RASPBERRY PI EDITION

PYTHON UNLEASHED

BRUCE SMITH

NOVICE TO NINJA

CONTENTS

IMPORTANT NOTE

Python programs follow a style guideline known as PEP8, which will be covered later in this book. Although PEP8 helps maintain consistency, it is not required for a program to run correctly. One of the guidelines suggests including two blank lines between certain sections of code, but this will not always be followed in the listings here. This is to reduce whitespace.

PROGRAMS AND QUESTIONS

There are over 250 programs in this book. I suggest that you type in as many of these as you can, rather than loading them in from the files. Typing is the best way to learn and get a feeling for the commands, syntax and flow of a Python program. Learn how to correct and ensure your coding is correct and works. Bear in mind that some program lines extend longer than the space available across the width of the book. As such they will often 'wrap-around' onto the next line. That said the programs are available to download from the website at:

xxxxxxxxxxxx

QUESTIONS AND EXERCISES

Most chapters, not all, contain questions at the end. These questions are related to the text in the chapter just read. There are up to 15 questions. No answers are provided in the book, they are there for you to answer or review the contents of the chapter prior to answering. Exercises include answers, but remember my solution may be different to yours, but as long as you get the right result that's it! You'll also find more exercises on the downloads file.

00: Novice to Ninja

What does **novice** and **ninja** mean regarding learning to program Python on the Raspberry Pi? There are many books for programmers who are starting their Python experience. But they cover the basics and **don't** actually get under the 'hood' and into the detail. They leave you in the hallway, and don't show the rest of the house. Within these pages I seek to correct that and take you to those places beyond the hallway, providing a higher level of knowledge and expertise to transform you from a complete beginner (or intermediate), who's just learning the ropes, to a skilled and confident Python coder. Merriam-Webster defines each word thus:

- A **novice** in computer terms can be defined as a beginner or someone who has no previous experience in a particular field or activity.
- On the other hand, **ninja** isn't typically defined in the same way in dictionaries. In popular usage, it's often used to describe someone who has achieved a high level of skill or expertise. An expert, maybe someone to be feared?

You may not be a novice in use of the Raspberry Pi, and may have some programming knowledge, but this tome will just accelerate your learning. Being a '**Python ninja**' doesn't mean you've mastered every single aspect of the language, after all, there's always more to learn. Instead, it means you've reached a point where you're comfortable and efficient with the language, can solve problems creatively, and write clean, effective code. You know how to handle different challenges, think like a programmer, and confidently create your own projects or collaborate with others.

Programs

There are a **lot** of programs in this book. You can download the source from the authors website .

They are there for your convenience. andI would **strongly** suggest that you **type these programs in yourself.** Unless you do you won't start to understand how a Python program goes together. How it is structured and what goes where as well as why? Most of demo programs are not that long, so it shouldn't be overly difficult. As you progress through the book then they will become much longer, so you could delve into the download at that point.

Typing is another key skill. If you can't type, to whatever degree, in today's world then things become long-winded. It is the biggest skill you can master and is starting to become part of the school curriculum here.

I'm always pleased to hear from readers, please feel free to contact me at:

`feedback@brucesmith.info`

Use the Website Please

If you didn't buy this through **The Coding Press** website, I'd be grateful if you consider purchasing any more of my books from there. You will find a larger choice of book formats, and help support me directly as a creative. I am one of a few independent publishers and certainly the only one writing seriously about the Raspberry Pi.

This means I can fully reap the benefits of my efforts in writing and publishing. Supporting creators directly allows us to continue producing more great content. People often think of the cost of the final project but not the six plus months or so it takes to go from first word to finish article!

Raspberry Pi Versions

The contents of this book have been tested on the Raspberry Pi 3, 4, 400 and 5, 500 and any future Pi releases running Python 3. There are some differences in the application here and there, but it is remarkably consistent across all these versions. Where differences occur, and these relate mainly to implementation issues, then I have noted this and provided the **relevant** information for each model. In general, this book should work for any Raspberry Pi that can run Python 3. Below is a guide to how Python 3 reacts to being run on various Raspberry Pi models.

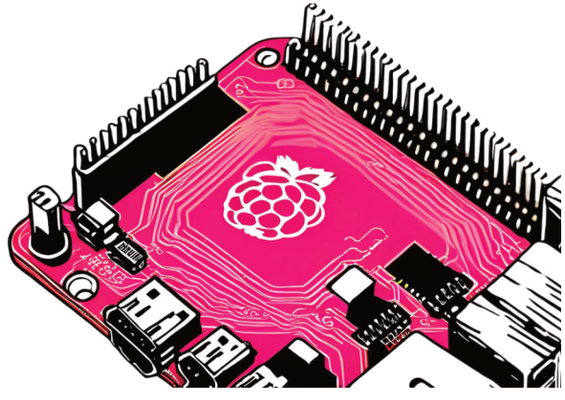
Raspberry Pi 1 (2012): Python 3 can run on this model, but due to its limited processing power and memory (512 MB RAM), it may not perform well for more complex programs, such as graphics and use of high intensity modules such as NumPy.

Raspberry Pi 2 (2015): Python 3 runs smoothly on this Raspberry Pi. It features more processing grunt (Quad-core 900 MHz CPU) and 1 GB of RAM, allowing for better performance with Python applications.

Raspberry Pi 3 (2016): Python 3 runs very well on the Raspberry Pi 3,

which has a Quad-core 1.2 GHz CPU and 1 GB RAM, making it suitable for more demanding Python applications, including GUI-based ones.

Raspberry Pi 3B+ (2018): Python 3 runs smoothly on this model, which has improved performance over the Pi 3 supporting a 1.4 GHz CPU.



Raspberry Pi 4 (2019): Python 3 comes into its own on the Pi 4. This model offers 8 GB of RAM, making it ideal for larger Python programs, and can handle graphics with ease.

Raspberry Pi 400 (2020): Python 3 works well on the 400. This model is a keyboard-integrated Raspberry Pi 4 with a 1.8 GHz CPU. It is effectively a 4 inside a keyboard!

Raspberry Pi 5/500 (2024) Python 3 is impressive on the 5. This model offers up to 8 GB of RAM, making it ideal for memory intensive programs.

Raspberry Pi Zero and Zero W (2015 and 2017): Python 3 runs on the Raspberry Pi Zero, but given the limited processing power (single-core 1 GHz CPU) and RAM (512 MB), performance may be slow.

Raspberry Pi Zero 2 W (2021): Python 3 runs on this model. It has improved performance over the original Zero with a Quad-core CPU and 512 MB RAM, making it more capable for Python tasks.

Raspberry Pi Pico: This **i**s a different type of device compared to the standard Raspberry Pi models. It is a **microcontroller**, not a single-board computer, so it doesn't run a full operating system like Raspberry Pi OS. Instead, it runs programs directly on the hardware but uses a derivative of Python called MicroPython. This is discussed in Chapter 45.

Operating Systems: All Raspberry Pi versions running Raspberry Pi OS (formerly Raspbian and updated to Bookworm from Raspberry Pi 5), which

come pre-installed with Python 3, will support Python 3 out of the box. Raspberry Pi OS maintains support for Python 3.x.

