



Software Engineering for Absolute Beginners

Your Guide to Creating Software
Products

Nico Loubser

Apress®

Software Engineering for Absolute Beginners

**Your Guide to Creating
Software Products**

Nico Loubser

Apress®

Software Engineering for Absolute Beginners

Nico Loubser
London, UK

ISBN-13 (pbk): 978-1-4842-6621-2
<https://doi.org/10.1007/978-1-4842-6622-9>

ISBN-13 (electronic): 978-1-4842-6622-9

Copyright © 2021 by Nico Loubser

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Spandana Chatterjee
Development Editor: James Markham
Coordinating Editor: Divya Modi

Cover designed by eStudioCalamar

Cover image designed by Pixabay

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, Suite 4600, New York, NY 10004-1562, USA. Phone 1-800-SPRINGER, fax (201) 348-4505, email orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at <http://www.apress.com/bulk-sales>.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-6621-2. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

*This book is dedicated to Kim,
who keeps me going when I feel like stopping.*

Table of Contents

About the Author	xv
About the Technical Reviewer	xvii
Acknowledgments	xix
Introduction	xxi
 Chapter 1: Editors	 1
The Different Families of Programming Editors	2
Shell-Based Editors	2
Text Editors	3
IDEs	4
The Benefits of an IDE or an Editor like VS Code	5
Installing Visual Studio Code	6
Workspaces	6
Built-In Features	7
Features to Install	9
Summary	10
 Chapter 2: Containerizing Your Environment	 11
What Are Containers?	11
The Main Components of Docker Explained	14
The Dockerfile	14
The Docker Image	15
Docker Containers	16

TABLE OF CONTENTS

Setup and Usage	16
Preparation	16
How to Install Docker	17
Creating the Dockerized Environments for Your Software's Infrastructure	19
Preparations Before You Start Cooking.....	20
First Docker Image and Container	20
Building the Image and Pushing It to the Repository	22
Pushing the Image to a Docker Repository	26
Docker Orchestration with Docker Compose.....	27
Final Docker Experiment	31
Docker Checklist and Cheat Sheet.....	34
Docker Commands	34
Docker-compose Commands.....	35
Chapter 3: Repositories and Git.....	37
A Word About Windows Git Usage and Hidden Files	37
What Is Source Control?.....	38
Additional Functionality	41
Installing Git and Creating a GitLab Account.....	41
Using GitLab.....	42
Commits.....	44
Branches.....	45
A More Advanced Use Case	49
Merging Conflicts.....	51
Removing the Need to Type Your Password Every Time: SSH.....	54
Gitignore.....	56
git stash	57
git reset and revert	58
Cheat Sheet.....	60

Chapter 4: Programming in Python	63
What Is Programming?	64
Python	65
Setup for This Chapter and How to Use It.....	65
Basics	66
Commenting Your Code	67
Variables.....	67
Sequences and Maps.....	75
Lists and Strings.....	76
Tuples	79
When to Use a List and When to Use a Tuple.....	79
Dictionaries	80
Decision-Making Operators and Structures.....	81
Operators.....	81
Scope and Structure of Python Code.....	90
Control Statements.....	91
Functions	99
Custom Functions.....	99
Classes and Objects.....	106
The Anatomy of a Class	108
Instantiating the Class.....	109
Inheritance	113
Polymorphism.....	115
Composition.....	116
Magic Methods	116
Exceptions.....	118
The Anatomy of an Exception	119
Raising an Exception	120

TABLE OF CONTENTS

Catching an Exception.....	121
Writing an Exception.....	121
Imports.....	122
Static Access to Classes	124
Cheat Sheet.....	125
Scope.....	125
Variables.....	126
Arrays	126
Control statements	127
Functions	127
Classes	128
Exceptions	129
Import	129
Reference.....	129
Chapter 5: Object Calisthenics, Coding Styles, and Refactoring.....	131
Object Calisthenics	132
1. Do not exceed one level of indentation per method. (Or rather, limit the levels of indentation as much as you can.)	133
2. Do not use the else keyword	134
3. Wrap all primitives and strings.....	136
4. Use only one dot per line.....	137
5. Do not abbreviate	140
6. Keep entities small.....	140
7. Limit classes to use no more than two instance variables.....	140
8. Use first-class collections	141
Refactoring Code	143

Coding Styles	144
Linting.....	145
Commenting Your Code	145
Maximum Line Length.....	149
Indentation	149
Blank Lines	151
Encoding.....	151
Imports	152
Whitespace.....	152
Naming Conventions.....	153
Chapter Summary	156
References	157
Chapter 6: Databases and Database Design	159
Three Things You Can Do with Data	160
Overview of Database System Components	161
Setting Up Your DBMS.....	162
Ports	163
Environment	163
Volumes.....	164
The Final Docker File.....	164
Viewing Your Database Using Adminer.....	165
Cleaning Up and Pushing to the Remote	168
Preparing Your Database.....	169
Primary Keys.....	170
Indexes	170
Index Caveats	171

TABLE OF CONTENTS

Data Types.....	171
Creating a Database.....	172
Creating the Table.....	174
Filling the Database with Data	175
Your First SQL Queries.....	175
Normalizing the Current Classes Table	180
First Normal Form	181
Second Normal Form.....	182
Third Normal Form	188
Last Word on Joins	190
Conclusion	191
Cheatsheet and Checklist	191
References.....	192
Chapter 7: Creating a RESTful API: Flask.....	193
The Project.....	194
What Is REST?.....	195
JSON.....	196
HTTP Verbs	197
REST Query Routes.....	199
HTTP Status Code.....	199
HATEOAS.....	200
The Technology You Will Use	201
Setting Up the Environment	202
Creating the GitLab Project.....	202
Project Layout.....	203
Creating the docker-compose and Docker Files.....	204
The Final Steps: Coding	216

The ORM Version	228
GET Endpoint	231
POST Endpoint	231
PATCH Endpoint	232
DELETE Endpoint	232
Takeaway of This Chapter	233
References	233
Chapter 8: Code Quality	235
Overview of Code Quality Steps	236
Automated Testing	237
Unit Tests	238
How to Run the Unit Test	242
Integration Tests	242
How to Run the Integration Test	244
A Last Issue and Some Refactoring	247
Testing the New Code	250
The Downside of Automated Testing	252
The Validity of the Tests	252
Time Pressure	253
Peer Reviews	253
Walk-Through	255
Staging Environment and UAT	256
Chapter 9: Planning and Designing Your Code	257
Software Development Lifecycle	258
Why Use a Software Development Lifecycle?	258
Steps in the SDLC	259

TABLE OF CONTENTS

Modelling	263
Where Does Modelling Fit In the SDLC?	263
Why Create Diagrams and Models?	264
Tools	265
High-Level Models and Diagrams.....	265
Low-Level Models	275
Summary.....	291
Chapter 10: Security	293
Securing Your Code.....	294
Code-Level Security	295
SQL Injection	295
Cleaning Variables	297
Keeping Errors a Secret.....	300
XSS	301
CSRF	303
Session Management.....	304
System-Level Security	305
Keep Your Systems Up to Date	305
Database Users	306
Ports	307
Docker Images.....	307
HTTPS	307
Password Policy	308
Social Engineering	309
Summary.....	310

Chapter 11: Hosting and CI/CD	313
Types of Hosting.....	314
Cloud and Serverless Technologies	314
Shared Hosting	315
Virtual Private Hosting	316
Cloud Hosting	316
Serverless.....	317
Which Hosting Technology to Choose?	317
Continuous Integration and Continuous Deployment (CI/CD).....	318
Creating the Pipeline	319
Summary.....	323
Index.....	325

About the Author



Nico Loubser is a software engineer by trade, with 16 years of experience in various industries and technologies. As an experienced team lead, he has mentored numerous developers and has developed a passion for it, which was the inspiration for writing this book. He believes that the so-called software crisis¹ can be alleviated by proper mentorship, but that mentorship is not always available. He currently lives in London, where he seeks exposure to an even greater variety of ideas, methods, and technologies in today's software development industry. He holds a post-graduate degree in software engineering from the University of South Africa.

¹https://en.wikipedia.org/wiki/Software_crisis

About the Technical Reviewer



Andy Beak is an experienced technical manager with an extensive development background and sound decision-making skills. He is the author of a cybersecurity microdegree course for the EC Council and the author of the Zend PHP study guide published by Apress. He's naturally entrepreneurial and able to zoom in on implementation details while retaining a "30,000 feet" overview of the organizational strategical context in which development occurs. An evangelist for agile working practices and delivery automation, he follows the entire development process and has a high degree of ownership for the quality of the finished product.

Acknowledgments

This book would not have been possible without all of the junior developers I have mentored over the years, as this is where my inspiration for this book originated.

I would also like to thank Andy Beak for reviewing this book, and in doing so, improving the quality of it.

Lastly, to the team at Apress who helped me produce and publish this book, thank you very much.

Introduction

Writing software is a multi-disciplinary exercise. This makes it especially difficult for people who want to learn how to create software but are without someone guiding them and helping them navigate their way between all of the technologies and methodologies there are to learn. The aim of this book is not just to teach, but also to guide the newcomer, showing where the learning efforts should be concentrated, what is good practice, and what are some of the industry standards in the current software development industry. This book bridges the divide between just writing code and creating software systems.

About This Book

This book is not just for the complete newcomer. It is also for someone who can already write code, but is interested in creating complete software projects, from inception to delivery, as well as software design practices.

As a software developer, I can wholeheartedly tell you that writing code is only a part of today's software development paradigm. In today's world, you need to have learned, and in some cases mastered, a set of specific tools, skills, and methodologies that will help you achieve your goals as a creator of software. Whether that goal is to become a hobbyist developer, whether you want to create a startup or work for a corporation, good software engineering skills are very important. Most people will pick up a book about programming, or go on the Internet and start learning how to write code. Very few people will read a book on software engineering principles, and not everyone is so lucky to start in a job where serious engineering principles are followed and enforced and where proper tools are used.

Why Are Good Practices Essential?

Certain principles in software development remain the same, regardless of which company you work for. If you consider a company with 200 employees and a 1,000,000 clients, and compare it to a company with 2 employees and 150 customers, you should notice two things. The bigger company has different problems to solve with regards to infrastructure, scalability, and keeping their code base clean with a potentially large development team. The second thing you should notice is that both companies also have similar problems to solve, such as security, keeping the code base clean, deployments, writing clean code, and using proper software engineering principles to design good code. Even if you build a website for your cousin's brake skimming business, security, proper coding principles, and architectural principles are very important.

A company that serves 100 customers a month should not have its software written in an ad hoc, shoot-from-the-hip fashion. If your complete user base, whether it is 100 people or 1,000,000 people, depends on a software product, then it means buggy code will affect 100% of your client base. No company can afford to have their client base affected to this extent.

By no means does this book suggest to over-engineer your software solutions. If you are writing software that reads the temperature in your garden into a database every minute of every day, emails you a graph every month, and your brother can log in online to check the temperature in your garden, then you don't need a supercool Kubernetes cluster on AWS. You do, however, need a clean code design that is easily modifiable, secure code, and version control. You may think no one will hack into your system, and you will be dead wrong. Not all hacking is for financial gain. Some hacking attempts are for bragging rights, which is more than enough incentive to deface your website. If a simple input field is left unprotected, like a telephone number input field, your whole database can be trashed, stolen, or corrupted.

Why Did I Write This Book?

A while ago a friend started to learn how to program. He struggled initially because some of the concepts that were covered were intended for someone with a coding background. Reviewing the literature he was using, I noticed that they also basically all excluded a comprehensive approach to creating software. I saw this as two problems. Firstly, some of the beginner material out there caters to people with some knowledge about programming. This is not the end of the world. You are all intelligent enough to put the pieces together and learn from material that is intended for someone with more knowledge. But it was this aspect that made learning more difficult for my friend. Secondly, there was also the absence of the processes to build comprehensively good systems. So I decided to create this book.

My aim for this book is to show a complete beginner the cornerstones of creating easily readable, maintainable, editable, and releasable software that can be adapted and changed as needed. I wanted to touch on the principles and knowledge needed to create great software products—more aspects of software development than just writing code. As mentioned, software engineering is a vast discipline that requires many technical skills and knowledge to create great software products.

Good software methodologies, tools, and approaches go back a very long time. Having said that, today's software development world is different from what I was generally exposed to when I started out in the early 2000s. Back then, we manually FTP'd our files to the server. Before we FTP'd anything, we would make a copy of that script on the server. It was not uncommon to see files with names such as `index_1.php`, `index_backup.php`, and `index_final_backup.php`. File management is now handled by version control software. Version control systems are not new, but I believe they are now incredibly widespread and more commonplace than ever before. I also believe they are imperative to a programming project.

How This Book Is Organized

Since the intent of this book is to teach you most of the basic aspects of creating software products, it has been designed so that the chapters build on each other.

The first three chapters look at setting up your system. Chapter 1 looks at software editors, Chapter 2 looks at setting up your software environment using containerization, and Chapter 3 looks at setting up your source control system where you can save your project remotely. These first three chapters form the basis on which you create software and what your software runs on.

Chapter 4 teaches you how to write code using Python. The work in Chapters 1, 2 and 3 contribute to this chapter. Chapter 5 builds on Chapter 4, showing you how to write better code. Chapter 6 shows how to design databases.

Taking Chapters 4, 5 and 6 in consideration, you can move on to Chapter 7, in which you build a small project using the skills you learned in the preceding chapters.

Chapter 8 teaches you how to test for code quality, and Chapter 9 looks at design concepts.

Chapter 10 looks at security issues, and we round it all off with Chapter 11, where you look at hosting your software, as well as continuous integration and deployment.

CHAPTER 1

Editors

Creating software is all about solving problems. And your software development editor is a great place to increase productivity and lessen the cognitive load you will experience while solving these problems. Within your editor lies the ability to automate some of your tasks. It will allow you to defer some tasks that would have strained your memory or would have consumed too much time, to your editor. A decent editor will allow you to optimize the layout of your screen so that your database browser, shell, and code editor are easily and readily available within seconds. It will have a large collection of shortcut keys to simplify certain actions. It will also allow you to change the look of your editor, to soften the colors, and to choose a font that is easier on the eyes to lessen eye strain.

In this chapter, we will look at the differences, and benefits, of the different styles of editors available for software development. Selecting an editor may sound like a very trivial issue, but in the end it can lead to bad decisions that can affect your productivity. When we program, we basically create a text file containing different commands. This file will not have a text file extension (.txt for example), but rather an extension indicating what language it was written in, for instance .php or .py. But it is nothing but a text file. These files containing the commands are interpreted (or compiled) and executed by your chosen language's interpreter (or compiler). Because of this, we can, in general, create our programming language's code files in almost any editor we choose, as long as we can give it the right file extension. Because we can choose almost any editor, there are many editors to choose from. This makes the decision more

complex. This chapter will highlight some of the editors available and their drawbacks and benefits.

Before we delve into our discussion about editors, just some background about why we selected the editors we did in the section below. The language we will use is called Python. It is a very popular and powerful language, plus it's easy to learn. And like most languages, you can use a myriad of editors to achieve your goal of creating software.

The Different Families of Programming Editors

There are three broad sets of editors to use to write your code, and each set is useful in its own way. Using and supporting a specific editor is normally a matter of experience. It may literally take you weeks or months to realize there is something about your editor that you just do not like. A certain editor may give you a slick modern look, but may be slow when it opens files. Or you may be forced to choose one with specific built-in functionality, like support for FTP. Editors often have quirks that will slow you down or start to irritate you as time goes by, and in many cases, you will only learn about these quirks when you test the editors yourself. You should also consider the non-programming aspects of an editor, things like background color, font types, font sizes, and font colors. Staring into a screen for hours on end is very hard on your eyes, and being able to customize your setup to lessen eye strain is important.

Shell-Based Editors

The first set of editors consists of shell-based editors like Vi, Vim, and Nano. Shell-based editors run in a Windows, Linux, or Macintosh command shell. You will get some exposure to command shells in this book, but not with shell-based editors. Shell-based editors have a

high learning curve, are purely text-based with no fancy graphical user interface, and are indispensable in certain circumstances. For instance, they're great for fixing code or putting in a temporary code fix on a remote system that has no graphical user interface while someone works on a permanent fix. In a lot of instances, if your career is going the Linux route, you will encounter Vi or Vim. You can also get Vim for Windows but I doubt you will ever need to use it. Vim is quite powerful, but can be made even more powerful if you install the plugin SPF13 for Vim. I believe that these editors have their place in software engineering, but they should not be considered your primary editor.

Text Editors

The second group is text editors like Notepad, Gedit, Atom, Sublime, and Visual Studio Code (also known as VS Code). Text editors are a cost-effective way of getting a GUI-based editor to write your code in (they vary from low-powered to very high-powered). In the case of an editor like the Windows-based Notepad, you get absolutely no features and you cannot add any features. Yet you can create files with the correct extensions which can be interpreted or executed by a programming language. I absolutely do not recommend Notepad, but I include it in the list to prove my point that you can make bad decisions when choosing editors.

Linux's Gedit is good for a quick test script, and it gives some basic features which can aid in development. Just like Notepad, I don't recommend it, unless you need to churn out a 30-line script that does something small.

Under the same umbrella as these text editors, you will also find editors that can be very powerful. Two of these editors are worthy of a mention: Atom and Visual Studio Code. Both come with a myriad of plugins and built-in features, and are customizable to a degree. It may be difficult to choose between the two, and since both are free, I feel there is no harm in you trying out both.

Atom comes across as very modern and approachable, but Visual Studio Code is elegant and in some cases boasts faster startup times than Atom. According to the website www.software.com, VS Code leads the race in the most popular free editor for software developers. But both can deliver the power you need for a perfectly free, feature-rich development experience. In this chapter, we will look at VS Code, but I will encourage you to experiment and play around with a lot of editors. This experience will help you notice certain drawbacks or benefits between editors. Personally, I like text editors, but I am not fond of searching for plugins. There are also potential speed issues compared to IDEs, such as when opening large projects or indexing your files to improve searching.

IDEs

A third option is an IDE (integrated development environment) like Pycharm, Eclipse, and Wing (to name a few). They come packed with a debugger, interpreter or compiler, web server, shell terminal, database editor, and fully fledged code editor. An IDE can also syntactically evaluate your code based on the version of the programming language you are using. Some are also integrated with different version control systems, and even keep a local history, just in case you delete something by accident. Some IDEs come at a price, but normally they are well worth it.

By default, your IDE will index your projects, making them instantly searchable. You can follow code from implementation to integration and back again. There are also many shortcuts that speed up certain actions. Granted, some of these features can be added to a text editor via plugins, but speed-wise I have not yet seen editors perform at the same level as IDEs do. On the topic of plugins, IDEs normally also come with a plugin system. I have used many IDEs. At the time of writing, I am using Pycharm, which is really hitting the spot with me. JetBrains, the company that produces Pycharm, offers a free community edition, which has less functionality than the professional edition (which has a fee) but is still packed full of features.

The Benefits of an IDE or an Editor like VS Code

Having your work environment set up in the best possible way has quite a few benefits. You will definitely increase your productivity if you get to understand aspects like code navigation and debugging. A few IDEs allow you to query your database and run your shell commands in different panes next to where you are coding. This may not seem like a big deal but it does increase your productivity. Source code navigation and code completion will absolutely increase your productivity, while the ability to step through your code line by line during execution time, and injecting data into it at runtime, are incredibly powerful tools.

Let's go back to code completion. Code completion is such a simple concept. But using code completion frees up your mind so that you do not have to worry about remembering all of the different keywords you find in programming languages, for instance, or even how to implement different program-specific aspects, since the IDE can remind you how to do them. It is great to know these aspects by heart, but remember that writing software can be very taxing on your cognitive system, and breaking your train of thought while solving a complex problem can be problematic, especially if you had to do so just because you could not remember a specific keyword. Having code completion alleviates that burden.

This is the premise on which this whole chapter hinges. Choose an editor, whether it is a text editor with the correct plugins or an IDE that takes the strain off of you having to remember simple things that the editor can just remind you about, and do automatically (or a 100 times faster) the things you did manually, and you can get on with what creating software is all about: solving problems.

Installing Visual Studio Code

I suggest that you install Visual Studio Code because it is a great editor with great features. After installing it, you will take a quick look at some of the features of VS Code. At time of writing, VS Code can be installed from the following location: <https://code.visualstudio.com/download>.

The default layout of VS Code has two sections. The left-hand pane contains the structure of your folders as well as files. The pane on the right-hand side has the code editor in it. The left-hand pane also contains your workspaces, as explained below.

To create a file, just click the File menu item and select New File. When you save this file, remember to save it with a `.py` extension in order for VS Code to recognize it as a Python file.

Workspaces

A workspace shows a project's contents. The files and folders that make up the project are visible in the workspace in the left-hand pane of the editor. It is not mandatory to have workspaces. You may just reopen the directory with your project's code each time. However, workspaces give you a convenient way to organize your projects. Creating a workspace is easy.

1. From the File dropdown menu on the top menu bar, select Open.
2. From there, open the project directory, with your code in it.
3. Once that is open in the left hand-pane in VS Code, open the File menu once again, select Save Workspace As, and give it a name.

You will see that in the left-hand pane, you have created a workspace with the name you gave it and (Workspace) after it. To reopen a workspace, you have two choices:

4. From the file menu, select Open Recent.
5. From the file menu, select Open Workspace. From there, look for your project directory, and click the file inside that directory with this name: *the-name-you-gave-it.code-workspace*.

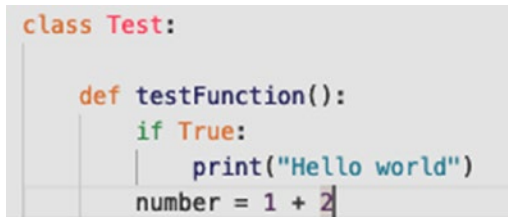
What happens in the background is that VS Code creates a file in the directory your project is in, and now considers that directory your workspace. Inside this file you will find the following text. This file is a skeleton and can be filled out to be more complete, but we are not concerned with that. It is noteworthy, though, that the path value in this file points towards your workspace. You can move this file to another location as long as you update the path value. This is good to know, but not something we are going to do now. The default values will do just fine.

```
{
  "folders": [
    {
      "path": "."
    }
  ],
  "settings": {}
}
```

Built-In Features

VS Code comes with some handy built-in features, such as syntax highlighting. Syntax highlighting is when an editor presents different words, which have specific meanings in a programming language, in

different colors. These words can also be grouped by color; for instance, specific keywords belonging to Python, even though not the same word, will have the same color because they belong to the same group called keywords. See Figure 1-1.



```
class Test:
    def testFunction():
        if True:
            print("Hello world")
        number = 1 + 2
```

Figure 1-1. *Keywords*

In Figure 1-1, you can see that specific keywords (in this case, `class`, `def`, and `True`) are in blue. The numbers (1 and 2) are in light green, and words indicating function names (we will get to functions later in this book) are in yellow.

Taking the above code into consideration, when you implement the code, you won't have the function written in front of you. You will only use the names, in this case `Test` and `testFunction`. This means that if you need to see how `testFunction` works on the inside, you need to browse to the page where test function is written. But with a decent editor like VS Code, this becomes a lot easier. The following may all be a bit hard to envision at this very moment, but once you start writing code, it will all start to make sense. See Figure 1-2. With VS Code, you can view the code written even though you are in another script by hovering your mouse over the word `testFunction()` and pressing `Shift + Ctrl`. A popup will appear with the code in it. Holding `Shift` down and clicking the name of the function will actually take you to where the implementation is written. These two small portions of functionality make it a hundred times easier to navigate a codebase and take the task of browsing out of your hands.



Figure 1-2. Code popup

Features to Install

You may find that some of the features you want are not built in. They are called extensions and they are installable via the Extensions Marketplace. To install an extension, click the four squares in the left-hand shortcut menu, as seen in Figure 1-3. This will open the Extensions Marketplace, allowing you to search for extension that can make your life even easier. In this example, I searched for “git” in the search text box, and underneath it, all of the potential extensions appeared.

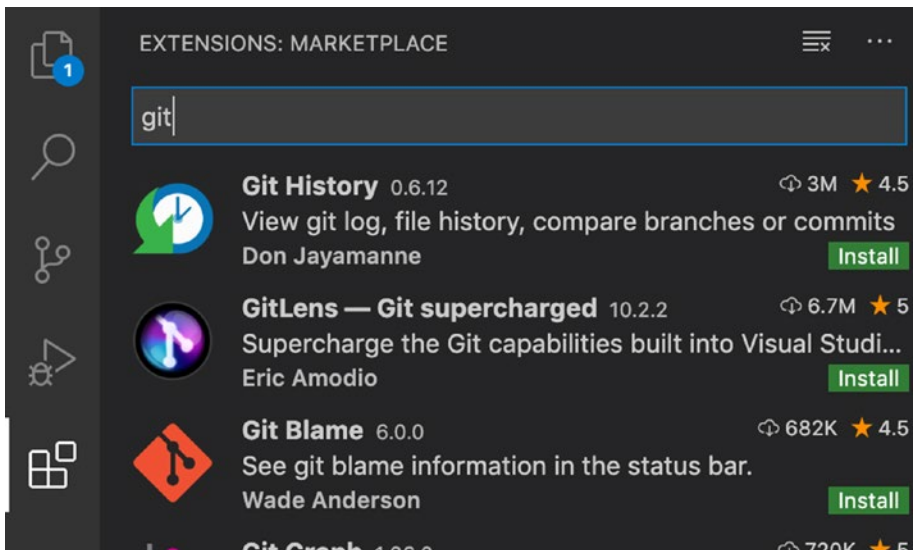


Figure 1-3. Searching for extensions

Summary

This was an easy chapter, but the subject is no laughing matter. Choosing an editor that is right for you is important, but may take some practice. I still remember how I thought the editor I used back in 2003 was the best PHP editor ever and that I would not need anything else in my life. Now, many editors and many years later, I can reflect on that simple choice and clearly see the error of my ways. You took a quick look at VS Code, but it should be enough to get you going and a good first step as you learn how to create software.

CHAPTER 2

Containerizing Your Environment

It is important to finish this chapter because the next chapter requires it.

Gone are the days of struggling to deploy your software on different machines, or having to rig and clutter your development machine in inventive ways to simulate your staging or production environment to some degree. Gone are the days of a new colleague starting to work at your company and slowly installing the development environment over a matter of a few days. Gone are the days when you deployed code and struggled with missing dependencies. Virtualization and containerization have completely changed all of this. Containerization especially has had a profound effect on how we create software. In the modern world, that new employee will just need access to your container orchestration file and to the repository with the source code. The next two chapters will deal with containerization and repositories. In this chapter, we will discuss how to implement containerization and not virtualization.

What Are Containers?

One overly simplified way to think of containers is that they create separate servers on your computer. Potentially each of these servers can be set up to serve a single purpose only, so one server can be your database and another can be your webserver. They can be set up to easily communicate

with each other. In this overly simplified explanation, you will end up with a distributed system of servers on your computer at home or at work that does not cost your machine a lot of resources. A more complex way to explain it is that containers create isolated environments for your executable code on your computer. They don't isolate the underlying operating system's kernel; it is shared between containers. This allows for the containers to remain very small, with fast start-up times of often less than a second. The containers run in a headless state, which means you will not have a graphical user interface to interact with, just the command line. The program the container serves may have a user interface. The container itself won't.

Although it is not a requirement to know how containers and virtual machines work in the background, I want to offer a quick word about containers vs. virtualization. One difference between the two technologies is that the virtualized environment offers complete isolation from the machine it is running on, in contrast with containers, which only isolate the execution environment. This makes virtualized environments much bigger and more resource-heavy than containers because they represent complete, isolated servers. They also take longer to start up. A big benefit of using containers is that because of the shared underlying operating system model, which leads to size reduction of the running containers, you can pack a lot more containers into an individual system and spin them up and down quickly.

Even though learning about containers adds a slight bit of complexity before you start to learn how to code, it is well worth the effort. The container technology we will use is called Docker, and it has had a major impact on how we develop, deploy, and distribute software. When we build containers using Docker, we refer to it as “dockerizing our application.”

Some benefits of dockerizing your environment and applications:

- You can emulate your staging and production environments a lot better.
- You can distribute your software with all dependencies available, without worrying whether they exist in your production environments.
- You can add functionality in the form of containers without needing to install that functionality locally. For instance, if you want to test or play around with MongoDB, you can just use a container instead of installing MongoDB locally.
- It vastly simplifies setting up your complete software system on other machines.

One potential drawback of containerization vs. virtualization is that containers can be less secure than virtual machines. Therefore, it is very important that you construct your Docker containers properly and not blindly use prebuilt images from untrusted sources.

A great benefit of dockerizing your applications is that you do not need to worry about whether the machine you will deploy on, be it your own computer, your friend's computer, or AWS, has the modules or software your program needs. It is all contained and assembled inside your Docker image. If you have set up your image correctly, it will yield the exact same container each and every time you use it.

The Main Components of Docker Explained

In Docker terminology, you get three main components, which we will discuss:

- A Dockerfile
- A Docker image
- A Docker container

When we dockerize an application, we, in general, work in the above order. We create a Dockerfile first. Then we create the Docker image using the Dockerfile. Then, with the Docker image, we create our container.

The Dockerfile

A Dockerfile is a type of recipe telling Docker how to build an image. You place text instructions in your Dockerfile, such as what base image to use (this can be an operating system, or an application with an operating system, or even an image you created previously), what operating system commands to run, how to share files with your local directories, what additional software to install, and so on. It basically takes a base image, and from there you can extend its functionality.

Some sample commands you will encounter in a Dockerfile will look like this:

- FROM specifies which base image to use.
- COPY copies files from the machine the Dockerfile is running on into the image.
- RUN runs various commands, such as installing extra software that is not available on the base image.
- CMD specifies a command to run once the container has started.

The Docker Image

There are two ways to create a Docker image. I will discuss the Dockerfile method in more detail, but just for the record, let's talk about the other method quickly without going into much detail.

The Step-by-Step Way to Create a Docker Image

With this way, you will start with an already existing base image. Let's assume you selected an Ubuntu image with PHP and Python installed. You will spin up a container from the image and mount the container. I will discuss mounting containers later. Now, inside the container, you can manually install and remove aspects of the container you do not need. You can, for instance, take an image that has both PHP and Python installed and remove Python. You can then install Golang. From this point, you can save it as a new image using the container as a template. We are not too concerned with using this method at the moment. We are more interested in the Dockerfile method at the moment, as you can share the Dockerfile and let anyone get the exact same results.

The Dockerfile Method

This is the method we will look at in this book. You can use Docker's `build` command and the Dockerfile in order to create the Docker image. As with the step-by-step method, you still need a base image, which you will select based on the functionality you need. But unlike the step-by-step method, you will have all of your requirements written down in a file. Docker will read the file and create your image. The image, after it has been built, remains inert and cannot be interacted with. It is basically a file, or rather set of files, containing all the steps needed to spin up your container. The Docker image can also be distributed via a Docker image repository, and this is something we will also look at later.

Docker Containers

A Docker image serves a very important function. The image is used to create a container. The container is the running software that will execute your code and react to your commands. **Note that a container has two states: stopped and running.** This is important. You can also mount a container and run programs in it as if you are in a separate server. So in short, you build an image from a Dockerfile. When you run your image, you create a container. A container can operate in more than one way. One way it can operate is as a server that awaits requests, like a webserver. In this case, the service will stay alive and can be inspected and mounted. A container like this needs to be explicitly killed. A container can also live for a very short time, where the container spins down once the software has executed successfully. In these instances, the container spins up again on a next request. It is important to know about short-lived containers and long-lived containers, and even the fact that you can create a container in a stopped state. Not knowing this can cause some confusion for beginners when they inspect their container environment and don't see any containers. They may be expecting to see a container running permanently when in fact they should have been expecting a short-lived container that only lives as long as your code executes. You will encounter both containers in this chapter and inspect them.

It should be mentioned that even though you can find many very helpful prebuilt images on the Internet, you cannot just implicitly trust them. In a high-security environment, it is advisable to build your own image.

Setup and Usage

Preparation

You will execute the Docker commands in a command-line environment. If you have command line knowledge, you can skip this preparation part. Windows, Linux, and Mac come with command-line utilities.

You only need some basic operating system command-line knowledge. The rest of the Docker-based commands will be shown during the tutorials and repeated in the cheat sheet section at the end of the chapter. The aim is to keep things simple and understandable, so you will only concentrate on the command line tools you need. Here is a list of basic commands and what they mean:

- `cd` changes to a new directory.
 - `cd ~` will always take you to your home directory.
 - `cd ..` will take you one directory back.
 - `cd /app` will take you to a directory called `app` (if it exists) in your root directory.
 - `cd ./app` (notice the dot before the slash) will take you to an `app` directory in the directory you are currently in.
- `mkdir` creates a new directory.
 - `mkdir /app` creates an `app` directory in the system root.
 - `mkdir ./app` creates an `app` directory in the directory you are currently in.
- `pwd` prints the directory path you are currently in.
- `ls` lists all the files in the directory you are in.

How to Install Docker

You need to install the Docker engine in order to use Docker. The instructions on how to install Docker are well documented on the Docker website. I am omitting the install instructions here but I do need to make a special note about installing Docker on Windows. Installing Docker on Windows introduces some issues, but fortunately there are two solutions

for you to try. This first solution is to use the Windows subsystem for Linux. This is the preferred solution to this problem. It allows Linux containers to run natively on Windows. Instructions on how to install it can be found at <https://docs.docker.com/docker-for-windows/wsl/>. The second solution is to use the Docker toolbox. At the time of writing, the toolbox lives at https://docs.docker.com/toolbox/toolbox_install_windows/. One step that I disagree with in the website's install instructions that the website suggests to just accept the default settings as provided by the installer. The installer will ask you whether you want to install VirtualBox. Make sure you select this option as well, because by default it is set to Off. This system will run in VirtualBox and therefore it is mandatory that you select it. You can always attempt to install Docker directly on your Windows computer, as that is supported. But it is only supported for the very latest versions of Windows.

Important for Windows Users

Once you have installed the Docker toolbox, you will have a **Docker quickstart terminal** link on your computer. Always use this link to start up your Docker environment and allow it a minute or so to boot up. This runs in a virtualized Linux environment and gives you a nice Linux command line environment to run your tests in. It will also allow you to use all the commands in the book, as they are Linux-based.

The Docker website does a great job explaining how to install Docker, and including the instructions for all possible operating systems in this book may detract you from what this chapter is about, which is using and understanding Docker. The website can be found at <https://docs.docker.com/>. Select your operating system and install the software according to the instructions.

Creating the Dockerized Environments for Your Software's Infrastructure

Once Docker is installed, you should be able to create images and containers. There are many ways of creating your software's underlying architecture with Docker, and normally a good amount of thought must go into designing your infrastructure. For your purposes in this book, you will have an image for Python, an image for your database, and an image to browse your database with.

But it is not necessary to have a different image for each aspect of your architecture. All three technologies can easily live in one image. However, there are some good benefits to having multiple images. Simplified deployments and improved resource management per container are two bonuses. You can also completely switch off, for instance, your database container, while letting the webserver run and serve up a maintenance page. This is already a lot better than having both services run in one Docker container and having to switch off the complete system. Whereas it will certainly annoy some of your customers, it will not cause them to think your company does not exist anymore, or that you do not have a grasp on the technical aspect of your company.

The main reason you will be using different images in this book is not an architectural decision. It is to get a better understanding of Docker orchestration software and get a better feel for networking between Docker containers.

Preparations Before You Start Cooking

As part of this chapter you will push your image to the Docker Hub registry. This registry serves as a place to store and distribute your Docker image.

- In your web browser, go to <https://hub.docker.com/> and sign up for a free plan.
- You will see a Create Repository button.
- For this exercise, name the repository `python-docker-tutorial`.
- You can keep the repository set as public.

Two points to remember:

- The repository name will be prepended with your Docker login name. If your username is *abcde*, then your repository name will be *abcde/python-docker-tutorial*. You will use your username, password, and repository name later in the chapter.
- Your Docker engine needs to run for any of the Docker commands to work.

First Docker Image and Container

In this section, you will create a very simple Python application server. You will create it using a Docker file.

Create a directory in your home directory (or wherever you can easily find it) and call it `software-projects`. Inside that directory, create a file called `Dockerfile`. The file can be called anything, but let's stick to the defaults for now. This file does not have a file extension. It is just called `Dockerfile`. If you are using Sublime, you should be able to create your `Dockerfile` using Sublime.

Add these three lines into your Dockerfile:

```
FROM python:3.7.5-slim
RUN python -m pip install DateTime
RUN apt update && apt -y install vim
```

`FROM python:3.7.5-slim` indicates what base image should be used. This will correlate to an operating system (normally very trimmed down) with some selected programs. In this case, it is Python version 3.7.5. The version number can be omitted, but you're including it to tie the service down to a specific version. Without it, the service will install a later version of Python the next time you build your image or when someone else builds your image. This can cause unpleasant surprises if you are using libraries that are incompatible with a newer version of Python. This image must live in the Docker image repository. Note that you want your image to be as slim as possible. Consider the following example command:

```
#Example
FROM ubuntu:18.04
RUN apt-get install Python
```

Now you will get the larger Ubuntu operating system, which will just fill your image with components you don't need, making it heavier to distribute. Instead, start minimal and install what you need along the way.

`RUN python -m pip install DateTime` indicates the system commands to run after the image has been installed. Both `pip` and `apt` are package managers. So in this example, you are running `pip`, which is Python's package manager and installing a Python date utility.

`RUN apt update & apt -y install vim` first updates the local package lists for your operating system and then installs a package called Vim. `-y` means that every time during the install process, when the system prompts for a yes or a no, the install process will respond with a yes.

Building the Image and Pushing It to the Repository

Make sure you are in the directory where the Dockerfile is located. Also, refer to the steps in the “Preparations Before You Start Cooking” section. Run the following command in a shell:

```
docker build -t your-user-name/python-docker-tutorial:v1.0.0 .
```

Explanation of the command:

- `docker` runs the `docker` executable. Note that the Docker engine still needs to run for this to work.
- `build` tells Docker it is going to build an image.
- `-t` means you are going to name and tag your image.
- `your-user-name/python-docker-tutorial:v1.0.0` is the actual name and tag.
- `.` is a full stop and it means the Dockerfile is located in the directory the command is being run in and that it is using the default name for a Dockerfile.

When you run this command, you should see something like Figure 2-1.

```
Sending build context to Docker daemon  2.048kB
Step 1/2 : FROM python:3.7.5-slim
--> 9f4008bf3f11
Step 2/2 : RUN python -m pip install DateTime
--> Using cache
--> bc30897ffbd0
Successfully built bc30897ffbd0
Successfully tagged nicoloubser/python-test:v1.0.0
```

Figure 2-1. Docker build output

To verify that you have a tagged image, use the following command in your shell:

```
docker images
```

This command lists all of your images, and you should see output like this:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
nicoloubser/python-test	V1.0.0.0	Bc43827bd	1 hour ago	188mb

Now that your image is created and tagged, you have a few options of what you can do. So firstly, you know that the `docker build` command builds an image from the Dockerfile. An image is inert and needs to be used to create a container. As mentioned, a container can be in a stopped state or a running state. So now you have two commands to achieve those two states:

- `docker create` creates the container from an image and prepares it for running but does not run it.
- `docker run` creates the container from an image and runs it.

After running either of these commands, you will have a container on your computer. Whether the container is running or stopped is not just dependent on the create and run command; it is also dependent on what the container is supposed to do. Some containers are long-lived and some are short-lived. Some are designed to finish executing in less than a second. They live only as long as it takes the software on it to run to completion. And some are long-running services that live as long as the container is not stopped on purpose, or crashes due to some system failure.

You will use the `run` command. This will build the container. Name it `my_service` and run it.

```
docker run --name my_service your-user-name/python-docker-tutorial:v1.0.0
```

You will notice how nothing really happens. You will fix that soon. For now, I want you to inspect the containers on your system. This command will show all running containers on your system. If this is your first time with Docker, I suspect the output will be clean, with only the headers of the empty columns as output:

```
docker ps
```

Now run the following command. The `-a` flag means *show all*:

```
docker ps -a
```

If everything went according to plan, you should see something that resembles the following:

Container ID	IMAGE	COMMAND	STATUS	NAMES
Db9803bc	nicoloubser/python-test:v1.0.0	Python3	Exited 2 minutes ago	my-service

You now have a stopped container on your system. The first command shows only running containers, and the second command shows all containers, regardless of their stopped or start state.

Remember the name you gave your container when you ran the `run` command? You can start your container using that name:

```
docker start my_service
```

Once again, there is no output when you run this command, even though your environment has successfully executed. Let's go through the steps to make the image a bit more interesting.

You will do this in two ways. This first is through Docker commands, and the second way will be using a Docker orchestration tool, which you will study in the next section of this chapter. For this exercise, you will create a very small script that prints out the words "Hello world, from python." You will write the one-line script in Python. For simplicity's sake, create a directory called `test` in the same directory as the one where your Dockerfile is located. This will make it easier to have everything in the same directory for now.

Open your editor and create a Python script called `hello.py`.

Inside the file, add this line:

```
print ("Hello world, from python")
```

At this stage, you cannot execute this script on your local machine, unless you already have Python installed on your computer. But you are interested in executing it inside your containerized environment. Make the following changes to your Dockerfile. Add these two lines after the first three lines of your Dockerfile. Make sure that the first path in your `COPY` command exists. If you have followed the tutorial verbatim, it should.

```
COPY ./test/test.py /home  
CMD ["python", "/home/test.py"]
```

Explanation

COPY: You copy your test script, which is in the `test` directory, into your images `/home` directory.

CMD: Once the container is running, it will execute the parameters between square brackets that you pass to `CMD`. Between the square brackets, `"python"` refers to the application that will do the execution, and `"/home/test/test.py"` refers to the script that will be executed.

Your Dockerfile should now resemble this:

```
FROM python:3.7.5-slim
RUN python -m pip install DateTime
COPY ./test/test.py /home
CMD ["python", "/home/test.py"]
```

Because you reconfigured your Dockerfile, you need to rebuild it. So rebuild the image. Also, change the version number.

```
docker build -t your-user-name/python-test:v2.0.0 .
```

Let's create your container in a stopped state and give it the name `hello-test`:

```
docker create --name hello-test your-user-name/python-
test:v2.0.0
```

Now, let's run it! Add the `-a` flag in order for the output to be printed to the screen:

```
docker start hello -a
```

Voila! You should now see the sentence "Hello world, from python" printed on your screen.

Pushing the Image to a Docker Repository

At this stage, you have your Docker Hub username and password, which you set up at the beginning of the chapter. You will use these account details to push your image from your local machine to the Docker repository. This action will happen from the command line.

Using your command-line utility, type the following command. This will allow you to log into your Docker Hub account. When you press Enter,

it will ask for your username and password. This is the username and password you provided when you first set up your account.

```
docker login
```

Once logged in, you will be able to push your image to the repo using this account.

```
docker push your-user-name/python-test:v2.0.0
```

If you log into your Docker Hub account now, you will see your image, tagged and ready to be used!

What I have shown you is more of a way to dockerize your application than it is a way for development and testing. This will be a bit cumbersome for testing. A drawback of this technique is that you need to rebuild every time you make a change in your code. In the next section, I will show you a very valuable tool called Docker Compose. You will use it to manage your containers and to aid a little in setting up a development environment.

Docker Orchestration with Docker Compose

Where Docker is a container technology, Docker Compose is a container orchestration tool. It is used to run and maintain multiple Docker instances that work together as one, and it allows you to orchestrate all your containers from one single setup file. A lot of applications will need multiple services, like a database, web server, and application server like Python in order to execute their software successfully. I believe that Docker Compose simplifies things a lot. Also, I find the Docker Compose file very easy to maintain and read.

Installing Docker Compose

As with Docker, I will leave the setup instructions up to the Docker website to explain. The website does a great job, and once again, it allows the book to do what it is intended to do: to show you how to use the tools. Here is the link to the install setup: <https://docs.docker.com/compose/install/>.

Docker Compose Explanation

Docker Compose's task is, among other things, to build images and maintain, spin up, and spin down your Docker containers. Whereas a single Dockerfile helps you build one image, Docker Compose manages the creation of multiple different images. You will create a `docker-compose.yml` file, which is very similar to the Dockerfile, consisting of commands on how to build images and run your containers. You can also build an image from your Dockerfile within your Docker Compose file. This is what you will do in this tutorial. The `docker-compose` file is in a format called YAML. YAML is a layout that is especially useful in configuration files. Your `docker-compose` file for this tutorial will be super simple. You will use your current Dockerfile and add it to your `docker-compose` file. Then you will share the local folder where the code lives between the local folder and the Docker instance. And at this point, you will be able to edit code and spin up your container with the edited code, without rebuilding the image. In subsequent chapters, you will add more complexity and services to your Docker file.

You can create your `docker-compose.yml` file in most editors. You should create this file in the same directory as your Dockerfile.

Add the following code to the `docker-compose.yml` file:

NB: A hint about YAML files: indentation matters! Your file won't be interpreted if you have your indentation wrong. Indentation can be two spaces.


```
-----  
version: "3"  
  
services:  
  python-dev:  
    container_name: python-dev-container  
    build:  
      context: .  
      dockerfile: Dockerfile  
    volumes:  
      - ./test:/home  
-----
```

This file can be interpreted as follows:

- `python-dev`: Name of the image. You can use this name to spin up the container.
- `container_name`: Name of the container.
- `build`: Gives information about how to build the image. In this instance, you are building it using a Dockerfile.
- `build>context`: Location of the Dockerfile. A full stop means the directory the docker-compose file is in.
- `volumes`: Here you map your local folder, `./test`, up to a folder in your Docker container, `/home`. The colon is the separator. Remember that `./` refers to the local directory and `/` refers to the root directory.

With regards to the volumes directory, remember that in the Docker file you have this command:

```
CMD ["python", "/home/test.py"]
```

CHAPTER 2 CONTAINERIZING YOUR ENVIRONMENT

Since the `test.py` file is located in the `./test` directory, and the `docker-compose` mounts that directory in its own `/home` directory, the CMD `["python", "/home/test.py"]` will execute perfectly.

