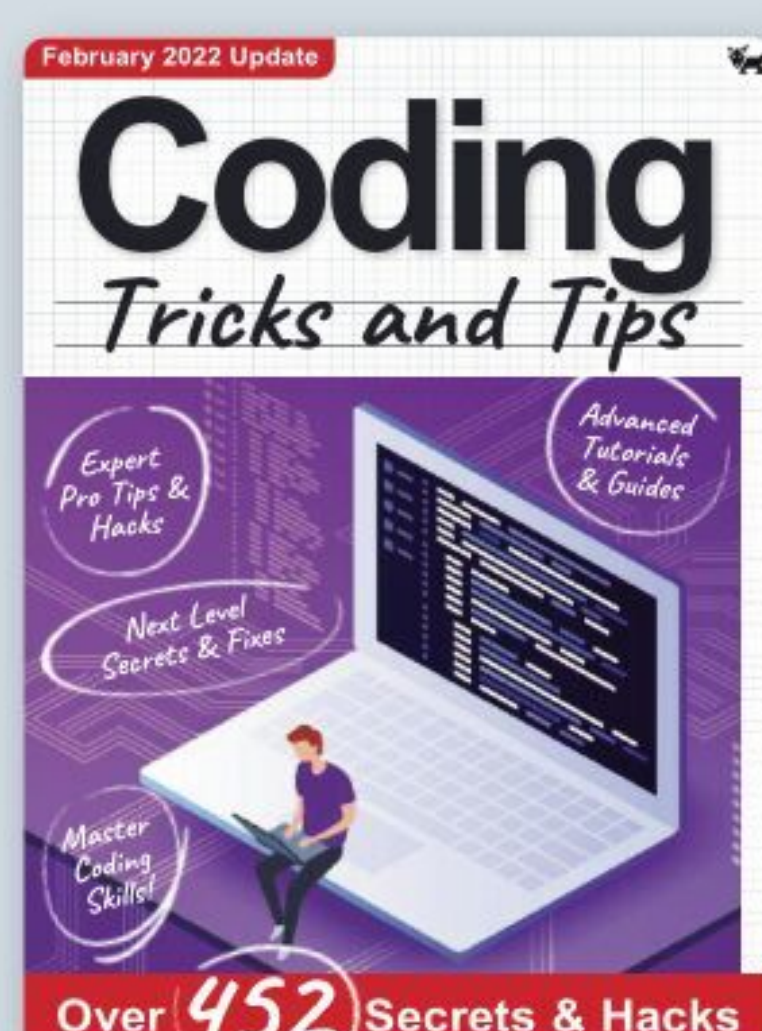
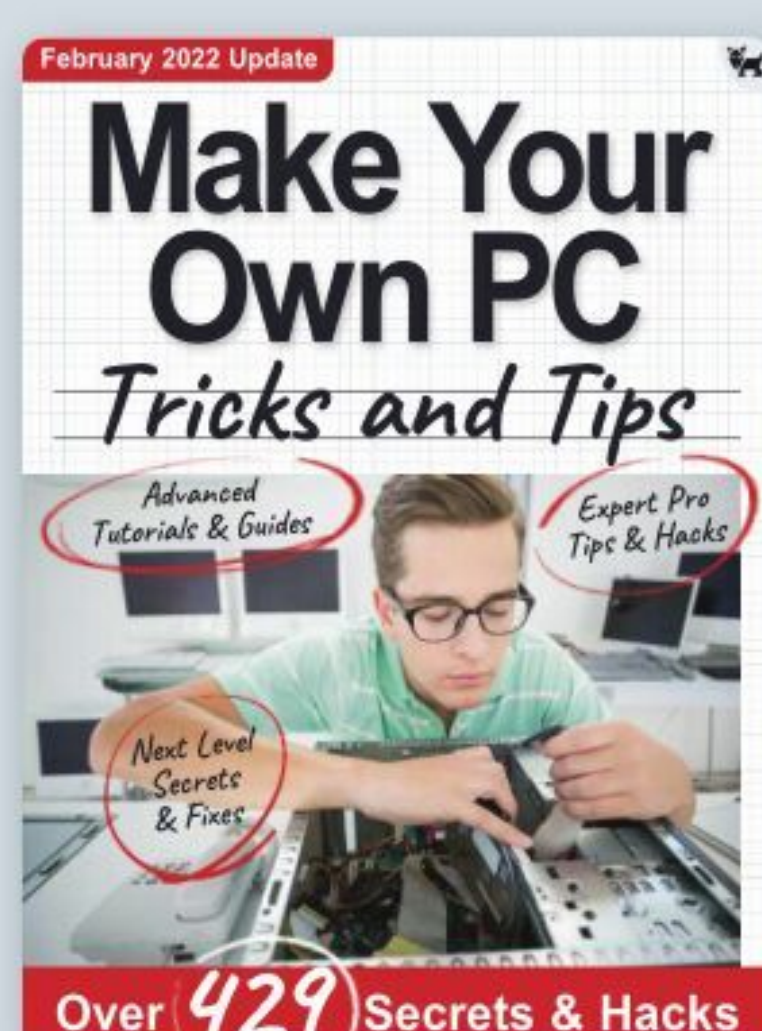
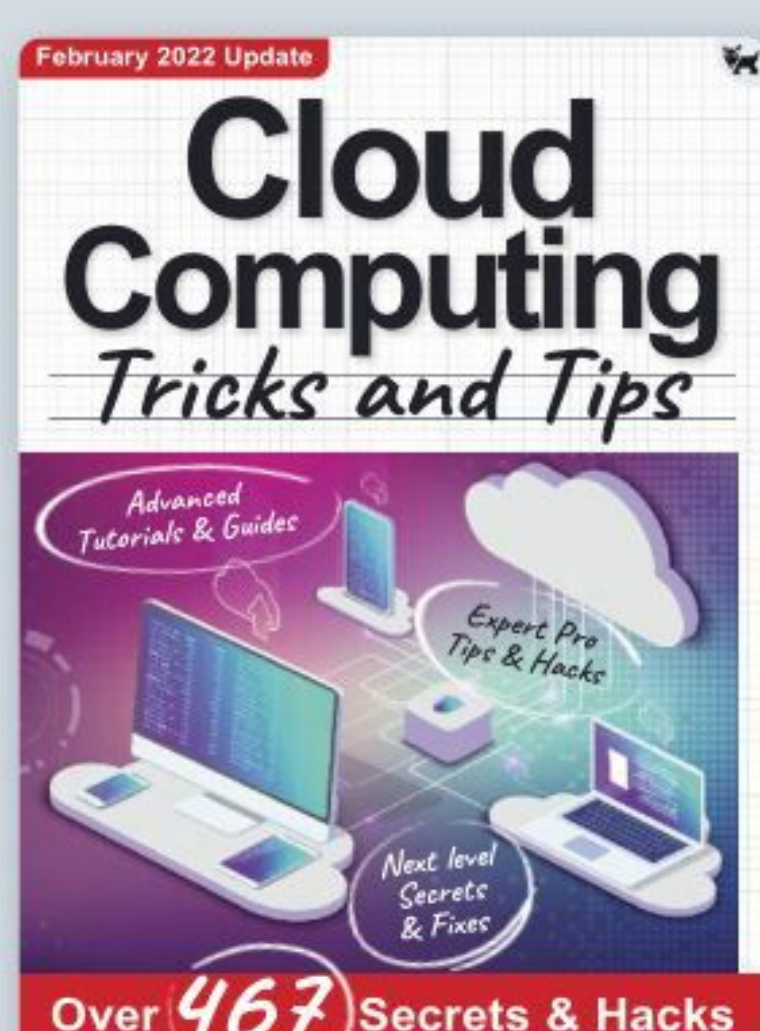
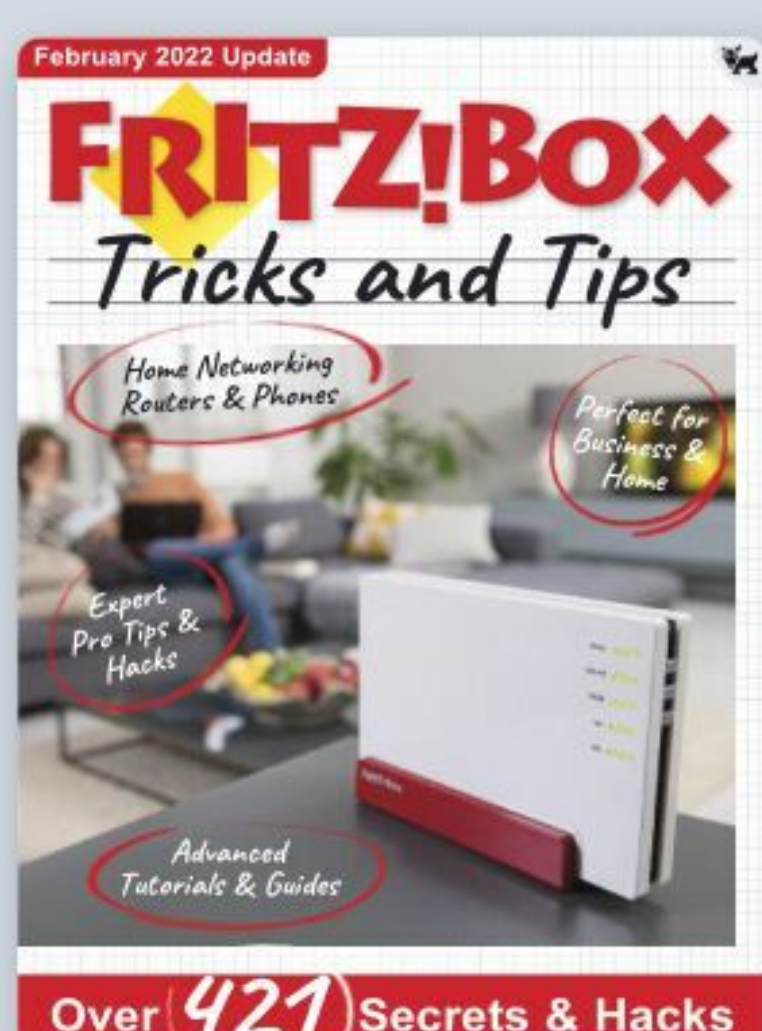
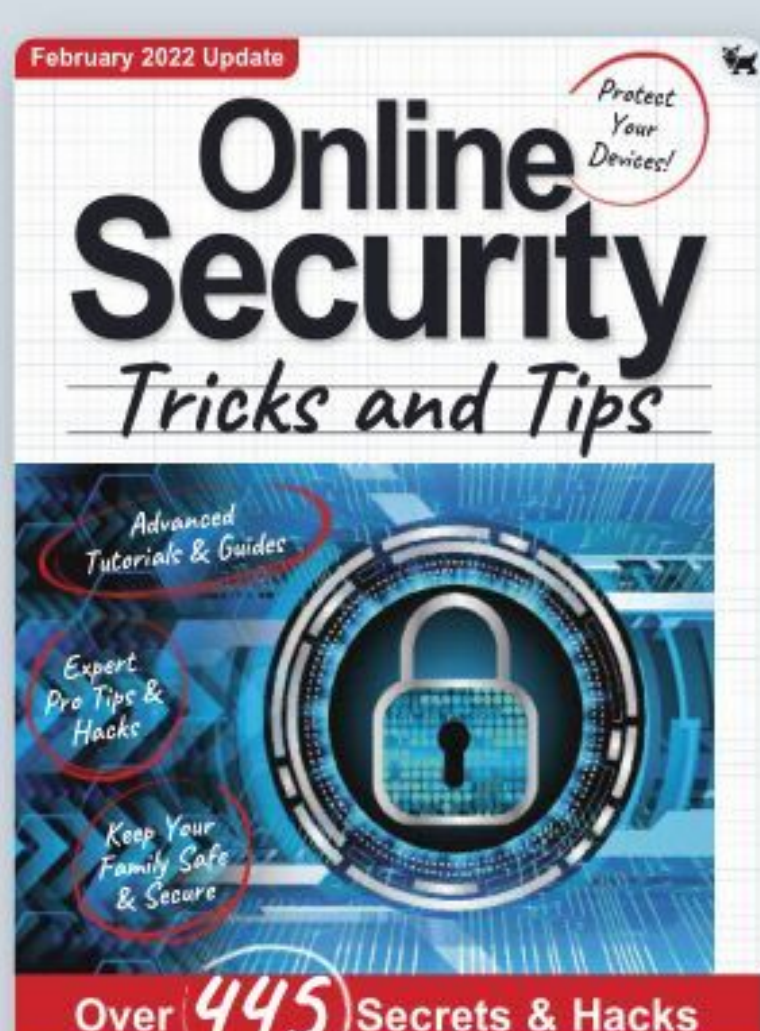
