

RASPBERRY PI EDITION

PYTHON UNLEASHED

BRUCE SMITH



NOVICE TO NINJA

CONTENTS

00: Novice to Ninja.....	13
Raspberry Pi Versions	14
Python Types	16
01: Hello	17
Pi Space.....	19
Python Uses.....	20
Learning.....	21
Checking In.....	24
02: Python Interactive	25
IDE	25
Trying It Out.....	28
A Real Program	28
Thonny Windows.....	30
The Raspberry Pi File System.....	31
Important Line Wraps.....	33
Questions	34
03: A Matter of Style	35
Object-Oriented Programming	35
The Block.....	36
Constructor and Attributes	37
Inheritance and Polymorphism.....	38
Class Instance	38
Derived Classes	40
Methods in the House Class.....	40
Classes Galore	41
Polymorphism Example.....	42
Programming OOP	43
Combining Snippets into a Full Program	44
Questions	44

04: Foundations	45	10: Sets	123
Modules and Libraries.....	48	Modifying Sets	126
Python Interactive Shell.....	50	Which One Where?	128
Terminal Shell.....	51	Questions	128
Bytecode.....	52		
Python Virtual Machine (PVM).....	54		
The Answer.....	54		
05: Environments	57	11: Tuples	129
Version Numbers	58	So Special.....	129
Virtual Environments with Thonny.....	59	When and How to Use Tuples.....	130
Questions	62	Tuple Methods.....	133
06: Variables & Strings	63	Tuple Packing and Unpacking.....	134
String Theory.....	65	Single-Element Tuples.....	135
Chopping Up Made Easy.....	68	Why Choose Tuples Over Dictionaries?	135
Haystack Needle.....	69	Tuples vs. Lists:	137
The Dynamic Duo.....	70	Using enumerate()	138
The Other Dynamic Duo.....	70	Tuples vs. Lists: What's the Difference?	140
Case Swapping.....	72	The Zipper	140
Reversing a String.....	73	Practical Uses for zip	142
Passing Multiple Arguments.....	74	Questions	143
Object Type.....	77		
f-Strings	79		
Question	79		
07: Loops.....	81	12: Stacks & Queues	145
For Loop	83	Using a List as a Stack or Queue	147
While Loop	83	First-In-First-Out (FIFO)	148
Loop Control Statements	83	Queue Module (Queue Class)	148
Nested Loops	84	Deque	150
Else Statements	85	Use Cases for Deques	151
Looping with an Index.....	87	Exercise: Mastering Stacks and Queues	152
Nested Loops	89		
Question	91		
08: Lists	93	13: Deep and Shallow	155
Common Lists	94	Whoo!	156
Working with List Operators.....	95	Another Look at Shallow Copy	157
Working With Different Lengths	100	Deep Copy: The Real Deal.....	160
When to Use List Comprehension.....	101	When to Use a Shallow Copy	162
Changing List Elements	102	Exercise: Deep and Shallow Copying	164
Adding Elements to a List.....	103		
Some Handy List Operations.....	103		
Counting, Finding, Sorting & Multidimensional.....	106		
Exercise: Putting It All Together	107		
09: Dictionaries	109	14: Environments +	167
Dictionaries and OOP.....	113	Common Issues and Troubleshooting	170
Similar But Different	115	APT and PIP	170
		Pip Pip3 Hooray!	173
		Is it Apt for the Raspberry Pi ?	175
		Questions	175
		15: More Thonny	177
		Use a Debugger.....	181
		Debugging in Thonny	181
		Thonny In and Out	182
		Questions	185
		16: In and Out	187
		Handling Errors and Validation.....	188
		Capturing Multiple Inputs at Once	190

Input with Validation: A Menu Example	192	Metacharacters	247
Common Input Mistakes to Avoid.....	193	Greedy Matching.....	248
Exercise: Using Input and Output	193	Lazy (Non-Greedy) Matching:.....	249
17: File Handling	195	Exploring Regular Expressions.....	251
Opening, Reading and Reading Files	195	22: Exceptions.....	253
Renaming Files	198	Handling Multiple Exceptions	254
Checking File Existence	199	Exception Hierarchy.....	256
Inserting Data into a File	199	Common Mistakes Using Exceptions.....	258
Shutil Module.....	199	23: Math	261
File Tree Management.....	200	Trig, Exponential, and Log Functions	264
Archiving and Compression.....	200	Using Math Constants.....	265
Deleting and Copying a Directory Tree	202	More Math Operations	267
Context Managers	203	The Math Module	268
Error Handling.....	203	Trigonometric and Logarithmic Functions.....	269
Handling File Errors with Try-Except	204	Questions	276
Using else and finally	204	24: Advanced Functions.....	277
File Permissions	204	Namespace and Scope.....	278
Stat Module	206	The Lambda Function.....	281
Processing a File.....	209	It's About Closure.....	283
Handling Large Files.....	210	Understanding Decorators	284
Passing File Objects:.....	210	How Decorators Differ from Regular Functions	288
Tempfile Module.....	210	Understanding Recursion	289
Questions	212	Generator Power	290
18: CSV and JSON	213	LEGB Rule	292
Dictionary of Lists:.....	217	Shadowing.....	293
What is JSON?	219	Avoid Shadowing	294
Configuration Files	221	Shadowing Example and Resolution	295
JSON config File.....	223	Questions	297
Handling JSON Errors	225	25: Matrix	299
19: Path & Pythonpath	227	Creating a Matrix Using a Nested List.....	301
Virtual Environments and \$PATH	228	Adding Matrices Together	301
pathlib	229	Subtraction	302
PTYHONPATH	231	Questions	303
PATH vs. PYTHONPATH.....	232	26: Linters	305
Questions	234	Who Decides?	305
20: OS Module	233	Style Documentation	305
Questions	234	Popular Linters	306
Writing a List of Files in a Directory to a List	235	Pylint	309
Listing Directories and Sub-directories	236	Autopep8 Linter	310
Directories, Sub-directories, and Files	236	Format a Python File.....	311
Capturing Output.....	237	Black Linter.....	312
Environmental Variables.....	239	Questions	313
21: Regular Expressions.....	241	27: Geany IDE	315
Useful Functions	242	Debugging with PDB.....	319
Common Regular Expressions	243		

Virtual Environments.....	322	Array Properties in NumPy.....	398
Geany Plugins.....	323	Common Dot Options for NumPy Arrays	399
Makefile	234	Exploring NumPy Data Types (dtypes)	401
28: Enviro Switching.....	325	Why is this all so Important?.....	405
Your Project's Best Friend.....	326	Questions	406
Tips for Using requirements.txt.....	327		
pipenv	328		
Environmental Variable	329		
Folders	330		
Welcome to the Magnificent virtualenvwrapper	331		
Commands Available.....	333		
Geany Virtual Environment Switching	333		
29: sys Module	341		
30: abc Module.....	345		
Real-World Scenarios.....	348	Creating a Pandas Series.....	408
Metaclasses	349	DataFrames from Different Types of Data	409
What Are Decorators Really Doing?	350	All Together Now	412
Common Pitfalls with ABCs.....	352	Data Cleaning	414
Questions	352	Chained Assignment in Pandas.....	416
31: datetime Module	353	Essential Pandas Functionality in Action	417
Time Zone Handling with zoneinfo	355	Exploring Pandas Dot Methods	420
Comparing Dates and Times	355	Wrapping Up	422
Questions	356	DataFrame-NumPy Array Structure Differences	423
32: RPi Graphics	357	Data Manipulation Together	424
Tick-Tock It's a Clock.....	359	Moving Data Between Pandas and NumPy	425
Organise with Frames	365	Key Differences.....	425
Framing Menus.....	367	When to Use Each	426
Questions	368	Exercise: Numpy and Pandas.....	427
33: PyGame	369		
Setup.....	370		
Paddle Operation	372		
Catch a Falling Star and Scoring	373		
Sound	377		
Event Handling and Program Loop	378		
Optimising Performance.....	379		
34: Pillow	381		
35: NumPy	387		
Values as a Variable Name.....	391		
Indexing and Slicing NumPy Arrays.....	392		
Broadcasting.....	393		
Append, Delete, and Insert	395		
Copying Arrays in NumPy	396		
36: Pandas	407		
Creating a Pandas Series.....	408		
DataFrames from Different Types of Data	409		
All Together Now	412		
Data Cleaning	414		
Chained Assignment in Pandas.....	416		
Essential Pandas Functionality in Action	417		
Exploring Pandas Dot Methods	420		
Wrapping Up	422		
DataFrame-NumPy Array Structure Differences	423		
Data Manipulation Together	424		
Moving Data Between Pandas and NumPy	425		
Key Differences.....	425		
When to Use Each	426		
Exercise: Numpy and Pandas.....	427		
37: Matplotlib Visuals	429		
Geographical Representations	436		
38: Dunder Methods	439		
Practical Use.....	441		
Why Use Dunder Methods Instead of List	442		
Operator Overloading:.....	443		
Custom Comparisons	444		
Callable Objects.....	442		
Dunder Methods for Containers	445		
39: APIs	447		
OpenWeatherMap Step-by-Step	451		
How Many API keys Needed?	455		
Questions	456		
40: Writing Modules.....	457		
What Functions Are in a Module?	463		
41: Building Websites.....	461		
Adding HTML templates.....	463		
Background Flask.....	465		
42: Docstrings	467		
Sphinx	468		
Worked Example.....	469		

43: GPIO and HATS.....	473
The Hardware Shift	473
Common HAT Patterns.....	477
44: Makefile	481
45: A Final Word	487
Index	485

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, and I 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.

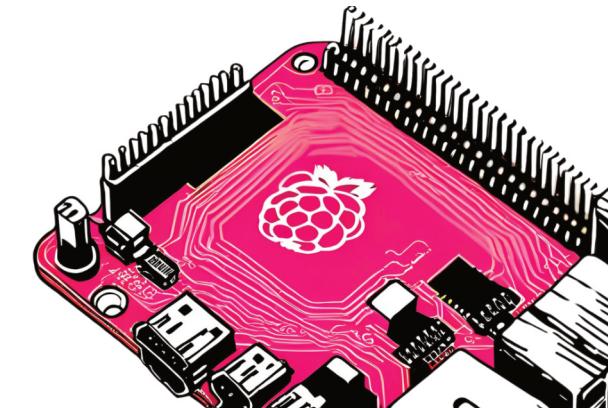
Raspberry Pi 1 (2012): Python 3 can run on this model, but due to its limited processing power and memory (512 MB RAM), it may not perform well for more complex programs, such as graphics and use of high intensity modules such as NumPy.

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 is 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.

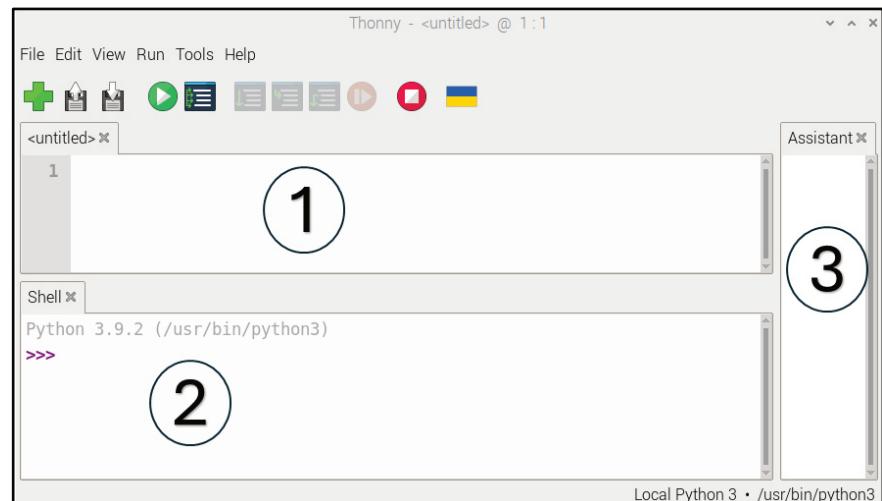


Figure 2a. Typical Thonny start-up screen.

