Learn Coding: C++ & Python

Crystal clear guides to help you discover coding

FREE DOWNLOAD
60 Programs & 21,500 lines of code

For Windows, macOS and Linux users

• Packed with top tips and tutorials
• Learn Python 3 and apply it to real-world programs
• Start to learn the basics of C++
Discover more of our complete manuals on Readly today...
Learn Coding: C++ & Python

“Most good programmers do programming not because they expect to get paid or get adulation by the public, but because it is fun to program.”
— Linus Torvalds (developer of the Linux kernel)

We live in a world full of code: the Internet, our cars, phones, kitchen appliances, TV’s and more. Everywhere you go and nearly everything you will interact with on a daily basis has some form of code behind it, driving it and helping us humans manage the countless connected devices that make up our digital lifestyle.

Learning to code, therefore, offers us a chance to better understand how everything works and how it’s designed to work with us. Learn Coding: C++ & Python will help you build a solid foundation on which to expand your knowledge of the digital world. Within these pages, you will discover how to start coding, what coding can do for you and how you can apply code to help you or just to entertain you.

Learning code isn’t difficult, but it demands patience and imagination. So let’s begin your coding journey and see how far it will take you.
Master Python with the help of our fantastic Code Portal, featuring code for games, tools, and more.

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

Contents

100 C++ Fundamentals
102 Your First C++ Program
104 Structure of a C++ Program
106 Compile and Execute
108 Using Comments
110 Variables
112 Data Types
114 Strings
116 C++ Maths

118 C++ Input/Output
120 User interaction
122 Character Literals
124 Defining Constants
126 File Input/Output

128 Loops and Decision Making
130 While Loop
132 For Loop
134 Do...While Loop
136 If Statement
138 If...Else Statement

140 Keep::Coding
142 Passing Variables to Python
144 Retro Coding
146 Text Animations
148 Creating a Loading Screen
150 Tracking the ISS with Python
154 Using Text Files for Animation
156 Common Coding Mistakes
158 Python Beginner's Mistakes
160 C++ Beginner's Mistakes
“How did you know so much about computers?”
“aristral Grace Hopper (pionier programmer)
when interviewed by David Leterman
Why Python? For some years, Python has led the beginner’s coding community, thanks to its ease-of-use interface, effective and streamlined code, and the fact that with just a few lines you can get some great results fast.

Python is more than just another programming language, though. Its community is full of like-minded coders, from all walks of life and levels of ability, who strive to create great code and drive the future of Python, and coding, to ever-greater things.

In this section we look at what you can do with Python, what you will need to get started, and how you can get it on your computer.

8  Why Python?
10  What can You Do with Python?
12  Getting to Know Python
14  Equipment You Will Need
16  How to Set Up Python in Windows
18  How to Setup Python in Linux and macOS
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 multiplayer 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 these 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...
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—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, iOS, OS/2, VMS and Apple IIe 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.

Why Python?

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

What can You Do with Python?

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

Python’s ability to create highly readable code, within a small set of instructions, has a considerable impact on our modern digital world. From being an ideal first programmer’s choice through to being able to create interactive stories and games, from scientific applications through to artificial intelligence and web-based applications, the only limit to Python is the imagination of the person coding it.

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

**BIG DATA**

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

Python plays a substantial role in the world of Big data. It’s extensively used to analyse huge chunks of available big data and extract specific information based on what the user/company requires from the wealth of numbers. Python is also used in an impressive set of data processing libraries. Python makes the set of getting to the data, in amongst the numbers, that count and processing it in a fashion that’s readable and usable for humans.

There are countless libraries and freely available modules that enable fast, secure, and more importantly, accurate processing of data from the likes of supercomputing clusters. For example, CERN uses a custom Python module to help analyse the 460 million collisions per second that the Large Hadron Collider (LHC) produces. A different language handles the raw data, but Python is presently helping sift through the data so scientists can get to the content they want without the need to learn it for more complex programming languages.
ARTIFICIAL INTELLIGENCE

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

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

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

WEB DEVELOPMENT

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

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

GAMING

Although you won’t find too many triple-A rated games coded using Python, you may be surprised to learn that Python is used as an extremely large part of the top-ranking modern games.

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

A lot of the online video gaming is also Python in nature. You can find Python in Minecraft, providing the server-side elements. These include code to search for potential cheating, load balancing across the game’s servers, player skill matchmaking, and to check whether the player’s client-side games match the server’s version. There’s also a Python module that can be included in a Minecraft server, enabling the server admin to add blocks, send messages and alter a lot of the background complexities of the game.

PYTHON EVERYWHERE

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

Whatever route you decide to take on your coding journey, you will do well to have Python in your corner.
Python is the greatest computer programming language ever created. It enables you to fully harness the power of a computer, in a language that’s clean and easy to understand.

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

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

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

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

```python
bowl = []
four = 100
butter = 100
milk = 100
bowl.append([four, butter, milk])
```

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

Computer languages that are easy to read and are known as "high-level". This is because they fly high above the hardware (also referred to as "the metal"). Languages that fly close to the metal are Assembly, known as "source". Low-level languages commands read a bit like this: mg db .0x0a len eq 1 => mg

ZEON OF PYTHON

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

PYTHON 3 VS PYTHON 2

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

WORLD OF PYTHON

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

PYTHON 2.X

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

PYTHON 3.X

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

3.X WINS

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

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

WHAT WE’RE USING

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

□ COMPUTER

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

□ AN IDE

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

□ PYTHON SOFTWARE

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

□ TEXT EDITOR

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

□ INTERNET ACCESS

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

□ TIME AND PATIENCE

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

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

RASPBERRY PI

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

RASPBIAN

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

BOOKS

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

FUZE PROJECT

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

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

**INSTALLING PYTHON 3.X**

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

**Step 1**
Start by opening your web browser to www.python.org/downloads/. Look for the button detailing the download link for Python 3.8. The latest version at the time of writing is 3.7.0b2 as Python 3.8 is frequently updated: this may be a different version for you.

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

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

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

---

ISO's Developer Guide Series - Volume 11
How to Set Up Python in Windows

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

**STEP 6** Before you close the install wizard window, however, it's best to click on the link next to the shield detailed Disable Path Length Limit. This will allow Python to bypass the Windows 768 character limitation, enabling you to execute Python programs stored in deep folders arrangements. Again, click Yes to authenticate the process, then you can close the installation window.

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

**STEP 8** Clicking on the IDE (Python 3.7 32-bit) link will launch the Python Shell, where you can begin your Python programming journey. Don’t worry if your version is newer; as long as it’s Python 3.x our code works inside your Python 3.1 Interface.

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

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

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

**PYTHON PENGUIN**

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

**STEP 1**

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

```bash
$ python3 --version
Python 3.7.6
```

**STEP 2**

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

```bash
$ python3 --version
Python 3.7.6
```

**STEP 3**

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

```bash
$ sudo apt-get update
$ sudo apt-get upgrade
```

**STEP 4**

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

```bash
$ python3 --version
Python 3.7.6
```

**STEP 5**

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

```bash
$ sudo apt-get install build-essential checkinstall
$ sudo apt-get install libreadline-gplv2-dev
$ sudo apt-get install libssl-dev libglib2.0-dev tk-dev libgnome-dev libtinfo-dev
```

Next, you'll need to download Python 3 from the official repository. This will give you a clean version of Python 3.7.6. If you want to make sure you have the latest version (3.8.2), do the following:

```bash
$ sudo apt-get install python3-dev
```
How to setup Python in Linux and macOS

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

**STEP 7** In the Terminal, go to the Downloads folder by entering cd Downloads. Then unzip the contents of the downloaded Python source code with: tar zxvf Python-3.7.7.tar.gz (replace the Y’s with the version numbers you’ve downloaded). Now enter the newly unzipped folder with: cd Python-3.7.7/

**STEP 8** Within the Python folder, enter: ./configure

```
# For i686 Linux:
./configure --enable-optimizations
# For x86_64 Linux:
./configure --enable-optimizations
```

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

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

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

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

**STEP 10** You’ll also need PIP (Python Installs Package), which is a tool to help you install more modules and exerts. Enter:

```
sudo apt-get install python3-pip
```

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

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

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

**PYTHON ON macOS**

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

www.bitpublications.com
Getting Started with Python

“The purpose of software engineering is to control complexity, not to create it.”

— Pamela Zave (Developer, scientist and telecommunications expert)
Getting started with Python may seem a little daunting at first, but the language has been designed with simplicity in mind. Like most things, you need to start slow, learn how to get a result and how to get what you want from the code.

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

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

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

STARTING PYTHON

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

STEP 1

With the Raspbian desktop loaded, click on the Menu button followed by Programming > Python 3 (IDE). 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 2

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

STEP 3

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

STEP 4

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

**STEP 5** While that’s very interesting, it’s not particularly exciting. Instead, try this:

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

Just enter it into the IDLE as you’ve done in the previous steps.

**STEP 6** This is a bit 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!’ list.

```python
>>> 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

**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 powerful 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.dcs.python.org/3/library/idle.html, which details many of the IDLE’s features.
Your First Code

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

PLAYING WITH PYTHON

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 IDE, reopen it in whichever operating system version you prefer. In the Shell, enter the familiar following:

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

**STEP 2**

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

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

**STEP 3**

You can see that instead of the number 4, the output is the 2+2 you asked to be printed to the screen. The quotation marks are defining what’s being output to the IDE/Shell. To print the total of 2+2 you need to remove the quotes:

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

**STEP 4**

You can continue as such, printing 2+2, 460+2345, 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:

```
"2+2"
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```

```
2+2
```
STEP 5 What you have done here is assign the letters a and b two values: 2 and 3. These are now variables, which can be called upon by Python to output, add, subtract, divide and so on, as long as their numbers stay the same. Try this:
```
print(a)
print(b)
```

STEP 6 The output of the last step displays the current values of both a and b individually, as you’ve asked them to be printed separately. If you want to add them up, you can use the following:
```
print(a+b)
```
This code simply takes the values of a and b, adds them together and outputs the result.

STEP 7 You can play around with different kinds of variables and the Print function. For example, you could assign variables for someone’s name:
```
name="David"
print(name)
```

STEP 8 Now let’s add a surname:
```
surname="Hayward"
print(surname)
```
You now have two variables containing a first name and a surname and you can print them independently.

STEP 9 If we were to apply the same routine as before, using the + symbol, the name wouldn’t appear correctly in the output in the Shell. Try it:
```
print(name+surname)
```
You need a space between the two, defining them as two separate values and not something you mathematically play around with.

STEP 10 In Python 3 you can separate the two variables with a space using a comma.
```
print(name, surname)
```
Alternatively, you can add the space 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.
Getting Started with Python

Saving and Executing Your Code

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

EDITING CODE

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

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

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

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

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

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

**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!" will be printed.

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

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

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

```python
a=2
b=3
c=1
d=2
name="David"
surname="Rowland"
print([name, surname])
print(c+b)
```  
If you press F5 now you'll be asked to save the file, again, as it's been modified from before.

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

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

```python
c="Python"
b=""t"
c=eval(c)  
print(a, b, c)
```  
You will use this code in the next tutorial.
Executing Code from the Command Line

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

COMMAN D THE CODE

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

**STEP 1**

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

**STEP 2**

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

**STEP 3**

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

**STEP 4**

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

```python
print("Hello, World!")
```

You can see that it works exactly the same.
Executing Code from the Command Line

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

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

```python
python3 hello.py
```

This will execute the code we created, which is a reminder to:

```python
# Python
print("Hello, world!")
```

**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` by entering:

```bash
python hello.py
```

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

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

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

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

```bash
python3 hello.py
```

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

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

IT'S ALL MATHS, MAN

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

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

**STEP 2**
In the shell, enter the following:

```
2+2
54354+3553245
986/344+2734445842221
```

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

**STEP 3**
You can see 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+23)/(3+4)
```

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

```
1/2
```

And so on.
**Numbers and Expressions**

**STEP 5**
You can also use an operation to see the remainder left over from division. For example:

10/3

Will display 3.33333333, 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.

**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

18*13

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:

```python
2*3
18*13
```

**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:

```python
bin(3)
```

**STEP 8**
This will be displayed as '0b11', converting the integer into binary and adding the prefix '0b' to the front. If you want to remove the '0b' prefix, then you can use:

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

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

**STEP 9**
A Boolean Expression is a logical statement that will either be true or false. We can use these to compare data and text to see if it’s equal to, less than, or greater than. Try this in a New File:

```python
a = 6
b = 7
print(a == b)
print(a < b)
print(a > b)
```

**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 practice.

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

```python
x = 10
print("The value of x is ", x)
```

Save the file and execute the code.

**STEP 2** Running the code will return the line: The value of x 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:

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

**STEP 3** Reveal 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 others, of what's going on without the user being aware.

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

```python
# Set the value of A to 10
A = 10
# When the player dies, decrease the value of A by 1
A = A - 1
# Print the current value of A
print("The value of A is", A)
```

```python
# Set the value of A to 10
A = 10
# When the player dies, decrease the value of A by 1
A = A - 1
# Print the current value of A
print("The value of A is", A)
```

```python
# Set the value of A to 10
A = 10
# When the player dies, decrease the value of A by 1
A = A - 1
# Print the current value of A
print("The value of A is", A)
```

```python
# Set the value of A to 10
A = 10
# When the player dies, decrease the value of A by 1
A = A - 1
# Print the current value of A
print("The value of A is", A)
```
**STEP 5**

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

**STEP 6**

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

```python
# 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:
print("You've just lost a life!")
print("(lives)
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:

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

**STEP 8**

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

```python
# 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:

```python
# print("The value of A is:", A)
```

**STEP 10**

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

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

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

**VARIOUS VARIABLES**

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

**STEP 1**

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

**STEP 2**

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

**STEP 3**

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

**STEP 4**

You can also combine variables within another variable. For example, to combine both name and title variables into a new variable we use: `character=name + ", " + title`

Then output the content of the new variable as: `print(character)`

Numbers are stored as different variables:

- age=44
- Type: (age)

Which, as we know, are integers.
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)
```

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.

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 "example.py", line 1, in <module>
    print(name + age)
TypeError: Can't convert 'int' object to str implicitly
```

```
>>> print(character + " is " + str(age) + " years old.")
David Haywood: Descended from Vikings 44 years old.
```

```
>>> print(character, "is", age, "years old.")
David Haywood: Descended from Vikings in 44 years old.
```

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

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

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

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:

```
shirts = 19.99
```

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

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.
Getting Started with Python

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, open the Python 3 IDLE Shell, and start a new file in the editor. Let's begin with something really simple, enter:

```python
print("Hello")
first_name = input("What is your first name? ")
print("Thank you")
surname = input("And what is your surname? ")
```

**STEP 2**
Now that we have the user's name stored in a couple of variables, we can call them up wherever we want:

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

**STEP 3**
Save and execute the code, and you should see:

```python
IDLE shell> print("Hello")
Hello
IDLE shell> first_name = input("What is your first name? ")
What is your first name? Fred
IDLE shell> print("Thank you")
Thank you
IDLE shell> surname = input("And what is your surname? ")
And what is your surname? Smith
```

**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:

```python
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:

```python
print(f"Hello, what's your name?")
nome = input()
```

**STEP 6** The code from the previous step is often regarded as being a little easier 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:

```python
print(f"Hello! Who goes there?")
nome = input()
```

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

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

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

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

```python
rate = float(input("Enter a rate and a distance:"))
```

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

```python
distance = float(input("Distance:"))
print(f"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 a 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 3**

You can pass values 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):

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

Prints Fs and save the code to execute it.

**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 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.does.python.org/library/functions.html. These are the predefined functions, but since users have created many more, they’re not the only ones available.
Creating Functions

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
Enter:

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

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

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

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

This will now accept the variable name, otherwise it prints Hello. David in the Shell, enter `hello("Bob")`, then `hello("Jane")`. Your function can now pass variables through it.

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

```
from hello import Hello
Hello("David")
```

Press F5 to save and execute the code.

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

```
def say_hi(name):
    return name
```

Result: "Bob"
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:

```python
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:

```python
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 confirmation if it applies to the rule:

```python
if word_length == 4:
    print("This 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 a line in Python means to do everything after the colon that’s indented. Next, move the cursor back to the beginning of the editor:

```python
word = input("Please enter a four-letter word: ")
word_length = len(word)
if word_length == 4:
    print("This is a four-letter word. Well done.")
else:
    print("This is not a four-letter word.")
```
**Conditions and Loops**

**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 run the program but this time enter a five-letter word. The Shell will display that it's not a four-letter word.

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

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

**LOOPS**

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

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

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

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

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

```
words = ['cat', 'dog', 'elephant']
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 += 1 part isn't needed here because the range function creates a list between the First and last numbers used.
Python Modules

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

MASTERING MODULES

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

STEP 1

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

STEP 2

Some modules, especially on the Raspberry Pi, are included by default, the Math module being a prime example. Sadly, other modules aren’t always available. A good example on non-48 platforms is the Pygame module, which contains many functions to help create games. Try import pygame.

STEP 3

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

```
pip install pygame
```

STEP 4

The pip installation requires an elevated status due to installing components at different locations. Windows users can search for CMD via the Start button and right-click the result then click Run as Administrator. Linux and Mac users can use the Sudo command, with sudo pip install package.
STEP 5: Close the command prompt or Terminal and relaunch the IDLE Shell. When you next enter:
```
import pygame, the module will be imported into the code without any problems. You'll find that most code downloaded or copied from the Internet will contain a module, mainstream of unique, these are usually the source of errors in execution due to them being missing.
```

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

STEP 7: This code, when saved and executed, will display ten random numbers from 1 to 31. You can play around with the code to display more or less, and from a greater or lesser range. For example:
```
import random
for i in range(10):
    print(random.randint(1, 31))
```

STEP 8: Multiple modules can be imported within your code. To extend our example, use:
```
import random
import math
```

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 import commands, such as:
```
from random import randint
```

STEP 10: This helps create a more streamlined approach to programming. You can also use import modules, 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
```

Of course, adding comments helps to tell others what's going on.
Working with Data
“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++)

Data is everything. With it, you can display, control, add, remove, create and manipulate Python to your every demand. Over these coming pages, we look at how you can create lists, tuples, dictionaries and multi-dimensional lists, and then how you can use them to forge exciting and useful programs.