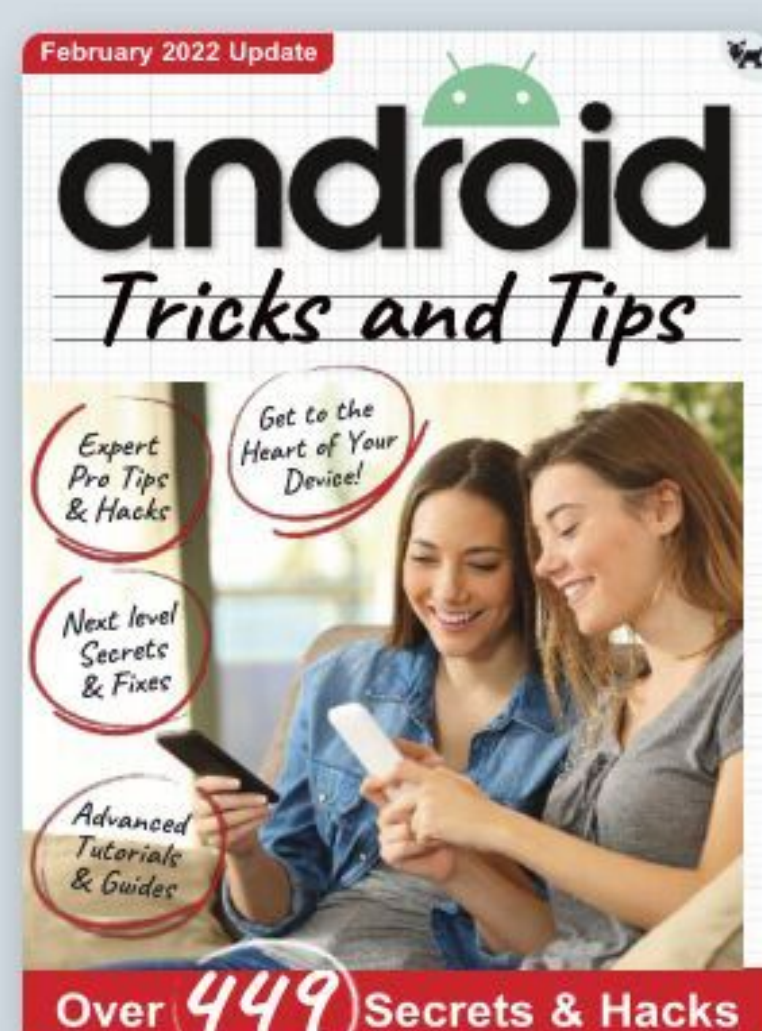
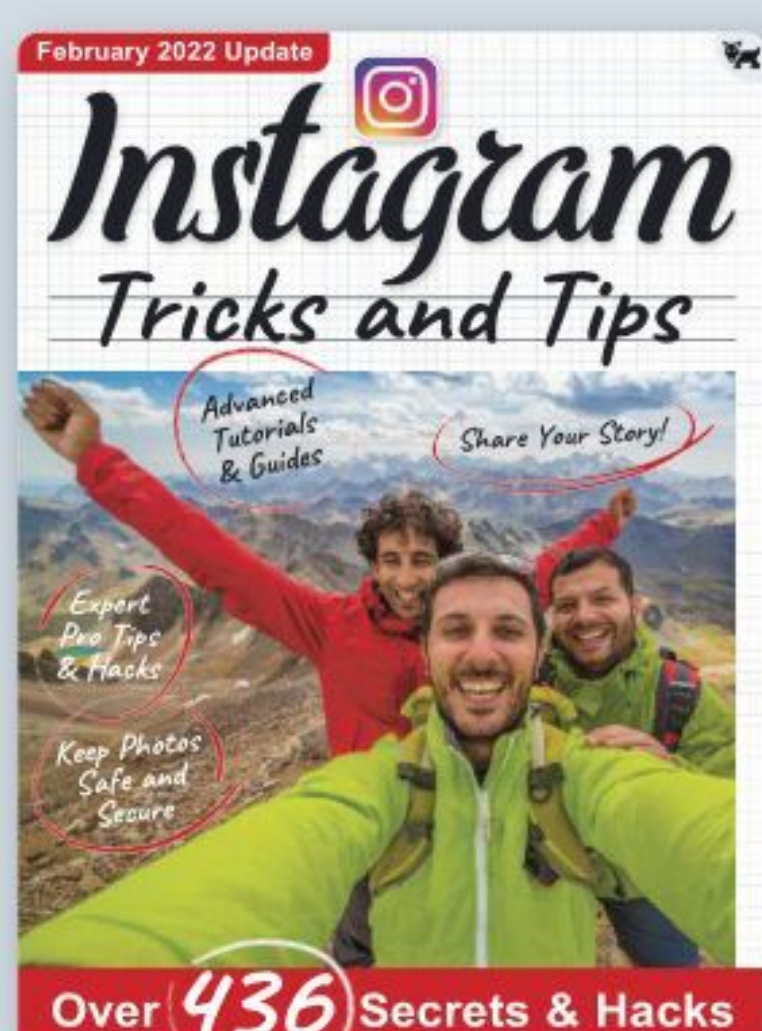
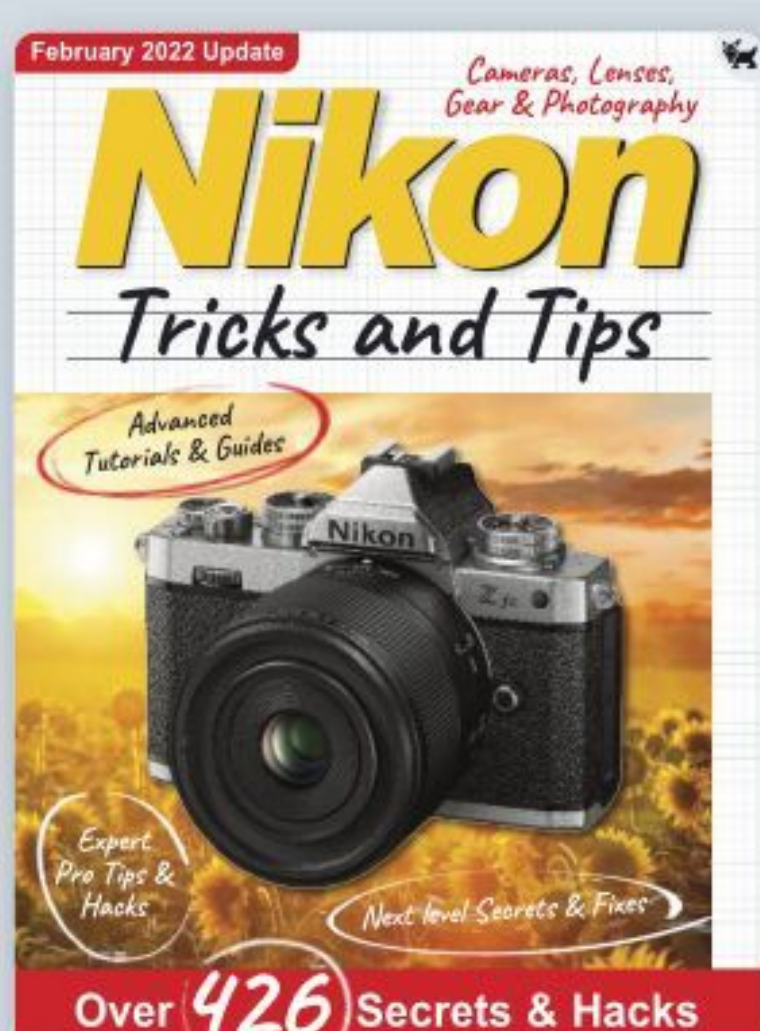
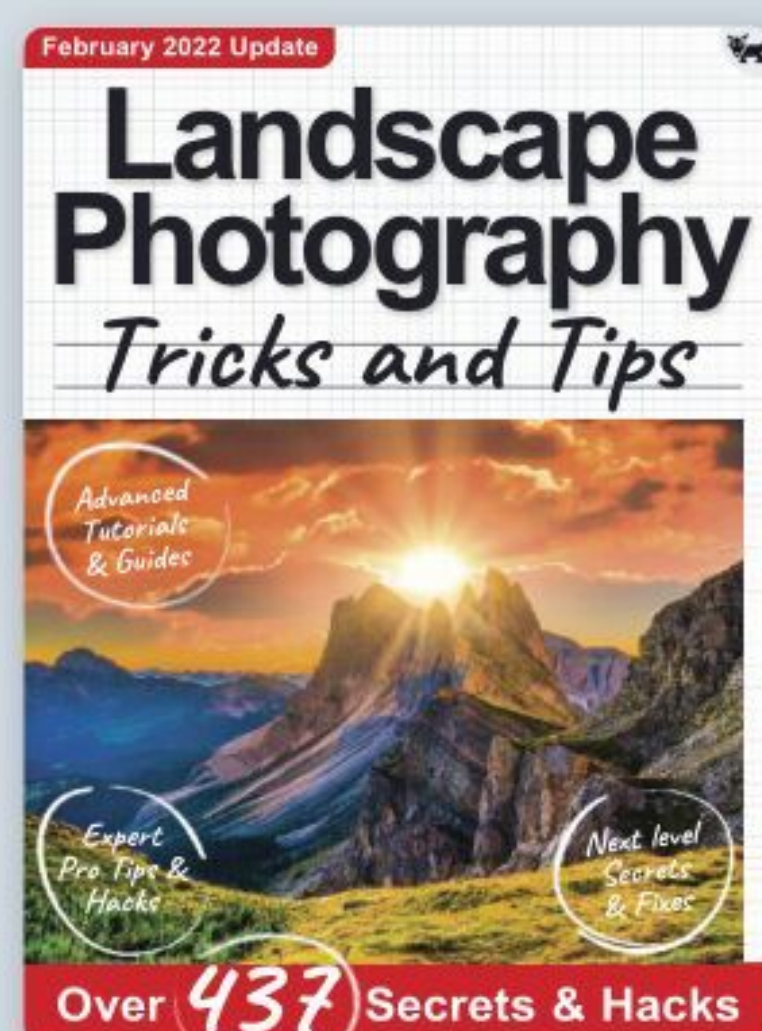
