



THE WELL-GROUNDED Python Developer

Doug Farrell

MEAP



MANNING



MEAP Edition
Manning Early Access Program
The Well-Grounded Python Developer
Version 5

Copyright 2022 Manning Publications

For more information on this and other Manning titles go to
manning.com

welcome

Thank you for purchasing the MEAP for *The Well-Grounded Python Developer*. The goal of this book is to take you past beginning Python programming to the point where you think and feel like a software developer. Where the syntax and grammar of Python are not only comfortable for you but become the tools of a craftsman. With these tools you'll be able to take on bigger and more complex projects.

I first discovered Python when I moved from the world of Windows development to the Linux environment. I was an accomplished C/C++ developer, but was interested in learning the scripting languages available in Linux. Fortunately, I came across Python because of its strong support for Object Oriented Programming, which I enjoyed in C++. Since then I've never looked back, and Python has become my primary language.

There are a lot of beginner Python books, Python cookbooks and reference titles, and many of them are very good. To get the most from this book you should be past the beginner level and comfortable with Python. My hope is this book provides a middle ground that gives you more context about how and why you should take certain paths when developing a program.

You'll take the loops, conditionals and statements you know now and create larger constructs to handle bigger programming challenges. You'll learn some conventions and patterns that will reduce the cognitive load on you solving smaller syntactic issues and begin to think how to solve the big picture problems. How the tools you have, and will learn about, can be combined to take goal from problem to solution.

The book does this by taking you along on a development journey to build a web application providing a simple blogging platform. While I think that's an interesting application to build, the creation of that application isn't the end goal of the book. The journey is far more important, the ideas and implementation of those ideas as code, is the real value. The chapters progress and build on the previous ones to illustrate why certain choices were made, and how those choices can be altered and improved as more is learned later in the book.

As developers we are at an interesting place in the history of the industry. Engineering standards are as important in software development as they are in other disciplines. However, the pace of change in the software world changes those standards as well and provides an opportunity for a great deal of craftsmanship and artistry from a developer. Python gives you a powerful palette of tools to express that.

Please feel free to share your comments, questions, and suggestions in Manning's [liveBook Discussion Forum](#) for my book.

—Doug Farrell

brief contents

PART 1: GROUNDWORK

- 1 Becoming a Pythonista*
- 2 Your Python Environment*
- 3 Names and Namespaces*
- 4 Application Programming Interface*
- 5 Object-Oriented Coding*
- 6 Exception Handling*

PART 2: FIELDWORK

- 7 Your First Web Server*
- 8 Getting started with MyBlog*
- 9 Authenticating the User*
- 10 Persistence*
- 11 MyBlog content*
- 12 Prologue*

1

Becoming a Pythonista

This chapter covers

- Who is the intended audience?
- Being a developer
- Reaching goals
- Using Python
- The Version of Python to use

Welcome to Becoming a Pythonista! I'm happy to have you here for this journey, which I hope takes you to a place where you'll become not only a more robust programmer but one who thinks like a developer and expresses those thoughts using Python.

Right away, I want to clarify the use of a few terms, programmer, developer, and Pythonista. You might think programmers and developers are interchangeable; indeed, many job descriptions use the words that way. It's easy to think that way because the two are related, but developing is a superset of programming. A programmer is very focused on creating program code, using technology in that process, and implementing designs. A developer can, and often does, all of those things as well, but additionally thinks about the bigger picture. The big picture includes the architecture of more extensive applications, the design to achieve it, and pulling all the disparate parts together to implement it.

If you've read material online in the Python community, you've probably seen the term Pythonista. This term is an endearing name in the community to refer to ourselves and others who are enthusiastic about Python.

Along the way, we'll create programs and applications that will build upon each other, introducing ways to take on larger projects and break them up into smaller, more manageable pieces. As a developer, that's what you'll do. You'll be able to take on complex projects using multiple technologies and see not only what you need to do, but how you can

create them. You'll be able to see your way clear to develop the code that brings a project to life.

Rather than start with a complex system breaking it down into smaller pieces and work through them in the chapters of this book, we'll take a different approach. We'll start by building a more straightforward application to get comfortable with the challenges it presents. Then modify that application to incorporate new features. Each step will build on this knowledge to introduce one or more new abilities, techniques, modules, and solutions.

Python can take you to wonderful places. You only need to take those first steps. With that in mind, let's get started!

1.1 Who is the intended audience?

One of the stated goals of this book is to help you become a Pythonista. With that in mind, who is the intended reader of this book? I think there are a few broad types of readers who will benefit from the material presented here.

The first type of reader is someone who knows the basics of Python programming, how to write loops, conditional statements, and use variables to get things done. This reader has written utility programs that exist in a single code file and knows how to run them from the command line. They might want to build more complex applications, but don't understand how to do so. They could be thinking about writing a web server using Python, and what technologies they'd have to know to do that. Essentially this reader has a basic toolset, has hammered things together, and wants to build something more significant.

The second type is currently a developer in another language who is interested in broadening their skill set to include Python. They could because be curious, and Python is growing in popularity in both usage and interest. Or it could be out of necessity for a project at their current job. They know how to get things done in the language they currently know and want to learn how to do similar work with Python, and in a Pythonic way.

The third type could be someone involved with data science and big data research. Python is becoming one of the key players in this space, with many well-developed libraries serving the needs of this kind of work. This book won't get into the use of those libraries; that would be another book onto itself, -- but it will help readers involved in that space. Many who work in data science aren't necessarily software developers. Coding for them is a means to an end, a tool to help reach a goal. Learning to use that tool better, with greater expressiveness, by becoming a Python developer will take coding out of the problem domain, giving more room to focus on the task at hand.

1.2 Being a developer

Being a developer is an unusual pursuit. Developers spend time creating something out of nothing, and even then, it's hard to describe the thing we just created. Ever try explaining what development is at a party? Even if you're a good enough storyteller to keep people from wandering off immediately, it's still challenging to reach the "aha" moment where someone might know what you're talking about. And it's not a failing on the listener's part. It's just objectively hard to describe what developing software is.

Putting Artificial Intelligence aside for the moment, computers are exceptionally inert devices. Programs are a way for humans to express what we want a computer to do in a language we can read and write, and the computer can understand. The trick is being exact enough to get the computer to do what's intended, rather than something else.

People can function in the world and achieve great things because we thrive with the inexactness of human communication. We get meaning from context, intention, inflection, and subtlety - all of the things that provide great richness to our communication. None of that exists for a computer. They require an almost maddening exactness to function. The attention to minutiae to express that exactness, the patience to do so, and the ability to learn and stay open to new ideas -- are part and parcel of being a developer.

1.2.1 Solving Problems

There is a great deal of software available in the world: retail, commercial and industrial software, spreadsheets, databases, web applications, and tons of software that almost no one would ever know existed. All of this software exists to solve problems. And that's great. If there is an existing software solution to a problem you're trying to solve, absolutely use it and don't re-invent the wheel.

On the other hand, the world is filled with unique problems, and it is generating more all the time. These unique problems require custom software solutions. And this is the primary role of a developer. Whether you're solving a problem for your projects, or those of your employer, the goal of being a developer is to solve problems.

1.2.2 Process

Developing software involves a great deal of process and has steps that help lead to a successful outcome. Much like a classic story, it has a beginning, middle, and end.

On a personal level, it means how you approach a programming problem and break it down to produce a software solution. Being a developer includes the ability to recognize the boundaries of different parts of a problem and to understand how the technologies you know (or know about) can help decouple things along those boundaries. An example of this is knowing how to use a database to persist data and how to get data to and from that database to the user interface, which provides separation between the two.

If you work as a developer in an organization, there are defined processes that apply to your work. What coding standards, if any, does the team support? What kind of source code control model does the organization practice? Do you follow an Agile management process or some other methodology? How is software tested, approved, and deployed? All of these things and more make up the Software Development Life Cycle (SDLC). The process can help and hinder your development life, but it must exist in a team context and in the broader context of that team in an organization.

There is one other process that, in my experience, is worth mentioning and valuable to your success as a developer. Unless you're developing entirely on your own, you'll be working with other people. The list of people you may be working with can be pretty long and includes designers, accountants, managers, customers, both internal and external), clients, contractors, and more. Being able to communicate well across disciplines is an invaluable skill. The written word (email, wikis, messaging) is a critical conduit of information

and deserves your attention. Think about how you present yourself personally when talking to people; connections and relationships are essential to a better process. Not only will good communication skills immeasurably improve the information, help, and support flowing between groups, but it just makes your life as a developer more enjoyable.

1.2.3 Your Customers

Even if you're writing software only for yourself, you have a customer, you. All software has a user or users. Even software that's only ever used by another piece of software has a customer. The customer experience you create has a significant impact on the success or failure of the programs you write. It's not a revelation that a great deal of the success of Apple products has to do with the customer experience those products provide.

If you write a command-line program that works and gets the job done, and other developers can use it, that might be good enough. However, you shouldn't be surprised if a user in customer service dislikes the command line experience. If you're creating software for public consumption, an Internet web application with a user interface, for example, bad customer experiences can make people leave and possibly never return. Studies show patience for bad customer experience is approaching zero.

Even when you are creating a library module intended only for other developers, if the interface isn't well thought out and reasonable, you're going to hear about it. And if those users are sitting next to you in the office, you're going to hear about it forever.

From a high-level point of view, any single problem you're trying to solve with software means you have two problems. One is the actual work you're trying to capture in code. The other is the interaction, or user interface, to get that work done.

It's challenging to put yourself in the place of your customer when designing software. You'll be well acquainted with how you've created the application in the way you intended, rather than how a new user will use the application. It's worth thinking about how someone else will use what you've written. It's also advantageous to talk with your customers, if possible, to learn what they want and expect, and listen to their feedback about what you've created. Adopting an iterative process and mindset about development design and implementation can make a great deal of difference in keeping a customer or losing them.

1.2.4 Commitment to Learning

Learning how to get better at technology and developing with Python is a valuable skill. Working to improve yourself as a Python developer has two benefits. The first being able to take on larger projects with the confidence you can complete them and create a working system. The second is the practice of learning. Being a lifetime learner isn't just a catchy educational phrase; it's the reality of being a software developer.

For example, during my career as a developer, I've worked in several languages; Fortran, Pascal, C/C++, PHP, and now Python and JavaScript. Some of those languages I learned because that was what was in use where I was working. In other cases, the language was well suited to the task at hand. I considered myself a strong C/C++ programmer and enjoyed working on the applications I wrote with it. However, I don't have an interest in dusting off my C/C++ skills and doing that kind of coding again. Right now, for me, Python is the sweet spot as a language I want to use. It appeals to my desire to work in an object-

oriented programming style but doesn't limit me to only that. Python's syntax and grammar are clear and expressive enough that I can think about solutions in pseudocode that closely resembles Python code that would almost it's

If software development is your vocation, or you want it to be, keep in mind a career is long, and changes happen continuously. Committing to learning new technology and languages is the answer to both those things. In this rapidly changing world, there is very little job security, the only real security are the skills you can bring to the table.

1.3 Reaching goals

This book has some goals, one of which is implied in the title, helping you become a stronger developer. If you're reading this book, then I think that's a goal you have as well.

1.3.1 Thinking like a developer

Learning a programming language means learning the syntax and grammar of that language: how to create variables, build loops, make decisions, and execute program statements. These are your basic tools, but thinking like a developer also means knowing how to put those tools together to create a useful program. The analogy goes much further towards building bigger and more powerful tools.

This process of seeing how to use smaller tools to build bigger tools is key to thinking like a developer. The steps of creating one thing from other things eventually help you see the big picture. As you learn how to construct more powerful blocks of code, seeing the big picture as a developer means understanding the problem you're trying to solve and mentally being able to travel back and forth along with steps you can take to implement a solution. From the smallest block of code to more extensive functionality, you'll be able to follow the path to success.

1.3.2 Building Applications

In developer terms, an application is a complete program providing useful functionality and a user interface. An obvious one I'm sure you know already is Microsoft Word, a big desktop application. Google's Gmail is a big web application. These are both examples of large applications that provide many features with a great deal of functionality.

There are many smaller applications; for example, if you're familiar with the command line available on most computer systems, you may have used the *ping* command. This application is often used to determine if another computer on a network is responding to a '*ping*' request. Using *ping* is a simple troubleshooting test to see if the remote computer is running at all before digging further into any problem that might exist with it.

The *ping* application is pretty much on the other end of the spectrum from applications like Word or Gmail, but it is a complete application in its own right. It provides a useful function, and it has a user interface from the command line in a terminal window.

There are other blocks of code that developers work on, and these are libraries of code. They provide useful functionality and have interfaces, but for the most part, they are used by a larger application that wants access to the contained functionality. The standard modules

that come with Python, a feature commonly referred to as "batteries included," is an excellent example of library code.

Libraries provide an Application Programmers Interface (API), which defines how to use, or 'call', the functionality provided by the library. The API can be called directly, as in the case of importing and using a standard library module in Python. They can also be called remotely over a network. Examples of this would be an application that makes calls to a database server, or calling an HTTP API interface, like that provided by Twitter.

You'll be creating library modules to use in the applications we'll develop as we move forward through this book. We'll also be creating applications that provide an HTTP API interface as well as their main functionality.

1.4 Using Python

For the most part, everything you've read up till now about thinking like a developer could apply to just about any programming language. What makes Python an excellent choice to pursue thinking like a developer? As I mentioned in the section about reaching goals, I believe Python provides a sweet spot for application development. Let's talk about why I think that, and I hope you come to feel the same way.

If you're already on board with Python, feel free to jump to section 1.5 of this chapter, which introduces what version of Python this book will use.

1.4.1 The Syntax

When first introduced to Python, people often have a strong opinion about the language syntax. They either think that it's good or that it's not. I find the language syntax to be terse but wonderfully expressive. The ability to code functionality almost as fast as you can type minimizes the cognitive load of programming. It also allows you to continue to the big picture, rather than getting tangled in syntactical requirements.

But let's talk about one thing you might have heard about Python in a negative context: the required whitespace. In Python, a space is required after the 'for' keyword and before the iteration variable. It's also required after the 'if' keyword and before the conditional statement that follows. This use of whitespace replaces the use of parenthesis in other languages.

The start of a block of code is indicated by the ':' character at the end of a program statement line. Every line after the ':' that's indented with whitespace is intended to be in the block. The next outdented line after the block indicates the block of code has ended. The whitespace used to create the indented lines within the block are significant to the language and therefore required.

The whitespace required as part Python's syntax causes consternation among developers who use other languages, particularly languages that derive some of their syntax and structure from the C language. In those languages, the beginning of a block of code is denoted by the use of a '{' character. This is followed by the program lines intended to be in the block, and the block is ended, or closed, by the use of a '}' character. This is known as curly brace notation. Many popular programming languages use curly brace notation, like C, C++, Java, JavaScript, PHP, and more.

Those languages are also 'free-form' because no particular coding style is necessary. JavaScript takes advantage of this to pack code files by removing all unnecessary whitespace to reduce the number of bytes transmitted over a network to a browser. Because these languages are free form, there has been much debate over what is the right way to style code.

- Do curly braces stand alone on their own line or part of another line?
- How many spaces should be used to indent a code block for readability: 2, 4, 8?
- Should space characters be used after a 'for' or an 'if' keyword for readability or not?
- Should the code in a block maintain indentation at all?
- And quite a few more

As a developer, you should adopt a consistent coding style. If you work in a team, it's beneficial for the team to agree on and choose a coding style as well. Code is read far more than it's written, and having a consistent style reduces the white noise of reading code in a different style than what you've read before.

Code oriented editors are aware of the syntax of the language you're writing within them and will do what they can to automate the writing process. For example, if you're coding in a C derived language and type the opening curly brace character '{', followed by a carriage return, the editor will most likely auto-indent for you. It will continue to do so for all the lines of code you write after that until you type the closing curly brace character '}', followed by a carriage return. At which point, the editor will automatically outdent for you. Many editors give the user options to change the style you'd like to use, but whatever you choose, the style will be consistent within that editor for that language.

It's at this point where the argument against Python's required whitespace gets silly. If a coding style is beneficial to developers to read and write programs, and code editors automatically provide assistance to write with a coding style, what was the objection to Python's use of whitespace again?

1.4.2 Variables

Variables in Python are dynamically typed. What does this mean? A variable is declared in interactive Python with a line like this:

```
>>> x = 10
```

The variable 'x' is created and bound to an integer object whose value is 10. The type of variable 'x' is inferred at runtime from the object to which it is bound. As an example, these Python statements:

```
>>> x = 10
>>> type(x)
<class 'int'>
>>>
```

Create the variable 'x' and bind it to an integer object with a value of 10. The `type(x)` function call returns the type of the variable 'x', which tells you it's an integer.

Being dynamically typed also means the type of a variable can be changed at runtime by binding it to a differently typed object. This means program statements like the following are perfectly fine:

```
>>> x = 10
>>> print(x)
10
>>> x = 'Python'
>>> print(x)
Python
```

Here the variable 'x' is created with a type of integer by binding it to an integer object with a value of 10. Then the `print(x)` statement prints out the value object x is bound to, which is the integer 10. The type of the x variable is changed to String when it is re-bound to a string object with the value of 'Python'. Finally, the `print(x)` statement prints the value of the object bound to x, which is the string 'Python'.

In statically typed languages like Java, the type information (integer, string, float, etc.) is bound to the variable itself. For example, declaring an integer variable 'x' in Java looks like this:

```
int x = 10;
```

This indicates to the Java compiler the variable 'x' is of type integer, and to assign the value 10 to it. This also means no type other than an integer can ever be assigned to the variable 'x'. For example, this:

```
x = 'Java';
```

will generate an error at compile time in Java. Developers who use statically typed languages sometimes deride Python because of its dynamically typed variables. And to a degree, they have a point, binding a variable declaration to the type at compile time does catch a certain class of error conditions. It's something to be aware of when using Python. However, I've been writing Python code for a long time, and the dynamically typed nature of Python variables has rarely caused a problem.

1.4.3 Programming Paradigms

Most, if not all, of the languages in everyday use draw their abilities from other languages and programming paradigms. Python is a member of this club in good standing. If you've programmed at all in Python, you know it's a flexible language that covers a lot of ground. Part of the flexibility of the language is the many ways in which you can work with it:

- Structured programming where control flow is provided by loops, nested loops, conditional evaluation, and procedure calls.
- Python is a procedural language in that you can create functions (procedures), allowing you to create blocks of code that can be re-used in other parts of your program.
- You can code using class-based Object-Oriented Programming (OOP), which captures state information along with the blocks of code that operate on that state.
- Python, though not strictly a functional language, provides features that allow you to

program in a functional way. And since functions in Python are first-class objects and can be passed to other functions, this becomes useful when working in a functional style.

- Event-driven programs, like a windowing GUI application, where events determine the program control flow, is entirely possible with Python.

Any and all of these paradigms can be brought to bear using Python to solve programming problems and create applications.

1.4.4 Creating Maintainable Code

When you create an application, you expect it will be used, which means it will have a lifetime. During that lifetime, bugs will manifest themselves in your code, and testing doesn't always reveal every potential problem. Even if you're the only user of the application, a change in how you use it, or the environment in which you do so, could well reveal problems you can resolve and improve.

If other people use your application, what's required of it will change. Changing requirements mean changes will need to be made to the existing code to add new features.

Nothing in the software development world is more constant or happening faster than change. When I said earlier that code is read more than it is written now has meaning. If you come back to your code after a surprisingly short amount of time has passed, it will amaze you how much you have to read of your work to get back into the context in which it was created. If you work in a team, and someone else in the team will be modifying your work, that person will bless or curse you based on how maintainable and readable your code is.

Writing maintainable code is a developer strength worth pursuing. Adopting a coding style and consistently using it goes a long way towards this goal. Using intelligent and meaningful variable names, function names, and class names are important. I'm a firm believer that no programming language, even Python, is completely self-documenting. Comments that clarify the intention of a section of code can go a long way towards understanding the code's purpose and intent.

Another important aspect of writing maintainable code is making it flexible. It's difficult to anticipate how the functions and classes you create might be used further on in the development of an application.