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

<table>
<thead>
<tr>
<th>Page</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>46</td>
<td>Lists</td>
</tr>
<tr>
<td>48</td>
<td>Tuples</td>
</tr>
<tr>
<td>50</td>
<td>Dictionaries</td>
</tr>
<tr>
<td>52</td>
<td>Splitting and Joining Strings</td>
</tr>
<tr>
<td>54</td>
<td>Formatting Strings</td>
</tr>
<tr>
<td>56</td>
<td>Date and Time</td>
</tr>
<tr>
<td>58</td>
<td>Opening Files</td>
</tr>
<tr>
<td>60</td>
<td>Writing to Files</td>
</tr>
<tr>
<td>62</td>
<td>Exceptions</td>
</tr>
<tr>
<td>64</td>
<td>Python Graphics</td>
</tr>
</tbody>
</table>
Lists

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

WORKING WITH LISTS

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

**STEP 1**

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

```
numbers = [1, 4, 7, 21, 98, 156]
```

```
mythical_creatures = ['Unicorn', 'Rolrrog', 'Vampire', 'Dragon', 'Minotaur']
```

**STEP 2**

To call up the entire contents of the list:

```
numbers
```

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

```
numbers[2]
```

**STEP 3**

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

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

**STEP 4**

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

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

Will output the 4 and 7, being item numbers 1 and 2. Note that the returned values don't include the second index position (so you would numbers[1:3] to return 4, 7 and 21).
**STEP 5** You can update items within an existing list, remove items and even join lists together. For example, to join two lists you can use:

```python
my_list1 = ['a', 'b', 'c']
my_list2 = ['d', 'e', 'f']
my_list1.extend(my_list2)
```

Then view the combined list with:

```python
my_list1
```

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

```python
numbers.append(21)
```

Or for strings:

```python
my_list = ['a', 'b', 'c']
my_list.append('d')
```

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

```python
del numbers[3]
```

Alternatively, by item name:

```python
deity_creatures.remove('Nessie')
```

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

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

To remove it:

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

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

```python
list('David 54')
```

This can then be passed to a new list:

```python
name = list('David')
age = 54
user = name + age
```

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

```python
name = input('What's your name? ')  
age = int(input('How old are you?: '))
user = name + age
```

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

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

THE IMMUTABLE TUPLE

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

**STEP 1**

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

```python
months=('January', 'February', 'March', 'April', 'May', 'June');
```

**STEP 2**

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

```python
months[2] = 'July';  # TypeError: 'tuple' object does not support item assignment
```

**STEP 3**

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

```python
NPC=[('Corin', 100), ('Belli', 80), ('Valerio', 90)];
```

**STEP 4**

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

```python
NPC[0][1] = 100;  # alter the combat rating for Corin
```

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

```python
NPC[0][1]  # will display 100.
```
STEP 5
It's worth noting that when referencing multiple tuples within a list, the indexing is slightly different from the norm. You would expect the 3rd combat rating of the character Valeria to be $\text{NPC}[3][3]$, but it's not; it's actually:

$\text{NPC}[3][1]

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

$0, 1, 3
0, 2, 3
0, 3, 3
1, 0, 3
1, 1, 3$

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

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

$\text{NPC} = (\text{"Conan", 100})$

STEP 8
Now unpack the tuple into two corresponding variables:

$(\text{name}, \text{combat_rating}) = \text{NPC}$

You can now check the values by entering name and combat rating.

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

$\text{NPC}[3][-1]$

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

$\text{numbers} = (10.3, 23.45, 32.1, 189.3, 6.1, 56.7, 99)$

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

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

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

KEY PAIRS

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

**STEP 1**

Let’s say you want to create a phonebook in Python. You would create the dictionary name and enter the data in curly brackets, separating the key and value by a colon.
For example:

```python
phonebook = {"Emily": 1234, "David": 3456, "Nancie": 6789}
```

**STEP 2**

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

```python
phonebook2 = {"David": 9876545213}
```

**STEP 3**

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

**STEP 4**

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

```python
phonebook["Emily"]
phonebook["Nancie"]
```
**Dictionaries**

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

```
phonebook["David"] = "0987-654-321"
```

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

```
cellphonebook["David"]
```

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

```
phonebook=
```

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

```
nome=input("Enter name: ")
numero=int(input("Enter phone number: "))
```

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

```
phonebook["nome"] = numero
```

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

```
phonebook telefonbook["David"]
```

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

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

STRING THEORIES

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

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

```python
text = "Daniel, Hannah, Timo"
```

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

```python
rows = text.split(" ")
```

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

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

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

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

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

The returned value is ‘D’, ‘a’, ‘v’, ‘i’, ‘d’; whereas it may seem a little useless under ordinary circumstances, it could be handy for creating a spelling game for example:
**STEP 5** The opposite of the `split` function is `join`, where you will have separate items like a string and can join them all together to form a word or just a combination of items, depending on the program you’re writing. For instance:

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

This will display ‘abcdef’ in the Shell.

**STEP 6** You can therefore apply `join` to the separated name you made in Step 4, combining the letters again to form the name:

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

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

**STEP 7** A good example of using the `join` function is when you have a list of words you want to combine into a sentence:

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

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

**STEP 8** As with the `split` function, the separator doesn’t have to be a space, it can also be a comma, a full stop, a hyphen or whatever you like:

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

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

```
title = "Conan the cimmerian".title()
title
```

**STEP 10** You can also use logic operators on strings, with the `in` and `not in` function. These enable you to check if a string contains (or does not contain) a sequence of characters:

```
message = "Have a nice day"
"nice" in message
"bad" not in message
"day" not in message
"right" in message
```
Formating Strings

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

STRING FORMATTING

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

**STEP 1**

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

```python
name = "Conan"
print("The barbarian hero of the Hyborian Age is: ", format(name))
```

**STEP 2**

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

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

**STEP 3**

You can of course also include integers into the mix:

```python
number = 20000
print("{} of {} was a skilled mercenary, and thief too. He once stole {} gold from a merchant.", format(name, number, place))
```

**STEP 4**

There are many different ways to apply string formatting, some are quite simple, as we've shown you here; others can be significantly more complex. It all depends on what you want from your program. A good place to reference frequently regarding string formatting is the Python Docs webpage, found at: [www.docs.python.org/3.1/library/string.html](http://www.docs.python.org/3.1/library/string.html). Here, you will find tons of help.
STEP 5  Interestingly you can reference a list using the string formatting function. You need to place an asterisk in front of the list name:

```python
numbers = [1, 45, 567546, 342346345]
print("Some numbers: {0}, {1}, {2}, {3}".format(*numbers))
```

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

```python
name = input("What’s your name? ")
print("Hello {0}. ".format(name))
```

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

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

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

```python
name = input("What’s your name? ")
print("Hello {0}. ".format(name))
name = name[0]
print("The first letter of your name is a {0}".format(name))
```

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

```python
characters = ["Conor", "Beltit", "Valerio", 19, 27, 28]
print("{0} is {1} years old. Whereas {2} is {3} years old.".format(*characters))
```

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

```python
names = ["Conor", "Beltit", "Valerio"]
ages = [25, 21, 22]
Creating two lists. Now you can call each list, and individual items:
print("{0[0]} is {0[1]} years old. Whereas {0[2]} is {0[1]} years old.".format(names, ages))
```

www.biteofcode.com
**Date and Time**

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

**TIME LORDS**

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

#### STEP 1

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

```
import time
print(time.gmtime())
```

The output is displayed as such:

```
('2021', 1, 10, 2, 0, 0, 0, 0, -1)
```

This obviously depends on your current time as opposed to the time this book was written.

#### STEP 2

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

<table>
<thead>
<tr>
<th>Index</th>
<th>Field</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4-digit year</td>
<td>2016</td>
</tr>
<tr>
<td>1</td>
<td>Month</td>
<td>1 to 12</td>
</tr>
<tr>
<td>2</td>
<td>Day</td>
<td>1 to 31</td>
</tr>
<tr>
<td>3</td>
<td>Hour</td>
<td>0 to 23</td>
</tr>
<tr>
<td>4</td>
<td>Minute</td>
<td>0 to 59</td>
</tr>
<tr>
<td>5</td>
<td>Second</td>
<td>0 to 61</td>
</tr>
<tr>
<td>6</td>
<td>Day of week</td>
<td>0 to 6 (Sunday to Saturday)</td>
</tr>
<tr>
<td>7</td>
<td>Day of year</td>
<td>1 to 366</td>
</tr>
<tr>
<td>8</td>
<td>Daylight savings</td>
<td>-1, 0, 1, -1 means library determines DST</td>
</tr>
</tbody>
</table>

#### STEP 3

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

```
from time import localtime
print(localtime())
```

This obviously depends on your current time as opposed to the time this book was written.

#### STEP 4

If you can’t find the function you’re looking for, it’s possible you may have misspelled it. For example, the `sleep` function is located in the `time` module. It allows you to sleep for a given duration in seconds. Enter:

```
import time
print(time.sleep(3))
```

This will sleep for 3 seconds.

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

```
import time
print(time.strftime("%A"))
```

This will display the current day of the week.
STEP 5: This naturally means you can incorporate various functions into your own code, such as:
```
time.strftime("%Y")
time.strftime("%M")
time.strftime("%H")
time.strftime("%M")
```

STEP 6: Note the last two entries, with %M and %H%M, as you can see the time is in hours and minutes and as the last entry indicates, entering them as %H%M does not display the highlighted time correctly in the Shell. You can rectify this with:
```
time.strftime("%H%M")
```

STEP 7: This means you’re going to be able to display both the current time or the time when something occurred, such as user entering their name. Try this code in the Editor:
```
import time
name = input("Enter login name: ")
print("Welcome", name, ", you")
print ("User", name, ", logged in at", time.strftime("%H%M") )
```

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

STEP 8: You saw at the end of the previous section, in the code to calculate Pi to however many decimal places the user wanted, you can time a particular event in Python. Take the code from above and add it slightly by including:
```
start_time = time.time()
```
Then there’s:
```
end_time = time.time() - start_time
```

STEP 9: The output will look similar to the screenshot below. The time() function needs to be either side of the input statement, as that’s when the variable name is being created, depending on how long the user took to login. The length of time is then displayed on the last line of the code as the end_time variable.

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

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

OPEN, READ AND WRITE

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

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

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

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

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

```python
poem.read()
```

Note that a hyphen in the text represents a newline, as you used previously.

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

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

This time, the in entries are removed in favour of new lines and readable text.
Just as with lists, tuples, dictionaries and so on, you’re able to Index individual characters of the text. For example:

```python
poem.read(3)
```
Displays the first three characters, whilst again entering:

```python
poem.read(3)
```
Will display the next five. Entering (1) will display one character at a time.

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

```python
poem=open("/home/pi/Documents/Poem.txt")
poe.readline()
```
Will display the first line of the text with:

```python
poem.readline()
```
Displaying the next line of text once more.

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

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

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

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

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

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

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

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

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

```python
import time
poem-open("/home/pi/Documents/Poem.txt")
lines=poem.read()
for lines in lines:
  print(lines, end="."")
time.sleep(.15)
```
The output is fun to view, and easily incorporated into your own code.
Writing to Files

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

WRITE AND CLOSE

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

### Step 1

Start by opening OLE and enter the following:

```python
open("/home/pl/Documents/text.txt","w")
```

Change the destination from `home/pl/Documents` to your own system. This code will create a text file called `text.txt` in write mode using the variable `$` file. If there's no file of that name in the location, it will create one. If one already exists, it will overwrite it, so be careful.

### Step 2

This works opposite to `read()`, writing lines instead of reading them. Try this:

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

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

### Step 3

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

```python
close()
```

### Step 4

You can now write to the text file using the `write()` function. Try this:

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

If you now open the text file with a text editor, you can see that the line you created has been written to the file. This gives us the foundation for some interesting possibilities, perhaps the creation of your own log file or even the beginning of an adventure game.
STEP 5  To expand this code, you can reopen the file using 'a', for access or append mode. This will add any text at the end of the original line instead of wiping the file and creating a new one. For example:

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

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

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

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

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

import math
print("Value of PI is: ",math.pi)
print("Now writing to a file now.")

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

pi=math.pi
You also need to create a new file in which to write PI:

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

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

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

t.write("Value of PI is: {0:.5f}\n")
t.close()

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

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

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

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

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

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

**Step 4**
To trap any errors in the code, you can encase the potential error within a try block. This block consists of `try`, except, else, where the code is held within `try`, then if there’s an exception do something, else do something else.
STEP 5 For example, use the divide by zero error. You can create an exception where the code can handle the error without Python quitting due to the problem:

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

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

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

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

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

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

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

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

www.idlepitbull.com
Python Graphics

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

GOING GRAPHICAL

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

**STEP 1**

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

```python
import turtle
```

**STEP 2**

Let’s begin by drawing a simple circle. Start a new file, then enter the following code:

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

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

**STEP 3**

The command `turtle.circle(50)` is what draws the circle on the screen, with 50 being the size. You can play around with the sizes; if you like, going up to 100, 150 and beyond, you can draw an arc by entering `turtle.circle(50, 200)`, where the size is 50, but you’re telling Python to only draw 180° of the circle.

**STEP 4**

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

```python
import turtle
print("Drawing a square...")
for c in range(4):
turtle.forward(50)
turtle.left(90)
turtle.getscreen()._root.mainloop()
```

You can see that we’ve inserted a loop to draw the sides of the square.
**Python Graphics**

---

**STEP 5** You can add a new line to the square code to add some colour:
```python
turtle.color("Red")
```

Then you can even change the character to an actual turtle by entering:
```python
turtle.shape("turtle")
```

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

---

**STEP 6** You can see that the `turtle` module can draw out some pretty shapes and become a little more complex as you begin to master the way it works. Enter this example:
```python
from turtle import *
color("Red", "Yellow")
begin_fill()
left(170)
if abs(cos(3)) < 0.1:
    break
define()
done()
```

It's a different method, but very effective.

---

**STEP 8** Now let's get the code by importing the `pygame` module:
```python
import pygame
```

**STEP 9** In the previous step, you imported `pygame`, initiated the `pygame` engine and asked it to import our saved Raspberry Pi logo image, saved as `PI.png`. Next, you defined the background colour of the window to display the image and the window size as per the actual image dimensions. Finally, you have a loop to close the window.
```python
window = pygame.display.set_mode((m, h))
screen.fill((0, 0, 0))
screen.blit(image,(0,0))
pygame.display.flip()
```

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

“Good code is short, simple, and symmetrical; the challenge is figuring out how to get there.”

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

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

- 68 Calendar Module
- 70 Using the Math Module
- 72 OS Module
- 74 Random Module
- 76 Tkinter Module
- 78 Pygame Module
- 82 Basic Animation
- 84 Create Your Own Modules
Calendar Module

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

WORKING WITH DATES

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

**STEP 1**
Launch Python 3 and enter: `import calendar` to call up the module and its inherent functions. Once it’s loaded into memory, start by entering: `sep=calendar.TextCalendar(calender=SUNDAY)` `sep=september(2019, 9)`

```
"""Python 3.8.3 Shell"

>>> import calendar
"""

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

```
Python 3.8.3 Shell
>>> calendar.TextCalendar(calender=SUNDAY)
```

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

```
leap=calendar.leapdays(1980, 2015)
p=print(leap)
```

The result is 29, starting from 1904 onward.

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

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

![Calendar Module Example](image.png)
**STEP 5**  You can also create a program that will display all the days, weeks and months within a given year:

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

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

**STEP 6**  Interestingly we can also list the number of days in a month by using a simple for loop:

```python
import calendar
cal=calendar.TextCalendar(calendar.SUNDAY)
for i in cal.monthdays2calendar(2019, 6):
    print(i)
```

**STEP 7**  You can see that at the outset, the code produced some errors. This is due to the starting day of the week, Sunday in this case, plus overlapping days from the previous month. Meaning the counting of the days will start on Saturday 1st June 2019 and will total 30, as the output correctly displays.

**STEP 8**  You're also able to print the individual months, or days of the week:

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

**STEP 9**  The Calendar module also allows us to write the functions in HTML, so that you can display it on a website. Let's start by creating a new file:

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

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

**STEP 10**  Of course, you can modify that to display a given year as a web page calendar:

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

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

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

E = MC²

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

**STEP 1**
The Math Module is built into Python 3.x, so there’s no need to PIP install it. As with the other modules present, you can import the module’s functions by simply entering `import math` into the shell, or as part of your code in the Editor.

```python
Python 3.4.2 Shell
In [1]: math.sin(2)
```

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

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

This displays the sine of 2.

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

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

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

```python
import math as m
m.trunc(123.45) # Truncate removes the fraction
```
Using the Math Module

**STEP 5**

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

```
from math import floor as f
```

**STEP 6**

Importing all the functions of the Math Module can be done by entering:

```
from math import *
```

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

**STEP 7**

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

```
sin([1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

will return the value of 0.999999999. Whereas:

```
fabs([1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

returns the value of 1.0.

**STEP 8**

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

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

**STEP 9**

This level of accuracy is really quite impressive, but quite note for the most part, probably the two most used functions are exp and log, where is the mathematical constant equal to 2.71828 (where the circumference of a circle is divided by its diameter):

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

**STEP 10**

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

www.nzpublishing.com
OS Module

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

INTO THE SYSTEM

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

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

```python
import os
home = os.getcwdu()
print(home)
```

**STEP 2** The returned result from printing the variable `home` is the current user’s home folder on the system. In our example that’s `/home/jbk` — it will be different depending on the user name you log in as and the operating system you use. For example, Windows 10 will output: `C:\Program Files (x86)\Python38-32`.

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

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

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

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

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

```python
import os
os.system("chromium-browser "http://bhppublications.com/")
```

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

```python
import os
e="chromium-browser "http://bhppublications.com/"
b="chromium-browser "http://www.google.co.uk"
os.system(e + b)
```

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

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

This creates a new directory within the current working directory, named according to the object in the `mkdir` function.

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

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

To delete them:

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

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

```python
import os, shutil, time
root_src_dir = "~/home/pi/Documents"
root_dst_dir = "~/home/pi/backup/" + time.asctime()
for src_dir, dirs, files in os.walk(root_src_dir):
    dst_dir = src_dir.replace(root_src_dir, root_dst_dir)
    if not os.path.exists(dst_dir):
        os.makedirs(dst_dir)
    for dst_file in files:
        src_file = os.path.join(src_dir, dst_file)
        dst_file = os.path.join(dst_dir, dst_file)
        if os.path.exists(dst_file):
            os.rename(dst_file)
        shutil.copy(src_file, dst_file)
print("Backup complete")
```
Random Module

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

**Random Numbers**

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

**Step 1**

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

```python
import random
print(random.randint(1,5))
```

**Step 2**

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

**Step 3**

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

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

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

**Step 4**

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

```python
import random
random.choice(["Cap"], ["Valero"], ["Belit"])
```

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

**Step 5**

You can extend the previous example somewhat by having random.choice select a list of mixed variables. For instance:

```python
import random
random.choice([1st])
```

Print("me")
### Step 6
Interestingly, you can also use a function within the Random module to shuffle the items in the list, thus adding a little more randomness into the equation.