Python Types

Technically, there is only one Python programming language, but there are multiple implementations of Python that cater to different needs and environments. The most common Python implementations are:

Python: The standard implementation of Python edition and the one we are learning herein.

CPython: The reference implementation of Python, written in the C programming language. CPython = Python (the language) + C-based implementation.

Jython: Python implementation written in Java. Useful when you need to integrate Python with Java programs or Java-based frameworks.

PyPy: A Python implementation focused on speed, written in RPython (a restricted subset of Python). PyPy is discussed in Chapter 44.

IronPython: A Python implementation targeting the .NET framework and Mono (cross-platform implementation of .NET.)

MicroPython: A lean and efficient Python implementation designed to run on microcontrollers and small embedded systems.

Stackless Python: A Python implementation based on CPython but with added support for microthreads.

Brython: A Python implementation that runs entirely in the browser, converting Python code into JavaScript.

01: Hello

Python, alongside JavaScript and Java, is one of the most widely used programming languages in the world. Some might even say it's the most popular and significant, especially in the business world where it's the go-to software. If you're aiming to become a commercial programmer and haven't yet explored Python, you might find fewer doors open—**it's that essential**.

As a taster, look at these famous organisations and their uses of Python:

YouTube: the world's largest video-sharing platform, extensively uses Python for back-end services, including video sharing, website operation, and system administration. Python is known for being simple and easy to maintain, making it ideal for a platform like YouTube, which requires handling massive amounts of data and user interactions efficiently. Its strong libraries for web development, and support for data handling allow YouTube engineers to scale the platform easily.

Instagram: one of the most popular social media platforms, relies heavily on Python and its modules, for handling millions of active users and managing its back-end services. Instagram chose Python for its simplicity and ability to help developers write clean, maintainable code. It also helps Instagram scale its infrastructure efficiently. Python's scalability and speed in development cycles allowed Instagram to keep up with its explosive growth without compromising performance.

Spotify: the popular music streaming service uses Python for data analysis, back-end services, and machine learning to provide personalised recommendations. Python excels at handling large amounts of data, which Spotify needs for features like personalised music recommendations and user behaviour analysis. Its data science libraries for analytic and machine learning tasks. Additionally, Python's asynchronous framework capabilities, like Tornado and asyncio, enable Spotify to handle multiple concurrent connections (such as streaming requests) efficiently.

Reddit: is one of the largest online communities, is primarily written in Python. It uses Python for its back-end to manage user submissions, interactions, and content. Reddit originally started with Lisp but migrated to Python for its simplicity and wide range of libraries. Python allows Reddit to scale easily, handle millions of daily interactions, and manage a large amount of content without sacrificing performance. Python's versatility and Reddit's use of frameworks enable it to support its massive user base while remaining flexible for future growth. Reddit's decision to use Python also makes it easier to maintain and add new features over time.

Google: has used Python since its early days, and it plays a significant role in various parts of Google's infrastructure, including search algorithms, system management tools, and back-end services. Google values Python for its simplicity, speed of development, and readability. These attributes allow developers to write and maintain code quicker, which is crucial in a large-scale environment like Google. Python's flexibility also allows it to be used in everything from system administration to machine learning. For example, Google's internal systems (like parts of Google Search) and tools like YouTube Data API rely on Python. In fact, Guido van Rossum, the creator of Python, worked at Google for several years, and Google actively supports Python's development.

Netflix: utilises Python for content delivery, data analytic, and automation, playing a critical role in its recommendation algorithms and internal systems. Netflix uses Python for data streaming and analysis to track user preferences and optimise content recommendations. Python's powerful libraries, such as NumPy, Pandas, and TensorFlow. Additionally, Python helps Netflix automate content delivery and infrastructure management, making their systems more efficient and scalable.

NASA uses Python in various scientific computing and space research applications, including data analysis and simulations. Python's extensive scientific libraries (like SciPy and NumPy) and its ease of integration with other technologies make it ideal for complex scientific tasks. Python is widely used in scientific research because of its readability and vast array of scientific libraries. Python's ease of integration with other languages (such as C or Fortran for performance-critical code) also makes it an ideal choice for NASA, where various specialised tools and systems need to work together seamlessly.

Uber: uses Python for back-end services and data science tasks, helping manage its large-scale ride-sharing operations. Python's ease of use and ability to handle large-scale, real-time data processing makes it a natural fit for Uber's, fast-paced environment. Uber processes millions of ride requests, driver updates, and trip calculations in real-time, and Python's capabilities help manage these concurrent processes efficiently. Python also plays a

critical role in Uber's data science efforts, where it's used for calculating estimated time of arrival (ETA), optimising routes, and pricing algorithms.

Pi Space

No wonder, then, that Python has found a cosy spot on the Raspberry Pi; it comes bundled with your Raspberry Pi OS installation at no extra charge!

So, what makes Python so special? It's known as a high-level language, which means it's designed with us humans in mind—**easy to read and write**. High-level languages are user-friendly and more abstract compared to low-level languages. Programmers love them because the code is easy to understand and maintain. Fun fact: another high-level language, C, was used to create Python itself, which is why Python's official name is C Python.

And no, the name has nothing to do with snakes. Python is named after a cult 70s British comedy show, *Monty Python's Flying Circus*. Remember John Cleese's 'Ministry of Silly Walks'? Classic! (If you haven't seen it, give it a search on YouTube.) There are more nods to the show sprinkled throughout the language.

At first glance, Python code might look a bit intimidating. Don't let that fool you. Despite its appearance, it's all about **readability**. Python emphasises 'structured programming,' nudging you to write clean and tidy code. It's like the language itself is helping you craft perfect programs. Structure is key in every aspect of life, so why should a programming language be any different?

One of Python's biggest charms is the availability of ready-made building blocks for writing programs, kind of like assembling a house from bricks. Think of these bricks as the building blocks of the language. In Python terms, they're called **libraries** and **modules**. They do exactly what they say on the tin—libraries of code with specific functions, and modules that provide exactly what you need.

The best part? You don't need to create each step from scratch. You simply pick the blocks you need and snap them together. Python's popularity ensures there are plenty of these resources available, all designed to make your coding life easier.

Python's syntax is simple and readable, making it a breeze for beginners. The way Python uses **line indentation** to define code blocks makes it easy to spot different parts of a program at a glance. Plus, Python allows you to execute commands and segments of code 'on the fly' thus allowing you see results immediately when typing commands at the prompt. You can almost test your code on the fly.

And let's not forget about the massive **'standard library'** that comes with Python—a treasure trove of pre-written code. You don't have to worry much

about getting access to these tools because the standardised interface between them makes it super easy.

The standard library itself is ever **expanding**, or added to in the case of Bookworm on the Raspberry Pi 5. Modules such as NumPy that have normally had to be installed, is now part of the library, so well worth checking if its installed first.

This not only saves you time but also keeps your programs lightweight by only using what you need. Python's **cross-platform compatibility** means your programs can run smoothly on various operating systems like Windows, macOS, and Linux. Write it once, use it many times.