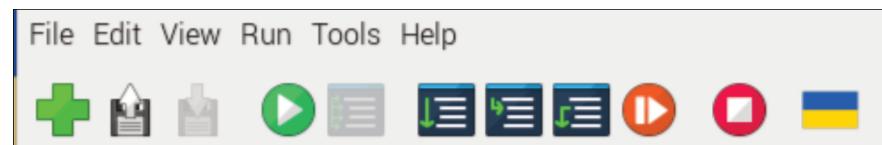


Figure 2b The Thonny Menu Bar.

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!

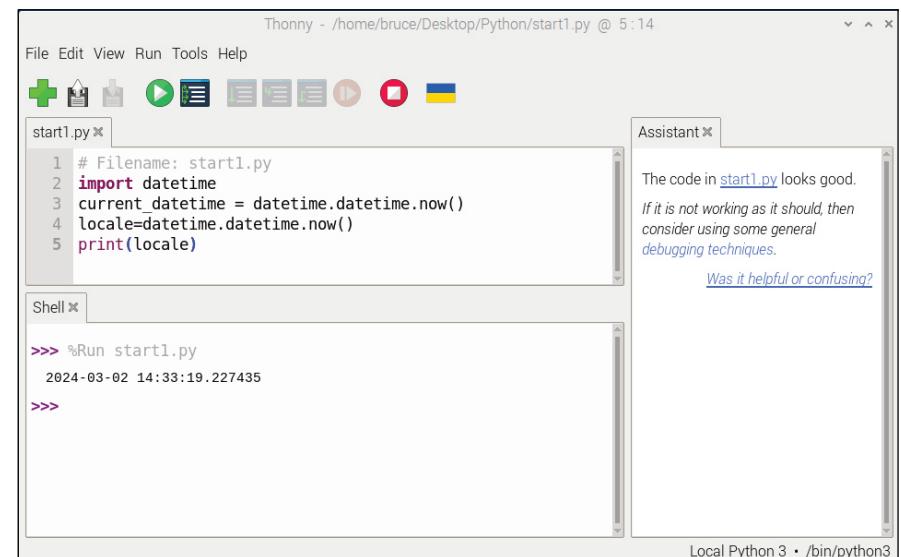


Figure 2d. Date and Time in the Shell window.

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.

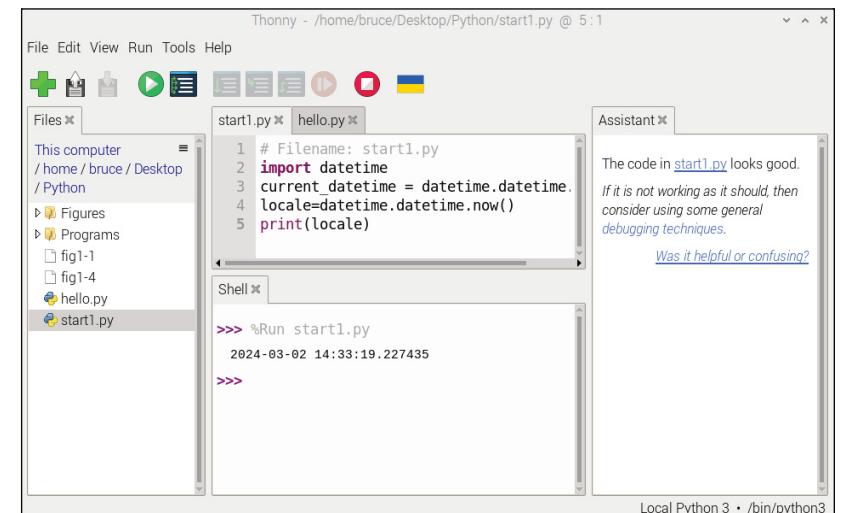


Figure 2e. Additional Thonny windows. Files is useful to keep open.

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.

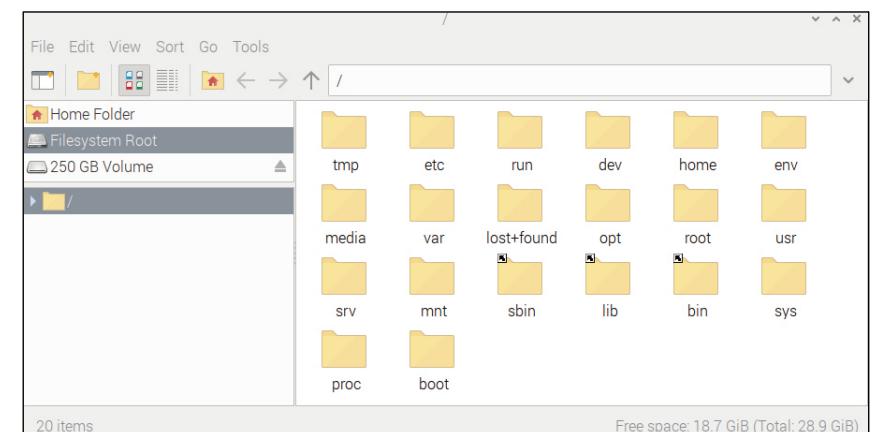


Figure 2f. Folders in the root directory.

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 you 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.

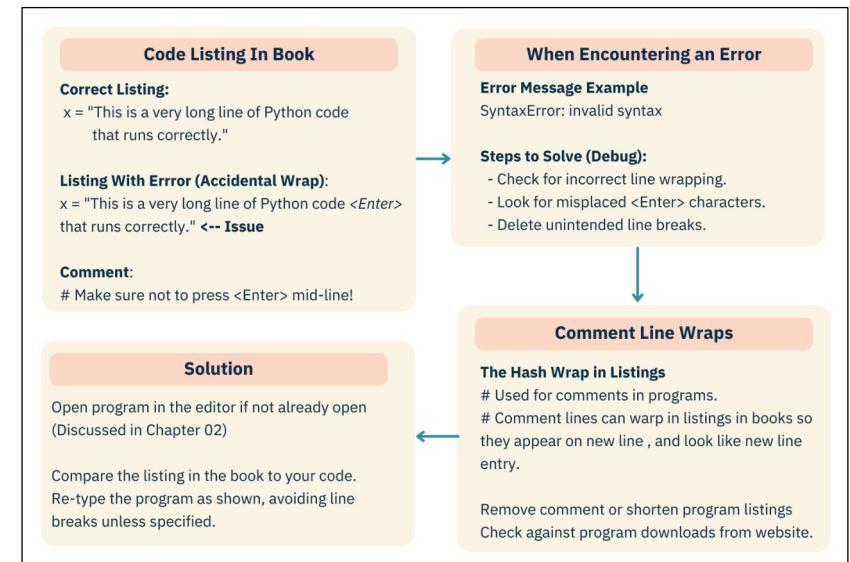


Figure 2g. Investigating Line Wraps

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.

HOUSE (Base Class)	
Attributes:	<ul style="list-style-type: none"> ○ <code>self.bedrooms = 4</code> ○ <code>self.restrooms = 2</code> ○ <code>self.garage = 2</code> ○ <code>self.basement = 1</code> ○ <code>self.pool = True</code>
Methods:	<ul style="list-style-type: none"> ○ <code>open_garage()</code> ○ <code>describe(self)</code>

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.

HOUSE (Base Class)	HOUSE (Instance)
<p>Attributes:</p> <ul style="list-style-type: none"> ◦ <code>self.bedrooms = 4</code> ◦ <code>self.restrooms = 2</code> ◦ <code>self.garage = 2</code> ◦ <code>self.basement = 1</code> ◦ <code>self.pool = True</code> <p>Methods:</p> <ul style="list-style-type: none"> ◦ <code>open_garage()</code> ◦ <code>describe(self)</code> 	<p>Attributes:</p> <ul style="list-style-type: none"> ◦ <code>self.bedrooms = 4</code> ◦ <code>self.restrooms = 2</code> ◦ <code>self.garage = 2</code> ◦ <code>self.basement = 0</code> ◦ <code>self.pool = False</code> <p>Methods:</p> <ul style="list-style-type: none"> ◦ <code>open_garage()</code> ◦ <code>describe(self)</code>

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 ($\pi * \text{radius}^2$).

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, the 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

__add__ 444, 448, 451
__call__ 449, 450
__dict__ 326
__doc__ 469
__eq__ 449, 451
__getitem__ 449, 450
__init__ 37, 38, 42, 43, 67, 114, 136, 146, 288, 353, 368, 369, 377, 378, 379, 392, 444, 445, 446, 448, 449, 450, 451, 468, 478
__iter__ 450
__le__ 449
__len__ 450
__lt__ 449
__main__ 202, 224, 237, 260, 285, 286, 287, 325, 347, 348, 369, 381, 382, 447, 458, 467, 471, 473
__name__ 202, 224, 237, 260, 285, 286, 287, 325, 347, 348, 369, 381, 382, 447, 458, 467, 471, 473
__new__ 353
__repr__ 288, 444, 445, 446, 447, 448, 450
__setitem__ 449, 450
__str__ 444, 445, 446, 447, 450, 451
__version__ 392
__build__ 479
distutils_hack 392
files 237
static 479
templates 479

A

abc 288, 350, 351, 352, 354, 355, 356, 357
abcs 352, 353, 356
abs 268, 390
absolute 25, 268, 425
abspath 479
abstract 19, 288, 304, 350, 351, 352, 356, 357
abstractclassmethod 350, 352
abstraction 35, 36, 447
abstractmethod 288, 350, 351, 352, 355
abstractproperty 350, 352
abstracts 54, 464

acronym 291
act 37, 57, 370, 452
activate 168, 169, 173, 228, 233, 326, 330, 331, 341, 343, 344, 456, 481
actuators 20
add 18, 20, 21, 30, 36, 38, 40, 48, 441
add_numbers 49, 56, 75, 314, 315, 317
add_subplot 440
addhandler 259
72, 376, 381, 383, 400, 431, 448, 472
addition 14, 49, 100, 101, 114, 116, 126, 151, 195, 221, 262, 266, 267, 305, 306, 307, 308, 320, 354, 360, 393, 399, 410, 448, 451, 477
additions 116, 336
address 27, 58, 60, 156, 251, 260, 336, 420, 448, 471, 472, 473
adds 49, 56, 75, 100, 109, 149, 163, 195, 198, 234, 266, 284, 285, 366, 373, 393, 399, 400, 406, 418, 431, 442
admin 228, 470
administrator 32
ads 328
algebra 49, 64, 304, 392, 431, 432
algorithm 151, 351, 355
algorithms 18, 19, 151, 274
align 370, 473
alpha 408, 437
alphanumeric 243, 248
analogous 47
analogue 270
analogy 36, 44, 45, 52, 53, 57, 145, 278
analytics 17, 18, 20
anchors 244, 245
api 18, 223, 224, 239, 452, 453, 454, 455, 456, 457, 458, 459, 460, 461, 464, 465, 477
api_key 224, 454, 457, 459
api_keys 223, 224
api_weather 457
apt 21, 24, 25, 121, 170, 171, 172, 173, 175, 313, 328, 335, 392, 450, 474, 477
architecture 407, 462, 463
archive 200, 201, 202
archived 201, 312
archives 201
archiving 201
arg 178, 347
args 76, 285, 286, 326
argument 46, 47, 66, 86, 130, 138, 178, 222, 281, 282, 283, 284, 286, 288, 296, 297, 315, 326, 361, 388, 414, 417, 421, 422, 433
arguments 45, 46, 75, 76, 77, 211, 277, 281, 282, 284, 285, 287, 293, 346, 347, 349, 466
argv 339, 346, 347, 349
array 18, 20, 48, 109, 223, 305, 308, 393, 394, 395, 396, 397, 398, 399, 400, 401, 402, 403, 404, 405, 406, 408, 409, 410, 413, 426, 428, 429, 430, 431, 432, 443
array1 398, 399, 400
ascii 407
asctime 259
aspect_ratio 390

assembler 482
 assembly 208
 assets 384
 astype 409, 419, 420
 asynchronous 17
 asyncio 17
 attribute 36, 42, 43, 44, 114, 126, 136, 207, 287, 326, 346, 409, 450, 469, 476
 attributeerror 126
 auto 384, 479
 autodoc 479, 480
 autopep8 312, 313, 316, 319
 autoremove 173
 average 123, 215, 216, 217, 226, 414, 416, 419, 420, 421, 440, 456
 axes 395, 396, 405, 435, 441
 axes3d 440
 axis 395, 396, 403, 404, 406, 410, 424, 429, 433, 438, 440, 441
 axs 443