```python
random.shuffle(list)
print(list)
```

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

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

```python
import random
list=[1,2,3,4] for i in range(200)
random.shuffle(list)
print(list)
```

Keep shuffling the list, and you can have a different selection of items from 0 to 200 every time.

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

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

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

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

```python
import random
output={"Heads":0, "Tails":0}
coin=random.choice([0,1])
for i in range(10000):
    output[random.choice([0,1])]=1
print("Heads": output["Heads"])
print("Tails": output["Tails"])
```

### Step 10
Here’s an interesting piece of code. Using a text file containing 46,000 words, you can pick a user generated number of words from the file text file found at:

[www.github.com/dlw/english-words](http://www.github.com/dlw/english-words)

```python
import random
wds=list(input("How many words shall I choose?
with open("/home/pi/Downloads/words.txt", "rt") as f:
    words = f.readline()s
    words = [w.rstrip() for w in words]
    print("----------")
for w in random.sample(words, wds):
    print(w)
```

www.bigbtpublications.com
Tkinter Module

While running your code from the command line, or even in the Shell, is perfectly fine, Python is capable of so much more. The Tkinter module enables the programmer to set up a Graphical User Interface to interact with the user, and it’s surprisingly powerful too.

GETTING GUI

Tkinter is easy to use but there’s a lot more you can do with it. Let’s start by seeing how it works and getting some code into it. Before long you will discover just how powerful this module really is.

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

**STEP 2**
The ideal approach is to add `mainloop()` into the code to control the Tkinter event loop, but we’ll get to that soon. You’ve just created a Tkinter widget and there are several more we can play around with:
```
btn=Button()
btn.pack()
btn['text']='Hello everyone!
```
The first line focuses on the newly created window. Click back into the Shell and continue the other lines.

**STEP 3**
This creates a small, basic window. There’s not much else to do at this point but click the X in the corner to close the window.

**STEP 4**
It’s not recommended to import everything from a module using the `as` keyword, but it won’t do any harm normally. Let’s begin by creating a basic GUI window, enter:
```
window=Tk()
```

The ideal approach is to add `mainloop()` into the code to control the Tkinter event loop, but we’ll get to that soon. You’ve just created a Tkinter widget and there are several more we can play around with:
```
btn=Button()
btn.pack()
btn['text']='Hello everyone!
```
The first line focuses on the newly created window. Click back into the Shell and continue the other lines.

You can combine the above into a NewFile:
```
import tkinter as tk
from tkinter import *
```
```
btn=Button()
btn.pack()
btn['text']='Hello everyone!
```

Then add some button interactions:
```
def click():
    print("You just clicked me!")
    btn['command']=click
```
Then add some button interactions:
STEP 5  Save and execute the code from Step 5 and a window appears with ‘Hello everyone!’ inside. If you click the Hello everyone! button, the shell will output the text ‘You just clicked me!’ It’s simple but shows you what can be achieved with a few lines of code.

![Image of window with text and button]

STEP 6  You can also display both text and images within a Tkinter window. However, only GIF, PCX, or PPM formats are supported. So find an image and convert it before using the code. Here’s an example using the EDM publishing logo:

```python
from tkinter import *  
import PhotoImage

img = PhotoImage(file=’/home/pi/Downloads/EDM_Logo.gif’)

w1 = Label(root, image=img).pack(side=’right’)
content = ‘From its humble beginnings in 2004, the EDM brand quickly grew from a single publication produced by a team of just two to one of the biggest names in global bookazine publishing, for two simple reasons. Our passion and commitment to deliver the very best product every time. While the company has grown with a portfolio of over 250 publications delivered by our international staff, the foundation that it has been built upon remains the same, which is why we believe EDM isn’t just the first choice it’s the only choice for the smart consumer.’

w2 = Label(root, justify=LEFT, text=content).pack(side=’left’)
```

STEP 7  The previous code is quite lengthy, mostly due to the content variable holding a part of EDM’s About page from the company website. You can obviously change the content, the position, and the image to suit your needs.

STEP 8  You can create radio buttons too. Try:

```python
from tkinter import *

root = Tk()

var1 = IntVar()
var2 = IntVar()

Label(root, text=’Choose a preferred language: ’).pack()
Radiobutton(root, text=’Python’, variable=var1, value=1).pack()
Radiobutton(root, text=’C++’, variable=var2, value=2).pack()
mainloop()
```

STEP 9  You can also create check boxes, with buttons and output to the shell:

```python
from tkinter import *

root = Tk()

btn1 = IntVar()
btn2 = IntVar()

Label(root, text=’Choose a preferred language: ’).pack()
Checkbutton(root, text=’Python’, variable=btn1).grid(row=1, sticky=W)
Checkbutton(root, text=’C++’, variable=btn2).grid(row=2, sticky=W)
mainloop()
```

STEP 10  The code from Step 9 introduced some new geometry elements into Tkinter. Note the sticky=W, E and V arguments. These describe the locations of the check boxes and buttons. (North, East, South and West). The row argument places them on separate rows. Have a play around and see what you get.

![Image of check boxes and buttons]
Pygame Module

We've had a brief look at the Pygame module already but there's a lot more to it that needs exploring. Pygame was developed to help Python programmers create either graphical or text-based games.

PYGAMING

Pygame isn't an inherent module to Python but those using the Raspberry Pi will already have it installed. Everyone else will need to use: pip install pygame from the command prompt.

STEP 1

Naturally you need to load up the Pygame modules into memory before you're able to utilise them.

Once that's done Pygame requires the user to initialise it prior to any of the functions being used.

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

STEP 2

Let's create a simple game ready window, and give it a title.

```python
gamewindow=pygame.display.set_mode((800, 600))
gamewindow.display.set_caption("Adventure Game")
```

You can see that after the first line is entered, you need to click back into the IDLE Shell to continue entering code. Also, you can change the title of the window to anything you like.

STEP 3

Sadly you can't close the newly created Pygame window without closing the Python IDLE Shell, which isn't very practical. For this reason, you need to work in the editor (New + File) and create a True/False while loop.

```python
import pygame
from pygame.locals import *
pygame.init()
gamewindow=pygame.display.set_mode((800, 600))
pygame.display.set_caption("Adventure Game")
running=True
while running:
    for event in pygame.event.get():
        if event.type==QUIT:
            running=False
    pygame.quit()
```
STEP 4  If the Pygame window still won't close don't worry, it's just a discrepancy between the IDLE (which is written with Tkinter) and the Pygame module. If you run your code via the command line, it closes perfectly well.

STEP 5  You're going to shift the code around a bit now, running the main Pygame code within a while loop. It makes it neater and easier to follow. We've downloaded a graphic to use and we need to set some parameters for pygame:

```python
import pygame
pygame.init()
running=True

while running:
    gamewindow=pygame.display.set_mode((600, 600))
    pygame.display.set_caption("Adventure Game")
    black=(0,0,0)
    white=(255,255,255)
    gamewindow.fill(black)
    sprite(x,y)
    pygame.display.update()
```

STEP 6  Let's quickly go through the code changes. We've defined two colours, black and white together with their respective RGB colour values. Next we've loaded the downloaded image called sprite1.png and allocated it to the variable img and also defined a sprite function and the Bit function will allow us to eventually move the image.

```python
img=pygame.image.load("/home/pi/Downloads/sprite1.png")
def sprite(x,y):
    gamewindow.blit(img, (x,y))
```
Using Modules

STEP 7 Now we can change the code again, this time containing a movement option within the while loop, and adding the variables needed to move the sprite around the screen:

```python
import pygame
from pygame.locals import *
pygame.init()
running=True
gameworld=pygame.display.set_mode((800,600))
pygame.display.set_caption("Adventure Game")
block=(0,0,0)
white=(255,255,255)
img=pygame.image.load("/home/pi/Downloads/sprite1.png")
def sprite(x,y):
gameworld.blit(img,(x,y))
x=(800*.45)
y=(800*.8)
xchange=0
while running:
    for event in pygame.event.get():
        if event.type==QUIT:
            running=False
        if event.type==pygame.KEYDOWN:
            if event.key==pygame.K_LEFT:
                xchange=-5
            elif event.key==pygame.K_RIGHT:
                xchange=5
        if event.type==pygame.KEYUP:
            if event.key==pygame.K_LEFT or event.key==pygame.K_RIGHT:
                xchange=0
        x+=xchange
gameworld.fill(white)
sprite(x,y)
pause.screen.blit()()
pygame.quit()
```

STEP 8 Copy the code down and using the left and right arrow keys on the keyboard you can move your sprite across the bottom of the screen. Now, it looks like you have the makings of a classic arcade 2D scroller in the works.
**STEP 9**
You can now implement a few additions and utilise some previous tutorial code. The new elements are the `subprocess` module, of which one function allows us to launch a second Python script from within another, and we’re going to create a new file called `pygame_test.py`:

```python
import pygame
import subprocess
pygame.init()
screen = pygame.display.set_mode((800, 250))
clock = pygame.time.Clock()
font = pygame.font.Font(None, 25)
pygame.time.set_timer(pygame.USEREVENT, 200)
def text_generator(text):
    tmp = ''
    for letter in text:
        tmp += letter
        yield tmp

class DynamicText(object):
    def __init__(self, font, text, pos, autoreset=True):
        self.done = False
        self.font = font
        self.text = text
        self.text_gen = text_generator(self.text)
        self.pos = pos
        self.autoreset = autoreset
        self.update()

def reset(self):
    self.text_gen = text_generator(self.text)
    self.text = self.text_gen
    self.update()

def update(self):
    if not self.done:
        try:
            self.rendered = self.font.render_next(self.text_gen), True, (0, 128, 0)
        except StopIteration:
            self.done = True
            time.sleep(10)
    self.rendered = pygame.transform.rotate(self.rendered, 90)
    self.screen.blit(self.rendered[0], self.rendered[1])

def draw(self, screen):
    self.update()
    screen.blit(self.rendered, self.pos)

text = DynamicText(font, text, (65, 120), autoreset=True)

while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT: break
        if event.type == pygame.USEREVENT: message.update()
        else:
            screen.fill(pygame.color.Color('black'))
            message.draw(screen)
```

**STEP 10**
When you run this code it will display a long, narrow Pygame window with the intro text scrolling to the right. After a pause of ten seconds, it then launches the main game Python script where you can move the warrior sprite around. Overall the effect is quite good but there’s always room for improvement.
Basic Animation

Python’s modules make it relatively easy to create shapes, or display graphics and animate them accordingly. Animation though, can be a tricky element to get right in code. There are many different ways of achieving the same end result and we’ll show you one such example here.

**LIGHTS, CAMERA, ACTION**

The Tkinter module is an ideal starting point for learning animation within Python. Naturally, there are better custom modules out there, but Tkinter does the job well enough to get a grasp on what’s needed.

**STEP 1**
Let’s make a bouncing ball animation. First, we will need to create a canvas (window) and the ball to animate.

```python
from tkinter import *
import time

# Create window and canvas
root = Tk()
root.title("Custom Animation")
canvas = Canvas(root, width=400, height=300, bg='white')
canvas.pack()

# Create ball
ball1 = canvas.create_oval(5, 5, 60, 60, fill='red')

# Main function
def mainloop():
    pass

mainloop()
```

**STEP 2**
Save and Run the code. You will see a blank window appear, with a red ball sitting in the upper left corner of the window. While this is great, it’s not very animated. Let’s add the following code:

```python
# Initial conditions
a = 5
b = 5

# For loop to animate ball
for x in range(0, 100):
    canvas.move(ball1, a, b)
    root.update()
    time.sleep(.01)
```

**STEP 3**
Insert the code between the `ball1 = canvas.create_oval(5, 5, 60, 60, fill='red')` line and the `root.mainloop()` line. Save it and Run. You will now see the ball move from the top left corner of the animation window, down to the bottom right corner. You can alter the speed in which the ball traverses the window by altering the `time.sleep(.01)` line. Try it.

**STEP 4**
The `canvas.move(ball1, a, b)` line is the part that moves the ball from one corner to the other,obviously with both a and b equaling 5. We can change things around a bit, and with the previous code: `ball1 = canvas.create_oval(5, 5, 60, 60, fill='red')` and we can change the values of a and b to something else.

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

```
ball1 = canvas.create_oval(7, 7, 60, 60, fill='red')
```

```python
for x in range(0, 100):
    canvas.move(ball1, a, b)
    root.update()
    time.sleep(.01)
```
**STEP 5** Let's see if we can animate the ball so that it bounces around the window until you close the program.

```python
x0 = 5
y0 = 10
while True:
canvas.move(ball1, x0, y0)
    pos = canvas.coords(ball1)
    if pos[0] > width or pos[0] < 0:
        x0 = -x0
    if pos[1] > height or pos[1] < 0:
        y0 = -y0
    canvas.update()
time.sleep(0.025)
```

**STEP 6** Remove the code you entered in Step 5 in its place again. Between the `ball1 = canvas.create_oval(3,5,60,60, fill='red')` and the `gui.mainloop()` lines, save the code and run it as normal. If you've entered the code correctly, then you will see the red ball bounce off the edges of the window until you close the program.

**STEP 7** The bouncing animation takes place within the `while True:` loop. First, we have the values of `x0` and `y0` before the loop, both of 5 and 10. The `pos = canvas.coords(ball1)` line then takes the value of the ball's location in the window. When it reaches the limits of the window (`width` or `height`), it will make the value negative, moving the ball around the screen.

**STEP 8** Pygame, however, is a much better choice at producing higher-end animations. Begin by creating a new file and entering:

```
import pygame
from random import randint

MAX_STARS = 250
STAR_SPEED = 2

def init_stars(screen):
    # Create the starfield
    global stars
    stars = []
    for i in range(MAX_STARS):
        # A star is represented as a list with this format: [x, y]
        star = [randint(0, screen.get_width()) - 100, randint(0, screen.get_height()) - 100]
        stars.append(star)

def move_and_draw_stars(screen):
    # Move and draw the stars
    for star in stars:
        star[0] = star[0] + STAR_SPEED
        if star[0] > screen.get_width() or star[0] < 0:
            star[0] = randint(0, 600)
        if star[1] > screen.get_height() or star[1] < 0:
            star[1] = randint(0, 600)
        screen.set_at(star, (255, 255, 255))
```

**STEP 9** Now add the following:

```python
def main():
    pygame.init()
    screen = pygame.display.set_mode((480, 480))
    pygame.display.set_caption("Starfield Simulation")
    clock = pygame.time.Clock()

    init_stars(screen)
    while True:
        # Lock the framerate at 50 FPS
        clock.tick(50)

        # Handle events
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                return

        screen.fill((0, 0, 0))
        move_and_draw_stars(screen)
        pygame.display.flip()
```

**STEP 10** Save and Run the code. You will agree that the simulated starfield code looks quite impressive. Imagine this as the beginning of some game code, or even the start to a presentation? Using a combination of Pygame and Tinker, your Python animations will look fantastic.

www.indiastatpublications.com

83
Using Modules

Create Your Own Modules

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

BUILDING MODULES

Modules are Python files, containing code, that you save using a .py extension. These are then imported into Python using the now familiar import command.

**STEP 1**

Let’s start by creating a set of basic mathematics functions. Multiply a number by two, three and square or raise a number to an exponent (power). Create a new file in the C:\try and enter:

```python
def times_two(x):
    return x * 2

def times_three(x):
    return x * 3

def square(x):
    return x ** 2

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

**STEP 2**

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

```python
print(times_two(5))
print(times_three(4))
print(square(3))
print(power(2,3))
```

Save the program as basic_math.py and execute it to get the results.

**STEP 3**

Now you’re going to take the function definitions out of the program and into a separate file. Highlight the function definitions and choose Edit -> Cut. Choose File -> New file and use Edit -> Paste in the new window. You now have two separate files, one with the function definitions, the other with the function calls.

**STEP 4**

If you now try and execute the basic_math.py code again, the error ‘NameError: name ’times_two’ is not defined’ will be displayed. This is due to the code no longer having access to the function definitions.

**STEP 5**

Return to the newly created window containing the function definitions, and click File -> Save As. Name this minimath.py and save it in the same location as the original basic_math.py program. Now close the minimath.py window, so the basic_math.py window is left open.
**STEP 6**
Back to the basic_math.py window at the top of the code enter:

```
from minumath import *
```

This will import the function definitions as a module. Press F5 to save and execute the program to see it in action.

**STEP 7**
You can now use the code further to make the program a little more advanced, fully utilizing the newly created module. Include some user interaction by creating a basic menu the user can choose from:

```
print("Select operation: ", end=" ")
print("1. Times by two")
print("2. Times by three")
print("3. Square")
print("4. Power of")
```

```
choice = input("Enter choice (1/2/3/4): ")
```

**STEP 8**
Now we can add the user input to get the number the code will work on:

```
num1 = int(input("Enter number: "))
```

This will save the user-entered number as the variable num1.

**STEP 9**
Finally, you can now create a range of if statements to determine what to do with the number and utilise the newly created function definitions:

```
if choice == '1':
    times_two(num1)
elif choice == '2':
    times_three(num1)
elif choice == '3':
    square(num1)
elif choice == '4':
    num2 = int(input("Enter second number: "))
    power(num1, num2)
else:
    print("Invalid input")
```

**STEP 10**
Note that for the last available option, the Power of choice, we've added a second variable, num2. This passes a second number through the function-defintion called `power`. Save and execute the program to see it in action.
Introduction to C++

"Measuring programming progress by lines of code is like measuring aircraft building progress by weight."

– Bill Gates (Co-founder of Microsoft)
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++ as 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. In this section, we look at how to install a C++ IDE and compiler on your computer.

88  Why C++?
90  Equipment You Will Need
92  How to Set Up C++ in Windows
94  How to Set Up C++ on a Mac
96  How to Set Up C++ in Linux
98  Other C++ IDEs to Install
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.