Next, make a change in your `./test/test.py` file. Add this additional line underneath the current line of code:

```
print ("First change.")
```

Save this change to the file. Now build your image first and create a stopped container using `docker-compose` using the following command. You build the images using the `docker-compose build` command:

```
docker-compose build
```

The following command creates a stopped container:

```
docker-compose up --no-start
```

If you run one of the following commands, you will see the stopped container:

```
docker-compose ps , or docker ps -a
```

Now run the following command:

```
docker-compose up
```

You will see the changes you made to the `test.py` file. You may also realize that this is not so special. Since you ran the `build` command, the image was rebuilt and I promised you no more frequent rebuilds! You would be right. Now go to the `test.py` script, and add another line to it. For instance,

```
print ("Second change")
```

Now without building the image, just spin the container up:

```
docker-compose up
```

or

```
docker-compose up python-dev
```

You will see the changes to the `test.py` reflected without rebuilding the image. This is very useful especially in instances when you are running a web server, which you will do later in the book.

Final Docker Experiment

In order to prepare you for chapters later in this book, I want you to create a long-lived Docker process in order to mount it. This will be very easy. It consists of five steps.

- Two temporary changes, which you will revert after this process is done
 - Adding a specific command to the `docker-compose` file
 - Commenting out a command in the `Dockerfile`
- Rebuilding the images using Docker Compose
- Spinning up a container
- Inspecting the container

Let's go through the steps.

In your `docker-compose` file, add the following directive, marked here in bold:

```
version: "3"
```

```
services:
```

```
  python-dev:
```

```
    container_name: python-dev-container
```

```
    build:
```

CHAPTER 2 CONTAINERIZING YOUR ENVIRONMENT

```
context: .
dockerfile: ./Dockerfile
volumes:
  - ./test:/home
tty: true
```

Change your Dockerfile to look like this:

```
FROM python:3.7.5-slim
RUN python -m pip install DateTime
COPY ./test/test.py /home
# The hash is meant for commenting and commands next to it will
be ignored
#CMD ["python", "/home/test.py"]
```

Now run the following commands:

```
docker-compose build
docker-compose up -d
```

Notice on the last command you added a `-d` flag. This allows the Docker command to detach from this shell terminal and run in the background so that you can keep on using your terminal.

Then, you run either of these two commands:

```
docker-compose ps
docker ps
```

At this stage, you can also mount the container or execute code that is inside the container.

To mount the container, you need to know the name of the container. This you get by running the `docker-compose ps` command. But in fact, you already know the name is *python-dev-container*. To execute code in your running container, you use this command:

```
docker exec python /home/test.py
```

Remember that in your Dockerfile you copied your Python script to `/home/test.py`. You specify this as a parameter when you run `exec` on your container. When you run this command, you should see the text you added to the `test.py` script printed to your screen.

In order to mount the container, use this command:

```
docker exec -it python-dev-container bash
```

`-it` means you will get an interactive input/output screen, and `bash` means in a bash terminal.

Once inside the environment, type

```
ls /home
```

You should now see `test.py` printed to your screen. This means you have successfully mounted your container, and you can see the file inside it!

Both of these commands will now show a running container. You can kill this long-running container in various ways. One of the commands you can use is

```
docker-compose down
```

Lastly, do not forget to revert the changes you have just made to your `docker-compose` file and Dockerfile.

This guide should have given you the knowledge to start using Docker in a capacity where it is useful. Docker has a lot more to offer, but what I have shown you is a great starting place. You've learned commands you will use quite often. As a reminder, I include a cheat sheet of the commands.

Docker Checklist and Cheat Sheet

Check

- Is your Docker engine running?
 - You can check using this command: `docker info`.
- If you are using Windows and couldn't do a normal Docker install (you had to use the Docker toolbox), are you using the Docker quickstart terminal?
- Do you have a Dockerfile?
- Do all of the paths to files in your Dockerfile and `docker-compose` file exist?

Docker Commands

`docker login` logs into the remote Docker Hub system.

`docker build` creates the image.

`docker create` creates the container from an image and prepares it for running but does not run it.

`docker run` creates the container from an image and runs it.

`docker start` starts the container. Add `-a` if you need the output to be printed to screen.

`docker images` shows all images.

`docker ps` shows running containers.

`docker ps -a` shows running and stopped containers.

`docker rm` removes a container.

`docker rmi` removes an image.

`docker exec -it 'name/id of container' bash`
mounts the container.

`docker exec 'name/id of container' 'command to run'` executes a command in the container.

Docker-compose Commands

`docker-compose build` builds the orchestration environment.

`docker compose up` runs the environment.

`docker-compose up -d` runs the environment but detaches from the terminal.

`docker-compose up --no-start` creates the container but does not run it.

`docker-compose down` brings the environment down.

`docker-compose ps` shows all containers relating to the orchestration environment.

CHAPTER 3

Repositories and Git

You cannot build great code using Docker alone. You still need a convenient way to distribute your code, to revision control it, and to source control it. Years ago, in less technologically inclined working environments, you would FTP your code to your server. When you needed to make changes to the code, you made backup copies of it yourself. Two software developers could not work on the same script independently, because of the very real danger that they would overwrite each other's work when they copied the same script to the same server. The same went for working independently on complex systems, where the thread of files you edited eventually crossed paths with the thread of files your colleague edited. In my very first software development job, the question "Are you working on file so-and-so?" was a very real, and very often asked, question. It's not that source control didn't exist back then, it was just not widely adopted. Nowadays it is almost unthinkable that a company won't have source control. Online source control services provide the ability to automatically deploy code, a very valuable feature which we will also look at later in this book.

A Word About Windows Git Usage and Hidden Files

In this chapter, you will come across something called hidden files. Hidden files are files prepended with a full stop (for instance, `.myfile.txt`), which means they cannot be seen by standard file browsers. You need an editor

that can see hidden files, or you can use your command line editor to view them. If you are using your Docker emulators command line (for instance, the quickstart terminal), which emulates Linux, then this command will help you see hidden files:

```
ls -ltra
```

What Is Source Control?

Source control is the practice of managing as well as tracking changes to your codebase. We will use Git as source control. These are two very important concepts to understand. Managing changes to your codebase with source control allows for multiple software developers to work on the same files simultaneously. The basic gist of it is as follows. You have a remote location where all your source code is stored. The code is saved and shared from this location. Then you have a local location where you develop code. This will be your own computer. Technically, in Git terms, there is no difference between the remote location and the local location on your computer. These locations exist in your code repository (repo).

Within a repo exists at least one branch (but usually multiple ones). Only one branch is active on one repo at a time, but you can push any branch to the remote repo regardless of which branch is active. A branch is a representation of your code in a specific state of work. In simple terms, a branch is your working code environment. Let's say your main branch, which will get deployed to your server, is called master. Using the master branch, you can create other branches from which you will develop your code, and eventually you will update the master branch with your other branches. These branches isolate your development phase from the master deployment branch. See Figure 3-1. Effectively you can create a branch called feature 1, which will add credit card payments to your system, and your master branch remains untouched until you decide to merge your add_cc_payments branch into the master branch. Leaving the

master branch untouched allows someone else to create another branch, called feature 2 of master, without any problems. By separating your branches in this fashion, you get at least three great benefits.

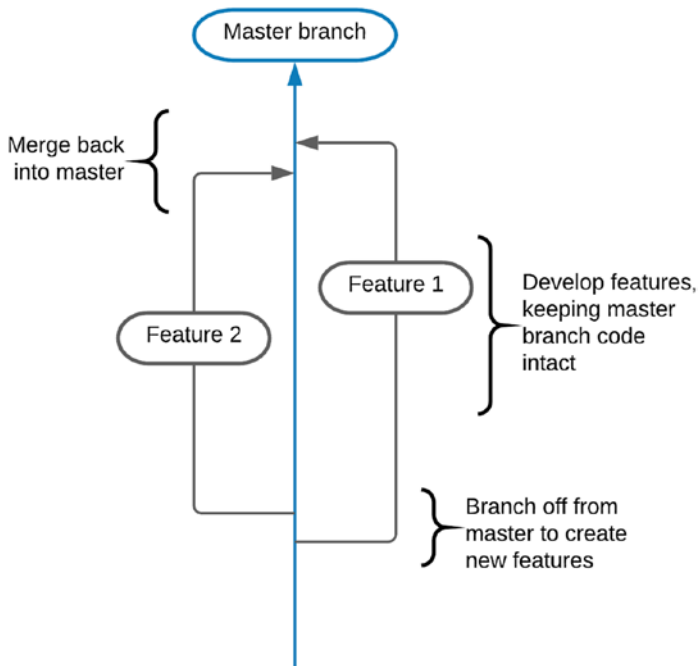


Figure 3-1. Illustration of branching

- The master is untouched, so your colleagues can use it to create their development branches, in order to create new features.
- With master untouched, it means it is always deployable. You can do bugfixes in master without worrying that someone is still developing a feature that is only halfway finished.

- Because you are in your development branch (also called a feature branch), you can push it to the remote repo where everything is stored and ask a colleague to pull it in order to review without breaking any of his work.

Your source control system will handle the merging process, where developer X's files and developer Y's files get merged together into one file. The system is intelligent enough to know how it should merge the different work into one file, because it timestamps each change in each line of code. This process is not always without problems. When two people touch the same line of code, you will get a merge conflict, or even worse, a possible silent rewrite of your colleague's code. This must be resolved manually. We will look at resolving merge conflicts later in this chapter.

The ability to track changes means it keeps a complete history of all the changes made to the source code. This allows you to revert back to past work, as well as see who made changes to the source code. This allows you to compare, side by side, what changes were made in the code. It also allows you to revert back to previous work or to selectively pick pieces of work out of other branches.

Source control in its functional capacity is used in order to manage the code changes made by software teams. It automates the merging of different people's work and diminishes the potential for overwriting other people's code, while retaining memory of who did what and when. Unfortunately, the feature to see who changed lines of code in Git is called *blame*. This feature should be used to find out who the person was so that you can ask them why it was done that way. It's very important to remember that egoless programming is great and publicly name shaming creates a really bad office culture.

Additional Functionality

You get some great functionality with source control software. You get the ability to run your unit tests and style tests once you have pushed your code to the remote repository. You can automate your deployment to your staging environments as well as to your production environment. This is called continuous integration and deployment, and we will cover them in Chapter 11. Another great feature is a pull request. Decent software teams work with a process called peer reviews. A peer review is when developer A asks developer B to review the work done for accuracy.

Note Every test or review to see if code is bug-free. You test to see how well it does what it is intended to do, and hopefully you find bugs along the way.

A pull request assigns a feature to be reviewed to a reviewer, and this can be viewed from the online application that hosts your Git repositories. The reviewer can add comments in the source control system, inline with the code where the potential issue has been spotted. You can also merge on the remote repository's side instead of on your local machine. We will go through these steps, apart from continuous deployment, in this chapter.

Installing Git and Creating a GitLab Account

There are a few companies with Git-based online tools to host your repositories that you can use. We are going to use GitLab. At the time of writing, registering with GitLab can be done at this link: https://gitlab.com/users/sign_up. This is a very straightforward process and it has a free option, which you will choose for this exercise.

Next, you need to install Git locally on your development computer. At the time of writing, the install instructions can be found here: <https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>.

If you are using Docker quickstart for Windows, you should have Git already. The best way to test is to just type `git` or `git --version` in the command line editor.

Using GitLab

To start out, you will log into GitLab and click New Project. Fill in the project name. Make it `TestProject`. On the following screen, GitLab will show the instructions you need to follow to initialize the repository on your local machine. Here is what you are going to do.

Using the command line, go to the directory where you saved your Docker instances. You will create your repo using the work you did there. If you type the following in your command line, you will see your Dockerfile and the `docker-compose` file:

```
ls
```

If you are in Windows and not in the Docker quickstart, you type

```
dir
```

Once in the desired directory, run the `git global config` commands, as provided by GitLab:

```
git config --global user.name "name surname"
git config --global user.email "your_email_address"
```

GitLab will give you some instructions to use, as shown in Figure 3-2. Select the option called **Push an existing folder**. You are going to take your Dockerized environment and add it to your repository and push it to GitLab.

Command line instructions

You can also upload existing files from your computer using the instructions below.

Git global setup

```
git config --global user.name "nico loubser"
git config --global user.email "nicoloubser@gmail.com"
```

Create a new repository

```
git clone git@gitlab.com:nicoloubser/test.project.git
cd test.project
touch README.md
git add README.md
git commit -m "add README"
git push -u origin master
```

Push an existing folder

```
cd existing_folder
git init
git remote add origin git@gitlab.com:nicoloubser/test.project.git
git add .
git commit -m "Initial commit"
git push -u origin master
```

Push an existing Git repository

```
cd existing_repo
git remote rename origin old-origin
git remote add origin git@gitlab.com:nicoloubser/test.project.git
git push -u origin --all
git push -u origin --tags
```

Figure 3-2. *Gitlab command line instructions*

Here are the commands you should follow, and their explanations.

This command initializes your local repository:

```
git init
```

This command lets your local repository point towards the remote repository. You should use the command that GitLab provides.

```
git remote add origin git@GitLab.com:nicoloubser/testproject.git
```

This command will add everything in the local repository to your local git index:

```
git add .
```

This command takes everything that was added in the above command and adds it to your local repository:

```
git commit -m "Initial commit"
```

This command pushes your work to your remote repository. This command will ask you for the username and password that you use to authenticate with GitLab:

```
git push -u origin master
```

After you have run these commands, go back to GitLab and go to your project overview. You should now be able to see the work you have pushed. All safe and sound from accidental deletion, and ready to be pulled from anywhere.

A WORD ON WHAT WAS PUSHED

It is important to note that you have only pushed the Dockerfile and docker-compose file (a.k.a. the recipes to create your images). You did not push a created image to GitLab. That you push to Docker Hub. In the future, you can pull this code to a new computer and run the `docker build` commands on it to create the images. What you have here is a very basic dockerized application that is stored on GitLab, ready to be distributed.

Commits

Whenever you commit work, you create a commit hash. You will have a long history of commit hashes, each pointing to when you committed some work. Your HEAD, which is where you actively are in your code, will point to the latest commit hash. With commit hashes, you can check out older work, compare current work with older work, and roll back to previous versions or even selectively choose files from older work linked to commit hashes.

Branches

One of the main aspects of Git is branches. If you run the following command, you will see the branch you're currently in. This will only list local branches.

```
git branch
```

You should be in the master branch. You will change to a feature branch, edit some code, add it, and then push it. The act of changing into another branch is called *checkout*. In this tutorial, you will also compare your master branch with your feature branch.

Open your terminal and make sure you are in your working directory. Type the following command:

```
git checkout -b count-to-ten
```

This command creates a new branch called count-to-ten and checks out that branch. **The -b flag creates the branch before the checkout command runs.** Run the following command:

```
git branch
```

You will see you are in the count-to-ten branch. To move back to master, you just need to type

```
git checkout master
```

and back to count-to-ten

```
git checkout count-to-ten
```


You do not need the `-b` flag because the branches exist already. Now, once you are in the `count-to-ten` branch, you will make a little change to your code. Open the test script in the `test` directory. Below the current text, add the following:

```
for counter in range(1, 11):  
    print(counter)
```

This is a structure you will encounter often in programming. Although not part of this lesson, here is what it is basically doing. `range(1, 11)` generates numbers between 1 and 11. You will see that it prints numbers from 1 to 10. This we will discuss in the next chapter.

The first thing you will do is inspect the changes you made. One way to get a high-level overview is the following command:

```
git status
```

This will show you all the files that have been changed. It's a very handy command indeed. But you can go one better. Type

```
git diff
```

This will show you the actual lines that have changed. You should see something like this in red (assuming you did not need to change the `Dockerfile` and `docker-compose`, which will give you even more feedback):

```
+for counter in range(1, 11):  
+    print(counter)
```

This shows the lines you have added. The plus signs at the beginning of the lines indicate that they have been added (likewise, minus signs indicate lines that have been removed). This data is generated by comparing the current state of your software (your uncommitted changes) with the latest commit. A commit can be seen in layman's terms as the last point of saving and preparing your work before sending it to the repository, but can also be the latest point of work saved by someone else before you

pulled the repository to your computer. Let's demonstrate how commit affects a diff or status command.

Just a reminder, in the previous chapter, you added `tty: true` to the `docker-compose` file. You must remove it if you haven't yet. You also added a hash in front of the `Dockerfile`'s `CMD` command. You must remove this hash.

Type the following two commands. You may recognize them from earlier.

```
git add .
git commit -m 'Added the count to ten code'
```

You should see something resembling the following output, and this means you have added another commit point:

```
[count-to-ten d6acbc2] Added the count to ten code
3 files changed, 5 insertions(+), 2 deletions(-)
```

Now, if you run `git status`, you should get

```
On branch count-to-ten
nothing to commit, working tree clean
```

And if you run `git diff`, you should get no output.

But wait, there's more! You can (and you will, very often) compare your branch with another branch. This is very important if you are a bit nervous before merging your branch back into master. Let's do it quickly:

```
git diff master
```

This will compare your `count-to-ten` branch with the master. The `HEAD` of `count-to-ten` points to a different commit than `HEAD` in master. This comparison highlights not only what you added now in your branch, but also what was added in the master branch and so not present in your

branch. In this case, sentences in red are in the master branch but not in your branch, and green are in your branch, but not the master branch.

Now you will push the local branch called count-to-ten to the remote repository. This is a very simple command:

```
git push origin count-to-ten
```

You should see some output that ends with this:

```
To GitLab.com:nicoloubser/testproject.git
* [new branch]      count-to-ten -> count-to-ten
```

If you open GitLab in your browser and go to your project, you will see a dropdown with the word “master” in it. Clicking that dropdown box, as shown in Figure 3-3, should show you that your branch has now been added to the list.

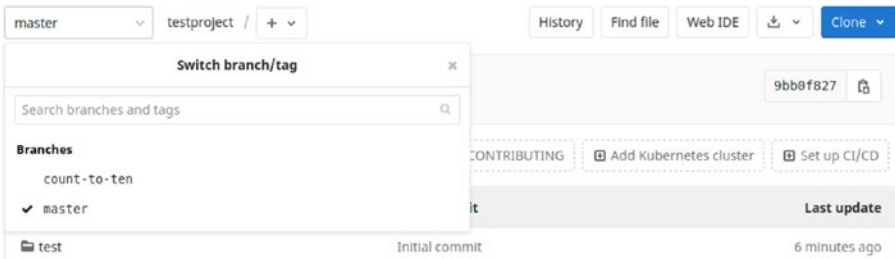


Figure 3-3. *Gitlab branch interface*

That was quite easy. But your deployment branch is master, and you still do not have the code you added to the count-to-ten branch in master. Next, I will explain how to get the code from one branch to another. The first thing you do is check out master:

```
git checkout master
```

Now you merge count-to-ten into master, in your local repository:

```
git merge count-to-ten
```

You should see affirmation on the screen that the merge succeeded. If you now run the following command, you will not see any changes:

```
git diff count-to-ten
```

All you need to do now is push master to the remote, where it will one day be released to the server on production!

```
git push origin master
```

That was all quite simple! As with Docker, Git can do very complex things. What I am aiming with this book is to get you up to a functional level.

A More Advanced Use Case

You will now explore a more advanced use case. You will create a new directory and clone the branch into it. You will then have the same codebase in two locations on your computer. The reason you are doing this is to emulate two people working on the same code base, and to reinforce some of the commands. You will then edit the master branch in the latest directory you created and push that work up to your remote repository in GitLab. After that, you will change into your original directory and ensure that master is checked out in that branch.

To start with, from the command line, make sure you are in the directory where your test code is. Once again, if you are using Windows, use the Docker quickstart as it gives you access to Linux.

If you type `ls` (short for list), you should see the test directory listed, where your current code is located. In my instance, it is just `test_project`. Now log into GitLab, and open your testProject project. You will see a blue button with the word Clone on it. Click it and copy the HTTP link. After

CHAPTER 3 REPOSITORIES AND GIT

that, go back to your command line interface and create the directory you want the code to be cloned into, like so:

```
mkdir testProject2
```

Type `git clone` and then copy the HTTP link you got from GitLab after it, plus the name of the new directory you want it to go in, like so:

```
git clone https://gitlab.com/nicoloubser/testproject.git  
testProject2
```

After the cloning is done, type the following command to enter the directory:

```
cd testProject2
```

Make sure you are in the master branch. The output of this command should be

```
* master
```

```
git branch
```

Now, while you are in the `testProject2` directory, you will make some changes to the `test.py` script. Add the following words to the end of the script:

```
print ("This is another change")
```

At the beginning of the file, where you wrote `print("Hello world, from Python")`, remove the `from Python` portion:

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

Now run the following commands to push this code to your repository:

```
git add .  
git commit -m 'Editing code for our example'  
git push origin master
```

Your next command is to go into your previous test directory:

```
cd ../testProject
```

At this point, you are in a local branch of master and you have a remote branch that has code changes in master. Next you will fetch the code. In general, you will just do a pull, but in order to demonstrate something, you will fetch first.

```
git fetch origin master
```

This fetches the remote master code and saves it to your local repository but does not integrate it with your working files. If you inspect your files, you will see they remained untouched. Next, you will compare your local files with the code from master you just fetched:

```
git diff origin/master
```

This will show you all the recent changes that you are bringing in from master. These are the changes that you created and pushed from the other directory.

Merging Conflicts

Before you merge the work, let's explore the bane of many software developers: the dreaded merge conflict. To create a conflict, you will edit your local `test.py` file, on line one. Remember you edited it in the other directory where you removed the `from Python text`? Go to that line and replace

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

with

```
print ("Hello world, from my computer")
```

This should be enough to let Git decide it is not up for a merge, and will give you the option as to what to keep and what to let go. Just a normal diff won't show the conflicts. First, you merge the remote work with the local branch:

```
git merge origin/master
```

You should see this message:

```
CONFLICT (content): Merge conflict in test/test.py
Automatic merge failed; fix conflicts and then commit the
result.
```

This conflict is in your `test/test.py` file, exactly where you expect it to be. Open that file, or refresh it in your editor if your editor does not auto refresh. You will see some code that you did not add in the file. Leave these added lines for now.

At this stage, it is up to you to decide which line should go and which line should stay. This is a judgement call, and you need to understand why both lines of code would have been edited by two people at the same time. There is no immediate right or wrong answer here. The conflict markers are surrounded by `<<<<<` and `>>>>>` and separated by `=====`. This is a simple example, but it is not unheard of for this situation to become very complex.

How do the markers indicate conflict?

The section marked as HEAD is what has changed on your branch. HEAD always points to the last commit on your machine for the specific branch you are in. The portion below it is what has been merged in from the other branch.

A simple example may look as follows:

```
<<<<<<< HEAD
```

```
This text is from the current branch,
```

```
=====
```

```
This text was merged in from the other branch and clashed with  
the text from the current branch
```

```
>>>>>>> 468754A654B654CDE65468768
```

I would like to elaborate on these two commands, `git fetch` and `git pull`. Normally, especially if you work with a large group of developers, you need to pull the remote branch before you can push your work to it. The `git fetch` command fetches the code from the remote but does not merge it into your working environment. This is a very safe way to fetch data from the remote (also called upstream) and do a compare to see what has changed, before you merge the fetched work (origin/master) into your local working tree. The `git pull` command, on the other hand, does the fetch and merge steps automatically as one command.

Let's test this. If you have followed all of the steps, you should now have a `test.py` file with conflict markers in it. One easy way to get rid of them is to undo the merge.

```
git merge --abort
```

If you inspect your file now, you will see the code has reverted to what it was before the merge. The code in master, however, will still have the same changes you made to it earlier, and merging upstream into your local repository will once again cause the merge conflict to be introduced. You can test this by using the pull command:

```
git pull origin master
```

With `git pull`, you have now fetched and merged all in one.

You should never push conflict markers back upstream to the remote. Your code cannot execute with these markers in it. You must remove them.

Let's remove the conflict markers. Let's assume the code you pulled is less important than the code you wrote. So you have this code:

```
<<<<<<< HEAD
print ("Hello world, from my computer")
=====
print ("Hello world")
>>>>>> ebba576e4226077c5a059bccc17b4b126b2d9f5c
```

Remove the bottom section and all the markers so that only the following is left:

```
print ("Hello world, from my computer")
```

Removing the Need to Type Your Password Every Time: SSH

Typing your password every time can be annoying, especially if you have chosen a very long password. There is a very secure way around it. In this section, you will generate SSH keys and add them to your GitLab repository, change your local repository to use SSH, and from then onwards you won't need to use your username and password anymore. SSH is a cryptographic protocol to secure route data between unsecured networks. There are multiple algorithms you can use. RSA and ED25519 are popular ones. Either of them will create two certificates on your local computer. A public key (which you can distribute) and a private key (which you should keep secret and on your computer, or the computers you want to use it from). A guideline to set up SSH can be found [here](https://docs.GitLab.com/ee/ssh/), but we will still discuss it, as it is good knowledge to have: <https://docs.GitLab.com/ee/ssh/>.

Type in the create key command:

```
ssh-keygen -t rsa -b 2048 -C "your-email@email.com"
```

You may be prompted to overwrite the current key. You do not want to do this. Type a new location, preferably in the same location as your other keys. When you are prompted for a password, leave it blank. It is, however, a good idea to have a password, so if someone steals your private key they cannot use it without the password.

With the key generated, you will now copy the contents of the public key and paste it into your GitLab's SSH directory. You can use any method to copy and paste the code. You can go to the directory where you created the key, open the key, and just copy the content, or you can use one of these methods, depending on your operating system. I called my public key **GitLab_rsa**.

- Mac:

```
pbcopy < ~/.ssh/GitLab_rsa.pub
```

- Linux:

```
xclip -sel clip < ~/.ssh/GitLab_rsa.pub
```

- Windows:

```
cat ~/.ssh/GitLab_rsa.pub | clip
```

Once you have the public key in your clipboard, click the right-hand corner icon. A dropdown will appear;; choose Settings. Once you have clicked it, a new page will appear with a menu on the left-hand side. Select SSH Keys from that menu. In the text box that appears, add your key.

There is one step left to do. When you cloned the branch from the remote to your local, you selected the HTTP method. You need to change

that to SSH. This is easy. Using your command line terminal, go into the first directory you created and type

```
ls -ltra
```

In the output you will see a `.git` directory. Inside the `.git` directory is a config file where you can change your Git settings. `.git` is an invisible directory and not all software will see it. Your command line can see it with the aid of the `-a` flag. Sublime can also see it. You will, however, reconfigure this using a command line call and not by opening the `.git/config` directory. In GitLab's projects, click your project and then click the blue Clone button. Copy the clone with the SSH address. Then, back in your command line, type the following and make sure your SSH address is where my address is:

```
git remote set-url origin git@GitLab.com:nicoloubser/  
testproject.git
```

That should be it. If you type any command, for instance the following, you should not be prompted for a password:

```
git remote show origin
```

Gitignore

When you add files to your Git repository, you use the `git add` command. You can also add individual files. For instance, adding two PHP files can be done like this:

```
git add file1.php file2.php
```

But what if there are files that you do not want to commit and push up to the repository? For that purpose, you can use the `.gitignore` file. `.gitignore` files specify the files and directories that should not be added and committed to the repository. For instance, you may want to ignore the

logs in your log file directory and your complete vendor directory. Then your `.gitignore` file would look like this:

```
/storage/logs/*.log  
/vendor
```

If you don't have a `.gitignore` file, you can just create one. It may happen that you have already added a file to the repository, meaning it is already being tracked. Adding it to the `.gitignore` file now won't make a difference. Once it is being tracked, you need to remove it from tracking first, and then your `.gitignore` command will take effect.

```
git rm --cached file-or-directory-name
```

Explanation of the command:

- `rm` removes a file.
- `--cached` removes it only from the index, not your working space. Your files will still be there after you have run this command.

git stash

`git stash` is another neat utility in Git's toolbox. Let's say you are working on a branch called `dev`. You are halfway into your work and not ready to add and commit the changes you have made. Now you get a request from your manager that you need to fix something in the master branch ASAP. Obviously, you cannot check out the master branch without carrying over all of your current file changes. You can always just add and commit the work you have done in your `dev` branch and then check out the master branch. You will have a bit of a dirty commit, and another commit point in your branch. Or worse yet, you may have a pre-commit hook (not included in this book). A pre-commit hook may run style checks and unit tests

upon commit, and without them passing, you cannot commit your code. This means you cannot check out the master branch, because you cannot commit your code. `git stash` is a great way around this problem. It stashes all of your changes away from your working index, allowing you to check out other branches without having to commit. In general, you use it as follows:

1. `git stash`
2. Check out another branch, do work in it, and commit it. If you do not commit your work, you will carry the changes made in this branch over to the previous branch once you check it out.
3. Check out the previous branch.
4. `git stash pop`

Here is a short list of stash commands that will be helpful.

Stashing your work: `git stash`

Stashing your work with a descriptive message: `git stash save 'descriptor'`

Viewing your stash: `git stash list`

Removing pulling the first element of your stash into your working environment: `git stash pop`

Selecting an element from your stash based on the stash index: `git stash apply stash@{1}`

git reset and revert

Every so often you need to undo changes you made or changes someone else has made. Git uses commit points which can serve to indicate restore points.

`git reset` is a bit more destructive than `git revert`. `git reset` will reset to a previous commit point, removing all the commits between your current state and the desired commit you want. For instance, if you have commits marked as commit hashes A to E, like

A -> B -> C -> D -> E -> current uncommitted work.

if you run

```
git reset --hard C
```

your commits will look as follows afterwards, and the rest will be gone, although still present in the remote repository:

A -> B -> C

To push these changes to the remote, you need to force it using the `-f` flag. Use the `-f` flag very cautiously, and only if you really must, as you can overwrite other developers work if you do so.

```
git push origin branch-name -f
```

`git revert` will do the same, but with a big difference in what is going on in the background. `git revert` will not remove the commits you don't want. Instead, it will do the reversing of them as commit points themselves. You can also specify one commit at a time to revert or a range of commits.

```
git revert E or git revert E..D
```

The commit history will look something like this, where HEAD is pointing to a reverted commit:

A -> B -> C -> D -> E -> reverted E -> reverted D

There is a lot more to Git than this chapter can cover, but a lot of the aspects you will encounter in your day-to-day work were covered here.

Cheat Sheet

`git clone repo_location`. clones the repos into the current directory.

`git clone repo_location location-name` clones the repo into the location-name.

`git checkout branchname` lets you check out an existing branch.

`git checkout -b branchname` creates and then checks out a branch.

`git branch` lists branches on your local machine.

`git add .` adds all files to your index.

`git add filename` adds a file called filename to your index.

`git add filename1 filename2` adds multiple files to your index.

`git commit -m 'message goes here'` commits your work and makes it ready to be pushed.

`git pull origins branchname` pulls the latest changes from a remote branchname.

`git push origin branchname` pushes branchname to the remote.

`git stash` stashes your work away from your working environment.

`git stash pop` takes the added stash and adds it to your working environment.

`git stash pop stashId` takes a specific stash and adds it to your working environment.

`git stash list` shows all of your stashes.

`git remote set-url origin https://github.com/USERNAME/REPOSITORY.git` changes the remote URL of your Git system.

CHAPTER 4

Programming in Python

Solving problems is the main job of the software engineer. Programming is important because a lot of tasks can easily be automated. You can create systems that schedule parcel delivery, that avoid accidents, that recognize your friends, that drive cars. You can create a system that waters your plants and one that fetches your desired news feeds and shows them to you. The list is endless. I remember when I first saw the power of automating mundane tasks by giving them to a computer to do.

I was about 22 years old, working at a bus transport company. My job on weekends at that stage was to take all the passenger lists, every 45 minutes, write down how many people were on each route, and type that into an Excel spreadsheet. Then I would print the sheet and go to this big board with all the routes and busses on those routes and write down the number of passengers per route. The operations team would then make decisions based on that data as to how many busses they needed, how many seats per bus, and so on. This would take me about 35 minutes at a time and I did not like doing it at all. Soon the company hired a developer, installed some computer screens in the operations room, and automated this process. I was incredibly impressed when I saw the board updating every 5 minutes and taking what can only be called “no time at all” to do so.

I saw these changes spread through the company. If I had to work night shifts, we had a large number of manual tasks to do before the next morning. Soon, those tasks disappeared as well. The magic in the system handled it for us. I was very impressed with those automated changes, but at the same time I became aware of the fact that we needed less manpower, and that was the way the world was moving. Fortunately, automation has opened markets and created a lot of new opportunities.

In this chapter, you will look at writing code. Writing code is only part, albeit a big part, of solving a problem. You have already looked at Docker and Git, which form the basis of running code and controlling the codebase. This chapter is just your first step at writing code, and it's not a very complex introduction. It will take a long time to fully learn your first programming language, but this chapter is a starting point.

What Is Programming?

Writing code on a basic level is nothing more than telling a program what it has to do. The catch is, programs cannot think for themselves, and you have to be very precise in what you tell them to do. Your goal as a programmer is to change the state of a program, or a small portion of a big program, from one state to the other. You must envision the possible states the system can be in, as well as unknown problems that can occur. You will have a start state in your program, and an end state. As an example, say you need to send an email. In order for you to achieve your goal of sending that email, you need to give the software some commands that can manipulate the state of the program from “unsent email” to “sent email.”

Programming can be rather difficult, but it is not the act of writing code that is difficult. It depends on how difficult the problem is you are trying to solve.

Python

We will use Python, an easy-to-learn and very powerful programming language. Your Python source code, which this chapter is all about, is executed by the Python interpreter, which you installed in Chapter 2. So, at this point, you should have a Docker environment capable of running Python applications. You do not need the Docker environment, though. You can just install Python on your local machine, but I strongly recommend that you use Docker to get used to it. You may even learn things about Docker not discussed in this book while playing around with it.

Setup for This Chapter and How to Use It

You will use the Docker environment you created in Chapter 2 to create the example files. You may remember that you set up your docker-compose file to use a shared directory called test. Whatever you put in that directory can be interpreted by Python. Let's test this quickly.

In your Dockerfile, change the CMD command from

```
CMD ["python", "/home/test.py"]
```

to

```
CMD ["python", "/home/main.py"]
```

In the /test directory, where test.py is living, add a new file called main.py. Add this line:

```
print('This is the first line')
```

Because you made a change to the Dockerfile, you need to rebuild it:

```
docker-compose build
```

Now run it:

```
docker-compose up
```

Whenever you make a change to `main.py`, just run `docker-compose up` to execute it. All of the examples below can be added to `main.py` for you to test. At your own discretion, you can delete old lines of code in the `main.py` file, but it is sometimes handy to have if you want to retest an old line of code.

Basics

Before we get into coding, there are some basics you need to be aware of.

In Python, all statements end in a colon. I will cover all the statements throughout this chapter. The colon indicates the start of a code block. A code block is the block of code belonging to the preceding statement, and code blocks can contain more code blocks. A code block, which follows a colon, is indented with four spaces or a tab. The industry standard prefers four spaces and not tabs, purely because not all systems display tabs the same way. In Python 3, you cannot mix tabs and spaces to indent code blocks. Your editor should be set up to provide four spaces when you press tab, and Visual Studio Code does this automatically.

The following piece of code displays the concepts I explained above. It is important to bear in mind that not all editors display spaces with the same width. In general, a Python editor should because it makes it easier to spot indentation errors.

```
def my_function(): # Start of a code block 1
    if 1 == 2 : # Part of code block 1 and start of code block 2
        print('Part of code block 2') # Part of code block 2
        print('Part of code block 2') # Part of code block 2
    print('Part of code block 1') # Part of code block 1
```

This was a quick overview, but you will definitely see this whenever there are code blocks and statements, and it will become intuitive to you in no time.

Commenting Your Code

To make your code more understandable, you can add human-readable comments to it. The Python interpreter will ignore these comments and they will have no effect on the execution of the program. Chapter 5 offers a detailed description on how to do this, but for now, you just need to know there are two ways to add comments. The first way is to add a hash symbol (#) in front of a line, and the Python interpreter will ignore that line. The second way is to encase the description between two sets of triple quotes. This is very handy when you want to explain why you did something in your code. I do so throughout the book to give inline explanations on code, so you will benefit from reading them. Now look at Listing 4-1.

Listing 4-1. Commenting code

```
print('The interpreter will interpret me')
# The interpreter will ignore me
"""
The interpreter will ignore this line as well
as this line.
"""
```

Variables

Now let's move on to variables, which is something you will use constantly. Most people get an intuitive understanding of variables pretty quickly once they start reading about them. Imagine you have one tennis ball. Now imagine you have small box, and you put your tennis ball into it. The box in this instance is the memory location that Python has reserved for your tennis ball. This box is called a variable, and the tennis ball is the value assigned to that variable. You will need to give your box a name. Let's call it `my_ball`.

Whenever you want to access your tennis ball, you use the name `my_ball`, which tells everyone that they must use the box with that name. Whenever you need to pass a tennis ball to someone else, you give them the name of the box, and they will instantly know which box to use. Instead of putting a tennis ball into the box, you can also put a basketball or a rugby ball into the box. You can have many boxes, each with its own name, containing different or similar balls. You can also swap balls and change the ball inside the box. This is true for variables. You have a memory location, which is represented by the box; a value, which is represented by the ball; and then the name, which points to the memory location which stores the value, which is the name which points to the box.

Variables are used by a program to store data. It is temporary storage and the values will not persist past the execution of your program. This data is used by the program to base decisions on, to output it in one way or the other to the user, or the program can save the variables in long-term or short-term storage. An example of a variable in your system can be a password, username, mobile number, or something more complex, even something you created yourself. There are a few types of basic variables that you will use regularly to build your system. It is safe to say that all software, no matter how complex, uses these basic types.

Five of the most basic built-in types are integer, float, string, arrays, and boolean. You can also build your own much more complex data types that will be used as variables, and we will discuss this later in the book. Unlike some programming languages, Python is dynamically typed, meaning you do not need to tell Python that the variable you are creating should be a string or an integer. Python infers the type from the data you give it, meaning Python selects whether it is a string, integer, float, or boolean. So whatever type of data you have assigned to a variable, that will be the variable's type.

A variable consists of a variable name and a value. You declare a variable by using the variable name, the assignment operator, and the variable type. In the next example, `number` is the variable name, and the

integer value 200 is assigned to it using the assignment operator. This gives us a variable of type integer, containing the value 200. The variable `number` points to the value 200. You can let more than one variable point to the same value.

```
number = 200
```

In Python, and most programming languages, a single `=` means the value on the right is assigned to the value on the left. `==`, a double `=`, tests whether the values on the left and right are equal to each other. We will also get to this a bit later.

Integers

Integers are numbers without a fraction component. In Python 3, integers can be of unlimited size. Let's go through the following example step by step.

In `main.py`, type the following and press Enter:

```
age = 20
```

You now have a space in memory in Python holding the value 20. You can reference this value by using the word `age`. The following line should print 20:

```
print(age)
```

Now add the following line just below that:

```
max_age = 30
```

You now have two variables, one holding the integer 20 and another the integer 30. The latter can be accessed with the name `max_age`. Let's subtract the two values and print the result to the screen. You can either assign the result to another variable and then put that result in the `print`

function or you can place the equation directly into the `print` function. Enter the following line, and after that, execute the code from the command line. The response on the screen should be 10.

```
# Use the minus sign to subtract max_age from age
age_difference = max_age - age
print(age_difference)
```

As a refresher, to execute the code, run this command from the command line :

```
docker-compose up
```

You can inspect your variable's type by using Python's built-in function called `type()`. In the example below, `variable_type` may look like a string when you print it, but it actually is a more complex type and, coincidentally, a type called `type`. This new complex type holds as a value the type that `my_integer` is, which is `int`.

```
my_integer = 4
variable_type = type(my_integer)
print(variable_type)
```

The result should be `<type 'int'>`.

Float

Floating point values are values with fractions, for instance:

```
distance_in_meters = 8.5
print(distance_in_meters)
```

They can be used with integers and the result will be a float.

Boolean

Boolean values can either be True or False. In Python, True and False are always capitalized. They can be used in a variety of scenarios. For instance, a program may need to know if you are an admin, so there may be a boolean variable called `is_admin` that is set to True to signal to the rest of the program you are indeed an admin.

```
is_admin = True
```

Strings

Strings are words and sentences, denoted by putting double or single quotes around them.

```
name = 'John'  
print(name)
```

You can join strings using concatenation with a `+` concatenation operator. This is something you will encounter a lot.

```
greeting = 'Hello '  
print(greeting + 'John')
```

Strings that are enclosed in single quotes, which have more single quotes inside them, need to be escaped, and the same goes for double quotes with double quotes within them. To escape the middle `'`, you need to add a backslash before it. This tells Python that it is an actual `'` that should be printed, and not one of the enclosing quotes. For instance, this string assignment will break the system:

```
# This will cause an error.  
message = 'let's go'  
print(message)
```

Escaping fixes it:

```
# This will print.  
message = 'let\'s go'  
print(message)
```

A non-escaping method to fix it is to put double quotes on the outside, like so:

```
# This will print  
message = "let's go"  
print(message)
```

Last Word on Variables

A variable's value can change. It is not a fixed value. In other words, you can reassign data to a variable as need be. You can also change a variable's type. You cannot, however, let two variables of different types work together without casting one of the values to be the same as the other. This is called type casting and it is discussed in the next section.

Constants

A lot of programming languages have a concept called a constant. A constant is a type of variable that can only be assigned once and never be changed, hence it is a constant value. This is a very useful feature to have. There are many use-cases for a variable that cannot have its value changed once assigned, like a file location. Python does not have this functionality. In Python, a constant is identified by declaring a normal variable but writing the variable name in uppercase. This merely signals to other developers that this value should not change.

A normal variable:

```
file_location = '/bin/bash/'
```

A constant:

```
FILE_LOCATION = '/bin/bash/'
```

Type Casting

Now that you have been introduced to different basic types of variables, it is important to tell you that you will sometimes need to change basic data types. Why? Because data types cannot be intermixed. If you have a string with the value of 10 and an integer with the value of 20, you cannot add the two numbers without changing the string to an integer. In software development, this is called type casting or type juggling.

The type casting functions are in Table 4-1.

Table 4-1. *Type Casting Functions*

<code>str(x)</code>	If x is not a string, this converts x to a string.
<code>float(x)</code>	If x is not a float, this converts x to a float.
<code>int(x)</code>	If x is not an int, this converts x to an int.
<code>bool(x)</code>	If x is not a boolean, this converts x to a Boolean.

Here is an example where you tell Python to treat an integer as a string when concatenating the integer and the string. Type the following commands:

```
first = 'There was '
string_part = '1' # 1 is surrounded by quotes, making it a
string
int_part = 1 # 1 is standing alone, making it an integer
```

Python should have no problem with the following command:

```
print(first + string_part)
```

The following command should produce a warning, something to the extent of “TypeError: can only concatenate str (not “int”) to str”, so try it out:

```
print(first + int_part)
```

You will encounter these instances many times. In this instance, the solution is to cast the variable called `int_part` to a string, using Python’s built-in function `str()`. You put the value you need to cast to string between the brackets, like so:

```
print(first + str(int_part))
```

To illustrate casting from a float to an integer, run the following code and see if the output is as you expected:

```
print(int(4.567))
print(float(5))
print(bool(1))
print(bool('false'))
```

Shorthand Assignment Operators

As you know, `=` assigns data on the right to the variable on the left.

The following are some shorthand operators. For these shorthand operators to work, variable `a` must be associated with a value already.

<code>a = 10</code>	
<code>a += 5</code>	This equates to <code>a = a + 5</code> . This will concatenate strings.
<code>a -= 5</code>	This equates to <code>a = a - 5</code> .
<code>a *= 5</code>	This equates to <code>a = a * 5</code> .
<code>a /= 5</code>	This equates to <code>a = a / 5</code> .

Sequences and Maps

Remember the tennis ball analogy I used earlier? I need to change it slightly to explain how sequences and maps work. In this instance, imagine you have a bigger box than previously. Now, instead of only being able to add one ball into the box, you can add more. So, you end up with one box containing multiple balls, plus they don't have to be the same kind of ball. Not all boxes are the same in this case. Some boxes can take an unlimited amount of balls, and you can remove balls, add balls, or change balls as you please. Some boxes, however, are sealed once you have placed the balls inside, and the balls cannot be removed or edited.

Sequences and maps (the boxes, in the above analogy) are variables, called data structures in this case, containing sets of values. They are referred to in general as arrays, even though there are behavioral differences between real arrays and map, lists, etc. They are normally used to represent data that are related, such as customer details or a shopping list, but your use case will determine what data is inside it. In this section, I will show how these data structures are used and how the data is accessed using slicing and using indexes. A bit further along in the chapter, you will encounter arrays again and see how to access the elements sequentially using loops.

Before we jump in, just a final high-level overview of arrays. Arrays in Python come in the following forms: sequences and maps. They are slightly higher-level terms. On an implementation level, that is on the level you write your code, sequences come as lists, strings, and tuples, and maps are dictionaries. You will work with lists, tuples, and dictionaries, but can refer to them via their higher-level names as well. You can iterate through sequences and maps and do computations and logical checks on the values. Similar to how Python comes with built-in functions, so do lists, strings, sequences, and maps. You will look at how to use built-in functions a bit later in this section.

Lists and Strings

Lists and strings are both sequences. You have already encountered strings, so I only mention it here as it is technically a type of sequence. You can also access a string's data using indexes and slicing, exactly the same as a list. Both slicing and indexes will be covered in this section. A list is a collection of changeable data. You can add values, remove values, and edit values. Lists are initialized as comma-separated values between square brackets. Or you can use the list's built-in functions to add and remove values. This is how you declare a list:

```
my_list = ['apple', 'pear', 'orange', 50, 4]
print(my_list)
```

Remember that I said lists, tuples, and maps are complex variables? This means they are self-contained objects (you will encounter objects a bit later in this chapter). But it is good to know that they are objects and have their own built-in functionality. To access the functionality of any object, you use the dot operator (.) followed by the function name. In this example, I demonstrate how to do it, using lists as the example object.

Here are more ways to get data into a list:

```
# Declare lists with a few default values, although it can be
left empty.
new_list = ['value1', 'value2', 'value3']
# new_list is now an object
# Use the dot operator, and the built in 'insert' function to
add elements to the list.
# The insert function's parameters are as follows insert
(position, value)
# position indicates the position at which you want to insert
your value
new_list.insert(1, 'value4')
print(new_list)
```

```
# You can also just add data to the end of the list using the
append method().
my_list.append('another value')
print(my_list)
```

To add data to the start of a list, you use `insert` it with a position of 0.

Accessing Data Inside Lists

You need to be able to read and write data in the lists. In general, there are two ways to do this, indexes and slicing. Slicing is based on indexing so we will take a quick look at indexing first.