B
 backup 76, 201, 335, 461
 bar3d 440, 441
 bare 199, 255, 258, 323
 base_url 457
 bash 51, 167, 322
 bashrc 335, 336
 bcm 465
 bcm2711 462
 bdfl 310
 bicubic 387
 bin 27, 32, 33, 58, 62, 169, 173, 227, 228, 229, 275, 327, 330, 331, 334, 341, 343, 440, 441, 456, 479
 binaries 32, 228
 binary 32, 261, 274, 275
 bit 19, 26, 28, 50, 52, 53, 58, 82, 92, 95, 104, 134, 181, 190, 205, 235, 262, 265, 273, 274, 275, 276, 327, 333, 334, 338, 366, 370, 380, 392, 406, 407, 457, 477, 482
 bits 57, 273, 274, 275, 276, 322
 black 23, 312, 313, 317, 318, 319, 375, 376, 377, 378, 380, 383
 blur 387
 blurred_image 387
 blurring 386
 bmp 386
 bmw 113
 bookworm 16, 24, 170
 bool_ 407, 408
 bool_array 408
 boolean 43, 69, 129, 223, 270, 272, 273, 274, 407, 408
 booleans 129, 219, 220
 boxplot 427, 438
 boxplots 427
 bpython 248
 branches 32
 branching 281
 broadcom 462
 broader 67, 203, 311, 312
 brython 16

bubble 57
 button_frame 371, 372, 373
 byte 78, 79, 410
 bytecode 52, 53, 54
 bytes 78, 118, 207, 208, 408, 409, 410
 bytes_ 408
 bytes_array 408

C
 calculate_ 286
 calculate_area 476
 calculate_sum 77, 286, 287, 297, 347
 calculated 294, 295, 391, 395, 399, 430
 calculated_result 295, 296
 calculates 56, 70, 107, 226, 286, 287, 296, 297, 347, 396, 405, 420, 476
 calculus 261, 264
 call 25, 26, 36, 43, 44, 46, 47, 49, 59, 66, 67, 75, 115, 145, 197, 203, 218, 222, 280, 285, 286, 289, 290, 297, 325, 327, 357, 359, 374, 382, 455, 471
 cat 246, 247
 catalog 355, 356
 catalogue 235
 cdvirtualenv 337
 ceil 265, 268, 269, 276
 celsius 79, 457, 459
 celsius_to_fahrenheit 79
 cfg 62
 chaos 190
 char 120
 char_count 120
 chdir 238
 chmod 206, 207, 209
 chunk 20, 102, 156, 210
 chunk_size 210
 chunks 95, 210
 class 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 47, 49, 55, 56, 67, 77, 114, 136, 146, 148, 149, 150, 152, 255, 256, 278, 287, 288, 322, 350, 351, 352, 353, 354, 355, 356, 357, 358, 359, 361, 368, 377, 378, 379, 384, 444, 445, 446, 447, 448, 449, 450, 451, 476, 478
 class_ 353
 class_dict 353
 classify 391
 classmethod 288
 clock 270, 364, 365, 366, 375, 377, 378, 379, 383
 close 60, 129, 197, 203, 209, 211, 228, 363, 382, 384, 473, 474
 close_garage 43
 closed 51, 179, 203, 211
 cloud 20, 187, 334
 cls 287, 288, 353
 cmap 439
 cmd_build 343
 cmd_compile 343
 cmd_execute 342, 343
 cmyk 387
 col 398, 422
 collaborate 13
 collection 48, 78, 87, 115, 120, 121, 123,

124, 125, 130, 133, 135, 172, 278, 305, 328, 397
 collections 91, 93, 94, 117, 124, 128, 129, 132, 137, 140, 149, 150, 152, 153
 cols 403, 404
 4, 425, 426, 427, 429, 430, 432, 433
 column_means 429
 column_sum 403, 404
 combine 56, 70, 71, 72, 92, 103, 111, 140, 356, 359, 370, 470
 combined 25, 43, 47, 356, 358, 451, 462
 combined_list 103
 combines 143, 312, 359
 combining 70, 71, 120, 133, 273, 278, 447
 combo 366, 367
 combobox 367
 command_output 237
 compare 73, 98, 123, 125, 128, 151, 268, 361, 391, 409, 447, 451
 compartmentalise 33
 compile 243, 323, 327, 341, 342, 343
 compiled 53, 56, 171
 compiler 52, 53, 322, 342
 compiles 54, 243, 342
 complex128 407, 408
 complex256 407
 complex64 407
 complex_array 408
 complexities 54
 compressed_value 275, 276
 compute 263, 276, 395, 396, 405, 427
 con 289
 concat_list 104
 concatenate 68, 71, 72, 112, 114, 133, 71, 72, 104
 concatenating 112, 230
 concatenation 68, 70, 71, 72, 104
 concurrent 18, 19
 concurrently 148
 condition 81, 83, 85, 90, 91, 184, 255, 270, 288
 conditional 101, 107, 184, 188, 271
 conditionally 101
 conditions 83, 85, 91, 101, 270, 271, 273, 459, 460
 conf 478, 479, 480, 481
 config 32, 223, 224, 225, 363, 364, 366, 479
 config_file 224
 conformance 316
 conjunction 209
 container 47, 48, 49, 94, 136, 278
 containers 45, 63, 370, 449
 contextlib 288
 contextmanager 288
 contexts 264, 447
 contextual 151
 contiguous 305
 convert 36, 42, 79, 106, 122, 124, 128, 130, 131, 138, 152, 188, 191, 192, 219, 220, 221, 222, 223, 262, 360, 369, 387, 391, 396, 409, 419, 430, 435, 436, 437, 438, 439, 477
 convert_ 36
 convert_to_ 37
 convert_to_entertainmentroom 42

D

dashboard 453, 470
 dashboards 434
 data_dict 415, 416, 418
 data_from_dict 416, 417
 data_from_lists 416
 data_lists 414, 415, 417
 database 94, 130, 203, 223, 224, 225, 239, 259, 260, 360, 470
 databases 221, 412

dataclass 288
 dataframe 413, 414, 415, 416, 417, 418, 419, 420, 421, 422, 423, 424, 425, 426, 427, 428, 429, 430, 431, 432, 435, 436, 437, 438, 439, 443
 dataframes 412, 414, 416, 417, 418, 419, 428, 430, 431
 dataset 117, 196, 216, 217, 419, 420, 428, 431, 437, 443
 datasets 18, 91, 123, 124, 125, 126, 216, 217, 277, 290, 409, 425, 429, 431, 439, 442
 date 29, 30, 49, 125, 169, 171, 205, 206, 245, 322, 358, 359, 360, 361, 364, 365, 366, 456, 459, 460, 471
 date_string 358
 dates 30, 48, 137, 140, 358, 359, 360, 361
 datetime 28, 30, 48, 49, 205, 358, 359, 360, 361, 364, 365, 408, 456, 457
 datetime64 408
 datetime_array 408
 datetime_object 359
 dateutil 331
 db_host 224
 db_name 223
 deactivate 168, 229, 234, 330, 337
 deactivates 337
 debug 181, 182, 183, 184, 186, 224, 225, 259, 325, 326, 375, 447, 461
 debug_mode 119, 223, 224, 225
 debugged 326
 debugger 181, 183, 185, 324, 325, 326
 debuggers 181
 debugging 21, 53, 181, 182, 183, 184, 185, 255, 257, 258, 260, 284, 288, 310, 322, 323, 324, 325, 326, 346, 349, 434, 445, 446, 447, 455
 dec 266
 decode 79, 219, 220
 decoding 220
 decompress 201, 275, 276
 decompressed 275
 decompressed_num1 275, 276
 decorator 284, 285, 286, 287, 288, 296, 297, 351, 352, 354
 decorator_name 284, 351, 354
 decorators 277, 283, 284, 285, 287, 288, 350, 354, 355, 357
 deep_copy 119, 158, 159, 161, 163, 164
 deepcopy 119, 157, 158, 159, 161, 163, 165
 deg 454
 degree 14, 316
 del 65, 104, 105, 110, 111, 117
 delay 43, 457, 458, 459, 460
 delays 384
 delimiter 68, 69, 213, 214, 218
 delimiters 213
 delitem_ 450
 deliver 355
 deque 148, 149, 150, 151, 152, 153
 deque_length 150, 151
 dequeues 149, 151, 152
 dequeue 148, 149, 152, 153
 derivative 16
 destination 200, 202, 239

destination_dir 201, 202
 destination_file 200
 destination_folder 200
 deviation 425, 426
 df 419, 420, 422, 423, 424, 425, 426, 427, 428, 429, 430, 436, 437, 438, 439
 df2 424
 df_dropped 424
 df_dropped_na 424
 df_filled 424
 df_from_csv 416, 418
 df_from_dict 416, 417, 418
 df_from_lists 414, 416, 417, 418
 df_missing 424
 df_pivot 424
 df_sorted 424
 df_stacked 424
 df_unstacked 424
 dht22 225
 dict 120, 131, 132, 142, 293, 353
 dictwriter 213
 dimensional 89, 305, 395, 397, 413, 431, 432
 directory_list 236
 directory_path 201, 236, 237
 directory_structure 236, 237
 directory_to_zip 202
 display_current_datetime 359
 display_menu 192
 display_person_details 447
 display_weather_history 458
 distract 293
 distributed 440, 441
 divide 97, 241, 254, 256, 257, 394
 division 183, 254, 257, 259, 262, 267, 305, 394, 410
 division_result 262
 docs 479, 480
 docstring 288, 315, 469, 476, 480
 docstrings 311, 315, 469, 476, 477, 478, 480
 doctest 478
 doctype 472
 domain 251, 252
 domains 21, 453
 dpkg 168
 dr 14
 draw 35, 314, 368, 377, 380, 386, 387, 388, 389
 draw_labels 442
 drawing 161, 369, 378, 383, 387, 389
 drop 25, 26, 59, 60, 61, 62, 66, 169, 343, 362, 366, 367, 421, 424, 426
 drop_ 419
 drop_duplicates 420
 dropdown 367, 368, 372, 373
 dropdown_label 367
 dtype 403, 404, 405, 408, 409, 413
 dtypes 406
 dunder 444, 446, 447, 448, 449, 450
 dunder1 444
 duplicate 117, 118, 123, 124, 125, 132, 135, 155, 162, 200, 402, 419, 420, 421
 dx 440
 dy 440