So, as we move forward, the code we write on the Raspberry Pi will be, for the most part, transferable to other environments. How great is that?

Python Uses

On the first page of this chapter, I outlined just some of the large multi-nationals who make use of Python everyday, indeed you could say it underpins a large chunk of their business functionality.

Python plays a pivotal role in the educational landscape of the Raspberry Pi, offering versatility beyond just coding for learning. Some of the remarkable applications of Python on the Raspberry Pi include:

Home Automation: Python is a go-to for automating and controlling smart home devices, enabling users to script interactions with sensors, cameras, lights, and more.

Making: With an extensive array of add-on like hats, robots, displays, and weather monitors, Python's libraries control these attachments seamlessly.

Web Development: Python serves as a capable tool for crafting web applications, making it ideal for web-based projects on the Raspberry Pi.

Game Development: Crafting simple games using Python and libraries like PyGame on the Raspberry Pi provides an enjoyable introduction to programming and game development.

IoT (Internet of Things): Python finds its niche in IoT projects, connecting sensors, actuators, and other IoT devices to the Raspberry Pi, facilitating communication with cloud services.

Data Science and Analytics: Many of the Python libraries support data science, machine learning, and analytics on the Raspberry Pi, empowering users to analyse data and run machine learning models.

Robotics: Python's prowess extends to programming robots and robotic

systems on the Raspberry Pi, with add-on modules like 'GPIO Zero' simplifying hardware control.

Network Programming: Python's networking capabilities make it apt for projects involving device communication over a network, such as building a networked media centre or a file server.

Security and Penetration Testing: Python is an asset for security-related tasks, offering tools and libraries for penetration testing and network security on the Raspberry Pi.

Python on the Raspberry Pi is not just a programming language; it's a gateway to a multitude of exciting possibilities across various domains. The applications are endless.

Home Help

I personally use Python for a lot of things at home. It's so easy to use. Anything that involves, sorting, figures, text etc. For example, I use it for pulling together all my account data for the end of year tax return.

For this book, I typeset it using an application called *Affinity Publisher*. I created a program that extracts all the programs in the text and then test each one of them. Any errors are logged and changes can be made. This would otherwise be a time consuming copy and paste process.

It's up to you to come up with the ideas…

Learning

So, how do you go about learning Python? Well if you have this book you're well on your way. Given that fact there are some things to help:

Set Clear Goals

Decide why you want to learn Python: Do you want to use it for web development, data analysis, automation, or game development? Or do you just want to **learn**? Knowing your end goal will help guide your learning path.

Small Steps. For example, aim to write a small Python script within the first week, then move to more complex projects over time.

Get the Basics Right

Understand fundamental programming concepts. It's important, especially for Python. Keep an open mind and ensure you understand one chapter before jumping to the next one. Answer any questions and try a programming example. Get though correct. Then move on. Take existing programs and rework them for something you need.

Modify existing code: Try tweaking open-source code to see how things work. It helps you understand how small changes affect the program.

Practice, Practice, Practice

Consistent coding is the key. Even 20–30 minutes a day will build your skills faster than cramming once a week.

Stay Curious

Be open to learning new things as Python is vast and versatile. Try solving problems. Get online and check out the Python communities and forums. Keep in touch with the Raspberry Pi Python community.

Python 32-bit or 64-bit?

Python will run on either version of the ARM microprocessor. This is due to something called the Python Virtual Machine, which we'll look at later. So you are covered either way. There are some speed advantages with using A64 especially dealing with large sets of data.

02: Python Interactive

Open the Terminal window (that's the black box in the top left of your screen with '>_' inside it). Once you've got that open, type:

```
python3
```

And press the **<Enter>** key.

The Python Interpreter interactive command line will materialise, and include details such as the version of Python in use. You should see a prompt that looks like this:

```
>>>
```

This confirms that you're in the right spot. Just to be sure you're not mixing it up, the standard Terminal prompt is '>_'. Similar, but different enough to keep you on your toes! Note that the standard Terminal prompt will normally include details of your log-in such as:

```
pi@raspberrypi:~$
```

or similar

Now that you're in the **Python Interactive Shell** (no need to abbreviate here), you're ready to type in and execute Python code on the fly. Go ahead and type:

```
print("G'day mate")
```

Then press **<Enter>**. The code will run immediately, and the output will pop up on the line below. (Any guesses on what it might be?) I'm not Australian but living down under makes **"G'day mate"** a common term!

While the Python command line is super handy for quick tests and playing around with code, there's an even better way to work with Python as we shall see shortly.

To exit the interactive shell, just type:

```
exit()
```

at the **>>>** prompt, and hit the <Enter> key again.

You'll be whisked back to the Command Line of the Terminal, still in the same window.

Checking In

Some Raspberry Pi setups come with two versions of Python installed. On my Raspberry Pi 4B, when I typed:

```
python --version
```

(Note: That's **two** hyphens.)

It returned:

```
Python 2.7.2
```

However, when I typed:

```
python3 --version
```

It produced:

```
Python 3.7.3
```

Depending on your Raspberry Pi, you might see the same version for both commands, and it is probably different from mine. Python is constantly being updated, so you may have a newer version. Using **python3** ensures that version 3.x.x—the latest version on your Raspberry Pi—is used.

For those running the Bookworm version of the OS, you might only have one version of Python installed. Python 3.

This book focuses on Python version 3 and above. So, if you happen to have multiple versions installed, just remember to use Python 3, and I'll show you how to make sure of that shortly.

To ensure you have the latest version of Python, at the Terminal prompt type:

```
sudo apt update
```


and press <Enter>. Let everything update, and if it asks you anything, just reply with 'Y'.

When the cursor returns, type:

```
sudo apt full-upgrade
```

This command updates all the software 'packages' on your system, making sure they're the latest versions available for your OS, including those all-important Python packages. (You can check for any changes afterwards using the '**--version**' command.) Note that version numbers after 3.x can change quickly.

```
sudo apt install python3
```

Would ensure that Python3 was installed.

Don't worry if you don't have the absolute latest version of Python. Like I mentioned earlier, it's horses for courses. The version you have is almost certainly the right one for your version of the Raspberry Pi OS. Older versions of the Pi often run older versions of Python—makes sense, right?

Equally, as you upgrade your installation as the Raspberry Pi often suggest you do, you may well get an update on the version of Python you're using.

IDE

An **Integrated Development Environment**—what we in programming speak call an IDE—is pure bliss, like the best thing since **apple pie** and custard. So, let's get set up for some coding comfort. I recommend creating a directory in your Home folder. Maybe give it a name like 'PYTHON'. The name doesn't really matter, but something specific will help keep your coding projects organised. If you've downloaded the program files from my website, pop them into this folder.

Python programs are simply text files, a bunch of statements that, when combined just right, perform tasks. We group these statements into sets, each set handling a specific job. Bundle them together, and bingo—you've got a program.

The IDE is the hero of our coding adventure. Think of it as a magical space where you can create, run, and fix your programs, all within one window.