Indexes

Data inside lists can be accessed for reading and writing using their indexes. An index denotes the location of an item in an array and always starts at 0. You do not need to specify the indexes yourself; it is done automatically when you assign the data to the list.

```
my_list = ['apple', 'pear', 'orange', 50, 14]
print(my_list[0]) # Lists and tuples are both referenced using
square brackets.
print(my_list[2])
```

The first print function should output the word *apple*, and the second, the word *orange*. The index structure will look like this, where the index is to the left of the arrow and the value to the right:

```
[0 -> 'apple', 1 -> 'pear', 2 -> 'orange', 3 -> 50, 4 -> 14]
```

Slicing

You can also perform operations called slicing on your lists. It's a bit more confusing at first, but very powerful once you get the hang of it. Take our previous example of `my_list`. The anatomy of a slice looks like this:

```
my_list[start-point : stop-point]
```

The start point and the stop point are both from the start of the list. Listing 4-2 has some examples.

Listing 4-2. Slicing

```
my_list = ['apple', 'pear', 'orange', 50, 14]

print(my_list[0:1]) # Prints apple
print(my_list[0:2]) # Prints apple, pear
print(my_list[-1:]) # The last item in the array
print(my_list[:-1]) # Everything except the last item
```

There is also another parameter that can be added to the slicing operator called `step`, which indicates at what interval numbers should be returned in the slice:

```
my_list[start-point : stop-point : step]
```

Keeping to the `my_list` list, I'll illustrate how to skip every second item in the array.

This example states that slicing should start at position 0, go until position 5, and skip every second entry:

```
print(my_list[0:5:2])
```

But what if your list will constantly be changing in size? The drawback of the above method is that if you want to operate on a whole list that changes in size, you would manually have to alter the middle argument of the slice, which is now 5. Using the built-in Python function called `len()`,

which returns the length of a list, you can automate the stop-point to always be the length of your list. The lines below show how `len()` operates on lists:

```
print('my_list has ' + str(len(my_list)) + ' entries')
print(my_list[0:len(my_list):2])
```

Tuples

Tuples are of a fixed size and are created using round brackets or without any brackets, like so:

```
my_tuple = ('eat', 'sleep', 'repeat')
my_tuple = 'eat', 'sleep', 'repeat'
```

Tuples are sliced exactly the same way as lists. Refer to the code in Listing 4-2. *Fixed size* (immutable) means that once it has been created, you cannot add or remove elements from it, nor can you edit values. This means that you use a tuple when you know exactly how many elements you will have. This makes tuples a bit faster than lists, and also safer, since you cannot accidentally delete data. Although there are many similarities between lists and tuples, you must remember you cannot add or remove elements from tuples. Tuples can be sliced in the same way as lists.

When to Use a List and When to Use a Tuple

You should use a list when you have a variable amount of data to populate it with. For instance, if your code searches your newsfeeds daily for keywords like *invest* and adds that news article into your data structure before displaying it, then a list is appropriate because you will get a different amount of news articles every day. If you are looking to store the daily hourly temperature in your structure, then a tuple is the way to go because every day only has 24 hours.

Dictionaries

A dictionary, also called a map or key/value pair, is a data structure where each indexed value has a corresponding key. Data can be accessed using this key. The index should be a meaningful word, describing what the value relates to.

Anatomy of a dictionary:

```
details = {
    key_unique_name : value,
    key_unique_name : value
}
```

This would translate into an actual example looking like this

```
user_details = {
    "name": "Nico",
    "surname": "Loubser",
    "age": 68,
    "house_number": 68,
    "street_name": "Mitchell street"
}
```

Below is what the list equivalent looks like. Compared to lists and tuples, the keys in both lists and tuples are not really absent, they're just implicitly assigned and start at 0. Compared to the dictionary implementation, you can see how the dictionary is a lot more readable. In the list example, which follows, without a readable key, we have no idea what 68 is, whereas in the dictionary version it is easy to understand.

```
details_list = ["Nico", "Loubser", 68, 68, "Mitchell street"]
```

Getting data from a dictionary is very easy. As with lists and tuples, you access data using the indexes. You cannot slice a dictionary because it is not a sequence and slicing won't make sense.

```
print(user_details['name'])  
print(user_details.get('name'))
```

We will go through list, tuples, and maps in the “Loops” section of this chapter.

Decision-Making Operators and Structures

All decisions made in programming are based on a true or false outcome. This can become very complex as there may be a multitude of factors to base a single decision on. But even the most complex algorithm will boil down to a true or false.

Comparison operators, combined with logical operators, form the basis of decision-making algorithms. But don't be fooled. You can use these humble components to build systems that are complex. Yes, you can even build AI systems using these components.

Operators

When combining operators (or, in fact, when only using a single one) we form what is called an expression. You will need a lot of expressions to achieve your program's goal. Python provides you with a set of comparison operators as well as logical operators to help you build your expressions, and you will deal with them in this section. While building your expression, you will use the comparison operators to evaluate different values and logical operators to include (and exclude) certain outcomes.

Comparison Operators

This section is important to understand before you get to decision making structures.

When we want to make decisions based on data, we need some regular mathematical operations to do so. The operators in Table 4-2 are of interest at a beginner level. The results of these operators will always be True or False. A comparison operator takes a value to its left and compares it to a value on its right. The outcome of that evaluation will be positive or negative, in other words, True or False.

Table 4-2. *Comparison Operators*

==	Equals
!=	Not equal to
<>	Not equal to
>	Left is bigger than the right-hand side
<	Right is bigger than the left-hand side
<=	Right is bigger than or equal to the left-hand side
>=	Left is bigger than or equal to the right-hand side

Examine the following code and run it in your environment. You will see how the outcomes are boolean values.

```
# Initialise 2 variables
age = 100
limit_age = 120
# Do logical comparisons.
print(age == limit_age)
print(age > limit_age)
print(age < limit_age)
```

Logical Operators

This section is important to understand before you get to the decision-making structures. When we combine comparisons operators, we need to specify how those comparisons are related to each other. For instance, you can have a variable called `car` and variable called `year_model`.

Here's a demonstration of the *and*, *or*, and *not* logical operators:

Assume `car = 'Ford'` and `year_model=1996`.

and

Let's start with *and*:

```
if car == 'Ford' and year_model != 1998:
```

This will result to True if both comparisons are True. Should the very first comparison, `car == 'Ford'` in this case, result in False, then Python will not evaluate the rest of the logical operation.

Boolean *and* logic works as follows:

```
True and True == True
False and True == False
True and False == False
False and False == False
```

or

Here is the *or* version:

```
if car == 'Ford' or year_model != 1996:
```

This will result to True if one of the comparisons is True. The whole logical operation will be evaluated.

Boolean *or* logic works as follows:

```
True or True == True
False or True == True
True or False == True
False or False == False
```

not

Finally, *not*:

```
if not(car == 'Ford'):
```

This will negate the comparison, so if `name == 'Ford'`, this will result in `False`.

```
not True == False
not False == True
```

Truth Tables

These expressions can get very complex and need a lot of practice, but they form the basis of the logical evaluations in your code. One of the issues is that you may have a simple expression, as in Listing 4-3, but there are three variables, each of which's comparison may be `True` or `False`, so how do you accurately determine what values will pass the expression successfully? There is something called *truth tables*, which can help you deduce what the outcome of your expression will be, especially in the beginning as you are just starting out. Truth tables hold all the possible values of your expression and calculate their outcomes. I will give a quick overview here.

Suppose your expression looks like Listing 4-3.

Listing 4-3. Logical expression

```
(amount == 10 and tax == 14) or discount != 100
```

This means, if your amount is 10 and your tax is 14 or the discount is not 100%, then this whole expression will evaluate to True. If the amount is indeed 10, the tax is 14, and the discount is not 100%, then this can be rewritten as (True and True) or True.

However, if the discount is 100%, this can be rewritten as (True and True) or False

You can construct a truth table to further see which values will make the expression pass or fail. See Table 4-3.

Table 4-3. Truth Table

amount == 10	AND	tax == 14	1 st result of the 'and'	OR	discount != 100	Result of 1 st result or discount
True	And	True	True	Or	True	True
True	And	True	True	Or	False	True
True	And	False	False	Or	False	False
False	And	False	False	Or	False	False
False	And	False	False	Or	True	True
False	And	True	False	Or	True	True
False	And	True	False	Or	False	False
True	And	False	False	Or	True	True

Table 4-3 has seven columns. The first and the third columns are all possible results for the first set of comparisons. Column two holds the logical operator for reference and column four holds the results. So the first four columns hold all the possible potential outcomes for (amount == 10 and tax == 14).

Column five just holds the *or* operator for reference. The *or* operator will now work on the data of column four and column six. Column six forms part of all possible results that the individual logical expressions, which make the whole expression, can have. The result of that last *or* is in column seven. Now, reading the table row by row from left to right, you can see what the evaluations should be for your expression to succeed. The successes are marked in blue in the last column.

Identity Operators

Remember in the beginning I said that more than one variable can point to the same value? With the identity operators, we can check whether that is the case. This is not the same as checking whether two variables have the same value using `==`. Two variables can have the same value, where the values are separate and occupy two different memory locations in Python. Identity operators do not care about the equality of two values. They care about whether both variables are pointing toward one single value in Python's memory. You can manually inspect variables to see if they are pointing toward the same memory location using the built-in Python function `id (variable)`. This will print/return the id of the memory location where the variable is pointing.

is

The identity operator *is* compares data on its left-hand side with data on its right-hand side to see if they point to the same object, as demonstrated in Listing 4-4.

Listing 4-4. Using `is`

```
set_one = [1, 2, 3, 4, 5]
set_two = set_one
if set_one is set_two:
    print('Both variables are pointing to the same list object')
```


is not

The identity operator *is not* compares data on its left-hand side with data on its right-hand side to see if they do not point to the same object, as demonstrated in Listing 4-5.

Listing 4-5. Using *is not*

```
set_one = [1, 2, 3, 4, 5]
set_two = set_one
if set_one is not set_two:
    print('Both variables are pointing to the same list object')
else:
    print('Both variables are not pointing to the same list
        object')
```

Membership Operators

Membership operators check whether the value on the left is in the array on the right.

in

The membership operator *in* checks whether the data on its left-hand side is in the array on its right-hand side. See Listing 4-6.

Listing 4-6. Using *in*

```
fruit = ['apple', 'pear', 'orange', 'kiwi']
if 'apple' in fruit:
    print('Apple is in the list')
```

You can also use the *in* keyword in loops, which I will demonstrate later.

not in

The membership operator *not in* checks whether the data on its left-hand side is not in the array on its right-hand side. See Listing 4-7.

Listing 4-7. Using not in

```
fruit = ['apple', 'pear', 'orange', 'kiwi']
if 'watermelon' not in fruit:
    print('watermelon is not in the list')
```

Precedence of Operators in Expressions

Operator precedence is a very important aspect of writing accurate algorithms. Precedence of operators, from a high level, is shown in Table 4-4. Note that binary operations are left out of this list. Going through this list, it becomes quite evident that ignoring precedence in your expressions can have dire consequences.

Table 4-4. Operator Precedence

()	Code enclosed in brackets are evaluated first.
**	Exponents are evaluated next.
*, /, //, %	Multiplication and division are evaluated next.
*, -	Addition and subtraction are evaluated next.
==, !=, >, >=, <, <=, is, is not, in, not in	Comparison, identity, and membership are evaluated next.
not	Logical operator <i>not</i> is evaluated next.
and	Logical operator <i>and</i> is evaluated next.
or	Logical operator <i>or</i> is evaluated next.

The following example demonstrates operator precedence:

```
1 + 2 * 10 == 30
```

What is the result? Would this equation as a whole return True or False? Most people familiar with mathematics will correctly point out that $1 + 2 * 10 == 30$ will never yield True. Because multiplication takes precedence over addition, the value is 21, not 30. You can solve this by adding brackets around the $1 + 2$. In the following example, the equation in the brackets takes precedence over the multiplication, and now $(1+2)*10$ is actually equal to 30:

```
(1 + 2) * 10 == 30
```

Proving that the logical operator *and* takes precedence over *or* is a bit more tricky and not as obvious. The following equation should do fine. When you execute it, you will see the result is True.

Type the following command:

```
print(True or False and False)
```

Because *and* is evaluated before *or*, you have the following evaluation pattern, where the bold portion is executed first:

True or **False and False**

The bold portion (**False and False**) is equal to False, so you can rewrite this as **True or False**.

True or False == True

Hence the end result is True.

If *or* was executed first, the following would happen:

True or False and False

You know that the bold portion, **True or False**, will yield True, so you just rewrite it in your algorithm as True for clarity. True and False will yield False.

True and False == False

With *or* taking precedence, you get the result as False. But since running the equation returns True, you can safely conclude that *and* takes precedence over *or*. If you really need to let the *or* portion execute first, just wrap it in brackets:

```
print((True or False) and False)
```

Bitwise Operators

I won't go into bitwise operators. Even though it is a very important concept, it is beyond the scope of this book to teach you binary calculations. I advise you to look into it once you understand the basics of programming.

Scope and Structure of Python Code

Scope is a rather simple but very important aspect in programming. Scope refers to the visibility of a variable under certain conditions. These conditions can be local, enclosed, global, and built-in. I will discuss scope here, but you really start handling it once you see more code.

Local Scope

Local scope refers to any variable that has been declared within a function or a class. We will deal with classes in the next section. These variables cannot be accessed directly, without referencing the class or function, by any code running that class or function.

Enclosing Scope

This scope specifically refers to nested functions, which are functions within functions. If you have a function within a function, then the inner function's scope includes that of the outer function's scope.

Global Scope

Global scope occurs when a variable is declared at the top level of a file. This means the variable is available to all scripts that import that file. Global scope variables are best avoided when possible. Due to its nature where it can be edited from anywhere, it is not good to rely on the accuracy of that variable.

Built-in Scope

These are built-in values provided by Python, which are loaded into the system whenever the system executes. This includes functions, among other things.

Control Statements

There are dedicated control structures that use comparison and logical operators to influence the state of your program. You will learn about them in this section.

If Statements

You use *if* statements when you need to execute a specific codeblock based on certain conditions. An *if* statement takes a logical expression and evaluates it, using equality operators, to obtain a value of True or False. It can also take a single Boolean value which will equate to True or False. Using the *if* statement, we link blocks of code to certain conditions, and we execute that code if those conditions are met.

An *if* statement starts with the word *if*, followed by an expression consisting of comparison operators and logical operators, and ends with a colon.

```
if temperature < 15 and rain == True :  
    return 'The weather will not be great today'
```

Some important things to remember:

- All code nested with four spaces belongs to this *if* statement.
- Remove the spaces and you are out of the *if* statement's code block.

So, an *if* statement is followed by an indented code block of at least one line of code. All the code belonging to this *if* statement should be at that level of indentation. The *if* statement can also contain two other parts. One is *elif*, and the other is *else*. Elif is a contraction of “else if” and you use it if you need more control over what block of code should be executed. In your program, you may want to execute a different block of code for every day of the week. Using *elif* in this case is more optimized than running seven different *if* statements, plus it tells us that the whole code block is related, which increases readability.

```
if day == 'monday':  
    some code  
elif day == 'tuesday':  
    come code  
elif day == 'wednesday':  
    some code  
# and so on.....
```

There will also be instances where your *if* statement does not evaluate to True, and if it has *elif*s, that they did not evaluate to True either. Even if none of your code blocks linked to your conditional statements have executed, you may still need it to execute something. For that you use the *else* statement.

```
if 1 == 2:
    print('1 is equal to 2')
else:
    print('1 will never be equal to 2')
```

Let's look at another example. In Listing 4-8, you can see how the *if* statement first evaluates whether `money_in_wallet` is less than the price, then it evaluates whether the amounts are equal to each other, and then if `money_in_wallet` is more than the price. It bases its decision of what to print on the outcome of these evaluations.

Listing 4-8. *if/elif* example

```
price = 30
money_in_wallet = 25
change_left = 0
if money_in_wallet < price:
    print('You do not have enough money')
elif price == money_in_wallet:
    print('You have the exact amount')
elif money_in_wallet > price:
    change_left = money_in_wallet - price
    print('You have ' + str(change_left) + ' left')
```

Play around with the code in Listing 4-8, changing the `money_in_wallet` value to 30, and then to an amount higher than 30.

If statements can also be a singular Boolean value:

```
if True :
    print('Hello')
```

In short, *if* and *elif* evaluate different expressions (or mathematical equations), and *else* executes if all of the preceding *if* and *elif* statements linked to that *else* statement failed.

Loops

Loops are structures that allow us to do a task repeatedly. This repetition of tasks is based on certain conditions. Loops are often used to iterate over lists, tuples, and dictionaries. There are two loops in Python, the while loop and the for loop. Some languages have even more loops.

While Loops

While loops are executed as long as the expression in them is true. This makes while loops best suited for when you are not sure how many times the loop must execute, and you have a logical algorithm that determines the amount of times it can execute.

```
while expression : # statement declaration and expression ended
with a colon
    run this code # All subsequent code linked to the while
    statement is indented.
    run this code
```

The while loop will execute as long as the expression is equal to True. There is a second way to exit a loop called a break, which we will look at later in this chapter.

Let's look at two actual examples. The first example prints numbers from 0 to 9. If you do not add a mechanism to exit the loop, the loop will keep on running. On the first line in Listing 4-9a, you set a variable

called `number` to 0. On the second line you start your loop. You can consider the variable `number` as a control variable, as it controls how many iterations the loop will execute. On the second line is the condition, where the loop will execute while `number` is smaller than 10, and then on the fourth line you update the variable to ensure that your loop stops execution.

Listing 4-9a. Simple while loop example

```
number = 0
while number < 10:
    print(number)
    number += 1
```

It is very important to have a control variable that will stop the execution of your while loop. If you do not have this, your loop will keep on executing and become what is called *an infinite loop*. The biggest drawback of an infinite loop is that the rest of your code does not get executed, rendering your software useless.

Listing 4-9b shows a more complex example. Here `number` is still your control variable, but the condition governing the amount of times the loop will execute is more complex. You are not just evaluating whether `number` is smaller than 10; you also consider whether `number` multiplied by 10 is smaller than 50.

Listing 4-9b. Complex while loop example

```
number = 0
while number < 10 or number * 10 < 50:
    print(number)
    number += 1
```

For Loops

For loops are used to iterate over lists, tuples, dictionaries, and also strings, making a for loop best suited for when you have a fixed set of data to iterate through. The syntaxes to iterate over a list, tuple, dictionary, and string are mostly the same in a lot of respects, but there are different variations in syntax or ways to apply them, which we will cover here.

A for loop's anatomy is very similar to that of a while loop:

```
for expression:
    code
```

The use case for a for loop is different than a while loop so its expression will look different. In general, a for loop operates on a structure that is iterable. Iterable means you can iterate, or step through its data, like a sequence or a map. You will recognize the `in` keyword in the example below. It is the keyword you use to test for value membership in arrays and was mentioned in the section about operators.

```
students = ('Justin', 'Ron', 'Andy', 'Jonathan')
for name in students:
    print(name)
```

Here is a slightly more complex example:

```
for name in students:
    if name == 'Ron':
        print('Found Ron')
```

Dictionaries are iterated over differently from lists, tuples, and strings, due to the fact that they possess a key. Since a map is also an object, it has a function called `items()`. You use this function to extract the items from the map. You can also specify a key in your for loop if you want to see the key.

```
map_of_values = {'name': 'Andy', 'surname': 'Bieber', 'marital_
status': 'married', 'country': 'Great Britain'}
for key, value in map_of_values.items():
    print(key + ' ' + value)
```

List Comprehension (Shorthand Loops)

Python also has something called *list comprehension*. This is a list that creates itself based on an internal loop. This loop returns a list. Consider the code in Listing 4-10.

Listing 4-10. List comprehension

```
# The list we will be iterating over.
numbers = [1, 2, 3, 4, 5]

# Function that will be called in the loop
def multiply(amount):
    return amount * 10

# The loop is wrapped in square brackets, and that list is
# returned to the variable
# called 'results'.
results = [multiply(value) for value in numbers]
# This will contain a new list, where each value in the old
# list will be multiplied by 10.
print(results)
```

Continue and Break

Loops do have some more tricks up their sleeves. Three important things to look at is how you can nest loops and how to use continue and break clauses inside your loops. Nesting loop are not always ideal, but sometimes they are needed to achieve your goal. Let's create a complex list. This list

will have complex values as items, as opposed to just having integers or strings. This example has dictionaries as items. Note that `continue` tells the system to stop executing subsequent code and go back to the start of the loop and `break` tells the system to stop executing subsequent code and to leave the loop. This is demonstrated in Listing 4-11.

Listing 4-11. Continue and Break

```
people = [ {'name': 'ron', 'position':'middle'},
{'name':'nico', 'position':'bottom'}, {'name':'andy',
'position':'top'} ]

# One example using continue.
for person in people:
    for details_key, details_value in person.items():
        if person['position'] == 'bottom':
            continue
        print details_value

# One example using break.
for person in people:
    for details_key, details_value in person.items():
        if person['position'] == 'bottom':
            break
    print details_value
```

The sets of loops are very similar. Each set has two loops. The first loop (the outer loop) iterates over the list called `people` and sends back the values in the list one by one. The values it sends back are dictionaries. Then, in order for you to iterate the dictionary, you iterate over the values that the list sends back in your second loop. This line `person['position'] == 'bottom'` just means that the variable `person` is a dictionary (since you received `person` from the list, and the list contains dictionaries). If the value at the key called `position` is equal to `'bottom'`, then run the `continue`

clause. This ignores whatever is next in the loop and goes back to the initial loop and carries on iterating. The second example uses `break`. This is exactly the opposite, and it breaks out of the control structure, effectively stopping the loop from iterating, and continuing outside the scope of the loop.

Functions

Functions are small reusable blocks of code, which once defined can be accessed by using their name. They consist of a name, a parameter block, which is always between round brackets `()`, and a function body. All the concepts you have encountered in this chapter can be applied inside a function. You can, and will, write your own functions, but Python comes with built-in functions as well. You encountered a built-in Python function in Chapter 3 called `print`. Here is an example of the `print` function again:

```
print('Hello world')
```

I will not dig into all of the functions that Python provides, but I will introduce more functions as we proceed through the chapters of this book. As a rule of thumb, if you want Python to achieve something small, see whether Python has a function for it first.

Custom Functions

Custom functions are something you will write constantly. We write functions for a few reasons.

- It helps us reuse code. We can call the function numerous times.
- It encapsulates tasks, making our code easier to understand.

Functions should be designed to do one of three things:

- Print something to the screen (not always recommended).
- Return data so that the code calling the function can use it.
- In the case of classes, an object can update the internal state of the class. These functions can still return data as well.

Principles of Function Design

When designing a function, it is very important that the function achieves one goal, and one goal only. For instance, if you write a function that calculates VAT, then that function should only calculate VAT. If you first need to sum a bunch of values and then calculate VAT, you should have two functions: one to sum the values and one to calculate VAT on that summed value. One function to sum the values and then calculate the VAT amount sounds innocent enough, but detracts from the understandability of the code and opens you up to bad coding practices.

The Anatomy of a Function

A function's identifier is the word `def`, followed by the function name, parameter list (which is between the round brackets), the function return type (which is optional), and the function body. When you need to get data into your function to do operations on, you pass it in via the parameter list.

- The keyword `def` is the statement declaring a function that follows it.
- The function name is used to access the function.

- The parameter list, between (), takes external data and hands it to the function.
- The optional return type tells the function that it may only return data of that type.

The parameters are also called arguments, and the order in which they are entered into the parameter list, matters. The order in which they are specified when the function is designed, is the order in which you should provide them to the function when the function is used. The function name, and arguments, should be meaningful names reflecting their purpose. The parameters, or arguments, are values that you receive from the program, and many times this is data provided by user input. This data will be passed into your function to do calculations on. The result of the calculations will be passed back to the system, helping you achieve the goal of manipulating the program's state.

```
Def function-name(argument1, argument2) -> function_type_to_
return :
```

Code you have written.

Code you have written.

Code you have written.

Return or print

Should Functions Return or Print?

This question is a design issue. It depends on what the function needs to do. In general, I prefer letting my functions return data instead of printing it, as displaying data should not necessarily be the job of that function, plus the function calling your function may need to do some more calculations on the returned data, something it cannot do if you print data to screen. There is another option, though, which you will come across when you look at classes.

Example Functions

Listing 4-12 shows a simple function that just prints a name and surname. This is just to illustrate how parameters are passed into a function.

Listing 4-12. A simple function

```
def name_and_surname(name, surname):
    return 'Your name is ' + name + ' and surname is ' +
        surname

print(name_and_surname('Nico', 'Loubser'))
```

Listing 4-13 shows a slightly more complex example that does some calculations.

Listing 4-13. A more complex function

```
# Import the datetime library
import datetime
"""
Here the function name, calculate_age, reflects what it will be
used for.
The one argument it accepts, year_born, is named after the
value you want.
It should always return an integer due to the return type being
specified
"""
def calculate_age(year_born) -> int:
    current_year = datetime.datetime.now().year
    # This function returns a value. Y
    return (current_year - year_born)
```



```
# Here we are calling the function. It returns the age and
stores it in the variable
# called 'age'
age = calculate_age(2000)

# Print age to screen
print(age)
```

Let's have a little test. You can either try to create it yourself, taking into consideration that you may not have encountered all of the aspects inside the solution, or you can read the question and see how the answer relates to the question. You have a person (and you will provide their name as a function argument) who has x amount of money (which you will provide in the function's second argument) in their wallet. They want to buy a cup of coffee that costs \$4. Write a function that accepts their name and available money as arguments. The system should determine whether they have enough money or to buy coffee or not. The function should return the string "Janet (or whatever name you gave her) has enough money" if she has more than \$4, or "Janet does not have enough money" if she has less than \$4.

Here is a possible solution:

```
def enough_funds(name, funds):
    if (funds >= 4):
        return name + ' has enough money'
    else:
        return name + ' does not have enough money'

# To run this function
print(enough_funds('Jane', 3))
print(enough_funds('Jane', 10))
```

Here the custom function called `enough_funds` returns the result to a built-in Python function called `print()`, which you have encountered already. `enough_funds` does the calculations, and `print()` returns the answer to your screen.

Parameter Type-Hinting

When defining parameters in your function, you can specify the type of parameter you want. This is called *parameter type-hinting*. For instance, if you want `age` to come through as an integer, and not ever as a string, you define your function as follows:

```
def set_age(age: int)
```

Default Parameter Values

If you do not provide values for your parameters, your program will not execute and it will complain that some arguments are missing. You can provide default values in situations where you are unsure whether you may receive all parameter arguments or not. Although this is not a good method to ensure your code does not break when data is missing, it is a great way to ensure you have one or two default values that do not need to be provided if they are not available.

The function in Listing 4-14 will either print directly to screen or return the value to the code calling this function. You use a parameter with a default value called `return_error`, which is set to `True`, in the parameter list to indicate that it should always return its data, unless you provide a different value for the `return_error` parameter in the argument list.

Listing 4-14. A function with a default parameter value

```
def log_error(error, return_data=True):  
    if return_data:  
        return error
```

```

    else:
        print(error)

# Equivalent calls, prints directly to screen
log_error('error message 1')
log_error('error message 2', True)
"""
Returns its variable to the code calling the function, where
you can do more operations on it or in this case, just print it
"""
error = log_error('error message 3', False)
print(error)

```

Variable Parameters

If you do not know how many parameters you will send to your function, you can use variable parameters. This is handy if your function is expecting *x* amount of similar data, but you do not know how many values there will be. To illustrate, look at the following function in Listing 4-15. The variables are added as individual variables, but inside the function they are treated as a list. The variable parameter is prepended with an asterisk, ***.

Listing 4-15. Variable parameter example

```

def sum (*numbers):
    total = 0 # initialise total to 0
    for number in numbers:
        total += number
    return (total)

print(sum(10,56))
print(sum(1,2,3,4,5,6,7,8,9))

```

Named Keyword Arguments

At the start, I mentioned that the order of the arguments matter when they are passed through to the function. This can be circumvented by using the name of the argument in the function call. In this case, Python does not care about the position of the argument, but at the name. See Listing 4-16.

Listing 4-16. Named keyword arguments

```
def details(name, surname, mobile_number):
    print(name + ' ' + surname + ' mobile :' + mobile_number)

# Run these two functions and see how the first one basically
# breaks the output because
# the order is wrong
details('Johnson', '0123456789', 'Don')
details(surname='Johnson', mobile_number='0123456789',
name='Don')
```

Classes and Objects

Python is an object-oriented programming language. You have read the terms “object” and “class” a few times in this chapter and have encounter them briefly with sequences and maps. A class is the code you wrote, and an object is that class that has been loaded into memory as a useable object. A class encloses, or *encapsulates*, behavior and data. Let’s take a real-world example of a dog. As a class, a dog encapsulates the following behavior and data (attributes):

Behavior

Barking

Running

Attributes

Name

Breed

Color

Your dog class can be written as shown in Listing 4-17. We will go through more examples after this example.

Listing 4-17. The Dog class

```
class Dog:

    def __init__(self, name, breed, color):
        self.name = name
        self.breed = breed
        self.color = color

    def bark(self):
        print('woof')

    def run(self):
        print('Running')

# End of the class
```

Creating a new object is called *instantiation* and looks like this:

```
variable = Class()
```

So, in your case, your dog object is instantiated like this:

```
dog = Dog('Fluffy', 'Poodle', 'white')
```

You can now run the encapsulated functions on the dog object using the dot operator:

```
dog.bark()
```

When we instantiate a class, we create an instance of that class via the constructor. It is optional to include your own constructor, and it is dependent on the class you created. The constructor is defined inside the class using the `__init__` function. All classes that have a constructor use this method.

```
def __init__():
```

This function can be used to declare and initialize all of the supporting classes and variables that it needs to create this class. You can see a constructor as taking materials and constructing your class into an object using those materials. Say you have a class called `Rectangle`. You create its constructor as follows:

```
def __init__(self, width, height):
    self.width = width
    self.height = height
```

Just to rehash, classes are used to create objects. An object is an instance of a class, and the variables passed into the instance are called instance variables. A class can be seen as the code that was written down when you created the class. An object is a class after it has been created and loaded into memory; this is also called instantiation. An object is referred to as an instance of a class. An object, just like an integer, string, and so on, is assigned to a variable. So, in short, you write a class and then instantiate the class into an object.

The Anatomy of a Class

Let's go through a class definition.

```
"""
```

class is the identifier for a class declaration. We use the 'self' keyword in order to

have access to the methods and properties inside a class. You do not need to pass the keyword 'self' in when you call the class or its functions.

Constructor. A very important aspect, a constructor is created using `__init__()`.

The `__init__` function runs on every instantiation of the class into an object, and is a great place to get data and dependencies into your object.

"""

class MyClassName:

 # Optional constructor. Gets called automatically, and always takes 'self' as a

 # parameter. The 'self' keyword allows us to access the properties and

 # functions inside the class

def `__init__(self, value):`

 # property is an instance variable of MyClassName
 self.property = value

 # Any amount of functions can follow

def `function1(self):`

 actions

 actions

def `functionN(self, param):`

 actions

Instantiating the Class

The round brackets in an instantiation refer to the constructor. If the constructor has parameters, then you pass it in here. You do not pass `self` because it is already implied.

```
a_class = MyClassName(value)
```

`a_class` is now a variable of type `MyClassName` and has access to the functionality inside it. To access that functionality, you use the dot notation:

```
# Accessing a function
```

```
a_class.function1()
```

```
# Accessing a property
```

```
a_class.property
```

Let's look at an actual example of a class in Listing 4-18. In it, you create a class that calculates the amount of money left in a bank account.

Listing 4-18. The Funds class

```
class Funds:
    # These two variables are in the class scope. Initialised
    # to 0
    total_expenses = 0
    total = 0

    # Constructor. We populate it with the total amount
    # available when we
    # construct the object
    def __init__(self, total):
        self.total = total

    # Stores the money we have spend
    def set_expense(self, expense):
        self.total_expenses += expense

    # Calculates the amount of money we have left
    def get_funds_left(self):
        return self.total - self.total_expenses
```



```
# Code implementation part. Note that you do not have to put
'self' when calling the
# functions
funds = Funds(200)
funds.set_expense(10)
funds.set_expense(15)
# Prints the amount left to screen
print(funds.get_funds_left())
```

You can make this class a bit better, though, so see Listing 4-19. It is not very reliable that you can just deduct money and take your total expenses into the negative. You will also add code to handle the errors a bit better. Keep an eye on the indentation as well; I have exaggerated the indentation to make this a bit easier to follow.

Listing 4-19. The improved Funds class

```
class Funds:
    total_expenses = 0
    total = 0
    # Added a variable to hold any error messages we may
    encounter
    error = '';

    # Constructor. We populate it with the total amount
    available
    def __init__(self, total):
        self.total = total

    # Sets the money we have spend
    def set_expense(self, expense):
        # Checks whether we have enough money left to give out
        if self.get_funds_left() > expense:
            self.total_expenses += expense
            return True # Exit the function
```

```

        # If we don't, set the error message and return false.
        Returning false here
        # allows us to detect a failed expense deduction
        self.error = 'Out of funds'
        return False

    # Calculates the amount of money left
    def get_funds_left(self):
        return self.total - self.total_expenses

    # Returns the error
    def get_error(self):
        return self.error

# Code implementation part
funds = Funds(20)

# Check whether the set_expense function returns True or False.
Using the 'not'
# keyword indicates false. Then we print the error.
if not funds.set_expense(10):
    print(funds.get_error())

if not funds.set_expense(2):
    print(funds.get_error())

if not funds.set_expense(15):
    print(funds.get_error())

# Prints the amount left to screen
print('You have £' + str(funds.get_funds_left()) + ' left.')
```

Inheritance

We will look at inheritance in this chapter. Classes can be structured hierarchically, in an *is-a* relationship. This is often called a parent-child relationship. The child will inherit behavior from the parent. Let's say your parent class has a function called `email()`. The child can use that `email()` function as it is, or override the parent's function with its own function. This is super easy; just redeclare the exact same function in the child class and give it another body.

Let's say you have a bicycle shop and need to create software to order bike parts. One way to model the classes for bicycles is shown in Listing 4-20. You have a top-level class called `Cycle`. Below is what the `Cycle` class may look like. This is almost a template with the functionality of what the children, the classes that inherits from it, should look like.

Listing 4-20. Inheritance

```
class Cycles:
    def set_as_assembled(self, is_assembled):
        self.assembled = is_assembled

    def get_wheel_count(self):
        return self.wheel_count

    def who_am_i(self):
        return 'I am the original function'

# Note the change in how we define our class. We added round
# brackets and placed
# the parent it inherits from name in between it.
# Monocycle IS-A Cycles and inherits from Cycles
class Monocycle(Cycles):
    # Override the parent function with a similar function that
    # behaves differently
```

```

    def who_am_i(self):
        return 'I am overriding the parent function'
    wheel_count = 1

# A second child class. Bicycle IS-A Cycles
class Bicycle(Cycles):
    wheel_count = 2

# A third child class. Tricycle IS-A Cycles
class Tricycle(Cycles):
    wheel_count = 3

monocycles = Monocycle()
print(monocycles.get_wheel_count())

cycles = Bicycle()
print(cycles.get_wheel_count())

# In this section we show how the function overriding works
print(monocycles.who_am_i())
print(cycles.who_am_i())

```

In this block of code, you have three classes of type cycle, and all three have an is-a relationship with the parent class called Cycle. Notice how neither Monocycle, Bicycle, nor Tricycle objects implement a `get_wheel_count()` function, yet they can all use that function. They all inherit it from their parent class called Cycle. Also of interest is that the function called `who_am_i` is overridden in the Monocycle class, but not in the Bicycle class. In the last two lines of Listing 4-20, you can see how Monocycle uses the overridden code and Bicycle the original code.

Polymorphism

Polymorphism is the ability of an object to take on many different forms. It can be a complicated subject but is often explained in the sense of animals. Listing 4-21 shows a quick example. You have a parent class called `Pet`, and `Pet` has two children, `Cat` and `Dog`. The function `getSound` takes a parameter of type `Pet`, of which you have two, a cat and a dog. Polymorphism resolves the type for you.

Listing 4-21. Polymorphism

```
# Parent object
class Pet:
    def sound(self):
        # Pass is called a null statement in Python, and
        # nothing happens when Python encounters it
        pass

# IS-A pet
class Cat(Pet):
    def sound(self):
        return 'Meow'

# IS-A pet
class Dog(Pet):
    def sound(self):
        return 'Woof'

# Function getSound receives a Pet as a parameter,
# but we are not specifying what kind of pet. Polymorphism is
# used to resolve this
def getSound(pet: Pet):
    return pet.sound()

print(getSound(Dog()))
print(getSound(Cat()))
```

In Listing 4-21, polymorphism is achieved by the is-a relationship as well, where the parent can take many forms via its children. This allows us to use the same functions on the children but let those functions yield different outputs.

Composition

Composition is an easy-to-understand design concept, and it's not related to Python functionality per se. It is a design aspect. This will be covered in a later chapter.

Magic Methods

Magic methods are methods that you declare inside a class, and write code for, but you never call explicitly yourself. You can find them in other programming languages like PHP as well. They act like a sort of a catch-net for certain scenarios, so when Python encounters the scenarios linked to the class they were declared in, it knows what magic method to execute. There are quite a few of them. You can see three interesting ones in Listing 4-22. An explanation of what the different magic methods are doing is provided above the function.

Listing 4-22. Magic methods

```
"""
```

```
Class Member consists of a constructor that sets a name.  
We will use this to demonstrate the magic method for operator  
overloading
```

```
"""
```

```
class Member:  
    def __init__(self, name):  
        self.name = name
```

```
"""
```

Class Group contains three magic method. You have already encountered `__init__`. This is the constructor, and is called whenever you instantiate an object.

```
"""
```

```
class Group:
```

```
    def __init__(self):
        self.members = []
```

```
    """
```

`__add__` is a magic method that does operator overloading. Overloading means we add new behaviour to an already existing operator or function, and call that new behaviour under specific conditions. In this case it overloads `+`, and inside the body of the `__add__` function we add the logic of what the overloaded `+` must do. We need to specify in the parameter list, as the second parameter, what is to be expected on the right hand side of the `+` sign. In this case it is something of type `Member`. All we want to do when we add something of type `Member` to type `Group` is to take `Member`'s name parameter and add it to `Group`'s list of names. Whenever Python sees a `Group` object followed by a plus, it will run the `__add__` function.

```
    """
```

```
    def __add__(self, x: Member):
        self.members.append(x.name)
```

```
    """
```

`__str__` is a magic method that returns a string whenever you treat your object as a string. Whenever I say for instance do this. `print(MyObject)`, then `MyObject` will run the `__str__` function.

```
    """
```

```

    def __str__(self):
        return ','.join(self.members)

group = Group()
member1 = Member('nico')
member2 = Member('john')

group + member1
group + member2

print(group)

```

The output of this function code is: nico,john

Whenever a Group object encounters a + sign, for instance : 'group + member1', it will run the `__add__` function which we added ourselves. This takes the member object to the right of the + sign and add it to an array maintained inside the Group object.

Exceptions

Exceptions occur when your system encounters an error or goes into a state from which it cannot recover. You will often encounter exceptions. Exceptions can be system generated, but you can also generate them yourself, as well as write them yourself. Exceptions are cases where your software encountered a problem that was not anticipated and should not be handled in a normal way. Attempting to divide a number by 0 is a very common case that throws an exception.

Let's take the example of a system that orders wheels for bicycles. If you order wheels, and there are wheels in stock, your software should place the order and return with a value of True. If there aren't any wheels in stock, the software should not place the order and should return with

a value of `False`. Sometimes, a software developer will let the system react with an exception, but this is wrong in my opinion. Wheels being out of stock is a predictable situation and can be handled normally. But let's say, for instance, your system cannot communicate with the system that places the order for the wheels. This would be a great place to put an exception. Not being able to place an order because a sub-system was not found is an "exceptional" case.

Think carefully about the following when using exceptions:

- Is it a unique enough case that I should create my own exception? Can it be dealt with better in another way?
- When an exception occurs, must the system be allowed to go on, or should it completely abandon execution? Some errors may leave the system in a unusable state, such as not being to find the wheel ordering system, and the system cannot continue, it can only retry or stop.
- Log your exceptions to a logfile, or create a way to notify someone. Exceptions mean something went wrong and people may want to know about it.

The Anatomy of an Exception

Exceptions are handled by `try/except` blocks. The `try/except` structures are wrapped around code blocks that may potentially throw an exception. As with all Python code, the code blocks are indented one level deeper than the `try/except` code around it. At a basic level it says *try this code, but if it throws an exception, then do something else*. A very basic construct will look like this:

```
try:
```

```
    Some code that throws an exception
```

```
except:
```

```
    Do a recovery action here. All exceptions will be handled here.
```

```
finally: #optional
```

```
    This action will always be executed.
```

A more complex example will look as follows. This example specifies which specific exceptions are to be caught:

```
try:
```

```
    Some code that may throw an exception
```

```
except specific_exception as e:
```

```
    Handles specific_exception
```

```
except BicyclePartSystemNotFoundException as e:
```

```
    Handles BicyclePartSystemNotFoundException
```

```
except:
```

```
    Handles all other exceptions
```

```
finally: # Optional
```

```
    This action will always be executed.
```

Raising an Exception

You can raise Python's built-in exceptions in your code. There are quite a number of exceptions, which I will not list here. Raising an exception is as simple as locating the area of code where the exception should be raised and using the `raise` keyword, as follows:

```
raise Exception('custom optional error message')
```

```
# It is preferable to always have a meaningful error message,  
as it will help you
```

```
# debug your problem better.
```

Catching an Exception

You can catch an exception anywhere in your code. It does not need to be where the exception gets raised. At the point of catching it, you should define clearly what should happen to the execution flow of the program. Let's take the division-by-zero error. Should your system be able to continue if it tries to do division, but encounters a 0 and cannot do the division? This is one of those questions you have to ask yourself when you create your software. You will encounter questions like this a lot. But you have options. You can, for instance, log the error to a file or email it to yourself. You may even take the option of a default value denominator, as shown in Listing 4-23. In Listing 4-23, you take an amount of money and calculate how much each friend gets. If friends are 0, then the owner keeps all the money.

Listing 4-23. Catching an exception

```
wallet = 100
friends = 0
try:
    per_friend = wallet / friends
except ZeroDivisionError as e:
    # if friends are 0
    per_friend = wallet / 1
print(per_friend)
```

Writing an Exception

Sometimes you may want to write your own exceptions. It is very handy to have custom exceptions because it helps separate different error states and helps you handle errors on a much more specific level. Let's have a look at writing, raising, and catching your own exceptions in Listing 4-24a.

Listing 4-24a. Writing an exception

```

"""
Here is a very simple exception. Note the class declaration.
Its parent class is the
built in class called Exception. Extending on the parent class
'Exception' helps us create a class with an IS-A relationship
with exception. Pass is a null statement, and nothing happens
when Python encounters it.
In this instance, we also pass the MyError exception a null
statement. We do however still get the benefit of a custom
exception, however we are not overriding any built in Exception
functionality.
"""
class MyError(Exception):
    pass

# Test the exception
name = 'pierre'
try:
    if name != 'john':
        raise MyError('Name is not equal to John')
except MyError as e:
    print(repr(e))

```

Imports

In a larger system, your classes will all be in their own separate files and directories, where they can be imported from. They are called packages. A package is a collection of related scripts that is grouped together in a directory. For a package to be importable, it needs a file called `__init__.py` in that directory.

Why do we need importing? A Python script can only see itself, and because of this we need to import files to allow it to access other functionality. This also keeps your codebase clean and allows you to separate your files into directories where it makes sense for them to live. After we have moved our files around, we need to make them available to our `main.py` script. Most systems will have one central point of execution, where all the execution calls are handled and the correct objects are built. In our case, it is `main.py`.

You will explore imports by using a custom exception file. In the directory where `main.py` is, create a new directory called `errors`. Create an `error.py` file inside this directory containing the code found in Listing 4-23. Inside the `errors` subdirectory, which should now contain a file called `errors.py`, create a new file called `__init__.py` and leave it empty. There should be two underscores on each side of the word `init`. The `__init__.py` file tells Python that the `errors` directory is a package and can be imported from. Finally, you need to add an import statement to your `main.py` script, which will take this form: `from package.filename import class`. See Listing 4-24b.

Listing 4-24b. Imports

```
"""
This is the same code as in Listing 4-24.a, Wit the exception
that following line now lives in /errors/errors.py.

class MyError(Exception):
    pass
"""

from errors.errors import MyError

# Test the exception
name = 'pierre'
try:
```

```

    if name != 'john':
        raise MyError('Name is not equal to John')
except MyError as e:
    print(repr(e))

```

Your code should be able to find the correct package to import from and execute the code without a problem. That was quite easy. In big systems and especially old systems, there can be hundreds or even thousands of files in different folders. Keeping them separated in logical divisions is very important. I won't be going in depth with all the intricacies of importing a file in Python, but I wanted to show a more interesting way where you created a package.

If you run the code now, it will import the `Exception` class from a different directory.

Static Access to Classes

You can access functions and elements on a class level. That means on a level where the class has not yet been instantiated into an object. This is valuable for a lot of reasons. For instance, you may need some functionality that does not require you to construct a whole object. Take the following example. When you use a class statically, there is no object to which the keyword `self` can refer. For these reasons, when you want to create a static function, you need to design for it. See Listing 4-25.

Listing 4-25. Static access

```

class VAT:
    vat_rate = 15

    # This function can be used statically. No 'self' is being
    # passed through.
    def static_get_vat_price(price):
        return price * (1.0 + VAT.vat_rate/100)

```

```

# This function cannot be used statically, as it is reliant
on self, and
# self only
# exist when the object gets created
def get_vat_price(self, price):
    return price * (1.0 + VAT.vat_rate/100)

# These two examples will work. No object instantiation has
happened
print(VAT.vat_rate) # Prints the vat rate
print(VAT.static_get_vat_price(50)) # Calculates the amount
after after VAT

# This wont work without object instantiation and will throw an
error, the parameter self does not exist
print(VAT.get_vat_price(50))

```

So basically, whenever you need to use the self keyword, or are completely reliant on a constructor, you cannot access functions statically. But variables are a different story, as long as they do not need to get assigned via the self keyword.