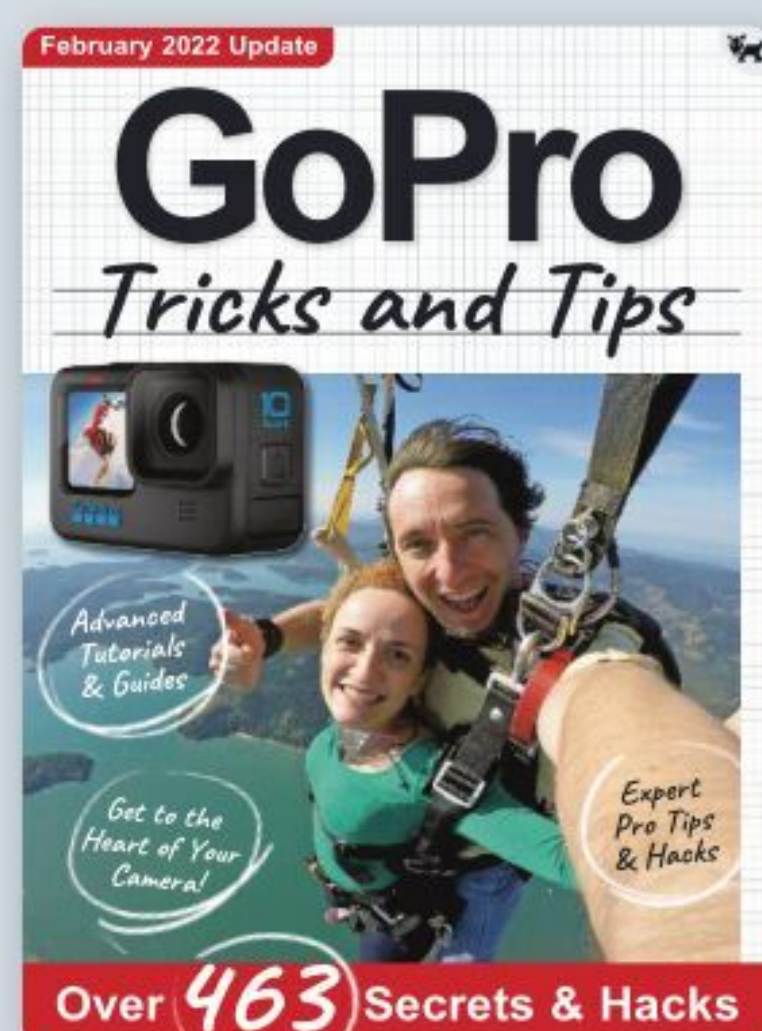
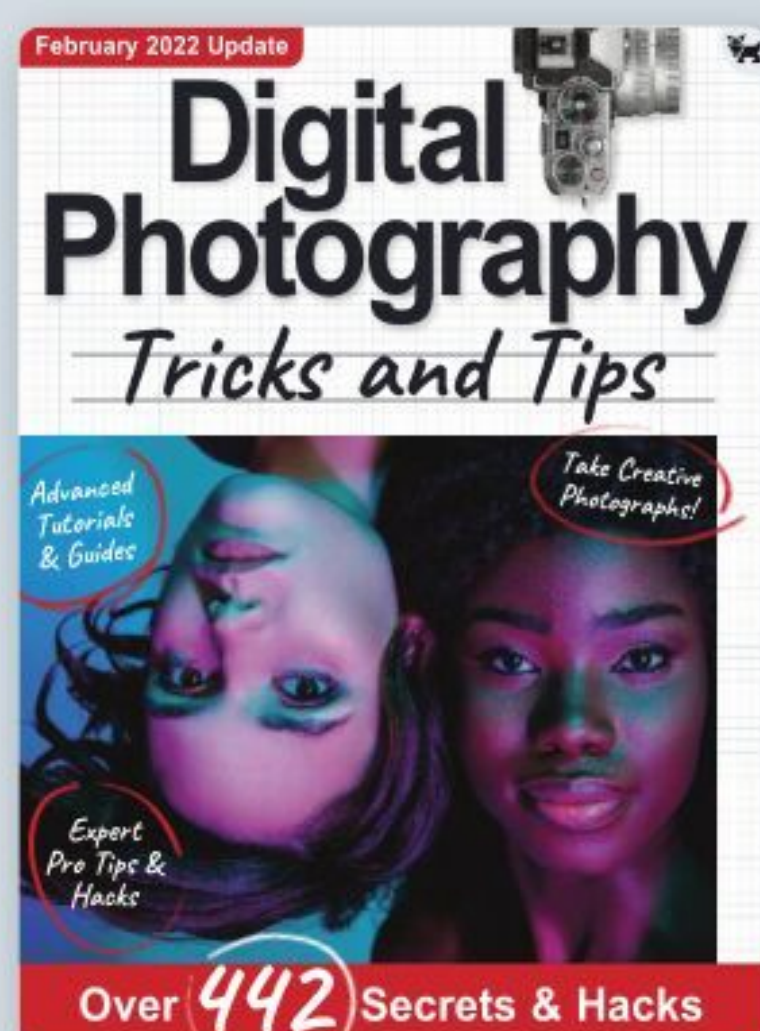
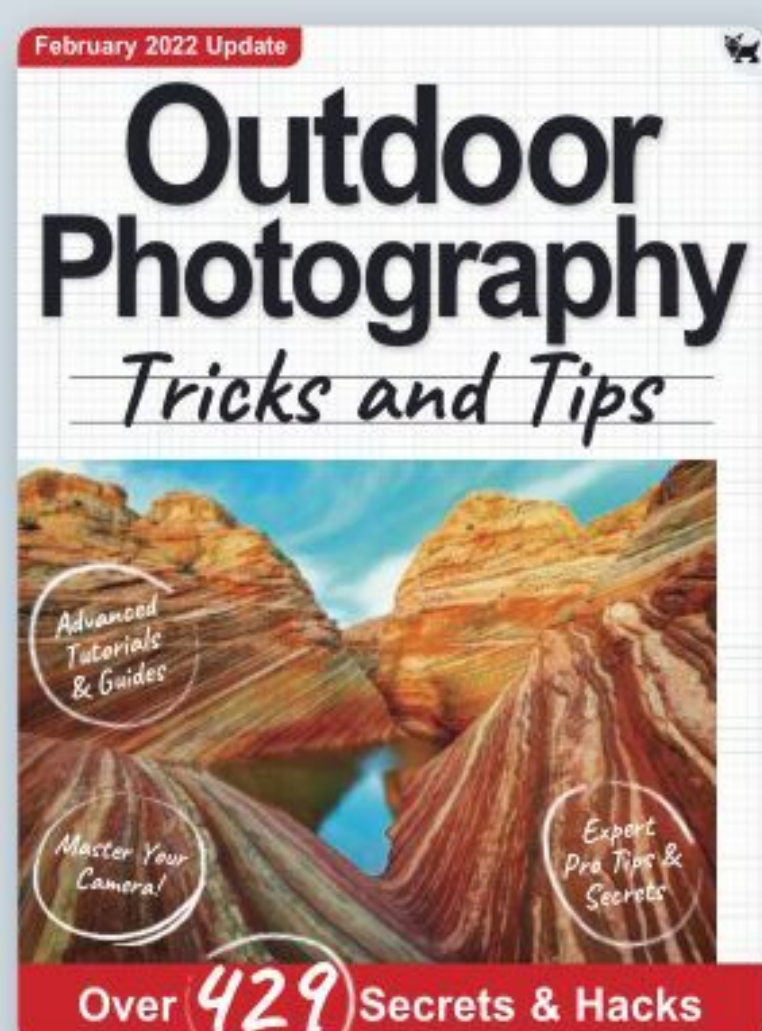
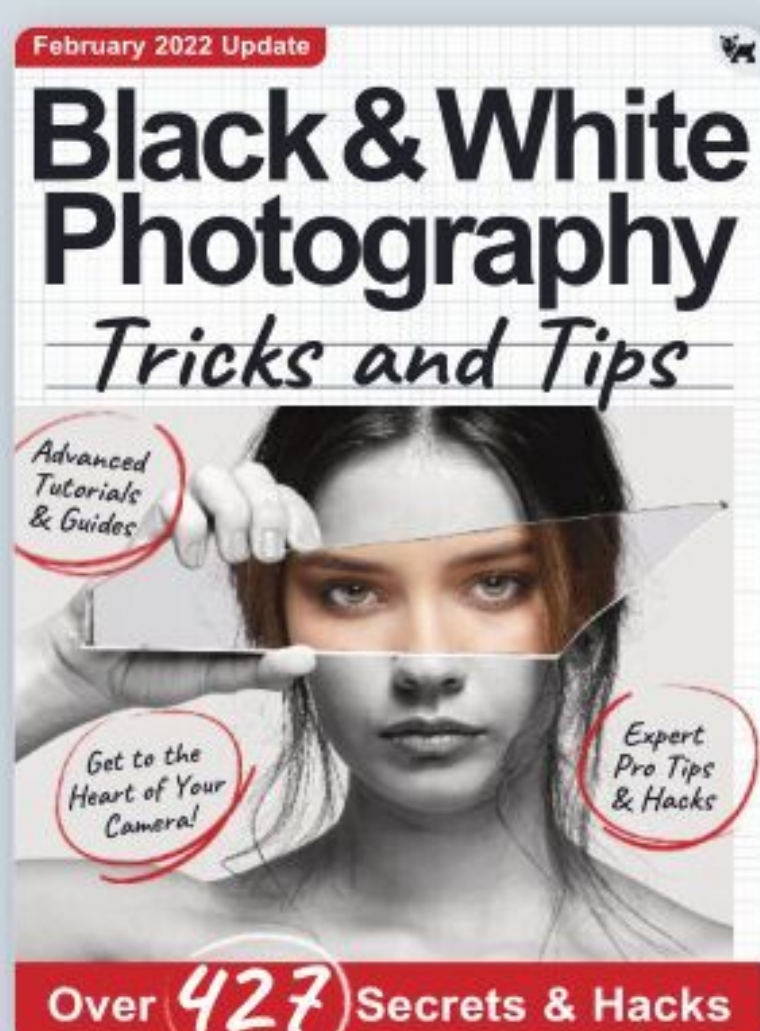
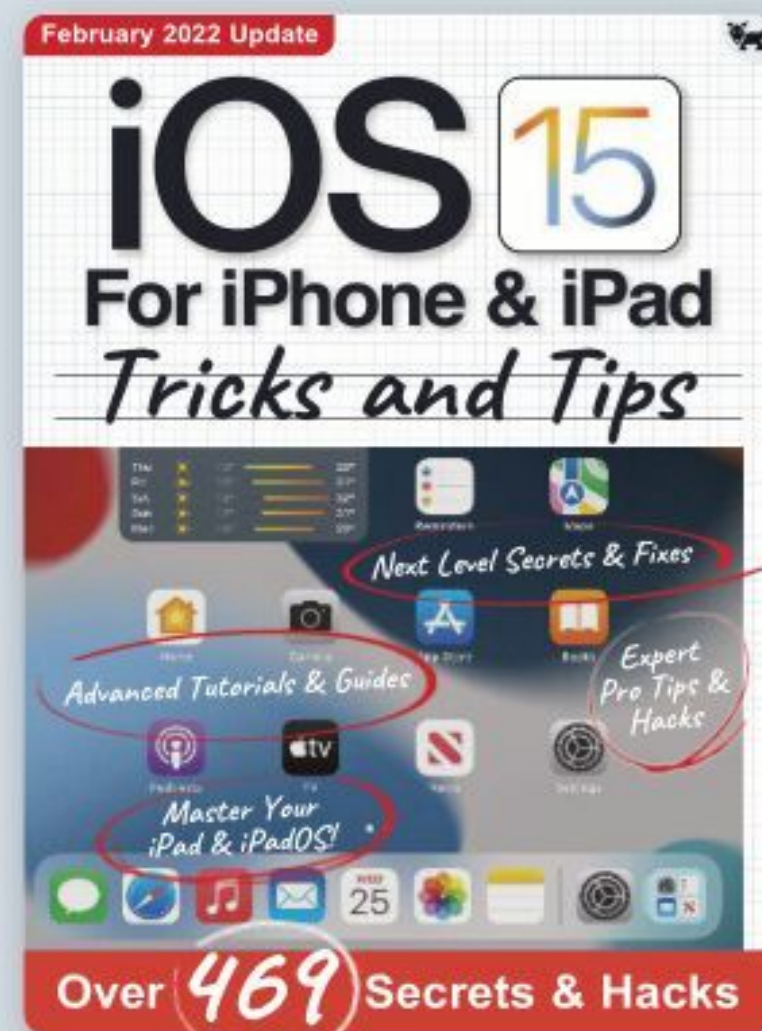