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

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

void main()
{
    char ch;
    cout << "Enter a character to check it is vowel or not:"
    cin >> ch;
    switch (ch)
    {
    case 'a': case 'A':
    cout << ch << " is a Vowel";
    break;
    case 'e': case 'E':
    cout << ch << " is a Vowel";
    break;
    case 'i': case 'I':
    cout << ch << " is a Vowel";
    break;
    case 'o': case 'O':
    cout << ch << " is a Vowel";
    break;
    case 'u': case 'U':
    cout << ch << " is a Vowel";
    break;
    default:
    cout << ch << " is not a Vowel";
    break;
    }
}
```
Microsoft's Visual Studio is a great, free environment to learn C++ in.

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 Java, 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 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.
Equipment You Will Need

You don’t need to invest a huge amount of money in order to learn C++, and you don’t need an entire computing lab at your disposal either. Providing you have a fairly modern computer, everything else is freely available.

C++ SETUPS

As most, if not all, operating systems have C++ at their core, it stands to reason that you can learn to program in C++ no matter what OS you’re currently using.

☐ COMPUTER

Unless you fancy writing out your C++ code by hand on a sheet of paper (which is something many older coders used to do), then a computer is an absolute must have component. PC users can have any recent Linux distro or Windows OS. Mac users the latest macOS.

☐ AN IDE

As with Python, an IDE is used to enter and execute your C++ code. Many IDEs come with extensions and plugins that help make it work better, or add an extra level of functionality. Often, an IDE will provide enhancements depending on the own OS being used, such as being enhanced for Windows 10.

☐ COMPILER

A compiler is a program that will convert the C++ language into binary that the computer can understand. While some IDEs come with a compiler built-in, others don’t. CodeBlocks is our favourite IDE that comes with a C++ compiler as part of the package. More on this later.

☐ TEXT EDITOR

Some programmers much prefer to use a text editor to assemble their C++ code before running it through a compiler. Essentially you can any text editor to write code, just save it with a .cpp extension. However, Notepad++ is one of the best code text editors available.

☐ INTERNET ACCESS

While it’s entirely possible to learn how to code on a computer that’s not attached to the internet, it’s extraordinarily difficult. You will need to install the relevant software, keep it up to date, install any extras or extensions, and look for help when coding. All of which require access to the internet.

☐ TIME AND PATIENCE

Yes, as with Python, you’re going to need to set aside significant time to spend on learning how to code in C++. Sadly, unless you’re a genius, it’s not going to happen overnight, or even a week. A good C++ coder has spent many years honing their craft, so be patient, start small, and keep learning.
OS SPECIFIC NEEDS

C++ will work in any operating system, however, getting all the necessary pieces together can be confusing to a newcomer. Here’s some OS specific for C++:

**LINUX**

Linux users are lucky in that they already have a compiler and text editor built into their operating system. Any text editor will allow you type out your C++ code, when it’s saved with a .cpp extension, use g++ to compile it.

![Linux Code](image)

---

**WINDOWS**

As we’ve mentioned previously, a good IDE is Microsoft’s Visual Studio. However, a better IDE and compiler is Code::Blocks, which is regularly kept up to date with a new release twice a year; or so. Otherwise Windows users can enter their code in Notepad++ then compile it with MinGW – which Code::Blocks uses.

![Windows Code](image)

---

**RASPBERRY PI**

The Raspberry Pi’s operating system is Raspbian, which is Linux based. Therefore, you’re able to write your code out using a text editor, then compile it with g++ as you would in any other Linux distro.

![Raspberry Pi Code](image)

---

**MAC**

Mac owners will need to download and install Xcode to be able to compile their C++ code natively. Other options for the macOS include Netbeans, Eclipse or Code::Blocks. Note the latest Code::Blocks isn’t available for Mac due to a lack of Mac developers.

![Mac Code](image)
How to Set Up C++ in Windows

Windows users have a wealth of choice when it comes to programming in C++. There are loads of IDEs and compilers available, including Visual Studio from Microsoft. However, in our opinion, the best C++ IDE to begin with is Code::Blocks.

**CODE::BLOCKS**

Code::Blocks is a free C++, C and Fortran IDE that is feature rich and easily extendible with plugins. It’s easy to use, comes with a compiler and has a vibrant community behind it too.

**STEP 1**

Start by visiting the Code::Blocks download site, at [www.codeblocks.org/downloads](http://www.codeblocks.org/downloads). From there, click on the ‘Download the binary releases’ link to be taken to the latest downloadable version for Windows.

**STEP 2**

There you can see, there are several Windows versions available. The one you want to download has mingwsetup.exe at the end of the current version number. At the time of writing this is codeblocks-17.12mingw-setup.exe. The difference is that the mingwsetup version includes a C++ compiler and debugger from TDM-GCC (a compiler suite).

**STEP 3**

When you’ve located the file, click on the Sourceforge.net link at the end of the line and a download notification window appears; click on Save File to start the download and save the executable to your PC. Locate the downloaded Code::Blocks installer and double-click to start. Follow the on-screen instructions to begin the installation.

**STEP 4**

Once you agree to the licensing terms, a choice of installation options becomes available. You can opt for a smaller install, missing out on some of the components but we recommend that you opt for the Full option, as default.
**How to Set Up C++ in Windows**

**STEP 5**  Next choose an install location for the Code::Blocks files. It’s your choice but the default is generally sufficient (unless you have any special requirements of course). When you click Next, the install begins, when it’s finished a notification pops up asking if you want start Code::Blocks now, so click Yes.

**STEP 6**  The first time Code::Blocks loads it runs an autodetect for any C++ compilers you may already have installed on your system. If you don’t have any, click on the first detected option: GNU GCC Compiler and click the Default button to set it as the system’s C++ Compiler. Click OK when you’re ready to continue.

**STEP 7**  The program starts and another message appears informing you that Code::Blocks is currently not the default application for C++ files. You have two options, to leave everything as it is or allow Code::Blocks to associate all C++ file types. Again, we would recommend you opt for the last choice, to associate Code::Blocks with every supported file type.

**STEP 8**  There’s a lot you can do in Code::Blocks, so you need to dig in and find a good C++ tutorial to help you get the most from it. However, to begin with, click on FILE > NEW > Empty File. This creates a new, blank window for you to type in.

**STEP 9**  In the new window, enter the following:
```cpp
#include <iostream>
int main()
{
  std::cout << "Hello World!\n";
}
```
Notice how Code::Blocks auto-inserts the braces and speech quotes.

**STEP 10**  Click File > Save as and save the code with a .cpp extension (HelloWorld.cpp, for example). Code::Blocks changes the view to colour code according to C++ standards. To execute the code, click on the Build and Run icon along the top of the screen. It’s a green play icon together with a yellow cog.
How to Set Up C++ on a Mac

To start C++ coding on a Mac you need to install Apple’s Xcode. This is a free, full featured IDE that’s designed to create native Apple apps. However, it can also be used to create C++ code relatively easily.

XCODE

Apple’s Xcode is primarily designed for users to develop apps for macOS, iOS, tvOS and watchOS applications in Swift or Objective-C, but you can use it for C++ too.

**STEP 1**
Start by opening the App Store on your Mac. Apple Menu > App Store. In the search box enter Xcode and press Return. There will be many suggestions filling the App Store window but it’s the first option, Xcode, that you need to click on.

**STEP 2**
Take a moment to browse through the app’s information, including the compatibility to ensure you have the correct version of macOS. Xcode requires macOS 10.12.6 or later to install and work.

**STEP 3**
When you’re ready, click on the Get button which then turns into Install App. Enter your Apple ID and Xcode begins to download and install. It may take some time depending on the speed of your Internet connection.

**STEP 4**
When the installation is complete, click on the Open button to launch Xcode. Click Agree to the license terms and enter your password to allow Xcode to make changes to the system. When you’ve done that, Xcode begins to install additional components.
How to Set Up C++ on a Mac

**STEP 5** With everything now installed, including the additional components, Xcode launches, displaying the version number along with three choices and any recent projects that you've worked on, although for a fresh install, this is shown blank.

**STEP 6** Start by clicking on Create New Xcode Project; this opens a template window from which to choose the platform you're developing code for. Click the macOS tab, then the Command Line Tool option and finally Next to continue.

**STEP 7** Fill in the various fields but ensure that the Language option at the bottom is set to C++. Simply choose it from the drop-down list. When you've filled in the fields, and made sure that C++ is the chosen language, click on the Next button to continue.

**STEP 8** The next step asks where to create a Git Repository for all your future code. Choose a location on your Mac, or a network location, and click the Create button. When you've done all that, you can start to code. The left-hand pane details the files used in the C++ program you're coding. Click on the main.cpp file in the list.

**STEP 9** You can see that Xcode has automatically completed a basic Hello World program for you. The differences here are that the int main() function now contains multiple functions and the layout is slightly different. This is just Xcode utilizing the content that's available to you Mac.

**STEP 10** When you want to run the code, click on Product > Run. You may be asked to enable Developer Mode on the Mac; this is to authorize Xcode to perform functions without needing your password every session. When the program executes, the output is displayed at the bottom of the Xcode window.
How to Set Up C++ in Linux

Linux is a great C++ coding environment. Most Linux distros already have the essential components preinstalled, such as a compiler and the text editors are excellent for entering code into, including colour coding; there’s also tons of extra software available to help you out.

LINUX++
We’re going to be using a fresh installation of Linux Mint for this particular tutorial. More on Linux Mint can be found in the next section of the book.

STEP 1 The first step when setting up Linux is to ensure your C++ code is up to date. Open a Terminal and enter **sudo apt-get update** & **sudo apt-get upgrade**. Press Return and enter your password. These commands update the entire system and any installed software.

STEP 2 Most Linux distros come preinstalled with all the necessary components to start coding in C++. However, it’s always worth checking to see if everything is present. So still within the Terminal, enter **sudo apt-get install build-essential** and press Return. If you have the right components, nothing is installed but if you’re missing something they are installed by the command.

STEP 3 Amazingly, that’s it. Everything is all ready for you to start coding. Here’s how to get your first C++ program up and running. In Linux Mint the main text editor is Vi. can be launched by clicking on the Menu, typing **Vim** into the search bar. Click on the Text Editor button in the right-hand pane to open Vi.

STEP 4 In Vi, or any other text editor you may be using, enter the lines of code that make up your C++ Hello World program. To remind you, it’s:

```cpp
#include <iostream>

int main()
{
    std::cout << "Hello World!\n";
}
```

#include <iostream>

int main()
{
    std::cout << "Hello World!\n";
}
STEP 5 When you’ve entered your code, click File > Save As and choose a folder where you want to save your program. Name the file as `helloworld.cpp`, or any other name just as long as it has .cpp as the extension. Click Save to continue.

STEP 6 The first thing you can see is that Xcode has automatically recognised this as a C++ file, since the file extension is now set to .cpp. The colour coding is present in the code and if you open up the file manager you can also see that the file’s icon has C++ stamped on it.

STEP 7 With your code now saved, drop into the Terminal again. You need to navigate to the location of the C++ file you’ve just saved. Our example is in the Documents folder, so we can navigate to it by entering `cd Documents`. Remember, the Linux Terminal is case sensitive, so any capitals must be entered correctly.

STEP 8 Before you can execute the C++ file you need to compile it. In Linux it’s common to use gcc, an open source C++ compiler and as you’re now in the same folder as the C++ file, go to the Terminal, enter `gcc helloworld.cpp` and press return.

STEP 9 There will be a brief pause as the code is compiled by gcc and providing there are no mistakes or errors in the code you are returned to the command prompt. The compiling of the code has created a new file. If you enter `ls` into the Terminal you can see that alongside your C++ file is a .o file.

STEP 10 The .o file is the compiled C++ code. To run the code enter `javac` and press Return. The words ‘Hello World!’ appears on the screen. However, javac isn’t very friendly. To name it something else post-compiling, you can recompile with: `gcc helloworld.cpp -o helloworld`. This creates an output file called helloworld which can be run with: `helloworld`.

www.idnpublishations.com
Other C++ IDEs to Install

If you want to try a different approach to working with your C++ code, then there are plenty of options available to you. Windows is the most prolific platform for C++ IDEs but there are plenty for Mac and Linux users too.

DEVELOPING C++
Here are ten great C++ IDEs that are worth looking into. You can install one or all of them if you like, but find the one that works best for you.

**ECLIPSE**
Eclipse is a hugely popular C++ IDE that offers the programmer a wealth of features. It has a great, clean interface, is easy to use and available for Windows, Linux and Mac. Head over to www.eclipse.org/downloads/ to download the latest version. If you’re stuck, click the Need Help link for more information.

**GNAT**
The GNAT Programming Studio (GPS) is a powerful and intuitive IDE that supports testing, debugging and code analysis. The Community Edition is free, whereas the Pro version costs; however, the Community Edition is available for Windows, Mac, Linux and even the Raspberry Pi. You can find it at www.adacore.com/download.

**CODELITE**
Codelite is a free and open source IDE that’s regularly updated and available for Windows, Linux and macOS. It’s lightweight, uncomplicated and extremely powerful. You can find out more information as well as how to download and install it at www.codelite.org.   

**NETBEANS**
Another popular choice is NetBeans. This is another excellent IDE that’s packed with features and a pleasure to use. NetBeans IDE includes project based templates for C++ that give you the ability to build applications with dynamic and static libraries. Find out more at www.netbeans.org/features/cpp/index.html.
**VISUAL STUDIO**
Microsoft’s Visual Studio is a mammoth C++ IDE that allows you
to create applications for Windows, Android, iOS and the web. The Community version is free to download and install but the other
versions allow a free trial period. Go to [www.visualstudio.com](http://www.visualstudio.com) to see what it can do for you.

**ANJUTA**
The Anjuta DevStudio is a Linux-only IDE that
features some of the more advanced features you
would normally find in a paid software development studio. There’s
a GUI designer, source editor, app wizard, interactive debugger and
much more. Go to [www.anjuta.org](http://www.anjuta.org) for more information.

**QT CREATOR**
This cross platform IDE is designed to
create C++ applications for desktop and
mobile environments. It comes with a code editor and integrated
tools for testing and debugging, as well as deploying to you chosen
platform. It’s not free but there is a trial period on offer before

**MONODEVELOP**
This excellent IDE allows developers
to write C++ code for desktop and
web applications across all the major platforms. There’s an advanced
text editor, integrated debugger and a configurable workbench to
help you create your code. It’s available for Windows, Mac and Linux
and is free to download and use. [www.monodevelop.com](http://www.monodevelop.com/).

**DEV C++**
Bloodshed Dev C++, despite its colourful name,
is an older IDE that is for Windows systems only.
However, many users praise its clean interface and uncomplicated
way of coding and compiling. Although there’s not been much
updating for some time, it’s certainly one to consider if you want
something different. [www.bloodshed.net/devcpp.html](http://www.bloodshed.net/devcpp.html).

**UI++**
UI++ is a cross-platform C++ IDE that boasts a rapid
development of code through the smart and aggressive
use of C++. For the novice, it’s a beast of an IDE but behind its
complexity is a beauty that would make a developer’s knees go
wobbly. Find out more at [www.ultimatex.org/index.html](http://www.ultimatex.org/index.html).
C++ Fundamentals

“Controlling complexity is the essence of computer programming.”

- Brian Kernighan
  (Co-developer of UNIX and author)
Within this section, you will begin to understand the structure of C++ code, and how you can compile and execute that code. These are the core fundamentals of C++, which teach you the basics such as using comments, variables, data types and strings, along with how to use C++ mathematics.

These are the building blocks of a C++ program. With them, you will be able to form your own code, produce an output to the screen, and store and retrieve data.

102 Your First C++ Program
104 Structure of a C++ Program
106 Compile and Execute
108 Using Comments
110 Variables
112 Data Types
114 Strings
116 C++ Maths
Your First C++ Program

You may have followed the Mac and Linux examples previously but you’re going to be working exclusively in Windows and Code::Blocks from here on. Let’s begin by writing your first C++ program and taking the first small step into a larger coding world.

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
As mentioned, we’re using Windows 10 and the latest version of Code::Blocks for the rest of the C++ code in this book. Begin by launching Code::Blocks. When open, click on File > New > Empty File or press Ctrl+Shift+N on the keyboard.

STEP 2
Now you can see a blank screen, with the tab labeled “Unlinked”, and the number one in the top left of the main Code::Blocks window. Begin by clicking in the main window, so the cursor is next to the number one, and entering:

```
#include <iostream>
```

STEP 3
At the moment, it doesn’t look like much, and it makes even less sense, but we’ll get to that in due course. Now click on File > Save File As. Create or find a suitable location on your hard drive and in the File Name box, call it hello_world.cpp. Click the Save as type box and select C/C++ Files. Click the Save button.

STEP 4
You can see that Code::Blocks has now changed the colour coding, recognising that the file is now C++ code. This means that code can be auto-selected from the Code::Blocks repository. Delete the `#include <iostream>` line and re-enter it. You can see the auto-select boxes appearing.
Your First C++ Program

**STEP 5**
Auto-selection of commands is extremely handy and cuts out potential typos. Press Return to get to line 3, then enter:
```
int main()
```
Note there’s no space between the brackets.

**STEP 6**
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 7**
Notice that Code::Blocks has automatically created a corresponding closing curly bracket a couple of lines below, linking the pair, as well as 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 8**
Note again the colour coding change. Press Return at the end of the previous step’s line, and then enter:
```
std::cout << "Hello, world!"
```

**STEP 9**
Just as before, Code::Blocks auto-completes the code you’re entering, including placing a closing speech mark as soon as you enter the first. Don’t forget the semicolon at the end of the line; this is one of the most important elements to a C++ program and we’ll tell you why in the next section. For now, move the cursor down to the closing curly bracket and press Return.

**STEP 10**
That’s all you need to do for the moment; it may not look terribly amazing but C++ is best absorbed in small chunks. Don’t execute the code at the moment. As you need to look at how a C++ program is structured first, then you can build and run the code. For now, click on Save. The single floppy disc icon.
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 will 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 restrictions to the already popular C programming language, while making it a more user-friendly environment through new structure.

Bjarne Stroustrup, Inventor of C++

**#INCLUDE**

The structure of a C++ program is quite precise. Every C++ code begins with a directive `#include <c>` in the preprocessor to include a section of the standard C++ code. For example, `#include <iostream>` includes the iostream header to support input/output operations.

**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.

**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.
**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, explain what’s going on. There are two types of comment: /* covers multiple line comments, // a single line. 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, explain what’s going on. There are two types of comment: /* covers multiple line comments, // a single line.

**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 using namespace std: not both. For example:

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

int main()
{
    // My first C++ program
    cout << "Hello, world!\n"
    return 0;
}
```

**OUTPUTS**

Leading on, the “Hello, world!” part is what we 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. The brackets aren’t needed but some compilers insist on them. The \n part indicates a new line is to be inserted.

**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 we leave std: out we have to declare it at the start of the code, as mentioned previously.

```cpp
int main()
{
    using namespace std;
    // My first C++ program
    cout << Hello, world!;
}
```

**; AND }**

Finally you can see that lines within a function code block (except comments) and with a semicolon. This marks the end of the statement and 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.
Compile and Execute

You’ve created your first C++ program and you now understand the basics behind the structure of one. Let’s actually 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 C++ code from Code::Blocks is extraordinarily easy; just a matter of clicking an icon and seeing the result. Here’s how it’s done.

**STEP 1**
Open Code::Blocks, 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 missing semicolons at the end of the statements.

**STEP 3**
Start by clicking on the Build icon, the yellow cog. At this point, your code has now been run through the Code::Blocks compiler and checked for any errors. You can see the results of the build by looking to the bottom window pane. Any messages regarding the quality of the code are displayed here.

**STEP 2**
If your code is looking similar to the one in our screenshot, then look to the menu bar along the top of the screen. Under the Fortran entry in the topmost menu, you can see a group of icons: a yellow cog, green play button and a cog/play button together. These are Build, Run, Build and Run functions.

**STEP 4**
Now click on the Run icon, the green play button. A command line box appears on your screen displaying the words: Hello, world!, followed by the time it’s taken to execute the code, and asking you press a key to continue. Well done, you just compiled and executed your first C++ program.
STEP 5 Press any key in the command line box closes it, returning you to Code::Blocks. 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 8 Create a deliberate error in the code. Remove the semicolon from the `cout` line, so it reads:

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

STEP 6 In order to apply the new changes to the code, you need to recompile, build, and run it again. This time, however, you can simply click the Build icon, the combined yellow cog and green play button.

STEP 9 Now click the Build and Run icon again to apply the changes to the code. This time Code::Blocks refuses to execute the code, due to the error you put in. In the Log pane at the bottom of the screen you are informed of the error. In this case, expected `\' before `\n` tokens, indicating the missing semicolon.