For this journey, our IDE of choice is 'Thonny', which is just perfect for beginners. You can find it under the Raspberry menu, nestled in the Programming drop-down menu. Once you open it, a screen like that shown in Figure 2a greet you.

(Don't worry if your screen looks a bit different. Looks and layout can vary slightly from version to version. I've put the numbers there for descriptive purposes!)

Maximise the Thonny window to fill your screen if you like, or just keep it as a normal floating window that you can resize as needed. The choice is yours. Thonny's interface is all about simplicity, designed to keep things smooth and easy as you write and run your Python code. While it might not be the top pick for larger projects, it works perfectly for our purposes..

Let's break down the Thonny window into three handy mini windows, each with its own job. I've numbered them 1, 2, and 3 for easy reference:

1. **Script Editor**: This is where the magic happens. Write your Python code here. It's clean, it's spacious, and it's your coding canvas.
2. **Shell**: Just below the Script Editor is the Interactive Python Shell. This is your playground for testing snippets of code on the fly, like what we did in the command line earlier. We call this a REPL (Read-Eval-Print Loop).
3. **Assistant**: On the right, this window gives you feedback about your program when it runs. If there's an error, it'll offer some hints on what might need fixing.

You can resize any of these windows individually by dragging their borders.

Across the very top of the window (Figure 2b), you'll find the menus as drop-downs and icons. These are laid out in a standard format, so they should feel familiar. Most are intuitive, but a few might make you pause—no worries, we'll explore their purposes as we go along. If you hover your pointer over the icons, a tool tip will pop up explaining what they do.

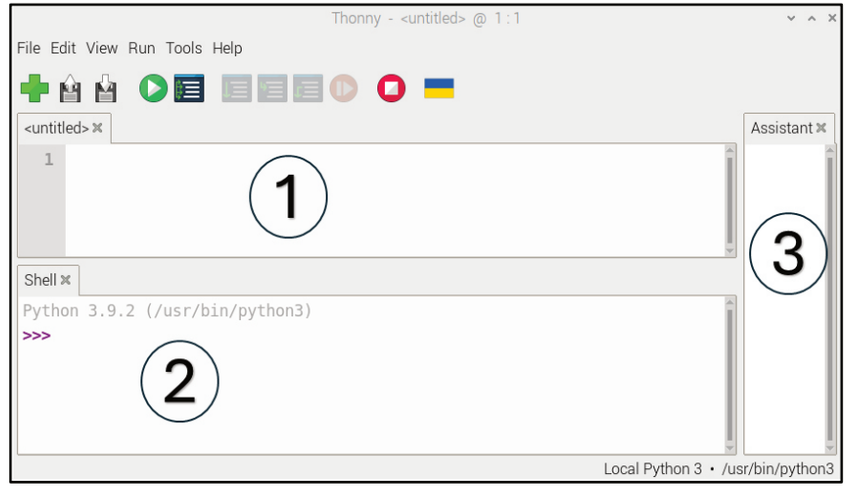


Figure 2a. Typical Thonny start-up screen.

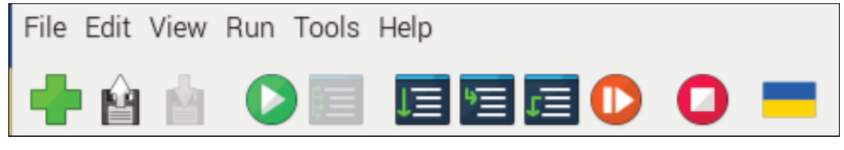


Figure 2b The Thonny Menu Bar.

Now, if you glance at the bottom right-hand side of the Thonny window, you'll see something like this:

```
/usr/bin/python3
```

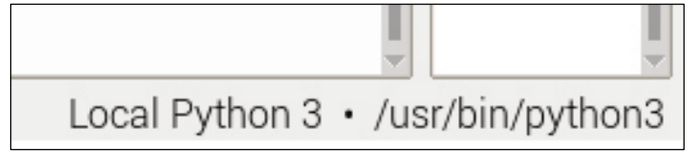


Figure 2c. Python version being used by Thonny, and its location.

This shows the version of Python that Thonny is using. The address following the version points to the location of the system version of Python.

Now, you're all set. The interface of Thonny. is a model of clarity, uncluttered and honed to the essential elements for Python coding. Thonny's user-friendly design boasts simplicity to ensure smooth writing and running of your programs. While it might not be the go-to IDE for larger, more complex programs, those are likely beyond the scope of this book.

Trying It Out

In Thonny, you can either use the existing program window (‘1’ above) or create a new one by clicking the green ‘+’ icon. You can also go to "File" in the menu bar and select ‘New’, or simply press **<Ctrl+N>** (meaning press the ‘Ctrl’ and ‘N’ keys together). There are so many ways to start—just pick one and get going! This action will open a new tab in the code area, and you can switch between tabs by clicking the one you need.

Type the following Python code in the editor:

```
print("Hello from Thonny!")
```

Save the file by clicking on ‘File’ and selecting ‘Save’, or by pressing <Ctrl+S>. Choose a filename and location for your Python program, making sure to give it a **.py** extension. For example, you could save it as: **hello.py**

You’ll see the name and directory of the file in the Thonny bar at the very top of the window.

To run the program, click on the green ‘Run’ button in the toolbar, select ‘Run’ from the menu bar, or simply press the ‘F5’ key as a shortcut. I’ve put a small yellow sticker on my F5 key—it makes it easy to find . Well worth doing. But any method is fine to run a program.

Thonny will execute the Python program, and you should see the output "Hello from Thonny!" displayed in the Shell pane at the bottom of the Thonny window.

If you look at the program window, you’ll notice that the small tab might say **<untitled>*** at the top. This indicates that the file hasn’t been saved yet. If it had the files name would be displayed here.

In case you’re curious, Thonny draws its name from a fictional snake character known as '**Thonny the Python**'—kinda like the character on the front cover!

When writing Python programs, you can use either uppercase or lowercase characters, but there are some general conventions for when to use each. As we delve deeper into Python, we’ll uncover and define these conventions.

A Real Program

Open a new code editor window by pressing the big green cross. Click in the Code Editor window and carefully enter the following:


```
#Filename: start1.py
import datetime
now = datetime.datetime.now()
print(now)
```

If you’ve downloaded the programs then this isn’t included. You need to type it in. Now, let’s bring this Python program to life. Hit the green '**Run**' button at the top of the window, and watch the result unfold in the '**Shell**' window. You should see the current date and time displayed down to six decimal points of a second!

Take a peek at the Assistant window—you should find a reassuring message confirming that your program is working perfectly, which is great considering we just ran it successfully!