A simplistic example would be a function performing some complex calculation, formatting the results, and then outputting those formatted results to standard output. The future use of that function is severely limited to how it's currently implemented and likely can't be re-used for anything else. If explaining what a function does has an "and" in the explanation, it's should be implemented as two functions.

Re-factoring this example leads to a function that does the complex calculation and returns the raw results. Then the function can be used later in the application to format and output the results to a destination device. By leaving the formatting and output until it's needed, the output can be directed to any device: a display screen, a web page, a printer, or perhaps a response to an API call. The underlying calculation function remains unchanged.

1.4.5 The Language Community

The most popular programming languages currently in use have large and active communities of people who are willing to share their experiences, problems, and expertise with others. Python has a particularly welcoming community with a minimum of flame wars or bullying. The community is a valuable resource for the newcomer to programming, as well as old hands who are working out new problems.

Beyond searching Google for Python help, here is a list of some of the available, useful Python resources:

- <https://realpython.com> – Real Python is an excellent source of tutorials about Python
- <https://pythonbytes.fm> – a Python podcast delivering interesting headlines and banter
- <https://talkpython.fm> – The Talk Python To Me podcast has interviews with people and personalities in the community.
- <https://pythonpodcast.com> – Podcast.__init__ is another good interview podcast
- <https://testandcode.com> – Test and Code, a podcast about software testing and Python
- <https://www.pythonweekly.com> – the sign-up page for a weekly Python newsletter containing links to useful articles and information
- <https://pycoders.com> – The sign-up page for another great Python newsletter

1.4.6 The Developer Tooling

As a developer, one of your goals is to get your thoughts and ideas from your mind and into a Python code file with as few impediments as possible. A good keyboard that works for you, proper lighting, a decent screen; all of these things contribute to the flow of the work you're trying to do. A powerful editing tool is also essential. There are many good editors out there that recognize Python code and syntax highlight it as you write, making it easier to find errors and keywords.

Notice that I said "editing tool" above; this was intentional. A good editor is essential, but beyond that, a good IDE is even more so. An IDE, or Integrated Development Environment, is a big step up from an editor regarding productively writing code. Not only will it have a good editor with syntax highlighting, but it will have knowledge of the language itself. This gives you additional assistance when writing code, which is commonly called IntelliSense. Intellisense provides code completion aide interactively, refactoring of existing code, symbolic name information and usage, and much more.

One last thing a good IDE should provide is a debugger. A debugger allows you to run a program interactively and set breakpoints. A breakpoint is a marker you can set on a program line where the code will stop running when it attempts to execute that line. While the program is paused, you can examine the variables that are within the current scope and see what the program is doing at that point. You can even modify the value of a variable, which will affect the execution from that point forward. You can single-step through the code from the breakpoint following the behavior of the program on a line by line basis. You'll be able to step into a function call and follow the behavior within it.

Being able to debug a program is an invaluable tool and skill to have at your disposal. It goes far beyond inserting `print()` statements in your code to try and glean what's happening inside. Python has mature and powerful IDE's available to it.

Visual Studio Code by Microsoft is an advanced source code editor that has extensions making it a complete IDE for Python. With the Microsoft extension for Python installed, it's not only an excellent editing environment with syntax highlighting and IntelliSense, but it's also a very powerful debugger. It's available across Windows, Mac, and Linux platforms, which is a win if you work on multiple different computers. It's also free to download and use, which is why we'll be using it in this book when working with the example code.

PyCharm is one of the suites of development tools provided by JetBrains and is a commercial Python IDE. It also has syntax highlighting, IntelliSense, and a powerful debugger, as well as tools to integrate with databases and source code control systems. It's a powerful tool for the development of Python code and applications and runs on Windows, Mac, and Linux.

WingIDE is yet another powerful, commercial Python IDE with syntax highlighting, IntelliSense, and has an advanced debugger with features useful in data science work. This platform is also available for Windows, Mac, and Linux.

1.4.7 Performance

The runtime performance of any programming language is an often debated, highly charged, and complex topic. Python is often compared to other languages, like C or Java, regarding execution performance. Beyond blanket statements about this or that being faster, it's here those comparisons get more complicated.

What's being compared, CPU speed, or memory speed? How is it being measured? Is the benchmark software optimized for one language, but not the other? Does the benchmark make the most of efficient coding practices in both languages being compared?

At the risk of sounding flippant, I don't care very much about any of this. It's not that I don't care about performance (we'll get to that), but the argument about this language being faster than that language is not worth engaging.

Computers are well past the point in time where CPU cycle and memory access times are worth considering very much in any performance calculation. To steal a business idiom:

Optimize your most expensive resource

You are the most expensive resource. And if you work for a company as a software developer, you are the most expensive resource connected to their computer resources. Optimizing your performance as a developer is paramount, and if you can transpose the big picture in your mind directly into pseudocode that practically runs, you have become invaluable. This is where Python shines, to borrow part of an old joke, "Python is executable pseudocode...". If you can express an idea into code and get it up and running faster and improve time to market, that is a huge win.

All of this is not to say I don't care about performance. When I first got into programming, I was obsessed with speed and would go to great lengths to shave CPU cycles from my code. Along the way, I learned a lot about what's important and what's not.

The first thing you should do before beginning any optimization effort is to determine if it's necessary at all. Is there a speed requirement your application needs to meet? If so, does

a metric exist to measure your application defining when it is fast enough? If the answer to these questions determines your application is fast enough, then you have struck upon the ultimate in terms of time spent optimizing, zero.

On the other hand, if it's determined your application does need to be faster, then you need to take the second step. This second step is to profile the application to measure where it is spending time. Python comes with a good profiler, and the IDE's I mentioned above help you use it.

With this measurement in hand, you can apply the 90/10 rule of code optimization. The rule states that 90% of an application's execution time is spent in 10% of the code. This rule is a generalization, to be sure, but it does provide a roadmap where you should pursue optimization. Spending time in anything other than the 10% is time poorly spent that won't improve the overall speed of your application.

Any optimization work needs to be done iteratively and in tandem with profiling. This tells you if your optimizations are making improvements or not. It will also help you decide if the improvements you've made are incrementally or orders of magnitude better. Small gains in performance need to be balanced against the complexity of the code. Optimization efforts often lead to more complex implementations, which can make code maintenance and feature improvement harder. Incremental optimizations that bring more complexity should be looked at carefully. Consider taking a broader view of the problem and think about changing the algorithm or data structure instead. These kinds of changes can lead to more significant gains in performance.

Lastly, know when to quit. With a target of what performance metric your application has to meet, you'll know when to stop optimizing and ship. Shipping is a feature that can't be overstated.

1.5 Version of Python to use

The code in this book is based on Python version 3.8.0. If you are relatively new to Python, you might know there are two major branches of Python versions in existence, 2.* and 3.*. The 3.* branch has been around for a long time, since December of 2008. It took a while for the 3.* version to gain traction with users because libraries and frameworks those users depended on weren't compatible with this branch, so they stayed with the 2.* branch. The time when that was true is well behind us, and there is no legitimate reason to start new Python projects in anything other than the 3.* version.

From this point in time forward, the Python 3.* branch will have newer features, the latest syntax, and more developer support. It also means important libraries and frameworks are dropping support for the 2.* branch. This implies developing programs with Python 2.* will have to pin the use of those libraries and frameworks to older versions that will no longer get new features or bug fixes. This last item is particularly important in relation to security concerns.

In addition, the Python 2.* branch reached EOL (End Of Life) happened on January 1, 2020. This means the core Python development team will stop supporting that branch completely. This clean break by the core developers frees them from some compromises made to continue supporting the 2.* branch.

Lastly, and I think this is very important, the Python community at large has moved to the 3.* version. This means documentation, articles, books, questions, and answers on forums will leave the old version behind and focus more on the new version(s). As a developer, this is an important issue. Firstly because everything you need to know as a developer is too big to have in mind all at once. This makes finding relevant information of paramount importance. And second, the pace of change is ongoing and rapid, which makes trying to know it all an exercise in futility. It's much more useful as a developer to understand what you need and want to do and then be able to find how it is done. Hands down, this beats being a catalog of facts that become outdated almost as fast as you learn them.

1.6 Summary

This chapter has introduced you to information about Python and why you'd want to use it on your journey to become a developer. You've seen why Python is an excellent choice of programming language to learn and work in as a developer. I hope this has whet your appetite to take the next step in your development journey.

2

Your Python Environment

This chapter covers

- **Building a development environment**
- **Python virtual environments**
- **Setting up Visual Studio Code**

As a Python developer, you need an environment on your PC in which to work. This developer environment covers a lot of ground. In general, it includes a set of tools you're familiar with and some ideas about how you will structure project directories. Depending on your computer, it might also include environment variables you've set to help make things more automatic and useful.

You'll find what you're about to create and configure is a good starting point towards building your own useful and powerful development environment. For those of you who already have a development environment you're comfortable with, feel free to skim or move to the next chapter. However, be aware of the differences presented here, you'll need to account for them in your environment.

2.1 Installing Python

First, you're going to need Python installed on your PC. This might seem obvious, but it's a little more involved than it might first appear. Not all operating systems come with Python installed, and sometimes when Python is installed, it's an older version. Even if the version of Python already installed on your PC is recent, it's still a good idea to install Python for your use while developing.

If Python is installed along with the operating system, there's a good chance it's being used for system-level functions by the operating system. If that's the case, there's a good chance your operating system is dependent on the version of Python it came with, and any modules that were installed for that version to use.

The programming examples in this book will ask you to install additional third-party libraries using the Python package management tool pip. It's a bad idea to install these other libraries in your system installed Python. This is because a newly installed version of a library might change the functionality of an existing system requirement and break it.

You also don't want updates to the operating system to change the Python functionality you depend on for your development efforts.

With that in mind, you'll install version 3.8.0 of Python that will be distinct from any operating system installed version. The newly installed version of Python is entirely under your control and independent of the operating system. This means you can add, remove, and update library modules as you see fit, and only your program code will be affected.

You'll install the stable 3.8.0 version of Python so you'll have the same version the example programs in this book were built and tested against to minimize runtime problems.

2.1.1 Windows

Depending on when you're reading this book, Python may or may not be installed by default on Windows. The Windows operating system isn't known to use Python, so once Python is installed, it is available solely for your development work.

Windows 10 versions 1903 and later make Python available in the Microsoft Store, which makes installation very easy. The Microsoft Store version of Python at the time of this writing is being evaluated, and not all features are guaranteed to be stable. This doesn't mean you shouldn't use it, just that you should be aware of these issues.

If you prefer, use the stable 3.8.0 version of Python available for your version of Windows by navigating to <http://www.python.org> in your browser and following the Downloads links. For most users, the straightforward version to use is the executable installer suitable for your CPU and OS version.

During the installation process, the installer allows you to check a box that adds Python to your PATH environment variable. Check this box and save yourself some trouble later. If you miss this and Python doesn't run from within PowerShell at the command prompt, you can always re-run the installer and add Python to the path.

2.1.2 Mac

On the Mac, there is currently an older version of Python installed, which you should avoid using for your development. Instead, install a recent version of Python that's completely separate from the system. To perform this installation, use the `pyenv` utility program. This program lets you install as many versions of Python as you'd like, and switch between them. For your purposes, you'll install Python version 3.8.0.

You'll need to have the Homebrew program installed on your Mac to follow the next steps. Homebrew is a package manager for the Mac OS you can use to install many useful command-line tools, like `pyenv`. The Homebrew program and its installation instructions are available here, <https://brew.sh>. Open your terminal program and follow these command line steps to install `pyenv`:

1. Run the command: `brew install pyenv`

2. Enter this line into your shell configuration file (.zshrc for me, could be .bash_profile for you):

a) `echo -e 'if command -v pyenv 1>/dev/null 2>&1; then\n eval "$(pyenv init -)"\nfi' >> ~/.zshrc`

3. Run the command: `exec "$SHELL"`
4. Run the command: `pyenv install 3.8.0`
5. Run the command: `pyenv versions`

Line 1 installs the `pyenv` utility program on your Mac.

Line 2 adds useful setup information to your terminal configuration file to make using `pyenv` easier.

Line 3 re-runs your shell creation scripts, making the commands in line 2 active.

Line 4 installs Python version 3.8.0 onto your Mac in your home folder in the `.pyenv` folder.

Line 5 shows the versions of Python installed, and if this is the first time using `pyenv`, this will show the system version and the 3.8.0 version you just installed.

2.1.3 Linux

There are so many versions of Linux in general use it would be awkward and outside the scope of this book to present `pyenv` installation instructions for all those versions. However, if you're using Linux as your development platform, you're probably already familiar with how to find what you need to install applications like `pyenv` on the Linux system you're using.

Even on Linux systems that have Python 3 installed, it's better to install and explicitly control the version of Python you're going to use for development. Once `pyenv` is installed, use it to install Python version 3.8.0 with the following command lines:

- `pyenv install 3.8.0`
- `pyenv versions`

The first command installed the version of Python you're going to use for the examples in this book in a directory controlled by `pyenv`. That version is kept separate from any other Python versions installed with `pyenv`, and the system version of Python. The second will list the versions installed, which should show the version you just installed.

2.2 Python Virtual Environment

Now that you have Python 3.8.0 installed on your computer, completely separate from any system installed Python, you might think you're ready to go. Even with a version of Python installed using the `pyenv` utility, you'll want to use another level of decoupling from that version in your projects.

Python 3.8.0 provides the built-in ability to create virtual environments. Python virtual environments are *not* like a virtual machine, which allows an entire OS to run as a guest inside a different OS. A Python virtual environment only creates a Python installation inside your project directory. This Python installation can have modules added to it with the `pip`

command, and these modules are only installed and available from the project-specific Python.

2.2.1 Windows

In a Windows system, you installed Python directly rather than using `pyenv`, which works fine as Windows doesn't use Python as part of its operation. You'll still want to use a project-specific Python virtual environment to keep your system-level Python installation separated from any modules you're going to install with `pip`.

To run the Python virtual environment activation and deactivation scripts, you'll need to change the Execution Policy of your computer. To do that, you'll need to run the PowerShell program as an administrator. To do this follow these steps:

1. Click on the Windows Start icon
2. Scroll down to the PowerShell menu selection and drop down the sub-menu
3. Right-click on the PowerShell sub-menu item
4. From that context menu select Run as Administrator
5. Once you're in PowerShell running as administrator, run this command:
6. `Set-ExecutionPolicy Unrestricted`

The system will prompt you with a question, which you answer with 'y' and then hit the return key. At this point, exit PowerShell so you're no longer in Administrator mode. You only need to do this once as it's a system-wide setting.

Open the PowerShell program again, not as an administrator, to get to a command line prompt and follow these steps to create a new Python virtual environment specific to the project directory:

1. Run the command `mkdir <project directory name>`
2. Run the command `cd <project directory name>`
3. Run the command `python -m venv .venv`
4. Run the command `.venv/Scripts/activate`
5. Run the command `python -m pip install --upgrade pip`

Line 1 creates a new project directory with whatever name you want to give it.

Line 2 changes your current working context into the newly-created directory.

Line 3 uses the `pyenv` installed Python to create the Python virtual environment in the `.venv` directory. This might take a few moments to complete.

Line 4 activates the virtual environment, prepending the command prompt with `(.venv)`, indicating the environment is active. Once the environment is active, any additional libraries installed will be installed in the `.venv` directory and won't affect the system-level Python you previously installed. To deactivate the virtual environment, just enter `deactivate` at the command prompt.

Line 5 is optional and upgrades the version of the `pip` command that exists within the Python virtual environment you've just set up. If `pip` detects you are running an older version, it will print out a message informing you that you're running an older version, and you should update that version. You can ignore the information and skip line 5 if you like, I included it because I ran the command to stop seeing that message.

2.2.2 Mac and Linux

Setting up a Python virtual environment on the Mac is straightforward if you've installed the Python 3.8.0 version using `pyenv`, as described previously. Open your terminal program and follow these steps to create a new Python virtual environment specific to the project directory:

1. Run the command `mkdir <project directory name>`
2. Run the command `cd <project directory name>`
3. Run the command `pyenv local 3.8.0`
4. Run the command `python -m venv .venv`
5. Run the command `source .venv/bin/activate`
6. Run the command `pip install --upgrade pip` [optional]

Line 1 creates a new project directory with whatever name you want to give it.

Line 2 changes your current working context into the newly-created directory.

Line 3 creates the local file `.python-version`, which `pyenv` uses to control what version of Python to run when you're working in this directory, in this case, the 3.8.0 version.

Line 4 uses the local Python version to create the Python virtual environment in the `.venv` directory.

Line 5 activates the virtual environment, prepending the command prompt with `(.venv)`, indicating the environment is active. Once the environment is active, any additional libraries installed will be installed in the `.venv` directory and won't affect the `pyenv` Python version previously installed. To deactivate the virtual environment, enter `deactivate` at the command prompt.

Line 6 is optional and upgrades the version of the `pip` command that exists within the Python virtual environment you've set up previously. If `pip` detects you are running an older version, it will print out a message informing you that you're running an older version, and you should update that version. You can ignore the information and skip line 6 if you like, I included it because I ran the command it to stop seeing that message.

2.3 Setting up Visual Studio Code

It's possible to write Python code using just a text editor application. Within the context of writing Python program code, a text editor is precisely that -- an editor that adds no more or less than what you type at the keyboard. The notepad application on Windows and the Textedit application on Mac are both examples of simple, capable text editors.

Microsoft Word, while a powerful editor, is not a good choice to create plain text, as its default is to save what you type with a great deal of additional information about formatting, font choices, etc., that has nothing to do with program code. Any word processor application makes writing plain Python code more complicated than using a text editor.

Python programs are just plain text files having the `.py` file extension. Many text editors understand a file with the `.py` extension is a Python file. These editors will provide whitespace indenting automatically when you hit the enter key at the end of a line of code that includes the block indicator character ``:``. They will also syntax highlight your code,

which means changing the color of the text to highlight Python keywords, strings, variables, and other visual cues about the program you're creating.

Microsoft provides a free code editor called Visual Studio Code (VSCode for short), and it is a great choice to use as a Python program editor. Besides being an excellent editor, it has many extensions available to make it an even better system to work with when developing Python. You'll install one of those extensions for Python that turns Visual Studio Code into a nearly complete Integrated Development Environment (IDE). This extension provides Python syntax highlighting and other language-specific features. The most powerful feature is the ability to run and debug your code interactively from within the editor. You'll get more into this as you progress through this book.

2.3.1 Installing Visual Studio Code

At the time of this book's publication, you can download Visual Studio Code by navigating to this URL: <https://code.visualstudio.com>. This link takes you to the Microsoft web page that has links to download VSCode for Windows, Mac, and Linux. Visual Studio Code is a separate application from Visual Studio, which is Microsoft's much larger and commercial application development system.

For Windows and Mac installation, the process is relatively simple: click on the link to download the installation program, then double-click on the downloaded file to install the application. For Linux installations, click on the link and, depending on your Linux version, choose the package manager that will run the installation process.

After installation add the VSCode application icon to the Windows taskbar, Mac dock, or the Linux desktop/application menu to make it more easily accessible.

2.3.2 Installing Python Extension

Once VSCode is installed, you'll need to add the Python extension from Microsoft. This extension is what provides the syntax highlighting, IntelliSense, debugging capabilities, and many other features. To do this follow these steps:

1. Open the VSCode application
2. Within VSCode open the Extensions
3. Click on the extensions icon
4. Select View -> Extensions
5. In the "Search Extensions in Marketplace" labeled text box, enter "python" and hit the return key.
6. The first item in the returned list will be the Python extension from Microsoft.
7. On that first item, click the install button.

At this point, VSCode is configured to work with Python files. Take some time and read the documentation about the Python extension in the right-hand windowpane of the application.

2.3.3 Other Useful Extensions

Besides the Python extension provided by Microsoft, there are other useful extensions available.

PYTHON DOCSTRING GENERATOR

This extension automatically generates Python docstring comments when you enter triple quotes (""") immediately after a function or method definition and hit the enter key. The docstring is a template containing all the parameters and return value in an easy to navigate manner, making documenting your code much easier.