STEP 7 Just as we mentioned in the previous pages, you don't need to have std::cout if you already declare using `namespace std;` at the beginning of the code. We could have easily clicked the Build/Run icon to begin with, but it's worth going through the available options. You can also see that by building and running, the file has been saved.

STEP 10 Replace the semicolon and under the `cout` line, enter a new line to your code:

```
cout << "And greetings from Code::\n"
```

This is simply a line under the last line of outputted text. Build and Run the code, to display your handwork.
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 aspects 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 into 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. In an organization, many programmers work in teams alongside engineers, other developers, hardware analysts and so on. If you’re part of the team that’s writing a key 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 whose job it is to find out what’s wrong with a program. The program has in excess of 100,000 lines of code, spread across several different modules. You can 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 this line or section. You don’t need to comment on every line. Something along the lines of: “if x is 0, don’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 equals pin, it’s 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 publicly share it with others. 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 practice and once you understand how to add a comment where needed, you soon do it as if it’s second nature.
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:

```cpp
//My first C++ program
```  
```cpp
cout << "Hello, world!";
```  
```cpp
/* This is a multi-line comment. */
```  

**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:

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

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

```cpp
// This comment can cover several lines without the need to add more slashes */
```  
```cpp
Just remember to finish the block comment with the opposite asterisk and forward slash:
```  
**STEP 4** Be careful when commenting, especially with block comments. It’s easy to forget to add the closing asterisk and forward slash and then remove any code that falls inside the comment block.

**STEP 5** Obviously if you try and build and execute the code it won’t work, 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 quite time consuming to go back and rectify. Thankfully, the colour coding in CodeBlocks helps identify comments from code.

**STEP 6** If you’re using block comments, it’s good practice 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:

```cpp
/* This comment can cover several lines without the need to add more slashes */
```
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 variables you can declare. Here’s how it works.

**STEP 1**
Open up a new, blank C++ file and enter the usual code headers:
```cpp
#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:
```cpp
int a;
int b;
a = 10;
b = 5;
```

**STEP 3**
Now 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. To output the contents of the variables, add:
```cpp
cout << a;
cout << “The”;
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:
```cpp
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:
```cpp
int a = 10;
int b = 5;
int result = a + b;
cout << result;
```

**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 6** Specific to C++, you can also use the following to assign values to a variable as soon as you declare them:
```cpp
int a = 10;
int b = 5;
then, from the C++ 2011 standard, using curly brackets:
int result = (a+b);
```

**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:
```cpp
#include <iostream> using namespace std;
int main()
{
    auto pi = 3.141593;
    int main()
    {
        double area, radius = 1.5;
        area = pi * radius * radius;
        cout << area;
        }
}
```

**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:
```cpp
#include <iostream>
#include <iostream> using namespace std;
int StartLives = 3;
int main() 
{
    StartLives = StartLives;
    cout << StartLives;
}
```

**STEP 10** A couple of new elements here: first, auto won't work unless you go to Settings > Compiler and tick the box labelled 'Have C++11 ISO C++ Language Standard [std:c++11]'. Then, the new data type, double, which means double-precision floating point value. Enable C++11, then build and run the code. The result should be 7.06858.
Data Types

Variables, as we’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.

The seven basic data types are:

<table>
<thead>
<tr>
<th>TYPE</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>int</td>
</tr>
<tr>
<td>Floating Point</td>
<td>float</td>
</tr>
<tr>
<td>Character</td>
<td>char</td>
</tr>
<tr>
<td>Boolean</td>
<td>bool</td>
</tr>
<tr>
<td>Double Floating Point</td>
<td>double</td>
</tr>
<tr>
<td>Wide Character</td>
<td>wchar_t</td>
</tr>
<tr>
<td>No Value</td>
<td>void</td>
</tr>
</tbody>
</table>

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 2,147,483,647
Signed int = -2,147,483,648 to 2,147,483,647
Short int = -32,768 to 32,767
Unsigned Short int = 0 to 65,535
Signed Short int = -32,768 to 32,767
Long int = -2,147,483,647 to 2,147,483,647
Unsigned Long int = 0 to 4,294,967,295

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 practice 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:

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

double PI = 3.141592653589793238463;
int main()
{
    cout << PI;
    return 0;
}
```

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:

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

double PI = 3.141592653589793238463;
int main()
{
    cout << setprecision(22) << PI;
    return 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 up by the values 0 to 9. If you divide 53 by 3.5, you then 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:

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

typedef int metres;

int main()
{
    metres distance = 15;

    cout << "Distance in metres is: " << distance;
    return 0;
}
```

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 basically 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 are 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:
```cpp
#include <iostream>
using namespace std;

int main()
{
  // Your code here
}
```

**STEP 2**
It's easy to confuse a string with an array. Here's an array, which can be terminated with a null character:
```cpp
#include <iostream>
using namespace std;

int main()
{
  char greet[] = {'W', 'o', 'm', 'e', 'n', 's', 'a', 'i', 'd'};
  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 by a null character. The code can therefore be represented as:
```cpp
#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:
```cpp
#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:
```cpp
#include <iostream>
using namespace std;

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

### 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 string name `operation` build and run the code to see the results.

### Step 7
You can use one of the above operations together, or rather combine them to form longer strings:
```cpp
#include <iostream>
using namespace std;

int main()
{
    string greet1 = "Hello";
    string greet2 = " World!";
    string greet3 = greet1 + greet2;
    cout << greet3 << "\n";
}
```

### 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 in the code:
```cpp
#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:
```cpp
#include <iostream>
using namespace std;

int main()
{
    string str = "Here is a long sentence in a string.\n";
    cout << str << "\n";
    str.erase(18,5);
    cout << str << "\n";
    str.erase(str.begin()+8);
    cout << str << "\n";
    str.erase(str.begin(), str.end()-9);
    cout << str << "\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:
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++ 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 through brackets, for now, create a new file and enter:

```cpp
#include <iostream>
#include <cmath>

int main()
{
    int numbers = 100;
    numbers = numbers * 100; // This multiplies 10 to the initial 100
    cout << numbers << "\n";
    numbers = numbers / 10; // This divides 1000 by 10
    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:

```cpp
#include <iostream>

int main()
{
    float numbers = 100;
    numbers = numbers * 10; // This multiplies 10 by 10
    cout << numbers << "\n";
    numbers = numbers / 10; // And this divides 100 by 10
    cout << numbers << "\n";
}
```

**STEP 4**

Again, execute the simple code and see the results. While not particularly interesting, it’s a start into 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 <cmath>
#include <iostream>
using namespace std;
```
```
int main()
{
    double radius = 5.0;
    double area = M_PI * pow(radius, 2);
    cout << "The area is: " << area << "m^2."
    return 0;
}
```

STEP 6
Start by getting the square root of a number:
```
#include <iostream>
#include <cmath>
using namespace std;
```
```
int main()
{
    double number = 134;
    cout << "The square root of " << number << " is: " << sqrt(number) << "m."
    return 0;
}
```

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.
```
#include <iostream>
#include <cmath>
using namespace std;
```
```
int main()
{
    double radius = 5.0;
    double area = M_PI * pow(radius, 2);
    cout << "The area is: " << area << "m^2."
    return 0;
}
```

STEP 8
Calculating powers of numbers can be done with:
```
#include <iostream>
#include <cmath>
using namespace std;
```
```
int main()
{
    double number = 12;
    cout << "The power of 2 is: " << pow(number, 2) << "m."
    cout << "The power of 3 is: " << pow(number, 3) << "m."
    cout << "The power of 0.88 is: " << pow(number, 0.88) << "m."
    return 0;
}
```

STEP 9
Here we created a float called number with the value of 12, and the pow(number, 2) 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.
```
#include <iostream>
#include <cmath>
using namespace std;
```
```
int main()
{
    double number = 12;
    double area = M_PI * pow(number, 2);
    cout << "The area is: " << area << "m^2."
    return 0;
}
```

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 << "The area is: " << area << "m^2."
    return 0;
}
```
C++ Input/Output
“The best programs are written so that computing machines can perform them quickly, and so that human beings can understand them clearly.”

– Donald E. Knuth

(Computer scientist, mathematician and author)

There’s a satisfying feeling when you program code that asks the user for input, then uses that input to produce something that the user can see. Even if it’s simply asking for someone’s name, and displaying a personal welcome message, it’s a big leap forward.

User interaction, character literals, defining constants, and file input and output are all covered in the following pages. All of which will help you to better understand how a C++ program works.

120 User Interaction
122 Character Literals
124 Defining Constants
126 File Input/Output
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.

```cpp
#include <iostream>
#include <iomanip>

int main ()
{
    int age;
    cout << "What is your age: ";
    cin >> age;
    cout << "You are " << age << " years old." << endl;
    return 0;
}
```

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:

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

int main ()
{
    int age;
    cout << "What is your age: " << endl;
    cin >> age;
    cout << "You are " << age << " years old." << endl;
    return 0;
}
```

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.

```cpp
#include <iostream>
#include <iomanip>

int main ()
{
    int age;
    cout << "What is your age: ";
    cin >> age;
    cout << "You are " << age << " years old." << endl;
}
```

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:

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

int main ()
{
    string name;
    cout << "What is your name: ";
    cin >> name;
    cout << "Hello, " << name << ", I hope you’re well today!\n";
    return 0;
}
```
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 whenever you like in your code.

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

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

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:

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

int main()
{
    int num1, num2;
    cout << "Enter two whole numbers: \n";
    cin >> num1 >> num2;
    cout << "You entered \"" << num1 << \"\" and \"" << num2 << \"\"\n";
}
```

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:

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

int main()
{
    float num1, num2;
    cout << "Enter two numbers: \n";
    cin >> num1 >> num2;
    cout << num1 + num2 << \"\" << num1 - num2 << \"\" << num1 * num2 << \"\" << num1 / num2 << \n";
}
```

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:

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

int main()
{
    string myst;   
    cout << "Enter a sentence: \n";
    getline(cin, myst);
    cout << "Your sentence is: \"" << myst.size() << \" characters long.\n";
}
```

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 >> myst and try again, the result displays the number of characters up to the first space.

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

int main()
{
    string myst;  
    cout << "Enter a sentence: \n";
    cin >> myst;
    cout << "Your sentence is: \"" << myst.size() << \" characters long.\n";
}
```

STEP 10
Caret 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:

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

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

In C++, a literal is an object or variable that, once defined, remains the same throughout the code. However, a character literal is defined by a backslash, such as the \n you’ve been using at the end of a cout statement to signify a new line.

ESCAPE SEQUENCE

When used in something like a cout statement, character literals are also called escape sequence codes. They allow you to insert a quote, an alert, new line and much more.

**STEP 1** Create a new C++ file and enter the relevant headers:
```cpp
#include <iostream>
#include <sstream>

int main ()
{
    return 0;
}
```

**STEP 3** If you wanted to insert speech quotes inside a cout statement, you would have to use a backslash as it already uses quotes:
```cpp
#include <iostream>
#include <sstream>

int main ()
{
    cout << "Hello, user. This is how to use \"quotes\"."
    return 0;
}
```

**STEP 2** You’ve already experienced the ‘\n’ character literal placing a new line wherever it’s called. The line cout << “Hello\n” << “I’m a C++\n” << “Programmer” outputs three lines of text, each starting after the last ‘\n’.
```cpp
#include <iostream>

int main ()
{
    cout << "Hello\nI’m a C++\nProgrammer"
    return 0;
}
```

**STEP 4** There’s even a character literal that can trigger an alarm. In Windows 10, it’s the notification sound that chimes when you use \a. Try this code, and turn up your sound.
```cpp
#include <iostream>

int main ()
{
    cout << "ALARM! \a"
    return 0;
}
```
A HANDY CHART

There are numerous character literals, or escape sequence codes, to choose from. We therefore thought it would be good for you to have a handy chart available, for those times when you need to insert a code.

<table>
<thead>
<tr>
<th>ESCAPE SEQUENCE CODE</th>
<th>CHARACTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>\</td>
<td>Backslash</td>
</tr>
<tr>
<td>'</td>
<td>Single Quote</td>
</tr>
<tr>
<td>&quot;</td>
<td>Double Quote Speech Marks</td>
</tr>
<tr>
<td>?</td>
<td>Question Mark</td>
</tr>
<tr>
<td>a</td>
<td>Alert/Alarm</td>
</tr>
<tr>
<td>b</td>
<td>Backspace</td>
</tr>
<tr>
<td>f</td>
<td>Form Feed</td>
</tr>
<tr>
<td>n</td>
<td>New Line</td>
</tr>
<tr>
<td>r</td>
<td>Carriage Return</td>
</tr>
<tr>
<td>t</td>
<td>Horizontal Tab</td>
</tr>
<tr>
<td>v</td>
<td>Vertical Tab</td>
</tr>
<tr>
<td>\u0000</td>
<td>Null Character</td>
</tr>
<tr>
<td>\u0000xx</td>
<td>Unicode (UTF-8)</td>
</tr>
<tr>
<td>\u0000xxxx</td>
<td>Unicode (UTF-16)</td>
</tr>
</tbody>
</table>

UNICODE CHARACTERS (UTF-8)

Unicode characters are symbols or characters that are standard across all platforms. For example, the copyright symbol, that can be entered via the keyboard by entering the Unicode code, followed by ALT+X, in the case of the copyright symbol enter 00A9 ALT+X. In C++ code, you would enter:

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

int main ()
{
    cout << "©";
    return 0;
}
```

UNICODE CHARACTER TABLE

A complete list of the available Unicode characters can be found at www.unicode-table.com/en/. Move your mouse over the character to see the unique code to enter in C++. While it may be a little overwhelming to look at at first, bookmark the page as you will probably need to come back to it for reference as you dig deeper into C++, and indeed character literals. One more thing, the table will also display characters from different languages, such as Tibetan or Sudanese. This means your code can be truly universal.
Defining Constants

Constants are fixed values in your code. They can be any basic data type but as the name suggests their value remains constant throughout the entire code. There are two separate ways to define a constant in C++, the `#define` pre-processor and `const`.

### #DEFINE

The pre-processors are instructions to the compiler to pre-process the information before it goes ahead and compiles the code. `#include` is a pre-processor as is `#define`.

**STEP 1**

You can use the `#define` pre-processor to define any constants you want in our code. Start by creating a new C++ file complete with the usual headers:

```cpp
#include <iostream>
#include <string>
int main ()
{
    return 0;
}
```

**STEP 2**

Now let's assume your code has three different constants: length, width, and height. You can define them with:

```cpp
#include <iostream>
#include <string>
#define LENGTH 50
#define WIDTH 40
#define HEIGHT 60
int main ()
{
    return 0;
}
```

**STEP 3**

Now the capital for defined constants, it's considered good programming practise to define all constants in capitals. Here, the assigned values are 50, 40, and 60, so let's call them up:

```cpp
#include <iostream>
#include <string>
#define LENGTH 50
#define WIDTH 40
#define HEIGHT 60
int main ()
{
    cout << "Length is: " << LENGTH << " m."
    cout << "Width is: " << WIDTH << " m."
    cout << "Height is: " << HEIGHT << " m."
    return 0;
}
```

**STEP 4**

Build and run the code. Just as expected, it displays the values for each of the constants created. It's worth noting that you don't need a semicolon when you're defining a constant with the `#define` keyword.
**Defining Constants**

**STEP 5**
You can also define other elements as a constant. For example, instead of using `printf` for a newline in the `cout` statement, you can define it at the start of the code:

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

#define LENGTH 50
#define WIDTH 40
#define NEWLINE "\n"

int main()
{
    cout << "Length is: " << LENGTH << NEWLINE;
    cout << "Width is: " << WIDTH << NEWLINE;
    cout << "Height is: " << HEIGHT << NEWLINE;
    return 0;
}
```

**STEP 6**
The code, when build and executed, does exactly the same as before, using the new constant `NEWLINE` to insert a newline in the `cout` statement. Incidentally, creating a newline constant isn't a good idea unless you're making it smaller than `printf` or even the `endl` command.

**STEP 7**
Defining a constant is a good way of initialising your base values at the start of your code. You can define that your game has three lives, or even the value of pi without having to call up the C++ math library:

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

const double PI = 3.14159;

int main()
{
    cout << "The value of PI is: " << PI << endl;
    return 0;
}
```

**STEP 8**
Another method of defining a constant is with the `const` keyword. Use `const` together with a data type, variable and value. Constant variable `value = value`. Using `PI` as an example:

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

int main()
{
    const double PI = 3.14159;
    cout << "The value of PI is: " << PI << endl;
    return 0;
}
```

**STEP 9**
Because you're using `const` within the main block of code, you need to finish the line with a semicolon. You can use either, as long as the names and values don't clash, but it's worth mentioning that `#define` requires no memory, so if you're coding to a set amount of memory, `#define` is your best bet.

**STEP 10**
`const` works in much the same way as `#define`. You can create static integers and even newlines:

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

int main()
{
    const int LENGTH = 50;
    const int WIDTH = 40;
    const char NEWLINE = "\n";
    int area = LENGTH * WIDTH;
    cout << "Area is: " << area << NEWLINE;
    return 0;
}
```
File Input/Output

The standard iostream library provides C++ coders with the cin and cout input and output functionality. However, to be able to read and write from a file you need to utilise another C++ library, called fstream.

FSTREAMS

There are two main data types within the fstream library that are used to open a file, read from it and write to it, ofstream and ifstream. Here’s how they work.

**STEP 1**

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

int main ()
{
}
```

The first task is to create a new C++ file and along with the usual headers you need to include the new fstream header:

**STEP 2**

Begin by asking a user for their name and writing that information to a file. You need the usual string to store the name, and getline to accept the input from the user.

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

int main ()
{
    string name;
    ofstream outfile;
    outfile.open("name.txt");
    cout << "Enter your name: " << endl;
    getline(cin, name);
    outfile << name << endl;
    outfile.close();
}
```

**STEP 3**

We’ve included comments in the screenshots of step 2 to help you understand the process. You created a string called name, to store the user’s inputted name. You also created a text file called name.txt (with the ofstream ofstream and ofstream.open(line)), asked the user for their name and stored it and then written the data to the file.

**STEP 4**

To read the contents of a file, and output it to the screen, you need to do things slightly differently. First you need to create a string variable to store the file’s contents (line by line), then open the file, use getline to read the file line by line and output those lines to the screen. Finally, close the file.

```cpp
string line;
ifstream infile("name.txt");
cout << "Contents of the file: " << endl;
generate(infile, line);
close < line << endl;
newfile.close();
```
The code above is great for opening a file with one or two lines but what if there are multiple lines? Here we opened a text file of the poem 'Cimarria', by Robert E. Howard:

```cpp
string line; ifstream newfile("C:\Users\David\Documents\Cimarria.txt");
cout << "Cimarria, by Robert E. Howard: " << endl;
while (getline(newfile, line))
    cout << line << endl;
newfile.close();
```

You can no doubt see that we've included a while loop, which we cover in a few pages time. It means that while there are lines to be read from the text file, C++ gets them. Once all the lines are read, the output is displayed on the screen and the file is closed.

You may also see that the location of the text file 'Cimarria.txt' isn't in the same folder as the C++ program. When we created the first name.txt file, it was written to the same folder where the code was located; this is done by default. To specify another folder, you need to use double back slashes, as per the character literal/escape sequence code.