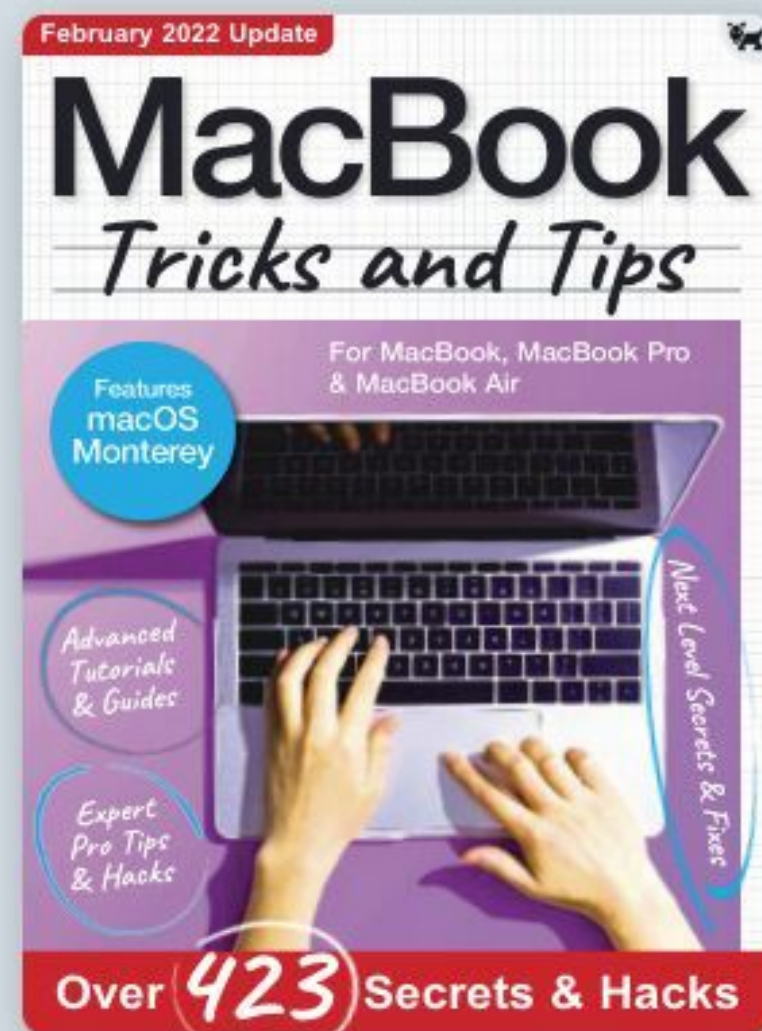
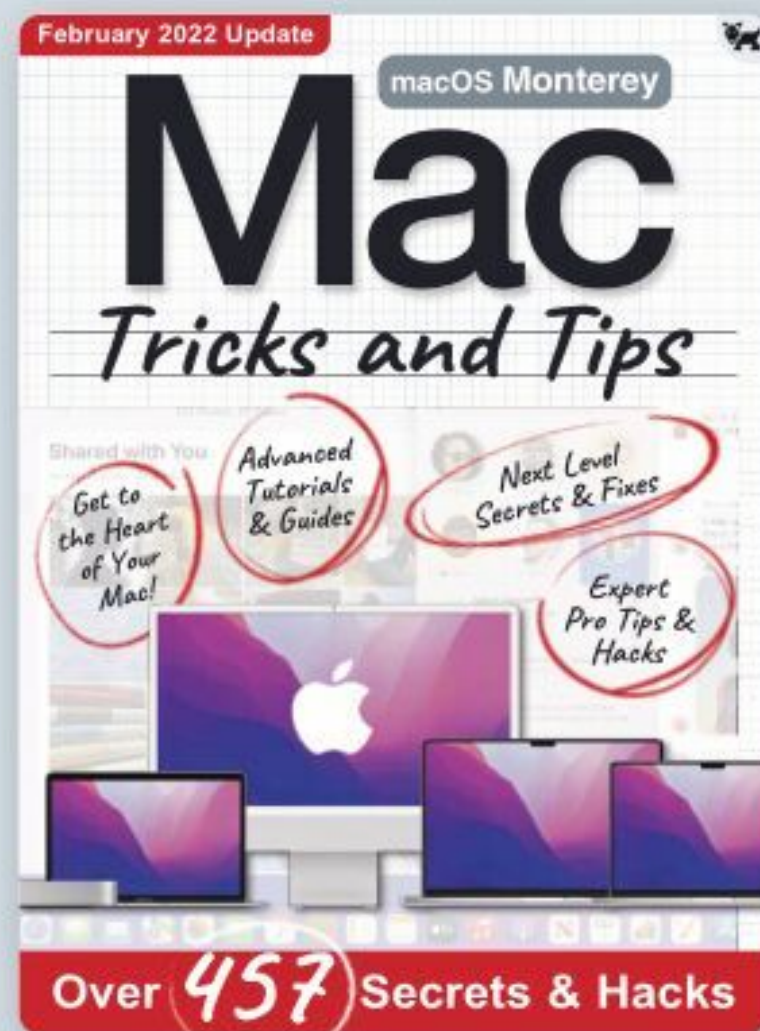
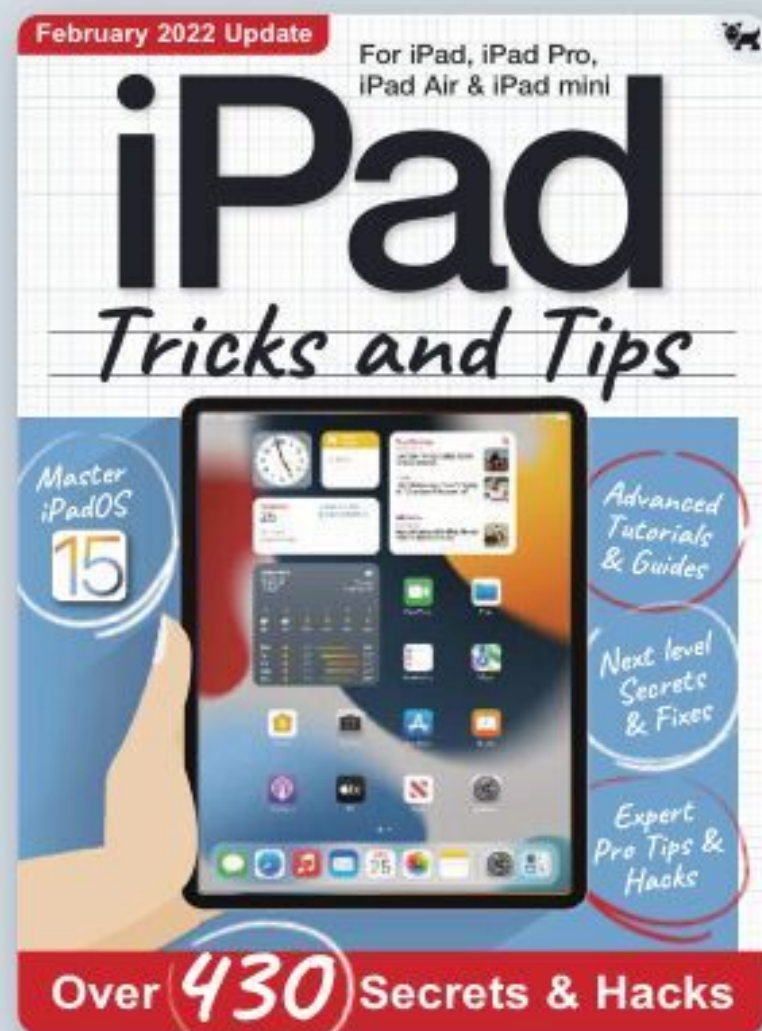
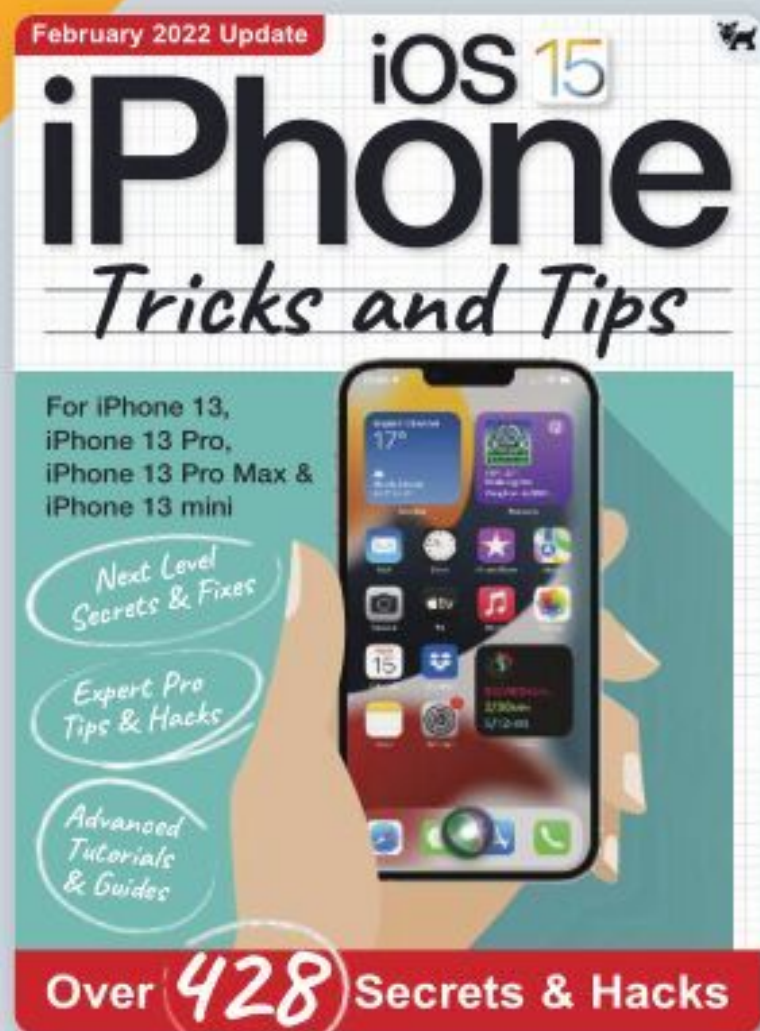
rmax = 5
rmax = 100