CODE RUNNER

The extension makes it easier to run Python code files from right-click context menus within VSCode

DotENV

This extension adds syntax highlighting for .env files, which are local environment files useful to initialize environment variables outside of code files.

JINJA

This extension adds syntax highlighting to the Jinja templating language, which is the default for Flask web applications.

AREPL FOR PYTHON

This extension provides a window alongside the editing window that will evaluate the code you're writing as you write it. This can sometimes be useful.

2.3.4 Starting From the Command Line

VSCode is a powerful GUI application which can be started by double-clicking on its icon from your computer, or by clicking on its name/icon from a visual menu. This is a common use case for visual tools on the desktop but isn't the most helpful way to start the application for your purposes. Because all of this book's example code will use a Python virtual environment, it's useful to create a virtual environment before starting VSCode. It's also helpful to be able to start VSCode from the command line from within the project directory containing the virtual environment. Starting VSCode this way will help it 'see' and use the virtual environment in the directory. To configure VSCode to start from the command line in the directory you're in, do the following:

1. Windows
 - a) After installing VSCode, the system is already configured to open the current directory by typing "`code .`" from a PowerShell command prompt.
2. Mac
 - a) Start VSCode
 - b) Navigate to the Command Palette (View -> Command Palette)
 - c) Enter "shell command" to find the "Shell Command: Install 'code' command in PATH command"

- d) Click on the above
 - e) Exit VSCode
 - f) Start or re-start a terminal window and enter `"code ."` in a project folder with a Python virtual environment. The command will start VSCode using the same directory the terminal window is in as it's working directory.
3. Linux
- a) After installing VSCode, the system is already configured to open the current directory by typing `"code ."` from a terminal.

2.3.5 Starting a Project

With VSCode installed and configured to start from the command line, you can go through the steps of creating a project directory and starting VSCode to use it. Starting VSCode this way is used for all the example projects in this book, and in general, is an excellent way to create projects of your own. Follow these steps to create a new project:

1. Open a terminal or PowerShell and get to a command prompt
2. Change your directory to where you'd like to create your project
3. In that directory create a new directory `"mkdir <project name>"`
4. Change your directory to the newly created `<project name>` directory
5. For Mac and Linux, enter this command `pyenv local 3.8.0`
6. Enter this command `python -m venv .venv`

At this point, the project is set up to use the Python version 3.8.0 virtual environment installed in the `.venv` directory. The name `.venv` is commonly used as the directory name for a locally installed virtual environment.

You can activate your local Python environment and see it's working by following these steps:

1. Activate your Python virtual environment
 - a) Mac and Linux enter this command `source .venv/bin/activate`
 - b) Windows enter this command `.venv\Scripts\activate`
2. Your command prompt will be prepended with `(.venv)`
3. Enter this command `python --version`
4. The system will respond with `Python 3.8.0`

Now that your Python virtual environment is created in your project directory, VSCode will discover this when you start it from the command line. Entering `"code ."` from the command line will open VSCode in the project directory. Follow these steps to continue configuring VSCode to work with any Python code you create within your project:

1. Open the Command Palette (View -> Command Palette)
2. Type "Select Interpreter" in the text box and click "Python: Select Interpreter"
3. In the popup menu that appears, select the virtual environment you created
 - a) Windows - `.venv\Scripts\python.exe`
 - b) Mac and Linux - `.venv/bin/python`

4. Create a new file (File -> New File)
5. In the file editor window that's created, enter `print('hello world')`
6. Save the file as a Python file (File -> Save -> "test.py")
 - a) VSCode will pop up a prompt informing you, "Linter pylint is not installed". Click the "Install" button. VSCode will use the pip command from your virtual environment to install PyLinter. A linter is a pre-runtime tool that checks your code for syntax errors, bugs, unusual constructs, and more, and is a useful tool to have installed for every project.
7. Right-click on the "test.py" editor window and select "Run Python File in Terminal"

After the last step, you should see "hello world" printed in a terminal window opened within VSCode. Congratulations, you've just run your first Python program within your powerful project environment!

2.4 Some Advice

There's one more thing to emphasize as the most useful tool available to you, yourself. Invest time to optimize how you work as a developer. Having a suitable development environment you're familiar with is powerful, but setting up a productive personal working environment is time well spent. If you're going to develop software, for yourself, professionally or both, you're going to spend a considerable amount of time doing so. A reasonable desk, a comfortable chair, a good monitor, and a suitable keyboard are all part of that environment.

This last piece of advice is my opinion, based on years of working as a developer, both singly and in teams. It may not apply to you, so please disregard it if it doesn't. Spend time making the interface between you and your code as fast and seamless as possible, learn to touch type. My mom made me take typing classes in junior high back in the days of the IBM Selectric typewriter. I can tell you I was none too happy about it and didn't appreciate it for many years. Now it is one of the many gifts she gave me I am thankful for every day.

There are many things you'll write as a developer besides code: documents, wiki pages, web pages, presentations, notes, emails -- the list is long and only getting longer. Being able to get your thoughts through your fingers and onto the screen quickly and accurately gets the mechanics out of the way and lets your thoughts and ideas flow.

2.5 Summary

In this chapter, you've accomplished quite a lot. You've got Python installed in a way that won't interfere with any system python on your computer. You've learned how to create Python virtual environments and installed and configured a powerful editor for creating and running Python code. Knowing how to do these things will help you now and in the future to configure a toolchain on a computer and start developing.

In subsequent chapters, you'll use this development environment to build and run example programs to begin the process of learning new skills, how and where to apply them, and ultimately use what you've learned to create more complex and professional applications.

3

Names and Namespaces

This chapter covers

- Names
- Naming Conventions
- Namespaces

The names we give things and concepts help us navigate the world and communicate with everyone else who shares this world. The idea that names matter is even more important in the world of software development. Programming languages have keywords, grammar, and syntax that is generally a subset of a common, in-use language. In the case of Python, that language is English.

For programming language, this means we have keywords, grammar, and syntax to create programs that will eventually run. Naming things in those programs, however, is entirely within your control. You can use anything from the rich set of English words to name the things you create in a program. You can even use strings of nonsense characters if that suits you. But should you?

"There are only two hard things in Computer Science: cache invalidation and naming things."

The quote is attributed to Phil Karlton, a programmer with Netscape, the developers of the first widely used web browser. Putting aside cache invalidation, you might be thinking, "what's so hard about naming things." Let's find out.

3.1 Names

Back when I first started out writing code, I worked on a system based on Pascal. It was the first language I worked on allowing almost unlimited choice when it came to naming variables in the programs. One of the other young guys on the team created two global

variables he used to test for True and False. He named them "cool" and "uncool." At the time, we both thought this was pretty funny and made for some laughs when writing conditional statements and testing function return results.

Over time those variable names were all over the code losing their humorous quality and just became more challenging to think about and maintain. What was the meaning of "cool" and "uncool"? If you didn't know the actual value behind the symbol, was the meaning distinct, or could it be more of the English use of the words, which in many ways implied a range of meaning?

Giving a name to something is a way for you and Python to identify it. Usually, this means you want to identify something uniquely, so it's distinct from all the other named things in a program. For example, Social Security numbers in the United States are given to people so they can uniquely identify themselves within usage contexts in the country. This unique string of numbers helps people get employment, do their taxes, get insurance, all kinds of activities that require a nationally unique identifier.

Does this mean a Social Security number is a good name for a unique thing? Not really. Unless you have access to the systems that use the number, it's entirely opaque. It conveys no information, or meaning, about the thing it's identifying.

Let's take this idea of unique names to another level. There are identifiers called Universally Unique Identifier (UUID) and Globally Unique Identifiers (GUID), where GUID's are a variation of UUID's used mostly in the Microsoft world. If you've ever installed Microsoft Windows, you're familiar with the installation key, which is a GUID. Both are mechanisms to generate a sequence of characters that, for all practical purposes, are unique across the entire world. A sample UUID looks like this:

```
"5566d4a9-6c93-4ee6-b3b3-3a1ffa95d2ae"
```

You could generate UUID values and use them as variable names to uniquely identify everything in your programs. These variable names would be unique within your entire application and across the known world. Naming variables this way would also be a completely unusable naming convention. The UUID conveys absolutely no information about the thing it identifies, that's part of its purpose. They are often used to identify something and keep any other information about the identified thing opaque, like any sequencing information or hints about the set of information containing the UUID. They're also very long to type, impossible to remember, and unwieldy to use.

3.1.1 Naming Things

Naming things is not only about uniqueness but also about giving the named things meaning. Trying to provide the name you assign meaning, or an indication of how the thing is used adds embedded information that's very useful when developing Python programs. For example, naming a variable `t` versus `total`. You'd have to examine the context of the surrounding code to know what `t` is, whereas `total` draws from its context to include meaning.

Based on the UUID example above, the length of the name you give to something is also relevant to the effort of writing code. Programming does involve a lot of typing, which means the balance between meaning and brevity matters.

You're suddenly in a position where an entire language is your hunting ground for words and phrases to name things. Your goal is to find words that imply meaning and yet are short enough to not get in the way of writing, or reading, a line of program code. This constrains what you could or should do when naming things. Like a painter working from a limited palette of colors, you can choose to be frustrated, or get creative within that constraint and build something with artfulness and creativity.

Many of the programs you'll write will include looping over a collection of things, counting things, and adding things together. Here's an example of code iterating through a 2-dimensional table:

```
table = [[12, 11, 4], [3, 22, 105], [0, 47, 31]]
for i, row in enumerate(table):
    for j, column in enumerate(row):
        for item in column:
            process_item(i, j, item)
```

There's nothing wrong with this code; it's perfectly functional. The `table` variable consists of a Python list of lists, which represents a 2-dimensional table. The `process_item()` function needs to know the row and column position of the `item` within the table to process it correctly. The variables `i` and `j` are entirely serviceable but give the reader no information about their intent.

You might be inclined to think it's not a big deal for this example code, but imagine if there were many more lines of code between each invocation of the for loop. In that case, the declaration of the `i` and `j` variables are visually separated from their use. The reader would probably have to go back and find the declaration to give the variable meaning. Keep in mind the reader might well be you six months after writing this code when the meaning and intent is not so fresh in your mind. Here's a better implementation of the above code:

```
table = [[12, 11, 4], [3, 22, 105], [0, 47, 31]]
for row_index, row in enumerate(table):
    for column_index, column in enumerate(row):
        for item in column:
            process_item(row_index, column_index, item)
```

The code has changed, so `i` is now `row_index`, and `j` are now `column_index`. The variable names indicate what they contain and the meaning of their intended use. If the variable declarations in the for loops are separated from their use by many lines of code, the reader could still quickly deduce what the variable means and how to use them.

Another common operation in development is counting things and creating totals. Here are some simple examples:

```
total_employees = sum(employees)
total_parttime_employees = sum[
    employee for employee in employees if employee.part_time
]
total_managers = sum[
    employee for employee in employees if employee.manager
]
```

You can see a couple of pretty good naming conventions in the above example. The name `employees` give the variable meaning. The use of the plural `employees` indicates it's an iterable collection. It also shows the collection has one or more things inside it that would represent an employee.

The variable `employee` inside the list comprehension indicates it is a single item from within the `employees` collection.

The variables `total_employees`, `total_parttime_employees`, and `total_managers` indicate what they refer to by the use of "total" as part of their names. Each of them is a total value of something. The second part of each variable name indicates the thing being totaled.

Besides numerical calculations, you'll often deal with things that have names already, like people within a company, community, or group. When you're gathering user input or searching for someone by name, having a useful variable name makes it much easier to think about the thing you're representing in code.

```
full_name = "John George Smith"
```

Depending on the purpose of the code you're writing, this might be a perfectly acceptable variable name to represent a person by name. Often when working with people's names, you'll need more granularity and will want to represent the parts of the name.

```
first_name = "John"
middle_name = "George"
last_name = "Smith"
```

The above variable names also work well, and like `full_name`, give the variable names meaning about what they represent. Here's another variation:

```
fname = "John"
mname = "George"
lname = "Smith"
```

This version adopts a convention for how the variables are named. A convention like this means you're choosing a pattern to create the variable names of people. Using a convention means the reader has to know and understand the convention in use. The trade-off in the example above is less typing, but still clear variable name meaning. It also might be more visually appealing to you as the variable names line up vertically in a monospaced editing font.

Adopting conventions is one technique to be more creative and productive within the constraints of variable naming. If the above shorthand naming convention is more visually appealing to you, this lends itself to recognizing patterns and identifying typos, when visually parsing code.

3.1.2 Naming Experiment

You may not remember, but in the early days of personal computers, they had tiny hard drives. Early operating systems also had no concept of sub-directories; all the files on the hard drive existed in one global directory. Additionally, file names were limited to eight characters, plus a three-character extension, which was generally used to indicate what the file contained. Because of this, bizarre and complex file naming conventions were invented to maintain uniqueness and prevent file name collisions. These naming conventions came at the cost of logically meaningful filenames.

The solution to this problem was adding support to the operating system for named sub-directories and removing the filename character length limit. Everyone is familiar with this now, as you're able to create almost infinitely deep structures of directories and sub-directories.

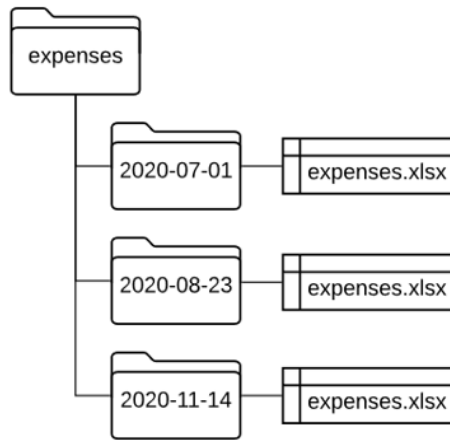
EXPERIMENT

Here's a specification you've been asked to meet. The accounting department where you work requires all expense reports to have the same filename: "expenses.xlsx". You need to create a directory structure where all your "expenses.xlsx" files can exist and not collide or overwrite each other to save and track these expense files.

The constraint is the requirement all expense report files have a fixed filename format. The implied constraint is that whatever directory structure you devise has to work for as many expense reports your work generates. The ability to create sub-directories is the tool you have to work with to solve this problem and keep the expense report files separated.

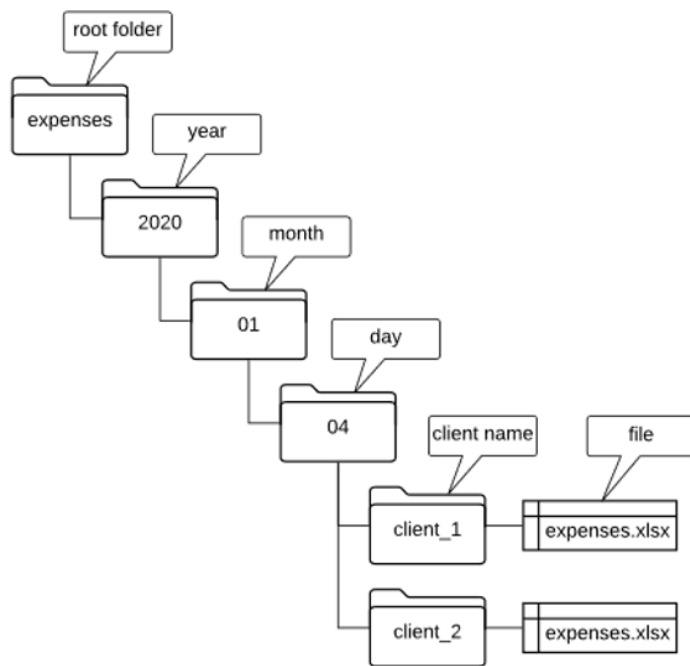
POSSIBLE SOLUTIONS

Any solution depends on how many expense reports you create to do your job. If you're working as a junior software developer, you might only travel a few times a year. In this case, you would only have to provide coarse granularity to keep your "expenses.xlsx" files separated. Something like this would be suitable:



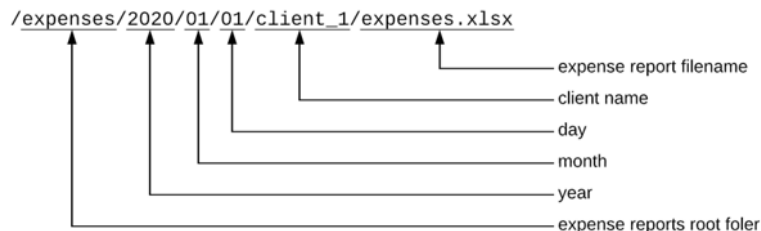
This simple structure gathers all the expense reports under a single root directory named "expenses". Each expense report exists in a directory named with the fully qualified date when the expense report was created. Using a date format of YYYY-MM-DD causes the directories to sort in a useful chronological order on many operating systems when displayed.

However, if you're a sales engineer, you're most likely traveling all the time and possibly meeting multiple clients per day. This constraint requires your directory structure to support much more granularity to keep all the "expenses.xlsx" files separate. Here's a possible solution for a sales engineer using the year, month, and day values as sub-directories:



The directory structure above breaks the dates into separate subdirectories based on year, month, day, and client name. Doing this allows you to keep the "expenses.xlsx" files distinct even when visiting multiple clients per day. This creates a convention that each part of the path to a particular "expenses.xlsx" file has meaning as well as a value.

It might not be evident based on the experiment above, but what you've created are variable names having meaning and convention. Look at the path to a particular expense report:



What you've done is to create namespaces, each one narrowing the scope of what it contains. Reading the path from left to right each segment of the path separated by the / character creates a new, narrower namespace within the context of the previous one.

Suppose you are the accountant who has mandated the file naming convention for expense reports. As the accountant, you'll have to save all the expense reports employees submit. You'd be under the same constraint as the employees who are generating the expense reports, but with the added complexity of keeping all the employees distinct and separated from each other.

Creating a directory structure to handle the added complexity could include higher-level abstractions of department and employee. Creating a directory structure providing this level of granularity to track and save all the employee expense reports is possible. Thinking about how to create the structure makes it clear it's time for the accounting department to rethink the requirement and constraints and design a better system.

3.2 Namespaces

A namespace creates an abstraction containing other named things, which can include other namespaces. The name of the city or town where you live is an example. The city name provides a namespace containing all the people who live in that city.

Going further, the streets and roads where people live all have names. The street and road names become a namespace within the city namespace. For example, there are many streets named "Main Street" throughout the United States. However, there is only one "Main Street" in Chicago.

This hierarchy of namespaces creates the convention that is a United States mailing address. The full address for John Smith, working at the Empire State Building might be something like this:

John Smith
 Empire State Building, Suite 87A
 20 W 34th Street
 New York, New York 10001

By convention, the mailing address namespace scope gets narrower reading from bottom to top. A software developer might remove redundant information and represent this address like the directory experiments above in a left to right form:

```
10001|20 W 34th Street|Empire State Building Suite 87A|John Smith
```

Here the city and state have been removed because the Zipcode contains that information. The namespace fields have been delimited with the '|' character because it doesn't appear in the addresses text. Continuing from left to right, you come to the final leaf node, the person the address applies too.

Like the directory structure experiment, reading from left to right, the scope of information contained within each distinct namespace gets narrower. Also, like the directory structure hierarchy, the position of each namespace follows a convention that gives each meaning.

3.3 Python Namespaces

The Python programming language provides the ability to create namespaces. Namespaces give you a great deal of power and control when handling the constraints of naming

variables, giving them meaning, keeping them relatively short, and avoiding collisions. You do this by placing variable names in namespaces.

Before you get to the point of creating namespaces of your own, let's look at the one provided by the language.

3.3.1 Builtins Level

When Python starts running, either in your program or at the command line, it creates a builtins namespace. builtins is the outermost namespace in Python and contains all of the functions you can access at any time. For example, the `print()` and `open()` functions exist in the builtins namespace.

You can see what's in the builtins namespace by entering this command at a Python interactive prompt:

```
>>> dir(__builtins__)
```

This command runs the `dir` (directory) command on the `__builtins__` object. You'll see all of the exceptions and functions listed that are available everywhere in Python.