E
 echo 227, 231, 233, 336
 ecosystem 328
 edge_enhance 387
 editor 2, 26, 27, 28, 29, 214, 294, 318, 321, 322, 328, 335, 342
 editors 5
 element_at_index_1 150
 element_diff 307
 element_sum 307
 elif 79, 85, 192, 361
 else 42, 43, 58, 59, 79, 82, 85, 102, 162, 183, 184, 186, 187, 189, 191, 192, 199, 200, 204, 208, 224, 230, 242, 245, 246, 247, 252, 253, 254, 258, 267, 273, 286, 289, 314, 317, 320, 322, 325, 331, 332, 348, 361, 370, 390, 467, 468, 469
 email 189, 248, 251, 252
 email_dict 252
 email_list 252
 empty_directory 238, 239
 encapsulate 35, 280, 287, 447
 encapsulated 67
 encapsulates 45, 67
 encapsulating 67, 169
 encapsulation 35, 36, 37, 44, 47, 55, 67, 114, 283
 encode 79, 219, 220
 encoding 214, 215, 218, 220, 237, 238
 encryption 274
 end_time 286
 ended 149, 150
 enqueue 148, 149, 152, 153
 ensure 2, 22, 24, 25, 27, 91, 117, 121, 122, 123, 133, 134, 152, 153, 164, 168, 170, 172, 173, 175, 192, 200, 202, 203, 212, 229, 234, 251, 259, 260, 281, 297, 308, 311, 318, 323, 335, 336, 352, 354, 356, 361, 383, 384, 391, 392, 416, 421, 423, 435, 441, 453, 471, 479, 481
 enumerate 84, 87, 88, 92, 138, 139, 140
 enumerated 88
 env_name 336, 337
 env_vars 238, 239
 enviro 330, 332, 334, 336, 338, 340, 342, 344
 environ 233, 238, 239, 333
 environment 18, 19, 25, 33, 52, 57, 58, 59, 60, 61, 62, 167, 168, 169, 170, 171, 173, 175, 227, 228, 229, 231, 232, 233, 234, 235, 238, 239, 282, 313, 314, 318, 320, 322, 324, 326, 327, 330, 331, 332, 333, 334, 335, 336, 337, 338, 339, 341, 343, 344, 374, 386, 392, 393, 439, 456, 461, 463, 464, 471, 477, 478, 479, 481
 environmental 333
 epoch 208
 epub 477
 eq_ 288, 449
 equal 83, 184, 263, 264, 309, 437, 438, 449, 451, 467
 equality 263, 449
 error 26, 65, 79, 126, 127, 153, 159, 177, 178, 179, 180, 181, 183, 184, 185, 186, 188, 191, 201, 202, 203, 204, 205, 212, 214, 215, 218, 224, 226, 227, 237, 238, 253, 254, 255, 257, 258, 259, 260, 279, 280, 281, 289, 312, 314, 315, 323, 347, 348, 384, 398, 457, 458
 error_ 260
 error_logger 259, 260
 errors 21, 52, 79, 91, 119, 120, 134, 170, 177, 178, 180, 181, 185, 188, 192, 199, 202, 203, 204, 212, 216, 225, 231, 239, 253, 254, 256, 257, 258, 259, 260, 278, 293, 294, 310, 312, 314, 315, 316, 322, 323, 348, 356, 453, 459
 ethernet 462
 eval 26, 446
 evaluate 270, 271, 272
 evaluates 50, 66, 281, 312
 evolve 304
 exact 30, 57, 75, 135, 145, 179, 217, 331, 376, 395, 463
 exception 188, 201, 202, 203, 204, 214, 215, 218, 253, 254, 255, 256, 257, 258, 259, 348
 exception1 254
 exceptions 204, 253, 254, 255, 256, 257, 258, 259, 280, 322, 457, 476
 exceptiontype 255
 exclude 200
 excludes 86, 256
 exe 52
 executable 32, 53, 54, 59, 60, 61, 169, 228, 232
 executables 227, 228
 execute 19, 23, 28, 29, 53, 85, 148, 158, 182, 204, 206, 207, 208, 237, 238, 239, 240, 285, 286, 290, 296, 322, 323, 324, 327, 337, 339, 341, 342, 343
 execute_command 237
 executed 30, 54, 81, 168, 228, 253, 325, 353
 executes 46, 52, 54, 324, 343
 executing 54, 78, 85, 174, 206, 232, 237, 238, 279, 282, 283, 314, 323
 exist 36, 112, 119, 120, 126, 127, 128, 196, 198, 199, 203, 204, 212, 216, 230, 236, 237
 exist_ok 200, 202
 existence 112, 199, 212
 exit 24, 33, 51, 91, 92, 182, 190, 192, 224, 347, 348, 349, 380, 381, 382
 exits 83, 92, 347, 382
 exp 264
 expand 322, 368, 370, 376, 463
 expanded 399, 400
 exponent 76
 exponential 264
 export 213, 228, 336
 ext 480
 extend 38, 71, 98, 121, 247, 260, 277, 283, 327, 350, 351, 354, 442, 452
 extract_to 201, 202
 extract_valid_emails 251, 252
 extrapolate 235

F
 factor 283, 449

factorial 184, 185, 186, 286, 287, 289
 factorials 184, 288
 fail 239
 failed 224, 226, 457
 false 40, 69, 83, 85, 91, 98, 112, 183, 186, 224, 268, 270, 271, 272, 273, 368, 377, 379, 404, 407, 408, 417, 418, 438, 449, 466, 467
 familiar 26, 91, 155, 197
 ffill 419
 fg 365, 366, 367
 fib 59, 188
 fibonacci 288
 field 13, 338, 341, 370, 371, 373, 412, 454
 fieldnames 215
 fields 261, 304, 362, 373, 454, 455
 fifo 145, 148, 149, 150, 153
 file_handler 259
 file_info 207, 208, 209
 file_list 236
 file_path 205, 214, 215, 218, 381, 382
 file_stat 205
 file_to_change 206, 209
 file_to_check 199
 filed 338
 filehandler 259
 filemode 208, 209
 filenotfounderror 196, 203, 204, 214, 215, 218, 224, 236, 237
 filername 225
 files 25, 28, 30, 31, 32, 33, 50, 51, 52, 62, 78, 96, 169, 171, 172, 195, 196, 197, 198, 199, 200, 201, 203, 204, 205, 206, 210, 211, 212, 213, 219, 220, 221, 223, 226, 228, 230, 231, 232, 235, 236, 238, 239, 240, 253, 260, 278, 288, 291, 313, 322, 324, 333, 335, 338, 340, 341, 342, 343, 381, 384, 389, 417, 418, 419, 463, 477, 478, 479, 480
 filesystem 229
 fillvalue 100, 143
 filter 101, 123, 282, 387, 469
 filtered 107
 find 14, 17, 21, 25, 26, 28, 29, 30, 32, 33, 35, 48, 57, 58, 62, 69, 70, 93, 100, 101, 103, 104, 105, 106, 110, 112, 123, 124, 125, 128, 133, 169, 185, 190, 213, 227, 231, 232, 233, 234, 241, 242, 245, 246, 247, 250, 263, 276, 278, 292, 304, 310, 313, 315, 328, 348, 376, 381, 383, 386, 391, 406, 410, 464, 471, 477, 479
 findall 241, 242, 244, 248, 251, 252
 finding 105, 125, 126, 183, 241, 289, 466
 flac 381
 flag 273, 316
 flags 205, 243, 273, 274, 342
 flake8 294, 312, 313
 flask 386, 456, 470, 471, 472, 473, 474
 flask_app 474
 flip 72, 73, 74, 97, 135, 270, 377, 378, 380, 381, 382, 383, 386, 387, 389
 flip_left_right 387
 flip_top_bottom 387
 float 63, 78, 87, 129, 188, 191, 193, 262, 347, 409, 419, 476
 float128 407

float16 407
 float32 407
 float64 407, 408, 409
 float_array 408
 font 364, 365, 366, 367, 368, 375, 376, 377, 378, 380, 387, 388, 389, 472
 font_size 119, 375, 376, 377, 378
 fonts 375, 377, 378, 384, 387, 389
 fontsize 436, 437, 438, 439, 442
 form 2, 54, 65, 67, 68, 134, 310, 317, 322, 412, 464
 format 26, 53, 66, 78, 114, 196, 201, 208, 209, 219, 220, 221, 222, 223, 240, 251, 254, 312, 313, 316, 317, 326, 342, 358, 359, 360, 361, 364, 365, 376, 381, 382, 386, 387, 393, 415, 417, 419, 429, 453, 454, 455, 460, 477
 format_permissions 205
 formats 14, 201, 213, 221, 251, 358, 359, 365, 386, 418, 434, 460, 477
 formatted_datetime 359
 formatter 259, 312, 313, 317
 forth 318, 430
 fortran 18
 found 19, 78, 106, 120, 133, 196, 204, 214, 215, 218, 224, 227, 242, 244, 245, 246, 247, 291, 292, 388, 389
 frame 371, 372, 373, 380, 383
 frames 370, 371, 372, 373, 378, 383, 439
 framework 16, 17, 353, 374, 470
 frameworks 16, 18, 354, 386, 470, 471
 fromtimestamp 205
 front_element 152
 frozenset 123, 124, 126
 fruit 57, 82, 84, 87, 88, 272
 func 284, 285, 286, 287, 296, 354
 func_obj 469
 function 37, 41, 45, 46, 47, 49, 56, 64, 65, 66, 67, 75, 76, 77, 78, 79, 81, 84, 86, 87, 88, 92, 94, 96, 100, 103, 106, 114, 115, 122, 130, 131, 133, 135, 138, 139, 140, 141, 144, 147, 153, 157, 164, 165, 178, 181, 182, 184, 187, 188, 190, 191, 192, 198, 200, 201, 207, 209, 210, 211, 218, 220, 222, 224, 241, 242, 243, 244, 245, 251, 260, 264, 265, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 290, 291, 292, 293, 294, 295, 296, 297, 315, 321, 322, 325, 326, 348, 349, 351, 354, 359, 364, 365, 366, 367, 368, 380, 382, 386, 387, 389, 394, 397, 398, 401, 417, 440, 446, 447, 449, 457, 458, 459, 460, 466, 467, 468, 469, 470, 476, 478
 function_name 45
 functional 101, 172, 380, 445
 functions 19, 30, 45, 46, 47, 48, 56, 64, 75, 79, 90, 91, 92, 112, 113, 131, 132, 134, 137, 138, 140, 144, 181, 182, 206, 211, 213, 219, 220, 238, 241, 242, 243, 261, 262, 264, 265, 266, 269, 270, 274, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 289, 290, 291, 293, 294, 295, 297, 308, 315, 322, 346, 350, 351, 354, 358, 359, 364, 383, 384, 388, 395, 396, 400, 431, 444, 448, 449, 460, 462, 463, 466, 467, 468, 469, 477, 480
 functools 288

fundamental 22, 35, 44, 113, 115, 145, 199, 207, 264, 266, 278, 281, 421

G
 gained 304, 334
 game 20, 22, 192, 228, 352, 374, 375, 376, 377, 378, 379, 380, 381, 383, 384, 386
 gameplay 384
 gatekeeper 273
 gateway 21
 geany 169, 316, 320, 321, 322, 323, 324, 325, 326, 327, 328, 334, 335, 337, 338, 339, 340, 341, 342, 343, 466
 generate 90, 110, 283, 333, 410, 417, 425, 430, 433, 440, 441, 453, 456, 461, 477, 478, 480
 generate_squares 290
 generated 54, 340, 380, 453, 480
 geodetic 442
 geographic 442
 get_choice 192
 get_coordinates 136
 get_name_and_age 135
 get_pos 377, 379
 get_rect 377, 379
 get_value 478
 get_weather_data 457, 458, 459
 getattr 468, 469
 getcwd 50, 205, 238
 getlogger 259
 getmtime 240
 gets 21, 30, 32, 51, 53, 54, 66, 68, 69, 77, 98, 104, 146, 171, 187, 188, 200, 227, 233, 253, 297, 337, 376, 467, 472
 getsize 240
 getssizeof 118, 346, 349
 ghz 15, 16
 github 312
 global 64, 78, 174, 228, 229, 234, 278, 279, 280, 281, 283, 291, 292, 293, 294, 295, 296, 342, 343, 463
 global_value 295, 296
 global_variable 280, 281
 google 18, 224, 264, 376, 389, 461
 google_maps 223, 224
 gpio 21, 462, 463, 464, 465, 470
 gpio 463
 gpiozero 464, 465
 gpu 462
 graphic 2
 graphics 15, 54, 304, 362, 364, 366, 368, 370, 372, 382, 386, 387, 439, 462
 graphs 288, 434
 grasp 35, 95, 120
 grayscale 387
 grayscale_image 387
 grep 168
 group 25, 118, 133, 207, 208, 242, 245, 246, 247, 249, 250, 252, 311, 332, 338, 370, 377, 379, 383, 427, 447
 gui 15, 362, 368, 370, 371, 373, 384
 guido 18, 310, 311
 guis 368, 370, 434