Cheat Sheet

Scope

Always nested with four spaces, or one tab, but not both. The deeper the scope, the deeper the nesting.

```

if True:
    if True:
        if True:
            print('all true')
print('In the same scope as the first if')

```

Variables

Variables are to the left of the assignment operator.

- String = 'string' or "string"
- Integer = Numbers like 18
- Float = Numbers with decimal points like 24.50
- Boolean = True or False, which are always capitalized.
- Object = Your own custom object

Arrays

Lists

- Can be resized
- Number-based indexing starting at 0

```
my_list = [1,4, 'my value']
```

Tuples

- Cannot be resized
- Number-based indexing starting at 0

```
my_tuple = (2,4, 'my tuple')
```

Dictionaries

- Can be resized.
- Indexed by an explicit key. Key and value separated by a colon.

```
my_dictionary = { 'index1':12, 'index2': 'value 2' }
```


Control statements

if

```
if expression:
    Something
elif another expression:
    Do something else
else:
    Do something else
```

while

Runs while a condition is equal to True:

```
While mathematical expression is true:
    Do something
    Do something to exit the loop
    (Either alter the expression, or use break)
```

for

In general, used to iterate over sets of data, like lists, tuples, and dictionaries.

```
for value in set_of_data:
    print(value)
```

Functions

```
def function_name(parameter1, parameter2):
    function body
    function body
    return out of function
```

Calling the function:

- `function_name(variable1, variable2)`
- `function_name(parameter2=variable2, parameter1=variable1)`

Type hinting:

- `function_name(variable: int, variable2: str):`
- `function_name(variable: int, variable2: str)->str:`

Classes

```
class MyClass:
    properties

    # Optional constructor
    def __init__(self):
        constructor body

    # Class functions Self gives us access to the class
    def class_function(self):
        function body

    # Can be called statically
    def static_function():
        function body
```

Instantiating a class:

- `object = ClassName()`

Calling a function on an object:

- `object.function()`

Calling a function on a class:

- `Class.static_function()`

Exceptions

Catching an Exception

```
try:
    Code
except:
    Handle exception
finally: #Optional
    This code will run regardless
```

Raising an Exception

Use the raise keyword plus the name of the exception:

```
raise exceptionName
raise exceptionName('optional error message')
```

Creating an Exception

```
# Declare a function using the class Exception as parent
class myException(Exception)
```

Import

```
from package.to.filename import class
```

Reference

A good place to read about Python development: <https://docs.python.org/3/tutorial/index.html>.

CHAPTER 5

Object Calisthenics, Coding Styles, and Refactoring

This chapter discusses how code is written and consists of three parts: object calisthenics, coding styles, and refactoring. This chapter will not concern itself with the design aspects of problem solving. It will purely look at your code's readability and how easy it is for other software developers to understand your code. Harold Abelson, author of *Structure and Interpretation of Computer Programs*, famously said, "Programs must be written for people to read, and only incidentally for machines to execute." And I must agree. Readability is a very important aspect of writing code. Writing code should always be an exercise in elegance, but this is easier said than done. Fortunately, there are some solid guidelines set out to help you achieve that goal.

From a code-writing perspective, there are numerous ways to solve a problem using code. Let's assume two different coders have come up with exactly the same algorithm to solve a problem. (An *algorithm* is the set of steps required to solve a problem.) Do you think their code will look exactly the same? The answer is no. In some cases, the code will look so different that a cursory glance won't be able to tell you that the code is solving the same problem. After a quick inspection, most seasoned

developers will quickly be able to point out whose code they favor. This bias is not something you can escape as a software developer. Your code will be critiqued, sometimes harshly, sometimes lightly, and a lot of times positively.

Writing easy-to-read and concise code is not a maze you need to traverse all on your own. There are styling tips that will greatly enhance your code's look and readability if you stick to them. After you have written your code, it is also a great idea to take some time to refactor your code. Refactoring your code is in essence equal to rewriting portions of it, using the code already written as a template. We will also look at object calisthenics, which is a set of guidelines designed to make your code more readable. I am personally very fond of object calisthenics and quite excited to add it to this chapter.

When you start to work as a software engineer, you should always take advice from your fellow developers, regardless of their level of experience. Try not having an ego about the software you write, and rather listen to any critique and advice, and evaluate if it is valid. If it is valid, take it on board. This is one way you will grow as a software developer.

Object Calisthenics

Object calisthenics is a set of nine guidelines to make your code more readable and maintainable. It is important to remember that these are guidelines, and not hard-and-fast rules, but it is well worth trying to follow them as closely as possible. Jeff Bay proposed object calisthenics in the book *The ThoughtWorks Anthology*. As with quite a few things in life, do not use them in instances where they just do not make sense. Having said that, when object calisthenics do not make sense to apply, you may want to take a step back from your work and rethink your code. Is your code so entangled and complex that you cannot apply calisthenics? In most instances, these guidelines will make sense and are pretty easy to follow.

However, I need to reiterate: if it truly doesn't make sense in a specific instance, then do not use an object calisthenic rule just for the sake of using it.

In this section, I will explain eight of the nine rules of object calisthenics with some examples. I am omitting the ninth rule as I believe it may cause some confusion.

1. Do not exceed one level of indentation per method. (Or rather, limit the levels of indentation as much as you can.)

Limiting levels of indentation means you are limiting the complex paths that your code can take, or at least, making the execution path through your code more readable. More nested levels mean that you inevitably have more control structures within other control structures, all increasing the number of potential paths your code can take through the method. Consider Listing 5-1. It has two functions doing the same thing. The first function is called `more_indentation` and the second function is called `less_indentation`. `more_indentation` relies on indentation to reach the goal of the function, whereas the function `less_indentation` is refactored to have only one level of indentation. Comparing the two functions side by side, it becomes clear that the `less_indentation` function is easier to read.

Listing 5-1. Comparing indentation levels

```
def more_indentation(number, name):  
    if number > 10:  
        if name == "John":  
            print("Number is bigger than 10 and Name is John")  
        if name != "John":  
            print("Number is bigger than 10 and Name is not John")
```

```

        if name == "Peter":
            print("Number is bigger than 10 and Name is
                Peter")

def less_indentation(number, name):
    # Exit early. Inspecting the above code we can see that if
    # the number
    # is 10 or lower, then nothing happens. We invert the logic
    # in the previous if
    # statement to detect if number <= 10
    if number <= 10:
        return

    # Initialise is_john to a default message. Now is_john has
    # a value
    is_john = "Number is bigger than 10 and Name is not John"

    if name == "John":
        is_john = "Number is bigger than 10 and Name is John"

    print(is_john)

    if name == "Peter":
        print("Number is bigger than 10 and Name is Peter")

# The two functions will yield exactly the same results
more_indentation(11, "John")
less_indentation(11, "John")

```

2. Do not use the else keyword

So you just learned about the `else` keyword in the previous chapter, and now I say you should not use it? Well, it is not that the `else` keyword is bad, but it does not aid the readability of your code. There are two methods I

use to prevent myself from using the `else` statement. One is returning early from the function, but not everyone likes this approach. The main reason normally given is that it makes very long functions harder to understand, as multiple returns makes it harder to follow the logic in the function. I believe, however, that if you have a long function, that not returning early will add more complexity to the code, as you need to wrap *if* statements around the code that would have been skipped by an early return, in order to prevent those lines of code from being executed. The other approach is to set a variable to a default value and see whether the single *if* condition changes it. Listing 5-2 demonstrates this.

Listing 5-2. Omitting the `else` keyword

```
number = 10
if number == 10:
    print(10)
else:
    print('not 10')
...
```

First refactoring initialises a default value before we check for 10

```
...
```

```
message = 'not 10'
if number == 10:
    message = '10'
print(message)
...
```

Second refactoring returns early when 10 is found. This works great in a function.

```
...
```



```
def check_ten():
    if number == 10:
        return '10'
    return 'not 10'

print(check_ten())
```

3. Wrap all primitives and strings

When you use primitives (which are built-in data types like boolean, integers, floats, and strings), if they have behavior or represent a business idea, they should be encapsulated within their own object. Let's take a password, for instance. A password is just a string, so how does a password have behavior? Your password can exhibit behavior such as validating password complexity and checking that you are not reusing a password. If you just use password as a string attribute in a User class, your password validation leaks into the User class, which is undesirable. See Listing 5-3.

Listing 5-3. Wrapping primitives and strings

```
'''
This class technically consists of one attribute, a string, and
two behavioural aspects, which is checking the validity of the
password, and whether the password has been used previously.
Actual algorithms are not included for brevity.
'''

class Password:
    password = ""

    def __init__(self, password):
        self.password = password
```

```
def is_valid(self) -> bool:
    # Password goes here
    return True

def is_reused(self) -> bool:
    # Reuse algorithm goes here
    return False
```

Your subsequent classes, which need password validation, do not need to handle passwords as a string, or have their own validation rules, but can now just use this class, which has its own internal behavior.

4. Use only one dot per line

This rule is not about the dot operator. It is about what your class may access. Your class may access its own functions. It may access its own properties, and it may access functions in class properties that it owns. For instance, if you have class A, which uses class B, and class B uses class C, then class A should never talk to class C directly through class B. It is only allowed to access class C's data indirectly through functionality provided in class B. Class A is only allowed to know what goes on inside itself and what its direct members allow. Consider Listing 5-4. Although the guideline states “only one dot per line,” you will mostly have two dots as you also need a `self.` to access your object's scope.

Listing 5-4. One dot per line

```
class TotalCalculator:
    def get_value(self) -> int:
        return 100
```

MemberFees has a TotalCalculator.

```
class MemberFees:
    def __init__(self):
        self.calculator_object = TotalCalculator()
```

Club has a MemberFees.

```
class Club:
    def __init__(self):
        self.member_object = MemberFees()
```

Club accesses data from TotalCalculator, even though TotalCalculator

does not belong to Club.

```
    def get_total(self) -> str:
        return self.member_object.calculator_object.get_value()
```

```
c = Club()
print(c.get_total())
```

"""

Refactored MemberFees and Club with TotalCalculator code omitted for brevity. This demonstrates that class A should access class B only. How class B's function gets its data is of no concern to class A.

"""

```
class MemberFees:
    def __init__(self):
        self.calculator_object = TotalCalculator()
```

```
    def total_fees(self) -> str:
        return self.calculator_object.get_value()
```

```
class Club:
    def __init__(self):
        self.member_object = MemberFees()
```

"""

The following two code snippets does the same thing. The first example slightly deviates from object calisthenics, whereas the second example is done by the book. However, the first example is better than the first , since we use our constructor to create the member object.

We use two dots in this example, but fine as the first dot indicates that it is an object variable.

"""

```
def get_total(self) -> str:
    return self.member_object.total_fees()
```

"""

This is a good example of using the object callisthenics as a guiedline and only where it makes sense. Here we are using one dot, but this is not ideal. Not having MemberFees in the constructor in this case makes it more difficult to keep track of which objects belong to class Club. This would negatively affect how we understand the class.

"""

```
def get_total_one_dot(self) -> str:
    member_object = MemberFees()
    return member_object.total_fees()
```

```
c = Club()
print(c.get_total())
print(c.get_total_one_dot())
```

5. Do not abbreviate

Do not abbreviate object or function names. The names should be one or two words. Anything longer than that and you should consider why the name is too long. You should also not duplicate the class name in the method names. If your class is named `Email`, and it has a method that sends emails, you do not have to name the method `sendEmail()`. Just call it `send`. This makes perfect sense when you look at it like this:

```
email = new Email()  
email.send()
```

6. Keep entities small

One of the tenets of creating software is that the objects and their methods should remain small. An object should do only one thing (and of course, closely related actions). If your objects and methods are becoming too big, you should consider whether your object is doing too much or should be refactored. An object is typically considered as “doing too much” when it does more than one thing. For instance, if one object waters the plants as well as paints the house, then it is doing too much and should be broken up into two objects. Object calisthenics suggests that no class should be over 50 lines. This is not always possible, but stick to the gist of it: keep it small.

7. Limit classes to use no more than two instance variables

Before I explain this rule, just a summary about instance variables. Instance variables are the variables that we pass into our class at instantiation time. The following line of code demonstrates them:

```
def __init__(self, town, country):  
    self.town = town  
    self.country = country
```

The above snippet of code has two instance variables, given to it at instantiation time. It belongs to the class and is accessible after the class has been instantiated. This rule can be a tricky one and is down to how you designed your classes in the first place. This is also one of the harder guidelines to follow, and in some instances, you will not be able to follow it at all, as even with good design, you may need more than two instance variables. This is a concept you will encounter in another chapter, so I won't go into much detail here, but it all comes down to a principle called *low coupling*. The idea behind low coupling is to decouple your objects from other objects as much as possible. You need to prevent the reliance of one object directly on a multitude of other objects. You should also ask yourself, if your object is dependent on, say, six other objects, is your object doing too much?

8. Use first-class collections

This point states that if you have a collection inside your class, especially if that collection has behavior (for instance, a search function), then that collection should be encapsulated in its own class, and that class should not have any other variables. Listing 5-5 contains an object called `WinningSequence`, and `Lotto`. `WinningSequence` consists of a tuple and one behavioral aspect, called `validate`, which takes one parameter called `number`. This design alleviates the `Lotto` class from the burden of validating the `WinningSequence`, and instead the validation now rests on the shoulders of `WinningSequence`. The latter is now a completely reusable object, with its own behavior and can be used in various places in your code.

Listing 5-5. First class collections

```
class WinningSequence:
```

```
    winning_sequence = (1, 3, 12, 14, 15)
```

```
    def validate(self, number) -> bool:
```

```
        for in_sequence in self.winning_sequence:
```

```
            if in_sequence == number:
```

```
                return True
```

```
        return False
```

```
'''
```

The Lotto class does not concern itself with how Lotto numbers are checked. It uses the WinningSequence class to do so

```
'''
```

```
class Lotto:
```

```
    def __init__(self):
```

```
        self.sequence = WinningSequence()
```

```
        def is_winner(self, number: int) -> bool:
```

```
            return self.sequence.validate(number)
```

```
lotto = Lotto()
```

```
# Initialise the message variable
```

```
lotto_message = 'You do not have a winning number'
```

```
if lotto.is_winner(3):
```

```
    lotto_message = 'You have a winning number'
```

```
print(lotto_message)
```

The Lotto class now does not need to worry how the sequence is stored or validated, making the class easier to understand and change.

Refactoring Code

Let's look at refactoring since it goes hand in hand with writing good code and goes well with object calisthenics. Code refactoring actually goes very well with most aspects of software design, and you will get a better feel for it after you have done the chapters on design. Also, refactoring needs lots of practice. Refactoring is the act of actively changing the code you wrote to make improvements to it. Refactoring code is not the same as fixing a bug. Code that needs to be refactored is working code that yields the correct results but may suffer from efficiency problems, readability issues, or scalability issues. *Scalability of code* refers to how easy it is to add more features to your codebase. You will, in general, not know the code needs refactoring until you look at what was written and how efficient and readable it is. Code with bugs in it, on the other hand, is very obvious and needs to be fixed as soon as possible, as it yields incorrect results or acts erroneously under specific circumstances.

Refactoring is a fact of life. In fact, I see it as part and parcel of my daily coding tasks. There are various reasons to refactor your code. You may, for instance, write code just to prove that your algorithm works or for it to pass basic testing. You may even have written code that you realize is not optimal. At this stage, it is a good opportunity to stop development on further features and assess what you have written. It is highly recommended to refactor code that was written very recently before you start coding on new features. You want to prevent a situation where you need to refactor two months' worth of work or even a small code block that was written four months ago. Do it while it is still fresh in your memory, and above all, refactor as soon as possible to get the code done and signed off.

It is inevitable that you will stumble upon code that needs refactoring. It may be your own code, a colleague's code, or even code that is ten years old. In those cases, schedule time for refactoring. It is also very important to make sure you know exactly what old code was supposed to do before you refactor it. Refactoring should aid in readability, extendibility, and

reusability of code. Extendibility also refers to how easily new features can be added to your current code, and reusability to whether your code can be reused in different portions of the system. Of these three aspects, never underestimate the power of readability.

So, in short,

1. Plan your code.
2. Write your code.
3. Refactor your code.

As a last word on refactoring, you want to look at the readability of your code, the design of your classes, the efficiency of your algorithms, as well as any PEP violations, which I will discuss in the next section. Refactoring should change the written code itself, and not the behavior or the results that it delivers.

Coding Styles

Coding styles are rules on how certain aspects of your code should be styled and are all about the visual aspect of it. Unlike object calisthenics, they are rules to adhere to, and not guidelines. You will look at these rules in the rest of the chapter, but as a taster, they include aspects such as whether you capitalize certain parameters, function names, etc. The first time a coding style was enforced upon me, I thought it was lame. Who cares whether my variables are spelled all in lowercase or my class names start with capitals? I soon realized the value of it, and after only two or three weeks, I started noticing if my colleagues forgot to use the predetermined style, and they noticed when I omitted it. Quite a few programming languages have predetermined coding styles. Coding styles add nothing to the execution of your code but add a lot to the readability and understandability of it.

The official style guide to Python is PEP 8,¹ and some extensions and deviations of it exist. I will cover standard PEP 8 here, and I encourage you to download a PEP 8 cheat sheet from the Internet to help you out when you start programming, for quick reference. When it comes to a coding style for your code, it is most important that you remain consistent with whatever style you have chosen. You can create your own style as well, but adopting the general publicly accepted styles is a good idea, as it will aid you in reading code from projects that may very well have been written in PEP 8, or if you switch careers, you may end up with a company that uses PEP 8. You also get tools that can point out errors in your style, and in some cases, even fix them.

Linting

Linters are software tools that help us detect coding style violations. In Visual Studio Code, by default, a linter called Pylint should be enabled. In Visual Studio Code, you can access the linter by pressing Ctrl + Shift + p. This will open the command palette. Type *lint* in the text area of the command palette. You will see options such as Select linting, Enable linting, and Run linting. You will not work with linters now, as you need to learn how to code according to the standards first before you dabble in the world of linters. However, it is great to know they exist and that they can detect coding standard violations.

Commenting Your Code

Commenting tells us more about the code we wrote. It explains what classes can do, what methods can do, and what variables are for. It makes your code more understandable and helps other developers understand how to use the code. You can add block comments, inline comments, and docstrings.

¹www.python.org/dev/peps/pep-0008/

Comments should be complete sentences and use normal capitalization.

Block Comments

Block comments precede the code they apply to, can span many lines, and should be on the same level of indentation as the code they apply to. The following are two examples of block comments:

```
if is_member:
    # This sends an email to the member
    # The configuration for this function can be found in the
    # config directory
    send_email()
```

```
if is_member:
    """This sends an email to the member
    The configuration for this function can be found in the
    config directory
    """
    send_email()
```

Inline Comments

An inline comment is on the same line as the code it refers to and is only applicable to that one line. They should be used sparingly, because PEP also suggests that you do not type more than 79 characters per line, making inline comments less useful due to this short length, as well as being prone to breaking the line length rule.

```
if is_member:
    send_email() # This sends an email to the member
```

Docstrings: Commenting Your Classes

Docstrings are used within classes and class methods to explain the functionality of that class and its functions and variables. There is more than one standard of formatting your docstrings. When selecting a docstring format, bear in mind that readability is key, and above all keep your docstring format consistent. Another great feature is that most editors can read docstrings and give you the docstring description when you implement your classes. With that in mind, select a docstring that your editor can read. The following docstring formats exist:

- EpyText
- reST
- Google's style of docstrings
- Numpydoc

I selected Google's style of docstrings² because I feel it is the most readable docstring format. I cannot go over all the styles in this book, and you should do some research into which one you like. Figure 5-1 shows an example of a Google style docstring.

²<https://google.github.io/styleguide/pyguide.html>

```

1  class BankAccount:
2      """A very short description of the class fitting on one line.
3
4      Example:
5      Usage examples can go here.
6
7      For instance
8      bank_account = BankAccount('John', 15342);
9      bank_account.transfer(843675421)
10
11     Attributes:
12     account (the type of attribute, for instance, int): Description of the account attribute.
13     valid (boolean): Contains the result of whether the account is valid or not
14
15     """
16
17     def __init__(self, user, code):
18         """Short description goes here. This function transfers money. This can also
19         go in the class-level docstring
20
21         Note:
22         Do not include the 'self' parameter in the 'Args' section.
23
24         Args:
25         user: The account number.
26         code: The verification code
27
28         Returns:
29         True if successful, False otherwise.
30         """
31         self.code = code
32
33
34     def transfer(self, account_number):
35         """Short description goes here. This function transfers money.
36
37         Note:
38         Do not include the 'self' parameter in the 'Args' section.
39
40         Args:
41         account_number: The account number.
42
43         Returns:
44         True if successful, False otherwise.
45
46         Raises:
47         Exception raising information goes here
48         """
49         return True

```

Figure 5-1. Google docstrings

The docstrings should explain the classes, functions, return data, parameters, and member variables.

Maximum Line Length

This is normally the first rule to be relaxed by companies. Line lengths should be limited to 79 characters. Having this limit means you can open multiple files side by side, as you often need to work in more than one file at once, or when you need to access your code on a server and you only have a command-line shell with a limited character length. It also has a limiting effect on nested statements, since they are a sure way to break the 79 characters rule.

Indentation

Use spaces for indentation, four spaces per indent to be exact. Visual Studio Code will automatically insert four spaces when you push Tab. In some other editors, you need to configure this first. You can use normal tabs, but when you use tabs, you cannot mix them with spaces for indentation. Python 3 won't allow this. So it is either spaces or tabs, but not both.

Indenting Line Wraps

Line wrapping happens when a line of code is longer than a prescribed amount of characters and has to continue on the next line. You perform a line wrap for readability purposes, and you wrap a line when it exceeds 79 characters as per PEP standards (or whatever length you prefer). You should wrap strings, comments, function declarations; in fact, nothing should exceed the line limit.

The following is an example of a hanging parameter, where the first parameter “hangs” under the function declaration. Wrapped parameters are indented with two spaces. This separates the parameters from the function body as they are nested deeper and do not line up with the code.

```
def this_is_a_function(
    variable1, variable2,
    variable3):
    function_body_code
    function_body_code
```

If you decide to start your parameter list on the same line as your function declaration, then align the variables with the top line of variables when you wrap to a new line, like so:

```
func = this_is_a_function(variable1, variable2,
    variable3, variable4)
```

When you are writing an *if* statement that is so long it has to wrap to a new line, you should add additional indentation to separate the rest of the *if* statement from the *if* statement's body.

```
#This is correct
if a == True and b == True
    and c == True # This line should not be aligned with
    the code block beneath it
    do_this
```

The following is not correct. The `and c == True` statement lines up with the body of the function, which makes it harder to read:

```
if a == True and b == True
    and c == True:
    do_this()
    do_that()
```

Arrays can be line-wrapped as well. Both of the following examples are correct. One level of indentation, and the closing bracket can be either the first character on a new line, or it can be indented one level.

<pre>fruit_list = ['apples', 'pears', 'strawberries']</pre>	<pre>fruit_list = ['apples', 'pears', 'strawberries']</pre>
---------------------------------------------------------------------------	---------------------------------------------------------------------------

Lines with multiple logical operators can be wrapped as per the following example, between parentheses, with logical operators starting on a new line, and lining them up with the first parameter:

```
total = (one
+ two
+ three
)
```

Line wrapping with the correct indentation can definitely aid in readability and is something that should be practiced while you code.

Blank Lines

Add one blank line before and after a function. You can separate logical divisions in your code with blank lines as well.

Encoding

Your source code file should be encoded in UTF-8. In Visual Studio Code, go to File ► Preferences ► Settings. In Settings (you can search for it in the provided toolbar) you will find **Files:encoding**. Note that Visual Studio Code is set to UTF-8 by default.

Imports

Imports are placed on top of the file, preceded by docstrings and comments about the script itself. They should be grouped by the origin of the code.

- **Standard library imports:** These are imports of native Python libraries.
- **Related third-party imports:** These are imports you have installed from third parties.
- **Local application/library-specific imports:** In general, these are imports you created yourself.

Whitespace

You may feel that extra whitespace makes your code more readable, but it is best avoided. Whitespace directly after square, round, or curly braces should be avoided. Whitespace after a comma is allowed.

The following two examples are correct:

```
my_data = [1, 2, 3, 4]
my_function(param1, param2)
```

Whitespace should surround assignment, binary, and logical operators, as follows:

```
amount = 10 - 4 * 4
```

Some tricky exceptions exist. For instance, when the assignment operator, which should be surrounded by whitespace, is used as a named parameter in a function or to assign a default value to a parameter, it should not be surrounded by spaces.

This function declaration is correct:

```
def email(email_address, from_address='server@server.com'):
```

The following code snippet is correct (calling the function with named parameters):

```
email(from_address='do-not-reply@server.com',  
email_address='client@mail.com')
```

Naming Conventions

There are numerous case styles for software developers to use when naming things.

- PascalCase
- camelCase
- kebab-case
- snake_case

Python uses PascalCase and snake_case. In PascalCase, the different words in the name are capitalized. In snake_case, the different words in the name are separated by an underscore. snake_case words are always in lowercase, apart from one exception: constants. This will be addressed a bit later in this chapter.

When naming your class, module, function, or variable, remember that readability matters. `banking_details` is a better variable name than `details`. I have literally seen a variable called `the_data` in code, which means nothing to the developer who comes afterwards to make changes. It is also recommended to not use single-character variables using `l`, `i`, or `o`. Certain character sets can make `i` look like `l` and an `o` like a `0`. In my opinion, using a single character for a variable name is bad practice anyway. Giving proper names to variables, functions, and classes makes code more self-commenting, which means that it can be understood to a certain extent even without docstrings.

Package and Module Names in snake-case

Package and module names should preferably be single-word names, but when they consist of two words, they should be snake_cased and all lowercase.

Class Names in PascalCase

When you encounter an abbreviation, you should capitalize all the letters in that abbreviation. For instance, SSHKey is correct. SshKey is not.

Method Names in snake-case

Method names use snake_case with all the letters in the name as lowercase. You should not surround method names with two leading and two trailing underscores because this is reserved for Python's magic methods.

Internal Methods

Methods intended to be used internally only get a single leading underscore. The single underscore signals to other developers to not depend on that method. An internal method is meant to be used internally to the class it was declared in, and can be changed or removed at any time, making it very unreliable to be used externally.

Preventing Inheritance

If you have a method that you do not want to be overridden by a subclass, then prepend it with two underscores. This causes Python to mangle the name³ of the variable and prepend the class name to it. Listing 5-6 demonstrates name mangling.

³https://en.wikipedia.org/wiki/Name_mangling#Python

Listing 5-6. Name Mangling

```
class Security:
    pin_number = 12345

    def __print_pin(self):
        return self.pin_number

secure = Security()
print(secure._Security__print_pin())
```

Variable Names in snake-case

Variable names use `snake_case` with all letters in lowercase. You should not surround variable names with two leading and two trailing underscores because this is reserved for Python's magic methods. Magic methods are special methods in a programming language that do not get invoked by the programmer, but rather by the programming language when it reaches a specific condition. A good example of this in Python is the constructor `__init__()`. We never call it directly, but Python calls it whenever we create a new object.

Internal Variables

Variables intended to be used internally only get a single leading underscore. The single underscore signals to other developers to not depend on that variable. A variable with an underscore is meant to be used internally to the class it was declared in, and can be changed or removed at any time, making it very unreliable to be used externally.

Preventing Variable Overwriting in Inheritance

If you have a variable that you do not want to be overwritten by a subclass, then prepend it with two trailing slashes. This causes Python to mangle the name of the variable and prepend the class name to it. Listing 5-7 demonstrates name mangling.

Listing 5-7. Name Mangling

```
class Security:
    __pin_number = 12345

secure = Security()
print(secure._Security__pin_number)
```

Constants in snake-case

Constants are capitalized using snake_case to join different words. This is an example of a constant:

```
TAX_RATE = 17
```

Chapter Summary

Readability is king. Whenever you write code, assume another developer will eventually have to expand on whatever you are building now. In order to do that, your code needs to be well designed and readable. To achieve readability, you need to apply good practices consistently throughout your code. Start with object calisthenics, refactoring, and coding standards straight away in your work and tests.

When developers need to look at code they wrote a year ago, they won't always immediately pick up where they left. They may have written 100,000 new lines of code in that year and built aspects of the system far removed from the work they did one year ago. When they look at that code

they wrote one year ago, they sometimes have to start from scratch in order to understand it again. Sure they will pick it up very quickly, but if you write it as neat and concisely as possible, it will alleviate the cognitive load and relearning. So go ahead and write neat and readable code that is well commented. Do not just do it for your fellow software developers, do it for yourself as well. As Damain Conway said, “Documentation is a love letter that you write to your future self.”

References

The ThoughtWorks Anthology, Pragmatic Press (March 2008), ISBN: 9781934356142.

Documentation on Google style docstrings: <https://google.github.io/styleguide/pyguide.html>.

CHAPTER 6

Databases and Database Design

This chapter does not just deal with creating databases, tables, and querying them. It also deals with setting your database up in your Docker environment, which is a technical aspect of database knowledge that you should have.

Data is the blood of your application. It flows through your software and gives it purpose and meaning. Data shapes the state your software is in, helps you log in and log out of systems, and delivers data needed by humans to base business decisions on. When you do a credit card transaction, or just “like” someone’s photo on a social media website, it all entails reading and writing to databases. There is a lot of talk about big data and new competing technologies that promises faster reading and writing times to and from databases. It is also very popular to talk about big data and visualizations of your data, and in general just really cool technologies that can help you leverage your data to help a business grow. But, before you even think of the latest technologies like big data, data analytics, and extrapolating behavior from data, you first have to look at data in the form you encounter every day and learn how to use that. Database knowledge and exposure to have designed databases are essential. Database design can be tricky. One wrong move and you are refactoring a database six months down the line. That can be a messy business. Refactoring a database means moving data around in the database and refactoring the backend code that was written to use it. Or even worse (but sometimes the easiest), keep it like it is and work around

it. In this chapter, I aim to help you get a grip on some of the aspects of database design and SQL queries. It is a huge field, but this chapter will leave you competent to set up a database and tables in a normalized form and run queries on them. Later in this book, you will build on this knowledge to let Python read and write from a database by building an application.

Three Things You Can Do with Data

There are three things you can do with data in a storage sense:

- You can discard it immediately after use.
 - This is data that does not need storage. For instance, when you type in your password, that password enters your system as plain text. It should never be saved anywhere as plain text, and is a good example of data that should be discarded as soon as it has been used.
- You can store it short term.
 - Ecommerce shopping baskets in some systems do not live more than a few weeks. However, it seems this trend is changing as I am noticing more and more e-commerce systems remembering shopping baskets.
 - This can be data that has been cached. Cached data is data that is stored short term and is faster to look up than long-term data.
- You can store it long term. This is our main concern in this chapter. This is data like usernames, addresses, purchase histories, and basically anything that would be of value in the future.

It is always a good idea to only store data you will need. Even though storage space is getting incredibly cheap, it is still data you need to look after. If you are not going to use it, lose it.

Overview of Database System Components

Your database management system, or DBMS, is the application that houses and manages your database users, their permissions, logs, and of course, databases. It also houses metadata about databases and handles aspects like replication and sharding, two topics we won't be covering in this book. Your database is maintained by a DBMS, and in fact, you will have multiple databases. Choosing a DBMS is a relatively simple task. With so many free, high quality DBMSs on the market, you literally have to have special needs in your system in order to not be able to select a DBMS quickly.

We will use a relational database. They are very popular and very powerful. With relational databases you get your data in a tabular, row-by-row format with a powerful query language called SQL to manipulate the data. One DBMS can house multiple databases. A single database can house many tables. Tables are the structures where your data is stored, and they consist of tabular lists of data, which can be linked to other tabular lists of data via primary and foreign keys, which I will explain later. A table consists of columns, where each column has a data type, such as integer, text, or boolean. The data in each row in a table spans these columns and that is what makes a record. Tables also have indexes. These are tables, basically hidden from the user, that store data in an ordered fashion in order to speed up data lookups. You will also look at them in a bit.

Figure 6-1 illustrate the concept of the database management system. It has two databases it manages in this instance, and each database has one table. Database 1 contains a table called `users`, and database 2 contains a table called `shopping_carts`. Both have three columns.

The users table has name, surname, and age as columns. The shopping_carts table has price, item, and quantity as columns. On the right you can see a representation of each table filled with two rows of data. In the representation, the first rows contain the column names.

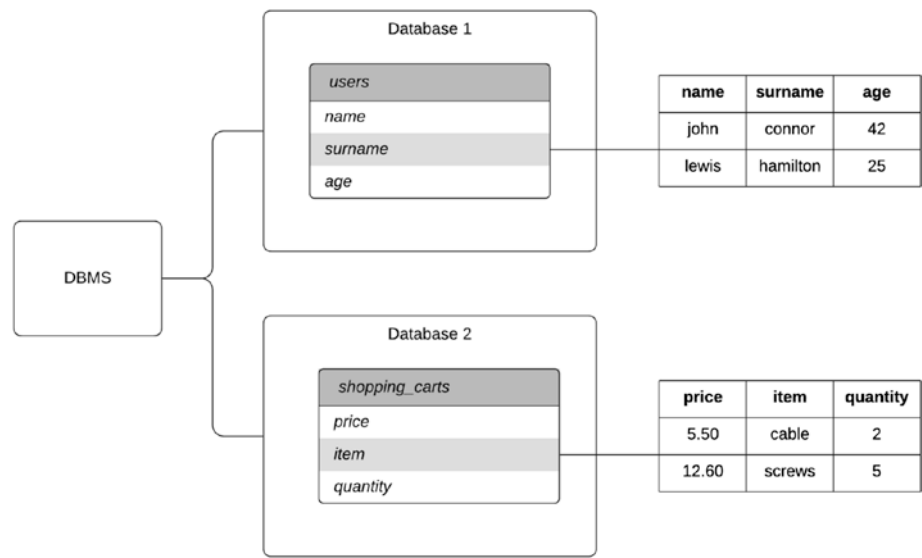


Figure 6-1. Illustration of an DBMS and it’s databases

Setting Up Your DBMS

First, you need to select and set up your DBMS. The DBMS I selected is called MySQL and it is a great one to use. It is always worth comparing databases and seeing what features the different databases have, but for our needs, MySQL will do just fine. To add the database to your setup, you will edit your `docker-compose.yml` file. Under the services section, and under the entry where you defined your Python interpreter, add the configuration code in Listing 6-1.

Listing 6-1. MySQL setup for the Docker-compose file

```
mysql-dev:
  image: mysql:8.0
  container_name: mysql-dev-container
  ports:
    - 6603:3306
  environment:
    MYSQL_ROOT_PASSWORD: "root"
  volumes:
    - /storage/docker/mysql-datadirectory:/var/lib/mysql
```

Most of this will look familiar to you because you dealt with it in Chapter 2. I will just go over the ports, environment, and volumes entries quickly.

Ports

These are the ports that will be opened on the container for access. They are separated with a colon, so A:B. Port A points towards accessing the database from outside your Docker images. For instance, you will use that port number to access the database. Port B is the port that will be used between Docker containers.

Environment

This is the database's main password, called the root password.

Volumes

You can mount local storage with storage inside your Docker container. This is needed especially if you develop locally. This entry is also separated with a colon. To the left is the local storage, which is your computer, and to the right of the colon are the storage files (in this case, your database files on the Docker instance).

The Final Docker File

So what you have now is a MySQL instance, password protected with a rather feeble password called root, accessible from within the docker-compose environment with port 3306 and from outside the dockerized environment with port 6603. I choose this last port at random to clearly demonstrate the different ports. The data is saved in the Docker environment in `/var/lib/mysql` and is saved locally as `/storage/docker/mysql-datadir`. What is the reason for this volume mapping? This allows you to persist your database's data even after you switch your Docker container off. Your complete Docker file should now look like the configuration in Listing 6-2.

Listing 6-2. The Complete Docker-compose File

```
version: "3"

services:
  python-dev:
    container_name: python-dev-container
    build:
      context: .
      dockerfile: ./Dockerfile
    volumes:
      - ./test:/home
```

```
mysql-dev:
  image: mysql:8.0
  container_name: mysql-dev-container
  ports:
    - 6603:3306
  environment:
    MYSQL_ROOT_PASSWORD: "root"
  volumes:
    - /storage/docker/mysql-datadirectory:/var/lib/mysql
```

Now all you need to do is run the following command:

```
docker-compose build --no-cache
```

After this has built, you should run the following command. The `-d` flag separates your running Docker instances from the terminal you are in, allowing you to type more commands.

```
docker-compose up -d
```

When you run `docker-compose ps`, you will see your DBMS instance running. Whenever you want to run your code, you can run it like this:

```
docker-compose run python-dev
```

Now that you have a database Docker instance, let's add the Docker instance that will allow you to view and edit your database.

Viewing Your Database Using Adminer

There are numerous ways to access your database. One of the most versatile ways is by using the command-line, but the method we will use is a web-based application called Adminer. I believe it will be beneficial for you to see the data tables in a visual format. You will add a Docker instance hosting Adminer to your `docker-compose` file.

Add the configuration shown in Listing 6-3 to your docker-compose file at the bottom, nested under services.

Listing 6-3. Adding Adminer to the Docker-compose file

```
adminer:
  image: adminer
  container_name: adminer-tool
  restart: always
  ports:
    - 8080:8080
  links:
    - mysql-dev
```

After you have added this code, your docker-compose file should look like this:

```
version: "3"

services:
  python-dev:
    container_name: python-dev-container
    build:
      context: .
      dockerfile: ./Dockerfile
    volumes:
      - ./test:/home

  mysql-dev:
    image: mysql:8.0
    container_name: mysql-dev-container
    ports:
      - 6603:3306
    environment:
```

```

    MYSQL_ROOT_PASSWORD: "root"
volumes:
  - /storage/docker/mysql-datadirectory:/var/lib/mysql

adminer:
  image: adminer
  container_name: adminer-tool
  restart: always
  ports:
    - 8080:8080
  links:
    - mysql-dev

```

Rebuild your environment.

```
docker-compose build --no-cache
```

And then, start the containers.

```
docker-compose up -d
```

If you run the following command in the command line, you should see Adminer running:

```
docker-compose ps
```

Once Adminer is running, as it should be now, open a browser and type the following in the address bar. This is the location and the external port, as defined by the port number on the left of the colon in the docker-compose file (8080:8080) where you can access the Adminer application from.

```
127.0.0.1:8080
```

On the web page that appears, you will see a few input fields: System, Server, Username, and Password.

1. Leave System as is. By default it should be on MySQL.
2. In the Server field, type the name of your MySQL Docker container. It should be `mysql-dev-container`. This can be quite confusing and it is important that you understand the following. No application on your computer, apart from Docker, will be able to connect to a Docker container based on the name. So it is important to remember that Adminer is running in a Docker container, and due to this has access to the location of `mysql-dev-container`. You won't be able to connect to `mysql-dev-container`, or any other container, if you are not doing it from within a Docker container or the Docker-based command line.
3. In the Username field, type `root`.
4. In the Password field, type `root`.

Click the Login button. You should now be logged in and be able to see the Adminer dashboard. You should now, for instance, be able to see a link called Create database.

Cleaning Up and Pushing to the Remote

Before you go ahead, let's make sure you do not lose your development environment due to an unexpected hard drive crash, for instance.

From the command line, rebuild your environment, remembering to change the version number to 3.0.0. If you have played around with Docker in the meantime and have a different version number, just make it a version higher than your last version number.

```
docker build -t your-docker-user-name/python-docker-tutorial:v3.0.0 .
```


Now, log into Docker from the command line.

```
docker login
```

After successful login, push the image to the Docker repository.

```
docker push your-docker-user-name/python-docker-tutorial:v3.0.0
```

Preparing Your Database

Before you start using your database, you should always do a few things to prepare the database. You will, for now, have one root user for your database. This is the user with all the possible permissions who can perform all actions in your DBMS. Do not use your root user for database authentication once you start coding. When you get to the Python code that deals with database access, you will create an additional user. The reason you create secondary users for your codebases to connect to the database is that the username and password will be saved in your codebase, somewhere. Should that username and password get leaked, you can easily change it or block it, and since the user will be created with limited permissions, there is a very limited amount of damage it can do. Having said that, a user will almost always have read permissions, and stealing sensitive data can be incredibly damaging. No matter how limited the secondary username's access, the password should remain secret. The username should have the following attributes:

- A username should preferably be linked to a single database. The last thing you want is someone stealing one username and being able to access all your databases.

- It should have limited permissions. For instance, it should only be able to read data, write data, and create tables. In general, the permissions should be pinned down to exactly what your system can do. If you never delete data, then do not give that user delete permissions. If you never delete a table from your database, do not give the user the permission to delete tables. Never ever give a secondary username the permissions to change permissions!
- The username should be linked to an IP address. An IP address identifies the location of a computer on the Internet. You will, for instance, allow the username to connect to the database server from one very specific location, or from a region, or a country. This is, however, not fool-proof.

Primary Keys

Every table should have a primary key. A primary key is a unique identifier that appears only once in a row. DBMSs normally supply us with an auto-increment key to generate an integer-based primary key automatically. This is very handy, but if used incorrectly can have security concerns. For instance, guessable primary keys may be misused if you do not have proper security in place, and someone can directly target rows in your database to read. Primary keys are indexed and are faster to search on than non-indexed fields.

Indexes

You can create an index on any field you want. An index field does not have to be unique, although you can create an index to be unique. Indexing a field creates a secondary table in your system, with the column ordered

numerically or alphabetically, and a link to the original row in the table containing the data. Having the data ordered means the system can find it a lot quicker.

Index Caveats

Indexes will grow as your table grows, and you should not index each and every field you have. Focus on the ones you will search on. You should also consider updates. When a field that has been indexed gets updated, the DBMS needs to update the index as well. Imagine you have a table with vehicle data. The vehicles have tracking information, and every 10 seconds the vehicle's `last_location` gets updated. If you have 1,000 cars each updating their last location every 10 seconds, your reindexing process will be very busy. Now imagine your fleet grows to 10,000 cars. This will cause definite issues with your system.

Data Types

Fields inside your tables need dedicated types. The types are quite extensive and I will quickly go over some of the main ones here.

<code>text</code>	Accepts text data. Comes in a few variants, like <code>tinytext</code> , <code>mediumtext</code> , etc.
<code>varchar</code>	String data, where you specify the maximum length of the string yourself. Uses only as much space as it needs when storing a string.
<code>int</code>	Integer data. Also comes in a few variants, like <code>tinyint</code> , <code>medium int</code> , etc.
<code>decimal</code>	Decimal point-based data. Comes in a few variants.
<code>datetime</code>	Saves date and time data. Comes in <code>datetime</code> , <code>date</code> , and <code>time</code> .
<code>boolean</code>	Comes as <code>tinyint</code> .

You need to make sure you select the exact data type you need. This requires in-depth knowledge of data types, which takes some research and time to learn. Different DBMSs may also have different data types. Always use the best possible value for your field. If you know that you will only be storing integers lower than 100, then `tinyint` is sufficient. If you will only store first names in a field, then `varchar(60)`, where 60 indicates the maximum number of characters, should suffice. Just creating name and surname fields as text is bad practice. When you design your database, and once you know what values you expect, make sure that you look into that DBMS's data types in order to select the best value possible. Making wrong decisions about datatypes won't really break your system but can have adverse effects in the long run.

Creating a Database

You will create a database first. This will help you assign the user you will create later to it. Creating a database is very easy. Log into your Adminer instance on 127.0.0.1:8080. Once logged in, you will see a link called Create database. When you click this link, it will go to a webpage with two input fields. One field is for the database name, and the second is for the collation. *Collation* refers to character sets that the database will use. We are not going to dabble in character sets here, as it is a rather large topic, so you can accept the default character set. In the first input field, type the database's name. Make the name `LanguagesClass`, and click the Save button. This should now have created your database for you. It still has no tables, but you will get to that in the next section.

Inside the `LanguagesClass` database, you will add one table for now. Table 6-1 is purposely designed rather badly. Here is an example of the data you can find in it. To the novice, this table may not look too bad. Apart from columns left out for brevity, this is already telling you a lot of what you want to know.

Table 6-1. *First datatable*

Salutation	firstname	Mobile	City	languages
Mr	Lewis	0798456325	London	German, English
Mrs	Jane	0796547896	Bristol	English
Miss	Cindy	0741598745	Hove	German, Dutch, English
Mr	Roger	0749856547	London	German

Let's call this table *classes* because it contains the classes that people belong to. As you will see later in this chapter, this table is not well designed. This table has been kept small so that it is easier to understand, but a design like this can cause your table to become massive. I have seen tables with well over 100+ columns in them.

You can see the columns on the top row, five of them. An entry spans over the columns, and that is what constitutes a data entry. Even though the design I am showing here is not optimal, you are going to create this table and query it anyway. This will help you understand how to refactor a database in a process called *normalization*.

To create this table, log into your Adminer instance, and click the database you created earlier called *LanguagesClass*. Look for the link called *Create table* and click it. You will see some input fields. By default Adminer gives you around three, but you can add more.

- To add another field, click the + sign to the right of the input fields.
- To reorder your fields, click the up and down arrow. Ordering can aid readability. Always have your primary key field first.
- Column name: This is the name of the column, such as `id` or `pet_name`.

- **Type:** This is the type the data is stored in, such as integer or text.
- **Length:** Certain types, like varchar, can take a length field. Enter that here.
- **Options:** This is outside the scope of this book.
- **NULL:** Whether the field can be NULL or empty if now data is received.
- **AI:** Auto increment. Normally you have one auto_increment field, the primary key. This field should be first, for readability.

Creating the Table

You do not have a primary key in this example. Primary keys are needed, and you will add it. You are creating a sub-optimal database table first, and as you fix it up, you will add the primary key.

salutation

Select varchar, and make length 5.

firstname

Select varchar, and make length 30.

mobile

Select varchar, and make length 12.

city

Select varchar, and make length 30.

languages

Select varchar, and make length 100.

Now click the Save button.

Filling the Database with Data

Filling your table using the Adminer interface is quite easy. Make sure you are viewing your table. It should say “Table: classes” on top in bold letters. Below that you should see a link called New item. Click that link. You should now see all your columns as input fields, and you can enter data into these input fields. You do not need to add a value to the auto_increment id field, as the DBMS will provide it. You can go ahead and fill in some data, using your own or using the table above. Enter about three rows of data. You will see the data displayed in the table view.