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

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

Read
More

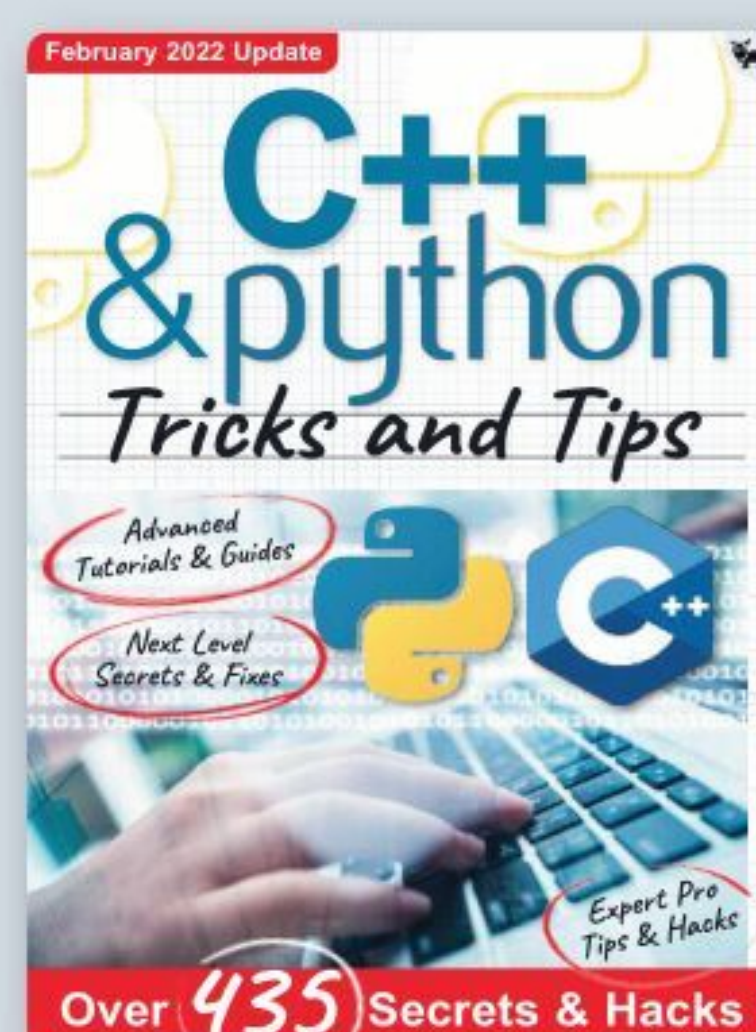
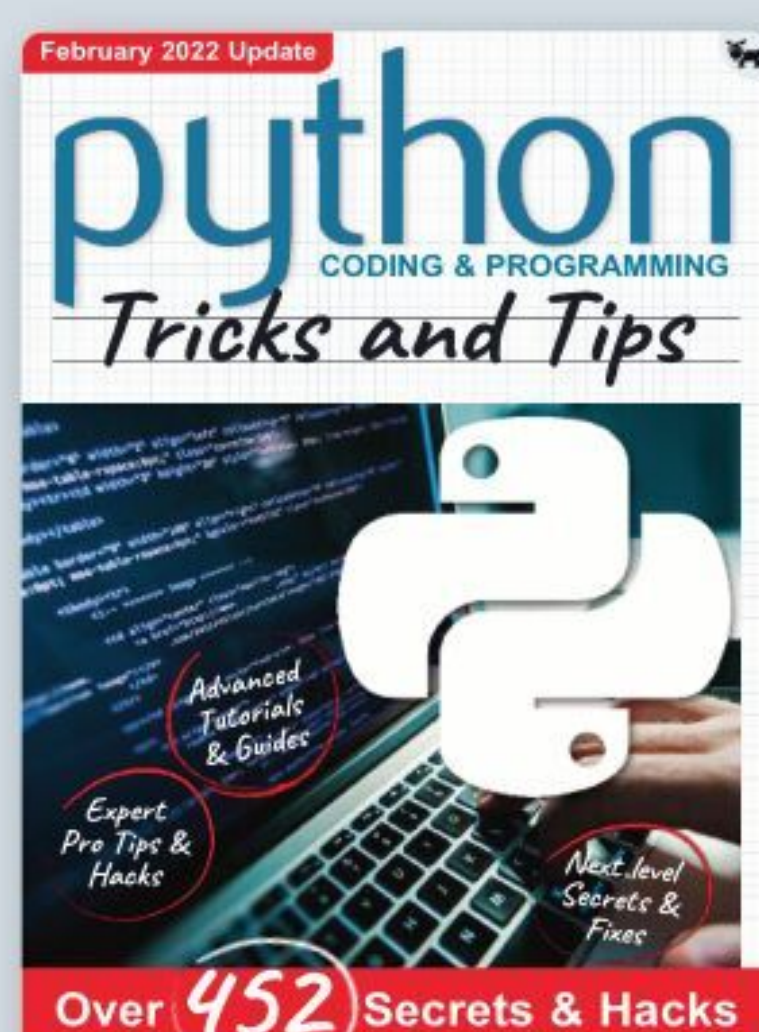
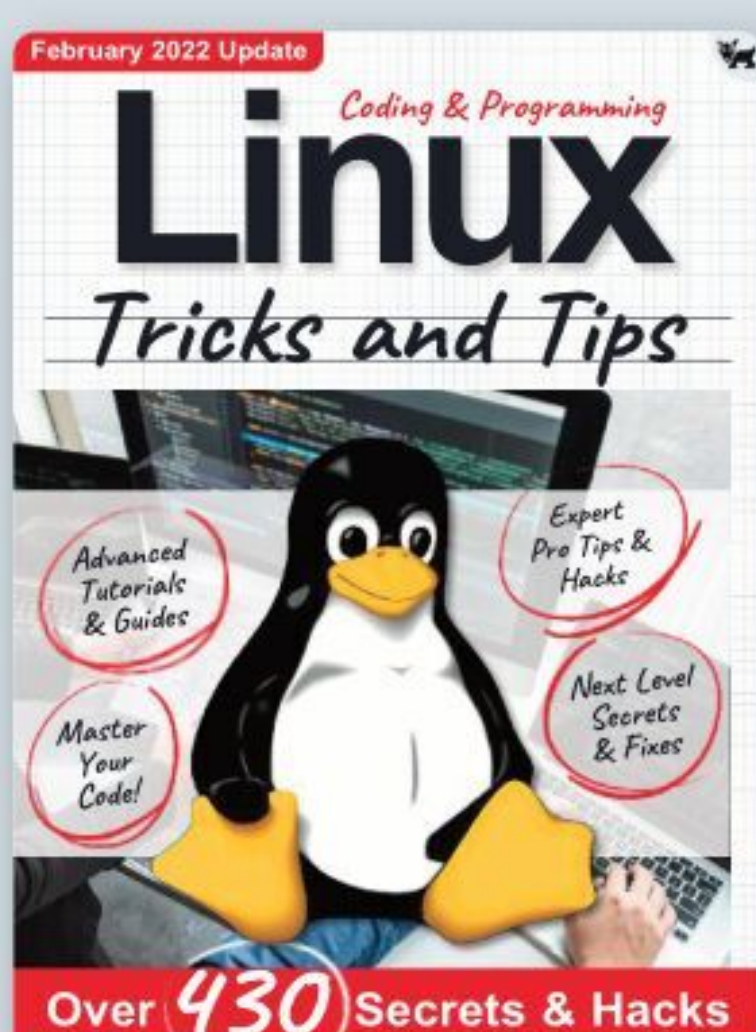
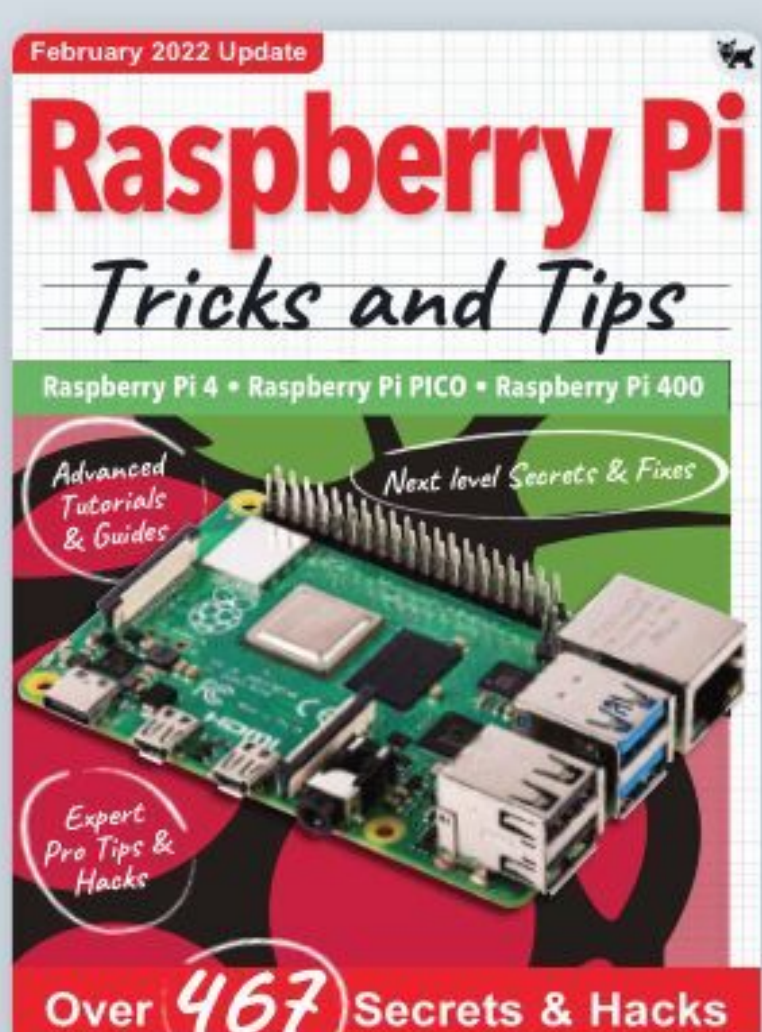