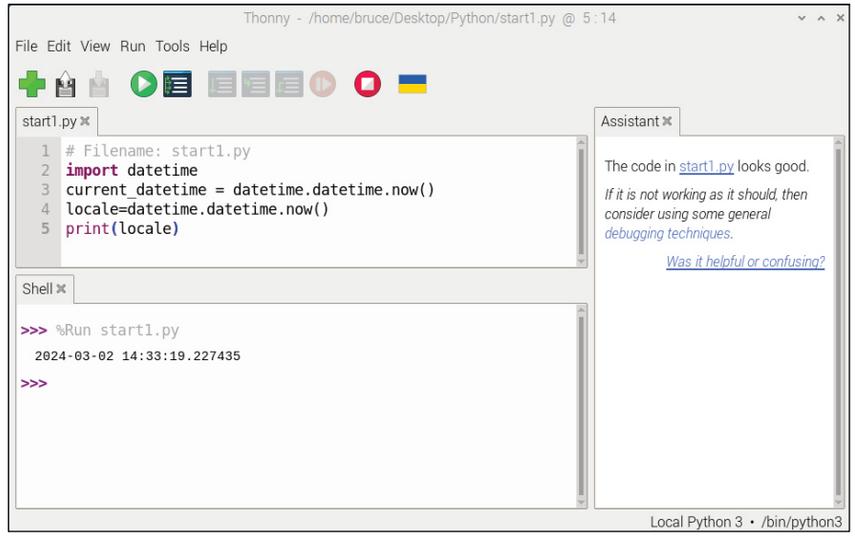


Figure 2d. Date and Time in the Shell window.

You can also run this program in the Shell window of the IDE. Follow these steps:

1. In the Code Editor, select all three lines of the program by clicking and pressing <Ctrl-A>. Then, press <Ctrl-C> to copy.
2. Navigate to the Shell window, click inside to select it, and press <Ctrl-V> to paste the program there.
3. Finally, press <Enter> to run the program.

The same process applies when using the Python Interpreter in the Terminal window. Select the Terminal window, choose 'Paste' from the ‘Edit’ menu, and press <Enter> to execute the program.

In both scenarios, every line you enter in either the Thonny Shell or the Python Interpreter in the Terminal window gets executed immediately. You'll see an output whenever a line involves an action that displays a result.

Things to Note: Here' are a few things to note about the program above.

- The first line starts with a **hash symbol, '#'**. A Python program ignores anything after a line that starts with a hash symbol. This allows you to put comments or notes in your programs. As you can see, I use this to denote the filename of the program. Thus, the program here is called 'start1.py'. You'll find it listed as such in the download programs. You can use as many comment lines as you wish. Don't go overboard otherwise you lose the program within the comments. And for the scope of this book, you don't need to type them in, if you are doing that.
- The program shows how we have used an imported item, by the name of **datetime**. This item is called a 'module' and contains the routines we needed to retrieve and print the date and time. A module is a file that contains Python code—which you can use in your own programs. For example, the datetime module helps you work with dates and times."
- The last lines gather information required and then displays it, and while you might now know the exact syntax of the rest of the program means you can read it and understand what is happening.

Thonny Windows

Thonny is flexible because it has several additional windows you can open and display. You can explore these by selecting the 'View' menu at the top of the window. A good one to add is the 'Files' window. You can navigate to your programs and load them with a double click, as shown in Figure 2e below. Note how the 'Files' window that the Python logo signifies Python programs.

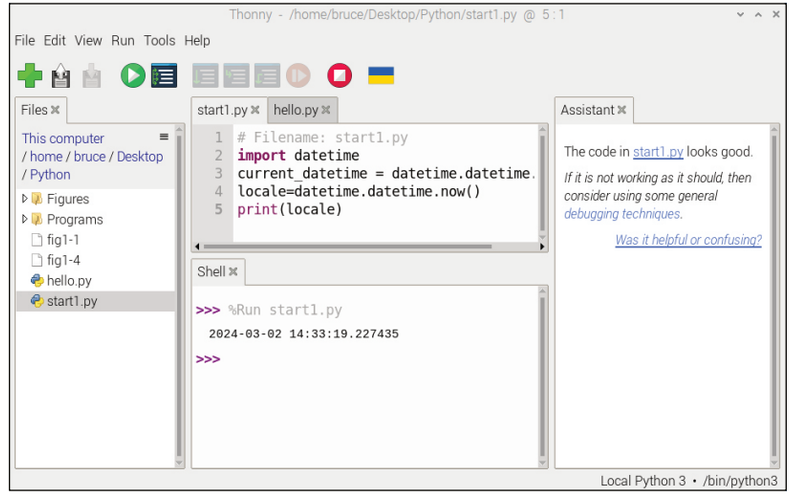


Figure 2e. Additional Thonny windows. Files is useful to keep open.

The Raspberry Pi File System

Understanding the file system on your Raspberry Pi is essential for any programmer. The system files are in what's known as the **'root'** directory, which is the starting point of the operating system. If you want to explore these files on Raspberry Pi OS, just open a Desktop window and use the **File Explorer** to check out the root directories.

Familiarising yourself with these system files helps you understand the structure of your Pi and makes customisation easier. Trust me, this knowledge will come in handy later, as we'll see in Chapter 03.

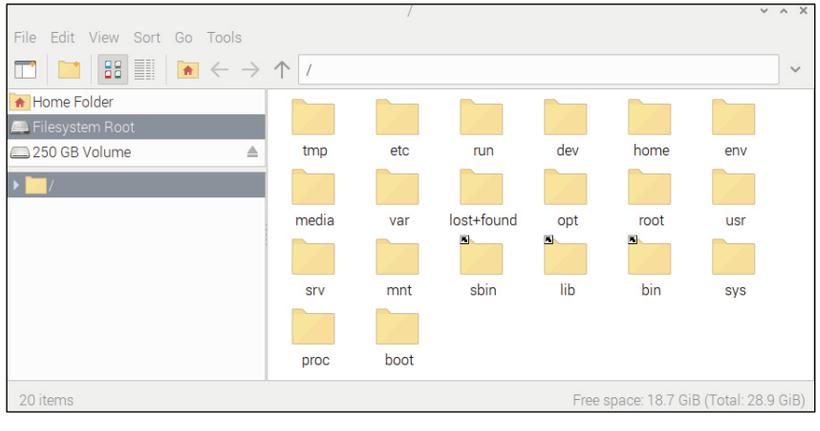


Figure 2f. Folders in the root directory.

Don't be shy about exploring these folders—just be careful not to delete or change anything.

- **/ (Root Directory):** The root directory is the top-level directory in the file system. Everything in the file system branches out from here. Note in **/root**, it's the home directory for the root user, but you should understand that regular users typically don't interact with it unless using administrative privileges.

- **/bin**: This directory holds essential binary executables (commands) necessary for system recovery and repair.
- **/boot**: Here, you'll find the files needed for your Raspberry Pi to boot up, including the bootloader, configuration files like '**config.txt**,' and the kernel.
- **/dev**: Contains device files that represent physical and virtual devices like disks, serial ports, and even random number generators.
- **/etc**: A treasure trove of system-wide configuration files and scripts. If you need to tweak installed software, you'll likely find its config files here.
- **/home**: This is where user home directories live. Each user on the system has their own sub-directory under '/home.'
- **/lib and /lib64**: These directories store essential shared library files that both the system and applications rely on.
- **/media**: If you plug in a USB drive or other removable media, it often gets mounted here.
- **/mnt**: This is a common spot for temporarily mounting file systems.
- **/opt**: Here, you might find additional software packages that aren't part of the default installation.
- **/proc**: A virtual file system offering a wealth of information about processes and system status.
- **/run**: This directory contains run-time data like process IDs and socket files.
- **/sbin**: Home to system binaries (commands) typically used by the system administrator.
- **/srv**: Intended for data served by the system, like web servers.
- **/sys**: Another virtual file system, exposing kernel and device information.