Your First SQL Queries

SQL stands for Structured Query Language. It is a language with very strict rules that allows you to fetch data from your database. But, not only can you fetch data using SQL, you can do almost anything you can think of to the database. I will not go through all of the functionality in this chapter, or even this whole book, but just enough to make you functional in SQL. In this chapter, you will be looking at the following functionality, identified by these keywords. This is not an exhaustive list of SQL keywords. By convention, SQL keywords are written in uppercase.

- **SHOW:** To show available tables. Handy when you are on the command line and not in Adminer.
- **DESCRIBE:** To describe a table’s columns. Handy when you are on the command line and not in Adminer.
- **SELECT:** To select data.

- **INSERT:** To insert data.
- **UPDATE:** To change data.
- **DELETE:** To delete data.
- **JOIN:** To join multiple tables.
- **CREATE:** To create tables.

Additional keywords

- **FROM:** Introduces the start section where you specify where the data the query needs comes from.
- **WHERE:** Sets search conditions on the SQL query

To start writing a query in Adminer, look to the left of the Adminer dashboard, where you will see the words SQL Command. Click on it. In the screen on the right-hand side a text field will appear. Type the following command in it and click the Execute button below the text input field. When you write SQL, there is a convention that you type all the keywords in uppercase letters. Whether you do it is up to you. It has no effect on the SQL itself apart from letting the keywords stand out. I will use this convention in this chapter.

SHOW TABLES

The result should show you the available tables that you can query. You should have one result.

Now type the DESCRIBE command. It takes the form of DESCRIBE TABLENAME:

DESCRIBE CLIENTS

This should show you the columns in the table and their types. It's very handy if you do not have a visual interface like Adminer.

SELECT

Let's select data. You should have at least three records in your table. Let's select them all at first. Type the following commands separately into the text box and run them separately as well:

```
SELECT * FROM classes
```

```
SELECT firstname, mobile FROM classes
```

```
SELECT * FROM classes WHERE firstname = 'Rogers' AND city =  
'London'
```

In SQL, you can also do logical operations on sets of data using `>`, `<`, `!=` and the `IN`, `NOT IN`, `IS`, and `IS NOT` keywords. For example, the following query will return results where `city` is equal to any of the values in the list:

```
SELECT * FROM classes WHERE city IN ('London', 'Cape Town',  
'Beijing')
```

This can be reversed using `NOT IN`:

```
SELECT * FROM classes WHERE city NOT IN ('London', 'Cape Town',  
'Beijing')
```

SQL fields can be declared nullable, meaning that it is okay to omit a value from that field. You can run a SQL operation on your data using null fields in your logical checks as well. The following query will return all records where `name` is not null. **Note that null is the absence of data, and not merely an empty string.**

```
SELECT * FROM classes WHERE name IS NOT null
```

You have encountered the other operators already, and they do bigger than, smaller than, and not equal to logical comparisons on data.

The important aspect to notice here is the `*` in the first command, and the `id`, `owner`, `location` in the second. The `*` means *every column you have*, whereas in the second command, `SELECT` is the action, meaning to

fetch data, and directly after the FROM is where you specify the tables the data should be retrieved from. The WHERE clause specifies data influencing which data will be selected. In the third query, you specify that you only want records with a name equal to Rogers and a city equal to London. The WHERE clause can use any column on your table to do a comparison on to return data, and the comparisons can be combined using AND and OR. I advise you to play around a bit and run queries on the different columns in your table. The logical operators you can use are =, <, >, !=, IN, NOT IN, IS, and IS NOT.

INSERT

SQL offers a way to input data into your database. The syntax for this has less variance because an insert cannot be down with conditionals and does not have a WHERE clause. You can do single inserts with it or multiple inserts.

```
INSERT INTO classes (salutation, firstname, mobile, city,
languages) VALUES ('Mrs', 'Richards', '0754896325', 'London',
'German, Dutch')
```

Multiple inserts are structured the same way as a single insert, where the inserts are separated by commas. There is a maximum of a thousand inserts you can add to a multiple insert statement. Multiple inserts like this are a lot faster than multiple single inserts:

```
INSERT INTO clients (salutation, firstname, mobile, city,
languages) VALUES ('Mr', 'Eagle', '07546458745', 'Bristol',
'Dutch'), ('Miss', 'Schofield', '0766984752', 'London', 'English')
```

Anatomy of the insert statement:

```
INSERT INTO tablename (list of columns to be updates) VALUES
(corresponding list of values), (if you want to insert multiple
entries, but them between braces and separate the datasets with
commas)
```

UPDATE

Once you have data in the database, you may need to change entries now and then. SQL provides the `UPDATE` clause. You need to be careful, however. An update statement in its most simple form will update each and every value in each and every column you specified in your table. The following is a simple update query but be very careful before you run it.

```
UPDATE classes SET salutation = 'Mr'
```

The query above will update all the salutations entries in the `clients` table to Mr. This is extremely undesirable and can be catastrophic. In almost all cases, it is very important to specify which records you want to change, and you do that using the `WHERE` clause and logical operators. Note that even though the following query is an example, it will still change ALL salutations where name is equal to Richards. You must use the logical operators to narrow your update down.

```
UPDATE classes SET salutation = 'Mr' WHERE firstname =
'Richards'
```

DELETE

Every now and then you need to delete data from your database. You should consider carefully what is deletable and what is not. A great alternative to removing columns from your table is called a soft delete. With a soft delete, you have an additional column in your tables, called for instance, `deleted_at`. When you delete using a soft delete, you don't remove that row of data. You merely set the `deleted_at` field to the current date and time, and then handle that row as deleted in your code, as follows:

```
SELECT * FROM classes WHERE deleted_at IS NOT null
```

There are times where you want to remove data completely from your system. For instance, clients have the right to ask that you remove all of their data from your system, in which case you have to be able to remove those lines from your database completely. Customers are protected by law and can legally force you to remove all of their data.

A typical delete query will look like this:

```
DELETE FROM classes WHERE firstname = 'Rogers'
```

As with UPDATE, running DELETE without a WHERE clause will remove all data from your table.

Normalizing the Current Classes Table¹

There are some serious problems with the design above, and I will show you how to get rid of them by normalizing your table. There is a lot of repeated data in this table, which will cause the following issues:

- It takes up a lot more space on your system.
- It makes updates very difficult. Let's assume that in your example table, you misspelled London. Because of the repeated data, you now need to update multiple rows. This may not look like a big problem, but consider a system with many millions of rows, and a misspelled word like London in multiple tables, and you need to change multiple incorrect entries. This can become a nightmare.

¹www.guru99.com/database-normalization.html

- The languages column consists of a string of multiple treatment elements, separated by commas. This makes it a lot trickier to update a single language element belonging to the multiple language string you have at the moment. If you have as an entry 'German, English' and now need to remove English from the language, you cannot just do it in a straightforward manner. You need to reconstruct the whole language string before you can do an update. This can become trickier if your language consists of 20 elements, and you need to retrieve the language data, remove one element from it, and add another before you can update the row.

I will show you how to normalize this table using the steps designed to decompose a single table into multiple small tables. The steps to normalize your database is called normalization, and the first three steps you will look at are called normal forms. After this, I will show you how to query the data using SQL join clauses. You will look at the first normal form, second normal form, and third normal form.

First Normal Form

- Each entry should be unique, and contain a single value per cell.

Take Table 6-2 as the current populated table. This is a bad design and not in any normal form.

Table 6-2. *Unnormalised datatable*

salutation	name	Mobile	city	languages
Mr	Lewis	0798456325	London	German, English
Mrs	Jane	0796547896	Bristol	English
Miss	Cindy	0741598745	Hove	German, Dutch, English
Mr	Roger	0749856547	London	German

Table 6-3 is Table 6-2 in the first normal form. Every cell contains one value. This is better but even in this small example there is a lot of data repetition.

Table 6-3. *First normal form*

salutation	name	Mobile	city	languages
Mr	Lewis	0798456325	London	German
Mr	Lewis	0798456325	London	English
Mrs	Jane	0796547896	Bristol	English
Miss	Cindy	0741598745	Hove	German
Miss	Cindy	0741598745	Hove	Dutch
Miss	Cindy	0741598745	Hove	English
Mr	Roger	0749856547	London	German

Second Normal Form

To qualify for the second normal form, a table should be in first normal form and every non-key fully dependent on a primary key. Hence it also needs a single primary key. Salutation, name, and mobile are functionally dependent on each other and can move to their own table, called *clients*,

as seen in Table 6-4. Functional dependency can be a tough one to crack. It states that all non-key columns should be dependent on the primary key, which uniquely identifies each row. Looking at `clients`, you can say that `salutation`, `name` and `mobile` are dependent on the `id`, because they all have the same purpose, to identify a client. The `id` identifies the client and so those fields are dependent on the key. But you cannot identify a client by looking at the language courses taken or in which city they are, so those two fields have nothing to do with identifying a client.

Table 6-4. *The clients Table*

id	salutation	name	mobile
1	Mr	Lewis	0798456325
2	Mrs	Jane	0796547896
3	Miss	Cindy	0741598745
4	Mr	Roger	0749856547

Languages and cities are not functionally dependent on each other, or on `clients`, and can be in their own tables. Create the two tables shown in Table 6-5 and Table 6-6.

Table 6-5. *The languages Table*

id	language
1	German
2	English
3	Dutch

Table 6-6. *The cities Table*

Id	city
1	Bristol
2	London
3	Hove

You are not done yet. An interesting and very important part still lays ahead. You have your data almost in second normal form, but it has lost its meaning. You do not know what cities your clients are in, nor do you know what language lessons they are signed up for. In the next step, I will show you how to link the data with each other using relationships and foreign keys.

One-to-Many Relationships, Joins, and Foreign Keys

A client can live in only one city, and one city can have many clients, so there is a one-to-many relationship between clients and cities. This is easy to portray in a database. For this relationship, you add another column to the clients table. Let's call it `city_id`. The clients table will now look like Table 6-7. You removed the duplicate primary value, which is the name of the city, and replaced it with the primary key of that table. The `city_id` field is what is now called a *foreign key*. That means it is a key that uniquely identifies a column, or columns, in another table.

Table 6-7. *Refactored clients table*

Id	Salutation	Name	mobile	city_id
1	Mr	Lewis	0798456325	2
2	Mrs	Jane	0796547896	1
3	Miss	Cindy	0741598745	3
4	Mr	Roger	0749856547	2

So, how do you query this? You use a SQL join. The important first step is to create these three tables in the database yourself. Playing around, failing, and succeeding will always be part of the learning process, so I will not take you through the steps again, but here are some pointers:

- Each table must have an Id column, and it must be auto-increment (AI).
- For the tables you need for this example, you can use varchars except for `city_id`, which must be integers. Make well-educated guesses as to the lengths of the varchars.
- Build the tables using Adminer.
- Make sure that you have realistic data in the foreign key columns. In this instance, you only have one foreign key column. It lives in `clients` and is called `city_id`. Make sure that a `cities` primary key is in fact present in this column.

You should now have four tables. One is called `classes`, which is the old table you created initially. The three new tables are called `clients`, `cities`, and `languages`. Make sure they are populated with data, and bear in mind you have not yet created the relationship between `cities` and `languages`.

Querying on a Join

Now to create the SQL that will return the results. For this you will use the `JOIN` clause. You use the `JOIN` clause between two tables in your query, specifying the columns it should join in. Here is an example query:

```
SELECT * FROM clients cl JOIN cities cit ON cl.city_id = cit.id
```

Here I introduce two new concepts. First, you will notice that right after `clients` you have `cl`, and right after `cities` you have `cit`. These are called aliases. The purpose of an alias is very simple:

- If you have a very long table name, this shortens it and makes it more convenient to type.
- If you need to join a table on itself, you can refer to the table by its two aliases.

You can also see the `JOIN` keyword. This tells MySQL that rows in tables `clients` and `cities` should be joined on columns `city_id` and `id`. Joining in this way tells MySQL that data in the relevant columns are related to each other.

If you want all clients from London, you can query as follows:

```
SELECT * FROM clients cl JOIN cities cit ON cl.city_id = cit.id
WHERE cit.city = 'London'
```

Many-to-Many

The many-to-many relationship is a more interesting relationship to model. For this, you need to understand the following:

- You need an intermediate table, also called a pivot table.
- This table is the relationship table between `client` and `languages`. Remember that one client can practice many languages and one language can be practiced by many clients.

- Because you cannot model many languages per client in the `customer` table without breaking normal form 1, and for the same reason you cannot model many clients per language in the `languages` table, you now need to insert them into a table with a structure like the one in Table 6-8. Let's call this table `client_languages`. Note that `client_id` in column one is the primary key of a client in the `clients` table and `language_id` is the primary key of a language entry belonging to the client. With this setup, one client can have many languages and one language can have many clients.

Table 6-8. *A many to many relationship*

<code>client_id</code>	<code>language_id</code>
1	1
1	2
2	2
4	1
3	1
3	2
3	3

In human-readable terms, you are saying the following. The clients whose ids are in the `client_id` column of the `client_languages` table are linked to the language whose id is in the `language_id` column of the `languages` tables. With this setup, you can easily link many languages to many clients and many clients to many languages. To further this demonstration, create the table `client_languages` in your database with

two columns of type integer and name them `client_id` and `language_id`. Populate them with the appropriate client language data, making sure that some clients have more than one language.

Look at Figure 6-2 for a graphical representation of how languages and clients are connected.

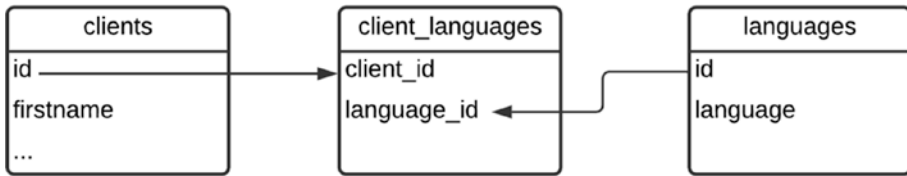


Figure 6-2. *Many to many relationship between languages and clients*

Then, run the following query:

```
SELECT * FROM clients c JOIN client_languages cl ON c.id=cl.
client_id
JOIN languages l ON l.id = cl.language_id
```

Here you are joining on all the keys from `clients` through `client_languages` to `languages`.

To find out which clients have, for instance, German as a language, you can add a `WHERE` filter clause.

```
SELECT *
FROM clients c JOIN client_languages cl ON c.id=cl.client_id
JOIN languages l ON l.id = cl.language_id
WHERE l.language = 'German'
```

Third Normal Form

There's still have another normal form left to look at. Technically, there are even more normalization techniques, but we are ending at the third normal form, as the third normal form is not just sufficient, it is also

practical. According to the third normal form, you should have no non-primary field dependent on another non-primary field. Now this may take some thinking and designing to get this right. But you have one very good candidate field, salutation. Note that salutation is dependent on name. This is a technicality, but not a bad example. Mr, Miss, and Mrs are gender-dependent salutations, which in this limited example can be derived from name. What I mean is that should a name entry change, for whatever reason, from Jane to John, then the functionally dependent field, salutation, also needs to change. A table should not have a field dependent on another field inside the same table. How you get around it is to pull the salutations out of the clients table into its own table, and create a one-to-many relationship between the new table, which you'll call salutations, and clients. The salutations table will have a primary key called id and a field called salutation. You will remove the salutation column from the clients table and replace it with salutation_id.

The new salutation_id table is shown in Table 6-9.

Table 6-9. *Salutation table*

Id	salutation
1	Mr
2	Mrs
3	Miss

The new clients table is shown in Table 6-10.

Table 6-10. *Clients table*

id	name	mobile	city_id	salutation_id
1	Lewis	0798456325	2	1
2	Jane	0796547896	1	2
3	Cindy	0741598745	3	3
4	Roger	0749856547	2	1

Your query will now be changed to look like this:

```
SELECT * FROM
clients c JOIN client_languages cl ON c.id=cl.client_id
JOIN languages l ON l.id = cl.language_id
JOIN salutations s on s.id=c.salutation_id
```

Last Word on Joins

First and foremost, you only join on SELECTS.

Secondly, you get different kinds of joins. I will quickly mention LEFT JOINS and RIGHT JOINS here. Consider the following query:

```
SELECT * FROM
clients cl JOIN cities cit ON cl.city_id = cit.id
WHERE cit.city = 'London'
```

In the case of the system coming across a clients entry that has no cities entry (in other words, no corresponding cities key in the city_id table), then the complete entry will be omitted. If you want to ensure that you display data for all clients, even if they have no city_id key, you should use a LEFT JOIN. A LEFT JOIN ensures that all data to the left of the JOIN gets returned, even if the data it joins on is empty. A RIGHT JOIN does the same, but just considering the right-hand side instead of the left-hand side.

Conclusion

In this chapter, you learned how to set up your database, set up Adminer, and create databases, tables, and queries. You saw how to normalize them, which is a very important aspect in database creation. You will get to a stage where you do not think in terms of normalization anymore; it is just naturally how you design your database. Later in this book, this knowledge will be used when you build a small application in Python, which will use database calls to store and retrieve data.

Cheatsheet and Checklist

Logical operators:

>,
<,
=,
!=
IN
NOT IN
IS
IS NOT

To select data from a database:

```
SELECT [columns to retrieve] FROM table-name alias WHERE field-
value [logical operator] value
```

Multiple joins:

```
SELECT [columns to retrieve] FROM table-name alias a
JOIN table-name alias b ON a.id=b.foreign_key
```

To insert data into a database:

```
INSERT INTO table-name (column-names) VALUES (values)
```

To update data in a database:

```
UPDATE table-name SET field-value = value WHERE field-value  
[logical operator] value
```

To delete data from a database:

```
DELETE FROM table-name WHERE field-value [logical operator]  
value
```

First normal form:

- Every entry should be unique.
- Every field should have one value only.

Second normal form:

- Every non primary value should be dependent on a primary key.

Third normal form:

- No non-primary values should be dependent on another non-primary value.

A primary key uniquely identifies a row in a database.

A foreign key identifies a relationship with another table. In general, a foreign key in table A is a primary key in table B.

Indexing stores all your data in a separate table, in an ordered fashion, in order to speed up searching for data.

References

Additional information about normalization:

- www.edureka.co/blog/normalization-in-sql/
- www.guru99.com/database-normalization.html

CHAPTER 7

Creating a RESTful API: Flask

In this chapter, I will show you how to build a small project in Python. You will use all the knowledge about the software development process you have learned so far from the previous chapters. Part of every project is the decisions you need to make as to what technologies you are going to use. Lots of things can affect these decisions, such as your own skillset, your team's skillset, general best practices, and your project itself. You will stick to a conservative but realistic approach to solve your problem. You will build a small system that communicates with a database. This will be a complete project setup, including the GitLab repositories, changes to the Docker file, and installing the software you need for your project. One new aspect I will introduce is the software framework. When we create systems, we don't necessarily write all the code from scratch. We may opt to use a framework. A framework houses all the basic tools we need in order to bring our software up quickly. The tools may include routing of requests, where your external request is served up via specific "routes" into the system, or the basic code for database access models.

This is all framework code that you should not need to write. You should concern yourself with solving business problems, so writing your own code to log errors, for instance, should not be part of your concerns. Frameworks provide the ability to bring a basic system up quickly, and the tools and libraries to develop faster. Basic actions are

handled by a framework, and you can concentrate on creating software for your business. If the framework is widely accepted and used by a large community of software developers, then it should be free of serious errors, since the more people use it, the more people test it. You will use a framework called Flask in this tutorial. It is a microframework, so it's very light and very appropriate for what you intend to create.

It is important to know that understanding concepts like migrations and setting up your environments are very important aspects of software development, and that is why this chapter spends quite a lot of time on these aspects before you start programming.

The Project

Modern systems are often separated into at least two separate entities, the front end and the back end. The front end is the graphical user interface you use to make requests for data retrieval and manipulation, while the back end serves the data to the front end for display and manipulates data on request of the front end. These two will commonly use an architectural design called REST to communicate. This communication is one-way. The front end initializes a request to the back end, and the back end responds. The back end does not issue requests to the front end. You can let the back end push data to the front end, but that is not REST-based technology and we will not handle it in this book.

In an architectural design where the front end (making the requests) and back end (receiving and responding to requests) are separated, the two systems can live on completely different servers. They are completely decoupled from each other, which is a great thing. Having them decoupled gives us two smaller codebases to work with, and also allow neat and clean access for other systems to access our backend system. You will focus on building a back end in this book, not the front end, but you will still be able to initialize calls to the back end as if from the front end.

The project you are going to build is very simple, and there are a lot of concepts for you to learn, even in such a small example. The project you are going to build will update a single database table and will update a very limited set of user data. Data in this table can be created, read, updated, and deleted. In software engineering, this is called CRUD (Create, Read, Update, Delete). The architectural pattern you will use is called REST, and you will create a REST API that will allow you to perform CRUD operations on your data table. The aim is for you to learn the following concepts during this project:

- Setting up dependencies in Docker using a `requirements.txt` file
- Setting up a REST API in Docker
- Database migrations
- CRUD operations using raw SQL
- CRUD operations using the SQLAlchemy ORM
- Using a config file in Flask

What Is REST?

REST, or RESTful, is a popular architectural pattern that allows a system to serve data to an external source, like a front end, in a reliable and predictable way. It allows different systems, written in completely different technologies, to communicate with your system. It is programming language-agnostic, but you have to adhere to certain standards to make the interoperability between different systems dependable. REST stands for Representational State Transfer. Completely dumbed down, REST is an HTTP, or preferably an HTTPS, request very similar to requesting a normal website, but with very specific rules that allow for dependable

and understandable data transfer. So, a REST server consists of different endpoints, each of which is identified by a URL, and the action on that URL is dictated by an HTTP verb, which is explained below.

Figure 7-1 demonstrates the request/response relationship between a user (who will be using a browser) and the server.

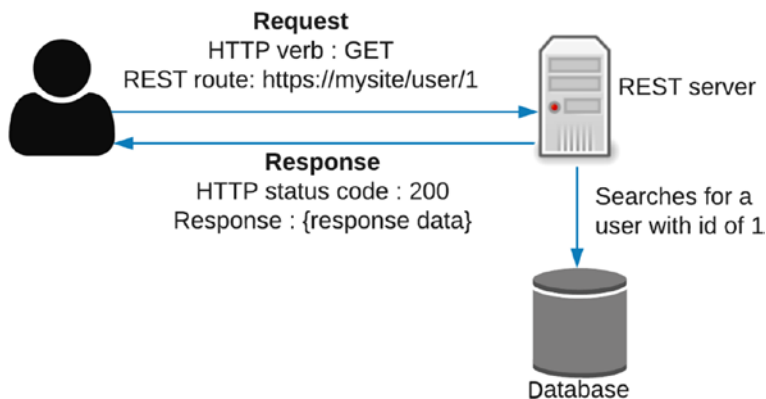


Figure 7-1. Request/response relationship

Here are some the basic concepts to understand when creating a REST API.

JSON

JSON as an acronym that stands for JavaScript Object Notation. It is a plain text data format that allows us to send data requests and receive data requests in an easy-to-interpret format. JSON is one of a few ways to send data between systems and it is the format we will use. A JSON string is encased in {} and consists of a key:value pair. The key should be between “” and the value should be between “” since it is a string. The value can also be a JSON string, so you can nest JSON within JSON. JSON can also take an array. This is a nested structure that is not a key-value pair and they are indicated by []. Some typical examples are as follows.

- Simple JSON string:

```
{"name": "Roy", "age" 60}
```

- JSON with an array:

```
{"name": "Roy", "age" 60, "hobbies": ["cooking",  
"reading"]}
```

- JSON with nested JSON:

```
{"name": "Roy", "age" 60, "address": {"street_name":  
"old street", "phone_number": "0556325541"}}
```

HTTP Verbs

An HTTP verb is something you use implicitly with every page load and every form submission on the Internet. It is a mandatory part of every HTTP call. In REST, you use the verb explicitly to tell the back-end system what the action is going to be. There are a few verbs, but five are of importance in a REST system:

- GET retrieves data from the system.
- POST add new data to the system.
- PUT replaces an entry completely in the system.
- PATCH updates an entry in the system.
- DELETE deletes an entry from the system.

These five verbs govern the actions on your REST system. They are merely indicators, though, and you still have to write the code to act accordingly when your system encounters a request from any of these verbs. I have seen some places implement only a single PUT or PATCH to do updates, instead of a PUT and a PATCH. It all depends on how strictly you want to stick to REST.

Based on the calls below, we can describe the anatomy of a REST call as follows:

VERB url-location/resource-we-want/<optional-identifier>
{Request body in JSON. Needed if VERB is not GET or DELETE}

These examples all exclude an element called HATEOAS, which you will encounter a little bit later in the book.

As an example, if you have a route called <https://api-vehicles.com/vehicles/100>, then the HTTP verbs above will signal the following to the system:

- Retrieve all vehicles from the system where vehicle id is equal to 100:

GET <https://api-vehicles.com/vehicles/100>

- Similar to this, this query will retrieve all vehicles because no id is present:

GET <https://api-vehicles.com/vehicles>

- This query will delete the vehicle with id of 100:

DELETE <https://api-vehicles.com/vehicles/100>

The following three queries need a body containing the data you want to change or add. This body is in general in JSON, which is a convenient format to transport data around. JSON is a format that follows strict rules. As explained above, the JSON-formatted text is between the curly braces under the URL.

- This endpoint will replace the complete record found at id 100:

PUT <https://api-vehicles.com/vehicles/100>
{ "car": "ford", "color": "red", "production_year": 2015 }

- This endpoint will update the record found at id 100:

```
PATCH https://api-vehicles.com/vehicles/100
{"car":"ford", "color":"red"}
```

- This endpoint will create a new record:

```
POST https://api-vehicles.com/vehicles
{"car":"ford", "color":"red", "production_year":2015}
```

REST Query Routes

When you access a website, you enter a URL. When code accesses a REST system, it does it via a URL as well, but it doesn't use a browser for it. The Flask Python library will handle your query routes for you. I will go through it in more depth, but for now it is good to know that you specify an entry point using what is called an *annotation*. This annotation is placed just before the function you want to run when that endpoint is accessed. The following snippet tells us that when the system received a request with a hello in the URL, to run the hello function:

```
@app.route('/hello')
def hello():
    return 'Hello'
```

HTTP Status Code

Each and every HTTP/HTTPS request returns an HTTP status code. This code means something to the browser doing the request, and tells the browser, or the requesting system, whether that request ran into any problems or whether it succeeded. In REST, we use this status code to convey a message directly relating to the status of the result of the request, NOT the status of the request itself. Your request can succeed in the sense that it has found your system, and your system has accepted the request.

It should respond with a 200. A 200 status code means that the location of the URL was found and executed correctly. However, if your request was to fetch data, and the request successfully reached your system, but no data was retrieved because there was no data associated with your request, then you will get a code that is normally reserved for when a web page cannot be found, a 404. So bear that in mind. The status code does not relate to whether your REST resource was found; it relates to the result of the computation behind the resource.

HATEOAS

HATEOAS is a fundamental aspect of REST. It is a bit of a mouthful but it will become clearer as you encounter it more often. It stands for “Hypermedia As The Engine Of Application State.” HATEOAS dictates that you should not only send back data, but also the actions you can perform on that data. This decouples the client that is making the requests from the server endpoint it is requesting data from. One of the things you can send back as part of your HATEOAS response is pagination data. Pagination refers to when you request, for instance, the first 10 values of 100 values, and the response will have embedded links to the next page and the previous page. Assume you have just called this endpoint: <https://api-vehicles.com/vehicles/100>. In the following response example, you link to the previous and next pages, taking the created URLs that the server has built for you. You also not only send back the vehicle data, but also what you can do with that vehicle. The server decides what the URL will be (in here as an href) and decouples the system calling the REST API from each other.

```
{
  "id": 100,
  "car": "ford",
  "color": "red",
```



```

"production_year":2015
"links": [
    {
        "rel": "remove",
        "href": "https://mysite/vehicles/100"
        "method": "DELETE"
    },
    {
        "rel": "self",
        "href": "https://mysite/vehicles/100"
        "method": "GET"
    },
]
"pagination": {
    "urrent_page": 2
    "next_page": "https://mysite/vehicles?page=3"
    "previous_page": "https://mysite/vehicles?page=1"
}
}

```

The Technology You Will Use

As an architectural pattern, you will use REST. REST was explained in the section above. Your environment will run in multiple Docker containers. Your REST endpoints will need something to handle the incoming requests, and for that, you will use Gunicorn as a webserver gateway. Gunicorn handles requests between the outside world and your webserver, but it can serve responses based on requests on its own. Ideally, you should have a webserver as well, but that is beyond the scope of what I want to show. For now, Gunicorn is a good solution for your needs. Your database will remain MySQL and your programming language will be

Python. As a framework, a concept I will discuss later, you will use Flask. I will also introduce a concept called *migrations*. Migrations handle the creation and modification of your database's structure and are the database equivalent of sliced bread. You will use a library called flask-migrate to handle your migrations.

Setting Up the Environment

First, you start by creating a GitLab project. This allows you to safely store and share your code.

Creating the GitLab Project

Log into your GitLab account, and create a project called Users. After it has been created, GitLab will show you different instructions on how to initialize the project locally. Follow the steps you will find under **Create a new repository**. They allows you to create a new repository on your computer, and set it up to point to your remote GitLab project. Just as a hint, the following command creates a directory called blog and initializes it:

```
git clone git@gitlab.com:xyz/users.git
```

The following command will not create a directory, but will initialize the directory you are in. Notice the full stop after the command. For this command, you must create the directory yourself and cd into it before you can run it. It is a good idea in the beginning to just stick to the tips GitLab gives you.

```
git clone git@gitlab.com:xyz/users.git .
```

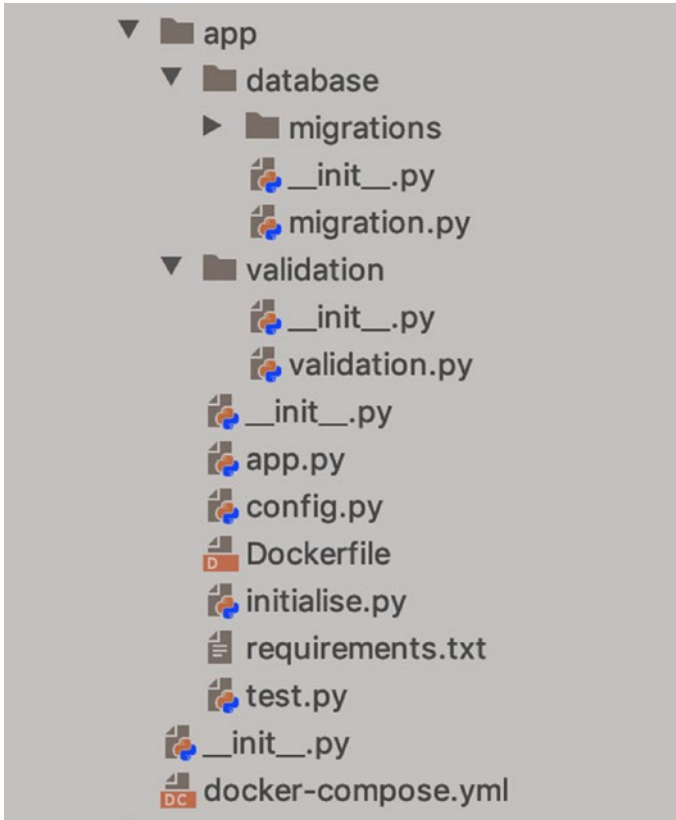
Project Layout

You should now have a directory called `users`. Follow these steps carefully:

- In the root directory of your new project (that is, inside the `users` directory you just created), create a file called `docker-compose.yml`.
- Also, create a directory called `app` in your root directory.
- Create the following five files and two directories inside the newly created `app` directory:
 - `app.py`
 - `config.py`
 - `initialise.py`
 - `requirements.txt`
 - `Dockerfile`
 - A directory called `database`
 - A directory called `validation`. You will use this directory in a later chapter.
- Inside the `app/database` directory, create the following file: `migration.py`
- Inside the `app/validation` directory, create the following file: `validation.py`

Your directory structure should look like this now:

users (This is your root directory)



This is the rough outline of your project. You will now start fleshing these files out to create your project.

Creating the docker-compose and Docker Files

You will be able to reuse most of your docker -compose file so copy that over and just make the changes in the new file.

File Changes to `docker-compose.yml`

Change the `python-dev` section's `container_name` attribute to `flask-server`, as per Listing 7-1.

In the `build` section, do the following:

- Change context to `./app`, as this is where your app's Dockerfile and code will live.
- Mount the `./app` directory to the working directory, `/var/app/flask_app`, inside your container.
- Expose port 8000 to the outside world using the `ports` directive. You will use this port to communicate with the system.

Listing 7-1. The Docker-compose file

```
python-dev:
  container_name: flask-server
  build:
    context: ./app
    dockerfile: Dockerfile
  volumes:
    - ./app:/var/app/flask_app
  ports:
    - "8000:8000"
```

Docker File

You need to jump to the Dockerfile next. The Dockerfile is linked to the `flask-server` container and needs the most changes. You are going to introduce a new file to the Dockerfile, the `requirements.txt` file. This file is just a list of dependencies that the Dockerfile must install using the `pip` command. Add these entries to the `requirements.txt` you created earlier.

The requirements file's content is in Listing 7-2.

Listing 7-2. The requirements file

```
Flask==1.1.2
Flask-Migrate==2.5.3
Flask-Script==2.0.6
mysql-connector-python==8.0.5
Flask-SQLAlchemy==2.4.4
gunicorn==20.0.4
psycopg2-binary==2.8.6
marshmallow==3.7.1
```

After you have added them, run this command:

```
pip -r install requirements.txt
or
pip3 -r install requirements.txt
```

Here you are basically saying you want to install Flask as your framework, flask migrate to handle your database's table creator, mysql-connector-python as your database connection driver, and very importantly, Gunicorn to serve up your requests. Marshmallow will be used when you validate incoming requests. You also pin the libraries you want down to specific versions to prevent incompatible updates from breaking your system.

The following changes are to be made to the Dockerfile itself. Its contents should look like Listing 7-3.

Listing 7-3. The Dockerfile

```
FROM python:3.7.5-slim

RUN mkdir -p /var/app/flask_app
WORKDIR /var/app/flask_app
COPY requirements.txt /var/app/flask_app
```

```

RUN pip install --no-cache-dir -r requirements.txt
COPY . /var/app/flask_app
ENV FLASK_APP ../app.py

CMD exec gunicorn -w 1 -b :8000 app:app --reload

```

In here you are creating the `/var/app/flask_app` directory path that you use in the `docker-compose` file, and you switch your working directory to it. Then you copy the `requirements.txt` file to it, and run `pip install` on every entry. After that, you copy the files in your local environment to that same directory, and tell the environment, that is the `ENV` variable, that `FLASK_APP`, when used in the system, points to `../app.py`. This variable is only needed in the migrations, which live in the database directory. It has to traverse one level back and that is why there is a `../` before the application name. On the very last line, you tell your system to run Gunicorn on port 8000 running the `app.py` script. The `--reload` option enables you to change code and lets Gunicorn reload on detecting a code change, without you having to manually restart the server. **You should note that this is for development only and not production.** You should reload Gunicorn explicitly yourself on production.

docker-compose.yml: DB Server and Adminer Sections

There are no changes to this portion of the `docker-compose.yml` file, apart from changing the container names to `flask-db` and `flask-adminer`, and adding the volume segment at the bottom. See Listing 7-4.

Listing 7-4. The Docker-compose file’s database section

```

mysql-dev:
  image: mysql:5.7.22
  container_name: flask-db
  ports:
    - 6603:3306

```

```
environment:
  MYSQL_ROOT_PASSWORD: "root"
volumes:
  - database-folder:/var/lib/mysql
```

```
adminer:
  image: adminer
  container_name: flask-adminer
  restart: always
  ports:
    - 8080:8080
  links:
    - mysql-dev
```

```
volumes:
  database-folder:
```

Your docker-compose.yml file should now look like Listing 7-5.

Listing 7-5. The Docker-compose file

```
version: "3"

services:
  python-dev:
    container_name: flask-server
    build:
      context: ./app
      dockerfile: Dockerfile
    volumes:
      - ./app:/var/app/flask_app
    ports:
      - "8000:8000"
```



```
mysql-dev:
  image: mysql:5.7.22
  container_name: flask-db
  ports:
    - 6603:3306
  environment:
    MYSQL_ROOT_PASSWORD: "root"
  volumes:
    - database-folder:/var/lib/mysql
```

```
adminer:
  image: adminer
  container_name: flask-adminer
  restart: always
  ports:
    - 8080:8080
  links:
    - mysql-dev
```

```
volumes:
  database-folder:
```

Bringing Up the New System

You must ensure that your other Docker instances from your previous lessons are not running, mainly because they are all using the same ports. You can change the ports, but it is also not good to have loads of containers running on your local machine. To ensure they are not running, just go into that project directory and run these commands.

This command will show if anything is running:

```
docker-compose ps
```

This command will take the system brought up by docker-compose in that directory down:

```
docker-compose down
```

Because you edited the Dockerfile, you need to rebuild your container environment. Make sure you are in the root folder of your Users project, and run this command:

```
docker-compose build --no-cache
```

This should take a minute or two to run but will install all your dependencies from the requirements.txt file. Before you bring your server up, make sure your app.py file is in a useable state so that it can actually serve requests to you.

Testing the Application

The initial application will be incredibly simple and will just demonstrate how the REST components, and database interactions work. If you are on Linux or Mac, the URL you use to access your system is `http://127.0.0.1:8000`.

If you are on Windows and running it through a virtual machine, as you set it up in the beginning, you need to do the following.

Run the following command:

```
docker-machine ls
```

The output will have a URL section in it. The URL will be an IP address preceded by a `tcp://` and may have a port number after a colon. It may look like this: `tcp://192.168.99.100:2343`. Copy the IP address, without the `tcp://` section and without the port, and use it in the URL to access your system. For instance, you will only use the text in blue, `tcp://192.168.99.100:2343`. Your URL will then look like this: `http://192.168.99.100:8000`.

Inside the app directory, you created an `app.py` file earlier. Add the code in Listing 7-6 to it.

Listing 7-6. Routes

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def home():
    return 'This is the home page'

@app.route('/hello')
def hello():
    return 'Hello'
```

In this code snippet, you import Flask and initialize the Flask app. Then you declare two entry points into your system using two functions. The first one is called `home()` and the second one is called `hello()`. These two access points can be accessed via a browser. Let's test it quickly.

First, you need to bring the Docker container up.

```
docker-compose up
```

If there are no errors, open a browser, and type the following into the address bar:

```
http://127.0.0.1:8000/
```

and separately

```
http://127.0.0.1:8000/hello
```

You should see the different outputs as you specified in your code. In your browser, the text “This is the home page” and “Hello” should appear.

Testing the Application with Postman

You will not be testing your system with a browser. You cannot easily simulate POST, PUT, PATCH and DELETE on the fly with a browser since the default HTTP verb used in a browser is GET, and you are not going to write HTML now to create the other verbs. You will also not go with the command line in this instance. Instead, you will use an application called Postman.

Postman is a great tool for remote calls like REST calls. Download and install Postman for your computer from this location: www.postman.com.

Once installed, you can start creating requests.

- You will see a layout screen where you can create tabs. Each tab can do a request.
- There is an address bar with the words “Enter request URL.” This is where your request URL will go.
- To the left of this bar is a dropdown menu and you will find the HTTP verbs in there. By default, it is set to GET.

Do the following steps to test a GET request with Postman:

- Keep the verb as GET.
- Type the same URLs you typed in the browser into Postman’s address bar.
- Click the Send button.

The response should appear in the response section in the bottom. You will use Postman to test all the endpoints of your system.

Migrations

You have one more step to figure out before you have a fully usable project: migrations. Migrations are the name you give to the automated creation and editing of your database tables and structures. You will not create your

tables inside Adminer, as in the previous chapter about databases. Instead, you will create it in your migrations. (You will, however, create your database in Adminer.) This allows you to port the exact same database structure to any new installation of this system. You can track changes to the system, and easily deploy your code and databases. You will also use the objects used by the migrations for your ORM models.

You will complete the following steps to get migration up. Log into Adminer and create a database called `users`. To log into Adminer, using Linux or Mac, use this URL from the browser:

<http://127.0.0.1:8080>

If you are on Windows and running it through a virtual machine, which is one of the two setup methods you had in the beginning, you need to do the following.

Run the following command:

```
docker-machine ls
```

The output will have a URL section in it. The URL will be an IP address preceded by a `tcp://` and may have a port number after a colon. It may look like this: `tcp://192.168.99.100:2343`. Copy the IP address, without the `tcp://` section and without the port, and use that in the URL to access your system. For instance, you will only use the text in blue: `tcp://192.168.99.100:2343`. Your URL will then look like `http://192.168.99.100:8080`.

In the database directory's `migration.py` file you created earlier, add the code from Listing 7-7.

Listing 7-7. Migration file

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate, MigrateCommand
from flask_script import Manager
```

```
app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] =
'mysql+mysqlconnector://root:root@flask-db/users'

db = SQLAlchemy(app)
migrate = Migrate(app, db)

handler = Manager(app)
handler.add_command('db', MigrateCommand)

class Users(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(128))
    surname = db.Column(db.String(128))
    identity_number = db.Column(db.Integer())

if __name__ == '__main__':
    handler.run()
```

The first four lines are imports of libraries you need for this code to work. On line seven, you create the database connector string. This string gives your system the username, password, location, and database name of the database you need to connect to. If you have followed the tutorials word for word, then this string should work, but here is an explanation of it.

Table 7-1 deconstructs the connection URI `mysql+mysqlconnector://root:root@flask-db/users`.

Table 7-1. *Deconstructed URI*

mysql+mysqlconnector	Connection protocol
root:root	Username:password
flask-db	The container name of the database
Users	The name of the database

SQLAlchemy is the library you will use as an ORM. This is an object-relational mapper that allows you to access the database without writing any SQL code. To achieve this, you create a Python class (see Listing 7-7 below), which via SQLAlchemy is mapped directly onto a data table. It is mostly a good idea to use an ORM. In systems with big complex databases, however, it may not always be a good thing. You will look at an example of an ORM as well as SQL. In these examples, SQLAlchemy will also be responsible for the raw SQL queries.

The following lines in Listing 7-7 are very important, and basically tell the migration system exactly how to create or edit the new table. Here you can see it addresses a table called `Users` and adds a primary key called `id`, two string columns called `name` and `surname`, and an integer column called `identity_number`. This will be your model, and you will use it later as well.

```
class Users(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(128))
    surname = db.Column(db.String(128))
    identity_number = db.Column(db.Integer())
```

This is the basic layout for a migration, but also very importantly, this is the SQLAlchemy model you will use later in order to access the database as well. All tables in your database will be added in this file.

Running a migration is initially a three-step process. Afterwards, it takes two steps.

Migration Preparation Step

You need to mount the flask-server Docker image. This is because you communicate between Docker instances using the container names, and those names are only recognizable within Docker.

```
docker exec -it flask-server bash
```

Navigate to the database directory.

```
cd database
```

You will now be in the directory where the migration script lives.

Initial step (run only the first time):

```
python migration.py db init
```

These two steps are run whenever the database schema changes:

```
python migration.py db migrate
```

```
python migration.py db upgrade
```

`python migration.py db migrate` is run to prepare the migrations and detect changes, and `python migration.py db upgrade` is run to perform an actual migration.

Run these three steps, and inspect your database in Adminer. You should see a newly created table called `users` with four columns. You will use this database and table in the next section.

At this point, commit your work and push it to the GitLab repository.

```
git add .
```

```
git commit -m 'Initial system setup'
```

```
git push origin master
```

The Final Steps: Coding

I would like to point out again that the project does not start when the coding starts (or what could be seen as the coding portion). The project, from a software engineering perspective, starts when you make your choices as to what software you will use. Setting up Docker (or just selecting a precreated docker-compose file) and remote repositories are all part of the project.

As a last step, you will create eight endpoints into the system. You only need four, but you will duplicate them so that one half uses the ORM and one half uses raw SQL. (When we write our SQL ourselves, we sometimes refer to it as raw SQL.)

The first version will use raw SQL, and the second version will use the ORM. This way you can see how both will work.

Step 1

You can make an improvement to the `migration.py` file. It connects to the database, but that database connection setup code can be reused. To achieve this, start with the `initialise.py` file. Open it and add the code in Listing 7-8 to it.

Listing 7-8. Initialise.py file

```
class Initialise():

    def db(self, app):

        app.config.from_object("config.Config")

        app.config['SQLALCHEMY_DATABASE_URI'] =
        'mysql+mysqlconnector://{user}:{password}@{host}/{db}'.format(
            app.config["DB_USERNAME"],
            app.config["DB_PASSWORD"],
            app.config["DB_LOCATION"],
            app.config["DB_DATABASE"]
        )
        app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = True
        return app
```

In this class, you are creating the Flask app. This constructor is called twice in your system, but that is not the main reason for creating a reusable object. The main reason is the database connection URI, which is used

in two places. The first place is in the `migrations.py` file, and the second place is in the `app.py` file, which you have yet to implement. Upon creating the Flask app, you can assign the mysql URI to it. You create the URI from a config file. This allows you to have one single point where the database connection details live. Notice the structure of the parameter of `app.config.from_object("config.Config")` in Listing 7-8. In the parameter `"config.Config"`, the first lowercase `config` refers to the filename and the capitalized `Config` refers to the class name inside the `config` directory, which you want to access. In Listing 7-9, the `Config` class in the `config` directory can be seen.

Step 2

Before you can go on, you need to populate the `config.py` file with values that the `Initialise` class can use. Add the code in Listing 7-9 to the `config` file and change the values to the values of your database.

Listing 7-9. Config.py file

```
class Config(object):
    DB_USERNAME = 'root'
    DB_PASSWORD = 'root'
    DB_LOCATION = 'flask-db'
    DB_DATABASE = 'users'
```

Step 3, A Bit of Refactoring

With the `Initialise` class complete, you can add it to the `migrations.py` file. Open `migrations.py` and change the following.

Add an additional import:

```
import sys
sys.path.append("..")

from initialise import Initialise
```

Remove these lines from the `migration.py` file:

```
app.config['SQLALCHEMY_DATABASE_URI'] =
'mysql+mysqlconnector://root:root@flask-db/users'
```

And add this in their place:

```
init = Initialise()
app = init.db(app)
```

Your migrations file should now have the following lines:

```
app = Flask(__name__)
init = Initialise()
app = init.db(app)
```

You now have an `app` object which is a Flask instance with the correct database connection URI.