You might not have thought about functions like `print()`, `open()`, and others as existing in a namespace, and you don't have to use them. The idea that they are in a namespace is useful as you learn more about creating namespaces and the scope of the objects within them.

There is something to keep in mind when working with the builtins namespace: it's entirely possible to overwrite an object in the namespace with something of your own. For example, you could define a function like this:

```
def open(...):
    # run some code here
```

Creating a function like this would be perfectly fine; however, the side-effect of doing this is shadowing the `open()` function already defined in the builtins namespace. It might make perfect sense for the program you're writing to name your function `open()`, but shadowing Python's `open()` function, and making it inaccessible, is probably not what you intended.

You can handle this by creating your function like this:

```
def my_own_open(...):
    # run some code here
```

The code works, but you've sacrificed brevity and simple meaning for uniqueness to avoid your function's name colliding with Python's `open()` function. Using namespaces provides a better solution.

3.3.2 Module Level

The Python program file you create that starts your program running is considered the entry point for your program as a whole. When it starts, the objects in the builtins namespace are created. In Python, everything is an object, variables, functions, lists, dictionaries, classes,

everything. Anything you create is also an object in the main program file and has the potential to collide with and overwrite the objects in builtins. You can, and should, avoid this.

Breaking up your program code into multiple files containing logically grouped functionality is a useful convention to adopt. Doing so has the following benefits:

- Keeps similar functionality together where it's easier to reason about it
- Prevents program files from becoming too long to edit and manage reasonably
- Creates namespaces

The last item creates namespaces, is key to how Python creates namespaces for your use. If you wanted to create two functions named `add()` having different behavior, and you created a `main.py` file that looked like this:

```
def add(a, b):
    return a + b

def add(a, b):
    return f'{a} {b}'

print(add(12, 12))
print(add(12, 12))
```

The program would run, but it wouldn't function the way you want. There's no way to indicate to the code above which `add()` function is being called in the `print(add(12, 12))` statement. When Python executes this code, it defines the first `add()` function, then immediately redefines it with the second, shadowing it and losing the first definition.

The behavior of the two functions is different, the first performs a mathematical addition on the two parameters, and the second performs a specialized "string" addition (concatenation) on the two parameters. However, as far as Python is concerned, the name of the function is the distinguishing feature. And since they are both defined in the same namespace, the second shadows the first and takes precedence.

To get both `add()` functions to exist, you need to create a namespace into which you can put one of the `add()` functions. To do this create a "utility.py" file that looks like this:

```
def add(a, b):
    return f'{a} {b}'
```

Then change your "main.py" file to look like this:

```
import utility

def add(a, b):
    return a + b

print(add(12, 12))
print(utility.add(12, 12))
```

When you run the "main.py" file, you get the intended output of:

```
24
12 12
```

Creating the "utility.py" file separates the two `add()` function definitions so they both can exist. In the "main.py" file, the `import utility` statement tells Python to pull all the objects in the "utility.py" file to a new namespace called `utility`.

Be aware the namespace created by importing a file is not tied to the filename, that's the default behavior. You can override this default behavior in this way:

```
import utility as utils
```

This statement tells Python to pull all the objects in the "utility.py" file into a namespace called `utils`. Being able to name the namespace specifically can be a useful feature if you want to replace a namespace with other objects that provide different functionality.

It's also possible to remove a namespace when importing functionality. Using your current "main.py" example, this is done like this:

```
from utility import *

def add(a, b):
    return a + b

print(add(12, 12))
print(utility.add(12, 12))
```

The code tells Python to pull all the objects in the "utility.py" file into the current namespace. This program now has an error in it because the `utility` namespace no longer exists, so the `print(utility.add(12, 12))` statement doesn't work. Removing `utility` from the statement makes the program work, but you're back to a variation of the original problem. The `add()` function defined in the "utility.py" file is shadowed by the `add()` function defined in the "main.py" file. For this reason, it's usually not a good idea to use the `from <filename> import *` form when importing files.

Being able to create namespaces based on files is useful, but Python's support goes much further. By capitalizing on the file system directory structure, you can create namespace hierarchies. Just like the previous directory structure naming experiment, this gives you more tools to create meaning and scope for the hierarchies you create.

If you take your example a little further, you might decide to get more specific with the functionality you're creating. The `utility.add()` function is specific to string handling, why not make that clearer?

Create a new directory called "utilities" in the same folder as your "main.py" file. Move the "utility.py" file to the "utilities" directory and rename it "strings.py". You now have a directory hierarchy that looks like this:

```
utilities/strings.py
```

This adds meaning just as the directory structure experiment does, "utilities" indicates that everything under the directory is considered a utility.

One thing to keep in mind when creating directory hierarchies to contain functionality is the need to create an `__init__.py` file. This file has to exist in each directory to let Python know the directory contains functionality or the path to it. The `__init__.py` file makes the directory it's in a Python package.

Often the `__init__.py` file is empty, but it doesn't have to be. Any code inside the file is executed whenever the path containing it is part of an import statement.

Based on this, create an `__init__.py` file in your "utilities" directory. Once that's done, modify your "main.py" file to look like this:

```
from utilities import strings

def add(a, b):
    return a + b

print(add(12, 12))
print(strings.add(12, 12))
```

The `from utilities import strings` statement tells Python to navigate to the utilities package and pull all the objects from the "strings.py" file into the `strings` namespace. The `print(strings.add(12, 12))` line has been changed to use the `strings` namespace to access the `add()` functionality. Now the namespace plus function name combine to increase the clarity and intention of the `add` function alone.

When you create a Python file that you intend to import into other parts of your program, it's common to think of the file as a module. The module contains functionality that's useful to your program. This idea is very much like the "Batteries Included" statement that's often associated with Python. Python comes with a large selection of standard modules you can import and use in your programs.

If you've used any of Python's standard modules, like `"sys"`, you might notice those standard modules don't exist in the working directory of your program like the "strings.py" module you've created above. Python searches for modules you want to import through a list of paths, the working directory being first.

If you start a Python interpreter and enter these statements at the prompt:

```
>>> import sys
>>> sys.path
```

You'll see output that looks something like this:

```
['', '/Users/dfarrell/.pyenv/versions/3.8.0/lib/python37.zip',
  '/Users/dfarrell/.pyenv/versions/3.8.0/lib/python3.8',
  '/Users/dfarrell/.pyenv/versions/3.8.0/lib/python3.8/lib-dynload',
  '/Users/dfarrell/tmp/sys_path_test/.venv/lib/python3.8/site-packages']
```

The output is the list of paths Python will search through when it runs across an "import" or "from" statement in your code. The above list is specific to my Mac, the listing you see will most likely be different depending on whether you're using a Windows or Mac computer and if you're running Python in a virtual environment.

The first element in the above list is an empty string. Python will look in the current working directory for modules. This is how it found the "utilities" package and the "strings" module in that package.

It also means that if you create a module and name it identically to a Python system module, Python will find your package first and use it, ignoring the system package. When naming your packages and modules, keep this in mind.

In our short example, the `"import sys"` statement causes Python to search the above list of paths. Since a `"sys"` module doesn't exist in your working directory, it looks in the other paths, where it does find the standard modules.

The list of paths above is used when you install a package or module with the `pip` command. The `pip` command will install the package in one of the paths from the list. It's recommended to use Python virtual environments to prevent `pip` from installing into your computer systems version of Python.

3.3.3 Function Level

There is another level of namespace control available to you. When you create a Python function, you are creating a namespace for variable name creation. The functions you create exist in a module, either the main Python file of your program or separate module files.

The module file creates a namespace, and any functions you create in the module exist within that namespace. What does this mean? Take your `"strings.py"` module and make the following changes to it:

```
preface = "added: "

def add(a, b):
    return f"{preface}{a} {b}"
```

The changes above create a variable named `preface` at the module level namespace and initialize it with the string, `"added: "`.

If you run your main program, you'll see the output of the `strings.add(12, 12)` now outputs `"added: 12 12"`. When the `add()` function is executed, Python looks for the `preface` variable inside the function namespace, and not finding one, looks at the module-level namespace. It finds the `preface` variable in the module and uses it in the string formatting returned by the function.

Change the `strings.py` code again and make it look like this:

```
preface = "added: "

def add(a, b):
    preface = "inside added: "
    return f"{preface}{a} {b}"
```

Inside the `add()` function, you've created a variable named `preface` and initialized it to a different string. If you rerun your code, you'll see the output of the `strings.add(12, 12)` function outputs `"inside added: 12 12"`.

What's happening here is Python now finds the `preface` variable in the `add()` function's namespace and uses it. Not only is the `preface` variable defined inside the `add()` function's namespace, it's created in the function's scope.

3.3.4 Namespace Scope

Names and namespaces are essential tools in your Python paint box and are related to another tool hinted at in the previous section. The scope of a variable is an important consideration when creating and using variables.

The scope of a variable relates to its accessibility and lifetime in a module, a Python file. In Python, a variable is created when a value is assigned to it:

```
prefix = "prefix:"
```

The statement above creates the variable `prefix` and assigns it the string value `"prefix:"`. The variable `prefix` also has a type of string, which Python determines at the time of assignment from the object being assigned, in this case, `"prefix:"`.

If you were to create a new module file named `"message.py"` that looked like this:

```
prefix = "prefix : "

def my_message(text):
    new_message = f"{prefix}{text}"
    return new_message
```

You have created things having different scopes and lifetimes. The `prefix` variable is in the global module scope. It is accessible anywhere within the `"message.py"` module. It also has the same lifetime as the module. If you `import message` into your code, the `prefix` variable and `my_message` function are around for as long as the message module is. It is still within the message namespace and would be accessible to programs that import it like this:

```
import message
print(message.prefix)
print(my_message("Hello World"))
```

The variables defined inside the `my_message(text)` function have function-level scope. This means they are only accessible within the function, and their lifetime is from the point of their creation when the function starts to execute to the end of the function statements.

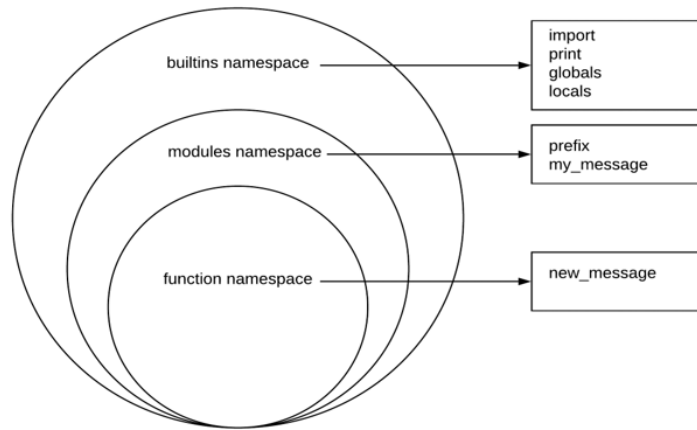
Because the `my_message(text)` function is contained within the module level scope, the code within the function can access the `prefix` variable.

At the module scope, what's declared at that level is accessible, `prefix`, and `my_message`. The `my_message` function is part of the module level (global) scope, but all the variables declared inside the function are not.

Inside the `my_message` function, the two variables `text` and `new_message` is accessible as they are in the local scope, but aren't accessible outside the function. The module variable `prefix` is in the global and is also accessible inside the function.

The program above shows scope is nested. Inner scopes have access to their enclosing scope, as demonstrated by the `my_message` function having access to the `prefix` variable. Outer scopes do not have access to the scopes they enclose.

Figure 1 below illustrates this nesting relationship:



3.3.5 Namespace Experiment

Using what you've learned about names and namespaces, try this experiment using the information to solve a problem. This problem is about using meaningful names and namespaces to solve an otherwise awkward development issue.

EXPERIMENT

You're the developer in charge of maintaining the software for an online community of users interested in similar things. The community wants the software to email them about upcoming events and to include non-community members who register to receive event emails. This latter group consists of potential new members who are interested but not yet committed to joining.

The software can send personalized emails to the mailing list of registered users, both members and non-members. When creating the personalized email, the current software calls a function `get_name(person)` to get the name to render into the email based on the person object passed to it.

The community wants to change how personalized email is rendered by creating a concept of "formal", "informal", and "casual" for the name. Email sent to non-members would always use the "formal" version. Email sent to members would base the name on the user's account settings and could use any of the three name versions.

This becomes your requirement: how can the logically named `get_name(person)` function return three different values for three different use cases?

POSSIBLE SOLUTION

One possible solution is to create three new versions of the `get_name(person)` function like this:

```
def get_name_formal(person):
def get_name_informal(person):
def get_name_casual(person):
```

The code above is understandable and direct but is awkward to make use of in the rest of the code currently calling `get_name(...)`. Using this approach requires you to modify every instance of a call to `get_name(...)` to be an `if/elif/else` conditional to call the correct function above. You'd also have to make the conditional information that selects the right function available for those `if/elif/else` condition tests.

Another approach would be to change the `get_name(person)` function to take an additional parameter that indicates how to format the response. Something like this would work:

```
def get_name(person, tone: str):
```

In the example above, the variable `tone` is a string theoretically set to "formal", "informal", or "casual". The value of `tone` would be used in an `if/elif/else` conditional statement to format the name in an expected manner.

The example above would work but is only a marginal improvement over the individually named functions. Choosing this method requires you to find and edit every instance where the `get_name(...)` function is called in the entire program and update it to include the new `tone` parameter. If there are many places where the function is used, this could become a maintenance headache.

The use of namespaces could provide a workable solution that doesn't disrupt the other parts of the software. Rather than change the name signature of the `get_name(person)` function, or changes its parameter list, you could use namespaces to create a solution.

As a contrived example here is a "main.py" program that simulates sending out an email to the community, before accounting for the required changes:

```
from utilities.names import get_name

# generate a community list of three of the same people
community = [{
    "title": "Mr.",
    "fname": "John",
    "lname": "Smith"
} for x in range(3)]

# iterate through the community sending emails
for person in community:
    # other code that calls get_name many times
    print(get_name(person))
```

The `get_name(person)` function from the "names.py" module might look like this:

```
def get_name(person):
    title = person.get("title", "")
    fname = person.get("fname", "")
    lname = person.get("lname", "")
    if title:
        name = f"{title} {fname} {lname}"
    else:
        name = f"{fname} {lname}"
    return name
```

This function looks at the `person` information, and depending on if the person has a title value or not, formats the name accordingly and returns it. The `get_name(person)` function is the formal version and can be used as-is.

The change requirements are to create a “formal”, “informal” and “casual” greeting for the emails based on the tone determined by the account. You already have a formal version of the `get_name(person)` function, and just need to create the informal and casual versions. Create a module file called “informal.py” in the utilities package directory that looks like this:

```
def get_name(person):
    fname = person.get("fname", "")
    lname = person.get("lname", "")
    name = f"{fname} {lname}"
    return name
```

This function concatenates the first and last name and leaves out the title. Create another module called “casual.py” in the utilities package directory that looks like this:

```
def get_name(person):
    fname = person.get("fname", "")
    name = f"{fname}"
    return name
```

This function returns the person's first name and nothing more.

Based on the change requirements, you also need to create a way to define the tone to use in the email based on the community members' account information. The information to examine is whether or not they are a member, and if they are a member, what is the greeting setting in the account.

For this experiment, you can create an “account.py” module in the utilities package directory. The “account.py” module contains this code:

```
from random import choice

def get_tone(person):
    return choice(["formal", "informal", "casual"])
```

This code returns a randomly selected value from the list of tone strings, “formal”, “informal”, and “casual”.

Now you have everything you need to meet the requirements and change how the mailing list is processed. Here's an updated listing of the “main.py” program showing how the namespaces you've created are used:

```

from utilities import names
from utilities import informal
from utilities import casual
from utilities import account

community = [{
    "title": "Mr.",
    "fname": "John",
    "lname": "Smith"
} for x in range(3)]

for person in community:
    tone = account.get_tone(person)
    if tone == "formal":
        get_name = names.get_name
    elif tone == "informal":
        get_name = informal.get_name
    elif tone == "casual":
        get_name = casual.get_name
    else:
        get_name = names.get_name

    # other code that calls get_name many times
    print(get_name(person))

```

This version of the "main.py" program imports the three new modules, `informal`, `casual`, and `account`. At the top of the community iteration, the tone is retrieved based on the person passed to `account.get_tone(person)` function call. The `tone` variable is used in an `if/elif/else` set of statements to set the `get_name` variable.

Notice the `get_name` variable is set to the `get_name` function from a specific module depending on the value of `tone`. The code sets the `get_name` variable to refer to a function, not call the function. Now that `get_name` is a function object; it can be used just like a function in the `print(get_name(person))` statement.

The `get_name(person)` function call will do the right thing because it refers to the desired modules `get_name(person)` function at each iteration when the `tone` variable is set.

All of this work upfront to create the modules, and the code within them was done to avoid losing a good logical name like the `get_name` function and allow it to be used unchanged anywhere else within the program. The work also prevents name conflicts through the use of namespaces.

3.4 Summary

You've learned a lot in this chapter about the names we give things, and how namespaces can help us collect those things into meaningful groups. There is no shortage of symbols you can use to name things; using good ones gives you benefits that pay off the larger and more complex the applications you build become.

The rest of this book will build on the concepts of logical names and namespaces to make the programs you develop easier to follow and manage.

4

Application Programming Interface

This chapter covers

- **What Is An Application Programming Interface (API)**
- **What Is A Well Designed API**
- **How To Create Good APIs**

Most people who have worked with a computer system are aware of something called the Graphical User Interface, or GUI. They have been around so long the term has almost become indistinguishable from the computer system itself, like Windows, Apple, iPhone, or Android. The operating system and the on-screen interface a person use have become synonymous.

The distinction between the operating system and the GUI isn't essential for most users, although it does exist. The GUI is all of the visual elements displayed on the screen that presents information to the user as a result of actions, and all the visual elements that let the user interact with and change the project the visual elements represent. Part of this interface encompasses all the devices that accept input from the user. These include keyboards, computer mice, trackballs, microphones, joysticks, virtual reality headsets, and an ever-growing list of other devices that can gather input from the user.

The GUI, and all of the attached devices, is the interface between the application program and the human user. The user doesn't have to know how to modify memory, save files, update databases, or get information from the network. The GUI abstracts those details away and presents a consistent, meaningful, and reasonable interface to the user.

4.1 Application Programmers Interface

An Application Programming Interface provides the same abstraction. However, it does so, not between a person and a computer, but between sections of programming code. All programs are composed of custom code, modules, and libraries of existing code. Even the

most straightforward Python program performing a `print("Hello World")` is using standard library code provided with the Python language. Having library code is a huge advantage allowing you to focus on what you're trying to accomplish rather than coding everything yourself. Imagine having to write a print function every time you started a new project or having to create something more complex like network access.

Python is well known as having "batteries included," meaning it comes with an extensive and powerful standard library of modules providing all kinds of functionality you can use and not have to create yourself. There's also a large selection of modules you can install from the Python Package Index (<https://pypi.org/>) that cover diverse and well-supported areas of interest.

Because of these numerous modules, Python is sometimes called a "glue language" as it provides a way to connect powerful libraries of code in interesting ways. Thinking of Python as a glue language doesn't diminish its power but instead shows its versatility.

Python as a glue language is possible because of the Application Programmers Interface (API) the modules support. Like calling the `print("Hello World")` function abstracts the complexities of outputting text to a display, the API a module supports makes it possible to use complex, sophisticated code in your programs.

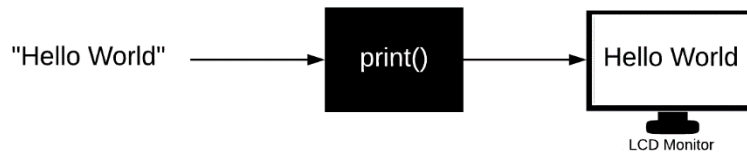
4.1.1 A Contract Between Pieces Of Code

Aside from the rather abstract discussion about what an API is, what is it in practice? One way to think about it is to consider an API a contract between your code and another piece of code whose functionality you want to use. Like a contract between people, it specifies if one party does this, then the other party does that.