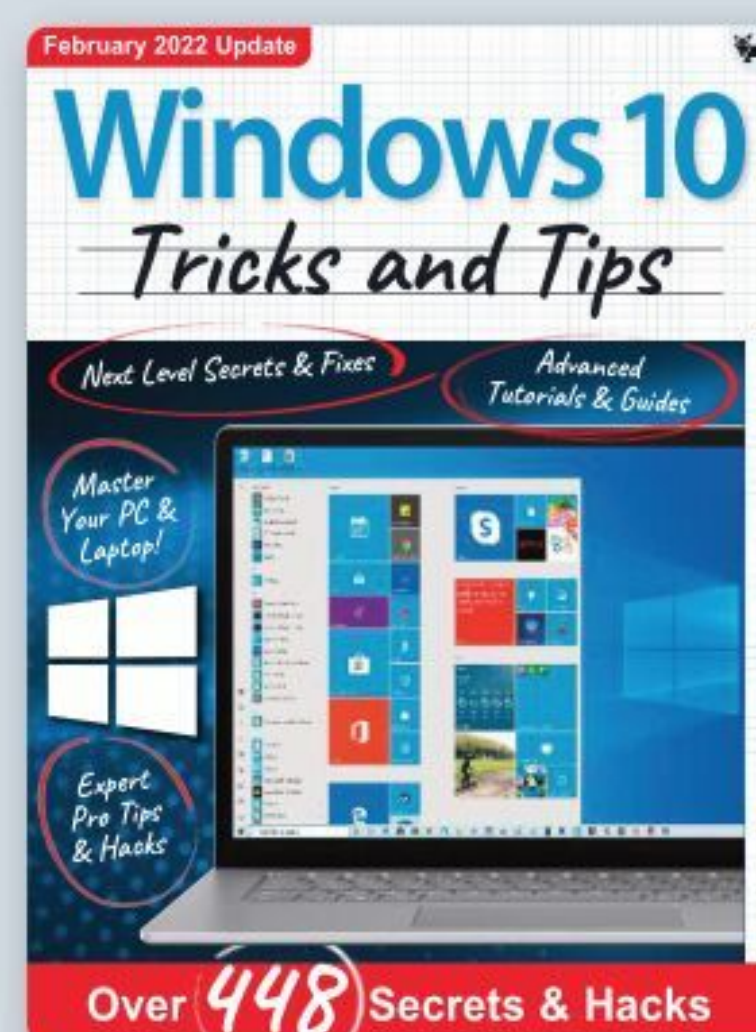
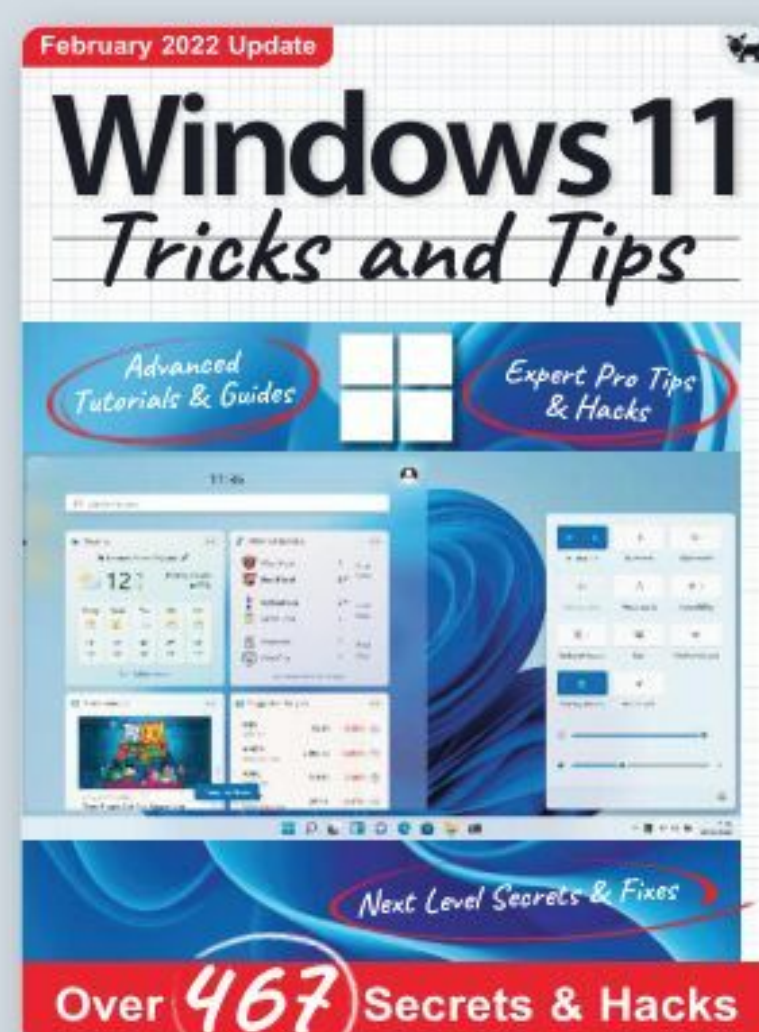
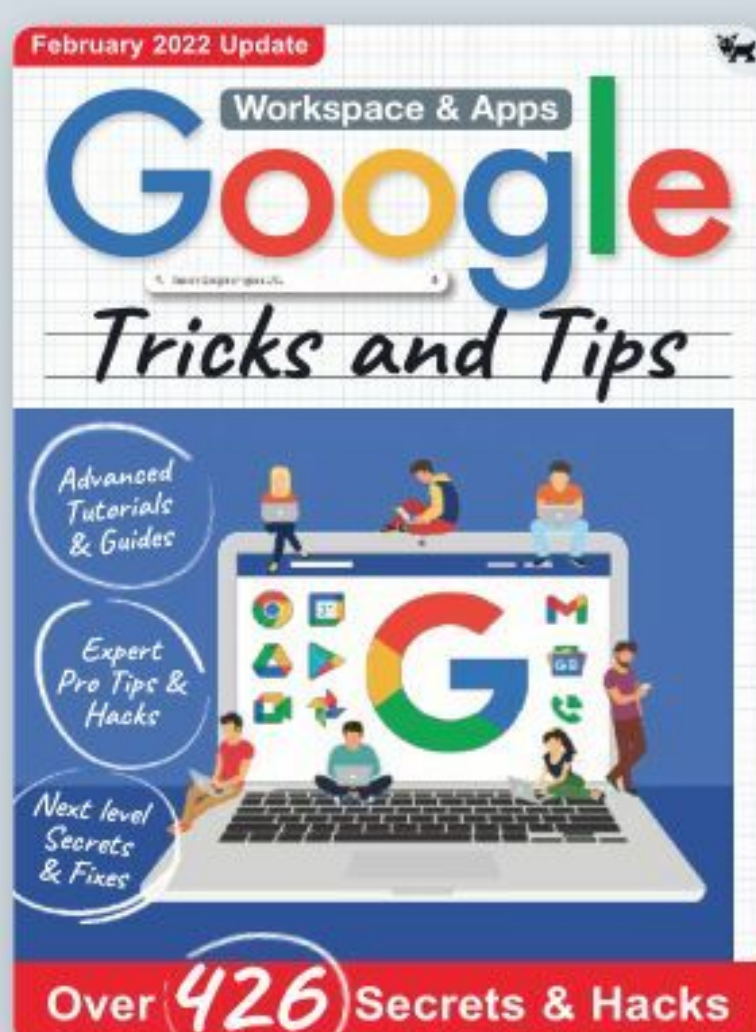
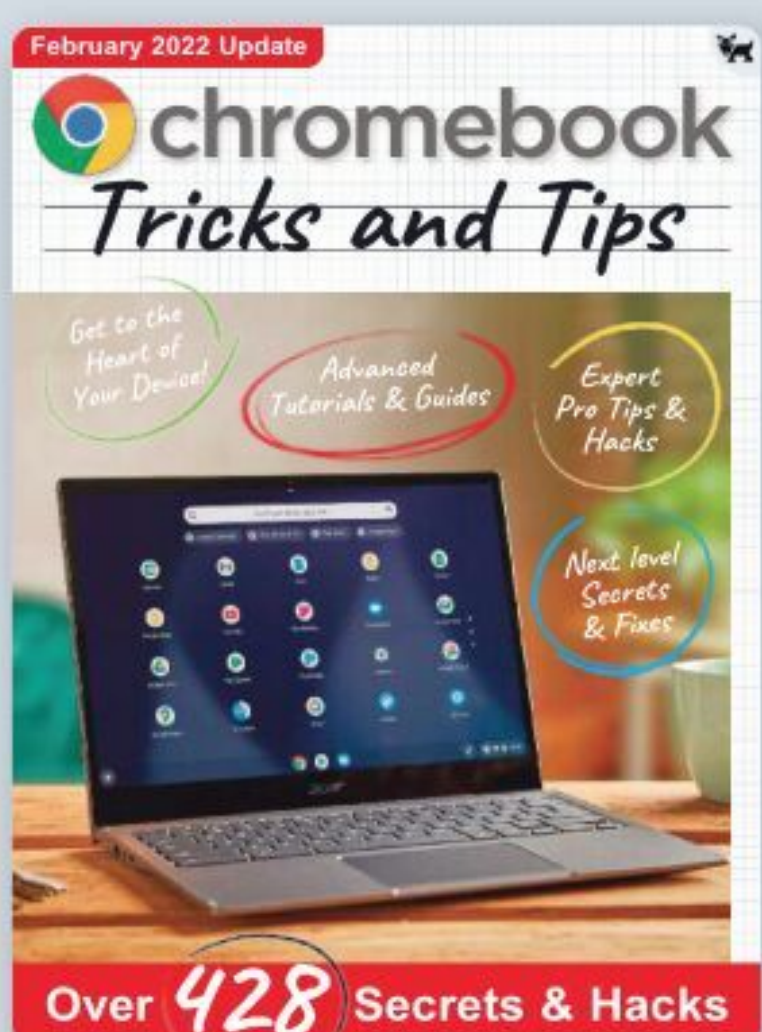
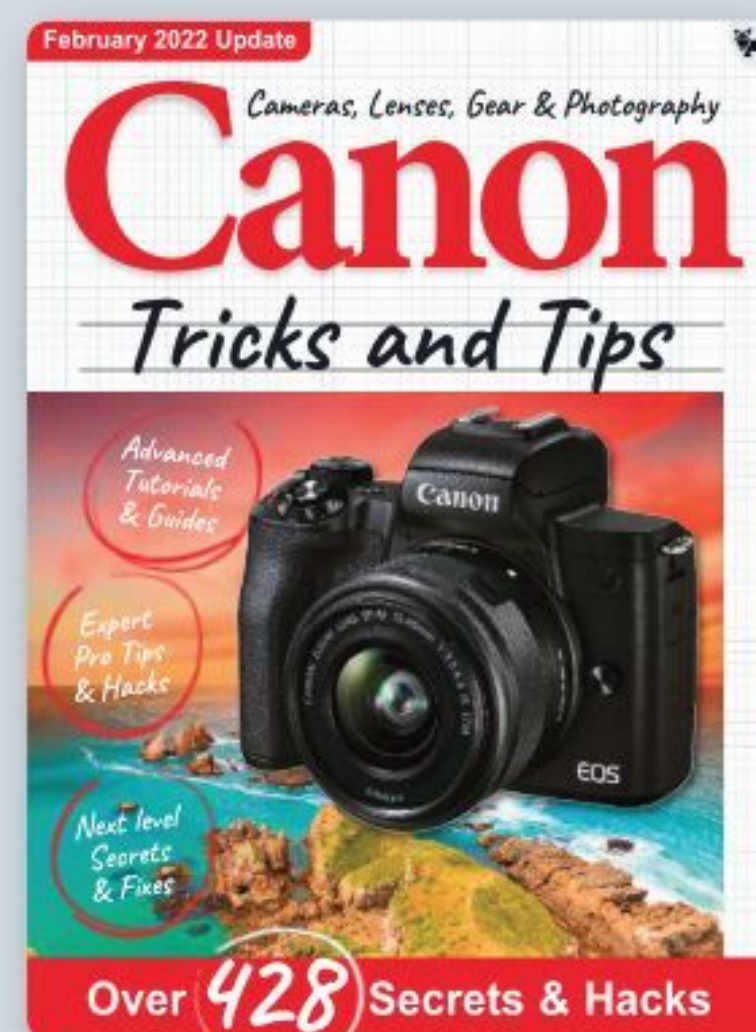
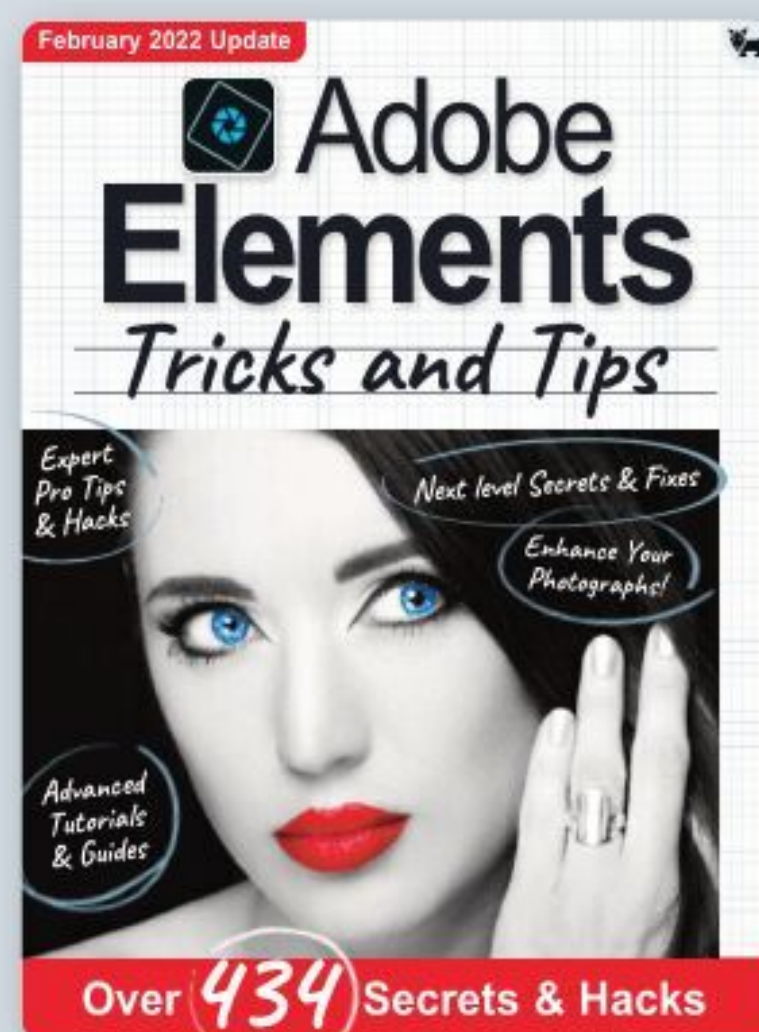