H
 hardware 16, 21, 53, 54, 57, 274, 462, 463, 464, 465, 470
 hash 30, 46, 109, 114, 115, 121, 123, 157, 274
 hashable 132, 136
 hashed 157
 hashing 274
 hat 179, 357, 463, 464
 heap 156
 heatmap 439
 heterogeneous 94, 412, 413, 429
 hierarchical 288, 431
 hierarchy 256, 352
 high 13, 15, 19, 213, 289, 355, 461, 464, 465
 histogram 437, 440, 441, 443
 histogram2d 440, 441
 homogeneous 121, 412, 413, 431, 432
 host 223, 224, 225, 471, 473
 hr 118, 420
 html 471, 472, 473, 477, 480, 481
 http 454, 457, 459, 472
 https 248, 456
 hub 452, 462
 hyperbolic 264

I
 iana 360
 ice 414, 416
 icon 27, 28, 59, 454
 icons 26, 27, 182
 id 156, 207, 293, 417, 418, 454
 identifier 132, 156, 207, 278
 ides 52, 294, 316, 320
 idle 320
 ignore 316, 455
 ignorecase 243
 iloc 423, 425, 426
 imag 262
 image 320, 321, 377, 378, 379, 384, 386, 387, 388, 389, 390, 391, 440, 480
 imagedraw 387, 388, 389
 imageenhance 387
 imagefilter 387
 imagefont 387, 388, 389
 images 374, 384, 386, 387, 391
 imaginary_part 262, 263
 immutability 101, 107, 130, 133, 137, 138
 immutable 78, 87, 94, 123, 124, 126, 128, 129, 132, 134, 135, 136, 137, 139, 140, 156, 162
 immutable_set 126
 implementation 15, 16, 147, 148, 152, 312
 implied 2, 58
 implies 152, 396
 importerror 232, 348
 inaccessible 203, 280
 inaccuracies 391
 include 20, 23, 40, 62, 64, 66, 68, 81, 94, 107, 123, 129, 134, 135, 139, 181, 182, 189,

200, 208, 213, 228, 232, 233, 247, 258, 260, 291, 311, 316, 318, 342, 359, 434, 454, 468
 increment 87, 122, 266, 369
 indent 222, 237, 310
 indentation 19, 85, 89, 90, 222, 311, 313, 316, 323, 342
 indented 50
 index 42, 57, 69, 70, 84, 87, 88, 92, 94, 95, 102, 103, 104, 105, 106, 115, 116, 118, 123, 133, 134, 138, 139, 145, 146, 147, 150, 151, 173, 216, 305, 406, 413, 417, 418, 420, 424, 431, 450, 471, 472, 473, 477, 478, 480, 483
 indexed 95
 indexerror 149
 indices 88, 102, 115, 117, 305, 418
 inheritance 35, 38, 41, 44, 55, 56, 354
 inherited 40
 ini 342
 init 375, 376, 378, 381, 382
 inner_function 182, 291, 292, 294, 295
 inner_value 295, 296
 inode 207
 inplace 420, 421, 422, 426, 427
 input 65, 67, 79, 115, 122, 134, 144, 151, 152, 187, 188, 189, 190, 191, 192, 193, 203, 222, 223, 249, 253, 254, 255, 257, 284, 308, 314, 317, 346, 358, 363, 364, 370, 371, 372, 373, 381, 467, 468
 input_frame 371, 372, 373
 input_image 389, 390
 input_value 295, 296
 insert 103, 152, 178, 180, 199, 212, 324, 327, 400, 401, 406, 458, 479
 insert_into_file 199
 inserted103, 199, 325, 406
 inserted_array 406
 inserting 102, 199, 324, 400, 401, 405
 insight 14, 139
 insights 304, 427
 instance37, 38, 39, 43, 44, 47, 48, 56, 77, 97, 101, 109, 114, 116, 118, 123, 133, 134, 136, 162, 184, 195, 196, 201, 225, 241, 271, 282, 287, 288, 292, 304, 316, 325, 333, 346, 350, 352, 353, 354, 444, 449, 451, 461, 471, 478
 instantiate350, 357
 instantiated 357
 instantiation 47
 int 77, 188, 189, 190, 191, 192, 254, 262, 266, 409, 420, 467, 468
 int16 406, 410
 int32 406, 407, 408, 409, 410
 int64 405, 406, 407, 409, 410, 413
 int_ 406, 409, 410
 int_ 407, 408
 int_array 408, 409
 integer 63, 77, 78, 87, 95, 107, 129, 134, 156, 161, 188, 190, 191, 193, 254, 262, 269, 273, 274, 275, 276, 405, 406, 407, 408, 409, 410, 419, 420, 421, 424
 integers 115, 123, 128, 129, 133, 156, 162, 184, 254, 274, 289, 403, 404, 409, 410, 412, 413, 429, 431, 432, 433
 intel 190
 intent 448

interpolation 387, 419
 interpret 206
 interpreter 23, 29, 30, 51, 52, 54, 59, 60, 62, 168, 228, 229, 231, 234, 291, 322, 327, 341, 477
 interpreterpath 327
 interpreters 52
 intersection 123, 124, 125, 128
 intersection_result 124
 invalid 79, 188, 192, 245, 254, 255, 315, 459
 invalid_json 226
 invaluable 304, 310, 349, 370
 invoke 343
 invokes 168
 ioerror 203, 204, 388
 iot 20, 452, 470
 ip 275, 453, 471, 472, 473
 ipsum 327
 ironpython 16
 is_ 467, 468
 is_authenticated 273
 is_empty 146, 147
 is_equal268
 is_even 183, 186
 is_greater 268
 is_prime 466, 467, 468
 is_raining 272
 is_subset 127
 is_sunny 272
 is_superset 128
 isbn 2
 isinstance 79, 263
 isnull 424, 425, 427
 isolate 174, 175, 274, 461, 463
 isolated 169, 228, 327, 332, 334
 iterable 71, 84, 88, 113, 126, 139, 142, 263, 450
 iterables 140, 143
 iterdir 230
 itertools 100, 142, 143

J

java 16, 17, 320
 javascript 16, 17, 219
 jazzed 366
 join 65, 68, 70, 71, 72, 104, 205, 240, 456
 joined 230
 joining 70, 72, 230
 jpeg 386
 jpg 387, 389, 390
 json 213, 215, 217, 219, 220, 221, 222, 223, 224, 225, 226, 237, 453, 454, 455, 457, 459
 json_data 220
 json_file 221, 222
 jsondecodeerror 224, 226
 jupiter 217, 219, 414, 415, 417

K

kernel 32, 33, 464
 key 14, 19, 22, 23, 24, 28, 35, 38, 45, 50, 53, 54, 55, 76, 79, 85, 93, 94, 109, 110, 111,

112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 123, 126, 128, 130, 131, 132, 135, 136, 137, 139, 148, 150, 151, 157, 161, 175, 181, 189, 190, 193, 195, 213, 215, 216, 217, 219, 223, 224, 226, 239, 242, 259, 272, 282, 287, 296, 304, 323, 333, 342, 346, 358, 362, 402, 404, 412, 416, 418, 419, 420, 423, 430, 432, 434, 453, 455, 456, 457, 458, 459, 460, 461, 471, 477
 keyboardinterrupt 256
 keyerror 127, 226
 kwargs 76, 285, 286

L

lambda 120, 277, 281, 282, 296
 lanczos 387
 large_file 210
 large_int_array 409
 large_list 118
 last_element 150
 last_modified 205
 latitude 441, 442
 latitudes 442
 latter 316
 led 465
 leds 464, 465
 legb 291
 len 100, 103, 112, 133, 147, 150, 280, 307, 309, 347, 450
 less 82, 83, 145, 184, 231, 263, 271, 284, 290, 314, 317, 334, 429, 434, 449, 464, 467
 lgpio 464
 lib 32, 62
 lib64 32
 libgpiod 463
 libraries 17, 18, 19, 20, 21, 33, 48, 50, 57, 58, 62, 173, 174, 209, 221, 223, 228, 229, 261, 293, 310, 311, 332, 386, 396, 430, 431, 434, 435, 463, 464, 465
 library 20, 32, 48, 49, 51, 56, 57, 148, 152, 155, 174, 206, 229, 232, 314, 324, 331, 350, 362, 374, 388, 392, 393, 420, 423, 424, 430, 439, 441, 454, 459, 463, 464
 librecalc 213, 214
 libreoffice213
 lifo 145, 146, 147, 152
 linked 342, 373, 470
 Pandas 294, 310, 312, 313, 316, 317, 319, 323, 337
 linter1 314
 linux 20
 lipsum 327, 329
 lisp 18
 list_current_directory 205
 list_example 117
 list_of_lists 106
 list_string 131
 list_with_duplicates 125
 listdir 50, 205, 236, 238
 listed 228, 231, 232, 313, 384, 478
 listing 51, 206, 236, 254
 listings 5, 335
 lists 38, 82, 91, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 110, 115, 116, 117, 118, 119, 121, 123, 128, 129, 130, 131, 133, 136, 137, 138, 139, 140, 141, 142, 143, 150, 151, 155, 156, 157, 159, 160, 161, 162, 163, 164, 165, 171, 215, 216, 217, 218, 220, 240, 305, 306, 311, 326, 337, 396, 397, 413, 414, 415, 416, 417, 418, 424, 432, 449, 453
 lists16 101
 literal 247, 250, 251
 llo 248
 load 28, 30, 169, 187, 192, 197, 210, 220, 221, 223, 224, 318, 335, 343, 381, 382, 384, 386, 387, 388, 389, 390, 418, 479
 load_config 224
 load_default 388
 loaf 68
 loc 423, 424, 425, 426
 local 58, 60, 64, 78, 227, 228, 229, 278, 279, 280, 281, 290, 291, 292, 294, 295, 296, 358, 361
 local_variable 280, 281
 locale 48
 localhost 223, 224, 225
 log 23, 196, 209, 223, 225, 228, 254, 259, 260, 264, 269, 270, 273, 284, 285, 447, 455
 log10 264, 270
 log10_value 270
 log_decorator 284, 285
 log_file 259, 447
 log_file_path 223
 log_level 223
 log_person_details 447
 log_to_file 223
 log_value 269
 logarithm 264, 276
 logarithmic 269
 logarithms 261, 264, 269, 270
 logger 259, 260
 logging 223, 225, 226, 255, 258, 259, 260, 284, 285, 446, 447, 470
 logging_interval_seconds 225
 logs 33, 65, 172, 259, 260, 285, 287, 291
 long 14, 21, 41, 67, 83, 95, 163, 187, 216, 234, 279, 286, 315, 464
 loop 26, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 96, 100, 113, 122, 138, 141, 142, 143, 182, 188, 190, 196, 309, 363, 364, 365, 366, 367, 370, 371, 372, 377, 378, 379, 381, 382, 383, 460, 467
 ls 52, 169, 237, 238
 lst 450
 lstat 207
 lsvirtualenv 337