In programming terms, this often means when calling a function or method in a particular way, it performs some work and returns some information or both. In Python, when you create a function to be used elsewhere in your code, you've created an API. The name of the function expresses some meaning about what the API does. The input parameters to the function pass information to the API to specify the work to perform and the data to perform it on. If the function returns information, this is the output of the API.

This idea of passing some information into a piece of code, and getting some information or action out of it, has been around in computer science for a long time and is known as the 'black-box' model. What a black-box expects as input and what it does to create output is well understood to the point that knowing what goes on inside isn't necessary. Only the behavior needs to be known, not the implementation. The term black-box comes from the internals of the invoked functionality being opaque, blocked from view.

Visually the use of a black-box function looks like this:



As a developer, you don't have to know about the internal functionality of the `print()`. All you need to know is that passing the function a string; its default behavior is to print that string to the screen.

A local API is a function or class you create, or a module your code imports. The API is within the context of your program code and is accessed directly by calling the functions and class instance methods provided.

It is also possible to call an API that's hosted remotely. An example would be connecting to a database server and accessing data. Here the API is accessed over a network connection providing the transport mechanisms for your code to make calls to the remote API and for the API to respond. We'll get more into this in a later chapter about REST APIs and how to use and create them using HTTP as the transport mechanism. REST is a convention to support common goals when implementing an API.

4.1.2 What's Passed As Input

When your code calls a function or method of an API, its engaging in one part of the contract between your program and the functionality provided by the API. The input arguments are the information passed from the context of your program code to the context of the API. Python functions and methods support positional and keyword arguments. The positional arguments ordering, and names of the keyword arguments, are considerations when using and building an API.

For example, Python's `print` function is most often used like this:

```
>>> msg = "Hello World"
>>> print(msg)
Hello World
```

When this function executes, it prints the string variable `msg` to the screen. The API provided by the `print` function is simple enough to understand; it takes the input argument and performs the work necessary to output it to the screen.

The complete API of the `print` function shows it is a more versatile function. Here is the `print` function signature:

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

This signature indicates that the first parameter is a tuple of non-keyword arguments followed by additional keyword arguments with default values.

The `*objects` parameter allows the caller to pass multiple, comma-separated values to the `print` function. This means:

```
>>> print("Hello World")
Hello World
```

Outputs the same thing to the display as:

```
>>> print("Hello", "World")
Hello World
```

Calling the print function this way works because the function iterates through the tuple of objects parameter, converting each to a string if need be and outputs each object to `sys.stdout` (the screen), separated by a space character.

The `sep=''` named parameter provides the default space separation character and lets you change it to something else to separate the objects.

The `end='\n'` parameter provides a carriage return as the default to end the output and lets you change how the output ends.

The `file=sys.stdout` parameter defines the default destination (standard output, the screen) where the output is sent and lets you change that destination. The object you set the file parameter equal to must have a `write(string)` method. If not, it defaults back to `sys.stdout`.

The `flush=False` parameter provides a way to forcibly push what's sent to the stream to the output destination rather than buffering it if set True.

All of this tells us the print function API is well designed and surprisingly powerful. The use of the initial non-keyword `*objects` tuple, followed by the keyword arguments with default values, lets you use the print function's most common use case. The rest of its functionality is there if needed but can be ignored otherwise.

Imagine if the print function was handled differently. A naïve API implementation might look something like this:

```
print(object)
```

A print function like this would satisfy the common use case but would start to fall apart beyond that. Suppose not long after this version is in use, it's required to print more than one object at a time. One way to extend the simple implementation is to create additional print functions variations like this:

```
print_two(object, object)
print_three(object, object, object)
```

What if additional requirements were placed on this naïve expansion of the API to have a different separator character, perhaps the pipe '|' character? Following the established simple variation pattern would lead to something like this:

```
print(object)
print_pipe_sep(object)
print_two(object, object)
print_two_pipe_sep(object, object)
print_three(object, object, object)
print_three_pipe_sep(object, object, object)
```

This solution doesn't scale well, and code that uses this would have to change to use a different permutation of the API.

The goal of this example is not to show the progression down an ill-conceived path of API development, but to draw closer attention to the details of what makes Python's default print function a good API. There is work going on inside the print function to support the function signature and the use cases where it's applied.

This is one of the earmarks of a good API. It provides useful functionality that can be expanded on without exposing the work involved in doing so. As a developer, you don't have to worry too much about how the print function works. You get to use it, knowing its functionality is well defined and contained.

When developing an API, how you define the input can dramatically influence its utility and future use.

4.1.3 What's Expected As Output

The other part of the contract provided by an API is its output. The output of a function consists of three parts:

1. Return value
2. Actions on the system sometimes thought of as side-effects
3. Exceptions

RETURN VALUE

The most commonly thought of output of a function is the return value. For example, this code:

```
>>> abs(-10)
10
```

The code looks like a mathematical expression, and that's very close to what it's modeled after. The input to the abs function is a number, and the output is the absolute value of that number.

A great deal of programming is creating and using functions that accept parameters, process those parameters, and return the results. Building up an application is a process of orchestrating function calls and feeding the returned values into other functions until you arrive at the desired outcome.

Because everything in Python is an object, a function return value is also an object. This means you can build a function as part of your API that returns more than just a single scalar value like the abs function.

One example commonly seen in Python is to return a tuple. Returning a tuple allows you to pass back more than one value to the calling function, which can then unpack the tuple into variables. Here's some code from `CH_04/example_01.py`:

```

from typing import Tuple

def split_fullname(full_name: str) -> Tuple[str, str, str]:
    fname = mname = lname = ""
    parts = full_name.split()
    if len(parts) >= 1:
        fname = parts[0]
    if len(parts) >= 2:
        mname = parts[1]
    if len(parts) == 3:
        lname = parts[2]
    if not lname:
        mname, lname = (lname, mname)
    return (fname, mname, lname)

# use case
fname, mname, lname = split_fullname("John James Smith")

```

The `split_fullname()` function takes in a full name and returns the name parts, `fname`, `mname`, and `lname`. Even if the `full_name` parameter contains only one or two names, the function behaves correctly. If there are only two parameters, it assumes the second is the last name and sets `mname` to an empty string.

The use case shows how the tuple returned by the function can be unpacked into three variables. You can also assign the return value from `split_fullname()` to a single tuple variable, but it's often useful to unpack the returned tuple directly into waiting named variables.

ACTIONS ON THE SYSTEM

Many API functions perform work to transform data passed to them, create new data, or perform calculations based on the data passed. They transform and create new data, and data structures, based on the input parameters and return the new and updated data to the caller.

API functions can also perform actions on the system it's running on. For example, if you are using an API that is part of a robot and call a function to rotate a motor attached to that robot, you'd expect the motor to start rotating.

The actions taken by API functions are what make applications useful. The ability to open, create, read, and write files, interacting with networks, printing documents, and controlling real-world devices are all actions an application can execute provided by API functionality.

An API function performing an action doesn't necessarily have to return any data to the caller if its primary purpose is to perform that action. That doesn't mean it can't return output data. The API function for the robot motor example could return a `True` or `False` value to indicate whether or not the motor is rotating.

EXCEPTIONS

Exceptions and how to handle them are a fact of life as a developer; mainly when an API function performs work on the system it's running on. Disk drives fail, networks can be unreliable, and any number of other unexpected behaviors can occur.

The API functionality you create can generate exceptions from operations, such as divide by zero or raise exceptions because the functionality created an unexpected or exceptional state.

When creating an API function, one of your goals is to prevent exceptions when you can and handle them gracefully when you can't. As an example, if your functionality is performing network IO and the network becomes unreliable, what can you do?

One possibility is to retry the operation several times with a gradually longer timeout between retries. If the network stabilizes within the retry attempts, the function can continue and succeed. However, if the retries all fail, a network exception is raised and passed upward to the caller.

On the other hand, if a divide by zero exception is raised because of an input parameter, there's nothing you can do but let the exception bubble upwards to some higher-level functionality that handles it.

Handling an exception is knowing whether you can do something about it or not. Never silence an exception without having a specific reason for doing so; this throws information away and makes an API untrustworthy.

Users of your API need to be aware of and prepared to handle the exceptions your API generates, as they do with any other exceptional condition when developing. Documenting your API is an excellent way to inform the users of your API what exceptions they might expect.

Exceptions are covered in more detail in a later chapter.

4.2 Function API

Functions provide the mechanism to interface with an API. In object-oriented programming where you think about methods on an object, they are functions tied to that object instance.

Let's spend some time talking about ideas you can put in place to create useful functions, and by extension, good APIs.

4.2.1 Naming

As we talked about in a previous chapter, names matter in development. How you name the functions you create goes a long way towards making the API you're creating make sense.

Function names should use full English words and use the snake_case format. There is no reason to use abbreviations to shorten the name. Every modern code editor has autocompletion abilities making typing the full name a one-time-only occurrence at the time of function definition.

Using domain-specific acronyms is also discouraged as users who aren't familiar with the domain the API is working in would find the different naming conventions confusing.

The name indicates or hints at the use case of the function, what it returns, and what it accepts. Additionally, the documentation string (docstring) can elaborate further on the intended use. In Python, a function docstring is a tripled quoted-string containing information about a function and immediately follows the function definition. Later chapters cover this in more detail.

In the case where function name collisions are possible because the logical name choice is similar, or the same, use namespaces and separate the functionality into modules. Functions provide the mechanism to interface with an API.

4.2.2 Arguments

When creating a function to provide an API for some functionality, you want to encapsulate, you can think about the Python `print` function presented earlier. That seemingly simple function offered a surprising amount of functionality because of the interface construction and the way the encapsulated code was built.

There are four ways to pass arguments to the functions you create:

POSITIONAL ARGUMENTS

These are the most common form of arguments used with functions and help define usability. Here is an example function definition with positional arguments:

```
def full_name(fname, mname, lname)
```

The name of the function indicates what it returns, and the positional arguments `fname`, `mname`, and `lname` make clear what it expects as input. Calling the function with string literals would look like this:

```
print(full_name("John", "James", "Smith"))
```

The code assigns the string literals to the positional arguments in the same order created when the function was defined.

It is possible to call the function using argument names in this manner:

```
print(full_name(fname="John", mname="James", lname="Smith"))
```

It's also possible to change the order of the arguments by calling the function and using keyword arguments:

```
print(full_name(mname="James", lname="Smith", fname="John"))
```

Positional arguments are mandatory and must have a value assigned to them when calling the function. Otherwise, Python raises a `TypeError` exception.

KEYWORD ARGUMENTS

Keyword arguments aren't mandatory as they have default values. Often these are used for optional arguments allowing the function to operate with known default parameter values when the caller supplies none. The `full_name` function defined above can be altered to use keyword arguments like this:

```
full_name(fname, mname=None, lname=None)
```

Now the function has one positional argument, `fname`, and two keyword arguments, `mname` and `lname`, each with a default value of `None`. The function makes `fname` the only mandatory argument implying the function operates correctly if `mname` and `lname` aren't assigned by the caller.

It's also possible to use the keyword arguments in a different order than defined by the function. For instance, calling the function in this manner:

```
full_name("John", lname=" Smith")
```

The code above indicates the function handles the case where `fname` and `lname` are supplied, but `mname` is assigned the default value of `None`.

When defining a function, once you create an argument with a default value (keyword argument), any arguments following it must also be keyword arguments and have default values.

ARGUMENT LIST

In the Python `print` function, the first argument was of the form `*objects`. The `*objects` parameter is an example of passing a variable number of positional parameters to a function. Inside the `print` function, the `objects` argument is a tuple containing all the remaining positional arguments in the function. A variable number of positional parameters is commonly named `*args`, but that's just a convention, not a requirement.

Modifying the `full_name()` function to use an argument list looks like this:

```
full_name(fname, *names)
```

In this form, the `full_name()` function needs to iterate through the `names` tuple to use the items in the `*names` tuple along with the `fname` argument. This form is useful to pass multiple, similar arguments but can be confusing to the users of the function. Defining the function in the original form where all the arguments have names is a better way to go in this case. From the Zen of Python, explicit is better than implicit.

KEYWORD ARGUMENT DICTIONARY

The keyword argument dictionary is akin to the argument list; it's a way of wrapping up all keyword arguments into a single argument to a function. You'll often see it defined as `**kwargs`, but again, this is only a convention. Changing the `full_name()` function to use this form looks like this:

```
full_name(**names)
```

This definition implies that internally the `full_name()` function examines the `names` dictionary looking for the keywords `fname`, `mname`, and `lname`. Without documentation, the user of this function would not know what to include as key-value pairs in the `names` dictionary.

The caller can also add other key-value pairs to the `names` dictionary that would possibly have no meaning to the `full_name()` function. The extra key-value pairs are ignored by the `full_name()` function but could have meaning to functions it calls and passing the `names` parameter along. Take care when using this form and do so intentionally.

ARGUMENTS IN GENERAL

The ability to create proper function signatures includes being aware of patterns and consistency. Many APIs consist of multiple functions working together to accomplish

something. This often means passing the same data into more than one function, so each function is aware of the state of the working data. If you're processing data contained in a dictionary that's passed to multiple functions, it's a good idea to make the position and name of the argument representing the common data the same for all (or as many as possible) functions that work with it.

For example, make the dictionary the first argument of the function, and all the additional arguments pass information about how to process the dictionary.

The same thing would apply to system resources passed to functions, file handles, database connections, and cursors. If multiple functions need these resource objects, it helps make the API more readily understandable if the functions have a consistent signature.

A function that has many arguments starts to stretch our cognitive abilities and often indicates the function does too much and should be refactored into smaller functions, each with a single purpose. It's tempting to make a long list of function arguments into one using the keyword argument ability of Python. Unless the dictionary passed as the `**kwargs` is documented, it just obscures what the function is expecting. It's also side-steps the original issue that perhaps the function needs to be refactored.

4.2.3 Return Value

As you've seen, one half of the API contract is what's returned by a function. In Python, even if you don't have a return statement in your function code, a `None` is returned automatically.

There's no reason not to return a value, even if it's unused by the caller. A meaningful return value can be tested for in unit tests, making your function or method easier to debug and fix.

If the function you create performs system actions (file IO, network activity, system-level changes), you can return a `True` or `False` to indicate the success or failure of the function.

4.2.4 Single Responsibility

Strive to create functions that do only one thing; this is the Single Responsibility Principle. Writing a useful function is already considerable work, especially if your goal is to make the function flexible through thoughtful input arguments and processing code. Trying to make it do two things more than doubles the difficulty.

Here's a contrived example function from `CH_04/example_02.py` that illustrates this:

```
def full_name_and_print(fname:str , mname: str, lname: str) -> None:
    """Concatenates the names together and prints them

    Arguments:
        fname {str} -- first name
        mname {str} -- middle name
        lname {str} -- last name
    """
    full_name = " ".join([fname, mname, lname])
    full_name = " ".join(full_name.split())
    print(full_name)
```

This function concatenates the arguments to create the `full_name` variable and print it to `sys.stdout`. This function is not as useful as it could be because it's trying to do too much.

It's not useful to other functions needing the full name because it isn't returned. Even if the full name was returned, any function calling this has to expect the full name to be printed.

Also, the function is difficult to test because it doesn't return anything. To test this, you'd have to redirect `sys.stdout` in some way so a test could see the output, which could get messy quickly.

Here is a better version from `CH_04/example_02.py`:

```
def full_name(fname:str , mname: str, lname: str) -> str:
    """Concatenates the names together and returns the full name

    Arguments:
        fname {str} -- first name
        mname {str} -- middle name
        lname {str} -- last name

    Returns:
        str -- the full name with only a single space between names
    """
    full_name = " ".join([fname, mname, lname])
    return " ".join(full_name.split())
```

This version only does one thing: it creates the full name and returns it to the caller. Now the return value of the function can be used with the print function, included in a web page, added to a data structure, converted to a JSON document, and more.

This function has also become easy to test because you can test the return value, and the same input arguments always produce the same output.

4.2.5 Function Length

Related to the Single Responsibility Principle is the length of the functions you write. As people, it's difficult to keep too much context and detail in our heads at once. The longer a function gets, the more difficult it becomes to reason about and understand its behavior.

There is no hard and fast rule to adhere to regarding the length of a function. A good rule of thumb is around 50 lines, but this is entirely dependent on your comfort level.

If you create a function that's too long to comprehend easily, it probably means the function is trying to do too much. The solution is to refactor it and break some of the functionality out to other functions.

If you follow good naming practice and make the new functions do only one thing well, you'll create more readable code.

4.2.6 Idempotent

Though an ominous-sounding word, idempotent in developer terms means a function that always returns the same result given the same argument values. No matter how many times it's called, the same input yields the same output.

The output of the function isn't dependent on outside variables, events, or IO activity. For example, creating a function that uses the arguments along with the clock time to create the

return value wouldn't be idempotent. The return value is dependent on when the function is called.

Idempotent functions are easy to test because the behavior is predictable and can be accounted for in test code.

4.2.7 Side Effects

Functions can create side effects that change things outside the scope of the function itself. They can modify global variables, print data to the screen, send information across a network, and a whole host of other activities.

In your functions, the side effects should be intended, what was referred to as actions on the system previously. Unintended side effects need to be avoided. Modifying a global variable is something a function can do but should be thought about carefully as other, possibly surprising, functionality could be affected by those modifications.

When opening a file, it's good practice to close it when through with it to avoid the possibility of corrupting the file. Database connections should be closed when unused, so other parts of the system can access them. In general, it's good programming practice to clean up and release system resources as soon as your application is finished with them.

There is another side effect to be aware of when working with Python. Because all arguments to functions are passed by reference, the function has the potential of altering variables outside of the function's scope. Example program `CH_04/example_03.py` demonstrates this:

```

from copy import copy

def total(values: list, new_value: int) -> int:
    """This function adds new_value to the values list,
    totals the contents and returns the total

    Arguments:
        values {list} -- list of values to total
        new_value {int} -- value to include in total

    Returns:
        int -- total of values in list
    """
    values.append(new_value)
    return sum(values)

def better_total(values: list, new_value: int) -> int:
    """This function adds new_value to the values list,
    totals the contents and returns the total

    Arguments:
        values {list} -- list of values to total
        new_value {int} -- value to include in total

    Returns:
        int -- total of values in list
    """
    temp_list = copy(values)
    temp_list.append(new_value)
    return sum(temp_list)

values_1 = [1, 2, 3]
total_1 = total(values_1, 4)
print(f"values_1 has been modified: {values_1}")
print(f"total_1 is as expected: {total_1}")
print()
values_2 = [1, 2, 3]
total_2 = better_total(values_2, 4)
print(f"values_2 unchanged: {values_2}")
print(f"total_2 is as expected: {total_2}")

```

When this program runs the following output is produced:

```

values_1 has been modified: [1, 2, 3, 4]
total_1 is as expected: 10

values_2 unchanged: [1, 2, 3]
total_2 is as expected: 10

```

Both the `total()` and `better_total()` functions return the same value, 10, but only `better_total()` is idempotent. The code in the `total()` function is changing the `values_1` list passed to it as a parameter and exists outside of its scope.

This happens because of the way the sum is calculated. The `total()` function appends `new_value` directly to the `values` argument passed to it. Because arguments are passed by reference, this means the list variable `values_1` outside the function and the variable `values`

inside the function both reference the same list. When `new_value` is appended to the list, it's modifying the same list that `values_1` references.

The `better_total()` function is idempotent because inside its scope it makes a copy of the `values` argument, creating a new list variable `temp_list` independent of the one referenced by the `values_2` list outside the scope of `better_total()`. Then it appends `new_value` to `temp_list` and returns the sum of `temp_list`. This leaves the `values_2` list variable untouched, which is the intended behavior.

4.3 Documentation

Documentation is a necessary process when building and defining an API. Functions and the modules they are part of should have documentation that briefly describes the functionality of the module and the functions it contains.

Modules should have docstrings at the top of the file describing the functionality the module provides and possibly the exceptions that might be raised.

Functions and class methods should have docstrings that briefly describe what the function does, what the parameters are for and the expected return value. The functions in the example programs shown previously include example docstrings.