- **/tmp**: A temporary storage area for files, often wiped clean when the system reboots.
- **/usr**: This directory is packed with user-related programs, libraries, documentation, and other files.
- **/var**: This is where you'll find variable data files, like logs, spool files, and temporary files that stick around even after a reboot.

In the **/home** directory, you'll find a folder with your username—your personal space on the Pi. This is where all your folders and files live. If you need, you can create additional users, each with their own separate environment, allowing you to compartmentalise your Pi experience.

Head over to your **/home/username** directory to access all your data. Here, you'll come across folders like **Desktop**, **Pictures**, **Documents**, **Downloads**, and more. This is the ideal place to organise your programming world. Jumping back to Figure 2c you'll remember that the Python version we're using is located in: **usr/bin/.**

Important Line Wraps

Many of the lines of Python code in the rest of these books are too long to sit on one line. They therefore wrap into a second or more. If your type a program in and there is an error when you run the program than look at the error message and see if you can solve the issue for yourself. This may simply be deleting the <Enter> you have inserted as part of the line wrap.

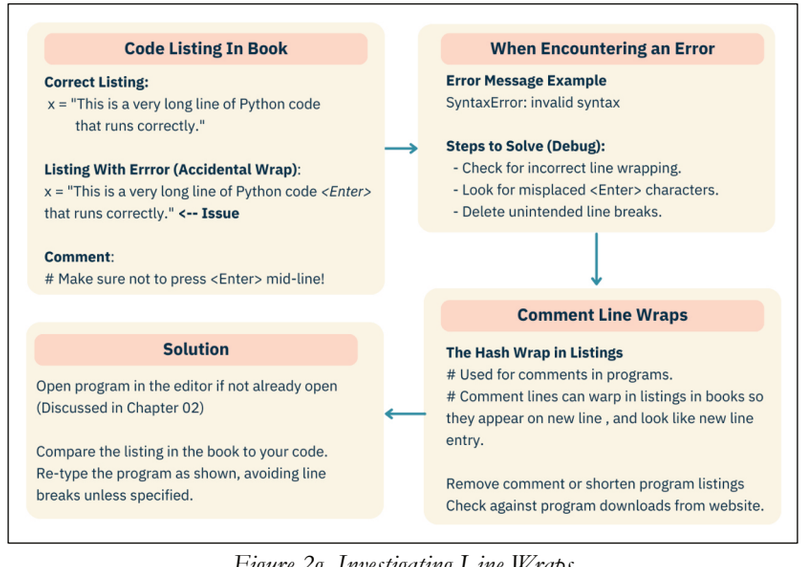


Figure 2g. Investigating Line Wraps

Questions

1. What command do you use to start the Python interactive shell from the Terminal window? And how do you exit the Python interactive shell and return to the regular Terminal prompt?
2. Which command should you run to check the version of Python 3 on your Raspberry Pi?
3. Why is it important to use the command ‘python3’ instead of ‘python’ when running Python code in this book?
4. What command updates the software packages, including Python, to the latest version on your Raspberry Pi?
5. What is displayed in the bottom-right corner of the Thonny window, and why is it important?
6. What is the purpose of a Python module? Give an example of a module used in the text.

03: A Matter of Style

Okay. You might find the opening section of this chapter requires some head-scratching—especially the terminology. But give it a go, work through the chapter, and then maybe come back here a second time to understand it better. It may not seem like it now, but these concepts will become second nature without you even noticing. Many people will shudder at me putting this chapter here, and many will simply ignore it. But I just want you to know how important it is. Feel free to skip it if you are confused, but do come back to it when you have completed the first dozen or so chapters.

Object-Oriented Programming

OOP is a style of programming that uses, as its name suggests, "objects" to construct programs. It allows you to model and manage the properties and behaviour of program code as 'real-world' concepts, making them more 'lifelike.' In your mind's eye, you can start to draw comparisons. Terms such as **inheritance** and **encapsulation** may seem complex at first, but they become clear with understanding and practice. They mean the same as they do in real life. You inherit something. You encapsulate something. Other terms, such as polymorphism and abstraction, may not be as intuitive but shouldn’t impede understanding or, more importantly, your learning of Python programming.

What’s important are the key building blocks of OOP. In Python, there are four fundamental concepts to grasp:

- Class
- Object
- Attributes
- Methods

The Block

Think of a **class** as a blueprint for a house. The blueprint outlines the structure, layout, and characteristics of the house, such as the number of bedrooms, the number of restrooms, and how many parking spaces there are. Is there a pool? These are the **attributes** of the class. A class contains attributes that define the characteristics of the object it creates.

From this blueprint, you can create multiple houses, each an **instance** of the class. For example, you could use the blueprint to construct a dozen identical houses on a street—a "block." However, not everyone wants an identical house. Some families might modify theirs: one might forgo a pool, another might convert the basement into an entertainment room, and so on.

To make these changes, you can adjust the **attributes** directly or use a **method** to do so. For example, if you want to convert a room into an entertainment room, you might create a method like **convert_to_entertainment_room()** that updates the relevant attributes. This is an example of **encapsulation**: bundling data (attributes) and methods (actions) within the class, so the class manages its own state and behaviours. **Abstraction** also plays a role here, as the method hides the complexity of the conversion, exposing a simple action for the user to call.

In this analogy:

- A **class** is the blueprint of the house.
- **Attributes** are the features of the house (e.g., number of bedrooms, presence of a pool).
- **Methods** are actions that can be performed by or on the house (e.g., unlocking a door, turning on the lights).

There are three methods here: **unlock_door():** A method to unlock the front door; **turn_on_lights()**: A method to turn on the house's lights; and **convert_to_entertainment_room()**: A method to re-purpose a room.

Changing a room's purpose is called modifying the state of the object. If done via a method like convert_to_entertainment_room(), it showcases how classes manage changes to their internal attributes or state.

In this case, modifying an attribute or changing the state of an object would be a way to describe the process of adapting the class to make a room into an entertainment room. If you are creating a specific method like `convert_to_entertainment _room()`, it's an example of encapsulation, where the logic to make the change is handled within the class itself. Encapsulation refers to bundling the data (attributes) and methods (actions) within a class, ensuring that the object manages its own state and behaviours. **Abstraction** also plays

a role here because the method hides the complexity of how the room is converted, only exposing a simple action for the user to call.

Think of a **method** as something the house can do or an action that can be performed on the house. Just like living in a house involves doing certain things, like unlocking the door, turning on the lights, or opening the garage, methods are actions that can be performed by an object created from a class.

This house blueprint can be used to construct multiple houses with shared features and design principles-but all based on the same plan, with adjustments as needed. Attributes are the ingredients in this 'House' recipe.