M

main_window 367, 368
 mainloop 363, 364, 365, 366, 367, 368, 369, 371, 372, 373
 mainly 15, 288
 make_ 201
 make_archive 201
 mkdirs 200, 202, 239
 makefile 324
 mask 276

masking 274
 masks 274, 276
 match 64, 68, 75, 116, 132, 181, 225, 241, 242, 243, 244, 245, 246, 247, 249, 250, 251, 293, 305, 308, 366, 394, 398, 399, 400
 match1 244
 match_dollar 250
 match_percent 250
 match_word 250
 matched 134, 242, 243, 245, 247
 matches 85, 241, 242, 243, 244, 246, 248, 249, 250, 251, 253
 matches_ 248
 matches_any 248
 matches_digits 248
 matches_either 248
 matches_email 248
 matches_non_ 248
 matches_non_digits 248
 matches_non_words 248
 matches_start 248
 matches_url 248
 matches_whitespace 248
 matches_word_ 248
 matches_word_boundary 248
 matches_words 248
 matchobject 243
 mate 23, 135
 math 48, 49, 56, 116, 188, 261, 262, 263, 264, 265, 267, 268, 269, 270, 271, 273, 275, 276, 293, 306, 368, 369, 439, 468, 469
 math_ops 56
 matplotlib 330, 332, 427, 431, 434, 435, 436, 437, 439, 440, 441, 442, 443
 matrices 304, 305, 306, 307, 308, 392, 393, 394, 410
 matrix 89, 304, 305, 306, 307, 308, 309, 393, 394, 395, 397, 398, 403, 404, 405, 410, 429, 432, 433
 max_result 263
 maxdepth 480
 mean 13, 35, 42, 53, 75, 96, 173, 281, 306, 350, 395, 396, 419, 420, 422, 425, 426, 427, 429, 430, 432, 433, 440
 mean_age 422
 mean_columns 395, 396
 mean_rows 395, 396
 mean_total 395, 396
 mean_value 396
 meana 395
 memory 15, 16, 67, 78, 94, 95, 110, 117, 118, 135, 137, 140, 145, 151, 156, 158, 159, 160, 196, 210, 274, 275, 277, 289, 290, 291, 305, 346, 398, 402, 403, 404, 409, 410, 431, 432, 462
 menu 25, 27, 28, 29, 30, 60, 62, 182, 192, 318, 320, 322, 323, 337, 342, 343, 366, 367, 372, 373, 383
 menus 26, 58, 362, 368, 372
 merge 120
 merged_dict 120
 merits 312
 meta 244, 472
 metacharacters 247, 248

metaclass 350, 353, 354
 metaclasses 353, 354
 metadata 156, 205, 206, 207, 208, 288, 476
 metaphorical 178
 method 28, 36, 37, 38, 40, 41, 42, 43, 44, 47, 49, 55, 56, 59, 65, 67, 68, 69, 70, 71, 72, 73, 75, 78, 98, 104, 105, 106, 111, 112, 113, 119, 122, 126, 128, 133, 145, 147, 151, 169, 170, 171, 174, 192, 198, 199, 200, 201, 216, 222, 226, 229, 241, 242, 260, 287, 288, 291, 334, 351, 352, 353, 355, 356, 359, 361, 363, 364, 365, 366, 369, 370, 373, 388, 389, 390, 393, 402, 406, 418, 419, 421, 422, 426, 428, 430, 444, 445, 446, 447, 449, 450, 451, 476
 methods 35, 36, 37, 38, 39, 40, 41, 42, 44, 47, 55, 56, 65, 67, 70, 71, 72, 79, 94, 111, 113, 114, 116, 121, 126, 132, 133, 136, 138, 140, 147, 159, 170, 195, 196, 200, 210, 216, 229, 283, 287, 288, 350, 351, 352, 354, 357, 383, 387, 391, 393, 403, 404, 405, 413, 419, 423, 425, 427, 428, 431, 432, 444, 446, 447, 448, 449, 450, 477
 metric 454, 457, 459
 mhz 15
 microchip 462
 microcontroller 16
 microcontrollers 16
 micropython 16
 microseconds 359
 microthreads 16
 middle 145, 199, 212, 362
 min 100, 225, 263, 426
 min_ 100
 min_length 100
 min_result 263
 mind 19, 22, 35, 59, 70, 82, 85, 95, 199, 233, 261, 262, 313, 315, 318, 320, 419
 mkdir 167, 238, 239, 336, 472, 479
 mkproject 337
 mkvirtualenv 336
 mnt 32
 mo 267, 323
 mode 181, 182, 195, 196, 198, 205, 207, 208, 209, 224, 225, 244, 382
 model 15, 16, 27, 35, 41, 114, 304, 445, 446, 464
 modelling 269
 modifiers 247
 module 30, 33, 48, 49, 51, 56, 100, 119, 142, 148, 149, 150, 153, 155, 161, 162, 167, 168, 169, 200, 206, 207, 208, 209, 210, 212, 213, 216, 219, 225, 229, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 244, 251, 259, 261, 262, 264, 265, 268, 269, 270, 276, 278, 280, 288, 291, 305, 306, 314, 315, 323, 330, 333, 344, 346, 347, 348, 349, 350, 351, 352, 354, 355, 356, 357, 358, 359, 360, 361, 363, 364, 368, 374, 376, 382, 386, 387, 389, 391, 392, 413, 439, 440, 463, 464, 466, 467, 468, 469, 476, 480
 modulo 88, 183
 modulus 262, 268
 modulus_result 262
 move 20, 22, 57, 58, 81, 167, 181, 182,

200, 239, 276, 277, 278, 326, 363, 374, 375, 376, 378, 430, 470
 mp3 381, 382
 mpl_toolkits 440
 mplot3d 440
 msgs 315
 multiplexer 474
 multiplication 101, 262, 267, 305, 308, 309, 393, 394
 multiplication_result 262
 mutability 138
 mutable 94, 102, 116, 123, 128, 129, 130, 131, 133, 135, 137, 138, 140, 156, 162, 283
 my_ 127, 233
 my_age 63
 my_archive 202
 my_custom_bin 233
 my_decorator 284, 354
 my_deque 150, 151, 152
 my_dict 111, 116, 120, 126, 131, 136
 my_directory 202
 my_house 43
 my_list 95, 102, 103, 104, 105, 106, 116, 118, 130, 138, 139
 my_list_with_duplicates 118
 my_name 63
 my_python_modules 234
 my_queue 148, 149
 my_scripts 234
 my_set 123, 125, 126, 127, 128
 my_string 131
 my_tuple 129, 130, 131, 136, 138
 my_var 333
 my_variable 293
 myapp 223
 myapp_db 223
 myclass 293, 353
 myenv 173, 229, 331
 mypy 312

N

namespace 278, 279, 281, 293
 namespaces 278, 281, 293
 nan 431
 nano 472
 narray 399, 406
 nbytes 409
 ncase 257
 ncustome 348
 ndescriptive 423
 ndetecting 424
 ndim 403, 404, 405
 ndropping 424
 nested_dict 117
 nested_list 106, 131
 nested_tuple 131
 nesting 85, 162
 new_car 446
 new_content 199
 new_directory 233, 238
 new_list 159, 160
 new_list1 99
 new_name 198, 238
 new_window 367

O

new_x1 369
 new_x2 369
 new_y1 369
 new_y2 369
 new_york 360
 newline 82, 83, 195, 214, 215, 218, 243, 244, 248
 newlines 213
 next 22, 38, 43, 46, 50, 69, 72, 78, 82, 83, 84, 93, 101, 104, 126, 146, 160, 168, 182, 187, 188, 209, 214, 218, 227, 277, 280, 290, 299, 324, 325, 326, 327, 330, 338, 341, 363, 364, 366, 368, 369, 373, 380, 382, 389, 398, 401, 404, 405, 420, 424, 426, 430, 442, 443, 457, 469
 next_year_age 188
 nextended 71
 nfailed 348
 nfetching 458
 nfiles 230
 nfilling 424
 nfinal 72
 nfirst 423
 nhandling 424
 ninja 1, 2, 3, 4, 11, 13, 15, 300, 301, 302, 303, 309, 463, 482, 484
 notnull 425
 np 392, 393, 394, 395, 396, 397, 398, 399, 400, 401, 402, 403, 404, 405, 406, 408, 409, 410, 415, 425, 426, 428, 429, 430, 433, 440, 441
 np_array 396
 npivoting 424
 nreshaped 403, 404
 nresult 399
 nselecting 423
 nshape 403, 404
 nsorting 424
 nstacking 424
 nsuccessfully 348
 nsum 403, 404
 nsummary 423
 nudged 103
 nudging 19
 null 219, 426
 num 76
 num1 254, 262, 263, 271, 272, 275, 276, 347
 numpy 15, 18, 48, 261, 305, 306, 307, 308, 331, 392, 393, 394, 395, 396, 397, 398, 400, 401, 402, 403, 404, 405, 406, 407, 408, 409, 410, 412, 413, 425, 426, 428, 429, 430, 431, 432, 433, 434, 440
 numpy_array 426, 430
 numstacking 424
 nusing 71
 nweather 458
 ny_time_zone 360

164, 165, 207, 210, 211, 219, 220, 221, 223, 229, 230, 231, 242, 243, 290, 316, 326, 343, 350, 359, 360, 361, 364, 379, 380, 389, 407, 408, 413, 444, 445, 446, 447, 465, 469
object_array 408
occupy 276
oop 35, 38, 40, 42, 44, 47, 65, 67, 113, 114, 136
open_button 367
open_garage 40, 43, 44
open_new_ 367
open_new_window 366, 367, 368
opened 211, 320, 340
openweathermap 452, 453, 454, 456, 457, 459, 460, 461
operand 257, 267, 272
operands 267, 272, 273, 274
opt 32, 116, 117
optimisation 274, 285
org 251, 252, 381, 454, 456, 457
origin 369
original_ 162
original_dic 119
original_list 155, 157, 158, 159, 160, 161, 162, 164
os 16, 19, 24, 25, 31, 50, 51, 58, 170, 171, 197, 198, 199, 200, 201, 202, 205, 206, 207, 208, 209, 227, 228, 231, 233, 234, 235, 236, 237, 238, 239, 240, 333, 346, 349, 479
oserror 205
outlines 36
28, 445, 446, 449, 450, 454, 465, 477
output_from_dict 417, 418
output_from_lists 417, 418
overflow 184, 289

P
package 57, 129, 133, 155, 168, 170, 171, 172, 173, 174, 175, 201, 324, 328, 331, 333, 392, 393, 466, 468
package_name 171, 173, 174, 175
packages 25, 32, 33, 58, 168, 170, 171, 172, 173, 174, 175, 228, 229, 232, 324, 330, 331, 344, 346, 347, 349, 392, 393, 456, 463
packed 33, 48, 140, 182, 201, 261, 276, 373
padding 366, 371, 473
paddle 374, 375, 376, 377, 378, 379, 380
palindrome 151
pandas 17, 18, 48, 330, 331, 412, 413, 414, 415, 416, 417, 418, 419, 420, 421, 422, 423, 424, 425, 426, 427, 428, 429, 430, 431, 432, 434, 435, 436, 437, 438, 439, 442, 443
parameter 47, 66, 75, 76, 88, 213, 282, 283, 287, 363, 380, 395, 410, 415, 418, 427, 428, 478
parameters 45, 56, 77, 201, 221, 223, 284, 296, 453, 455, 459, 476
parsing 196, 209, 213, 226, 358, 378, 383, 392, 407, 427, 435, 477
password 223
path 22, 199, 201, 202, 205, 212, 218, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 238, 239, 240, 318, 324, 325, 327, 337, 338, 341, 342, 346, 347, 348, 349, 381, 382,