You might hear people say documenting Python code is unnecessary because the code is so readable and self-documenting. The readability part is genuine, but the intent of any reasonably complex piece of code can only benefit from documentation, helping the reader understand that intent.

There's no reason to avoid this minor effort; modern code editors like VS Code make it easy to insert docstring templates. A template for the docstrings in the example programs was generated by hitting return at the end of the function definition, typing triple double-quotes (`"""`) and hitting return again.

Many tools extract and process Python docstrings as part of external documentation systems. Besides being a benefit to readers of the code, including the original author, there is another win. If you're at the Python prompt and type `help(<function name>)`, the built-in help system presents the docstring of the function for reference. This includes not only the builtin functions but those you create. The existence of a docstring makes this possible.

4.4 Summary

This chapter covered what an API is and what a good one looks like. More importantly, it gave you information about how to create a good API, specific things to think about, and steps to take while coding.

Creating a good API is important because ease of use is important. Users of your APIs, which includes yourself, want to make use of the functionality provided, not struggle with how to get that functionality to work.

Creating useful APIs is challenging; there are a lot of moving parts and considerations. Well named modules, functions, and classes with consistent, logical parameters and documentation help give an API good affordance or discoverability.

Like many things, the work put in to create something now pays rewards later when using what you've created as a trusted tool.

5

Object-Oriented Coding

This chapter covers

- Object-oriented APIs
- Designing Objects With Classes
- What Inheritance Is
- What Polymorphism Is
- What Composition Is

5.1 Object-oriented

The ability to place functions into modules provides many opportunities for structuring an API. The type and order of the parameters passed to the functions that make up an API offer possibilities to make your API more discoverable and useful.

Using the concept of single responsibility and keeping functions to manageable lengths makes it more likely your API will consist of multiple functions. Users of the API's functionality, which may itself call other API functions, further modifying the data or state, produces the final result returned to the user.

Often the data structures passed between functions are collection objects, lists, sets, and dictionaries. These objects are powerful, and taking advantage of what they offer is important in Python development.

By themselves, data structures don't do anything, but the functions they are passed to know what to do with the data structures they receive as input.

Since everything in Python is an object, you can create interesting objects using object-oriented programming. One of the goals of creating objects is to encapsulate data and the methods that act on that data into one thing. Conceptually you're creating something with the functionality you designed and implemented. Doing this makes something you can think

about as an object or thing that has behavior. Creating classes is how you design these objects, connecting data and functionality to them.

5.1.1 Class Definition

Python provides object-oriented programming by defining classes that can be instantiated into actual objects when needed. Instantiation is the act of taking something from a definition (the class) to reality. You could say the blueprint for a house is the class definition, and building the house instantiates it.

Here's a simple class definition for a `Person` class from the `CH_05/example_01` application code:

```
class Person:
    """Defines a person by name"""

    def __init__(self, fname: str, mname: str = None, lname: str = None):
        self.fname = fname
        self.mname = mname
        self.lname = lname

    def full_name(self) -> str:
        """This method returns the person's full name"""
        full_name = self.fname
        if self.mname is not None:
            full_name = f"{full_name} {self.mname}"
        if self.lname is not None:
            full_name = f"{full_name} {self.lname}"
        return full_name
```

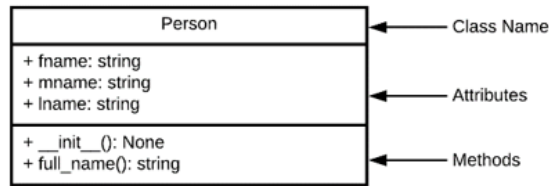
This class definition creates a template to create a `Person` object instance containing a person's first, middle, and last names. It also provides the `full_name()` method to get the person's full name based on the information passed to the object by its constructor, the `__init__()` method.

Creating and using an instantiated person object from the `Person` class looks like this:

```
>>> p1 = Person("George", "James", "Smith")
print(p1.full_name())
```

The `self` parameter passed as the first parameter of every method of the `Person` class is the reference to the person instance just created. In this way, your code can create as many `Person` instances as needed, and each will be distinct because the `self` of each will reference a particular instance.

This class can be represented visually in UML (Unified Modeling Language) as well. UML is a standardized way to present the design of systems visually. It's not necessary to use UML diagrams when designing and building a system, but it can be useful to introduce abstract concepts difficult to present concisely with text alone. Here is the UML diagram for the `Person` class:



The UML diagram for the Person class shows the name of the class, the attributes it contains, and the methods it provides.

The plus sign character '+' in front of the attribute and method names indicates they are public. In Python, attributes and methods of a class are public and have no notion of protected or private access.

Python's class design originates with the idea "we're all adults here," and the developers who use your classes behave accordingly. Using plain attributes should be the default when designing your classes. You'll see later how class properties can gain control of how attributes are accessed and used.

A simple use case for the Person class is presented in the `CH_05/example_01` application:

```
def main():
    # Create some people
    people = [
        Person("John", "George", "Smith"),
        Person("Bill", lname="Thompson"),
        Person("Sam", mname="Watson"),
        Person("Tom"),
    ]

    # Print out the full names of the people
    for person in people:
        print(person.full_name())
```

This code creates four instances of the Person class, each representing a different person and exercising all the variations of the constructor. The for loop iterates through the list of Person object instances and calls the `full_name()` method of each.

5.1.2 Drawing With Class

The rest of the examples you're going to create are object-oriented applications that animate some shapes on screen. Readers who have experience with object-oriented programming will probably recognize the analogy, a generic shape from which specific shapes, like rectangles and squares, inherit. This analogy has been used to present object-oriented techniques for a long time and has become somewhat contrived. I acknowledge that but am using it anyway because it has advantages.

The use of shapes is something familiar enough outside of programming that readers can relate to them and the idea that new shapes derive from them. Additionally, a program that moves shapes around on a computer screen is also familiar to most readers. The idea of

moving shapes having speed and direction, staying within the boundaries of the program window, are accepted behaviors for computer rendered graphics.

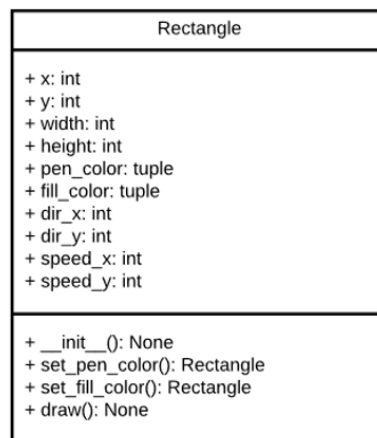
Because of this familiarity with shapes, the cognitive demand of learning about the object-oriented program can be focused on just that, not on any abstract quality of the object itself. For this reason, I'd ask you to bear with any contrived nature of the examples to see the larger picture.

Each of the examples that follow expands on the previous one to present the following concepts:

- Inheritance – parent/child relationships between classes
- Polymorphism – using an object as if it had multiple forms
- Composition – using composition instead of inheritance to give attributes and behavior to a class

To create the drawing application, you'll be using the arcade module available in the Python Package Index (<https://pypi.org/project/arcade/>). This module provides the framework to build a drawing surface on the computer screen and draw and animate objects on that screen.

The first thing to do is to define a class for a rectangle to draw on the screen. Starting with the UML diagram for the Rectangle class:



The UML diagram shows the attributes encapsulated in the class necessary to render a rectangle on-screen. All of these attributes are initialized during the construction of a Rectangle object:

- x, y, width, height – define the position of the Rectangle on screen and the dimensions to use when drawing it
- pen_color, fill_color – define the colors used to outline the Rectangle and fill it
- dir_x, dir_y – the direction of movement relative to the screen x and y axes, these are either 1 or -1
- speed_x, speed_y – the speed at which the Rectangle is moving in pixels per update

- The diagram also includes the definition of three methods the class supports:
- `set_pen_color()` - provides a mechanism to set the pen color used to draw the rectangle instance object
- `set_fill_color()` - provides a mechanism to set the fill color used to fill a rectangle instance object
- `draw()` - draws a Rectangle object instance on the screen

This UML diagram is converted to a Python class definition in code. Here's the Rectangle class based on the above diagram from `CH_05/example_02` application:

```
class Rectangle:
    """This class defines a simple rectangle object"""

    def __init__(
        self,
        x: int,
        y: int,
        width: int,
        height: int,
        pen_color: tuple = COLOR_PALETTE[0],
        fill_color: tuple = COLOR_PALETTE[1],
        dir_x: int = 1,
        dir_y: int = 1,
        speed_x: int = 1,
        speed_y: int = 1
    ):
        self.x = x
        self.y = y
        self.width = width
        self.height = height
        self.pen_color = pen_color
        self.fill_color = fill_color
        self.dir_x = 1 if dir_x > 0 else -1
        self.dir_y = 1 if dir_y > 0 else -1
        self.speed_x = speed_x
        self.speed_y = speed_y

    def set_pen_color(self, color: tuple) -> Rectangle:
        """Set the pen color of the rectangle

        Arguments:
            color {tuple} -- the color tuple to set the rectangle pen to

        Returns:
            Rectangle -- returns self for chaining
        """
        self.pen_color = color
        return self

    def set_fill_color(self, color: tuple) -> Rectangle:
        """Set the fill color of the Rectangle

        Arguments:
            color {tuple} -- the color tuple to set the rectangle fill to

        Returns:
            Rectangle -- returns self for chaining
```

```

    """
    self.fill_color = color
    return self

def draw(self):
    """Draw the rectangle based on the current state"""
    arcade.draw_xywh_rectangle_filled(
        self.x, self.y, self.width, self.height, self.fill_color
    )
    arcade.draw_xywh_rectangle_outline(
        self.x, self.y, self.width, self.height, self.pen_color, 3
    )

```

This class defines a simple `Rectangle` object. The object is initialized with the `x`, and `y` coordinates, the width and height, pen and fill colors, the direction, and speed of motion of the `Rectangle`. In the `arcade` module, the screen origin is in the lower-left corner, which is how most of us think about `x` and `y` axes on paper but is different from other screen rendering tools.

Modifying the values of the `x` and `y` attributes moves the `Rectangle` around the screen maintained by the `arcade` module and the instance of the `Window` class in the application. The `Window` class has two methods used to animate the objects on the screen; `on_update()` and `on_draw()`. The first updates the position of all the objects to render on the screen, and the second draws those updated objects on the screen. The `on_update()` method is called every refresh iteration and is where the application modifies the position of the rectangles in the `self.rectangles` collection. The `on_update()` method looks like this:

```

def on_update(self, delta_time):
    """Update the position of the rectangles in the display"""
    for Rectangle in self.rectangles:
        rectangle.x += rectangle.speed_x
        rectangle.y += rectangle.speed_y

```

This code iterates through the collection of rectangles and updates the position of each one by its `x` and `y` speed values, changing its position on the screen.

The updated rectangles are drawn on the screen by the `Window` instance method `on_draw()`, which looks like this:

```

def on_draw(self):
    """Called whenever you need to draw your window"""

    # Clear the screen and start drawing
    arcade.start_render()

    # Draw the rectangles
    for Rectangle in self.rectangles:
        rectangle.draw()

```

Every time the `on_draw()` method gets called, the screen clears and the `self.rectangles` collection is iterated through, and each `Rectangle` has its `draw()` method called.

The `Rectangle` class has behavior defined by the methods `set_pen_color()`, `set_fill_color()` and `draw()`. These methods use and alter the data encapsulated by the class definition. They provide the API you interact with when using the class.

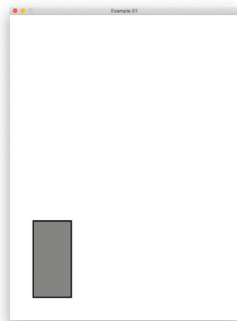
Look at the `set_pen_color()` and `set_fill_color()` methods and you'll see they return `self`. Returning `self` can be useful to chain methods of the class together into a series of operations. Here's an example from `CH_05/example_02.py` using the `Rectangle` class. This code changes the pen and fill colors when the arcade schedule functionality is code is called every second:

```
def change_colors(self, interval):
    """This function is called once a second to
    change the colors of all the rectangles to
    a random selection from COLOR_PALETTE

    Arguments:
        interval {int} - interval passed in from
        the arcade schedule function
    """
    for Rectangle in self.rectangles:
        rectangle.set_pen_color(choice(COLOR_PALETTE)).set_fill_color(
            choice(COLOR_PALETTE)
        )
```

The `change_colors()` method of the `Window` instance is called by an arcade schedule function every second. It iterates through the collection of rectangles and calls the `set_pen_color()` and `set_fill_color()` in a chained manner to set random colors picked from the globally defined `COLOR_PALETTE` list.

When the `CH_05/example02` application runs, it creates a window on the screen and animates a vertically aligned rectangle up and right at a 45-degree angle. It also changes the pen and fill colors of the `Rectangle` every second the application runs. Here's a screenshot of the running application:



PROPERTIES

As mentioned above, direct access to the attributes of a class should often be the default. The `Rectangle` example above follows this practice. However, there are situations where you'll want more control over how the attributes of a class are used or changed.

The definition of the Rectangle class includes attributes for the x and y origin of the Rectangle, which helps draw it in the window. That window has dimensions, and if you run the `CH_05/example_02` application long enough, you'll see the Rectangle move off the screen.

Currently, the origin of a Rectangle instance is set to any integer value. No known screen has a resolution as large as the range of integer values, and none at all deal with negative numbers directly. The window declared in the application has a width of 600 and a height of 800 in pixels.

The boundaries of where rectangle objects can be drawn should be constrained to within those window dimensions. Constraining the values of x and y means having code in place to limit the values that can be assigned to them. Your goal is to make the rectangle bounce around within the screen window.

If you've come from other languages supporting object-oriented programming, you might be familiar with getters and setters. These are methods provided by the developer to control access to attributes of a class instance. Those methods also give the developer a place to insert behavior when the attributes are retrieved or modified.

Adding getter and setter methods to the Rectangle x and y attributes could be done by defining methods like this:

```
def get_x(self):
def set_x(self, value):
def get_y(self):
def set_y(self, value):
```

Using these getter and setter functions also means changing the example code from this:

```
rectangle.x += 1
rectangle.y += 1
```

to this:

```
rectangle.set_x(rectangle.get_x() + 1)
rectangle.set_y(rectangle.get_y() + 1)
```

In my opinion, using getters and setters works, but sacrifices readability when compared to the direct attribute access version/syntax.

By using Python property decorators, you can control how class attributes are accessed and modified while still using the direct attribute access syntax. The Rectangle class can be modified to use property decorators offering this behavior. The updated portion of the Rectangle class from example program `CH_05/example_03` is shown below:

```

class Rectangle:
    """This class defines a simple rectangle object"""
    def __init__(
        self,
        x: int,
        y: int,
        width: int,
        height: int,
        pen_color: str = "BLACK",
        fill_color: str = "BLUE",
    ):
        self._x = x
        self._y = y
        self.width = width
        self.height = height
        self.pen_color = pen_color
        self.fill_color = fill_color

    @property
    def x(self):
        return self._x

    @x.setter
    def x(self, value: int):
        """Limit the self._x to within the screen dimensions

        Arguments:
            value {int} -- the value to set x to
        """
        if self._x + value < 0:
            self._x = 0
        elif self._x + self._width + value > Screen.max_x:
            self._x = Screen.max_x - self._width
        else:
            self._x = value

    @property
    def y(self):
        return self._y

    @y.setter
    def y(self, value):
        """Limit the self._y to within the screen dimensions

        Arguments:
            value {int} -- the value to set y to
        """
        if self._y + value < 0:
            self._y = 0
        elif self._y + self._height + value > Screen.max_y:
            self._y = Screen.max_y - self._height
        else:
            self._y = value

```

The first thing to notice is the attributes `x` and `y` are prefixed with a single underscore `'_'` character. Using the underscore this way is a convention to indicate the attribute should be

considered private and not accessed directly. It doesn't enforce any notion of a private attribute, however.

The second thing to notice is the new decorated methods in the class. For example the two new methods for accessing the `self._x` attribute are:

```
@property
def x(self):
    return self._x

    @x.setter
    def x(self, value):
        """Limit the self._x to within the screen dimensions

        Arguments:
            value {int} -- the value to set x to
        """
        if not (0 < value < SCREEN_WIDTH - self.width):
            self.dir_x = -self.dir_x
        self._x += abs(self._x - value) * self.dir_x
```

The `@property` decorator over the first `def x(self)` function defines the getter functionality, in this case, just returning the value of `self._x`.

The `@x.setter` decorator over the second `def x(self, value)` function defines the setter functionality. Inside the function `self._x` is constrained to within the screen x-axis maximum dimension. If setting the value of `self._x` would place any part of the Rectangle outside the screen area, the direction of travel is set to the negative value of itself to start it moving in the opposite direction.

Having the above-decorated methods in the `Rectangle` class means code like this works again:

```
rectangle.x += 1
```

The program statement appears to be setting the rectangle instance `x` attribute directly, but the decorated methods above are called instead. The `+=` operation calls the getter method to retrieve the current value of `self._x`, adds 1 to that value, and uses the setter method to set `self._x` to that new value. If the resulting change would place the Rectangle outside of the screen dimensions, the direction of travel along the x-axis is reversed.

The beautiful part of this is you can define your classes using direct attribute access initially. If it becomes necessary to constrain access to an attribute, you can define getter and setter property methods. Existing code using your class doesn't have to change at all. From the point of view of the caller, the API of the class is the same.

Take note of another feature of using setter and getter decorated methods. You don't need to create both setter and getter decorated functions on attributes. You can create only a getter, which creates a read-only attribute. Likewise, you can create only a setter creating a write-only attribute.

There is also a `@deleter` decorator to delete an attribute, but this feature is rarely used.

DECORATORS

Before moving on much further, let's talk about decorators. In Python, a decorator is a way to extend or modify the behavior of a function without changing the function itself. Decorating a function sounds confusing, but an example will help make the intent clearer. As has been stated before, functions are objects in Python. One thing this means is functions can be passed to and returned from other functions like any other object.

The function defined below is used to demonstrate the use of decorators:

```
def complex_task(delay):
    sleep(delay)
    return "task done"
```

When the above function is called, it uses the `delay` parameter to emulate some complex task that takes time to perform. It then returns the string "task done" when the task is complete.

Suppose it's required to log information before, and after this function is called, that includes how long it took to execute. That could be done by adding the logging information to the function itself, but that creates code maintenance issues as every function to be timed would have to be updated if the timing code changes. You can instead create a decorator function to wrap `complex_task` with the desired new functionality. The decorator function looks like this:

```
def timing_decorator(func):
    def wrapper(delay):
        start_time = time()
        print("starting timing")
        result = func(delay)
        print(f"task elapsed time: {time() - start_time}")
        return result
    return wrapper
```

This looks odd because the `timing_decorator` function defines another function inside itself called `wrapper`. The `timing_decorator` outer function also returns the `wrapper` inner function. This is perfectly fine Python syntax because functions are objects, the `wrapper` function is created and returned when the outer `timing_decorator` function is executed.

The `func` parameter of the `timing_decorator` is the function object to decorate. The `delay` parameter of the `wrapper` function is the parameter passed to the decorated function.

The code inside the `wrapper` function will execute, including calling the decorated `func` object. An example will help clarify what's going on:

```
new_complex_task = timing_decorator(complex_task)
print(new_complex_task(1.5))
```

Here the `complex_task` function object is passed to the `timing_decorator` function. Notice there are no parenthesis on `complex_task`, the function object itself is being passed, not the results of calling the function. The new variable `new_complex_task` is assigned the return value of `timing_decorator`, and since it returns the `wrapper` function, `new_complex_task` is a function object.

The print statement calls `new_complex_task` passing it a delay value and printing the following information:

```
starting timing
task elapsed time: 1.6303961277008057
task done
```

The output above shows the functionality added by `timing_decorator`, and the original functionality of `complex_task` is being executed.