Step 4

You can now clear the contents of `app.py` and replace it with the initial portion of your system, as seen in [Listing 7-10](#).

Listing 7-10. New `app.py` contents

```
import json
from flask import request
from sqlalchemy import text
from flask_sqlalchemy import SQLAlchemy
from http import HTTPStatus
from initialise import Initialise
from flask import Flask

# Creates the Flask app
app = Flask(__name__)
```

```
init = Initialise()  
app = init.db(app)  
  
# Creates the db connection  
db = SQLAlchemy(app)
```

This application is going to be a bit bigger and you will be using a few more libraries. Listing 7-10 contains the following:

json : Used to format json requests.
request : Used to read incoming request bodies with
text : Used to enable raw SQL in our classes
http : Used to access HTTP status codes as constants.
Initialise : Our own Initialise class, used to instantiate the Flask object.

Step 5

The first endpoint you will construct is the POST endpoint with raw SQL. Before you do that, let's just take a quick look at the anatomy of an endpoint. This is pretty much generic whether you use raw SQL or an ORM.

Anatomy of an Endpoint

You start with a route annotation. This indicates the entry point and method, defined as a verb, with which to access the system. By default, the method selected is GET. You can enter it as an array, since one endpoint may be accessed by many verbs if you so choose. The <identifier_variable> is only used to identify a resource, and POST will never have a value here.

```
@app.route('route/to/endpoint/<identifier_variable>',  
methods=['GET'])
```

The next line contains the function declaration. If you need to pass a variable from the route, you can pass it through here:

```
def function(<identifier>):
```

Typically, you return JSON. I will show clear-cut examples in the section that follows.

Anatomy of the Code Inside the Function

Let's handle the raw SQL examples first. Because you will be using raw SQL, you need to escape the SQL yourself. Escaping SQL means you protect it against SQL injection attacks. I will cover this in the chapter about security, but for now, it is very important to know to escape your queries. It is paramount that you protect your queries against SQL injection attacks. You escape your queries using parameter binding. The following steps inside your functions are important:

- You can get a parameter that is in the URL into your code by putting that parameter in the function's parameter list. See Listing 7-12 for an example.
- You can get your POSTed data body, handled in the very next section, by using `request.get_json()`. See Listing 7-11 for an example.
- You add your SQL to a function called `text()`. This improves back-end support for parameter binding. You put a `:parameter_name` wherever a parameter should be, like this:

```
sql = text('SELECT * FROM banking WHERE id = :id
AND code = :some_code') The :values are basically
placeholders for the real values.
```

- Then you bind it when you run execute on it:

```
result = db.engine.execute(sql, id=10, some_code=111).
first()
```
- `json.dumps` formats a string or array to a JSON-formatted string.
- `HTTPStatus` is responsible for the status code.

Add Listing 7-11 to your `app.py` file.

Listing 7-11. Post endpoint and HATEOAS

```
def hateoas(id):
    return [
        {
            "rel": "self",
            "resource": "http://127.0.0.1:8000/v1/
users/" + str(id),
            "method": "GET"
        },
        {
            "rel": "update",
            "resource": "http://127.0.0.1:8000/v1/
users/" + str(id),
            "method": "PATCH"
        },
        {
            "rel": "update",
            "resource": "http://127.0.0.1:8000/v1/
users/" + str(id),
            "method": "DELETE"
        }
    ]
```

```

@app.route('/v1/users', methods=['POST'])
def post_user_details():
    try:
        data = request.get_json()
        sql = text('INSERT INTO users (name, surname, identity_
            number) values (:name, :surname, :id_num)')
        result = db.engine.execute(
            sql,
            name=data['name'],
            surname=data['surname'],
            id_num = data['identity_number']
        )
        return json.dumps({"id": result.lastrowid, "links":
            hateoas(result.lastrowid)})
    except Exception as e:
        return json.dumps('Failed. ' + str(e)), HTTPStatus.
            NOT_FOUND

```

This endpoint is very important to understand, as all of this information trickles down to the other endpoints.

Look at the first function in Listing 7-11, `def hateoas(id)`. I add this function to demonstrate the concept of HATEOAS. You will add it to all of your returns in your functions. By inspecting the function you can see what value it brings to the API.

The very first line after the `hateoas` function, `@app.route`, declares the entry point's structure and verb. This line says that the entry point can be accessed using `http://127.0.0.1:8000/v1/users`, using the POST verb. Note that you are omitting the `http://127.0.0.1` part. You add a `v1` into your URL to make it easier for you to switch to `v2`, which you will use when you create the same calls using ORM models.

`data = request.get_json()` fetches the POST body from the request. The next two lines perform the SQL query. `json.dumps` formulates the JSON to be sent back and includes the HTTP status id. The status id relates to a code. For instance, `HTTPStatus.OK` results in a value of 200, and `HTTPStatus.NOT_FOUND` results in a value of 404. The name of the endpoint is `users`. This is a good name for this endpoint.

To call this endpoint, you need to do the following in Postman:

1. Open a new request. This is as simple as opening a new tab in Postman.
2. Set the HTTP verb in the dropdown to POST.
3. You need to pass a request body within the request. To do this, follow these steps.
 - Once you have selected POST as a verb, you should be able to click a Body link just below it. You cannot click this link if your verb is set to GET.
 - Once you have clicked Body, a menu will appear under that. Select Raw from that menu.
 - Once you have selected Raw, a dropdown will appear to the right. The first option should be Text. Open it and select `Json(application/json)` from it.
 - The request should be properly formed json. Here is an example. After you have added it, click Send.
 - `{"name":"John","surname":"Connor","identity_number":"902308"}`

You should see a result returned to you. The word “Added” should appear in the response screen at the bottom, and the resulting HTTP status code should be 200.

Step 6

First of all, log into Adminer and inspect your database. Can you see the entry in the database? If not, there may be something wrong with your POST method and you may need to revisit your implementation. If you can see the entry, then all is great! Put the GET endpoint into `app.py` and test it. See Listing 7-12. Add this code to your `app.py` script. Note that you pass a `user_id` through the router and pass that `user_id` into the function. The rest of the code is similar in explanation as the POST endpoint.

Listing 7-12. Get endpoint

```
@app.route('/v1/users/<user_id>', methods=['GET'])
def get_user_details(user_id):
    try:
        sql = text('SELECT * FROM users WHERE id=:id_num')
        result = db.engine.execute(sql, id_num=user_id).
            fetchone()
        return json.dumps({"name": result.name, "surname":
            result.surname, "identity_number": result.identity_
            number, "links": hateoas(user_id)}), HTTPStatus.OK
    except Exception as e:
        return json.dumps('Failed. ' + str(e)), HTTPStatus.
            NOT_FOUND
```

To test, open a new tab in Postman. Type the route to your endpoint in the address bar and set the verb to GET. You should now add the id you are looking for at the end of the URL. The id will identify the exact user you want to retrieve. The more users you create with POST, the more ids you have to test.

`http://127.0.0.1:8000/v1/users/1`

Step 7

The next endpoint you will look at is the patch endpoint. Patching does have a slight layer of complexity where you need to build the update query, especially relating to parameter binding. The rest is basically the same as the POST and GET endpoints. See Listing 7-13.

Listing 7-13. Patch endpoint

```
@app.route('/v1/users/<user_id>', methods=['PATCH'])
def patch_user_details(user_id):
    data = request.get_json()
    """
    An update query has a portion where you specify which
    values to update. Because we are sending through a
    variable amount of columns to change, we need to build
    the clause in the SQL that states what must change,
    programmatically
    """
    for key in data:
        update_string = key + '=: ' + key + ', '
        # Remove the last comma in the update portion
        update_string = update_string[:-1]
    try:
        data['id'] = user_id
        sql = text('UPDATE users SET ' + update_string + '
        WHERE id = :id')
        result = db.engine.execute(sql, data)
        return json.dumps(
            {
                "id": user_id,
                "links": hateoas(user_id)
            }
        )
```

```
except Exception as e:
    return json.dumps('Failed. ' + str(e)), HTTPStatus.
    NOT_FOUND
```

In Listing 7-13, in the for loop, you concatenate the values to be updated to the update string. PATCH has an id in the URL as well as a POST body. The id in the URL identifies the user you want to update, and the POST body contains the fields to be updated. You can test PATCHing in Postman by doing the following. Open a new tab, and select PATCH as the HTTP verb. The rest of the settings should be similar to a POST setting. There is one exception, though. The URL should contain the id of the field you wish to update. If you have a value in your database where id is equal to 1, then add 1 to the end of the URL, as is evident in the route annotation.

```
@app.route('/v1/user/<user_id>', methods=['PATCH'])
```

The body only needs to contain the value or values you wish to update. Add the following as an example to the request in your call to the update endpoint:

```
{"name": "newname"}
```

Add that above to your PATCH's request body in Postman and click Send. To check if the data has been changed, you can run the GET endpoint on it again. Alternatively, you can look it up in the database, but I strongly recommend using the GET parameter.

Step 8

The next endpoint you will create will be for deletions. Like GET, DELETE only takes an id to identify the resource to delete. In this case, it is also called `user_id`. You do not add the HATEOS function here. You do not have other default HATEOAS behavior, and after you have deleted a user, you cannot read, update, or delete it. See Listing 7-14.

Listing 7-14. Delete endpoint

```
@app.route('/v1/users/<user_id>', methods=['DELETE'])
def delete_user_details(user_id):
    try:
        sql = text('DELETE FROM users WHERE id=:id_num')
        result = db.engine.execute(sql, id_num=user_id)
        return json.dumps('Deleted'), HTTPStatus.OK
    except Exception as e:
        return json.dumps('Failed. ' + str(e)), HTTPStatus.
        NOT_FOUND
```

Select an id you want to delete, go to Postman, based on your previous experiences with Postman, and set up a delete call. Once again you can use GET to see whether the record has been deleted or not. You should now have a working RESTfull system, which can create, read, update, and delete

The ORM Version

Take a few minutes to look at the ORM versions of the above code. There are three changes:

- The route used to have a v1 in it. Now it is v2. With this, you can keep the route endpoints the same, as users, but you will invoke a different function on the back end when you call v1/users or v2/users.
- The PATCH endpoint does not need to build its own input query. Instead, you just give the whole dataset to the ORM to update your entry with.
- All the raw SQL disappears and is replaced by the ORM syntax.

The rest of the code stays exactly the same. Patch takes the biggest change in code due to the automatic handling of the update section. With your ORM handling the updates, you can literally just pass it your dataset. Listing 7-15 contains the complete changed set of functions, which you can just add to the bottom of the `app.py` file.

Listing 7-15. ORM version of the endpoints

```
@app.route('/v2/users/<user_id>', methods=['GET'])
def get_user_details_orm(user_id):
    try:
        result = db.session.query(Users).filter_by(id=user_id).
            first()
        return json.dumps(
            {
                "name":result.name,
                "surname":result.surname,
                "identity_number":result.identity_number,
                "links": hateoas(user_id)
            }
        ), HTTPStatus.OK
    except Exception as e:
        return json.dumps('Failed to retrieve record. ' +
            str(e)), HTTPStatus.NOT_FOUND

@app.route('/v2/users', methods=['POST'])
def post_user_details_orm():
    try:
        data = request.get_json()
        user = Users(name=data['name'], surname=data
            ['surname'], identity_number=data['identity_number'])
        db.session.add(user)
        db.session.commit()
```

```

        return json.dumps(
            {
                "id": user.id,
                "links": hateoas(user.id)
            }
        ), HTTPStatus.OK
    except Exception as e:
        return json.dumps('Failed to add record. ' + str(e)),
            HTTPStatus.NOT_FOUND

@app.route('/v2/users/<user_id>', methods=['PATCH'])
def patch_user_details_orm(user_id):
    try:
        data = request.get_json()
        user = db.session.query(Users).filter_by(id=user_id).
            update(data)
        db.session.commit()
        return json.dumps(
            {
                "id": user_id,
                "links": hateoas(user_id)
            }
        )
    except Exception as e:
        return json.dumps('Failed to update record. ' +
            str(e)), HTTPStatus.NOT_FOUND

@app.route('/v2/users /<user_id>', methods=['DELETE'])
def delete_user_details_orm(user_id):
    try:
        user = db.session.query(Users).filter_by(id=user_id).
            first()

```

```

    db.session.delete(user)
    db.session.commit()
    return json.dumps('Deleted'), HTTPStatus.OK
except Exception as e:
    return json.dumps('Failed. ' + str(e)), HTTPStatus.
    NOT_FOUND

```

NB: Remember to add this to your git repository using the following commands:

```

git add .
git commit -m 'Added CRUD v2'
git push origin master

```

Let's run through the changes in Listing 7-15 quickly.

GET Endpoint

The data selection portion changes from SQL to this:

```
result = db.session.query(Users).filter_by(id=user_id).first()
```

Users in `db.session.query` is your ORM, or object relational model. `Filter_by` filters the result based on `user_id`, and `first()` indicates you only want the first record. The `result` variable is now an object, and its members can be accessed with the dot operator. For instance, `result.name` will return the name value of the record.

POST Endpoint

The post endpoints changes to the code below:

```

user = Users(
    name=data['name'],
    surname=data['surname'],

```

```
        identity_number=data['identity_number']
    )
    db.session.add(user)
    db.session.commit()
```

You create the ORM model using the input data as parameters. You then add it to the session and commit it to the database. `Db.session.commit` means that the data is actually committed to the database.

PATCH Endpoint

Update the resource's code as follows. It's only two lines of code.

```
user = db.session.query(Users).filter_by(id=user_id).
update(data)
db.session.commit()
```

You specify the ORM model in the query function. The `filter_by` specifies which `user_ids` you want to update and the update function accepts the input data.

DELETE Endpoint

Deleting an entry is about three lines.

```
user = db.session.query(Users).filter_by(id=user_id).first()
db.session.delete(user)
db.session.commit()
```

Once again, you select the record you want to delete, feed that record into the delete function, and commit it.

It is quite evident that using an ORM can be a lot simpler and more straightforward than SQL, but it is not always the case. You may sit with really complex joins, in which case, SQL may be a better option. Decisions like whether to use an ORM or SQL all come down to planning.

Takeaway of This Chapter

This was a rather hectic chapter, filled with new ideas, and some old ones, albeit modified ones. There are even more aspects not covered in this book, and learning software development is a journey that takes a long time to complete. You looked at a more complex Dockerfile and a slightly more complex docker-compose. You explored new concepts like the requirements file, Gunicorn, and the concept of REST. A very important aspect you learned was migrations. It is a superb way to keep your databases up to date with the latest changes to the database.

I want the chief takeaways in this chapter to slot in with the previous chapters. Engineering software is not just writing code. It takes a multi-disciplinary approach to create software, and to top it all off, there are still important chapters ahead. There is still the design portion left, where code gets planned and designed. That happens before you start coding, and we always try to follow the overly optimistic saying of “plan twice, code once.”

It is always good practice to investigate the best way to set up your directory structures, technology to use, etc. What looks like a good idea today may prove to be a bad idea tomorrow. Do not just jump into a project by copying your previous project’s structure blindly. Make sure that that project was sound, and that the decisions made there were good decisions.

References

Some interesting websites where you can learn more about Python:

- <https://realpython.com/flask-by-example-part-1-project-setup/>
- <https://docs.python-guide.org/intro/learning/>

CHAPTER 8

Testing and Code Quality

Whether the system you are working on is big or small, as mentioned in the beginning of this book, the codebase's integrity is very important. Following some basic rules, like proper naming conventions, neat code structure, and just careful coding, will have an effect on the codebase's integrity. Easy-to-read code is slightly harder to break than code that is badly structured. The easier it is to read and follow the code, the easier it becomes to make changes to it. Proper naming conventions can also lead to code that will be less prone to introducing bugs than badly named variables, functions, and classes.

Not too long ago I started at a new company, and I was asked to introduce a new feature into the system. The system had a few steps. Two of these steps were *importing* and *ingesting*. Everything was fine, apart from the fact that a function that was called something like `file_import` actually did file ingesting. Not only was the name wrong. The function was actually called in the ingesting section of the code, even though its name said it did importing. Obviously, I picked up on this quickly, but imagine someone needs to implement something in the import section, decides to reuse this incorrectly named function (because hey, it says `file_import` on the tin), and above all edits that function because it does

not quite work. The knock-on effect will be that all implementations of the `file_import` function will break. So how do we prevent disasters like this from happening?

In this chapter, you will have a look at ways to mitigate these problems.

Overview of Code Quality Steps

There are a few steps you can use to mitigate introducing new errors in your code, especially if you have added features or were requested to amend an existing feature in the system. I will not delve too deep into the formalities of all of them. But you should know about them, and at least know some basics.

The very first one has no real official procedure to it. But it is simple. **Be careful.** Do not change aspects in the system if you do not know why they are there. Get to know the codebase if you don't know it already. If you do not know the codebase and get assigned work to do on it, ask for extra time to get to know it.

Run **automated tests**. If created correctly, and this covers your complete codebase, automated tests are great at picking up problems in your codebase. The idea behind automated tests is that they cover almost all aspects of your code. So should anyone implement a change that breaks something, the tests should pick up on it. The drawback is that a lot of old codebases do not have tests, or are only covered by a limited amount of tests. One of the first things that goes out of the window when pressure mounts for software developers is usually testing. But writing tests in retrospect, after the project is done, is even an slower and very painful process.

Peer reviews are a good way to determine whether the code was designed well, runs without obvious errors (because code is never considered bug-free), and above all, does what the business process requires. There are many ways of peer reviewing, but the one I will discuss

is simple and consists of another developer reviewing your code. Once again, this is a time-consuming process, and it will take the time of another developer. It is, however, a good tool to raise the confidence in the feature that was built.

The final step is a **user acceptance** test, also called a UAT. A UAT normally happens on a staging server that mimics the production server 100%. This test is very simple and aims to verify whether the user thinks you have created what they need.

Most of these steps you can only follow if you have a team of at least two people or have access to someone who can independently test your work in a user acceptance test (apart, of course, from automated tests). But it is great to know these things exist and to have an idea of what they entail. So, without further ado, let's get cracking!

Automated Testing

Automated tests come in a big variety, and there is no shortage of tools to test the back end and the front end of your system. You will look at unit tests and integration tests in this chapter. With automated testing, you set up a large group of scenarios to run against your codebase and the results expected from these scenarios. You then run the tests, where the resulting output of the scenarios are compared to the expected results for validation. If you cover enough ground with your tests, you should be able to comfortably make changes to your codebase, run the tests, and deploy. One great side-effect of automated tests is that if you cannot write a test for your code, then you may have designed it incorrectly. As mentioned in previous chapters, you should have classes and functions with a single responsibility. Once a function starts doing too much, it becomes very difficult to test. Testing forces you to write small, testable units of code, with a limited number of paths through them that can be tested.

First up to discuss is unit tests.

Unit Tests

Unit tests test specific units of code in your system. A unit test does not bother with a full execution of your software, or necessarily how these units interact. For instance, you may have a function called `valid_age(date_of_birth)`, which accepts someone's date of birth and calculates if they are old enough for a credit card. It will respond with a `True` or `False` value upon success or failure. Let's say the minimum age is twenty years old. To ensure the integrity of this function, you will set up testing scenarios, and in this case, you will have two testing scenarios. In the first scenario, you provide the function with a date of birth where the age is younger than twenty, and in the second scenario you give it a date of birth where the person's age is older than twenty. The automated test will run both the scenarios for this function only, giving the function a different date on each execution, and upon each execution, it will test the result passed back from the function against the expected result. It will not execute the code to complete a whole credit card application. This test will only test one specific function of the system.

Scenario 1

You give the function called `valid_age` a date of birth of December 12, 1978, and you expect in return a boolean value of `True`.

Scenario 2

You give the function `valid_age` a date of birth of December 12, 2010 (considering it is 2020 now), and you expect in return a boolean value of `False`.

If your tests are designed to do the above, and you get in response a `True` and `False`, respectively, then your tests have succeeded.

This is typically called a unit test, where a small unit of your code is tested for validity. Some people love them and write them for every function, whereas some people say that a large portion of unit tests are a waste of time. I can see why some unit tests can be described as a waste of

time. In the example above, where all you need to do is deduct two dates from each other, is that really worth the effort to be unit tested? What are the chances that such a simple algorithm will break? The chances are indeed super-slim that it will break, but the chance exists that a developer uses this function and passes through an incorrect date format, and that your function does not do date validation. Something like that can have unintended consequences. But in my opinion, the best consideration for whether a function, like in this example, should be unit tested, is how crucial its job is to the execution of business rules. Not issuing a credit card to someone younger than twenty does sound to me like a very crucial business rule, and I personally will want to test this function.

Writing a Unit Test

First of all, let's look at the technology you will use. For unit testing, you will use a library called `unittest`. It is built into Python, and to use it you just need to import it into your test script. You will then create an object which will extend `unittest.TestCase`. Having `unittest.TestCase` as your parent class gives you access to quite a few functions called assertions. With assertions, you can assert whether the response from a data source is equal to the data you are expecting. See Listing 8-1.

Listing 8-1. Demonstrated a Single Function and Two Unit Tests

```
import unittest

"""
This is the function we want to test
"""

def is_even(number):
    if number % 2 == 0:
        return True
    return False
```

```

"""
This is our unit test
"""

class TestEven(unittest.TestCase):

    def test_is_even(self):
        self.assertEqual(is_even(2), True)

    def test_is_uneven(self):
        self.assertEqual(is_even(3), False)

if __name__ == '__main__':
    unittest.main()

```

You will not normally have your actual code and unit tests in the same file, but for this example, it works well. Let me explain what happens in the code above.

Anatomy of the Unit Test

On the first line, you import the `unittest` library. Underneath that, you declare a function called `is_even`. This function purely checks if a value is an even number. As a side note, it uses Python’s modulus operator, `%`. This operator does division, but only returns the value to the right of the decimal point. The class declaration after the function is of importance. This class will be instantiated by Python and the unit tests live inside this object. This class extends the `unittest` parent class called `unittest.TestCase`. So your class, `TestEven`, is an instance of `unittest.TestCase`. Each unit test is inside a separate function, and as a rule of thumb, each function should only run one test. The first function in your test class is called `test_is_even`. It is important to note that a test function should start with the word “test.” This is how the `unittest` library detects that it is a function that should be run as a test and allows us to add helper

functions to our class, which will be ignored by the Python interpreter. Inside the function called `test_is_even` is a simple one-liner that starts with a function called `assertEqual` on `self`. You should remember that passing `self` into your function means you have access to all of the class's functions. Because your class is a child of `unittest.TestCase`, you inherit all of the functions inside `unittest.TestCase`, and that is where you get the `assert` function from. There are quite a few assertion functions available for different use cases. In fact, there are two assertion functions with better names you can use to test your boolean values. The reason you pass a hardcoded value of 2 is because you know exactly what the result should be if you pass it a 2. This way you can test exactly what you expect the function to return.

```
self.assertTrue(is_even(2))

    and

self.assertFalse(is_even(3))
```

This is a technicality but is more descriptive. Some other assertion functions that exists are as follows (to name only a few):

```
assertGreater
assertLess
assertGreaterEqual
assertIn
assertNotIn
```

But you do not have to learn them by heart. If you type `self.assert` and then press `Ctrl + spacebar`, you will get a list of all the assertions available.

How to Run the Unit Test

This goes for the integration tests below as well. In the end, you want your unit tests to be automated. There are also multiple ways to achieve this. For your example, mounting the flask-server and running the unit tests from the command line will suffice:

```
docker exec -it flask-server bash
```

```
python test.py
```

Integration Tests

Integration tests are the testing of various components of your system to see how they work together as a more complex unit. So, where a unit test tests one function, a integration test tests how various functions perform together to achieve a goal. Integration tests are better at picking up regression errors than unit tests. A regression error is when something else breaks because components linked to it have changed. Testing your system's connected modules is good at picking up these things. It does not need to be the complete system, but it will test how different components work together (are integrated with each other) to achieve the desired functionality. This can be, for instance, testing how the database-layer components interact with the business logic, or if an email can be sent. I am personally quite a fan of this kind of testing.

You will write two integration tests in Python. This will test your first get and post endpoint, without calling the endpoint directly from outside. After reading this chapter, you should write unit tests for the remaining functions as an exercise.

You will start by completing a few steps to set up your system to be more testable. At this stage, your system is not 100% testable, because you are bound to one single database. When your tests dig into your database layer, it is very important that they do not use the production

database. In fact, in a system that has been set up properly, it should be impossible to run unit tests where the production database is in reach. Your production system should not know about tests. All of the testing and reviewing should happen before your code has reached a production-ready state. It is good practice to remove unit tests when your code gets deployed to production. That is, remove it from the production system, not the development system. Not only do you need to protect the production system, you also want to avoid destroying your own, another developer's, or even your staging system's database by accidentally letting your unit and integration tests drop the database.

Just a quick word on terminology. *Staging* is a separate environment where users can test your system and unit tests can run.

Okay, so you know that you should make sure you do not destroy your databases. You should create a new one. Creating a new database and selecting it for use can be fickle, and the better your system has been set up, the easier it will be. Or rather, it will be easier if you have planned for it. For your unit testing database, you will use a fast in-memory database system called SQLite. This means that the database will be created from scratch in memory, not on disk. Tables will be created and populated with data for your unit tests. After the test has run, this database will technically be gone.

There are many ways to write unit tests and many tools. I selected a rather hands-on method to show you how to write a test, but it gets the point across. Create a file called `test.py`. All your testing code will be in this file.

First, import the `unittest` library, and let your test object inherit from it. The `if` statement at the end of Listing 8-4, `if __name__ == '__main__':`, is a strategy to prevent code from executing if the file gets imported. You will also notice that you are importing `app`. This is the `app.py` file your code is running in, and you need it to access the database criteria. This could have been in another script, but I kept it in there for simplicity.

How to Run the Integration Test

In the end, you want your integration tests to be automated. There are multiple ways to achieve this. For this example, mounting the flask-server and running the unit tests from the command line will suffice. Then see Listing 8-2.

```
docker exec -it flask-server bash
```

```
python test.py
```

Listing 8-2. First part of the unit test

```
import os,sys
sys.path.append(os.path.abspath(os.path.join(os.path.dirname
(__file__), '..')))
basedir = os.path.abspath(os.path.dirname(__file__))
import unittest
import app
import json
from http import HTTPStatus

class TestAPIMethods(unittest.TestCase):
    # Empty for now

# One the same indentation as the class, as this code does not
belong to the class
if __name__ == '__main__':
    unittest.main()
```

The next step is to set up the database. You do this by adding a function called `setup` to the `TestAPIMethods` class, as in Listing 8-3. This must go between the class declaration and the `if __name__...` declaration.

Listing 8-3. Unit test database setup

```
def setUp(self):

    self.db_uri = 'sqlite:/// ' + os.path.join(basedir, 'test.db')
    app.app.config['TESTING'] = True
    app.app.config['WTF_CSRF_ENABLED'] = False
    app.app.config['SQLALCHEMY_DATABASE_URI'] = self.db_uri
    self.app = app.app.test_client()
    app.db.create_all()
    cmd = "DROP TABLE IF EXISTS users"
    result = app.db.engine.execute(cmd)
    cmd = """
        CREATE TABLE users
        (id INTEGER PRIMARY KEY AUTOINCREMENT, name TEXT,
        surname TEXT, identity_number
        INTEGER)"""
    result = app.db.engine.execute(cmd)
    cmd = "insert into users (name, surname, identity_number)
    VALUES ('John', 'Connor', 200509)"
    app.db.engine.execute(cmd)
```

Just under the function declaration, you declare your database connection to the SQLite database. Underneath that you will see `app`. It refers to the `app.py` script. `App.app` refers to the Flask instance you assigned it to.

Why Do We Need the Flask App Instance?

You want to reconfigure the database connection to connect to a different database. To this purpose, you intercept the current database connection and rewrite its database configuration. When you run `init.db(app.app, "test")`, you effectively take the app instance and give it the `TestConfig` file.

Then, you confirm whether the users table already exists, and drop it if it does. After that, you recreate it and populate it with one entry. This allows you to query the test database in your first integration test. This test will look like Listing 8-4.

Listing 8-4. First actual test

```
def test_get(self):
    result = app.get_user_details(1)
    expected = json.dumps({"name": "John", "surname": "Connor",
        "identity_number": 200509,
        "links":
            [{"rel": "self", "resource":
                "http://127.0.0.1:8000/v1/users/1",
                "method": "GET"},
             {"rel": "update", "resource":
                "http://127.0.0.1:8000/v1/users/1",
                "method": "PATCH"},
             {"rel": "update", "resource":
                "http://127.0.0.1:8000/v1/users/1",
                "method": "DELETE"}
            ]}), HTTPStatus.OK
    self.assertEqual(result, expected)
```

After the function declaration, you call the `get_user_details(1)` function on the `app` instance you imported, and on the subsequent line you compare it to the line you entered into the database in the `setup` function. You know to look for id 1. Since you recreate your database for each test, you know that the insert you did in Listing 8-5 can only have a primary key of 1. This is a good example of testing integrated components without actually executing the complete flow of the program.

A Last Issue and Some Refactoring

You are not quite done yet. You have managed to test a really simple aspect of the system, which is to fetch data from the system. What if you want to POST or PATCH data to the system? When you look at the `patch` and `post_user_details` functions, as well as their ORM counterparts, you will notice those functions exhibit what is called *tight coupling*. You will encounter this phrase again later, but in general it refers to separate aspects of the system that are completely dependent on each other but do not need to be. One of the drawbacks is that not only does it make changes to your system more difficult, but it also makes testing more cumbersome. Consider the following function:

```
@app.route('/v1/users', methods=['POST'])
def post_user_details():
    try:
        data = request.get_json()
        sql = text(
            'INSERT INTO users (name, surname, identity_number)
            values (:name, :surname, :identity_number)'
        )
        result = db.engine.execute(sql, data)
        return json.dumps('Added'), HTTPStatus.OK
    except Exception as e:
        return json.dumps('Failed. ' + str(e)), HTTPStatus.
        NOT_FOUND
```

Granted it is not the worst case of tight coupling I have seen. But it is coupled tightly, and it binds the `post_user_details` function to the flask request object, as well as to SQLAlchemy and the json object that does the formatting on the return data. And it is not just that. In order to test this function, you need to fake the `request.get_json()` function's return type. *Faking*, in this sense, is called *test doubles*, and basically means you replace

the value or object that would run in production with another value that will only exist while testing. This function completely relies on an HTTP REST request to get data into the database. To make a long story short, this code is not easily testable, and bad design made it that way. To make it more testable, you will refactor this function somewhat. As an exercise, you can refactor the rest of the code on your own time, taking the steps in this chapter as a guide.

Refactoring this code will not just make the code testable, but will also make it reusable to an extent, as well as easily changeable.

You will create three functions.

The SQL portion, excluding the `db.engine.execute` part of your code, will be replaced by a function called `create`. This technically decouples the SQL insert part from the function called `post_user_details` and makes it testable. In the function called `create`, you pass the posted data through as a parameter. This allows you to do two things:

- Reuse the `create` function where needed.
- Test it without needing to fake the Flask request.

This `create` function does not rely on `request.get_json()` to get its data. It relies on its own `post_data` parameter for data, and it does not care where that data comes from. See Listing 8-5.

You will also create a function called `execute`, which will contain the actual execution of your SQL, `db.engine.execute` part. This function further decouples your code. With this function, you decoupled your `create` function from the actual SQL engine you use. For all of your raw SQL query functions, you should use this `execute` function. With the function called `execute`, you only have to change one function should you choose to change the database, instead of each `db.engine.execute` implementation. This is a very cool feature of your decoupled code. In the previous iteration, `db.engine.execute` was tightly coupled to the REST endpoints, meaning that among other problems, should you have

to change the SQL engine from SQLAlchemy to something else, you only need to change this one function. In other words, the `create()` function does not really care how the `execute` function does its execution, as long as it returns the data requested.

The same goes for the third function you create. The `return_message` function is responsible for sending your response back in a certain format. In a REST call, it is not compulsory to return JSON. You can send back any format. It is just that JSON is a great standard for these tasks. In this case, should you need to change the return format scheme from JSON to let's say, XML, you only need to touch one function. And once again, the `post_user_details` function do not care what format the data gets returned in.

You should now have three more functions, looking like Listing 8-5.

Listing 8-5. Populating the database

```
def create(post_data):
    sql = text(
        'INSERT INTO users (name, surname, identity_number)
        values (:name, :surname, :identity_number)'
    )
    return execute(sql, post_data)

"""
```

The return data here is handy to retrieve meta data from the sql operation, for instance, the id of the last inserted row.

```
"""
```

```
def execute(sql, data):
    return db.engine.execute(
        sql,
        data
    )
```



```
"""
```

```
Returns data in a json format
```

```
"""
```

```
def return_message(message):
    return json.dumps(message), HTTPStatus.OK
```

Your new `post_user_details`, combined with the newly introduced functions in Listing 8-5, now looks like Listing 8-6.

Listing 8-6. New `post_user_details` function

```
def post_user_details():
    try:
        # Get the details from the request object as usual
        data = request.get_json()
        # Call the create function, passing it the data from the
        # request object
        # This object return an object, but we don't need to use
        # it here
        create(data)
        # Return the message 'Added'
        return return_message('Added')
    except Exception as e:
        return json.dumps('Failed. ' + str(e)), HTTPStatus.NOT_
FOUND
```

Testing the New Code

To test the new code, you test the function you created called `create`, and passed in as a parameter is a dictionary object. Remember that `create()` returns an object. When you fetch the `lastrowid` property from it, you get the primary key of the last inserted object done in the specific database

session. You use that id to query the database. The output of this query should be similar to what the input was when you did the test insertion. See Listing 8-7.

Listing 8-7. Testing post user details

```
def test_post(self):
    result = app.create({"name": "Peter", "surname": "Watson",
                        "identity_number": 511247})
    result = app.get_user_details(result.lastrowid);
    expected = json.dumps({"name": "Peter", "surname":
                          "Watson", "identity_number": 511247,
                          "links": [{"rel": "self",
                                      "resource": "http://127.0.0.1:8000/
v1/users/2", "method":
                                      "GET"}],
                          {"rel": "update", "resource":
                          "http://127.0.0.1:8000/v1/users/2",
                          "method":
                          "PATCH"}],
                          {"rel": "update", "resource":
                          "http://127.0.0.1:8000/v1/users/2",
                          "method":
                          "DELETE"}}), HTTPStatus.OK
    self.assertEqual(result, expected)
```

That's it. You have decoupled one of your functions by introducing some more decoupled functions. Function `execute` can be reused wherever you have raw SQL, and `return_message` can be used in all your functions and it now testable.

The Downside of Automated Testing

The Validity of the Tests

You know that you can never say your system is bug-free, and automated tests certainly do not claim your code is bug-free. But with a good testing strategy, you can at least assume that current functionality is as predicted. Testing like this does have potential pitfalls. One of the biggest pitfalls, in my opinion, is the false sense of security it gives you. Here is a prime example showing how unit tests can create a false sense of security, which I encountered a few years ago. Somewhere around 2015, we had a simple piece of code that generated a GUID. A GUID can have different formats, but in general, it is a string consisting of the letters a - f and digits 0 - 9. This is arranged in combinations of 8 characters-4 characters-4 characters-12 characters, like this: 8711a51c-f18c-42c3-8d4b-85d485d5d33d. In any case, because the GUIDs were important to us, we unit tested them. Soon I noticed that the GUIDs were wrong. Instead of groups of characters of 8-4-4-4-12, we got 8-4-4-12. One of the center groups of four was missing. Upon inspecting the code, I saw that the unit test was indeed testing for the incorrect GUID format. I decided to dig a little deeper and looked at the history of the expression that validated the GUID and the code that generated the GUID. It turns out it was created correctly in the first place, but when a bug snuck into the GUID creation, instead of fixing the GUID, the unit test was “fixed.” The developer actually thought that the original GUID was wrong and so they changed it and the unit test. The test was now broken because it tested for a wrong GUID, but it passed unit testing. Even though that did not really have bad consequences, it could have been much worse.

When writing tests, you should focus on edge cases. It does not matter if you write 100 tests with perfect data that cause your unit tests to succeed. In this case, one test is as good as 100, and this tells us very little about the system’s integrity. If you do not test edge cases, such as

how your system's payment module handles payments on leap years, or what happens when a name exceeding the max character length in your database gets sent, then the sense of security in your system will remain false.

Time Pressure

Unit tests take time to write. That is a fact. As a software developer, you will feel the pressure as a deadline looms, and you know you can shave some time off if you neglect your automated tests. If reaching a deadline seems to be in jeopardy, and not writing unit tests can for some reason help you reach the deadline, then I go with a route that, as I mentioned earlier, can cause a false sense of security. I write one test that must succeed per aspect I want to test and one that must fail. This way, if something changes, there is at least some sort of catch net. Having tests are better than having no tests. Then, after the deadline, make sure to schedule a few days to complete the tests.

Unit tests also take time to run. The reason we use in-memory databases is to speed things up, but I have worked in a system where each test took about 2 seconds to run. This was mainly due to the large dataset in the test DB, and due to how things were structured in the DB, it was not possible to change that. We ran about 100 tests. It can feel like ages if you need to run tests for 200 seconds, but in the end, it was worth it.

Peer Reviews

An invaluable tool for code quality is the peer-review process. This is not something you can do if you are working on your own, but if you have a small team, then this is a great tool to get a second set of eyes on your work. Peer reviewing is a process that can go from very informal to very formal. You can peer review your design as well as the software that was

created. I will not go in-depth into the various methods. I will just show what I believe to be the two most wide-spread and common methods. One is called a peer review, and the other is called a walk-through. We will start with the former first.

A very effective method is purely to assign small units of work to a colleague, who will then review the work for coding standards, errors in your code, the effectiveness of your algorithms, whether the code actually works, and above all, did you understand the problem and was the problem solved. This is of paramount importance. The reviewer may not have been present when the work was assigned and may have no real clue of what the software should achieve. The project scope may have been written in an ambiguous fashion, causing the developer and the reviewer to have different opinions as to what the work should have achieved. If the reviewer at any stage doubts whether the code, as perfect as it may be, actually solves the problem it was intended to solve, they should speak up.

I have seen a few times (and indeed I have been in this situation) where work was done and sent for review, and upon receiving the code to review, it became quite clear that the developer and the reviewer had two different opinions of what the feature being developed had to achieve. In one case, the reviewer was right, and the developer had a misconception of what needed to be done. To be fair, their understanding of the system was very similar, but just a small difference in understanding a business rule would have caused the code to not perform what was desired. If the reviewer had not understood the business rule correctly, the review would have passed without an issue, and the problem may only have been detected in production. This could, of course, have had rather bad consequences. So to summarize, the following should be checked during a review, in no particular order.

The developer should be concerned with the following:

- Is there a good description of what the software is to achieve?
- Is it a small review? If you assign a review of 1000 lines to a colleague, you can expect to get a review that was not 100% thorough.
- Have you made any install or system update changes that the reviewer should know of? Are there database changes the reviewer should know of?
- Depending on how new the system is, it may also require some usage instructions. How do you actually use the new feature?

The reviewer should be concerned with the following:

- Coding styles
- Have automated tests been written?
- Are the algorithms optimal and void of any obvious and serious errors?
- Are there any weak points where an intruder can enter the system?
- Is the code being reviewed solving the problem that the business has?

Walk-Through

A walk-through is a bit more formal. In general, you schedule a meeting with specific stakeholders in a meeting room. The stakeholders should be given relevant data with relation to the work you are going to show them.

You go through the software product or feature on a step-by-step basis, allowing the stakeholders to critique it and make recommendations, while you take notes.

Staging Environment and UAT

This is the final stage before you can release your code to production. The code has been written, and unit tests have been run to detect regression errors and errors in general. After that, the code has been reviewed. Now you are ready to move the code to the staging server. The staging server is an environment that accurately mimics the production server. It is accessible by the people interested in the feature that has just been developed. It will also run against a staging database, where data can get edited and deleted without worry, so that the stakeholders can get the full experience of the software. On the staging server, you can demonstrate your code to stakeholders, and above all, ask the stakeholders to test the system. Once it has been confirmed that the code meets the stakeholders' standards, it can be released to production. The stakeholders can be employees in the company you work for or the product owner. A stakeholder is anyone with an interest in the feature being developed. But the best people to test are the actual users of that specific feature. One informal rule is that the software developer should not do the user acceptance test (UAT). I mentioned that UATs are tests conducted by the actual users of the system, to see if it does what is required of it. Software developers are infamously bad at testing their own code and features they created. User acceptance tests are great and should be mandatory, but in most cases will not pick up on regression errors, unless the user testing the system stumbles upon them.

That's it. You now know some simple steps to increase confidence in your codebase, especially pre-release. Each of these steps can fill a few chapters, and your knowledge about these topics will grow as you encounter them in your career.

CHAPTER 9

Planning and Designing Your Code

Knowing the tools of the trade is half the battle won. We have spent a few chapters looking at the technological tools you can use to create software. But tools do not make the software great. They do, however, make the software development process great. You can still, if you want to, do everything you have read in this book so far using Notepad and FTP. Instead of Docker, you can run MySQL locally on your machine and Python as well. Instead of Git, you can just make clumsy backups called `main.py`. 2020-01-15. Instead of running unit tests automatically, you can even choose to run some tests manually. But there is one thing you cannot compromise on, and that is the design of your code. In this chapter, you are going to look at a basic system development life cycle, and after that, some design tips and tricks. The processes feed well into each other, from the flow of project execution, to high-level design, to lower level design.

The aim of this chapter is to introduce the concepts of designing your code and having set steps to develop your code. I will not delve too deep into the system development lifecycle but will show basic steps in separating the actions you need to take to get your software from planning to production.

After the software development lifecycle, you will look at designing aspects and how to create abstract models. The first will be on a high level, where you will plan a portion of your executable code, and the second will take a more granular look at the creation patterns of your objects.

Software Development Lifecycle

In this section, you will look at the software development lifecycle (SDLC). I will use a basic approach to the development lifecycle. The idea of this section is to provide you with a generalized idea of the software development lifecycle that you can use when you create your own projects.

Why Use a Software Development Lifecycle?

The software development lifecycle enables us to focus and isolate the different phases of the software creation process. These phases will guide the software development process and allow us to complete our software in a sequence of steps. Some of these phases can be revisited during the process, as errors in understanding and errors in judgement are encountered. This is fine. But it is important to notice that only certain phases should be revisited. It is important to know that the phases feed into each other. The work in phase one will be the basis for the work in phase two. The work in phase two will form the basis for the work in phase three, and so on. Within each phase, there is a clear distinction of what tasks need to be done. For instance, if you are not in a development phase, you will not do any development. You will only focus on the work to be done in each phase.

The aim of the software development lifecycle is to reduce risks and costs by streamlining and formalizing the development process. It gives us an indication of where we are in the creation of our software and may help us calculate a time when we can potentially finish the software. By following a specific sequence of steps, all based on the initial specifications provided by the user, we can also control the features being added to the system for that specific development task and prevent features being requested that were not in the original design. This is a problem called “scope creep” and it is a very real, and very human, problem in software engineering. Scope creep adds to the total design time, and in the end, the total cost of a product.

The Reality of Scope Creep and Not Pinning Down Requirements

Many years ago, I decided to try my hand at some contract work. I found a contract that sounded incredibly simple. It was just a template-based emailing system that emailed the client's contacts with deals and specials. This was hardly a lot of work, so I quoted based on what was told to me over the telephone and started working on it. It soon transpired that my client himself did not know what he wanted, as more and more features were added. A week's worth of part-time work after hours turned into a month and then two months. In my naivety, I kept on building the software on the original quote and never adjusted the timeline either. I realized he intended spamming people when he gave me a list of many thousands of people and asked me to write an importer where he could just import his list and send emails from that list. Upon questioning the size of the list with his apparent client base, he said he found the list on the Internet. I didn't want to be part of spamming people, so we parted ways on a not-so-amicable footing. His argument was that he was doing the spamming, not me, whereas I felt I was the enabler. I lost two months' worth of time, plus getting no payment, and I have to say, it was my own fault. I did not pin the requirements down and after accepting the changes I did not adjust the quote.

What happened to me, in my first attempt to do contract work, can happen in the biggest corporation or the smallest one-man show. You run this risk whenever you are busy with a project.

Steps in the SDLC

As mentioned, we will show a generalized approach to the SDLC. The first thing to mention is that the word "lifecycle" in "software development lifecycle" is exactly that, a cycle. The process is never really complete. Even after you think you are done, you are not, because the last phase

of the process is to support the system you created. And that can last as long as the software is in use or you are available to support it. During the support phase, new features will be requested, and each feature will basically start its own SDLC and branch back into the original SDLC during the support phase. Your software will never be completed. If you stop supporting it, it will just grow old and eventually be unsupported by modern technology. Vendors like PHP and Python do not intend to keep backwards compatibility with software you wrote 10 years ago. It is just not feasible. Eventually enhancements in your own tech stack will fail your software. Your software will become redundant. This is indeed a sad end for the software you were so proud of at one stage.

On a lighter note, here are the phases of a common software development lifecycle.

Phase One: Planning

This phase, planning, includes aspects such as scheduling the project, organizing the teams that will work on it, doing a cost analysis and a feasibility analysis, and so on. It is not unheard of that this phase excludes the software engineering teams.

Phase Two: Requirements

This phase, requirements, solves the problem of gathering requirements from the stakeholders. The desired output of this phase is units of work, which will be assigned to what is called “the backlog.” This phase will generate a large amount of units of work, where each unit is a small task that needs to be completed in order to eventually complete the project. A unit of work may, for instance, be an endpoint that creates a user in the system. It must be a small quantifiable unit of work that can be measured and completed in a short time, normally within two days.

Phase Three: Design

The output of the design phase is what we will discuss in the rest of this chapter. The design phase is where system specifications are created. In this phase, you take the requirements and transform them into designs. These designs will be graphical, such as UML diagrams, or even in a human-readable format, like a Word document. Sometimes they may even be something more abstract explaining the concept from a high level. There is no real rule as to how the specifications should be created, or what technology they should be created in, as long as they are understandable and not ambiguous.