388, 389, 390, 392, 416, 418, 468, 470, 479, 481
path1 230
path_parts 237
pathlib 229, 230, 231, 234
pattern 85, 89, 90, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 351, 352, 354, 355, 356, 383
pattern_dollar 250
pattern_percent 250
pattern_word 250
pause 27, 181, 183, 290, 325, 375, 376, 384
pca 304
pcs 52
pdb 324, 325, 326
pdf 332, 477
pdfs 477
pep 310, 311, 312, 313, 316
pep8 5
peps 310, 311, 319
permanently 228, 232
person_dict 114
person_list 447
person_log 447
pf 201
pico 16
pigpio 464
pihome 59
pil 386, 387, 388, 389, 390
pillow 386, 387, 388, 389, 390, 391
pillow_text_ 389
pillow_text_example 388
pip 168, 170, 171, 173, 174, 175, 229, 313, 314, 324, 330, 331, 332, 386, 392, 439, 456, 471, 477
pip3 171, 173, 174, 175, 374
pipe 246
pipelines 291
pipenv 333
pipfile 333
pivot 419, 420, 421, 424, 425, 427
pivot_table 420
pixel 383
plaugins 329
play 38, 57, 73, 94, 187, 375, 376, 381, 382, 384, 425
play_audio 381, 382
plot 427, 428, 435, 436, 437, 438, 439, 440, 441, 440, 441, 442, 443
plotly 435
plt 435, 436, 437, 438, 439, 440, 441, 442, 443
plug_ 32, 327
plugin 327, 328
plugins 327, 328, 477
png 386, 387, 388, 389, 390
point 13, 14, 31, 54, 59, 86, 95, 135, 136, 146, 156, 168, 180, 183, 196, 262, 264, 275, 276, 287, 290, 310, 313, 323, 324, 325, 334, 336, 367, 391, 395, 407, 408, 409
point2d 136
polymorphism 35, 38, 40, 41, 44, 448
pop 21, 23, 25, 27, 52, 95, 104, 105, 110,

111, 120, 126, 127, 145, 146, 147, 150, 151, 152, 153, 253, 362, 364
popen 238, 239
popleft 149, 150, 151, 152
popped 127
popped_element 127, 147
port 223, 225, 471, 472, 473
pos 377, 379
pprint 457
practical 59, 142, 183, 196, 272, 284, 286, 355, 421, 446, 466
prefix 211, 212
preprocessed 420
prime 466, 467, 468
prime_factors 467, 468
prime_test 468
proc 32
process_data 210, 348
process_file 210
process_order 355, 356
product 76, 184, 250, 251, 289, 309, 393, 436
product_result 285
programmatically 455
pseudo 322
psf 310
ptyhonpath 231
pull 93
push 145, 146, 147, 152
pvm 54
pycodestyle 312
pycon 311
pydaily 59, 60, 62, 167
pyflakes 312
pygame 20, 374, 375, 376, 377, 378, 379, 380, 381, 382, 383, 384
pygamesound 1381
pylint 294, 312, 313, 314, 315, 316, 317, 319
pylint_test 315, 316
pypi 57, 173
pyplot 435, 436, 437, 438, 439, 440, 441
pypy 16
python3 23, 24, 25, 27, 33, 50, 52, 58, 167, 168, 169, 173, 174, 313, 322, 330, 331, 334, 336, 338, 392, 439, 456, 463, 472, 473, 474
python_version 349
pythononpath 227, 229, 231, 232, 233, 234
pytz 331
pyvenv 62

Q

quad 15, 16
quantifiers 249
query 453, 455, 459
query_params 457
queue 145, 147, 148, 149, 150, 152, 153
queues 93, 145, 147, 148, 149, 150, 151
quotecchar 214, 218
quotient 267
R

radians 269, 270, 276, 369, 370
radius 41, 55, 143, 416, 417
ram 15, 16
randint 379, 380, 410, 429, 433

random 32, 276, 374, 375, 376, 378, 379, 380, 410, 429, 430, 432, 433, 440, 441
range 18, 48, 81, 82, 83, 84, 86, 87, 89, 90, 100, 102, 110, 118, 138, 177, 178, 179, 186, 216, 280, 286, 290, 297, 307, 312, 327, 332, 333, 349, 361, 364, 386, 406, 407, 428, 431, 433, 477
raspbian 16
raw 407, 414, 415, 419
read 19, 26, 30, 54, 85, 91, 140, 162, 163, 187, 189, 195, 196, 197, 198, 199, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 215, 219, 220, 221, 222, 223, 225, 238, 251, 287, 288, 291, 292, 295, 310, 322, 323, 354, 373, 415, 416, 418, 453
read_csv 416, 417, 418
read_csv_file 214, 215
read_number 191
read_text 231
reboot 33, 62, 336
rect 377, 379
recursive 90, 99, 288, 289, 296
redundancy 45, 56, 421
redundant 462
regex 247, 248
regexes 244
remainder 183, 268
remainders 268
remote 260
removal 114, 172
rename 198, 212, 238, 426
render_template 471, 473
repair 32
repeat 81, 90, 133, 182, 287, 406
repl 26, 243
reports 65
repositories 57, 172, 173
repository 172, 173, 312
repr 445, 446, 447
request 172, 453, 455, 457, 458, 459, 460, 461
requested 91, 459
reshape 403, 404, 405, 410, 425, 427
reshaped_array 403, 404
reshaping 419, 420, 421, 424, 427, 428
reside 338
resize 26, 363, 386, 387, 391, 398
resized_image 387
resolve 170, 184, 259, 291, 293, 389
resource 203, 253, 288, 383, 470
result_array 426, 430
result_exp 264
result_list 97
result_log 264
result_log10 264
result_log_custom_ 264
result_log_custom_base 264
result_matrix 393, 394
result_pow_math 264, 265
result_pow_operator 265
reverse 41, 73, 74, 86, 88, 134, 138, 140, 142, 151
rgb 376, 387, 388, 389
rgba 387
rmdir 238, 239

rmtree 202, 239
 rmvirtualenv 337
 root 31, 32, 56, 58, 171, 223, 228, 232, 265, 269, 276, 334, 368, 369, 370, 371, 372, 373, 468, 471, 472
 rotate_button 368
 rotate_line 369, 370
 rotated_image 387, 390
 rotatelineapp 368, 369
 round 265, 268, 269, 276, 291
 rounded 265
 rounded_value 265
 row 40, 41, 81, 82, 83, 92, 213, 214, 215, 216, 218, 221, 222, 305, 306, 307, 309, 395, 396, 398, 399, 400, 403, 404, 405, 410, 414, 415, 417, 418, 429, 430, 432, 433
 row_sum 403, 404
 row_sums429
 rpython16
 rst 478, 480
 rw 205, 206
 rwx 205
 rwxr 209

S
 s_ifdir 208
 s_iflnk 208
 s_ifreg 208
 s_irgrp 208
 s_iroth 208
 s_irusr 206, 208, 209
 s_isdir 208
 s_iwusr 206, 208, 209
 s_ixusr 208
 sans 472
 save 28, 48, 101, 156, 221, 258, 260, 275, 316, 318, 324, 326, 331, 333, 336, 341, 347, 386, 387, 388, 389, 390, 402, 404, 416, 417, 418, 430, 466, 472, 473
 sbin 32, 227, 228, 229
 scalable 18, 91, 448
 scalar 305
 scale 17, 18, 67, 383, 387, 472
 scales 79
 scikit 17, 431
 scipy 18, 48, 261
 scope 26, 27, 30, 64, 78, 175, 277, 278, 279, 280, 281, 282, 283, 291, 292, 296, 326
 score 132, 314, 374, 375, 378, 380, 381, 438
 scratch 19, 38, 41, 172, 322, 481
 screen 5, 23, 25, 26, 47, 49, 68, 182, 313, 316, 320, 333, 364, 366, 368, 370, 374, 375, 376, 377, 378, 380, 381, 383, 461, 474
 screen_height 375, 376, 377, 378, 379
 screen_width 375, 376, 377, 378, 379
 screens 383
 scribble 322
 script 20, 22, 26, 168, 174, 202, 215, 232, 233, 234, 260, 274, 278, 280, 287, 291, 314, 316, 317, 323, 324, 325, 326, 327, 336, 341, 342, 343, 346, 347, 348, 349, 382, 418, 468, 471, 474

sd 187
 sdl 374
 seaborn 431, 439, 443
 seaborn 439
 segment 195
 segments 19, 36, 241
 select 27, 28, 29, 59, 60, 61, 169, 192, 210, 320, 323, 338, 367, 372, 423, 425, 426, 452, 456, 464
 sensor 225, 464, 470
 sensor_thresholds 225
 sentence 40, 70, 71, 72, 121, 122
 sentence_extended 71, 72
 sentence_join 71
 sentence_plus 71
 sentences 65, 121
 separator 70, 71, 234
 serial 32
 set 22, 25, 27, 37, 39, 48, 58, 59, 60, 64, 71, 81, 87, 88, 90, 91, 95, 122, 123, 124, 125, 126, 127, 128, 129, 130, 133, 138, 152, 167, 169, 178, 181, 182, 183, 191, 203, 206, 207, 210, 231, 232, 233, 260, 261, 291, 317, 320, 323, 324, 327, 330, 331, 333, 334, 335, 336, 337, 338, 339, 341, 362, 363, 364, 365, 366, 369, 372, 375, 376, 377, 378, 382, 389, 444, 455, 456, 459, 470, 474, 478, 479
 set_caption 375, 376, 377, 378, 381, 382
 set_mode 375, 377, 378, 381
 set_title 441
 set_trace 324, 325
 set_xlabel 440
 set_ylabel 440
 set_zlabel 440
 setdefault 237
 setformatter 259
 setlevel 259
 setmode 465
 setrecursionlimit 346
 sets 25, 91, 119, 123, 124, 125, 126, 127, 128, 136, 172, 225, 244, 260, 274, 304, 332, 337, 366, 370, 441, 444, 445, 446, 458, 471
 setup 48, 57, 58, 167, 202, 259, 260, 262, 288, 324, 331, 343, 367, 373, 375, 376, 392, 442, 452, 459, 462, 463, 465, 471, 478
 setup_logger 259, 260
 setvirtualenvproject 337
 shadow 292, 392
 shadow_example 292
 shadowed 292, 295
 shadowing 292, 293, 294, 295, 296, 310
 shadows 292, 294
 shallow 119, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165
 shallow_164
 shallow_copy 119, 155, 159, 160, 161, 162, 163, 164
 shallow_list 158
 shut 91
 shutil 200, 201, 202, 239
 sin 265, 269, 270, 369
 sin_value 269
 sine 265, 276
 slice 102, 104, 129, 397