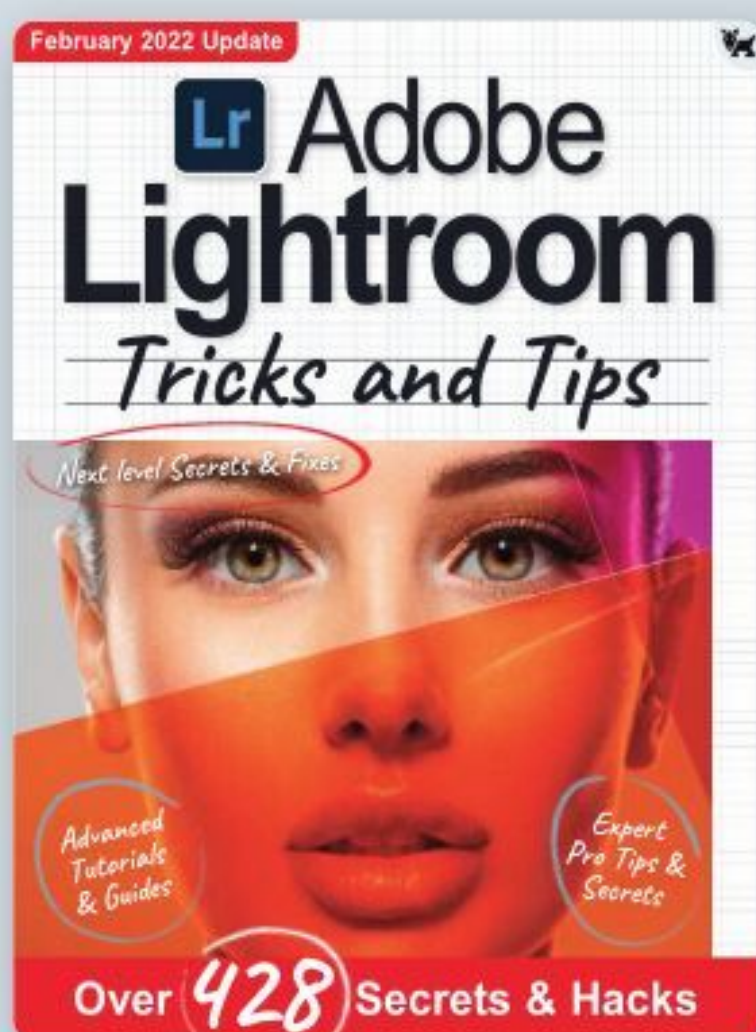
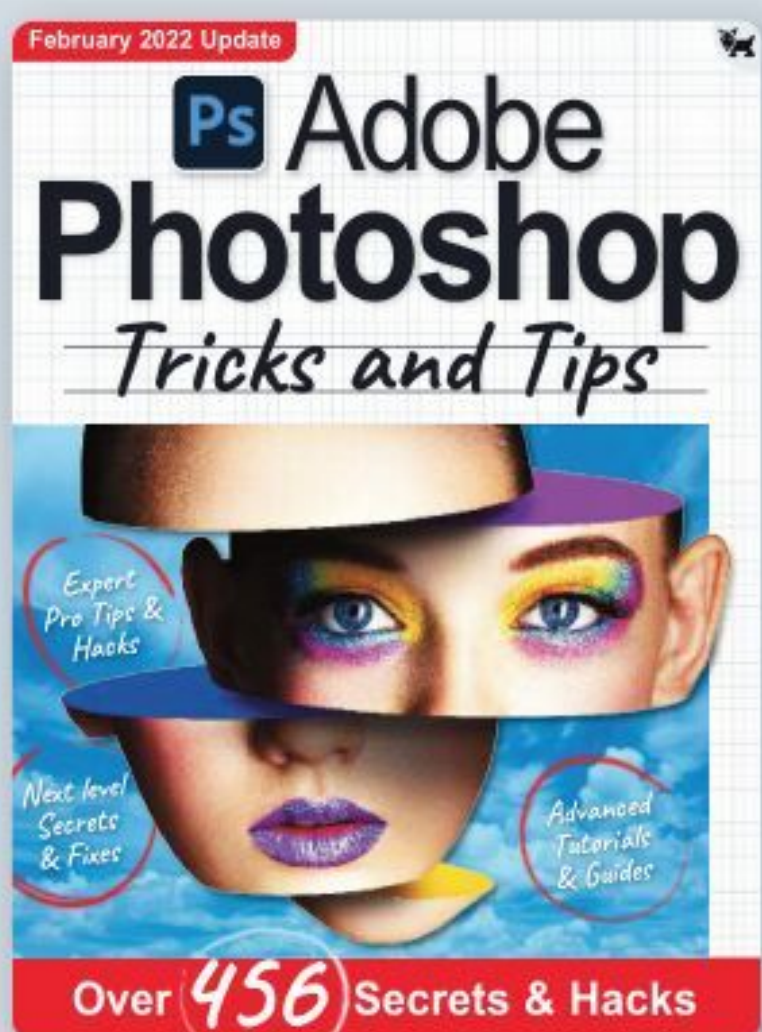
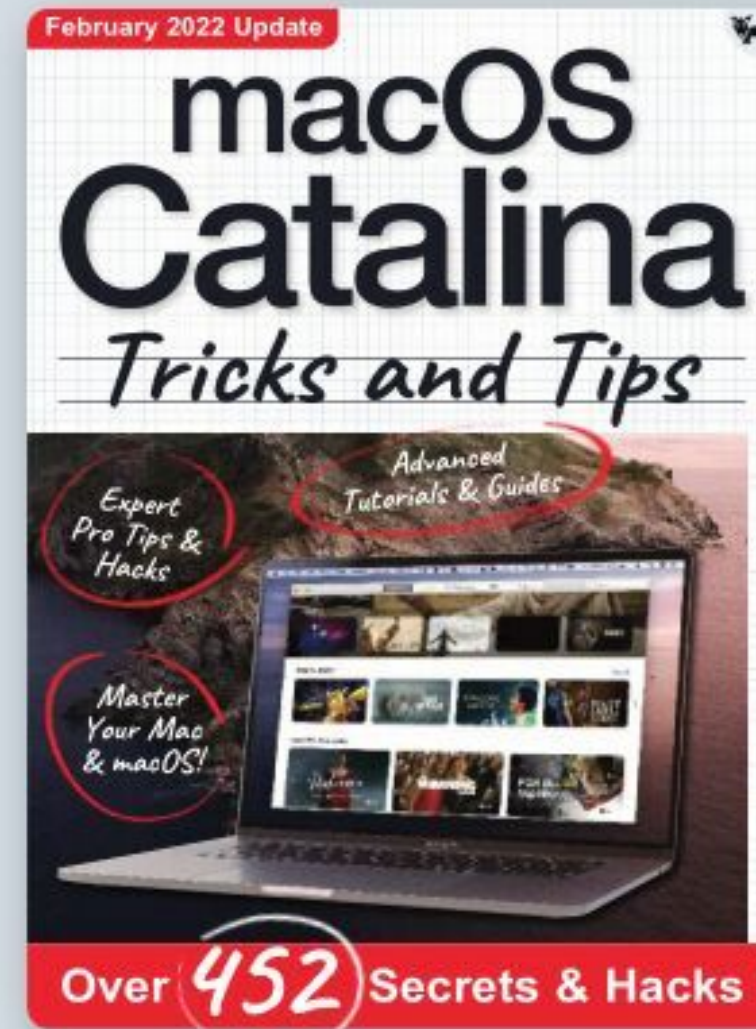
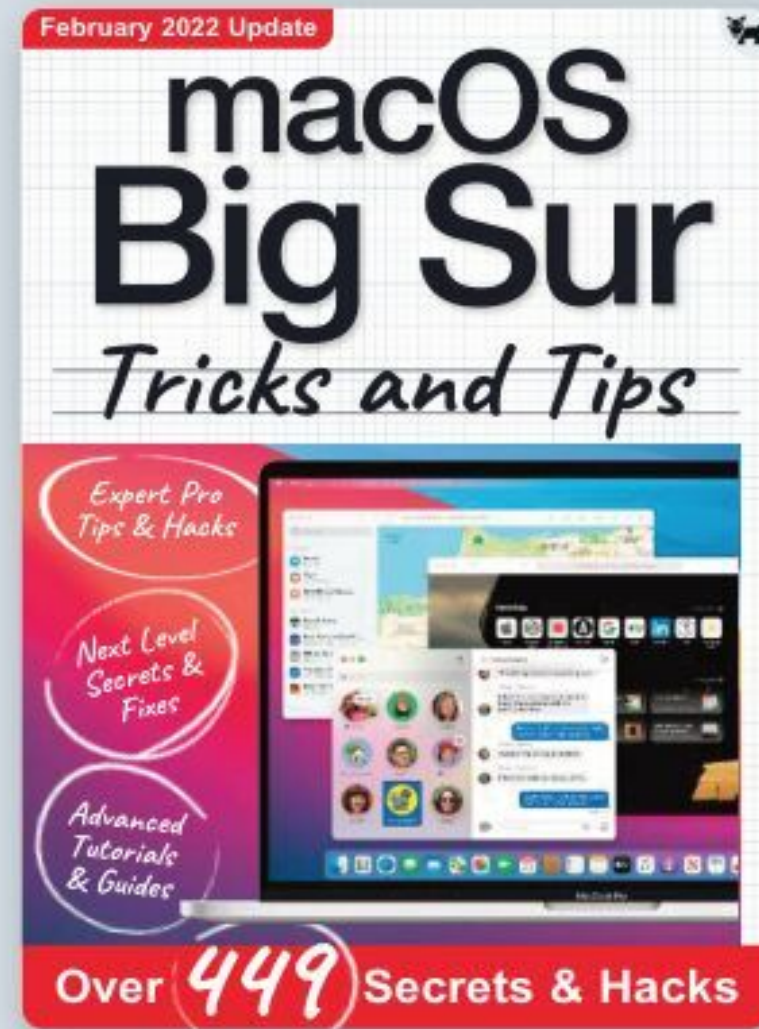
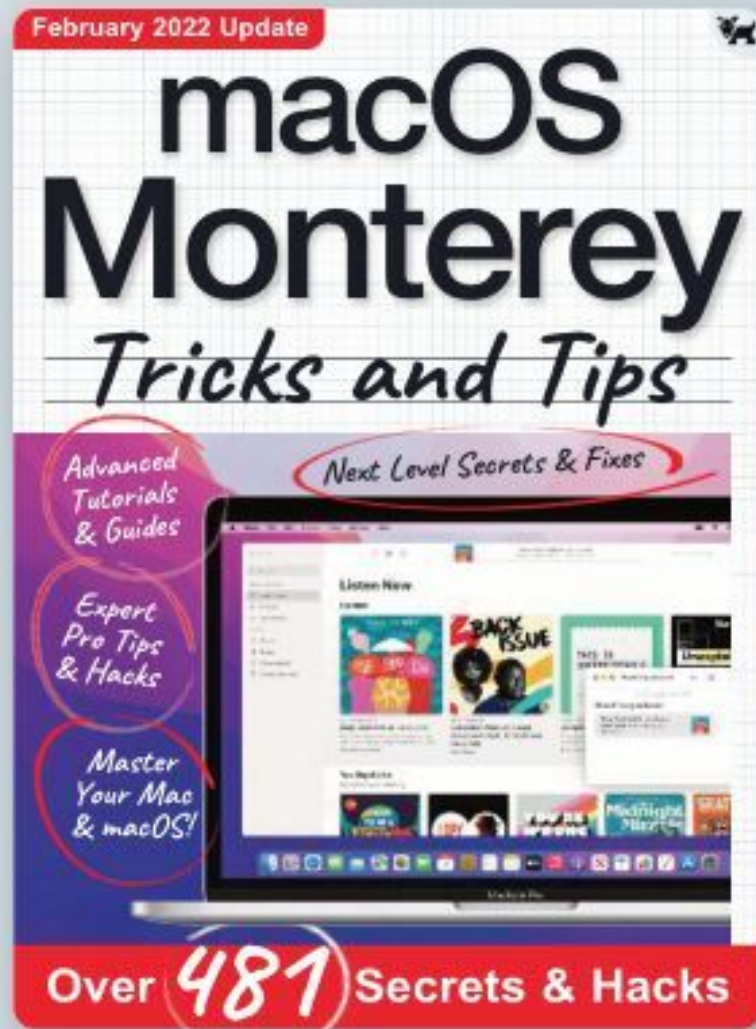
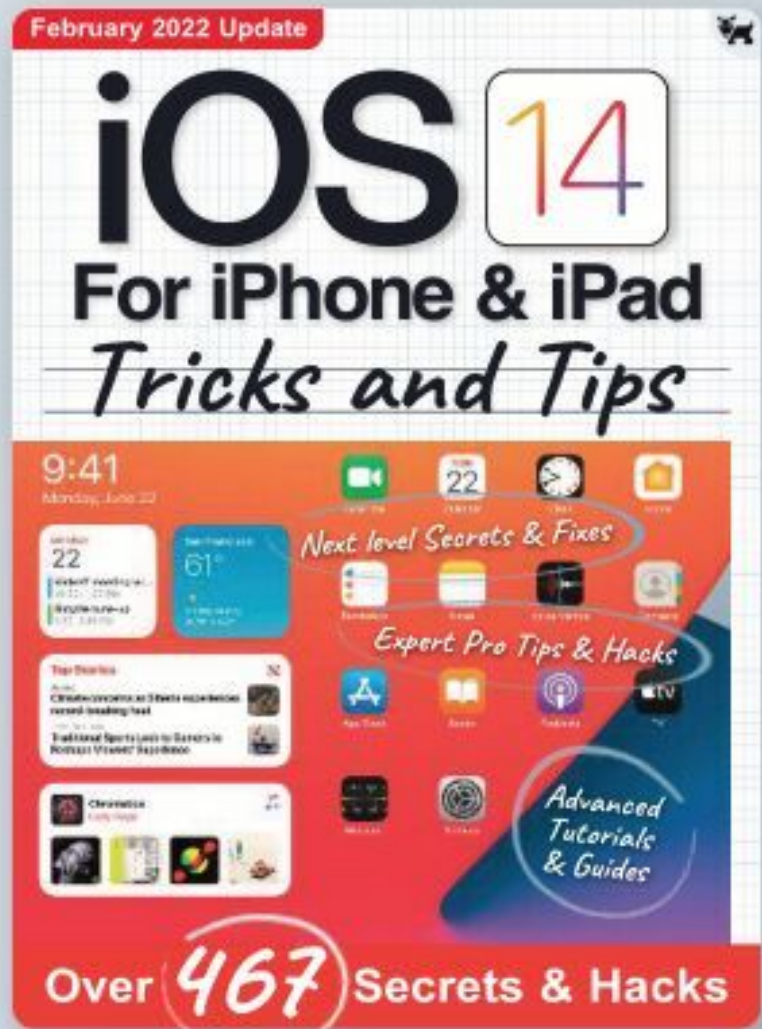
Tricks and Tips