```cpp
string line; ifstream newfile("C:\Users\David\Documents\Cimarria.txt");
cout << "Cimarria, by Robert E. Howard: " << endl;
while (getline(newfile, line))
    cout << line << endl;
newfile.close();
```

The code from step 8 differs again, but only where it comes to adding the age integer. Notice that we used cin >> age, instead of the previous getline(cin, variable). The reason for this is that the getline function handles strings, not integers, so when you're using a data type other than a string, use the standard cin.

Here's an exercise: see if you can create code to write several different elements to a text file. You can have a user's name, age, phone number etc. Maybe even the value of PI and various mathematical elements. It's all good practice.

```cpp
int main()
{
    int age = 19;
    char name[20];
    cout << "What is your name? " << endl;
    cin >> name;
    cout << "Enter your age: " << endl;
    cin >> age;
    ofstream newfile("C:\Users\David\Documents\Cimarria.txt");
    newfile << name << " age: " << age << endl;
    newfile.close();
    cout << "The square root of PI is " << sqrt(PI) << endl;
    return 0;
}
```
Loops and Decision Making

“The most important property of a program is whether it accomplishes the intention of its user.”

— C.A.R. Hoare (Computer scientist, developer of Quicksort)
Loops and repetition are one of the most important factors of any programming language. Good use of a loop will create a program that does exactly what you want it to, and delivers the desired outcome without issues or errors.

Without loops and decision-making events within the code, your program will never be able to offer the user any choice. It’s this understanding of choice that elevates your skills as a programmer, and makes for much better code.

130 While Loop
132 For Loop
134 Do...While Loop
136 If Statement
138 If...Else Statement
While Loop

A while loop’s function is to repeat a statement, or a group of statements, while a certain condition remains true. When the while loop starts, it initializes itself by testing the condition of the loop and the statements within, before executing the rest of the loop.

TRUE OR FALSE?

While loops are one of the most popular form of C++ code looping. They repeatedly run the code contained within the loop while the condition is true. Once it proves false, the code continues as normal.

**STEP 1**

What you’ve done so far and create a new C++ file. There’s no need for any extra headers at the moment, so set the standard headers as per usual:

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

int main()
{

}
```

**STEP 2**

Create a simple while loop. Enter the code below, build and run (we’ve added comments to the screen shot):

```cpp
int num = 1;
while (num < 30)
{
    cout << “Number: “ << num << endl;
    num++;
}
return 0;
```

**STEP 3**

We need to create a condition, so use a variable called `num` and give it the value 1. Now create the while loop, stating that as long as `num` is less than 30, the loop is true. Within the loop the value of `num` is displayed and adds 1 until it’s more than 30.

```cpp
int num = 1;
while (num < 30)
{
    cout << “Number: “ << num << endl;
    num++;
}
return 0;
```

**STEP 4**

We’re introducing a few new elements here. The first are the opening and closing braces for the while loop. This is because our loop is a compound statement, meaning a group of statements; note also, there’s no semicolon after the while statement. You now also have return 0, which is a clean and preferred way of ending the code.

```cpp
int num = 1;
while (num < 30)
{
    cout << “Number: “ << num << endl;
    num++;
}
return 0;
```

**STEP 5**

If you didn’t need to see the continually increasing value of `num`, you could have done away with the compound while statement and instead just added 1 by itself until it reached 30, and then displayed the value.

```cpp
int num = 1;
while (num < 30)
{
    num++;
    cout << “Number: “ << num << endl;
}
return 0;
```
**STEP 6**

It's important to remember not to add a semicolon at the end of a while statement. Why? Well, as you know, the semicolon represents the end of a C++ line of code. If you place one at the end of a while statement, your loop will be permanently stuck until you close the program.

```c++
while (condition) {
    // statements
}
```

**STEP 7**

In our example, if we were to execute the code, the value of num would be 1, as set by the int statement. Since the code hits the while statement, it reads that while the condition of 1 being less than 30 is true, loop. The semicolon closes the line, so the loop repeats; but it never adds 1 to num, as it won’t continue through the compound statement.

```c++
int num = 1;
while (num < 30) {
    // statements
}
```

**STEP 8**

In our example, if we were to execute the code, the value of num would be 1, as set by the int statement. Since the code hits the while statement, it reads that while the condition of 1 being less than 30 is true, loop. The semicolon closes the line, so the loop repeats; but it never adds 1 to num, as it won’t continue through the compound statement.

```c++
int num = 1;
while (num < 30) {
    // statements
}
```

**STEP 9**

You can further expand the code to enable each word of the poem to appear every second. To do so, you need to pull in a new library, `windows.h`. This is a Windows only library and within it you can use the `Sleep` function:

```c++
# include <iostream>
# include <fstream>
# include <windows.h>
using namespace std;

int main(int argc, char *argv[]) {
    cout << "Cimmeria, by Robert E. Howard: \n"
         << endl;
    while (remfile >> word) {
        cout << word << endl;
        Sleep(1000);
    }
    return 0;
}
```

**STEP 10**

You can manipulate the while statement to display different results depending on what code lies within the loop. For example, to read the poem, Cimmeria, word by word, you would enter:

```c++
# include <iostream>
# include <fstream>
using namespace std;

int main() {
    string word;
    ifstream remfile("C:\\Users\\david\\Documents\\Cimmeria.txt");
    while (remfile >> word) 
        cout << word << endl;
    return 0;
}
```

**Sleep()** works in milliseconds, so Sleep(1000) is one second, Sleep(10000) is ten seconds and so on. Combining the sleep function (along with the header it needs) and a while loop enables you to come up with some interesting countdown code.

```c++
# include <iostream>
# include <fstream>
using namespace std;

int main() {
    int a = 30;
    while (a > 0) {
        cout << a << "s... \n"
             << endl;
        a -= 1;
        Sleep(1000);
    }
    cout << "All Told Off!\n"
         << endl;
    return 0;
}
```
Loops and Decision Making

For Loop

In some respects, a for loop works in a very similar way to that of a while loop, although its structure is different. A for loop is split into three stages: an initialiser, a condition and an incremental step. Once set up, the loop repeats itself until the condition becomes false.

LOOPY LOOPS

The initialise stage of a for loop is executed only once and this sets the point reference for the loop. The condition is evaluated by the loop to see if it’s true or false and then the increment is executed. The loop then repeats the second and third stage.

**STEP 1**
Create a new C++ file, with the standard headers:

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

int main()
{
  // Code goes here...
}
```

**STEP 2**
Start simple and create a for loop that counts from 1 to 30, displaying the value to the screen with each increment:

```cpp
// For Loop Begins
for (int num = 1; num <= 30; num = num + 1)
{
  cout << "Number: " << num << endl;
}
return 0;
// For Loop Ends
```

**STEP 3**
Working through the process of the for loop, begin by creating an integer called num and assigning it a value of 1. Next, set the condition, in this case num being less than 30. The last stage is where you create the increments; here it’s the value of num being added by 1.

```cpp
// For Loop Begins
for (int num = 1; num <= 30; num = num + 1)
{
  cout << "Number: " << num << endl;
}
return 0;
// For Loop Ends
```

**STEP 4**
After the loop, you created a compound statement in braces (curly brackets), that displays the current value of the integer num. Every time the for loop repeats itself, the second and third stages of the loop, it adds 1 until the condition <30 is false. The loop then ends and the code continues, ending neatly with return 0.

```cpp
for (int num = 1; num <= 30; num = num + 1)
{
  cout << "Number: " << num << endl;
}
return 0;
```

**STEP 5**
A for loop is quite a neat package in C++, as contained within it’s own brackets, while the other elements outside of the loop are displayed below. If you want to create a 10-second countdown, you could use:

```cpp
#include <iostream>
#include <windows.h>
using namespace std;

int main()
{
  // For Loop Begins
  for (int a = -10; a >= -1; a = a - 1)
  {
    cout << a << endl;
    Sleep(1000);
  }
  cout << "Blast Off!" << endl;
  return 0;
  // For Loop Ends
}
```
**STEP 6**
With the countdown code, don’t forget to include the windows\library, so you can use the `Sleep` command. Build and run the code. In the command console you can see the numbers 10 to 1 countdown in one second increments, until it reaches zero and `BlacCoff` appears.

**STEP 7**
Naturally you can include a lot more content into a for loop, including some user input:
```cpp
int i, n; fact = 1;
cin >> n;
for (i = 1; i <= n; ++i) {
fact *= i;
}
cout << "\nFactorial of " << n << " = " << fact << endl;
return 0;
```

**STEP 8**
The code from step 7, when built and run, asks for a number, then displays the factorial of that number through the for loop. The user’s number is stored in the integer n, followed by the integer i, which is used to check if the condition is true or false, adding 1 each time and comparing it to the user’s number, n.

**STEP 9**
Here’s an example of a for loop displaying the multiplication tables of a user inputted number.
Handy for students:
```cpp
int n;
cout << "Enter a number to view its times table: 
";
cin >> n;
for (int i = 1; i <= 12; ++i) {  
cout << n << " x " << i << " = " << n * i << endl;  
}
return 0;
```

**STEP 10**
The value of the integer can be expanded from 12 to whatever number you want, displaying a very large multiplication table in the process (or a small one). Of course the data type within a for loop doesn’t have to be an integer; as long as it’s valid, it works.
```cpp
for (float i = 0.00; i <= 1.00; i = 0.01)  
{  
cout << i;  
}
return 0;
```
Do... While Loop

A do... while loop differs slightly from that of a for or even a while loop. Both for and while set and examine the state of the condition at the start of the loop, or the top of the loop if you prefer. However, a do... while loop, is similar to a while loop but instead checks the condition at the bottom of the loop.

**DO LOOPS**

The good thing about a do... while loop is that it’s guaranteed to run through at least once. Its structure is do, followed by statements, while condition is true. This is how it works.

**STEP 1** Begin with a new blank C++ file and enter the standard headers:
```cpp
#include <iostream>
using namespace std;
int main()
{
}
```

**STEP 2** Begin with a simple number count:
```cpp
int num = 1;
do{
    cout << "Number: " << num++ << endl;
    num = num + 1;
}while (num < 30);
return 0;
```

**STEP 3** Now, here's a look at the structure of a do... while loop. First you create an integer called num, with the value of 1. Now the do... while loop begins. The code inside the body of the loop is executed at least once, then the condition is checked for either true or false.

```cpp
while (num <= 30)
{
    // Do something with the value of num
    num += 1;
    // Display the current value of num
}
```

**STEP 4** If the condition is true, the loop is executed. This continues until the condition is false. When the condition has been expressed as False, the loop terminates and the code continues. This means you can create a loop where the code continues until the user enters a certain character.
**STEP 5** If you want code to add up user inputted numbers until the user enters zero:

```c
float number, sum = 0.0;
cout << "***** Program to execute a Do...While loop continuously *****" << endl;
cout << "Enter 0 to stop and display the sum of all the numbers entered!" << endl;
cout << "V" << endl;
while (number != 0.0)
{
    cout << "Please enter a number: ";
    cin >> number;
    sum += number;
}
cout << "Total sum of all numbers: " << sum;
return 0;
```

**STEP 6** The code from Step 5 works as follows: Two floating point variables are assigned, number and sum, both with the value of 0.0. There is a brief set of instructions for the user, then the do...while loop begins.

**STEP 7** The do...while loop in this instance asks the user to input a number, which you assigned to the float variable, number. The calculation step uses the second floating point variable, sum, which adds the value of number every time the user enters a new value.

```c
do {
    cout << "Please enter a number: ";
    cin >> number;
    sum += number;
}
```

**STEP 8** Finally, the while statement checks the condition of the variable number. If the user has entered zero, then the loop is terminated, if not then it continues indefinitely. When the user finally enters zero, the value of sum, the total value of all the user’s input, is displayed. The loop, and the program, then ends.

**STEP 9** Using the countdown and Blast Off! code used previously, a do...while loop would look like:

```c
int a = 10;
do {
    cout << a << endl;
    a = a - 1;
} while (a != 0);
cout << "Blast Off!" << endl;
return 0;
```

**STEP 10** The main advantage of using a do...while loop is that it’s an exit-condition loop; whereas a while loop is an entry-control loop. Therefore, if your code requires a loop that needs to be executed at least once (for example, to check the number of lives in a game), then a do...while loop is perfect.
If Statement

The decision making statement "if" is probably one of the most used statements in any programming language, regardless of whether it's C++, Python, BASIC or anything else. It represents a junction in the code, where IF one condition is true, do this; or IF it's false, do that.

**IF ONLY**

If uses a Boolean expression within its statement. If the Boolean expression is true, the code within the statement is executed. If not, then the code after the statement is executed instead.

---

**STEP 1**
First, create a new C++ file and enter the relevant standard headers, as usual:
```cpp
#include <iostream>
using namespace std; 
int main ()
{

}
```

**STEP 2**
If is best explained when you use a number-based condition:
```cpp
int num = 3;
if ( num < 30 )
{
    cout << "The number is less than 30." << endl; 
    cout << "Value of number is:" << num << endl;
    return 0;
}
```

**STEP 3**
What's going on here? To begin, an integer called `num` was created and assigned with the value of 3. The `if` statement comes next, and in this case we've instructed the code that if the condition, the value of `num` is less than 1, then the code within the braces should be executed.

**STEP 4**
The second `cout` statement displays the current value of `num` and the program terminates safely. It's easy to see how the if statement works. If you were to change the initial value of `num` from 1 to 31,
**STEP 5** When you change the value to anything above 30, then build and run the code, you can see that the only line to be outputted to the screen is the second if statement, displaying the current value of num. This is because the initial if statement is false, so it ignores the code within the braces.

**STEP 6** You can include an if statement within a do...while loop. For example:

```c
float temp;

do {
    cout << "Enter the temperature (or -10000 to exit): " << endl;
    cin >> temp;
    if (temp <= 0)
        cout << "Error, it’s really cold!" << endl;
    else if (temp > 30)
        cout << "At least it’s not freezing!" << endl;
    while (temp != 10000);
} while (temp != 10000);
return 0;
```

**STEP 7** The code in Step 6 is simplistic but effective. First we created a floating point integer called temp, then a do...while loop that asks the user to enter the current temperature.

**STEP 8** The first if statement checks to see if the user’s inputted value is lesser or equal to zero. If it is, then the output is ‘Error, it’s really cold!’. Otherwise, if the input is greater than zero, the code outputs ‘At least it’s not freezing!’.

**STEP 9** Finally, if the user enters the value -10000, which is impossible (and so is therefore an unrealistic value), the do...while loop is terminated and a friendly ‘Good bye!’ is displayed to the screen.

**STEP 10** Using if is quite powerful, if it’s used correctly. Just remember that if the condition is true then the code executes what’s in the braces. If not, it continues on its merry way. See what else you can come up with using if and a combination of loops.
Loops and Decision Making

If... Else Statement

There is a much better way to use an if statement in your code, with if... else. If... else works in much the same way as a standard if statement. If the Boolean expression is true, the code within the braces is executed. Else, the code within the next set of braces is used instead.

IF YES, ELSE NO

There are two sections of code that can be executed depending on the outcome in an if... else statement. It's quite easy to visualise once you get used to its structure.

**STEP 1**

Begin with a new C++ file and the standard headers:
```cpp
#include <iostream>
#include <string>

int main ()
{ }
}
```

**STEP 2**

Let's expand the code from the If Statement on the previous page:
```cpp
int num = 1;
if ( num < 30 )
{ }
else
{ }
```

**STEP 3**

The first line in the code creates the integer called num and gives it a value of 1. The if statement checks to see if the value of num is less than thirty and if it is, it outputs "The number is less than 30" to the console.
```cpp
if ( num < 30 )
{ }
else
{ }
```

**STEP 4**

The else comparison to if checks if the number is greater than 30 and if so, then displays "The number is greater than 30" to the console, and finally, the code is terminated satisfactorily.
```cpp
else
{ }
```

**STEP 5**

You can change the value of num in the code or you can improve the code by asking the user to enter a value:
```cpp
int num;
if ( num < 30 )
{ }
else
{ }
```

126  SDHC's Definitive Guide Series - Volume 50
STEP 6 The code works the same way as you would expect, but what if you wanted to display something if the user entered the number 30? Try this:

```cpp
int num;
cout << "Enter a number: " ;
cin >> num;
if (num < 30)
    cout << "The number is less than 30!" << endl;
else if (num > 30)
    cout << "The number is greater than 30!" << endl;
else if (num == 30)
    cout << "The number is exactly 30!" << endl;
return 0;
```

STEP 7 The new addition to the code is what's known as a nested if...else statement. This allows you to check for multiple conditions. In this case, if the user enters a number less than 30, greater than 30 or exactly 30 itself, a different outcome is presented to them.

STEP 8 You can take this up a notch and create a two-player number guessing game. Begin by creating the variables:

```cpp
int num, guess, tries = 0;
```

```cpp
cout << "***** Two-Player number guessing game *****" << endl;
```

```cpp
num = Player One, enter a number for Player Two to guess; " << endl;
```

```cpp
cin >> num;
```

```cpp
string("(0, " << num);  
```

STEP 9 The `cout << string("(0, " << num);` line clears the screen so Player Two doesn't see the entered number. Now you can create a do...while loop, together with if...else:

```cpp
do {
    cout << "" << Player Two, enter your guess: ";
    cin >> guess;
    tries ++;
    if (guess > num)
        cout << "Too High!" << endl;
    else if (guess < num)
        cout << "Too Low!" << endl;
    else if (guess == num)
        cout << "Well done! You got it in " << tries << " guesses!" << endl;
}
while (guess != num);
```

STEP 10 Grab a second player, then build and run the code. Player One enters the number to be guessed, then Player Two can take as many guesses as they need to get the right number. Want to make it harder? Maybe use decimal numbers.
Keep:: Coding
“First, solve the problem. Then, write the code.”

— John Johnson (Developer and author)

By now you will have a basic understanding of how code works, and you can apply both Python and C++ to create games, tools and interactive scripts. Even so, your journey as a coder has only just begun, so in this section you’ll find some fun and interesting code projects to help get you started.

In here, you’ll discover code that creates ASCII art animations, interconnecting variables between the OS and code, retro coding on an emulated 80s computer and code that can keep track of the International Space Station as it orbits the earth.

You’ll also find a collection of common pitfalls to look out for, plus how to avoid them. From there, it’s up to you and your imagination as to where to go next.

142 Passing Variables to Python
144 Retro Coding
146 Text Animations
148 Creating a Loading Screen
150 Tracking the ISS with Python
154 Using Text Files for Animation
156 Common Coding Mistakes
158 Python Beginner’s Mistakes
160 C++ Beginner’s Mistakes
Passing Variables to Python

Here's an interesting coding tutorial: how to pass a system variable to Python. By this we mean, if you create a variable within a Windows batch file or Linux Bash script, then you're able to use that same variable from inside Python.

**THE WINDOWS WAY**

The two systems use slightly different ways to accomplish the same task; Windows is slightly easier (by a single command) so we'll start with that.

**STEP 1**
Let's begin by creating a sample folder within your Python directory, call it batpy, for example, within the batpy folder, create a new text file and enter the following:
```
set /a name = "What is your name?"
```
```
echo Let's try and pass this to a Python script...
```
```
python batpy1.py
```
```
echo It worked!!!
```
```
set /a name """"" echo """"It worked!!!"
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
**THE LINUX WAY**

Linux’s version of passing variables to Python is slightly different, just because it’s Linux! It’s easy, though, and here’s how it’s done.