Phase Four: Development

The development phase starts the software creation process, based on the specifications from phase three and the work units from phase two. The work units from phase two will be assigned to you, and the designs from phase three will guide you in completing the work. This phase will in all possibility also deliver graphical designs, as the developers may create more granular designs of the components, and more granular process flows, in order to complete the process. This phase will last quite a while and run as a series of iterations, where each iteration has a duration as well as a fixed amount of work units per developer to complete. For instance, one iteration may last two weeks. Each developer in that iteration will be assigned the amount of work that they should be able to finish in two weeks. Once that iteration is done, another is started, and more work is assigned to each developer to complete. These iterations are referred to as *sprints*.

Phase Five: Testing (But Not the Sole Testing Phase)

This phase focuses on testing. Even though it follows the development phase, it is not to be left for that one day when all development is done. In reality, your code will be tested quite thoroughly inside a sprint, using

methods such as unit tests and peer reviews. There will be a separate testing phase before a big release, but no sprint, as discussed in phase four, should have untested code. After a sprint, and after you have tested the code yourself, written unit tests, and sent your code for a peer review, you can get the relevant stakeholders together for user acceptance testing. User acceptance testing will in general happen only after the developer has completed the work.

Phase Six: Deployment

A single large deployment mainly happens for a brand-new software project. Once the software has reached a level of functional maturity, which is normally quite quickly, the release process becomes a lot more regular, often every two weeks, or if your company believes in fast releases, whenever a work unit is done and tested.

Phase Seven: Support and Maintenance

This phase will kick off numerous mini SDLCs whenever a bug is encountered or a new feature is needed or even when the vendor software needs upgrading. This phase includes improvements, bug fixes, new features, and in general anything that keeps the software ticking. From this phase, numerous mini SDLCs will be created.

When you look at these phases, it is easy for a beginner to get confused. The SDLC has seven phases. However, often you will only need phase two (unless the work was grabbed from the backlog, which means the requirements have already been done, so you don't need this phase), three, four, five, and six. And within that, phase four can happen many times, with software developers deploying many batches of production-ready code. After that, they all kind of merge into phase seven, where phase seven is almost the de facto state your software is in. Above all, you may not even be aware that phase one even happened, as in some cases the developers are only involved from phase two. This is a lot less chaotic than it sounds.

In reality, when you have finished, tested, and deployed a unit, or a group of units of work, you have implicitly created a mini SDLC. You should still stick to the phase sequence. Do not code if there is no design. Do not deploy if you have not tested thoroughly. Do not see your mini-SDLC as ended if you have not deployed. In other words, do not take on more work. Actually, the latter may be unavoidable, and good judgement is needed for that decision. Not releasing code is something to avoid, as you cannot merge your undeployed branch into the main branch that everyone else is branching new work from. Not releasing code will cause your branches to become out of date if you do not keep them up to date regularly. But even if you keep your undeployed branch up to date, all subsequent work done by all the other developers will exclude the work you have done that is not getting deployed, and this can become a problem after a while.

There are quite a few competing SDLCs, but the steps are more than sufficient to keep your software projects from going astray, if you keep to them. The truth is that some companies follow an amended variation of the software development lifecycle. There are no silver bullets out there that will solve your problems out of the box. When we looked at code calisthenics and coding standards, I mentioned that they should only be applied where it makes sense. The same goes for an SDLC. If it does not make sense to have two phases where you can just have one phase, then you should not have two phases just to conform to the rules. The lifecycle should be optimized to suit your needs. The best approach is to think critically about it and not include aspects that you do not need.

Modelling

Where Does Modelling Fit In the SDLC?

Modelling is, in general, done in two phases in the SDLC: the main design phase, which is phase three, and the development phase, which is phase four. I have worked in companies where the design phase and the development

phase are the same phase. As mentioned, no process fits any company perfectly. In most cases, there will be some variation to the lifecycle but based on an industry accepted standard, and this is fine. It may be necessary to create complete models and diagrams before any work has started. This is a great idea, especially in a brand-new project. However, in subsequent iterations a few months down the line, those models and diagrams may become insufficient as a sole source of system knowledge to complete a new feature. New models or amendments to the old models may be required. Then there are two ways of looking at the problem. In a lot of cases, if the feature being developed is a two- or three-day task, modelling will happen inside a sprint, and as you may remember, sprints happen in phase four, the development phase. However, the task may be so big or so complex that you have to go back to phase two, where you first gather requirements and create the units of work, and then on to phase three where you start your design and create your models, diagrams, and documentation, and then on to phase four where development happens. And once again, more granular diagrams may be created in the development phase, phase four.

Why Create Diagrams and Models?

Mostly you won't create a diagram for the complete system. You will create multiple small diagrams which combined will form the complete system. To keep things readable, this is the best approach. A lot of times, software developers will listen to the requirements of the software they need to write and then just start coding. It is very easy to paint yourself into a corner doing this. Why do some developers proceed to solve a problem in this shoot-from-the-hip fashion? They may feel that they understand the requirements well enough to proceed without creating a visual model. They may have already formed a solution in their heads and trust in their abilities enough to execute that specific plan. To be honest, sometimes this works. The problem is not just that it does not always work well; it is that it also does not always fit into the current system properly or does not allow for potential future

expansion of the system without a refactor. Modelling gives you a change to verify your solution and your design. It gives you some time to really think about what you are about to create, how the program will be executed, and how your components will interact with each other.

Modelling also serves another purpose. It provides documentation. When you leave your current position, or even when you look at code you wrote 12 months ago, you need a reminder of what you did and what your thought processes was when the software was written.

So you create models and diagrams to enable you to think critically about the problem you want to solve. They guide you through the development process, and down the line, provide you with documentation relating to how you solved the problem.

Tools

You will use draw.io to draw your diagrams. You can find it at <https://draw.io>. It may prompt you to download a desktop application, which is advisable.

High-Level Models and Diagrams

High-level models and diagrams are used to explain the system at a non-granular level; that is, no actual classes get designed. This gives you an overview of the system or a portion of the system, how data will flow through it, and the execution paths. A range of diagrams exist for this, and they all have their different pros and cons. The high-level diagrams you will be looking at are the following:

- Activity diagram
- Use cases
- Swimlanes
- Database diagrams

You will use UML to create your diagrams. UML stands for Unified Modelling Language.

Activity Diagram

Activity diagrams are not very complex to create and are probably the most used diagram due to their ease of use. They have a limited set of shapes that are used to model the diagram, but those shapes make them quite powerful. They can show splits in the path of execution logic, basic decision-making structures where the system should make a decision and choose one of multiple paths to execute, iterative structures where something should happen multiple times, as well concurrent execution. I am wary of describing decision-making structures as `if` statements and iterative structures as loops, since this is still a high-level overview, and programming terms should not really come into play. But in general, those structures can be seen as `if` statements and for loops.

The following symbols are used in this diagram:

- The start of the flow is a black dot.
- Ellipses are actions being taken.
- Diamonds are decisions.
- Horizontal black bars represent the splitting or the joining of activities that run concurrently.
- Arrows link the actions.
- An arrow pointing back to a previous execution step represents iterations.
- A black dot within a circle represents the end of the flow.

Let's model a card payment system. In order to pay, you need to have the correct PIN, and you can retry upon entering the incorrect PIN. (We will not dwell on maximum amount of retries for this diagram.) You also need enough money in your account, and you can't exceed your spend limit for the

day. If you do not pass any of the last checks, you can select a new payment card. This description can be modelled as shown in Figure 9-1. The square speech bubbles are not part of the diagram; they are merely my comments.

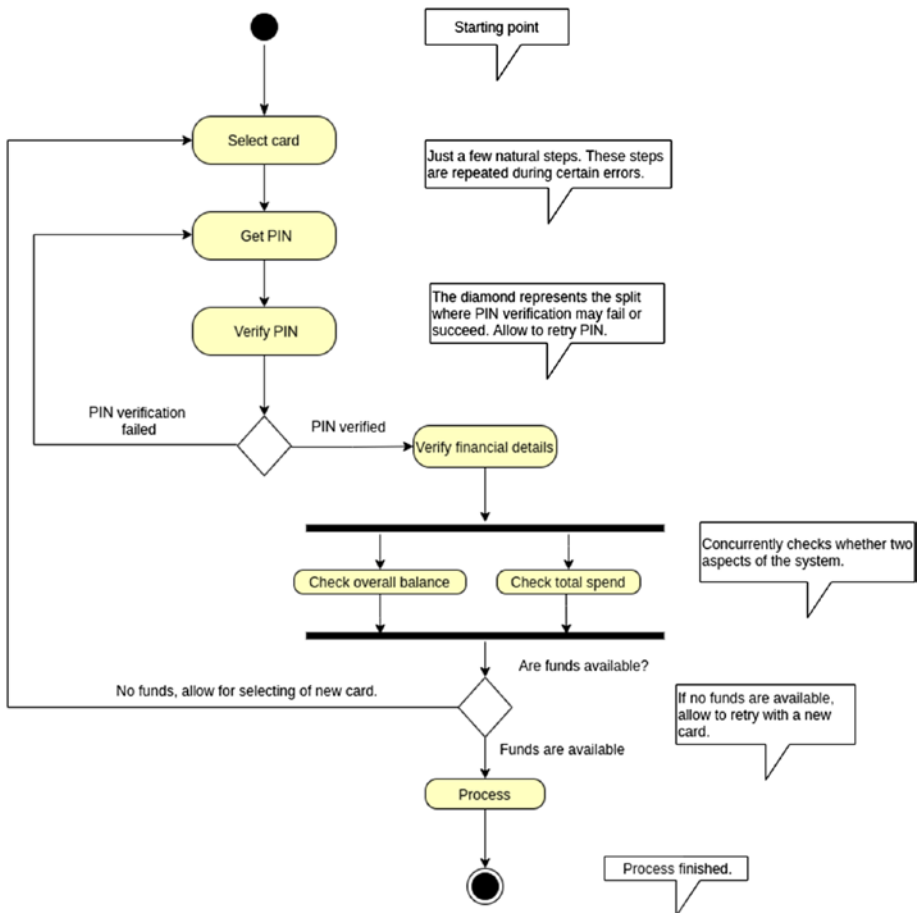


Figure 9-1. Activity diagram

Here you can see the starting point for this diagram is a black circle. A user will be able to select a card to pay with. After the card selection, the user will provide a PIN number. If the PIN is wrong, the user will be taken back to enter a PIN. If the PIN is correct, you verify that the user has the correct

amount of funds in their account. These checks are done concurrently, hence the two black bars below and beneath the balance and total spend check. If no funds are available, the user can select another card; otherwise, you process the funds, after which the process is seen as done.

Granted, these steps are inadequate for a real-life transaction processing process, but they serve pretty well to get the idea across. You will not have one diagram detailing a complete system. Instead, you will have multiple complete and detailed diagrams that each cover a specific portion of the system.

Use Case Diagram

A use case models the relationships and actions a user (called an actor) can perform on a system. It categorizes the events that can happen and does not delve into how those events are actually solved at runtime. They are quite handy for pinning down system requirements as well but offer a great overview of the feature you are looking at. They can become quite complex, drilling down deeply into what the system should achieve. Personally, I like to keep them simple, giving me and non-technical people the ability to easily understand the system from a high-level overview. I have seen how non-technical people quickly perceive how I understand their tasks with a simple use case diagram I drew of their tasks. Back to the diagram. An actor does not need to be a human. It can, for instance, be an external automated system that communicates with your system. A rule of thumb is to stick to the explicit problem you are trying to solve. If you want to show how a user can send an email on the system, do not show how the user can log in as well. Logging in can be a use case on its own. Show one main set or flow of actions and the users and contributors to those actions. You will once again have multiple sets of use cases, instead of one set explaining the complete system. Say you have finance clerk, and you need to model their current job in order to build it into the a new system. This is only a small portion of their tasks, but that is the way you want to model it. Each task that can be separated from the others will get its own use case.

You can provide a verbal description as well, such as “as a financial manager, I need to take the CSV file of payments and upload it into the system. I then confirm all the entries are in and run a fraud check, and if all looks well, I flag the entries as payable. After that, it can be authorized for payment, and the accountant will release the funds. This is modelled in Figure 9-2.

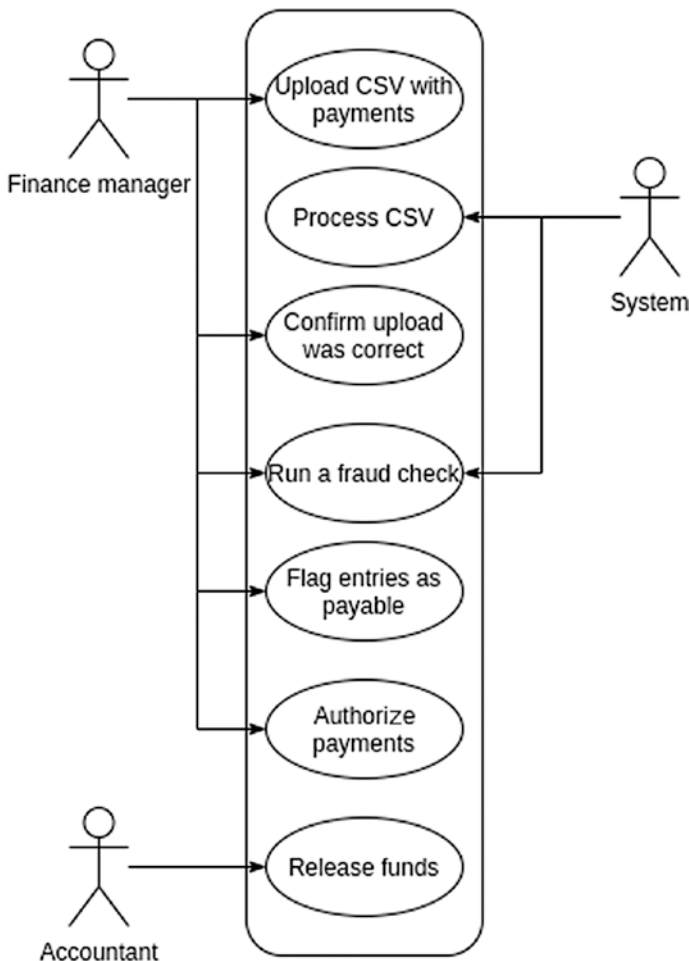


Figure 9-2. Use case diagram

As you can see in Figure 9-2, a glance at the diagram is easier to digest than even that short description, although having both is always a win.

Figure 9-3 is another example of a use case, and in this instance, what your use case should *not* look like. Notice that there are two issues in one use case.

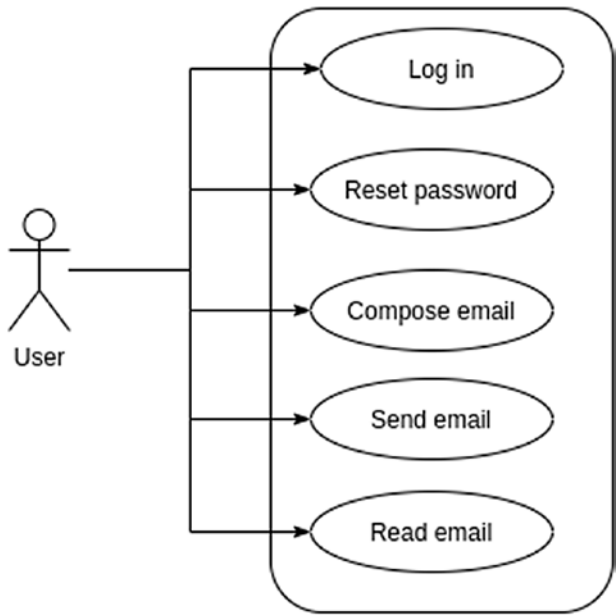


Figure 9-3. *Incorrect use case diagram*

In the above use case, logging in and resetting password should not be present because they are not part of this specific flow of actions to complete a task.

Swimlanes

Swimlanes are handy tools that show clear task separation between the entities that do them, as well as how they interact with each other and what the end result of such an interaction should be. Swimlanes can show

inefficiencies in existing systems, which is very handy when you need to rebuild a system to be more efficient. They also give a really good overview of how system aspects work together. In swimlanes, each lane has a clear division and task, and each lane belongs to an entity, where that entity can be a human, a complete system, or a subsystem. There is a bit more to the symbols of swimlanes, but the most commonly used ones are shown in Figure 9-4.

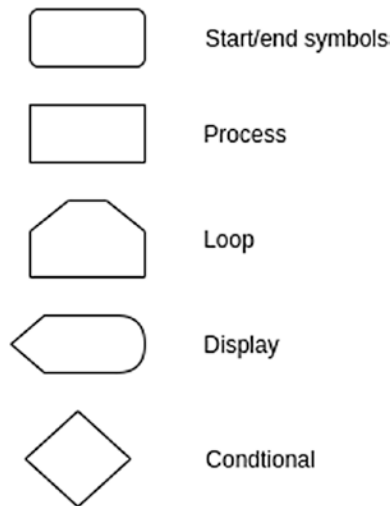


Figure 9-4. *Swimlane symbols*

Figure 9-5 shows the execution flow of the finance manager’s CSV upload in order.

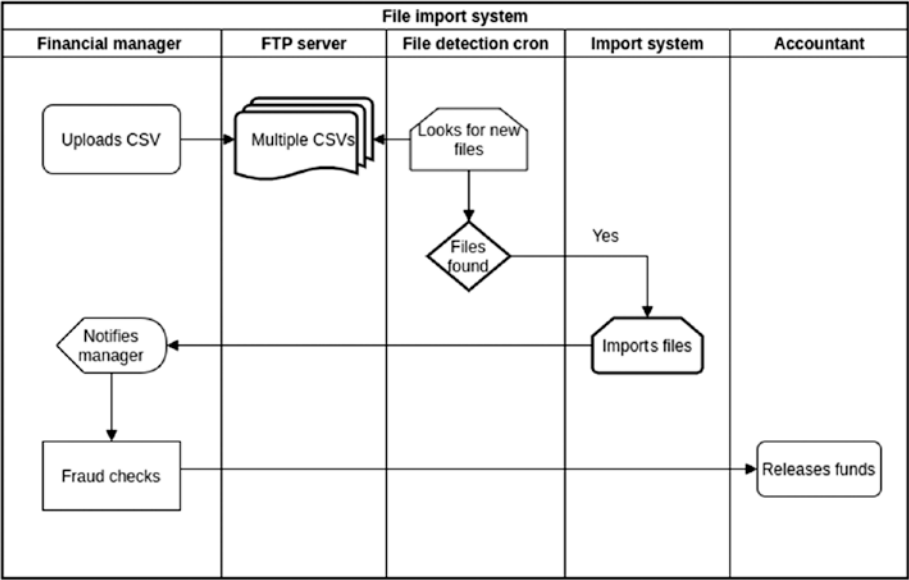


Figure 9-5. *Swimlanes*

In Figure 9-5, you can see how the execution starts with the finance manager uploading the CSV file to the FTP server. A cron checks for new files at specific intervals. If no files are found, then the system state does not change at all. If a file is found, it gets imported into the system, after which the manager is notified. The manager does some fraud checks, upon which the accountant is notified to release the funds. The releasing of the funds signals the end of the process.

It is quite easy to follow the flow of execution, and above all, the swimlanes provide a bit of an overview of business processes governing the system. There is still more to swimlanes than meets the eye, but this should set you up to be able to create your own swimlanes.

Database Diagrams

Database diagrams are quite straightforward. They model the relationships between datasets. Database diagrams are also called ERDs, or entity relationship diagrams. There are few sets of notations in ERDs, such as Chen notation, crow's feet notation, and UML notation. We will look at UML notation. I have encountered crow's feet notation a few times in my life, but in the end, all other database diagrams I have encountered use UML notation. To be honest, I prefer it as well. You can immediately read the relationships between the data tables without having to pause for a second to try to remember what the symbols mean. UML relational notation is very readable. In an ERD diagram, there are boxes connected with lines. These boxes represent tables, which basically represent an entity in your software. Each box has a table name, columns names, primary keys, and potential foreign keys. Between the data that your boxes represent are associated relationships. For instance, a user can have one address but many telephone numbers. The format of the box is arbitrary and depends on what modelling tool you use, but it can look like Figure 9-6. There is a field for tablename, primary key, and different field names and their types.

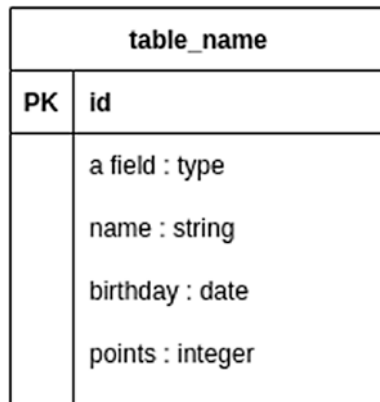


Figure 9-6. Table diagram

Modelling Relationships

A single line, with nothing else on it, in general means a one-to-one relationship. But you can also put a 1 on the actual line to indicate a mandatory one-to-one relationship. Figure 9-7 shows a one-to-one relationship. Figure 9-8 shows a one-to-zero-or-many relationship, meaning a user may have zero or many addresses. You can also use 0..1 to indicate a zero-or-one relationship and 1..* to indicate a one-or-many relationship between users and addresses.

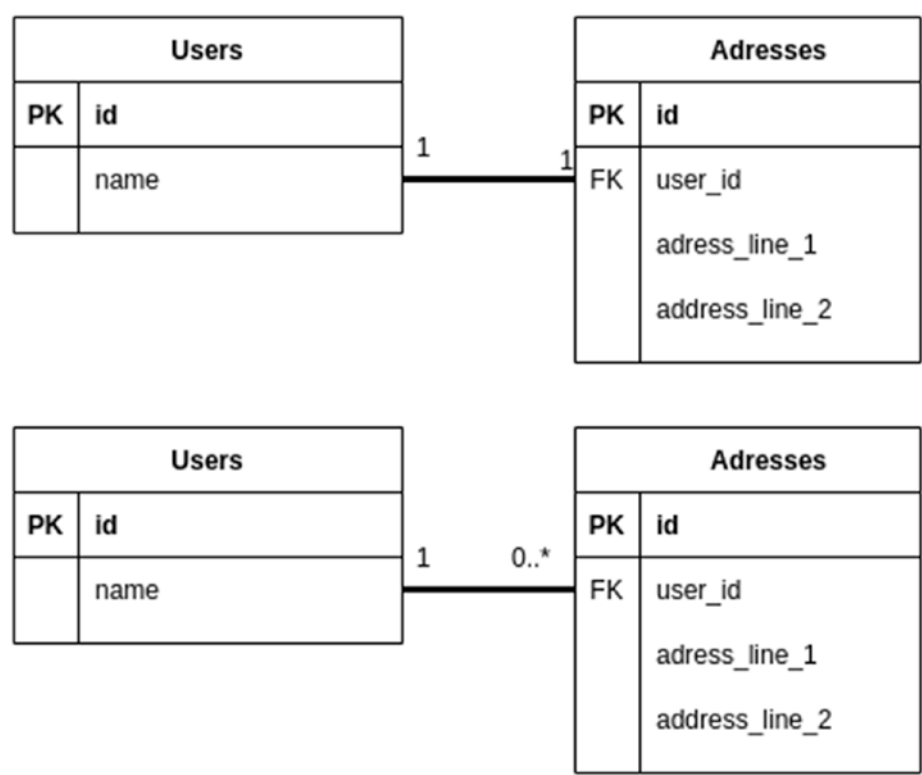


Figure 9-7. ERD

That is already very understandable. Figure 9-8 shows what a many-to-many relationship looks like.

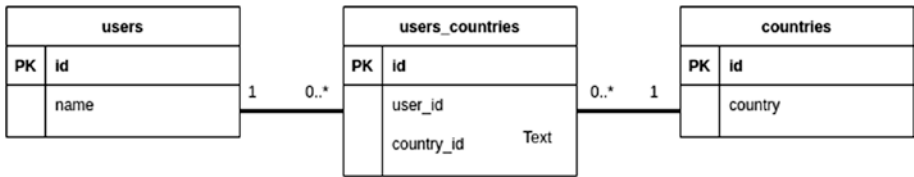


Figure 9-8. *Many to many ERD*

From the left, you have one user, which can be linked to zero or more countries, and countries can be linked to zero or more users.

You should be quite confident in modelling a database. Normalizing a database is the difficult part. Modelling the db is the easy part.

Low-Level Models

In this section, you will have a look at the more granular creation of components that make up your system. You will look at some tools you can use to create classes that will perform well under scaling and aid in understandability. The topics you will look at are the following:

- Types of objects (value and entity, etc.)
- Cohesion and coupling
- SOLID principles for object design
- DRY principles for code reuse
- Composition

Types of Objects

I will discuss two types of objects that you will create, a value object and an entity object. The first difference between the two is that a value object cannot operate on its own but is part of an entity object. A value object represents something that does not have an identity in the system but is part of something that does have an identity, like a User. It does, however,

encompass its own rules, methods, and attributes. An example of a value object is a password field. The password field can be its own class with its own password validation methods. This class is part of a bigger entity class called `User`. A `User` has an identity and is represented in the database. Value objects cannot (should not, because technically you can) be changed once they have been created, as this can lead to potential bugs. Entities can have their attributes changed without a problem. To come back to the password example, normally you would be tempted to declare the field `password` as a string within `User`, and all the validation methods, whether creating a password or validating a password, live in the `User` entity. By making `password` a value object, you relieve the `User` of the task of password validation, but you still have `password` as an attribute of `User`. `User` is still responsible for writing a password to the data table upon user creation but not responsible for password validation when the password gets created. This improves the cohesion of your code, a concept you will explore a bit later.

So, in short, try to distinguish between different classes: those with an identity inside the system, like a `Vehicle`, and those that do not have identity, like `Vehicle`'s `mileage`. Instead of having `mileage` as a float attribute inside `Vehicle`, and leaving `Vehicle` with the responsibility of calculating distance between `mileage` entries or whether the `mileage` is appropriate, let a `mileage` value object do it. You can also create, for instance, a `Service` class as a value object, which tells you when it is time to go for a service. This object can take a `mileage` object as parameter when it needs to see if it is time for a service. See Listing 9-1.

Listing 9-1. Mileage and vehicle class

```
class Mileage:

    def __init__(self, previous_miles: int, current_miles: int):
        if current_miles < 0 or current_miles < previous_miles
            or previous_miles < 0:
```

```

        raise Exception('Incorrect miles entered.')
    self.current_miles = current_miles
    self.previous_miles = previous_miles

    def miles_current(self):
        return self.current_miles

    def miles_previous(self):
        return self.previous_miles

class Service:

    SERVICE_MILES = 1000

    def __init__(self, mileage: Mileage):
        self.mileage = mileage

    def must_service(self):
        return self.mileage.miles_current() - self.mileage.
            miles_previous() > self.SERVICE_MILES

class Vehicle:

    def __init__(self, vehicle_type: str, numberplate: str,
        mileage: Mileage):
        self.vehicle_type = vehicle_type
        self.numberplate = numberplate
        self.mileage = mileage

    def time_for_service(self):
        service = Service(self.mileage)
        return service.must_service()

mileage = Mileage(1000, 2900)
vehicle = Vehicle('Ford', 'CAM123456', mileage)

print(vehicle.time_for_service())

```

In Listing 9-1, you can see your Value object, called `Vehicle`. `Vehicle` needs to see whether it is due for a service, but uses two value objects to do so, `Mileage` and `Service`. With this design, you take the logic to calculate mileage and service notifications out of `Vehicle`, so that `Vehicle` can concern itself only with what it really needs to do. This keeps the objects small and reusable.

Cohesion and Coupling

Coupling and cohesion are two aspects that address how interlinked classes are and to what degree a class does one thing. Let's look at coupling first.

Coupling

Coupling refers to how much your classes explicitly depend on each other. What do we mean when we say classes are dependent on each other? Surely that is why we create classes, so that they can collaborate and share data, and call methods to perform tasks? Well, the coupling concept refers to how much classes are dependent on the physical presence of very specific object instances. It can also refer to how many other objects your class needs in order to function. If your class needs 15 other classes, it is possible that your class is doing too much, or that you went overboard on creating classes, and that you have spread the task of one class over two, three, or potentially even more other classes. There are two types of coupling, loose coupling and tight coupling. Loose coupling refers to a low dependency on each other, and tight coupling refers to a high level of dependency. In short, loose coupling is good; tight coupling is, well, less good.

Fortunately, I have two examples at hand to demonstrate this. Look at Listing 9-1. Within the `time_for_service` function, you are tightly coupling the `Service` class to the `Vehicle` class. The `Vehicle` class is also now responsible for creating a `Service` object, something that is certainly not its job.

The second issue is that you have a `Mileage` class and a `Service` class. Although it is good to have these value classes, looking at this implementation, the class called `Service` does nothing but use `Mileage`'s function to achieve its goals. It knows too much about `Mileage`. In fact, as mentioned, without `Mileage`, the `Service` class would be useless. So now you have three classes, tightly coupled.

One of the solutions you can consider is shown in Listing 9-2.

Listing 9-2. Refactored mileage and vehicle class

```
class AbstractService:

    SERVICE_MILES = 0
    TYPE = ''

    def __init__(self, previous_miles, current_miles):
        if current_miles < 0 or current_miles < previous_miles
        or previous_miles < 0:
            raise Exception('Incorrect miles entered.')
        self.previous_miles = previous_miles
        self.current_miles = current_miles

    def must_service(self):
        print('Calculating ' + self.TYPE + ' service mileage')
        return self.current_miles - self.previous_miles > self.
            SERVICE_MILES

class Service(AbstractService):

    SERVICE_MILES = 1000
    TYPE = 'normal'
```

```

class MajorService(AbstractService):

    SERVICE_MILES = 10000
    TYPE = 'major'

class Vehicle:

    def __init__(self, vehicle_type: str, numberplate: str):
        self.vehicle_type = vehicle_type
        self.numberplate = numberplate

    def time_for_service(self, serviceObject: AbstractService):
        return serviceObject.must_service()

vehicle = Vehicle('Ford', 'CAM123456')

print(vehicle.time_for_service(MajorService(1000, 20000)))
print(vehicle.time_for_service(Service(400, 453)))

```

So, what's changed? First of all, you removed Mileage and all references to it. You will only use the Service class. Mileage is not needed because you are not interested in mileage at all. The Service class can handle all of the calculations and input rules to calculate the service period. In order to display how you decoupled the Service class from Vehicle, you made quite a few changes. Secondly, you added another Service class called MajorService. MajorService is exactly the same as Service and only differs in how it calculates its service period. Granted, it is not very impressive in its differences, but it should be sufficient to demonstrate how it works. After that you added an abstract class called AbstractService and added a constructor to it. Service and MajorService should extend this AbstractService class. They are both now of type AbstractService and they both overwrite the attributes TYPE and SERVICE_MILES. The function `def time_for_service(self, serviceObject: AbstractService)` now takes an object of type

`AbstractService` as a parameter. With this change, you can pass a `Service` or `MemberService` object to the `time_for_service` function. Should you think of other `Service` types, it's easy to add them. The objects are also created outside the `Vehicle` object, which further decouples it. Because you got rid of the `Mileage` class, you needed a new way to get miles into the `Service` object. You added this ability into the `Abstract` class's constructor.

A last word about coupling. Coupling will always exist. The goal is not to get rid of coupling, just to keep it as low as possible.

Cohesion

Cohesion is a rather important topic. It refers to the degree to which a module's internal components are related, or in simpler terms, they do things that are not really related to each other. It keeps a class's responsibilities focused on one thing. For instance, if you have a class that represents a user's shopping basket, then that object should not represent the user's login function as well. You should have a `User` object and a `ShoppingBasket` object. You can also inspect your classes and see whether all the functions make use of the attributes inside the class. You should not really need any functions that do not touch any of the attributes inside your class. Maybe that function should move? Another pattern that we are fortunately seeing less and less nowadays is the "helper" class. This was somewhat popular a while back. It consists of an object with a bunch of functions that "helped" you achieve a variety of goals. It was a mismatch of functions that could, for instance, strip spaces, do searches, flatten arrays, you name it. This should also be avoided because it points towards a lack of planning of the architecture.

SOLID

SOLID is an acronym for five principles that when applied to your software make for better design. They are listed below. Some of them should be familiar to you.

Single responsibility principle

Open/close principle

Liskov substitution principle

Interface segregation principle

Dependency inversion principle

Single Responsibility

We already discussed single responsibility, and it's the same as cohesion. It further states that a function or class should have only one reason to change. This is easy enough if your object or function does only one thing.

Open/Close Principle

The open/close principle states that an object should be open to be extended and closed for modification. Listing 9-2 shows code that displays the open/close principle. Initially, you had one `Service` class. When you decided that `Service` may have two behaviors, you could have just shoved the new behavior into the `Service` object and made it look roughly like Listing 9-3.

Listing 9-3. Open/Close principle

```
class Service:
```

```
    SERVICE_MILES = 1000
```

```
    MAJOR_SERVICE_MILES = 10000
```

```

def must_service(self, serviceType):
    print('Calculating normal service mileage')
    if serviceType == 'normal':
        return self.current_miles - self.previous_miles >
            self.SERVICE_MILES

    return return self.current_miles - self.previous_miles >
        self.MAJOR_SERVICE_MILES

```

This may look innocent enough, and you may even think but it looks better! It certainly looks like less code. But remember, less code is not always better. Here you have a function that not only does two things by calculating normal services and major services, but if you want to add another type of service, for instance some vehicles need a minor first service, then you need to add it here as well, meaning the function is open for modification, which is bad, plus it increases this function's complexity and this function now does three things. In Listing 9-4, you can see how adding more aspects to it just creates more complexity.

Listing 9-4. Refactored open/close principle

```

class Service:

    MINOR_SERVICE_MILES = 1000
    NORMAL_SERVICE_MILES = 1000
    MAJOR_SERVICE_MILES = 10000

    def must_service(self, serviceType):
        print('Calculating normal service mileage')
        if serviceType == 'normal':
            return self.current_miles - self.previous_miles >
                self.NORMAL_SERVICE_MILES
        elif serviceType == 'minor':

```

```

        return return self.current_miles -
            self.previous_miles > self.MINOR_SERVICE_MILES
    else
        return return self.current_miles -
            self.previous_miles > self.MAJOR_SERVICE_MILES

```

To adhere to the open/close principle, going back to Listing 9-2, you created a parent class, `AbstractService`. The goal of `AbstractService` is to provide an extendable parent class that is open for extending but closed for modification. To add another Service type, let's call it `MinorService`, you just need to overwrite the `TYPE` and `SERVICE_MILES` attributes, and extend `AbstractService`, like so:

```

class MinorService(AbstractService):

    SERVICE_MILES = 500
    TYPE = 'minor'

```

This layout also makes it the calling code's responsibility which Service gets called, and not the service class itself.

Liskov Substitution Principle

The Liskov substitution principle states that if you have a parent-child class relationship, then you should be able to swap the two with each other without breaking the system. What do we mean by breaking the system? Obviously, when you switch the objects out, your output will be affected because the two objects have differences in their internal algorithms. We do not see this as breaking the code. Breaking in this instance means the software ceased to execute properly, either by breaking or by incorrectly taking a completely wrong path in the system. Look at the code in Listing 9-5. This code can be added to the code in Listing 9-2 for execution purposes.

Listing 9-5. Liskov substitution principle

```

class MinorService(AbstractService):

    SERVICE_MILES = 100
    TYPE = 'major'

    def must_service(self):
        print('Calculating ' + self.TYPE + ' service mileage')
        return str(self.current_miles - self.previous_miles >
                    self.SERVICE_MILES)

if vehicle.time_for_service(MinorService(1000, 20000)) == True:
    print('You are due for a minor service')

```

In this snippet, you extend `AbstractService`, but you decide to override the `must_service` method. This is fine. But look at the line where you return from the `must_service` function. Instead of sending back a boolean `True`, you send back a string and cast the result to a string right before returning. Since all of your child classes need to be replaceable by the parent class without causing the system to break, this will actually violate the Liskov substitution principle. Look at the last two line of Listing 9-5. They are doing a boolean comparison. Because you are now sending back a string instead of a boolean, your “are you due for a minor service?” check fails. If you stuck to the principle, this would not have happened.

How can you avoid it? Well, you can set up stricter contracts between the extending types. In your case, the class named `AbstractType` is the contract, telling you what your functions should look like, because that is all a contract is basically: an agreement that if you extend me, you should do things my way. You can make this contract more understandable by changing the parent’s function as follows. The `must_service` method inside `AbstractService` can be given a return parameter typehint, as follows:

```
def must_service(self) -> bool:
    print('Calculating ' + self.TYPE + ' service mileage')
    return self.current_miles - self.previous_miles > self.
        SERVICE_MILES
```

The method now states exactly what type of value should be in its response, and it will be a lot easier for the developer to follow and extend on it.

Interface Segregation Principle

The interface segregation principle states that no object should be forced to implement methods it does not need. How do you force an object to use methods it does not need? Similar to parent classes, you have interfaces. Interfaces are like parent classes, but their functions do not have any executable logic. An interface is a way for a developer to force a contract upon a class. In other words, it is a way to force a developer to use very specific functions. Saying “contract” is a fancy way of saying “exactly the same function declarations and return types.” This way you can create similar objects that can be used in similar places. Looking back at Listing 9-2, your `Service` classes which extend the abstract class all adhere to a contract, well, to a lesser extent, because the parent provided to function `must_service`, but the principle stays the same. You can use all of the services because they are subtypes of one supertype and they adhere to its contract. To illustrate this idea even more clearly, let’s create an interface. To clarify, you use an interface to force a class using that interface to write implementations for the functions inside the interface. This is great because it means you can create many types that implement the same functions and use them interchangeably, like you did with the services. See Listing 9-6.

Listing 9-6. Interface segregation, incorrect example

```

import abc

"""
This is our interface
"""

class ActionsInterface(abc.ABC):
    @abc.abstractmethod
    def eat() -> str:
        pass

    @abc.abstractmethod
    def tree() -> str:
        pass

"""
This is the class implementing the interface
"""

class Human(ActionsInterface):
    def eat(self) -> str:
        return 'I am eating'

human = Human()
print(human.eat())

```

You use the `abc` library, an initialism for Abstract Base Class, to help create the interface. Looking at the methods inside the `ActionsInterface`, you can see that they only have the statement `pass` in their bodies. `pass` just means “go ahead and do nothing.”

To come back to the interface segregation principle, the interface you created is in violation of it. Whereas a human can eat, it certainly cannot tree and you have no reason to implement the `tree` function. If you want your human objects to conform to the `ActionsInterface`, you will be forced to let them implement the `tree` method as well. The code in Listing 9-6 won’t run unless you move `tree` out into its own interface, as in Listing 9-7.

Listing 9-7. Interface segregation, correct example

```

import abc

"""
This is our actions interface
"""

class ActionsInterface(abc.ABC):
    @abc.abstractmethod
    def eat() -> str:
        pass

"""
This is our botany interface
"""

class BotanyInterface(abc.ABC):
    @abc.abstractmethod
    def eat() -> str:
        pass

"""
This is the class implementing the interface
"""

class Human(ActionsInterface):
    def eat(self) -> str:
        return 'I am eating'

human = Human()
print(human.eat())

```

To get around this problem, the solution is easy. It's better to have many small interfaces that are focused on specific things than a big interface containing many contract items that will not be used by your class.

Dependency Inversion Principle

The dependency inversion principle may be a bit trickier to understand but fortunately you already have an example! This principle states that your higher level classes, the ones that get executed first, should not rely on concrete lower level classes, but instead rely on abstractions of those classes. Your higher level class does not need to know the exact details of the lower level class it is receiving. Instead it can rely on what is called *duck typing*, which basically means that if it has the right functions, then you're good to go. Where did you implement this? Once again, go back to Listing 9-2. Look at the Vehicle class.

class Vehicle:

```
def __init__(self, vehicle_type: str, numberplate: str):
    self.vehicle_type = vehicle_type
    self.numberplate = numberplate

def time_for_service(self, serviceObject: AbstractService):
    return serviceObject.must_service()
```

Looking at the `time_for_service` function, you implemented dependency inversion. This function, a higher level function, does not know or care about what service it is getting. It is dependent on the abstract class, and any subclass can be given to it.

Dependency inversion decouples you from the implementation details of the lower level functions and allows you to reuse the code more easily.

DRY

Dry is another acronym, standing for Don't Repeat Yourself. This rule is very easy to grasp. It means that for every aspect in the system, there should be one and only one authoritative instance or representation in the system. Once you start having duplicate sources of knowledge, you run

the risk of one getting out of date and being used by a new developer to yield incorrect results. It means that changes only need to be done in one place. This can be something small like a VAT rate or a class that retrieves a person's password. It should live in only one place.

A few years ago, South Africa's VAT rate changed. I was working on a financial system that relied heavily on VAT calculations. Turns out that VAT was only declared once in the system's calculations and used throughout as a constant, so to change the VAT rate on the system was as easy as changing a single constant, and the hundreds of other places it was used did not need to be touched.

Composition

Composition is a class creation technique considered to be the exact opposite of inheritance. The one technique is not better than the other. Both are great if applied correctly. Composition merely refers to creating an object built up from other objects as opposed to inheriting those objects' details. Look at Listing 9-8 as an example.

Listing 9-8. Composition example

```
class DB:
    def connect(self):
        print('DB connect class')

class LogFiles:
    def write(self):
        print('LogFiles connect class')

class InheritanceAccess(LogFiles, DB): #needs database access
    and log access
    def __init__(self):
        print('Inheritance')
```

```

class CompositionAccess(): #needs database access and log access
    def __init__(self, DB, LogFiles):
        print('Composition')

inheritance = InheritanceAccess()
composition = CompositionAccess(DB, LogFiles)

```

Look at the class named `Inheritance`. It works but is a really bad use of inheritance. You now have a `User` with supertypes of `DB` and `LogFiles`, meaning `User` is-a `DB` and `LogFiles`. This is completely wrong. You should never inherit from a class purely because it has cool things in it, like a `DB` connection for instance. You will also be very prone to breaking the Liskov substitution principle this way.

A better solution in this case is composition where you create a has-a relationship. Look at the class called `Composition`. Its constructor takes two classes, `DB` and `LogFiles`, and you cannot instantiate your class without them. Your `CompositionAccess` class is still a `CompositionAccess` class, with a has-a relationship to `DB` and `LogFiles`.

Summary

If you can stick to most aspects in the chapter for the design and code writing phase, you will be good to go. It does take a lot of practice and many errors to start recognizing where a SOLID principle gets violated or how to get all of the details in a diagram. The best approach is to (almost) never slack off. Approach all the code you write with seriousness, as each day of designing properly and thinking about your solutions and problems will bring you closer to intuitively noticing when you are about to make a mistake in your code or sensing an error in someone's code you are reviewing.

CHAPTER 10

Security

A very important aspect of writing code is securing it, and in this chapter, we will look at some ways to do so. The aim of this chapter is not to teach you how to break into systems, but to make you aware of the different attack angles and how to protect against them. There are a lot of people out there who will try and break into your system, for various reasons. In contrast to popular belief, people who break into systems are not called hackers. They are actually called crackers. As a software developer, a large portion of the task of securing the system will fall on your shoulders. Chances are that some weakness in your code will expose a path into the system, to steal data or gain unlawful access.

This chapter contains some common attack vectors that you should know about and that you can easily prevent. As with all the other chapters, there is a lot more to breaking into systems than this, but this is a great start. A great online resource to read is owasp.org. They publish yearly lists of the top ten threats, which are great insights as to how to protect your system.

There are various reasons people break into systems. It may be for financial gain, to deface a site of an entity they don't like, to steal data, or simply for bragging rights. Whatever the motive may be, no target is too small. You should never think that no one will attempt to break into your system. Crackers are aided by the anonymity of the Internet and can snoop around your system for months without you having a single clue that it is happening. They also employ automated tools, which makes it even easier for them. On a small system I built about three years ago, I decided

to install a web application firewall called modsec. This was after about a year of the site running. Within the first day of modsec running, I started seeing malicious access attempts. And it was happening daily. Looking at the timestamps of my modsec log entries, I could see it was an automated system doing the requests. One request exactly every 5 minutes. What could I do about this? Well, I had already done something about it. Installing Modsec prevented those requests from even reaching my system, and that was a great start. But I am still pretty sure some requests made it through to my system. For those requests, I employed the tricks I will mention below.

Before we start, I want you to remember that you will potentially become a software engineer one day. The emphasis of your job will be to build software. The emphasis for a cracker is to know systems and exploit them. Never think you know more than an attacker does, because the chances are, you don't. If you have 1000 input fields, of which one is unprotected, chances are that after four months of trying, an attacker will find it, so make sure everything is plugged up.

Securing Your Code

You are going to look at the following aspects of securing your system. The first section is on a code level and consists of the following. This is not an exhaustive list, and it is always good to look at owasp.org for a full threat list.

- SQL injection
- Cleaning variables
- Keeping errors a secret
- XSS
- CSRF
- Session management

The second section is more on a system level:

- Keep your system up to date
- Database users
- Ports
- Docker images
- HTTPS
- Password policy

The third section is an attack vector that is becoming more and more prevalent and is called social engineering.

Code-Level Security

SQL Injection

SQL injection is a very easy way to compromise databases. By using this technique you can steal, edit, and destroy data. But it is even easier to protect against SQL injections, and in fact, you already did so when you wrote your Flask application. First, let's look at the basics of a SQL injection attack. I will show a basic attack, but they can get very complex.

A SQL injection attack is performed when an attacker sends a partial or full SQL string as data instead of, for instance, a username or password, formatted in such a way that it will edit your SQL and change it. Let's say you have a table called `users`, and within this table, you have `username` and `password` fields. To retrieve login data, you run the following query. Bear in mind that the `username` and `password` fields are provided by the user using your system.

```
SELECT * FROM users WHERE username = 'peter' AND password =  
'h2*K64e';
```

In this case, the username is equal to 'peter', but peter would come through as a variable (as would the password, but we can ignore the password for now).

```
user_name = 'peter'
password = 'h2*K64e'
```

```
SELECT * FROM users WHERE username = user_name AND password =
password;
```

In SQL, you can comment out legitimate SQL by using two dashes, --. Everything after the -- gets ignored by the SQL engine. Let's can exploit the usage of these dashes to help us craft a SQL injection attack.