Constructor and Attributes

When creating an object, its attributes are set up using a **constructor**, a special method typically named **__init__** in Python. The constructor initialises the object's attributes with specific values. For example:

```
class House:
    def __init__(self, bedrooms, restrooms, garage,
basement, pool):
        self.bedrooms = bedrooms
        self.restrooms = restrooms
        self.garage = garage
        self.basement = basement
        self.pool = pool
```

This constructor initialises the object's attributes with values. A constructor is a method.

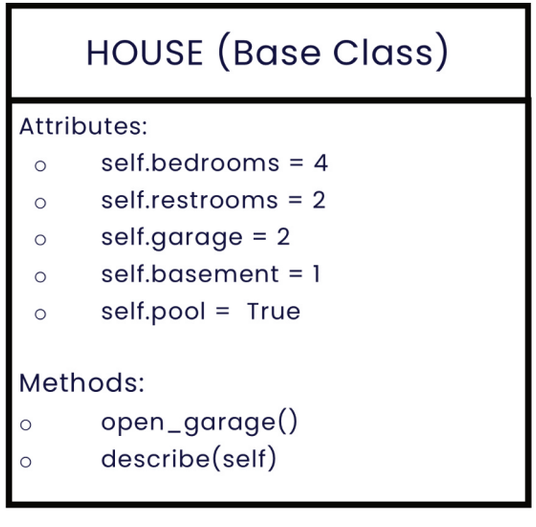


Figure 3a. The House Base Class blueprint.

Encapsulation ensures that these attributes are controlled and accessed only through specific methods, hiding the internal data from external modification. Each instance variable (e.g., self.bedrooms) belongs to a specific object, allowing each house (object) to have unique features.

Inheritance and Polymorphism

Inheritance allows us to extend the blueprint of a base class to create specialised versions. For instance, we might create subclasses like TownHouse, Bungalow, or Villa. These subclasses inherit attributes and methods from the base House class but can also add new ones or override existing ones. For example, a TownHouse class might inherit the bedrooms and restrooms attributes but introduce a new **number_of_floors** attribute. Inheritance allows us to reuse and extend the blueprint without rewriting everything from scratch.

Polymorphism allows these subclasses to define methods with the same name as those in the base class but with different behaviours. For instance, a **describe()** method in the House class might provide a generic description of a house, while the TownHouse class overrides it to include details like the number of floors. This flexibility makes OOP wonderfully flexible, enabling different classes to share a common interface while behaving differently.

- The **class** is the blueprint.
- **Attributes** are the features of the house.
- **Methods** are the actions the house can perform.
- **Inheritance** extends the blueprint for new types of houses.
- **Polymorphism** allows subclasses to implement shared methods in their own way.

Class Instance

The diagram opposite (Figure 3b) builds on the previous one and aims to provide a visual representation of how this works. We have the **House Base Class**, which lists the attributes of the class inside the box. This serves as a blueprint for all houses in a development. The base class house includes the following attributes:

- Four bedrooms, two restrooms, double garage, basement and pool

Next to the base class diagram, we have a specific house that is an instance created from this base class/blueprint. This individual house has the following attributes:

- Four bedrooms, two restrooms, double garage, no basement, no pool

This means it is based on the blueprint (the **base class**) but with altered attributes, demonstrating how instances of a class can have their own unique attribute values.

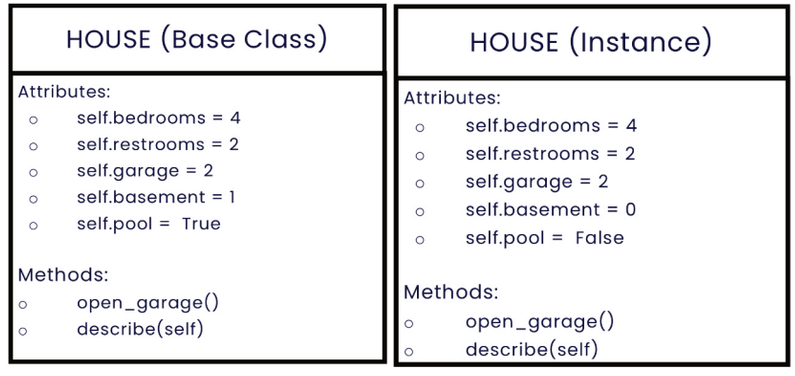


Figure 3b. Base class and an instance of a class.

As mentioned earlier, the attributes are defined using a special method called a constructor, typically named __init__ in Python. The constructor is a method that is automatically called when a new object of the class is created. It allows attributes to be customised during object creation by accepting parameters. For example:

```
class House:
    def __init__(self, bedrooms, restrooms, garage,
basement, pool):
        self.bedrooms = bedrooms
        self.restrooms = restrooms
        self.garage = garage
        self.basement = basement
        self.pool = pool
```

In this example, the constructor initialises the attributes of the house using the values provided as arguments when the object is created. For example:

```
house1 = House(4, 2, "double", False, True)
house2 = House(3, 1, "single", True, False)
```

Here, house1 and house2 are two different **instances** of the House class, each with their own unique attributes.

These attributes are known as **instance variables** because they are specific to each instance (or object) of the House class. Each object has its own set of these variables, meaning no two objects share attribute values unless explicitly programmed to do so. This allows for the creation of multiple

houses with different attributes. For example, one house might have a pool while another does not.

Derived Classes

Using House as a base class, we can create more specific types of houses like Townhouses, Bungalows, or Villas. These classes are derived from the original base class. They are still houses, but they differ in structure and behaviour while sharing some of the same attributes and methods. These derived classes can inherit the attributes and methods of the House class while also introducing their own unique characteristics.

For example, we might create a TownHouse class as a derived class of House. A townhouse shares many of the same attributes as a generic house (e.g., bedrooms, restrooms, garage) but has additional or modified characteristics, such as driveway_parking or end_of_row. These new attributes reflect the specific features of a townhouse that aren't present in the base House class.

In some cases, the derived class may override attributes or methods from the base class to better represent its specific type. For instance, while a generic House might have a describe() method that provides a basic description of the house, the TownHouse class could override this method to include details like "end-of-row" status or "shared walls."

Key Concepts

- **Inheritance**: Derived classes reuse attributes and methods from the base class, avoiding code duplication.
- **Overriding**: Derived classes can redefine methods or attributes to customise their behaviour.
- **Extensibility**: The base class (House) provides a foundation, while derived classes (TownHouse, Bungalow, etc.) add or modify features to create specific types of houses.

Methods in Derived Classes

Derived classes can override methods from the base class or define their own unique methods. For example, a **TownHouse** class might override the **describe()** method to exclude information about a pool (since townhouses typically don't have pools) and include details specific to townhouses, such as their position in a row of houses. Here's an example

```
class TownHouse(House):
    def describe(self):
        return f"This townhouse has {self.bedrooms}
bedrooms, {self.restrooms} restrooms, and is part of a
row of houses."
```


In this method, the placeholders (**{self.bedrooms}** and **{self.restrooms}**) are replaced with the values of the respective attributes. For instance, if self. bedrooms is 3, the output will show "This townhouse has 3 bedrooms...".