sliced 104
 sliced_list 104
 slicing 68, 73, 74, 102, 104, 130, 160, 161, 162, 397, 413
 sns 439
 soc 462, 463
 sort 106, 118, 120, 138, 140, 251, 252, 282, 361, 426, 456
 sort_emails_by_type 252
 sort_values 424, 426
 sorted 106, 118, 120, 206, 252, 282, 468
 sorted_by_keys 120
 sorted_by_values 120
 sorted_data 282
 sorted_emails 252
 sorted_list 106
 sorting 21, 105, 106, 117, 118, 120, 121, 138, 140, 282, 332, 360, 361, 415, 425
 source 14, 22, 54, 169, 173, 200, 201, 202, 223, 239, 291, 293, 326, 330, 331, 336, 341, 343, 351, 456, 479, 480
 source_dir 201, 202
 source_file 200
 source_folder 201, 202
 southbridge 463
 spellcheck 328
 spelling 310, 328
 sphinx 476, 477, 478, 479, 480
 sphinx_test 480
 split 65, 68, 69, 72, 121, 122, 190, 237, 241, 252, 328, 329
 sprite 377, 378, 379, 380, 381, 383
 spritecollide 380
 sprites 374, 380, 383, 384, 386
 spritesheet 384
 spritesheets 384
 sq_root 49, 56
 sqrt 49, 56, 265, 269, 293
 square 49, 56, 57, 95, 109, 115, 116, 219, 265, 269, 276, 290, 332, 342, 433
 squares 290, 378, 429, 430
 srv 32
 st_ 205
 st_atime 208
 st_ctime 208
 st_dev 207
 st_gid 207
 st_ino 207
 st_mod 207
 st_mode 207, 208, 209
 st_mtime 205, 207, 208
 st_nlink 207
 st_size 205, 207, 208
 st_uid 207
 stack 21, 145, 146, 147, 148, 152, 153, 255, 260, 289, 424, 425, 427
 stars 374, 375, 378, 380
 start_engine 114, 357
 start_heater 37
 start_time 286
 startangle 437
 statement 65, 66, 67, 69, 81, 82, 83, 84, 85, 88, 105, 111, 134, 203, 231, 255, 258, 290, 397
 std 440

str 77, 188, 191, 205, 255, 260, 293, 380, 446
 str_ 408
 strftime 205, 359, 361, 364, 365, 366, 457
 string 40, 63, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 77, 78, 79, 91, 112, 129, 131, 151, 156, 162, 188, 189, 190, 191, 192, 193, 196, 208, 210, 220, 223, 226, 241, 242, 243, 244, 245, 246, 247, 248, 249, 257, 262, 289, 358, 359, 361, 365, 407, 408, 415, 444, 445, 446, 450, 451, 473
 string_ 407
 stringvar 372, 373
 strip 79, 192, 196, 210
 sub_list 102
 subclass 350, 351, 356
 subclasses 256, 288, 350, 351, 352, 355, 356, 357
 subclassing 255
 subplot 443
 subprocess 237, 238, 239
 subtraction 96, 97, 101, 262, 266, 267, 305, 307, 308, 393
 subtraction_result 262
 sudo 24, 25, 171, 173, 313, 328, 335, 392, 456, 472, 473, 474, 477
 sum 54, 56, 197, 263, 282, 286, 287, 296, 297, 307, 314, 317, 347, 348, 403, 404, 405, 410, 427, 429, 430, 432, 433, 478
 sum_result 263, 285
 swap 102, 221
 swapcase 72, 73
 symbol 30, 46, 179, 321, 351
 symmetric 123, 124, 127, 128
 symmetric_ 124, 127
 symmetric_difference 126
 symmetric_difference_ 127
 symmetric_difference_result 124, 127
 sympy 48
 sync 477
 syntax 19, 30, 45, 74, 77, 86, 134, 177, 180, 185, 189, 247, 281, 284, 287, 291, 294, 297, 314, 322, 323, 351, 354
 sys 33, 118, 234, 339, 346, 347, 348, 349, 381, 382, 468, 479, 481
 sysfont 375, 377, 378
 system 16, 17, 18, 25, 27, 28, 31, 32, 33, 37, 51, 52, 54, 57, 58, 67, 120, 168, 169, 170, 171, 172, 173, 175, 190, 197, 205, 206, 207, 211, 212, 215, 221, 227, 228, 229, 231, 232, 235, 237, 238, 239, 240, 259, 333, 334, 335, 337, 346, 349, 382, 388, 392, 414, 452, 462, 463, 474, 477
 systemexit 256
 systems 16, 18, 20, 21, 32, 51, 58, 145, 173, 207, 219, 221, 223, 234, 304, 358, 386, 407

T

tan 269, 270
 tan_value 269
 tap 50
 tar 201
 target 138, 383
 tau 265

tau_value 265
 temp 212, 454, 455, 458, 460
 temp_dir 211
 temp_file 211, 212
 temp_max 454
 temp_min454
 temp_unit 457, 459
 temperaturelogger 225
 tempfile 210, 211, 212
 tensorflow 18
 terminal 23, 24, 29, 30, 33, 50, 51, 54, 62, 167, 169, 171, 215, 228, 232, 233, 315, 316, 320, 321, 322, 326, 330, 335, 336, 362, 363, 384, 455, 474, 480
 text_height 388
 text_to_insert 199
 text_width 388
 textbbox 388, 389
 textenv 336
 thonny 25, 26, 27, 28, 29, 30, 31, 33, 50, 54, 58, 59, 60, 62, 167, 169, 173, 177, 179, 180, 181, 182, 183, 184, 185, 205, 294, 310, 316, 318, 320, 322, 324, 330, 333, 334, 384, 392, 393, 466
 thony 179, 181, 185
 threaded 148, 150
 tick 364, 377, 379, 383
 ticker 151
 time 13, 14, 18, 19, 20, 21, 22, 28, 29, 30, 32, 35, 43, 48, 49, 52, 53, 54, 63, 77, 78, 79, 91, 95, 101, 114, 121, 123, 141, 142, 151, 152, 160, 167, 169, 171, 181, 184, 188, 195, 201, 205, 206, 207, 208, 239, 240, 283, 285, 286, 287, 290, 296, 297, 304, 313, 314, 316, 322, 324, 328, 335, 337, 338, 342, 343, 356, 358, 359, 360, 361, 363, 364, 365, 366, 370, 375, 376, 377, 378, 380, 383, 384, 398, 419, 430, 431, 435, 436, 445, 456, 457, 458, 459, 462, 463, 465, 471, 481
 timed 297
 timedelta 359, 408
 timedelta64 408
 timedelta_array408
 timer_decorator 286, 287, 296, 297
 timestamp 260, 457, 458, 459
 tin 19, 66, 269
 tk 362, 363, 364, 365, 366, 367, 368, 369, 371, 372, 373
 tkinter 362, 363, 364, 365, 366, 367, 368, 370, 371, 372, 373
 tmp 33
 tmux 474
 to_ 426
 to_csv 417, 418
 to_entertainmentroom 36
 to_numpy426, 430
 toctree 480
 toggle 273, 368, 369
 trace 182, 255, 260, 346
 traceback 255
 traits 140
 transparency 387
 transport 201
 tree 200, 201, 202, 327, 328

trig 264
 truetype 388, 389
 ttf 388, 389
 ttk 367
 tuple 71, 84, 87, 94, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 143, 144, 189, 190, 193, 282
 tuple_string 131
 typeerror 87, 96, 136, 257, 280, 350, 357
 types 16, 38, 39, 42, 52, 63, 81, 83, 93, 94, 109, 111, 116, 129, 138, 148, 155, 162, 177, 185, 192, 193, 207, 220, 251, 252, 254, 257, 277, 279, 293, 296, 342, 355, 372, 373, 406, 407, 408, 409, 410, 412, 413, 414, 419, 420, 421, 426, 427, 428, 429, 431, 432, 435, 439, 448, 471, 476
 tzdata 331

U
 uint16 407
 uint32 407, 408
 uint64 407
 uint8 407
 uint_array408
 unicode407, 408
 unicode_array 408
 union 123, 124, 128
 union_result 124
 unique 39, 40, 45, 52, 57, 75, 109, 110, 114, 115, 116, 117, 123, 125, 128, 132, 135, 136, 140, 152, 153, 156, 281, 282, 296, 312, 421, 444, 453, 455, 470, 471
 unique_keys 117
 unique_list 125
 unique_set 125
 unique_to_ 125
 unzip 142, 201, 202
 unzip_archive 201, 202
 update 24, 25, 48, 58, 126, 127, 132, 140, 171, 196, 198, 216, 221, 233, 267, 316, 327, 328, 364, 365, 366, 373, 375, 377, 378, 379, 383, 389, 422, 456, 468, 480
 update_datetime 364, 365, 366
 upper 60, 314, 316, 317
 upper_case 315
 url 454, 459, 470, 471
 urlencode 457
 urllib 457
 urls 248, 453
 usb 32, 462
 usr 27, 33, 58, 172, 227, 228, 229, 334, 336
 usual 50, 112, 253

V
 validate 188, 251
 24, 425, 427, 449, 450, 476, 478
 valueerror104, 106, 133, 178, 188, 191, 254, 255, 280
 var 33, 223, 326

variable 33, 47, 63, 64, 65, 66, 71, 73, 74, 76, 77, 78, 79, 83, 87, 91, 133, 134, 156, 181, 187, 191, 227, 228, 229, 231, 233, 234, 259, 266, 267, 273, 278, 279, 280, 281, 282, 291, 292, 293, 294, 295, 296, 297, 315, 321, 324, 325, 326, 333, 335, 364, 372, 396, 397, 400, 417, 432, 459, 460
 variable_name 326
 vector 304, 305, 448, 449
 venv 167, 168, 169, 170, 173, 175, 228, 324, 330, 331, 333, 338, 341, 343, 344, 456
 verbose 244
 version 23, 24, 25, 26, 27, 28, 33, 40, 46, 57, 58, 59, 60, 62, 73, 74, 82, 123, 126, 167, 168, 169, 173, 174, 175, 179, 184, 223, 289, 326, 328, 331, 332, 333, 334, 344, 349, 361, 388, 392, 421, 463, 479
 vert 438
 vertically 387
 via 28, 37, 62, 169, 171, 318, 333, 341, 342, 355, 356, 415
 virtual 28, 32, 33, 54, 57, 58, 59, 60, 61, 62, 167, 168, 169, 170, 173, 174, 175, 227, 228, 229, 233, 234, 314, 318, 320, 323, 324, 326, 327, 330, 331, 332, 333, 334, 335, 336, 337, 338, 339, 341, 343, 344, 374, 386, 392, 393, 439, 456, 463, 464, 471, 474, 477, 478, 481
 virtual_env 233
 virtualenv 335
 virtualenvs 336
 virtualenvwrapper 335, 336
 virtualenvwrapper_python 336
 virtualenvs 336
 void 407

W
 wav 381, 382
 weather_api 223
 weather_data 454, 455, 457, 458
 weather_history 458, 460
 weather_record 457, 458
 web 17, 20, 22, 32, 219, 221, 225, 386, 452, 454, 456, 470, 471, 472, 474, 477, 480
 whatis 326
 wheel 48
 whenever 30, 48, 64, 93, 174, 282, 330, 353
 wipe 105
 wiped 33
 word 13, 47, 63, 64, 68, 72, 121, 122, 241, 242, 243, 248, 250, 251
 word_count 122
 workon 337
 workon_home 335, 336, 337
 workspace 59
 write 13, 17, 18, 19, 20, 21, 22, 26, 41, 45, 49, 54, 64, 79, 90, 91, 92, 107, 134, 144, 193, 196, 197, 198, 199, 203, 206, 207, 208, 209, 211, 212, 213, 219, 220, 221, 222, 226, 234, 240, 244, 276, 278, 288, 293, 296, 306, 308, 313, 320, 328, 361, 374, 404, 410, 422, 432, 443, 447, 457, 464, 465, 468, 478
 write_text231, 363, 364
 writelines 196
 www 14

Wx 205
X
 x_int 262
 x_str 262
 xml 453
 xor 274
 xpos 440
 xr 206, 209
 xx 293

Y
 y_float 262
 y_str 262
 ypos 440

Z
 zerodivisionerror 254, 257
 zeros 274, 309
 zip 96, 97, 101, 140, 141, 142, 143, 201, 202
 zip_directory 201, 202
 zip_file_name 202
 zip_file_path 202
 zip_longest 100, 142, 143
 zip_name 201
 zip_path 201, 202
 zpos 440
 zsort 440
 π 264, 265



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.brucsesmith.info