What would happen if we change the user_name variable to look as follows?

```
user_name = " ' AND password != 'dfasdf' ;-- "
```

Our SQL will be executed to look as follows:

```
SELECT * FROM users WHERE username = 'aa' AND password !=
'dfasdf' ;--' AND password = password;
```

In the above statement, due to the injection, all that will execute is the following:

```
SELECT * FROM users WHERE username = 'aa' AND password !=
'dfasdf'
```

We start our attack with a ' to close the username variable field, and then we add our own SQL and a -- to ignore the more restrictive part of the SQL. We actually overwrite the more restrictive part with our own password condition.

```
password != 'dfasdf'
```

The chances are millions of times higher that you will guess the wrong password than that you will choose the right password.

If you are a beginner, getting even a simple example like this to run can be tricky, but once you have practiced a bit, it becomes very easy to craft.

So how do you protect against this? You use a technique called *SQL escaping*. In software packages, like SQLAlchemy, escaping is done with a technique called *parameter binding*. Remember that in Chapter 7, you used Flask to build your REST API. You also used SQLAlchemy to manage your database queries, and one of the examples is shown in Listing 10-1. In this example, you see parameter binding. Parameter binding is the strongest way of preventing SQL injection.

In the SQL string, you use a colon to denote where a parameter should be instead of just shoving the raw variable into the SQL. In this case, you have `:id_num`. When you execute the SQL, you provide the value to `id_num` as a parameter, on the second line of Listing 10-1. And that is how simple protecting against a SQL injection attack is.

Listing 10-1. Protecting against SQL injection

```
sql = text('SELECT * FROM users WHERE id=:id_num')
result = db.engine.execute(sql, id_num=user_id).fetchone()
```

On the other hand, when you use an ORM, as you do in the second part of your REST API, SQL binding and escaping of SQL happen automatically.

You must always escape all your SQL queries, or use parameter binding, no matter what field it is!

Cleaning Variables

Cleaning variables is an easy technique that not only can help prevent attacks upon system input but also help your system against crashes when it inserts data that is not in the correct format to be inserted into a table

column. Let's assume you have a column that takes three characters for an office telephone extension column. Then you should not even let a request with four non-integer characters reach your database. Assume you want to query all personnel with the office extension of 123. Even if the database is set up to not allow any integers of more than four characters to be inserted into the system, you can still send a request like this pseudocode:

```
ext = " 123'; drop table_name -- "
```

If this passes (if you have now SQL injection protection), it will drop your table called `table_name`. You can also drop a complete database like this. But, if you on a programmatic level say that an extension value may only be three characters long, and must be all digits, then this would not have even reached the code that executes the SQL queries.

This technique can protect against certain attacks, or at least make them a lot harder to execute. I also advise that your database tables size restrictions and your validation rules are exactly the same. If your database only accepts strings that are 10 characters long, then your validation rules must specify that exactly.

So how do you implement this? In your Flask app, you used a library called Marshmallow. Marshmallow contains all the code for verification you will need. One way to implement this is as follows. Create a directory called `validation` in your root directory, and inside that directory, create a Python script called `validation.py`, and also add an `__init__.py` file to make it a package. Inside the Python script, add the code in Listing 10-2.

Listing 10-2. Cleaning variables class

```
from marshmallow import Schema, fields
# import built-in validators
from marshmallow.validate import Length, Range
```



```

class UserSchema(Schema):
    # Required value shorter than 50 characters
    name = fields.Str(required=True, validate=Length(max=50))
    # Required value shorter than 50 characters
    surname = fields.Str(required=True, validate=Length(max=50))
    # Required value shorter than 12 characters
    identity_number = fields.Int(required=True,
                                validate=Range(min=1))

class IDSchema(Schema):
    identity_number = fields.Int(required=True,
                                validate=Range(min=1))

```

In the top two lines you import Marshmallow, and specifically `Length` and `Range`. There are loads more for you to choose from, but for your needs, you will only use these two. The classes you create under the import statement are called `UserSchema` and `IDSchema`. They are named like so because one will validate User data, name, surname, and `identity_number`, and `IDSchema` will only validate `identity_number`. To call this code, you just have to feed the `validate` method a dictionary object with the correct entries, as in Listing 10-3.

Listing 10-3. Using the cleaning variables class

```

errors = id_schema.validate({'identity_number': user_id})
if errors:
    return json.dumps(str(errors)), HTTPStatus.BAD_REQUEST

errors = user_schema.validate(data)
if errors:
    return json.dumps(str(errors)), HTTPStatus.BAD_REQUEST

```

In Listing 10-3, in `user_schema.validate(data)`, `data` is a dictionary object, and the `validate` function is inherited from `Schema`. In Listing 10-2 you can see the classes extend `Schema`.

Listing 10-4 demonstrates how this fits into the REST API. You populate data with a dictionary that you get from `request.get_json` and feed it into your `user_schema` class's `validate` method. You then return a json string with the appropriate HTTP code upon failure or continue execution upon success.

Listing 10-4. Cleaning variables inside the REST api

```
@app.route('/v1/user', methods=['POST'])
def post_user_details():
    try:
        data = request.get_json()

        errors = user_schema.validate(data)
        if errors:
            return json.dumps(str(errors)), HTTPStatus.BAD_REQUEST

        sql = text('INSERT INTO users (name, surname, identity_
number) values (:name, :surname, :id_num)')
        result = db.engine.execute(sql, name=data['name'],
surname=data['surname'], id_num = data['identity_number'])
        return json.dumps('Added'), HTTPStatus.OK
    except Exception as e:
        return json.dumps('Failed to add record. ' + str(e)),
        HTTPStatus.NOT_FOUND
```

All fields should be validated, without exception!

Keeping Errors a Secret

You should exercise caution when you handle errors generated by your system. Any attacker will do a lot of inspection of your system to try and coax it to break and spill the beans. Attackers can learn a lot by looking at certain outputs and error messages. They can learn what webserver

you are using, what language interpreter, what framework, or what CMS. Always make sure you send back normal, human-readable error messages that do not give any clues about your system. All of these bits of knowledge are baby steps towards attacking your system.

An interesting event that happened a few years ago, which is an extreme example of not keeping an error a secret, was when a colleague of mine tried to enter a legitimate string as a password into a system. Turns out the system did not accept all the characters in the string, and it returned an error to the screen. This was fine, apart from the fact that the error message was the exact SQL string that the MySQL server tried to run. At this point, he called me and showed me how he could craft a SQL injection attack with this feedback. Normally, a SQL injection attack happens blindly, where you feed it malformed SQL hoping that you get it right, but having an error like that printed on the screen, which also showed the SQL was not escaped, was like taking candy from a baby. Well, it would have been, if we took it further. We decided to send the owners a message, informing them about the flaw. Sometimes, if you have developed software for long enough, you get to dislike cracking software as much as you dislike the idea of getting your software cracked. And remember, be ethical. Just because someone's front door is open does not mean you can go in and browse around or steal.

XSS

XXS is one of the more prevalent attack vectors currently. XSS stands for cross-site scripting and happens when you inject malicious code into a system and let that system execute that code unintentionally. What can you do with this attack? You can steal someone's login credentials, you can redirect them to different websites, and actually do a fair amount of damage. In general, you inject JavaScript into a system. Even though we did not go into JavaScript, you will encounter it at some stage. Let's say you

submit a form on a webpage, and one of the fields is your name. Then it is possible to provide a string for the name field that will actually be executed by your browser when your name field is displayed.

To put it differently, when you log into a system from a browser, the browser executes certain actions. It will, for instance, render the HTML and the CSS and load the JavaScript. There may be JavaScript that can run on page render. If your username is valid JavaScript, then it will execute the JavaScript instead of displaying a username.

So, if instead of providing my username as `abcde`, I provide `<script>alert('hi there')</script>`, and there are no security measures in place, then when I log in and the page renders, I will see a popup that says 'hi there.' Funny as that example may seem, an attacker can steal your login credentials this way. A CSS string can be created that takes your authentication cookie from your browser and POST's it to a remote system. I worked for a company many years ago that used a popular CMS, which loaded its visual components from a database. An attacker managed to overwrite one of these components by injecting JavaScript code that redirected you to a different site whenever that component loaded. That component was now valid JavaScript and was executed by the browser. So, whenever you opened the site, you were redirected to another site. It puzzled me that the attacker did not do anything worse; surely all our login cookies could have been stolen, but maybe they just wanted to warn the company that the CMS had a weak spot.

How do you protect against an XSS attack? There are two schools of thought: one says to sanitize all incoming data and the other says to escape outgoing data. Because XSS is executed upon displaying data, I agree with always sanitizing upon display. All data displayed should be escaped. Escaping data means that no matter how clever the JavaScript string is, it will be treated as a string. Instead of executing the JavaScript, it will be printed as a string to the screen.

When it comes to sanitizing input, this can get tricky. You need to consider what the data will be used for and how it will be used. Sanitizing blindly can lead to data inconsistencies and errors. Just a quick word on sanitizing vs. validation. Sanitizing will actually strip the JavaScript out of the incoming text, or you can just stop execution when you detect it and return an error to the user. So, sanitizing entails taking the incoming data and inspecting it for malicious text that can be executed as JavaScript and removing it, whereas validation makes sure the string parameters are valid. There are so many variants of XSS attacks that it can be hard to say your code is detecting all of them. Your code may also catch false positives, meaning it may reject actual valid input. I lean more towards escaping all output, and not sanitizing all input. Quite a few frameworks nowadays escape output by default, which is great. The problem with sanitizing input is that it is based on the perception that you have indeed covered all potential XSS attack vectors, and you have to trust your code is safe. But attackers have time on their hands, and after months of trying, they may find an XSS string that works. An attacker will get past your sanitizing code and you will not know about it.

Whenever you display text in a browser, make sure each and every field is escaped! Make sure you know how the framework you use handles sanitization.

CSRF

CSRF stands for cross-site request forgery. I have never seen this attack personally, but I have removed code that was vulnerable to a CSRF attack from a codebase.

Let's say you are logged into a social network website. Let's call it MyFriends, and let's assume this site is vulnerable to CSRF attacks. You receive a random but cleverly crafted email that you open, and inside that email is a link to a website with something that may entice you to click it and open the website, such as 10 photos of adorable kittens. When you

open that link in your browser, it will not contain any photos of kittens. No, on the contrary. It will do a normal form post to MyFriends, with a stock standard form post that may request something like a password change, name change, or to make a payment. The action will depend on what MyFriends has to offer. If you are logged into MyFriends at that stage, or if your session is active but you think you are logged out, your browser will include your session cookie into the request automatically, causing whatever action is being posted to execute. All this is done from the attacker's "kitten" website to the MyFriends website.

How do you protect against this? It is easy to prevent a CSRF attack and nowadays a lot of systems come with built-in CSRF prevention techniques. The best method is a CSRF token. The token is shared between the MyFriends backend server and the front end, and it must change on every request. Whenever you do a request from the front end to the back end, you send the CSRF token with it, and the backend code will confirm that it is the correct token. It will then discard that token and send a new one to the front end. This is at this stage foolproof because the attacker's page does not have access to those tokens at all, because their page was precrafted, whereas your page now, due to the CSRF token, has an element of being dynamic to it.

In short, a CSRF attack is attempted when you are logged into the target system, and you open a browser tab from a completely different system that has code that sends an HTTP request to the target system. The browser will handle the session for the attack.

Make sure all forms being submitted have CSRF tokens!

Session Management

Most systems nowadays come with session management built-in. This does not mean it is perfect. You still need to make sure the session cannot be stolen. I urge you to look this up on OWASP's website, but here are some of the easier fixes:

- Make sure your session has a timeout value. You do not want a session to live for days.
- Make sure your session is only generated server-side.
- Do not allow multiple simultaneous logins with the same user id.
- Session data should not be in the GET string, as this is human readable, won't be encrypted by HTTPS and also dumped in log files. Remember, you never know who gets access to your log files.
- Provide the user with functionality to log out of your system. Logging out of a system in general destroys your session or access token.

The list goes on and can become quite technical. The gist of session control over the Internet is that we use cookies on the browser side to identify users or access tokens. When someone steals a cookie, they have access to that person's login profile. It is important to have proper session controls in place.

System-Level Security

Apart from on a coding level, there are various system strategies you should be aware of that can lower the risk significantly of your system getting compromised.

Keep Your Systems Up to Date

Keeping your systems up to date is very important. It plugs known vulnerabilities, which is a very tempting attacking vector for an attacker. You should have a list of the technologies that you use (for instance, Python, PHP, Nginx, Linux, and WordPress) and whenever a security update is available, you should update them.

Database Users

Database security is not just to prevent attackers from creating havoc in your system, but also to prevent your developers from accidentally breaking it. You will always have a master, known as root, database user. That user must remain secret. Only a privileged, trusted few may have those credentials. Other users of the database will include the following:

- Your software developers
- Your applications

The permissions you grant to your software developers are up to you, but it is a great idea, if you have the infrastructure, to let them work on a copy of the production database, and not on the actual production database. This is just in case they run a query that accidentally deletes data or drops a table. They must also each have their own username, and not shared credentials. Furthermore, they must have restrictive permissions, without the ability to create more users.

Your system is also a weak spot in database access and must have permissions that are very limited. It must not be able to

- Create, delete, or update users
- Drop a database
- Drop a table (although this is up to you as some frameworks need to drop tables during the migration process)
- One system username/password combo must be tied to the IP range of your application calling the database
- One system username/password combo must be used per database. If one set of credentials leak, you do not want to give an attacker access to all your databases.

Ports

On your system's side, when you have set up your server, you need to close down all ports that you do not need. Apart from needing an IP to access any system, you also need a port. And there are lots of them. Ports in general indicate a specific application. For instance, by default, HTTP uses port 80, HTTPS uses port 443, and SSH uses port 22. Databases have their own ports, FTP has its own port, and so on. You must only expose the ports of the systems you are using. How you open and close ports depends on what you use as a server. Different service providers may have different tools to do this, so once you choose a system, like AWS, you should do some research on how to open and close ports.

Docker Images

If you do not build your images yourself, then you have to make sure you find images that are trustworthy. It is not unknown that Docker images with backdoors and other severe security issues exist, and care should be taken to make sure exactly what is on the image before it gets used. Make sure you use a trusted vendor and not just someone's Docker image they uploaded to Docker Hub.

HTTPS

If you are dealing with web applications, for instance using REST, you have to use HTTPS and not HTTP. HTTPS encrypts all communications between the frontend system and the backend system in case the message gets intercepted by someone eavesdropping on the communications.

Password Policy

You will need a lot of passwords to secure the multiple entry points into your system. Do not skimp on this. Select a different password per system you need to log into and make sure it has lowercase and uppercase letters, digits, and some non-alphabetical characters. Enforce this password policy for everyone and make it against company policy to share passwords.

A new trend is to use passphrases.¹ Passphrases are easy-to-remember sentences consisting of non-related words.

About 11 years ago, I was asking someone in another department to show me how their work was done for a feature I was building for them. As they moved through the steps to complete a task on the system, I noticed that they could see very sensitive information. I stopped them and told them that this was a bug and they were not supposed to see that information. They replied, “I am logged in with the financial manager’s account.” I asked them how they got the password and they said, “He gave it to us.” So numerous people were using his login details to perform their tasks. This was such an immense risk to sensitive data and it blew my mind that he could have done this. It was promptly stopped that same day, and permissions were added to let the finance team see the additional aspects they needed to perform their tasks.

Here is another password story to point out the severity of bad passwords. We had an interview, and the interviewee gave us a website he built as an example of his work. I reviewed the site a little, but you cannot tell a lot from a site alone, since you cannot see any code, so out of slight boredom, I decided to see how security conscious he was. I went to his site’s login page and typed admin as the username, and something like 1234 as the password, and just like that I was logged in as admin. I really did not expect it to work, but suddenly I gained accidental illegal access to

¹www.avg.com/en/signal/how-to-create-a-strong-password-that-you-wont-forget

a system from my work computer and office IP address, which was against our company policies, so I had to tell my manager what I did. We decided not to hold this flaw against him in the interview. His software was not developed against our company standards, and it would have been unfair to penalize him on that. But let this event serve as a reminder that you should never ever skimp on your passwords.

Passwords matter.

Social Engineering

The last aspect to talk about is social engineering. Social engineering can take many forms, but in general, it exploits a human weakness in a company to divulge sensitive information. It can, for instance, be someone acting via telephone or email as if they are the head of technology asking you for your password because of some strange but very legitimate sounding reason. It may be a fake email, spoofed to look like it is from the CEO of the company asking you to divulge your PIN code to get into the office. In many instances, someone crafting this kind of attack has done their homework carefully. They may even sift through the company's trash to look for clues. This may sound ridiculous but imagine this scenario. Someone in your IT team writes down the name of your server in a meeting, and its IP address and open ports. Eventually, this paper ends up in the trash where the attacker finds it (this is not an unheard of method to harvest data for an attack). He can now craft his attack as follows. He can state he is from your hosting company and is doing an audit so he needs to log into your server. He knows the server name, he knows the server IP address, and he knows the open ports. This will make him sound trustworthy. With this information, he is slightly closer to his goal, which is to break into your system. All he needs is a human who will fall for his plan. Each piece of data he gathers is another baby step towards his goal.

At a previous company where I worked, the system administrator and IT manager did a test. They purposefully created a bogus email account for the system admin. By looking at it, you could see it was not his email address. They then set up a website that looked like ours, and once again if you inspected the URL, you could see it was not our company's URL. After that, they sent out an email to everyone in the company to "test" their logins. The email stated that they created a new fall-over system and people must please try to log in to see if it works. No login data was captured, but a count of how many people fell for it was recorded. It was quite staggering: a large number of people would have divulged sensitive information to what, once you looked a little closer, was clearly a social engineering attack.

How can this kind of attack be stopped? This can be difficult because sometimes these attacks are planned for a long time, and they are truly well designed. But educating people around you as to how these attacks happen is an angle. So is not divulging any sensitive information, which means shredding even your own notes and not chucking them into a bin.

Summary

We have gone through quite a few topics in a short time. Securing a system can be easy if you follow these steps, but you will never really know if your system is secure. I have seen attacks that are completely outside what even I would have dreamed of doing. You need only one vulnerable spot to give an attacker a chance.

Here is a little roundup of what you learned in this chapter:

- SQL injection
 - Use parameter binding, or if you use SQL, escape all your SQL queries.

- Cleaning variables
 - Make sure all your variables conform to specific data types and sizes. Otherwise, reject them.
- Keeping errors a secret
 - Send back meaningful error messages, but never send the exact error thrown by a system so that a user can read it.
- XSS
 - Always escape any output that is displayed in your browser
- CSRF
 - Always use a CSRF token to identify a request as coming from your own system.
- Session management
 - Make sure your sessions expire.
 - Never send the session-id via the GET string.
 - Read the OWASP recommendations for session management.
- Keep your system up to date.
 - Always patch your system with the latest security updates.
- DB users
 - Create users with limited privileges. Pin privileges down to exactly what is needed by that database user or system using the database. Do not give any privileges they will not use. Each user and system should have a unique login per database.

- Ports
 - Close all ports that are not being used. Only open the ones actively being used. That way you can secure the two or three ports you use instead of the thousands available.
- Docker images
 - Make sure you are using secure images. Do not go by the number of downloads when making this decision. You must look at the content, or the vendor, or build your own images.
- HTTPS
 - Always use HTTPS as your protocol to send data between servers and clients. Do not use HTTP.
- Password policy
 - Make sure your passwords are strong and do not share them. Do not use a system with shared details. Also, do not use the same password for different systems.

CHAPTER 11

Hosting and CI/CD

No one really builds software with the intent of hiding it somewhere and never using it. Sure, you may end up with a few half-finished projects that were aborted because something more interesting came along or you just ran out of enthusiasm for them. But more often than not, especially if it is a project that can generate money, a business venture of some sorts, you want it released.

Once your product is ready, you need to release it. Releasing your code can mean a few things. It all depends on the software you have built and what platform you want to serve it on. You may need to deploy an executable that can run on a Windows computer, or it may be a web-based application that is served up from a server and accessed via a browser. In this chapter, we will discuss some ways to get your software out in the wild if you are deploying web-based applications.

We will also look at some hosting solutions, as well as some steps to get the application on your hosting machine. We will discuss different types of hosting solutions, and we will do continuous integration on a practical level, and continuous deployment, which is based on the same file as continuous integration, on a theoretical level. The reason for this is because to get code on a hosting server (continuous deployment) you may incur costs, and I cannot choose a service that may incur costs for you. However, the rules to deploy remain roughly the same. Each hosting service provider will have its own way to get code onto it.

Types of Hosting

Hosting refers to a remote service, or set of services, that your project will use to be executed on and where it can be accessed by the public. There are different types of hosting available, and depending on your needs and budget, you can select something that closely suits your needs. In general, you can assume that the less you pay, the less functionality you will get with your hosting provider, but this is not always the case.

Cloud and Serverless Technologies

There is some technical jargon you need to understand before you can look at hosting services. Your application can be hosted on a single machine that also hosts other peoples' applications. Or you can have a more configurable virtual server to yourself. But you need to know about two new technologies before you can explore the hosting of your application. They are *cloud-based* and *serverless*.

These technologies are very popular but sometimes misunderstood. There is a joke that "There is no cloud. It is just someone else's computer." There is some truth to this joke, but it is a lot more complex than that. The cloud is not just someone else's standalone computer that you access via the Internet. It is a network of distributed and pooled resources that seemingly work as one. It offers redundancy, so should one of the cloud resources fail, you won't lose functionality in your application. It also offers scalability, so should you need more resources to handle increased traffic to your application, you can get them. It can even scale its resources automatically. With cloud-based technology, you still have a lot of configuration power over your system. Serverless is also cloud-based, but with serverless technology, you as the developer do not have to concern yourself with the server provided by the service at all. It is all completely managed by the provider. In general, serverless may be

associated with a pay-per-use scheme, where you for instance only pay per milliseconds of execution. With this setup, you only incur costs when you actually use the system. With the speed of setting up, automatic scalability, and absolute ease of use, serverless is a winner, and cloud-based architecture brings power and control to people who are not dedicated system administrators.

Shared Hosting

Shared hosting is the cheapest and easiest way to get up and running. It gives you a workable system, normally with a nice intuitive user interface to manage your services. It comes complete with various tools that make your life easy as a non-technical person. It is not for high-traffic scenarios, though. A fixed limit is placed on your resources, such as memory, both long-term storage and random access memory, as well as computing power in general. This is a great solution for a project that does not expect a lot of traffic or to do a lot of intensive operations. You may also be restricted with the amount of web traffic you can send and receive, as bandwidth is often also limited in these setups.

So, some of the drawbacks are, you are severely limited as to what you can do with this system. It normally comes with an interpreter, like PHP or Python, Git, and MySQL and that is it. In general, you cannot run Docker on it or install any other software on it (apart from the software you wrote yourself). As mentioned, it also comes with performance restrictions.

But do not be fooled by these limitations. For small projects, this may just be the perfect fit.

Getting your data on a shared server is normally quite easy. You can easily pull the code from you git repo into the root directory of your hosting site where your code will be executed.

Virtual Private Hosting

Virtual private hosting is a cost-effective way to get a more powerful, scalable, and configurable server. This can be a great choice if you are more technically minded and have a more demanding project. If you are releasing your code as a Dockerized application, then this is where you should start looking for hosting solutions.

A virtual private server, or VPS, will allow you to configure and install software on your server. On the backend, this is not a dedicated server for yourself. It is still a shared server in the sense that the resources are pooled and allocated to you. Due to the nature of a VPS in the original sense, you can still be affected by outages.

The benefits of this kind of service are really great. A lot of vendors who provide virtual private hosting still use pooled resources in the background. Although you do not need to worry about anything that happens in the background, you do need to worry (a little) about setting up your server, and technical aspects such as DNS, SSL certificates, and so on. This is not nearly as daunting as it seems, though. Services that provide VPSs for hosting normally try to make these aspects as easy as possible, with different tools that help automate parts of setting up your server.

Cloud Hosting

Cloud hosting provides a great low-cost way to host your system. Depending on what systems you are looking at, there is not always a difference between cloud hosting and a VPS from the angle that some VPSs claim to host cloud-based VPSs. But that is beside the point. Cloud-based services are becoming increasingly popular and increasingly cheaper. The word “cloud” in cloud hosting typical refers to a spread-out system (instead of single servers like shared hosting and virtual private hosting), where your application runs on pooled resources, and multiple copies of your code exist in different locations for redundancy. Your selected

resources (CPU, memory, bandwidth) are all duplicated across these locations as well. This helps with redundancy and helps your system switch over to a failover system without you even knowing your cloud-based “server” is down in the first place. You also get platforms on cloud services that handle automatic scaling and releases for you, giving your cloud service the feel of serverless technology, in the sense that you do not need to worry about scaling and load balancing. The technology is pretty vast and there is a huge amount to learn about cloud technologies.

Serverless

With serverless technology, you as a developer are completely decoupled from your server. A great example to look at is AWS Lambda. Serverless differs from a cloud implementation where it delivers exactly the computational power that you need when you need it. It does not preallocate resources and let those resources lie idle when you are not using them. Technically, your system does not even really exist when it is not being used. When you create software, and you want to run it using serverless, you must design specifically for this setup. It is not always the case, but for the most part you cannot just put your code inside the serverless architecture and expect it to run because there may be certain limitations that you should cater for. There may also be different ways to deploy your code on a serverless architecture.

Which Hosting Technology to Choose?

The above list is not an exhaustive list of technologies available for you to use, but it is a list of the most cost-effective and popular ones. Choosing one certainly won't be easy. Some factors to consider are whether you have the knowledge or time needed to maintain a VPS, or whether shared hosting will meet all your requirements. Going cloud-based or serverless is a great idea but also comes with a learning curve. The problem is when the

learning curve comes with an unanticipated invoice attached to it. My advice to you is to experiment with these technologies. But, make 100% sure of the pricing structures of the cloud and serverless tech that charge you per usage. When a cloud provider mentions something like a free tier, make sure that the free tier is not linked to certain conditions that you may inadvertently break.

Continuous Integration and Continuous Deployment (CI/CD)

You can simplify, manage, and maintain the integration of your newly developed features into the main branch, as well as deploy to your staging environment and production environment using CI/CD pipelines. CI and CD are two separate concepts, but they are defined in the same script and run in the same section of your deployment software, which I will describe in a minute. Continuous integration refers to the merging of feature branches into your main branch and running different tests on the codebase. This can be, for instance, building the application to see if all the components that need to be imported exist. You can also run tests with linters, which check your coding style and standards, and also run all of your unit tests. Whenever someone pushes in the main branch, it will start whatever tests you gave it to run.

The continuous deployment action refers to deploying your code to a hosting location. When the CI phase has succeeded, the system will push the code to your staging and production environments in what I called the continuous deployment stage. Depending on your needs, you may want the code pushed to staging but not released. You can discern between just pushing your code or releasing it, as well as which of your branches should be responsible for pushing the changes.

Creating the Pipeline

Let's discuss how to create the pipeline with a practical example of continuous integration and a theoretical example of continuous deployment.

Continuous Integration

Creating a CI/CD pipeline on GitLab (or any other provider, like BitBucket) is remarkably easy. The provider, GitLab in our instance, creates it for you. All you need to do is to provide it with a specific file with the details on what instructions the pipeline should execute. It will parse the file and run the instructions in the file in the pipeline.

So, your first step is to create a `.gitlab-ci.yml` file in your project's root directory. You add this file to your git repo and commit it. Upon pushing it, GitLab will automatically detect it and start executing the pipeline based on commands within this file, even if this file is empty.

The file can be divided into three sections, and each section is referred to as a stage. You will have the following stages:

1. Testing
2. Staging
3. Production

Testing is responsible for running tests. They can be unit tests, standards test, building the software, and so on. In the last line of Listing 11-1, notice `python3 test.py`. This command runs your unit tests. This is also where the magic of using SQLite for testing comes in. Not only does it run in memory, which makes it fast, but it also runs in memory on GitLab, meaning you don't have to worry about testing database details.

The staging and production stages are responsible for deploying the code to your staging and production environment.

That is technically all there is to it. Let's look at a few examples, starting with Listing 11-1. In this file, you declare the stages you will run. At this point, there is only one stage, and it is called test.

Listing 11-1. CI/CD pipeline

```
stages:
  - test

test:
  stage: test
  script:
    - apt-get update -y
    - apt-get install -y python-dev python3-pip
    - cd ./app
    - pip3 install --no-cache-dir -r requirements.txt
    - python3 test.py
```

Under the test stage, you name the stage and then you declare the actions you will run in the pipeline.

First, you update your system with the commands update. Because this update happens behind the scenes and you have no control over the output, you add the `-y` flag to signal to the command to select yes whenever the command line would have prompted a question requiring a yes or no answer. Second, you install your Python tools and Pip. Pip is used in the subsequent line to install your requirements. You then change the directory to `app` and run your Python unit test. `Python3 test.py` then simply runs your unit tests. If you have followed the tutorials, then you should be able to add a continuous integration test by just adding the file and these lines of code. If you have not followed it, but you have a project on GitLab, then I advise you to create this file and upload it to GitLab, even if it will fail. Log into GitLab, look for the pipelines link on the left, and click it. This way you will see that your continuous integration is at least getting started.

Continuous Deployment

You do not intend to deploy your code and use Docker as your database system. This is only for development. In production, as well as staging scenarios, you will use a database service like AWS RDS, or any other database system that your system can access. With this in mind, you need a way to swap between database configuration settings. You can achieve this by using a second `config.py` file, called `staging.config.py` and `production.config.py`, for instance. You also have a `local.config.py`. In general, `config.py` should not be committed to your Git repository; it should only exist in your local development system and be based on `local.config.py`. Your committed codebase should only have `staging.config.py`, `local.config.py`, and `production.config.py`.

Upon execution of the pipeline, you need to let the pipeline select the correct database configuration. This is easy enough to do. Note the changes in Listing 11-2 which will enable this change. The line `cp staging.config.py config.py` under the staging and production stages will copy the correct config file to be the correct config file in the relevant phases. The code in Listing 11-2 won't run out of the box, as the correct particulars are still needed to deploy the code.

Listing 11-2. More complete CI/CD example

stages:

- test
- staging
- production

test:

stage: test

script:

- apt-get update -y
- apt-get install -y python-dev python3-pip

```
- cd ./app
- pip3 install --no-cache-dir -r requirements.txt
- python3 test.py
```

staging:

stage: staging

script:

```
- echo "Deploying to the staging server"
- cp staging.config.py config.py
```

environment:

name: staging

url: https://examplelocation.com

only:

```
- master
```

production:

stage: production

script:

```
- echo "Deploying to the production server"
- cp production.config.py config.py
```

environment:

name: production

url: https://examplelocation.com

when:

```
- manual
```

only:

```
- master
```

In this listing, the `cd ./app` command takes you into the directory where the `testing.config.py` file lives. Then it gets copied from `staging.config.py` to `config.py`. Granted the details of the actual copying from GitLab to your host are omitted, but once you have selected your hosting provider, you should have no problem finding the details to fill in the environment section.

You also need to specify when the deployments happen, and this is GitLab-related. You selected two ways to achieve deployments. For staging, you selected to deploy whenever code gets released to your master branch using the `only: master` command. This will tell GitLab to run the deployment code. For production, you selected a more conservative approach. Using the command `when: -manual`, you indicate that deploying the code will be a manual process, and not automated. GitLab will provide you with a button on the pipeline which you can press to manually release the code.

Summary

You should always do proper research when selecting a hosting company. You need to consider your needs and budget and let them guide your decision. If you select a service like Amazon AWS, always make sure you know how the pricing structure works. AWS has great tools that can help you calculate monthly costs.

CI/CD pipelines can be a massive benefit in your deployment strategy. Once set up, you can deploy easily from anywhere. Once again, CI/CD pipeline are something you will encounter in the workplace, but there is no reason you should not have them in your own homegrown projects.

Here is a summary of what you learned in this chapter:

Shared hosting

- Incredibly easy to get up and running
- Limited operational resources
- Limited or even no capacity to add more software features
- Good customer support

Virtual private server

- Need some know-how to set up
- Depends on your provider and setup. More powerful, plus you can configure more services to run on it.
- Getting cheaper

Cloud hosting

- Great redundancy
- Depending on the service, a smaller learning curve
- Depending on your selection, can be very powerful, and this will influence the price
- Configurable
- Comes with platforms that allow automatic scaling

Serverless

- Great redundancy
- Software engineering is decoupled from the server, allowing development without worrying about the underlying architecture.
- Learning curve is not hard, with some potential pitfalls such as the documentation is not always easy to follow and calculating how much the service will cost you can be difficult.
- Pay-per-use plans may be cost effective but need to be monitored.

Index

A, B

Automated tests, [237](#)

C

Cleaning variables, [297–300](#)

Cloud-based hosting, [316](#)

Cloud-based technology, [314, 315](#)

Code-level security

 cleaning variables, [297–300](#)

 CSRF, [303, 304](#)

 handle errors, [300, 301](#)

 session management, [304, 305](#)

 SQL injection, [295–297](#)

 validation.py, [298](#)

 XSS, [301–303](#)

Code Quality steps

 automated tests, [236](#)

 codebase, [236](#)

 features, [236](#)

 peer review, [236](#)

 user acceptance test, [237](#)

Coding styles

 aspects, [144](#)

 blank line, [151](#)

 block comments, [146](#)

 docstrings, [147, 148](#)

 encoding, [151](#)

 imports, [152](#)

 inline comment, [146](#)

 line lengths

 arrays, [150](#)

 indentation, [149](#)

 line wrapping, [149–151](#)

 meaning, [149](#)

 linters, [145](#)

 naming conventions

 class names, [154](#)

 constants, [156](#)

 inheritance, [154](#)

 internal method, [154](#)

 internal variables, [155](#)

 method names

 (snake_case), [154](#)

 overwriting inheritance, [156](#)

 package/module

 names, [154](#)

 PascalCase/snake_case, [153](#)

 readability, [153](#)

 types, [153](#)

 variable names

 (snake_case), [155](#)

 official style guide, [145](#)

 whitespace, [152](#)

INDEX

- Command-line environment, [16](#)
- Containerization, [11](#)
 - disadvantages, [13](#)
 - dockerizing (*see* Dockerization)
 - serves, [12](#)
 - virtualized environments, [12](#)
 - webserver, [11](#)
- Continuous Integration and Continuous Deployment (CI/CD)
 - components, [318](#)
 - config.py, [322](#)
 - database system, [321–323](#)
 - GitLab, [319](#)
 - meaning, [318](#)
 - pipeline, [319](#)
 - stages, [319](#)
 - staging and production
 - stages, [319](#)
 - testing, [319](#)
 - test.py, [320](#)
- Coupling concept, [278–281](#)
- Crackers, [293](#)
- Cross-site request
 - forgery (CSRF), [303, 304](#)
- Cross-site scripting (XSS), [301–303](#)

D

- Database management
 - system (DBMS)
 - Adminer, [165–168](#)
 - cleaning up/pushing, [168](#)
 - configuration code, [162](#)
 - database creation
 - filling data, [175](#)
 - table creation, [174](#)
 - databases
 - collation, [172](#)
 - creation, [172](#)
 - LanguagesClass
 - database, [172, 173](#)
 - data types, [171, 172](#)
 - Docker file, [164, 165](#)
 - indexes, [161](#)
 - logical operators, [191](#)
 - metadata, [161](#)
 - MySQL, [162](#)
 - normalization
 - cities table, [184](#)
 - clients Table, [183](#)
 - data repetition, [180, 181](#)
 - first normal form, [192](#)
 - foreign key, [192](#)
 - JOIN clause, [185, 186](#)
 - languages table, [183](#)
 - LEFT JOINS and RIGHT JOINS, [190](#)
 - many-to-many
 - relationship, [186–188](#)
 - normal form, [181, 182](#)
 - one-to-many
 - relationship, [184, 185](#)
 - second normal
 - form, [182–184, 192](#)
 - third normal
 - form, [188–190, 192](#)
 - ports, [163](#)

- primary key, 192
 - caveats, 171
 - indexes, 170
 - meaning, 170
 - reading/writing, 159
 - refactoring, 159
 - relational database, 161
 - representation, 162
 - root password, 163
 - SQL (*see* Structured Query Language (SQL))
 - storage files, 164
 - storage sense, 160
 - tables, 161
 - username/password
 - attributes, 169, 170
 - Database security, 306
 - Dockerization
 - benefits, 12, 13
 - checklist/cheat sheet
 - commands, 34
 - compose commands, 35
 - steps, 34
 - command line tools, 17, 18
 - components, 14
 - compose/orchestration tool
 - container/execute code, 32
 - docker-compose.yml file, 28–31
 - experimental steps, 31
 - installation, 28
 - test.py file, 30
 - use of, 27
 - container, 16
 - Dockerfile, 14
 - image process
 - Dockerfile method, 15
 - step-by-step process, 15
 - infrastructure, 19
 - installation, 17, 18
 - Python application
 - server, 20, 21
 - repository
 - commands, 22, 23
 - COPY command, 25
 - COPY/CMD command, 25, 26
 - hello.py, 25
 - Hub username and password, 26, 27
 - output option, 22
 - run command, 24
 - test command, 25
 - VirtualBox, 18
 - Windows users, 18
- ## E, F
- Editing software (editor)
 - benefits, 5
 - meaning, 1
 - programming information, 2–4
 - Python, 2
 - text editors, 3, 4
 - text file extension, 1
 - Visual Studio code, 6–9
 - Escaping data, 302

G

GitLab project

- CI/CD pipeline, 319
- DB server and Adminer
 - sections, 207–209
- docker-compose.yml
 - file, 205, 206, 208
- Dockerfile, 205–207
- flask-server, 215, 216
- initial application, 210, 211
- migrations, 212–215
- postman, 212
- preparation step, 215
- project layout, 203, 204
- repositories, 202
- running system, 209
- source code
 - anatomy/endpoint, 220
 - app.py files, 219
 - config.py file, 218
 - DELETE, 227
 - functions, 221–224
 - GET endpoints, 226
 - initialise.py file, 217
 - migration.py file, 217, 218
 - PATCH, 227
 - POST method, 225
 - software programs, 216
- steps, 203

Git version control system

- cheat sheet, 60, 61
- commit hashes, 44
 - branches, 45–49

- checkout, 45
- commands, 47
- diff/status command, 47
- dropdown box
 - option, 48
- master branch, 45
- push master, 49
- test directory, 46

directory, 49–51

gitignore file, 56, 57

GitLab account

- Dockerfile, 42–44
- installation, 41
- push option, 42

git stash, 57, 58

hidden files, 37

merge conflict, 51–54

reset/revert, 59, 60

source control

- blame, 40
- codebase, 38
- code changes, 40
- continuous integration, 41
- features, 39, 40
- master deployment
 - branch, 38, 39
- peer reviews, 41
- pull request, 41
- repo information, 38
- source code, 38
- track changes, 40

SSH keys, 54–56

test directory, 51

H

Hosting solutions

- cloud-based/serverless,
314, 315, 317
- Cloud server, 316
- meaning, 313
- remote service, 314
- serverless, 317
- shared hosting, 315, 323
- virtual private, 316

I

Importing/ingesting, 235

Integrated development

environment (IDEs)

- benefits, 5
- features, 4
- meaning, 4

Integration tests, 242–246

- flask-server and
running, 244, 245
- get/post endpoint, 242
- production database, 243
- regression error, 242
- staging, 243
- TestConfig file, 245
- test.py file, 243

J, K, L

JavaScript Object Notation (JSON), 196, 197

M

Modelling

- activity diagrams, 266–268
- actor, 268
- database diagrams, 273, 274
- description, 267
- design/development
phase, 264, 265
- diagram/models, 264, 265
- high-level diagrams, 265
- low-level models
cohesion, 281
components, 275
composition, 290, 291
coupling concepts, 278–281
DRY, 289
object types, 275–278
SOLID, 282–289
- relationships, 274, 275
- swimlanes, 270–272
- tools, 265
- use case models, 268–270

Modsec installation, 294

N

Naming conventions, 235

O

Object calisthenics

- class collections, 141, 142
- class properties, 137–139

INDEX

Object calisthenics (*cont.*)

- control structures, 133
- else keyword, 134–136
- guidelines, 132
- indentation, 133, 134
- instance variables, 140, 141
- low coupling, 141
- object/function names, 140
- objects/methods, 140
- primitives/strings, 136, 137

P, Q

Preparations, 20

Python, 2

Python application server, 20, 21

Python programming, 63

- access functions, 124, 125

arrays

- dictionaries, 126

- lists, 126

- tuples, 126

basics, 66

classes, 128

classes/objects

- behavior/data

- (attributes), 106

- class definition, 108

- composition, 116

- constructor, 108

- dog class, 107

- rectangle class, 108

- email() function, 113

- encapsulation, 106

- inheritance, 113, 114

- instantiation, 107, 109–113

- magic methods, 116–118

- polymorphism, 115, 116

control statements

- for, 127

- if statements, 91–94, 127

- loops, 94–97

- while, 127

decision-making operator/

- structures

- algorithms, 81

- bitwise operators, 90

- combining operators, 81

- comparison operator, 82, 83

- expressions, 81, 88–90

- logical operators, 83–86

- membership

- operators, 87, 88

- operator precedence, 88–90

Docker environment, 65

exceptions

- built-in, 120

- catching, 129

- catching exception, 121

- creation, 129

- error message, 118, 119

- raise keyword, 120, 129

- try/except blocks, 119, 120

- writing code, 121

functions, 128

- anatomy, 100, 101

- arguments, 101

- custom functions, 99, 100

- named keyword
 - arguments, 106
 - parameter type-hinting, 104
 - principles, 100
 - return error parameter, 104
 - return/print data, 101
 - reusable code, 99
 - source code, 103–105
 - values, 104, 105
 - variable parameter, 105
 - human-readable
 - comments, 67
 - imports, 129
 - main.py script, 123, 124
 - packages, 122
 - logical operators
 - and, 83
 - not, 84
 - or, 83
 - truth tables, 84–86
 - loops
 - continue/break
 - clauses, 97–99
 - list comprehension, 97
 - for loops, 96, 97
 - while loops, 94, 95
 - membership operator
 - in, 87
 - not in, 88
 - scope, 125
 - scope/structure
 - built-in scope, 91
 - code, 90
 - enclosing scope, 91
 - global scope, 91
 - local scope, 90
 - sequences/maps
 - arrays, 75
 - data structures, 75
 - dictionaries, 80, 81
 - fixed size (immutable), 79
 - indexes, 77
 - keywords, 79
 - len() operates, 79
 - lists/strings, 76, 77
 - slicing, 78
 - tuples, 79
 - set up information, 65
 - spot indentation errors, 66
 - variables, 126
 - Boolean values, 71
 - built-in types, 67–69
 - casting, 72
 - constants, 72
 - floating point, 70
 - integers, 69, 70
 - shorthand assignment
 - operators, 74
 - strings, 71
 - type casting functions, 73, 74
 - writing code, 64
- ## R
- Readability, 131, 156
 - Refactoring code
 - coding tasks, 143
 - database design, 159

INDEX

Readability (*cont.*)

- extendibility/reusability, 144
- meaning, 132
- scalability, 143

Repositories

- commands, 22, 23
- COPY command, 25
- COPY/CMD command, 25, 26
- GitLab, 43
- hello.py, 25
- Hub username and
 - password, 26, 27
- output option, 22
- revision/source control, 37
- run command, 24
- test command, 25

Representational state

- transfer (RESTful)
- architectural design, 194
- concepts, 194, 195
- CRUD operation, 195
- framework code, 193
- front end/back_end, 194
- GitLab (*see* GitLab project)
- Gunicorn, 201
- HATEOAS, 200, 201
- HTTP
 - status code, 199
 - verb, 197–199
- JSON, 196, 197
- migrations, 202
- modern systems, 194
- ORM versions
 - app.py file, 229

code changes, 228–231

delete function, 232

GET, 231

PATCH, 232

POST, 231

query routes, 199

request/response

relationship, 196, 197

software development

process, 193

S

Secure shell (SSH) keys, 54–56

Security

cleaning variables, 311

code-level (*see* Code-level security)

DB users, 311

Docker images, 312

HTTPS, 312

keeping errors, 311

modsec, 294

owasp.org file, 293, 294

passwords, 312

ports, 312

sessions management, 311

social engineering, 309, 310

system level, 295

systems (*see* System-level security)

system strategies, 305

system updates, 311

Serverless technology, 314, 315, 317

Session management, 304, 305
 Shell-based editors, 2
 Social engineering, 309, 310
 Software development
 lifecycle (SDLC)
 approach, 258, 259
 calisthenics/coding
 standards, 263
 clumsy backups, 257
 deployment, 262
 design, 261
 development, 261
 maintenances, 262, 263
 modelling (*see* Modelling)
 phases, 258
 planning, 260
 requirements, 259, 260, 262
 software development
 process, 258
 specifications, 258
 testing, 261
 SOLID modeling
 dependency inversion, 289
 interface segregation, 286–288
 Liskov substitution, 284–286
 open/close principle, 282–284
 principles, 282
 single responsibility, 282
 Staging/production
 environment, 11
 Structured query language (SQL)
 Adminer, 176
 delete, 180
 DESCRIBE command, 176

injection
 binding and
 escaping, 297
 escaping, 297
 execution, 296
 parameter binding, 297
 queries, 310
 username/password, 295, 296
 INSERT, 178
 keywords, 175, 176
 SELECT, 177, 178
 UPDATE clause, 179
 System-level security
 database security, 306
 Docker images, 307
 HTTPS, 307
 keep systems updates, 305
 password policy, 308, 309
 ports, 307

T

Testing
 automated tests, 237
 time pressure, 253
 validation, 252, 253
 integration, 242–246
 new code, 250, 251
 peer-review process
 developer points, 255
 effective method, 254
 invaluable tool, 253
 reviewer points, 255

INDEX

Testing (*cont.*)

- refactoring process
 - functions, [248](#)
 - parameters, [248](#)
 - POST/PATCH data, [247](#)
 - post_user_details
 - function, [250](#)
 - request.get_json()
 - function, [247](#), [248](#)
 - return_message function, [249](#)
- staging server, [256](#)
- unit, *see* Unit testing

U

Unit testing

- data source, [239](#), [240](#)
- even number code, [240](#), [241](#)

flask-server/running, [242](#)

meaning, [238](#)

valid_age code, [238](#)

V, W, X, Y, Z

Virtualization, *see*

Containerization

Virtual private hosting, [316](#)

Visual Studio code

built-in features, [7-9](#)

extensions, [9](#)

features, [9](#), [10](#)

installation, [6](#)

keywords, [8](#)

testFunction()

function, [8](#)

workspace, [6](#), [7](#)