This structure exemplifies the principles of OOP, where objects (houses) have both data (attributes) and behaviour (methods) that model real-world entities.

Methods in the House Class

In the House class, there are two example methods: open_garage() and describe(). These methods define actions that a house (or an instance of the House class) can perform:

- **open_garage(self)**: This method simulates the action of opening the garage. It might provide details about the garage's size or indicate whether the house even has a garage.
- **describe(self)**: This method returns a string that describes the house, using its attributes, such as the number of bedrooms, restrooms, garage size, presence of an entertainment room, and pool.

These methods enable the objects (instances of House) to perform actions based on their attributes, reflecting the behaviour of real-world entities.

Polymorphism Example

Polymorphism allows objects of different derived classes to be treated as instances of the base class. Even though each derived class implements its own version of methods (like describe()), they can all be used through a common interface. Here's an example:

```
house = House(4, 2, "double", False, True)
townhouse = TownHouse(3, 2, "single")

print(house.describe())
# Output: A house with 4 bedrooms, 2 restrooms, and a
double garage.

print(townhouse.describe())
# Output: This townhouse has 3 bedrooms, 2 restrooms,
and is part of a row of houses.
```

In this example:

- The describe() method behaves differently depending on the type of object calling it (a House or a TownHouse), demonstrating polymorphism in action.
- Both objects can use the describe() method through the shared

interface of the base class, but each provides behaviour specific to its class.

Additional Examples: Classes Galore

To further understand inheritance and method customisation, let's consider a **Vehicle** base class. This class might have attributes like **make, seats, model, colour,** and **year.** It could also have methods like **power(), refuel(), drive(), stop(),** and **reverse()**. A derived class called **ElectricVehicle** could inherit most of these attributes and methods but replace the **refuel()** method with **charge()** to represent how electric vehicles operate.

Another Example: Shapes

Here's an analogy using shapes to illustrate inheritance and customisation:

- **Base Shape**: A generic shape with common characteristics, such as length and width. It might include a method to calculate the area (length * width).
- **Derived Shape (Rectangle)**: A rectangle is a specific kind of shape. It inherits the attributes length and width from the base shape and calculates the area using the base method.
- **New Shape (Circle):** A circle doesn't have length and width; it has a radius. While it may still use the concept of area, it overrides the method to calculate the area using the formula for a circle (π * radius^2

Here:

- **Inheritance**: Allows new shapes (e.g., Rectangle) to reuse and extend features of the base class (Shape) without rewriting everything.
- **Customisation**: New shapes (e.g., Circle) can override or define their own methods while remaining conceptually related to the base class.
- **Polymorphism**: Objects of different derived classes (Rectangle, Circle) can be treated uniformly as instances of the base class (Shape), but each implements its own specific behaviour.

Programming with OOP

Here's how you might typically construct a program to represent the OOP concepts we've discussed. These code snippets might not make sense immediately, but take your time to examine each one closely—you'll see how they fit together. The terminology used is intentionally simple and clear.

Example 1: The House Class


```
class House:
    def __init__(self, bedrooms, bathrooms, has_pool):
        self.bedrooms = bedrooms  # Attribute
        self.bathrooms = bathrooms  # Attribute
        self.has_pool = has_pool  # Attribute

    def unlock_door(self):  # Method
        print("The door is unlocked.")

    def turn_on_lights(self):  # Method
        print("The lights are turned on.")

    def fill_pool(self):  # Method
        if self.has_pool:
            print("The pool is being filled.")
        else:
            print("This house doesn't have a pool.")
```

In this example:

Attributes like bedrooms, bathrooms, and has_pool describe the house's characteristics.

Methods like unlock_door(), turn_on_lights(), and fill_pool() describe actions that can be performed by or on the house.

Example 2: Converting a Room to an Entertainment Room

We can add functionality to convert a room into an entertainment room:

```
class House:
    def __init__(self, bedrooms, bathrooms, has_pool,
rooms):
        self.bedrooms = bedrooms
        self.bathrooms = bathrooms
        self.has_pool = has_pool
        self.rooms = rooms

    def convert_to_entertainment_room(self, room):
        if room in self.rooms:
            self.rooms[self.rooms.index(room)] =
'entertainment room'
            print(f"The {room} has been converted into
an entertainment room.")
        else:
            print(f"There is no {room} to convert.")
```

Here, he convert_to_entertainment_room() method modifies the rooms list. It checks if the specified room exists and, if so, updates it to "entertainment room."

Example 3: Adding a Garage

```
class House:
    def __init__(self, bedrooms, bathrooms, garage_
size, has_garage):
        self.bedrooms = bedrooms
        self.bathrooms = bathrooms
```

```
            self.garage_size = garage_size
              # Number of cars the garage can hold
            self.has_garage = has_garage
              # Boolean to indicate if house has a garage

        def open_garage(self):
            if self.has_garage:
                print(f"The garage door is opening... This
garage can fit {self.garage_size} cars.")
            else:
                print("This house does not have a garage.")
```

You can create an instance of the House class and call its methods:

```
my_house = House(bedrooms=3, bathrooms=2, garage_
size=2, has_garage=True)
my_house.open_garage()
```

Output:

```
The garage door is opening... This garage can fit 2
cars.
```

You can expand this by adding more methods, such as:

```
def close_garage(self):
    if self.has_garage:
        print("The garage door is closing...")
    else:
        print("This house does not have a garage.")
```

Combining Snippets into a Full Program

You may feel that this chapter is way to advanced to be at the front of the book. It probably is. But OOP is essential to life on Python. Some of what you have read will stick, and as I've said, and promise, these concepts will go almost unnoticed by you and everything will fall into place as you continue on. Re-read this chapter every few chapters of the book. More will stick.

These snippets of code can be combined to create a comprehensive program. While we won't combine them here, I encourage you to experiment and build on these examples as you progress through the book. Keep tripping back here until you can create the completed program. Then you'll fully understand OOP! I promise it will happen.

INDEX

E

F

G

H

I

N

O

BOOKS FOR YOUR RASPBERRY PI

The Coding Press brings you the **latest** in hands-on computer programming confidence. The **Novice to Ninja** series caters to any beginner and will take you to a different level of programming — the Ninja Programmer. Whether you're completely new to coding or have some experience, our books provide the foundation knowledge and practical skills needed to write your own programs with clarity and confidence. With clear explanations and example programs, you'll explore coding in a way you never imagined. Our pages will transform you into a confident and capable programmer in the genre they cover.

You'll pick up best practices for writing clean, efficient code and gain the skills to debug and troubleshoot your programs effectively.

By the end of our books, you'll be equipped to tackle your own projects and solve problems in the subject matter covered. Our books provide a gateway to new ways of thinking and creating. Whether you're aiming to start a new career, enhance your current skills, or simply take on a new hobby, our Novice to Ninja guides will help you reach your goals. Step into the ongoing future of programming and unlock your coding potential today.

www.codingpress.com.au

www.brucesmith.info