**STEP 1** Start by opening your favourite Linux text editor, and entering the following:

```bash
#!/bin/bash
echo -n "Hello, what is your name?"
read surname
export surname
python3 babyp1.py
```

**STEP 2** Drop into a Terminal session and make a new directory called babyp1/ and enter the following:

```bash
mkdir babyp1
```

**STEP 3** Now create a new Python file, and enter the following:

```python
n = os.environ["firstname"]
print(f"Your name is {n}")
```

**STEP 4** Back in the Terminal, enter `./babyp1.py` to run the Bash script. As you can see, the results are the same as in Windows. The major differences are making the Bash script executable, and adding the `export` command to the Bash file.
There's a school of thought on the Internet, that to master the foundations of good coding skills you need to have some experience of how code was written in the past. In the past is a bit of a loose term, but mostly, it means coding from the 80s.

**THE GOLDEN ERA OF CODE**

Difficult may seem a little counterproductive to learn how to code in a language that's virtually obsolete, there are some surprising benefits to getting your hands dirty with a bit of retro coding. Firstly, learning old code will help you build the structure of code as, regardless of whether it is a language that was developed yesterday, or forty years ago, code still demands strict discipline to work correctly. Secondly, everyday coding elements such as loops, sub-routines and so on are a great visual aid to learn in older code, especially BASIC. Lastly, it’s simply good fun.

**GOING BASIC**

The earliest retro language to pop around with is, without doubt, BASIC. Developed back in the mid-60s, BASIC (Beginner's All-Purpose Symbolic Instruction Code) is a high-level programming language whose design was geared toward ease of use. In a time when computers were beginning to become more accessible, designers John Kemeny and Thomas Kurtz needed a language that students could get to grips with, quickly and easily. Think of BASIC as a distant relation to Python.

**THE BEEB**

The problem with BASIC is that there were so many different versions available, across multiple BASIC platforms, with each having its own unique elements on top of the core. BASIC functions. The BASIC that was packaged with the Commodore44 was different to that on the ZX Spectrum, or the Atari home computers, due to the differing hardware of each system. However, it is widely recognized that one of the first, and possibly most utilized, forms of BASIC from the 80s was that of BBC BASIC. BBC BASIC was written in the Acorn BBC range of computers, utilizing the M6502-based processor technology. It was one of the earliest examples of BASIC and, thanks to an in-line assembler, it was also capable of allowing the developers of the time to write code for different processor types, such as the Zilog Z80—a CPU present in the ZX Spectrums or, as well as many arcade machines. The BBC Mircos was designed and built by Acorn Computers—a company that is historically responsible for the creation of the ARM CPU, the processor that is used in virtually every Android phone and tablet, as well as smart TVs, set top boxes and so on. The BBC Mircos was built around a time when the UK government was looking for a countrywide computer platform to be used throughout educations' different companies bid, but it was the BBC's Computer Literacy Project (the BBC Micro) that was chosen, due to its ruggedness, upgradability, and potential for education. As a result, the BBC Micro, or the Beeb as its affectionately known, became the dominant educational computer throughout the 80s.

**BEEBEM**

Naturally, you could scour eBay and look for a working BBC Micro to play around on, and it'll be a lot of fun. However, for the sake of playing around with some retro code, we'll use one of the best BBC Micro emulators available: BeebEm.

BeebEm was originally developed for Unix in 1994 by Dave Gilbert and later ported to Windows. It is now developed by Mike Walsh and Jon Walsh, who maintain the Mac port of the emulator, and is therefore available for Windows 10, Linux and macOS, as well as other platforms. If you're using Windows, 10, simply navigate to http://www.mikewe.me.uk/beebem/index.html, and download the BeebEm4.14-0.exe that's displayed in the main screen.
Once downloaded, launch the executable and follow the on-screen instructions to install it. Linux users can find the installation files, as well as full instructions, at http:// peebeem- unix. bbcmicro.com/download.html. Mac and OS X users can get everything they need from http:// www. cfil.com/.

**BBC BASIC**

Once installed and powered up, BASIC will display the default BBC logo and start up, along with a routine of beeps. Those of you old enough to have been in a UK school in the 80s will certainly recall this setup.

In BASIC, variable names are determined which line of code is run in sequence. For example, to print something to screen we'd enter:

```
10 print "Hello"
```

Once you've typed the above in, press Enter and then Enter.

**FUN**

We can use almost any text as the code to include variables, multi-line print statements and so on:

10 print "Hello, what's your name?"
20 input n
30 print n
40 print "I hope you're well today"

Type RUN to execute the code, you can also type LIST to view the code you've entered.

```
HELLO, WHAT'S YOUR NAME? DAVID!
HI DAVID, I HOPE YOU'RE WELL TODAY.
```

As you can see, variables are handled in much the same way as Python, a print statement on its own displays a blank line, and CLR clears the screen -- as used in conjunction with the OS module in Python and when running Windows. We're also able to do some maths, work, and play around with variables too:

10 print "How old are you?"
20 input a
30 if a > 40 then print "You're over 40 years old."
40 if a < 40 then print "You're under 40 years old."

As you notice, variables with addition (represent strings, nothing after the variable or a hash (#) represent strings, nothing after the variable), or a hash (#) the single quotes after the Print on line 30 indicate a blank line, one for each line, while REM on line 1 is a comment, and then ignored by the BASIC compiler.

Naturally, you can also change the task with variables with addition (represent strings, nothing after the variable), or a hash (#) the single quotes after the Print on line 30 indicate a blank line, one for each line, while REM on line 1 is a comment, and then ignored by the BASIC compiler.

```
Hello, What's your name? David
```

There are a number of ways you can visit to the BBC Micro to play around with some retro code. If you grew up with a Commodore 64, then you can always by VC6, the C64 emulator. Likewise, the 65 Spectre has a slew of great emulators available for every modern system to play around on. In fact, you can probably find an emulator for virtually every 8-bit or 16-bit machine that was produced over the years. Each has their own unique perspective and coding nuances, so find a few and see what you can create.
Text Animations

There's a remarkable amount you can do with some simple text and a little Python know-how. Combining what you've already learned, we can create some interesting animation effects from the command line.

THE FINAL COUNTDOWN

Let's begin with some example code that will display a large countdown from ten, then clear the screen and display a message. The code itself is quite simple, but lengthy. You will need to start by importing the OS and Time modules, then start creating functions that display the numbers:

```python
# Initialize settings
start = 10
message = 'READY> SATELLITE OFF!'<

# Start the countdown
i = 0
if counter in range(start, 0, -2):
    i += 1
    time.sleep[1]
    if os.system('clear') or os.name == 'nt' else 'clear'
```

And finally, we can add a display for the message:

```python
# Display the message
print('-------------------------------------------')
prompt(x='>')
prompt(message)
prompt(x='>')
prompt('=' + str(counter) + ' ' + prompt('[i]s)')
```

The code in its entirety can be viewed from within our Code Repository: [https://bممعارم.com/code-portal](https://bممعارم.com/code-portal). You are free to copy it to your own Python IDLE and use it as you see fit. The end effect is quite good and it'll be worth adding to your own games, or presentations, in Python.
To extend the code, or make it easier to use, you can always create the number functions in their own Python file, call it Count.py for example, then import Count at the beginning of a new Python file called Countdown.py, along with the OS and Time modules:

```python
import os
import time
import count

From there, you will need to specify the imported code in the Countdown section:

```python
@Start the countdown
for counter in range(start, 0, -1):
    if counter == 10:
        count.sec = 9:
    if counter == 9:
        count.min = 8:
    if counter == 8:
        count.am = 7:
    if counter == 7:
        count.second = 6:
    if counter == 6:
        count.am = 5:
    if counter == 5:
        count.sec = 4:
    if counter == 4:
        count.second = 3:
    if counter == 3:
        count.seconds = 2:
    if counter == 2:
        count.seconds = 1:
    if counter == 1:
        count.started
```

This will pull the functions from the imported Count.py and print them to the screen.

**ROCKET LAUNCH**

Building on the previous countdown example, we can create an animated rocket that’ll launch after the Blast Off! message has been printed. The code for this would look something like this:

```python
def Rocket():
distanceFromTop = 20
while True:
    os.system('cls') if os.name == 'nt' else 'clear'
    print('A' + distanceFromTop)
    print() + distanceFromTop
    print() + distanceFromTop
    print() + distanceFromTop
    time.sleep(0.2)
    os.system('cls') if os.name == 'nt' else 'clear'
    if distanceFromTop > 0:
        distanceFromTop -= 1
        distanceFromTop = 20

[Main Code Begins]
Rocket()
```

Here we’ve created a new function called Rocket, which produces the effect of an ASCII-like rocket, taking off and scrolling upwards, using the distance from top variable.

To use this, add it to the end of the previous countdown code and at the end of the Blast Off! message, add the following lines:

```python
print("Welcome")
input('Press Enter to launch rocket...
```

This will allow your message to be displayed and then, when the user has hit the Enter button, the rocket will launch.

Again, the code in its entirety can be found in the Code Repository at: https://adambuildings.com/code-portal.

**ROLLING DIE**

Aside from the rocket animation, together with its countdown, another fun bit of text-based animation is that of a rolling dice.

A rolling dice can be a great animation to include in an adventure game, where the player rolls to see what their score is compared to that of an enemy. The highest roller wins the round and the losers’ health drops as a result. It’s an age-old combat sequence, used mainly in the Dungeone and Dragons board games and Fighting Fantasy novels, but it works well.

The code you’ll need to animate a dice roll is:

```python
import os
import time
from random import randint

roll = ['1', '2', '3', '4', '5', '6']
die = [roll[0] * 5, roll[0] * 5, roll[0] * 5, roll[0] * 5, roll[0] * 5, roll[0] * 5]
die.append('0' * 5 + '0' * 5) die.append('0' * 5 + '0' * 5) die.append('0' * 5 + '0' * 5) die.append('0' * 5 + '0' * 5) die.append('0' * 5 + '0' * 5) die.append('0' * 5 + '0' * 5)

def roll():
    for roll in range(10):
        os.system('cls') if os.name == 'nt' else 'clear'
        print(''
number = randint(0, 6)
print(die[number])
time.sleep(0.5)
```

You may need to tweak the O entries, to line up the dots on the virtual dice. Once it’s done, though, you’ll be able to add this function to your adventure game code and call it whenever your character, or the situation, requires some element of luck, combat or chance roll of the dice.
Creating a Loading Screen

If you're looking to add a little something extra to your Python code, then consider including a loading screen. The loading screen is a short introduction, or piece of art, that appears before the main part of your code.

LOAD***

Back in the 80s, in the B-Dit home computing era, loading screens were often used to display the cover of a game as it loaded from tape. The image would load itself, usually one-line-at-a-time, then proceed to color itself in while the loading pattern bars danced around in the borders of the screen.

Loading screens were part of the package, and they stay in for the whole game as an experience. Some loading screens featured animations, or a countdown for time remaining as the game loads, while others even went so far as to include some kind of playable game. The point being: a loading screen is not just an artistic part of computing history, but an introduction to the program that's about to be run.

While these days loading screens may no longer be with us, in terms of modern gaming we can still include them in our own Python content. Either for fun, or to add a little retro-themed spice to the mix.

SCREENS

Creating a loading screen in Python is remarkably easy. You have several options to hand: you can create a window and display an image, followed by a brief pause, before starting your main code; or you could opt for a console-based ASCII art version, that loads one-line-at-a-time.

Let's look at the latter and see how it works.

First you'll need some ASCII art, you can look up plenty of examples online, or use an image to ASCII art converters to convert any images you have to ASCII when you have your ASCII art, drop it into a newly created folder inside another text file.

Save the file, call it screens.txt for now.

ADVENTURE TIME

A good example of using loading screen ASCII art, text images, is when coding a text adventure. Once you've established your story, created the characters, events and so on, you could easily incorporate some excellently designed ASCII art to your game.

Imagine coming across a dragon, in game, and displaying its representation as ASCII. You can then load up the image lines one-by-one, and continue with the rest of the adventure code. If it's certainly worth having a play around with and will definitely add a little something else extra.
THE CODE

Launch Python and enter the following code to a new file:

```python
import os
time

def loading_screen(seconds):
    if os.name == 'nt':
        print('Loading...')
        percent = (loading * 100) // seconds
        print(f'{percent}%')
        time.sleep(1)
    else:
        print(f'{percent}%loading...')
        percent = (loading * 100) // seconds
        print(f'{percent}%')
        time.sleep(1)

while True:
    os.system('cls')
```

The code is a simple loop that creates a loading screen and displays the percentage of completion. It differs from the original code by using the `os.system()` function to clear the screen on Windows and Linux. This makes the loading process more visually appealing and less prone to memory issues.

```python
loading_bar(seconds):
    if os.name == 'nt':
        print('Loading...')
        percent = (loading * 100) // seconds
        print(f'{percent}%')
        time.sleep(1)
    else:
        print(f'{percent}%loading...')
        percent = (loading * 100) // seconds
        print(f'{percent}%')
        time.sleep(1)

while True:
    os.system('cls')
```

COMBINING THE TWO

How about combining the two elements we've looked at? Let's begin with a loading... progress bar followed by the loading screen. After that, you can include your own code and continue your program. Here's the code:

```python
import os
time

def loading_bar(seconds):
    if os.name == 'nt':
        print('Loading...')
        percent = (loading * 100) // seconds
        print(f'{percent}%')
        time.sleep(1)
    else:
        print(f'{percent}%loading...')
        percent = (loading * 100) // seconds
        print(f'{percent}%')
        time.sleep(1)

while True:
    os.system('cls')
```

Another favourite introduction screen is that of a simple loading animation, where the word loading is displayed followed by some characters and a percentage of the program loaded. While it may not be as appealing as your Python code to load, the effect can be amusing.

Create a New File in Python and enter the following code:

```python
import os

def loading_bar(seconds):
    if os.name == 'nt':
        print('Loading...')
        percent = (loading * 100) // seconds
        print(f'{percent}%')
        time.sleep(1)
    else:
        print(f'{percent}%loading...')
        percent = (loading * 100) // seconds
        print(f'{percent}%')
        time.sleep(1)

while True:
    os.system('cls')
```

You can of course use these functions up anywhere and whenever you want. Just replace the code with anything else. You should at the beginning. Remember that a character by character loading screen is more fun than the ASCII text file in the same folder as the Python code, so the screenopen("screens.txt").
Tracking the ISS with Python

Of the many amazing human achievements over the past decade or so, the International Space Station tops the bill. This incredible collaboration between nations sees vital experiments carried out in space as well as observations of our own planet.

TO BOLDLY GO...

Indeed, the ISS is something most of us consider to be the perfect example of what can happen when we work together. NASA, among other agencies, uses a wealth of Python libraries to help automate routines, as well as act as an interface to translate code from one language to another, and then into a human-readable format. If you’re considering a career in space, then learning Python is a must-have skill.

While we’re not able to fit you in on all the details, regarding the code the ISS utilities, we can look at a fun Python script that will track the ISS, display the number of astronauts on board, update the station’s current latitude and longitude every five seconds, while also displaying your current latitude and longitude.

Displaying all that information in a single screen can become a little cluttered, so in this instance we’re going to look at spreading all those details over three screens: a text document window displaying the astronauts and your current latitude and longitude, a command line (or Terminal window) displaying the continuously updating latitude and longitude of the ISS as it orbits Earth, and a final, large window displaying a map of the world, together with an icon representing the ISS, that’s updated as it orbits. Interested? Read on.

THE GRAPHICS

Firstly, we need to get hold of a map of the world and an image of the ISS to use as an icon, that will be updated according to the position of the space station as it travels over the surface. A quick Google of World Map will help you out here. Look for one that’s reasonably large, the one we used for this example was 1280 x 700, and one that has the names of the countries — if you’re using this with younger people, to help with putting shapes of countries to names.

Next, look for an ISS icon. As this is going to be a graphical representation of the location of the ISS, we need the image to be reasonably small so it doesn’t drown out the locations on the map, but also prominent enough to see when the map is loaded. We opted for an image that’s 32 x 32 pixels in size. Don’t worry too much if you’re not able to find one that small, you can always resize it in an image editing app such as Paint or GIMP.

As we’re going to be using Turtle, a component of Python, the downloaded images will need to be converted to GIF, since this is the default and recommended image format. You can easily look up a convert online, but using Paint in Windows 10, or GIMP, which is cross platform, will suffice.
The CODE

The code we’re using here utilizes an open-source API (Application Programming Interface) to retrieve real-time data online regarding the status of the ISS. An API enables applications to communicate with one another by providing the raw data that a programmer can pull out and interact with in their own code. In this case, the API in question is a web-based collection of raw data that’s stored in a JSON (JavaScript Object Notation) format — an accessible and easy-to-read data interchange interface.

To use the ISS API, you’ll need to install the `requests` module in Python. If you haven’t installed it yet, you can do so by running the following command in your terminal:

```
python -m pip install requests
```

Once you have the `requests` module installed, you can use it to make API requests. In this case, we’ll be using the `requests.get()` method to make a GET request to the ISS API.

```
import requests

response = requests.get("http://api.open-notify.org/iss-now.json")
```

This will return a JSON response that contains information about the current position of the ISS. You can access this information using Python’s built-in `json` module.

```
import json

data = response.json()
```

Now you have access to the JSON data, which contains information about the current position of the ISS. You can access this information using Python’s built-in `json` module.

```
import json

data = response.json()
```

Let’s begin by breaking the code into bite-sized chunks:

```
import json, turtle, urllib.request, time
webbrowser.open("https://api.open-notify.org/iss-now.json")
```

This section will use the JSON and urllib modules to make a GET request to the ISS API and retrieve the current position of the ISS. You can use the `urllib.request.urlopen()` method to make a GET request to the ISS API.

```
response = urllib.request.urlopen("http://api.open-notify.org/iss-now.json")
```

This will return a JSON response that contains information about the current position of the ISS. You can access this information using Python’s built-in `json` module.

```
import json

data = response.json()
```

Now you have access to the JSON data, which contains information about the current position of the ISS. You can access this information using Python’s built-in `json` module.

```
import json

data = response.json()
```

This section will use the JSON and urllib modules to make a GET request to the ISS API and retrieve the current position of the ISS. You can use the `urllib.request.urlopen()` method to make a GET request to the ISS API.