Tech Guides
Available on



Readly



Congratulations, we have reached the end of your latest tech adventure. With help from our team of tech experts, you have been able to answer all your questions, grow in confidence and ultimately master any issues you had. You can now proudly proclaim that you are getting the absolute best from your latest choice from the ever changing world of consumer technology and software.

*So what's next?
Do you want to start a new hobby? Are you looking to upgrade to a new device? Or simply looking to learn a new skill?*

Whatever your plans we are here to help you. Just check our expansive range of **Tricks and Tips & For Beginners** guidebooks and we are positive you will find what you are looking for. This adventure with us may have ended, but that's not to say that your journey is over. Every new hardware or software update brings its new features and challenges, and of course you already know we are here to help. So we will look forward to seeing you again soon.



www.bdmpublications.com



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



Apple iPhone, iPad, Mac, MacBook & Watch



PC, Windows 11 & 10



Samsung & Google



Photography, Photoshop, Lightroom & Elements



Coding Python, C++, Raspberry Pi & Linux

EXCLUSIVE OFFERS & DISCOUNTS on Our Tech Guidebooks

- Print & Digital Editions
- Featuring the Latest Updates
- Step-by-step Tutorials & Guides
- Created by BDM Experts



BDM's **Ultimate Photoshop**

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

When you sign up to our monthly newsletter!

Linux Tricks and Tips

9th Edition | ISBN: 978-1-912847-76-1

Published by: Papercut Limited

Digital distribution by: Readly, Pocketmags & Zinio

© 2022 Papercut Limited. All rights reserved. No part of this publication may be reproduced in any form, stored in a retrieval system or integrated into any other publication, database or commercial programs without the express written permission of the publisher. Under no circumstances should this publication and its contents be resold, loaned out or used in any form by way of trade without the publisher's written permission. While we pride ourselves on the quality of the information we provide, Papercut Limited reserves the right not to be held responsible for any mistakes or inaccuracies found within the text of this publication. Due to

the nature of the tech industry, the publisher cannot guarantee that all apps and software will work on every version of device. It remains the purchaser's sole responsibility to determine the suitability of this book and its contents for whatever purpose. Any app, hardware or software images reproduced on the front cover are solely for design purposes and are not necessarily representative of content. We advise all potential buyers to check listings prior to purchase for confirmation of actual content. All editorial 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 or content. This is an independent publication and as such does not necessarily reflect the views or opinions of the producers of apps, software or products contained within. This publication is 100% unofficial and in no way associated with any other company, app developer, software developer or manufacturer. All copyrights, trademarks and registered trademarks for

the respective companies are acknowledged. Relevant graphic imagery reproduced with courtesy of brands, apps, software and product manufacturers. Additional images are reproduced under licence from Shutterstock. Prices, international availability, ratings, titles and content are subject to change. Some content may have been previously published in other editions. All information was correct at time of publication.

Papercut Limited
Registered in England & Wales No: 04308513

INTERNATIONAL LICENSING
Papercut Limited has many great consumer tech, coding, photography and Photoshop image editing publications and all are available for licensing worldwide.
For more information email: james@bdmpublications.com