The example is interesting but not that useful as every invocation of `complex_task` would have to be passed as a parameter to `timing_decorator` to get the additional timing functionality. Python supports a syntactic shortcut making this easier. By adding `@timing_decorator` right before the definition of the `complex_task` function. This has the effect of "decorating" `complex_task` and creating a callable instance of the now wrapped function. This is shown below:

```
@timing_decorator
def complex_task(delay):
    sleep(delay)
    return "task done"

print(complex_task(1.5))
```

The `CH_05/example_04` program demonstrates wrapping the task directly and using the decorator syntax, and when run produces this output:

```
starting timing
task elapsed time: 1.5009040832519531
task done

starting timing
task elapsed time: 1.5003101825714111
task done
```

The output above shows `complex_task` running, but it also indicates the `@timing_decorated` has wrapped `complex_task` with additional functionality that is also running and generating log messages about the elapsed time. The `complex_task` code hasn't changed to provide this; the wrapper function inside `timing_decorator` does this work. Also, any function with the same signature as `complex_task` can be decorated with the `@timing_decorator` to generate timing information.

5.1.3 Inheritance

Being able to merge data and behavior relevant to that data into classes gives you very expressive ways to structure your programs. When building classes there arise situations where functionality is common to more than one class. As a developer, it becomes part of

our nature to follow the DRY (Don't Repeat Yourself) principle. You can follow this principle when creating objects in Python by using inheritance.

Like parents and children, a parent's child inherits attributes and behavior from the parent but aren't exact duplicates. When talking about object-oriented programming class design, the terms "parent" and "child" are used because the metaphor works well. The terms "base class" and "derived class" can also be used to indicate the parent => child relationship.

You'll also see the words "superclass" used to refer to the parent and "subclass" as the child. These are all terms applied to the relationship between objects when talking about inheritance.

One of the reasons to use inheritance is to add attributes and behavior unique to the child, possibly modifying the ones inherited from the parent. It's also useful to derive multiple children from a parent class, each with its own set of unique attributes and behaviors, but still imbued with characteristics from the parent.

To create an inheritance relationship between two classes in Python is performed like this:

```
class ParentClass:
    pass

class ChildClass(ParentClass):
    pass
```

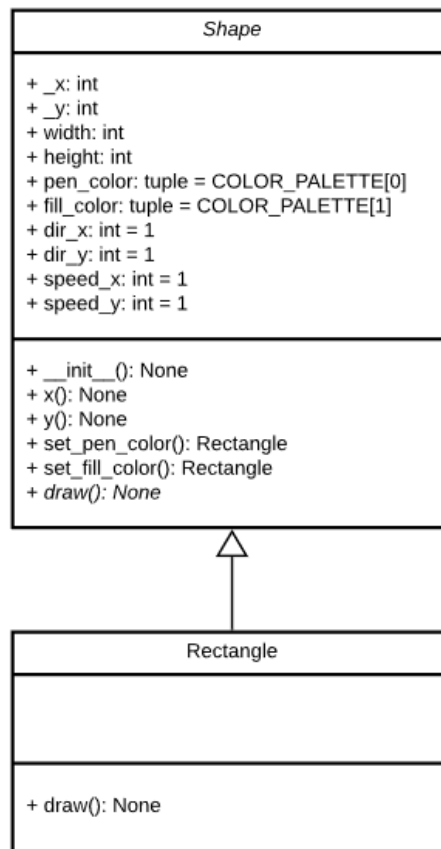
The `ParentClass` definition creates a root level class definition. The definition of the `ChildClass` includes the class to inherit from inside the parenthesis, in the example above `ParentClass`. The `pass` statement in both the class definitions above is a nop (no operation) in Python and is necessary to make the class definitions syntactically correct but have no functionality.

In the `CH_05/example02` code, a `Rectangle` class was created with a position on the screen, a pen color to draw with, and a color to fill the Rectangle. What if you wanted to create other shapes, like squares and circles? Each shape would have a position and dimension on the screen and a pen and fill color.

The direct approach would be to create complete `Square` and `Circle` class definitions and draw instances of each on the screen. Each class would have all of the attributes and methods of the `Rectangle` class but with a different `draw()` method to draw that unique shape. Creating separate classes for `Square` and `Circle` would work for the relatively small number of shapes involved but wouldn't scale well if many more shapes were required.

This presents an opportunity to use inheritance to gather these attributes and their associated behavior into a parent class you could call `Shape`. This common `Shape` parent class would be used to collect the common attributes and methods in one place. Any shape to be drawn on screen would be a child of the `Shape` parent.

You'll start by reproducing the functionality of the `CH_05/example_03` application by making use of inheritance. The examples below come from the `CH_05/example_04` application. The `Shape` and `Rectangle` inheritance is shown with the UML diagram below:



The diagram shows the attributes and methods of the `Rectangle` class definition have moved to the `Shape` class, and the `Rectangle` now inherits from it. The `Shape` class name is in italic to indicate it's an abstract class and shouldn't be instantiated directly. The `draw()` method is also in italic because it exists in the `Shape` definition, but has no functionality of its own. The functionality must be provided by the child class, in this case, `Rectangle`.

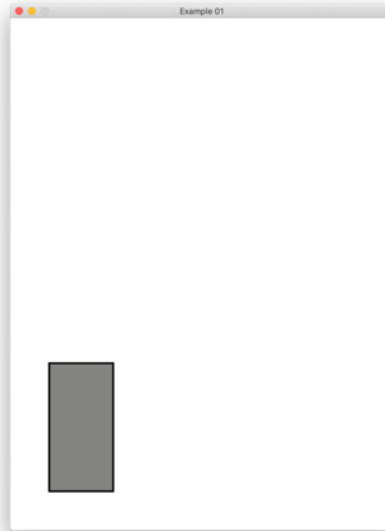
Because the `Shape` class is essentially what the `Rectangle` was, the code won't be shown here. Instead, the updated `Rectangle` class is shown below:

```
class Rectangle(Shape):
    """This class defines a simple rectangle object"""

    def draw(self):
        """Draw the rectangle based on the current state"""
        arcade.draw_xywh_rectangle_filled(
            self.x, self.y, self.width, self.height, self.fill_color
        )
        arcade.draw_xywh_rectangle_outline(
            self.x, self.y, self.width, self.height, self.pen_color, 3
        )
```

The first line of the `Rectangle` class has been modified to include the `Shape` within parenthesis. This is how the `Rectangle` class inherits from the `Shape` class.

The `Rectangle` has been refactored to have only a unique `draw()` method to draw itself on the screen. The `draw()` method overrides the empty one provided by the `Shape` class. Everything else is managed and maintained by the `Shape` class. Even the `__init__()` constructor has been removed because the constructor from the `Shape` class is sufficient.

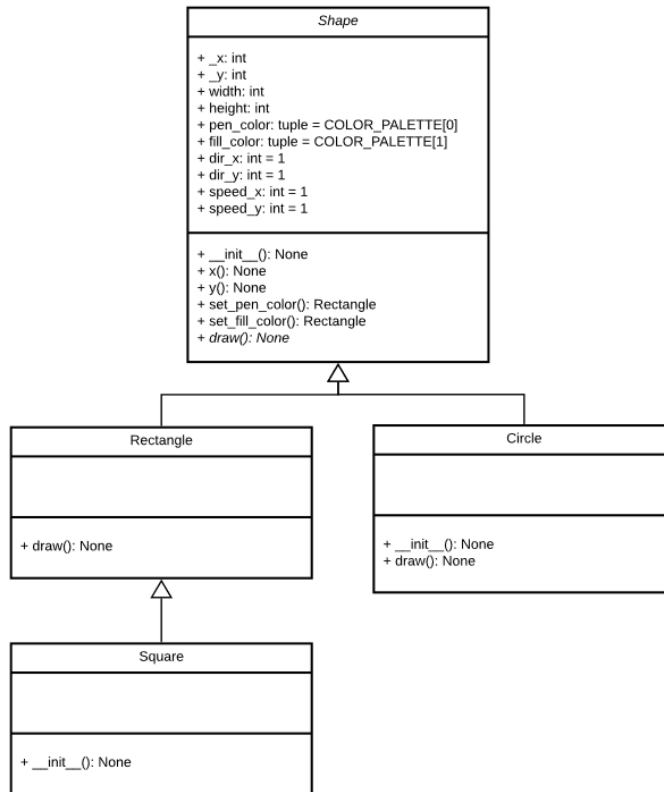


MULTIPLE SHAPES

Now that you have an inheritance structure defined, you can use it to create multiple kinds of shapes with different attributes and behavior. Adding a `Square` and a `Circle` class to the inheritance structure is straightforward. Each additional class inherits from a parent class

providing the most common attributes and methods useful to the new child class. A UML diagram, including `Shape`, `Rectangle`, `Square`, and `Circle` classes, is shown below:

This diagram shows a few interesting things about the inheritance structure. Notice the `Square` class inherits from `Rectangle` instead of `Shape`. This is because a square is a special case of a rectangle with the height and width being equal to each other.



This brings up a concept about inheritance and the relationships between objects. As just mentioned, a `Square` IS-A `Rectangle`, and a `Rectangle` IS-A `Shape`. This also means a `Square` IS-A `Shape` as well. Below is the class definition code for the `Square` class:

```

class Square(Rectangle):
    """This class creates a shape

    Arguments:
        Rectangle {class} -- inherits from Rectangle
    """

    def __init__(
        self,
        x: int,
        y: int,
        size: int,
        pen_color: tuple = COLOR_PALETTE[0],
        fill_color: tuple = COLOR_PALETTE[1],
        dir_x: int = 1,
        dir_y: int = 1,
        speed_x: int = 1,
        speed_y: int = 1,
    ):
        super().__init__(
            x, y, size, size, pen_color, fill_color, dir_x, dir_y, speed_x, speed_y
        )

```

The `Square` class has an `__init__()` constructor even though its parent class, the `Rectangle`, doesn't. The `Square` provides this unique `__init__()` method because it only needs to get a single dimension value, `size`, and not height and width. It then uses the parameters in the `__init__()` method when it makes a call to `super().__init__()`. Because the `Rectangle` class doesn't have an `__init__()` method, the `super().__init__()` calls the `Shape` class, passing its size for both height and width to set the attribute dimensions.

The `Square` class doesn't need to provide a `draw()` method as the one inherited from the parent `Rectangle` class works fine, just with the height and width attributes equal to each other.

The `Circle` IS-A `Shape` because it inherits directly from the `Shape` class. The code that creates the `Circle` class is shown below:

```

class Circle(Shape):
    """This class creates a circle object

    Arguments:
        Shape {class} -- inherits from the Shape class
    """

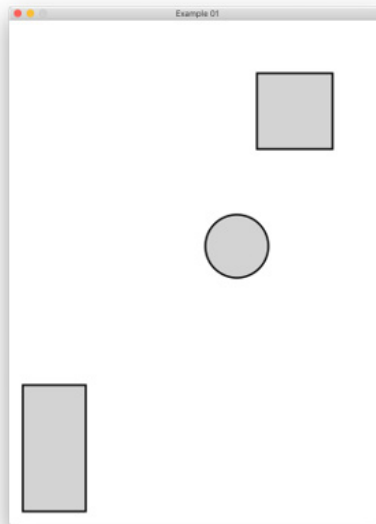
    def __init__(
        self,
        x: int,
        y: int,
        radius: int,
        pen_color: tuple = COLOR_PALETTE[0],
        fill_color: tuple = COLOR_PALETTE[1],
        dir_x: int = 1,
        dir_y: int = 1,
        speed_x: int = 1,
        speed_y: int = 1,
    ):
        super().__init__(
            x,
            y,
            radius * 2,
            radius * 2,
            pen_color,
            fill_color,
            dir_x,
            dir_y,
            speed_x,
            speed_y,
        )

    def draw(self):
        """Draw the circle based on the current state
        """
        radius = self.width / 2
        center_x = self.x + radius
        center_y = self.y + radius
        arcade.draw_circle_filled(center_x, center_y, radius, self.fill_color)
        arcade.draw_circle_outline(center_x, center_y, radius, self.pen_color, 3)

```

Like the `Square` class, the `Circle` provides its own `__init__()` constructor method so the caller can provide a `radius` for the circle. The `radius` parameter is used in the `super().__init__()` call to set the height and width dimensions of the area the `Circle` is within.

Unlike the `Square` class, the `Circle` does provide a unique `draw()` method because it calls different drawing functions in the `arcade` module to present itself on-screen.



When the `CH_05/example_05` application runs, it creates a window with three different shapes bouncing around within the window and changing colors. Initially, it looks like this:

NOTE

Inheritance in object-oriented programming is a feature allowing you to create useful and powerful class hierarchies. Keep in mind that just like human genealogies, the descendants become less and less like the root parent class as you descend the tree. Large hierarchies of classes can become complex to use and understand, and the root class functionality can become wholly obscured in distant child classes.

In my work, I've never gone beyond four levels in any parent/child relationships in class hierarchies I've built.

5.1.4 Polymorphism

There is another feature of inheritance that can be useful when creating class hierarchies, and it's called polymorphism. The word polymorphism means "many forms," and in relation to programming, it means calling a method of multiple objects by the same method name, but getting different behavior depending on which instance of an object is called.

The `CH_05/example_05` application has already taken advantage of polymorphism when rendering the different shapes in the window. Each of the shapes in the program supports a `draw()` method. The `Rectangle` class provides a `draw()` method to render itself on the application screen. The `Square` class uses the inherited `Rectangle` `draw()` method, but with a constraint on the height and width to create a `Square`. The `Circle` class provides its own `draw()` method to render itself. The `Shape` root parent class also provides a `draw()` method, but it has no functionality.

Because the `Rectangle`, `Square`, and `Circle` classes all have an IS-A relationship with the `Shape` class, they all can be considered instances of `Shape` and use the methods provided by the class.

This is what happens in the `Display` class when the `on_update()`, `on_draw()`, and `change_colors()` methods are called. The `Display` class has a collection of shapes in the `self.shapes = []` list created in the constructor. For example, here is the code in the `on_draw()` method:

```
def on_draw(self):
    """Called whenever you need to draw your window"""

    # Clear the screen and start drawing
    arcade.start_render()

    # Draw the rectangles
    for shape in self.shapes:
        shape.draw()
```

This code is called every time the system wants to draw the objects on the screen, which is approximately 60 times a second when using the `arcade` module. When the method is called, the first thing it does is clear the screen.

Then the code takes advantage of polymorphism to iterate through the list of shapes and call the `draw()` method of each one. It doesn't matter that each shape is different, they all support a `draw()` method, and all the shapes are rendered on screen.

Any number of different shapes could be defined in a class, and so long as they support a `draw()` method successfully rendering the shape on the screen, the above loop would work.

NOTE

It is common to use real-world objects or concepts when writing about inheritance. The examples above do exactly this, using the idea of shapes, rectangles, squares, and circles. Using concepts already familiar to you is a useful metaphor because they are things you know already. There are plenty of other new concepts presented when talking about inheritance to think about, using familiar ideas reduces the cognitive load while learning.

However, the use of this metaphor can get in the way of creating useful hierarchies of classes in your applications. Because we've been talking about things that have behavior like actual objects in the real world, this can color how you think about your class design. The objects you create from the classes you design don't have to model real-world objects at all.

Many of the objects which are useful to model with classes have no analog in the real world, and trying to adhere to the analogy too strictly can hinder the work you're trying to accomplish.

5.1.5 Composition

In the inheritance section, you saw the relationships between the `Rectangle`, `Square`, `Circle`, and `Shape` classes. These relationships allowed the child classes to inherit attributes and behavior from their parent class. This creates the idea that a `Rectangle` IS-A `Shape`, and a `Square` IS-A `Rectangle`, which also means a `Square` IS-A `Shape` as well.

These relationships also imply a certain similarity between attributes and behaviors of the parent classes and the child classes that inherit from them. But this isn't the only way to include attributes and behavior into classes.

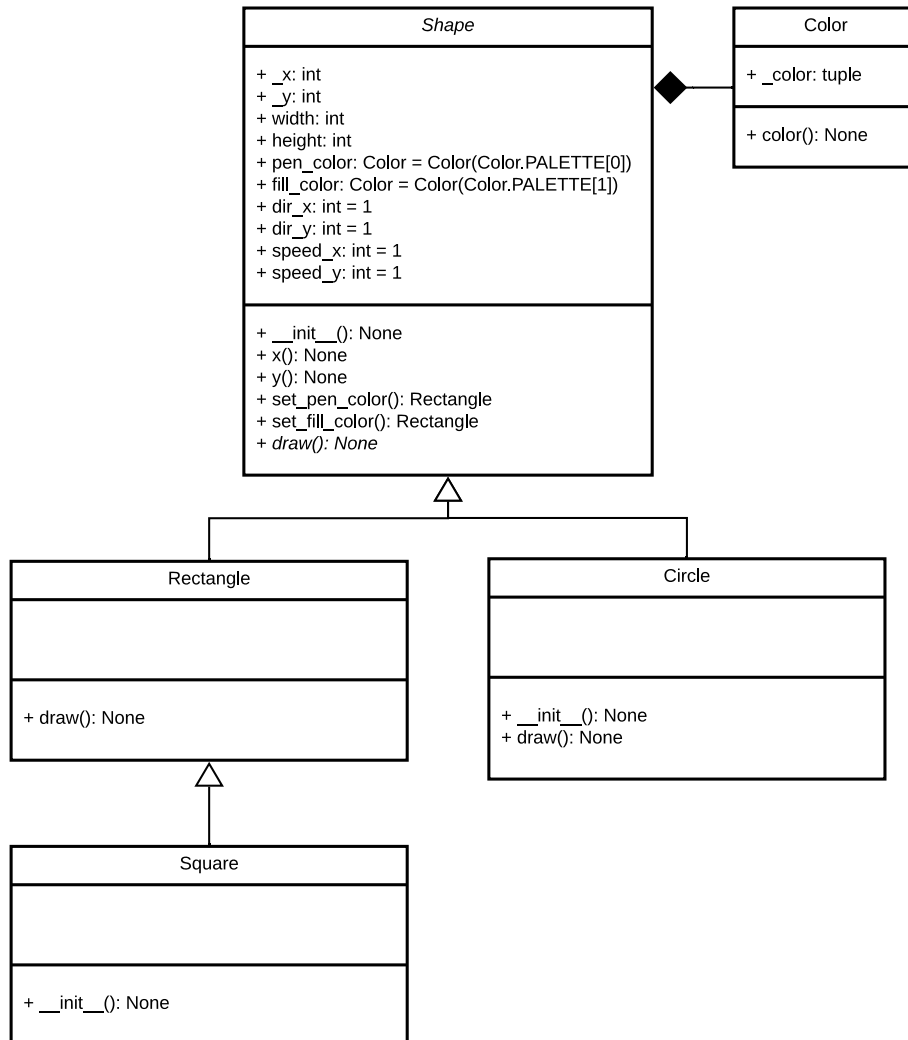
Look at the `Shape` class, it has two attributes for pen and fill color. These two attributes provide color to the shape and are distinguished from each other by their names. But they offer the same thing, a color, most likely from a palette of colors the system can create. This means the color is a common attribute within the `Shape` itself and expressed twice.

It's possible with inheritance to handle this and add to the hierarchy in the examples by creating a `Color` class having pen and fill color attributes and having the `Shape` class inherit from it.

Doing this would work, but the inheritance feels awkward. You can make a `Shape` have an IS-A relationship to a `Color` class in code, but logically it doesn't make sense. A shape is not a color, and it doesn't fit the IS-A mental model of an inheritance structure.

Instead of trying to force inheritance to provide the desired behavior, you can use composition. You've already been using composition when giving classes attributes that are integers and strings. You can take this further and create custom classes to be used as attributes, composing behavior into your own classes.