```
response = urllib.request.urlopen("http://api.open-notify.org/iss-now.json")
```

This will return a JSON response that contains information about the current position of the ISS. You can access this information using Python’s built-in `json` module.

```
import json

data = response.json()
```

This section will use the JSON and urllib modules to make a GET request to the ISS API and retrieve the current position of the ISS. You can use the `urllib.request.urlopen()` method to make a GET request to the ISS API.

```
response = urllib.request.urlopen("http://api.open-notify.org/iss-now.json")
```

This will return a JSON response that contains information about the current position of the ISS. You can access this information using Python’s built-in `json` module.

```
import json

data = response.json()
```

This section will use the JSON and urllib modules to make a GET request to the ISS API and retrieve the current position of the ISS. You can use the `urllib.request.urlopen()` method to make a GET request to the ISS API.

```
response = urllib.request.urlopen("http://api.open-notify.org/iss-now.json")
```

This will return a JSON response that contains information about the current position of the ISS. You can access this information using Python’s built-in `json` module.

```
import json

data = response.json()
```

This section will use the JSON and urllib modules to make a GET request to the ISS API and retrieve the current position of the ISS. You can use the `urllib.request.urlopen()` method to make a GET request to the ISS API.

```
response = urllib.request.urlopen("http://api.open-notify.org/iss-now.json")
```

This will return a JSON response that contains information about the current position of the ISS. You can access this information using Python’s built-in `json` module.

```
import json

data = response.json()
```

This section will use the JSON and urllib modules to make a GET request to the ISS API and retrieve the current position of the ISS. You can use the `urllib.request.urlopen()` method to make a GET request to the ISS API.

```
response = urllib.request.urlopen("http://api.open-notify.org/iss-now.json")
```

This will return a JSON response that contains information about the current position of the ISS. You can access this information using Python’s built-in `json` module.

```
import json

data = response.json()
```

This section will use the JSON and urllib modules to make a GET request to the ISS API and retrieve the current position of the ISS. You can use the `urllib.request.urlopen()` method to make a GET request to the ISS API.

```
response = urllib.request.urlopen("http://api.open-notify.org/iss-now.json")
```

This will return a JSON response that contains information about the current position of the ISS. You can access this information using Python’s built-in `json` module.

```
import json

data = response.json()
```

This section will use the JSON and urllib modules to make a GET request to the ISS API and retrieve the current position of the ISS. You can use the `urllib.request.urlopen()` method to make a GET request to the ISS API.

```
response = urllib.request.urlopen("http://api.open-notify.org/iss-now.json")
```

This will return a JSON response that contains information about the current position of the ISS. You can access this information using Python’s built-in `json` module.

```
import json

data = response.json()
```

This section will use the JSON and urllib modules to make a GET request to the ISS API and retrieve the current position of the ISS. You can use the `urllib.request.urlopen()` method to make a GET request to the ISS API.

```
response = urllib.request.urlopen("http://api.open-notify.org/iss-now.json")
```

This will return a JSON response that contains information about the current position of the ISS. You can access this information using Python’s built-in `json` module.

```
import json

data = response.json()
```

This section will use the JSON and urllib modules to make a GET request to the ISS API and retrieve the current position of the ISS. You can use the `urllib.request.urlopen()` method to make a GET request to the ISS API.

```
response = urllib.request.urlopen("http://api.open-notify.org/iss-now.json")
```

This will return a JSON response that contains information about the current position of the ISS. You can access this information using Python’s built-in `json` module.

```
import json

data = response.json()
```

This section will use the JSON and urllib modules to make a GET request to the ISS API and retrieve the current position of the ISS. You can use the `urllib.request.urlopen()` method to make a GET request to the ISS API.

```
response = urllib.request.urlopen("http://api.open-notify.org/iss-now.json")
```

This will return a JSON response that contains information about the current position of the ISS. You can access this information using Python’s built-in `json` module.

```
import json

data = response.json()
```

This section will use the JSON and urllib modules to make a GET request to the ISS API and retrieve the current position of the ISS. You can use the `urllib.request.urlopen()` method to make a GET request to the ISS API.

```
response = urllib.request.urlopen("http://api.open-notify.org/iss-now.json")
```

This will return a JSON response that contains information about the current position of the ISS. You can access this information using Python’s built-in `json` module.

```
import json

data = response.json()
```

This section will use the JSON and urllib modules to make a GET request to the ISS API and retrieve the current position of the ISS. You can use the `urllib.request.urlopen()` method to make a GET request to the ISS API.

```
response = urllib.request.urlopen("http://api.open-notify.org/iss-now.json")
```

This will return a JSON response that contains information about the current position of the ISS. You can access this information using Python’s built-in `json` module.

```
import json

data = response.json()
```

This section will use the JSON and urllib modules to make a GET request to the ISS API and retrieve the current position of the ISS. You can use the `urllib.request.urlopen()` method to make a GET request to the ISS API.

```
response = urllib.request.urlopen("http://api.open-notify.org/iss-now.json")
```

This will return a JSON response that contains information about the current position of the ISS. You can access this information using Python’s built-in `json` module.

```
import json

data = response.json()
```

This section will use the JSON and urllib modules to make a GET request to the ISS API and retrieve the current position of the ISS. You can use the `urllib.request.urlopen()` method to make a GET request to the ISS API.

```
response = urllib.request.urlopen("http://api.open-notify.org/iss-now.json")
```

This will return a JSON response that contains information about the current position of the ISS. You can access this information using Python’s built-in `json` module.

```
import json

data = response.json()
```

This section will use the JSON and urllib modules to make a GET request to the ISS API and retrieve the current position of the ISS. You can use the `urllib.request.urlopen()` method to make a GET request to the ISS API.

```
response = urllib.request.urlopen("http://api.open-notify.org/iss-now.json")
```

This will return a JSON response that contains information about the current position of the ISS. You can access this information using Python’s built-in `json` module.

```
import json

data = response.json()
```

This section will use the JSON and urllib modules to make a GET request to the ISS API and retrieve the current position of the ISS. You can use the `urllib.request.urlopen()` method to make a GET request to the ISS API.

```
response = urllib.request.urlopen("http://api.open-notify.org/iss-now.json")
```

This will return a JSON response that contains information about the current position of the ISS. You can access this information using Python’s built-in `json` module.

```
import json

data = response.json()
```

This section will use the JSON and urllib modules to make a GET request to the ISS API and retrieve the current position of the ISS. You can use the `urllib.request.urlopen()` method to make a GET request to the ISS API.

```
response = urllib.request.urlopen("http://api.open-notify.org/iss-now.json")
```

This will return a JSON response that contains information about the current position of the ISS. You can access this information using Python’s built-in `json` module.

```
import json

data = response.json()
```
mark the boundaries of the screen, creating a 4x4 grid of the four corners of the screen, so that the GS icon can freely move across the map. The GS icon is set as the turtle pen shape, giving it an angle of 45 degrees when moving.

```python
while True:
    screen.current_status_of_the_ISC_in_real-time
    url = "http://api.open.notify.org/is económico"
    response = urllib.request.urlopen(url)
    result = json.loads(response.read())

    @��beh the ISC location
    location = result["has_position"]
    lon = locations["Longitude"]
    lat = locations["Latitude"]

    @fetching Latitude and Longitude to the console
    print(f"Lon = {lon}\nLat = {lat}\n Longitude = {lon}\nLatitude = {lat}\n")

    @refreshing the ISC location on the map
    map.variation(lat, lon)
    refresh every 5 seconds
    time.sleep(5)
``` 

Now for the final part of the code: Here we collect the location data from the 55-state API, pulling out the latitude and longitude of the ISC. We then print the latitude and longitude data to the console. Finally, we send the latitude and longitude as unstable lat/lon to upload the 55 icon on the map every five seconds.

Here's the code in its entirety:

```python
import json, turtle, urllib.request, time, webbrowser
import geocoder # need to pip install geocoder for your lat/lon to work.

@fetching the names of all the astronauts currently onboard the ISS
url = "http://api.open.notify.org/astro.json"
response = urllib.request.urlopen(url)
result = json.loads(response.read())

@names of the astronauts
@on board
@names = result["people"]

for p in people:
    p = p["name"]
    print("name = " + p["name"] + " on board = " + p["on_board"] + "")

# geoip(IP=123.45.67.89) # need to pip install geocoder, and import
screen = turtle.Screen() # show current lat/lon in the text file.
screen.setworldcoordinates(-180, -90, 180, 90)
screen.register_shape("gs.gif")
shape = "gs.gif"
setworldcoordinates(-180, -90, 180, 90)
```

Now we have the code to display the astronaut names and their locations on the map.
Tracking the ISS with Python

```python
import json
import turtle
import time

iss_gif = json.load(open('iss_gif.png'))

while True:
    # Get the current status of the ISS in real-time
    url = "http://api.open-notify.org/iss-now.json"
    result = json.loads(request.urlopen(url).read())

    # Extract the ISS location
    lat = result["iss_position"]
    lon = result["longitude"]

    # Plot the ISS location on the map
    iss_gif[0][0,0].plot(is_gif[0][0,0], lat, lon)
    time.sleep(5)
```

Create a new folder in your system, called ISSTrack. For example, you can place the two graphics, as well as the Python code itself.

RUNNING THE CODE

The code is best executed from the command line or "Terminal". Clear your terminal, enter your command line and navigate to where you've saved the code plus the two graphics. Launch the code with either python ISSTrack.py or Python3 ISSTrack.py (depending on your system), and what you've called the Python code.

The code will launch two extra windows together with the command line window you already have open. One will be the first file containing the named astronauts along with your current latitude and longitude, and the other will be the world map with the ISS icon located wherever the ISS is currently orbiting. The command line window will start scrolling through the changing latitude and longitude of the ISS.
Using Text Files for Animation

Animation in Python can be handled with the likes of the Tkinter and Pygame modules, however, there's more than one way to achieve a decent end result. Using some clever text file reading code, we can create command-line animations.

**ASCII ANIMATION**

Let's assume you wanted to create an animated ASCII Happy Birthday Python script, with the words Happy and Birthday alternating in appearance. Here's how it's done.

**STEP 1**
First, we need to create some ASCII-like text. Head over to http://astarrek.com/software/text. This is an online text to ASCII generator, created by Patrick Gillespie. Start by entering `Happy` into the text box; the result will be displayed in the main window. You can change the font with the drop-down menu, or, to the side of the text box, we've opted for Big.

**STEP 2**
Now create a folder in your Python code directory on your computer (call it `Test`, for now), and open either Notepad for a Windows, TextEdit for a Mac, or, if you're using Linux, then the currently installed text editor. Click on the Select & Copy button at the bottom of the ASCII Generator webpage, and paste the contents into the text editor.

**STEP 3**
Save the text file as `t.txt`; you can call it whatever you like, but now for ease of use `t.txt` will suffice. Save the file in the newly created `Test` folder. When it's saved, copy exactly the same for the word `Birthday`. You can select a new font from the ASCII Generator, or add extra characters and when you're ready, save the file as `b.txt`.

**STEP 4**
Open up Python and create a New File. We're going to need to import the OS and Time modules for this example, followed by a line to clear the screen of any content. If you're using Windows, then you'll use the `CLS` command, whereas if it's `Clear` for Linux, we can create a simple if/else statement to handle the command.
**Using Text Files for Animation**

**STEP 5** Next we need to create a list of the names of the text files we want to open, and then we need to open them for display in the Terminal.

```python
filenames = ["1.txt", "2.txt"]
frames = []
for name in filenames:
    with open(name, "r", encoding="utf8") as f:
        frames.append(f.readlines())
```

**STEP 6** We’ve used the UTF-8 standard when opening the text files, as ASCII art as text, within a text file, often requires you to save the file as UTF-compliant – due to the characters used. Now we can add a loop to display the files as 1.0D, then 1.1D, creating the illusion of animation while clearing the screen after each file is displayed.

```python
for i in range(100):
    for frame in frames:
        print(*frame, flush=True)
    time.sleep(0.1)
os.system("cls") if os.name == "nt" else "clear"
```

**STEP 7** Note from the loop within the code, we’ve used the same CLS and Clear statement as before. Again, if you’re running on Windows then the OS module will use the CLS command, "CLS". If you’re using Linux or a Mac, the Clear command will work correctly. If you want, you could use a try/except statement instead.

**STEP 8** Here’s the code in full:

```python
import os, time
os.system("cls") if os.name == "nt" else "clear"
filenames = ["1.txt", "2.txt"]
frames = []
for name in filenames:
    with open(name, "r", encoding="utf8") as f:
        frames.append(f.readlines())
for i in range(100):
    for frame in frames:
        print(*frame, flush=True)
    time.sleep(0.1)
    os.system("cls") if os.name == "nt" else "clear"
```

**STEP 9** Save the Python code in the same folder as the text files and drop into a Terminal or Command Prompt. Navigate to the folder in question, and enter the command:

```
python NAME.py
```

Where NAME is whatever you called your saved Python code.

**STEP 10** You can spice things up a little by adding system/terminal colours. You’ll need to Google the system codes for the colours you want. The code in our example turns the Windows Command Line to green text on a black background, then changes it back to white on black at the end of the code. Either way, it’s a fun addition to your Python code.
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 recognize 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 computer 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, player? and so on.

//COMMENTS

Use comments. It’s a simple concept but commenting on your code saves you 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.

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: 123
You have entered wrong input
You have entered wrong input
123 is not a valid integer.
You have entered wrong input
123 is not a valid integer.
You have entered wrong input
123 is not a valid integer.
You have entered wrong input
The number is: 12
Process returned 0 (Normal exit) execution time: 0.009990s Press any key to continue.
```

**BACKUPS**

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

![Backup Image](data-backup.png)

**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, whilst it’s equally rewarding to solve the problem yourself, often the same code is out there on the internet somewhere. Don’t try and reinvent the wheel, look to see if someone else has done it first.

![Wheels Image](re-inventing-wheels.png)

**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 clear text. 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.

![Security Image](secure-data.png)

**HELP!**

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

![Help Image](help.png)

**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.

![Maths Image](maths.png)
Python Beginner’s Mistakes

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

Def Beginner(Mistakes=10)

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

Versions

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

Indents, Tabs and Spaces

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

The Internet

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

Commenting

Again, mention commenting, it’s a hugely important factor in programming, even if you’re the only one who is ever going to view the code, you need to add comments so to what’s going on. In this function where you lose a file, write a comment and help you, or anyone else, see what’s going on.
COUNTING LOOPS

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

```python
def main():
    n = list(range(1, 11))
```

which will return 1 to 10.

CASE SENSITIVE

Python is a case-sensitive programming language, so you will need to check any variables you assign. For example, `Lives18` is a different variable to `Lives18`, calling the wrong variable in your code can have unexpected results.

```python
>>> Lives18 = 10
>>> Lives9 = 10
>>> print(Lives18, Lives9)
10 9
```

OPERATORS

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

```python
b = 1
a = 10
b == a # a failure because 1 is not equal to 10
a == b # a failure because 10 is equal to 10
```

OPERATING SYSTEMS

Writing code for multiple platforms is difficult, especially when you start to utilise the external commands of the operating system. For example, if your code calls the screen to be cleared, then for Windows you would use `CLS`. Whereas, for Linux you need to use `clear`.

```python
debug = True
for letter in self.guessed_letters:
    print(letter)
```

```python
# Code to detect error for using a different OS
print("Python 3 file Manager")
```
**C++ Beginner’s Mistakes**

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

**VOID(C++, MISTAKES)**

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

**UNDECLARED IDENTIFIERS**

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

**STD NAMESPACE**

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

```cpp
using namespace std;
```

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

**SEMICOLONS**

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

**GCC OR G++**

If you're compiling in Linux then you will no doubt come across g++ and g++. In short, gcc is the GNU Compiler Collection (or GNU C Compiler as it used to be called) and g++ is the GNU++ (the C++ version) of the compiler. If you're compiling C++ then you need to use g++, as the incorrect compiler drivers will be used.
COMMENTS (AGAIN)
Indeed the mistake of never having any comments on a code is back once more. As we've previously mentioned, the lack of readable identifiers throughout the code makes it very difficult to look back at how it worked, for both you and someone else. Use more comments.

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

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

```cpp
#include
if (lines
switch (lines
If it sounds confusing don’t worry, the compiler lets you know where you went wrong.
```

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

INITIALISE VARIABLES
In C++ variables aren’t initialised to zero by default. This means if you create a variable called x then, potentially, it is given a random number from 0 to 16,446,744,073,709,551,516, which can be difficult to include in an application. When creating a variable, give it the value of zero to begin with: int x;

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

```
 gcc code.cpp
```

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

```
 gcc code.cpp -o nameOfProgram
```

www.it revolution.com
Black Dog Media

Master Your Tech
From Beginner to Expert

To continue learning more about Raspberry Pi & Coding visit us at:

www.bdmpublications.com

FREE Tech Guides
Raspberry Pi 4
Coding Guides, Tips & Tricks
Programming with Python
Learn & Master C++
Testing Linux Distros

EXCLUSIVE Offers on Tech Guidebooks
- Print & digital editions
- Featuring the latest updates
- Step-by-step tutorials & guides
- Created by BDM experts

PLUS SPECIAL DEALS and Bonus Content When you sign up to our monthly newsletter!
Discover more of our complete manuals on Readly today...
Learn Coding: C++ & Python

Learning to code is one of the most useful skills you can adopt in this ever-changing digital world. Discovering how code influences, drives, and controls every aspect of computing can lead to improved job prospects, mastery of IT, and a better understanding of how our devices communicate with the wider world.

Learn Coding: C++ & Python will take you through the initial stages of learning how to code with two of the most utilised, and powerful, programming languages available today. Python is a perfect beginner’s language: easy to learn, while being effective and efficient. C++ is the powerhouse of programming and with it, you can create great things.

Join us, then, and let’s start learning how to code.

Python
Python can be found in the background of websites, helping to control everything from the Internet, through to NASA, on-board the ISS. It’s a simple, but extremely effective, programming language and you can learn the basics in a matter of hours. Master Python, and you hold the future of AI, Big Data, and the Internet in your hands.

Python Power
Within, you will learn how to use the core selection of Python’s many commands and routines to create your own code that you can expand and improve on for use in the real world. We look at using variables, user input, loops, Python Modules, and even Python graphics.

C++
Windows, macOS, Linux, PlayStation and Xbox games, they all have one thing in common: at the heart of each lies C++ code. C++ is powerful, fast, and can control some of the more complex elements of technology. It’s little wonder that C++ is one of the most sought-after programming languages in the job market. Start your C++ adventure here, and see where it leads.

Working with C++
Discover how to install a C++ IDE and compiler, how to start coding in C++, and get to grips with elements such as strings, data types, user interaction, file input and output, loops and decision making concepts. This book will help you with your first steps into C++.

Keep on Coding
We’ve got some great coding projects for you to get your teeth into. Including code that can pass variables from the command line to Python, animations using ASCII art, code to track the International Space Station in real-time, and even some retro coding for a bit of old school fun, there’s something for everyone in Learn Coding: C++ & Python.