Creating a new class `Color` provides a consistent abstraction for color in the application. It has a class-level definition for the colors supported and has a mechanism to allow only defined colors to be set. The UML diagram showing the addition of a `Color` class to the hierarchy structure looks like this:



The `Color` class is connected to the `Shape` class as a composite, indicated in the diagram above by the connecting line with the filled black diamond symbol. Here is what the `Color` class looks like from the `CH_05/example_06` application program:

```

@dataclass
class Color:
    """This class defines a color and it's methods"""

    PALETTE = [
        arcade.color.BLACK,
        arcade.color.LIGHT_GRAY,
        arcade.color.LIGHT_CRIMSON,
        arcade.color.LIGHT_BLUE,
        arcade.color.LIGHT_CORAL,
        arcade.color.LIGHT_CYAN,
        arcade.color.LIGHT_GREEN,
        arcade.color.LIGHT_YELLOW,
        arcade.color.LIGHT_PASTEL_PURPLE,
        arcade.color.LIGHT_SALMON,
        arcade.color.LIGHT_TAUPE,
        arcade.color.LIGHT_SLATE_GRAY,
    ]
    color: tuple = PALETTE[0]
    _color: tuple = field(init=False)

    @property
    def color(self) -> tuple:
        return self._color

    @color.setter
    def color(self, value: tuple) -> None:
        """Sets the color in the class

        Arguments:
            value {tuple} -- the color tuple from COLOR_PALETTE to set
        """
        if value in Color.PALETTE:
            self._color = value

```

The `Color` class moves the allowable color list within the scope of the class and out of the global module namespace. It's also a Python dataclass, which can make defining simple classes that are mostly data easier to implement. The class provides getter and setter property decorators to make using the color within the class more straightforward.

In order to use the `Color` class, the `Shape` class is modified to use it for the pen and fill color attributes. The `__init__()` constructor for the class is shown below:

```

class Shape:
    """This class defines generic shape object"""
    def __init__(
        self,
        x: int,
        y: int,
        width: int,
        height: int,
        pen: Color = Color(),
        fill: Color = Color(),
        dir_x: int = 1,
        dir_y: int = 1,
        speed_x: int = 1,
        speed_y: int = 1,
    ):
        self._x = x
        self._y = y
        self.width = width
        self.height = height
        self.pen = Color(Color.PALETTE[0])
        self.fill = Color(Color.PALETTE[1])
        self.dir_x = 1 if dir_x > 0 else -1
        self.dir_y = 1 if dir_y > 0 else -1
        self.speed_x = speed_x
        self.speed_y = speed_y

```

The attribute names for pen and fill color have been simplified to just pen and fill because they are both `Color` class instances. The initial default values have been set to black for the pen and light gray for the fill colors. Adding the `Color` class to the `Shape` class this way creates a HAS-A relationship; a `Shape` has `Color` attributes but isn't a `Color` itself.

The `set_pen_color()` and `set_fill_color()` methods have also been modified to use the new pen and fill attributes. Setting a color for the pen now looks like this:

```

def set_pen_color(self, color: tuple) -> Rectangle:
    """Set the pen color of the Rectangle

    Arguments:
        color {tuple} -- the color tuple to set the rectangle pen to

    Returns:
        Rectangle -- returns self for chaining
    """
    self.pen.color = color
    return self

```

Running the `CH_05/example_06` application produces a screen exactly like you've seen before, three shapes bouncing around the window and changing colors every second.

The use of composition gives you a way to add attributes and behavior to a class without having to create contrived hierarchies of inheritance.

5.2 Summary

Creating classes and class hierarchies gives you another way to create code that's clean and well-controlled in its usage. Classes are another avenue to give your users an API into application functionality.

A class definition also provides another level of namespacing and scope control. A module provides a namespace, and classes defined within that module creates more namespaces within it. The attributes and methods of a class are within the scope of instance objects of the class.

In this chapter, you've learned about classes, inheritance, and how to create class hierarchies. Using classes is a powerful way to take advantage of code reuse and adhere to the DRY principle.

You've also learned about polymorphism and how that can make using related classes more straightforward by taking advantage of the relationships inherent in class hierarchies.

Lastly, you've seen how using composition can be used to apply the single responsibility principle to class design to avoid contrived and disjointed class design.

6

Exception Handling

This chapter covers

- **What An Exception Is**
- **Why They Occur In Programs**
- **How To Handle Exceptions**
- **How To Raise An Exception**
- **Creating Custom Exceptions**

Developing software puts you in a digital world where you're quickly drawn into the binary nature of computers and the systems running on them. Things are either on or off, work, or don't work and are True or False. Any software you create to perform some task for yourself or multiple users lives in the real world, which is far from binary.

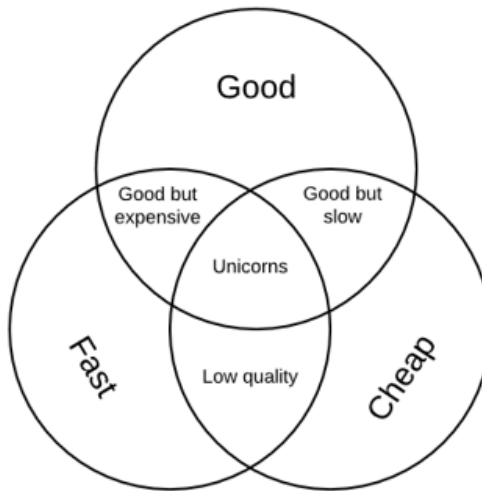
The real world is not black or white; it is an infinitely variable field of grey. Computer systems that run software lose power and fail. The networks connecting systems are slow, intermittent, or unavailable. The storage systems software depends on to save information become unreliable and fail. The users of your software make incorrect assumptions and enter wrong or misleading data.

In addition to the sometimes-problematic world where software runs, you will create bugs in the code you write. Software bugs are errors or failures, causing an application to produce unexpected results. Most software applications providing something useful are complex enough that bugs will creep in. These come from wrong assumptions on the part of the developer, oversights, syntax errors, and just everyday mistakes.

All of this shouldn't discourage you as a developer, but to broaden the way you think about creating software applications. The problems outlined above can be managed and handled and are part of the challenge of becoming a developer. How you handle these challenges is dependent on the requirements of the application and its users. If you're knocking together a quick program to solve an issue for yourself, creating something that

might crash if given incorrect input is probably acceptable. On the other hand, if you're creating something hundreds, or thousands, of users will access, it will take longer to develop and handle conditions that might arise.

Many of you might already be familiar with a Venn diagram like the one shown below:



This diagram shows the relationship between three things related to the development of software applications.

Good in this context represents the software application quality; it meets the requirements and the user's expectations with a low occurrence to bugs.

The intersection of Good and Cheap can represent time or cost (or both) to develop the software application and indicates taking more time. Good and Cheap can mean a small team of developers, or even one, working to create the application. It can also indicate perhaps less experienced developers creating the application.

The intersection of Good and Fast also represents time or cost and indicates creating the application quickly with good quality. Good and Fast almost always means more developers working on the project, and those developers are usually experienced. Either of those conditions indicates a higher cost to create a good application.

The intersection of Fast and Cheap implies taking short cuts to create the application and leads to a low-quality result. Fast and Cheap might be acceptable for a one-off utility application, but generally should be avoided.

Only the intersection of two of the circles can ever be chosen and achieved, even though the diagram shows the intersection of all three circles. The intersection of all three circles is where unicorns live. Trying to reach the land of the unicorns will keep you from actually creating something useful.

6.1 Exceptions

Python handles unexpected events in running applications by raising exceptions. You can think of these as exceptional conditions occurring while an application runs, and they're not all necessarily errors.

If you've done any Python programming at all, you've seen exceptions raised. A simple math error in a program statement will raise an exception, as shown below:

```
>>> print(10/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

The exception `ZeroDivisionError` is raised by Python because the results of dividing 10 by 0 are undefined. The exception `ZeroDivisionError` is a child, or subclass, of the base `Exception` class. The `Exception` class is the base, or parent, of most other exceptions Python can raise. In this way, the `Exception` class is precisely like the `Shape` class is the parent of the `Rectangle` class in the previous chapter.

When an exception occurs in a program, Python will stop executing your program code and start working its way back up the call stack, looking for an exception handler.

The call stack is the series of function calls leading to the code that raised the exception. Python is looking for a handler to intercept the exception and do something with it. If no exception handler is found, Python exits the application and prints a stack trace.

The stack trace shows the sequence of functions called that led to the raised exception. Each function listed in the stack trace also shows the line number in the module where the function in the stack trace is called. The stack trace continues until the last function called, where the line number that raised the exception is displayed.

The example program `CH_06/example_01.py` demonstrates this:

```
"""Exception example
"""

def func_a():
    dividend = float(input("Enter a dividend value: "))
    divisor = float(input("Enter a divisor value: "))
    result = func_b(dividend, divisor)
    print(f"dividing {dividend} by {divisor} = {result}")

def func_b(dividend: float, divisor: float) -> float:
    return func_c(dividend, divisor)

def func_c(dividend: float, divisor: float) -> float:
    return dividend / divisor121

func_a()
```

This program shows `func_a()` getting input from the user for the `dividend` and `divisor` and converting those input strings to floating-point values. It then calls `func_b()`, which calls `func_c()`, which performs the division operation on the two passed parameters. Running this program produces the following output for the values entered at the prompts:

```

Enter a dividend value: 12.2
Enter a divisor value: 0
Traceback (most recent call last):
  File
    "/Users/dfarrell/GoogleDrive/projects/manning_publishing/becoming_a_pythonista/code/
    project/CH_05/example_01.py", line 20, in <module>
    func_a()
  File
    "/Users/dfarrell/GoogleDrive/projects/manning_publishing/becoming_a_pythonista/code/
    project/CH_05/example_01.py", line 8, in func_a
    result = func_b(dividend, divisor)
  File
    "/Users/dfarrell/GoogleDrive/projects/manning_publishing/becoming_a_pythonista/code/
    project/CH_05/example_01.py", line 13, in func_b
    return func_c(dividend, divisor)
  File
    "/Users/dfarrell/GoogleDrive/projects/manning_publishing/becoming_a_pythonista/code/
    project/CH_05/example_01.py", line 17, in func_c
    return dividend / divisor
ZeroDivisionError: float division by zero

```

The code shows Python encountering an exception in `func_c()` when it tries to divide 12.2 by 0, then going back up the call stack to `func_b()` and then to `func_a()` looking for a handler to intercept the exception. Since there isn't a handler, Python exits the program, prints the exception raised, and prints the stack trace that caused the application to crash.

Another possible exception the program can raise happens if the user enters a string at either of the two prompts that can't be converted to a floating-point value. Here is an example of running the program raising that exception:

```

Enter a dividend value: Python
Traceback (most recent call last):
  File
    "/Users/dfarrell/GoogleDrive/projects/manning_publishing/becoming_a_pythonista/code/
    project/CH_05/example_01.py", line 20, in <module>
    func_a()
  File
    "/Users/dfarrell/GoogleDrive/projects/manning_publishing/becoming_a_pythonista/code/
    project/CH_05/example_01.py", line 6, in func_a
    dividend = float(input("Enter a dividend value: "))
ValueError: could not convert string to float: 'Python'

```

In this example, the stack trace only shows `func_a()` because the `ValueError` exception was raised within that function when the program tried to convert the string "Python" to a floating-point value.

6.2 Handling Exceptions

Handling exceptions in Python is done by using a `try / except` block in your program code, which looks like this:

```

try:
    # code that might raise an exception
except Exception as e:
    # code that executes if an exception occurs
else:
    # code that executes if no exception occurs (optional)
finally:
    # code that executes whether an exception occurs or not (optional)

```

The `try` statement begins a block of code that might raise an exception your program can handle. The `except Exception as e:` statement ends the block of code and is where an exception is intercepted and assigned to the `e` variable. The use of `e` is not required syntax and is just a convention.

Because of the `except Exception as e:` in the handler part of the block, the example above will catch any exception raised by the code within the `try / except` block.

The `else` and `finally` clauses of the `try / except` block are optional and used less often in practice.

6.2.1 Handle An Exception If The Code Can Do Something About It

When thinking about exceptions, it's easy to get into a frame of mind to handle them everywhere they might occur. Depending on where in the program the exception occurs, this might be a logical choice.

Often, however, exceptions happen within the context of a function where the code is acting on passed parameters. At this point, the scope of work the code is performing is narrow, and the broader context of what the program is trying to accomplish is at a higher level.

When an exception occurs in a function, it's possible the exception handler can make choices that make sense within the context of the function. The handler might be able to retry the operation for a fixed number of attempts before letting the exception flow upwards in the call stack to a higher context. It could make assumptions based on the exception and correct or change the state of data to make the code continue without raising an exception.

6.2.2 Allow Exceptions To Flow Upward In Your Programs

Unless the code where the exception occurs can do something useful about the exception, it's better to let the exception flow upwards through the call stack to a higher level of context.

At higher levels of context, decisions can be made about how to handle the exception. The higher-level context might be the point where choices about retrying operations are made. At the higher levels of context, more information might be available about what the program is trying to accomplish and what alternative paths can be taken. At this point, you can decide what information to present to the user so they can make choices about how to proceed.

The program should also log the exception, and the stack trace associated with it, so the developer has information about the path taken that generated the exception. This is incredibly useful information to have when debugging an application and trying to resolve problems.

It's also possible an exception is fatal to the program, and nothing can be done other than logging the exception stack trace and exiting the program. Exiting an application is an entirely reasonable course of action for some applications, like utility programs and command-line tools.

6.2.3 Never Silence An Exception

It's possible to handle an exception and silence it. Silencing an exception is shown in the next two examples:

```
try:
    # some code that might raise an exception
except:
    pass
```

and this:

```
try:
    # some code that might raise an exception
except Exception:
    pass
```

The first example catches all exceptions, including system and keyboard events like (CTRL-C to exit a program). The second catches a narrower scope of exceptions, but still far too broad.

Worse than catching too broad a scope of exceptions, the above code lets the exception pass silently. It doesn't inform the user or log the exception stack trace. The user is deprived of information about why the application is malfunctioning, and the developer isn't given any information about what the exception is or where it's occurring.

The presence of either of these blocks of code is an indication of a low-quality application. Trying to find the source of a problem in this kind of application is frustrating and time-consuming.

EXAMPLE 1 IMPROVED

The program `CH_06/example_01.py` can be improved to handle exceptions and provide a better user experience. The program `CH_06/example_02.py` demonstrates this improvement:

```

"""Exception example
"""

def func_a():
    dividend = float(input("Enter a dividend value: "))
    divisor = float(input("Enter a divisor value: "))
    result = func_b(dividend, divisor)
    print(f"dividing {dividend} by {divisor} = {result}")

def func_b(dividend: float, divisor: float) -> float:
    return func_c(dividend, divisor)

def func_c(dividend: float, divisor: float) -> float:
    return dividend / divisor

successful = False
while not successful:
    try:
        func_a()
    except ZeroDivisionError as e:
        print(f"The divisor can't be a zero value, error:", e)
    except ValueError as e:
        print(
            f"The dividend and divisor must be a string that represents a number, error:",
            e,
        )
    else:
        successful = True
    finally:
        if successful:
            print("Thanks for running the program")
        else:
            print("Try entering a dividend and divisor again")

```

In the example above the functions `func_a()`, `func_b()` and `func_c()` are unchanged and don't catch exceptions. They follow the pattern of letting any exceptions flow upward through the stack to a higher-level context.

That higher-level context is where `func_a()` is called. Now there is a while loop around the function that will keep trying `func_a()` until it can complete successfully.

Within the while loop, there is a handler catching two exceptions, `ZeroDivisionError` and `ValueError`. Both of these handlers prompt the user with information about what went wrong and provide advice about how to proceed.

The `else` clause of the handler only executes if `func_a()` can run successfully without raising an exception. When this happens, it sets the `successful` variable to `True`, which signals the enclosing while loop to exit.

The `finally` clause takes advantage of the state of the `successful` variable to either indicate the program is done or encourage the user to try again.

Running this program with possible input from the user looks like this:

```

Enter a dividend value: Python
The dividend and divisor must be a string that represents a number, error: could not
convert string to float: 'Python'
Try entering a dividend and divisor again
Enter a dividend value: 12.2
Enter a divisor value: 0
The divisor can't be a zero value, error: float division by zero
Try entering a dividend and divisor again
Enter a dividend value: 12.2
Enter a divisor value: 3.4
dividing 12.2 by 3.4 = 3.5882352941176467
Thanks for running the program

```

This program follows most of the recommendations to handle exceptions:

- Allow the exceptions to flow upward to a higher context
- Handle an exception if the code can do something useful about it
- Inform the user about the problem and suggest a solution
- Don't silence an exception

The program doesn't log the exception and the stack trace, as this would be distracting information for the user given the simplicity of this program. But that information could be added to the handlers for `ZeroDivisionError` and `ValueError`.

Logging an exception can be handled by using Python's logging module in this manner:

```

import logging

logger = logging.getLogger(__file__)

try:
    # code that could raise an exception
except Exception as e:
    logger.exception("something bad happened", e)

```

The code above imports the logging module and gets a simple logger instance. Calling `logger.exception()` inside the exception handler will print the message and the stack trace to the logger output.

In this simple example, the output of the logger is directed to `stdout`, which is the screen. Most use cases of loggers are directed to log files to keep logging information from interfering with any output the program is producing for the user.

The introduction to this section states an exception should never be silenced, and that is predominantly the right way to think about handling exceptions. There are situations where the developer's knowledge about the code would make silencing an exception acceptable, if taken with forethought. The recommendations above still hold, however. This kind of code:

```

try:
    # some code that might raise an exception
except:
    pass

```

and this:

```
try:
    # some code that might raise an exception
except Exception:
    pass
```

Is still far too broad as it will catch all exceptions. Something that catches a specific Python exception, or a custom exception, narrows the scope of what will be handled to something intentional. That is the real goal, to make exception handling intentional rather than accidental.

6.3 Raising An Exception

In Python, you can raise exceptions programmatically in your code. Raising an exception might seem like an odd thing to do since most of the time, an exception equates to errors. However, raising an exception is a useful way to handle conditions within your programs that you decide are exceptional and not necessarily programmatic errors.

As an example, suppose you're writing a quiz application providing calculations based on user input. One of the calculation functions only works if a user-entered parameter is greater than 0 and less than or equal to 100. The code should define a range of values acceptable to the function.

Since integers in Python have a much larger range, your program code will need to limit the user input to within the range $0 < \text{parameter} \leq 100$. Restricting the range is easy enough to do at the point of use in the function, but what should the function do about it if the range is violated?

Most likely, the function should do nothing as it doesn't have the context to do anything useful if the range restriction is violated. Keeping in mind the idea of letting exceptions flow upwards to where they can be handled, raising an exception can be useful.

Here is one way to handle the range restriction in the function:

```
def calculate(parameter):
    if not 0 < parameter <= 100:
        raise ValueError("parameter range exceeded", parameter)

    # continue with function
```

At the very top of the function, a conditional statement checks if the parameter is not within the acceptable range, and if not, a `ValueError` exception is raised. If the parameter is within range, the function continues normally.

The code passed the responsibility for handling the out of range `ValueError` exception up the call stack to the calling function. The calling function most likely does have the context necessary to handle the exception, perhaps by prompting the user to enter the parameter value again.

Handling the `ValueError` exception in the calling function might look like this:

```
def prompt_user_for_data():
    # initialization and gather user input
    try:
        calculated_result = calculate(parameter)
    except ValueError as e:
        print(e)
        # restart code to gather user input
```

The `prompt_user_for_data()` function calls the `calculate()` function inside a `try / except` block that handles a `ValueError` exception, prints the error out for the user, and restarts the process to get data from the user.

6.4 Creating Your Own Exceptions

Python allows you to create custom exception classes your code can raise. Creating a custom exception might seem unnecessary since Python has a rich set of exception classes already defined. There are a couple of good reasons to create custom exceptions:

- Exception namespace creation
- Exception filtering

In the previous section, the `ValueError` was raised by the `calculate()` function, so an exception handler at a higher level in the `prompt_user_for_data()` function can intercept and do something about it.

But suppose the `calculate()` function raised an unrelated `ValueError` later in the function? The exception would flow upwards through the stack to the calling function and be caught by the exception handler in `prompt_user_for_data()`.

At that point what should the `prompt_for_user_data()` function do? The code can't assume the correct behavior is to show the error to the user and restart the process of gathering data because the range check isn't the only possible source of the exception.

One option is to examine the exception arguments by looking at the `e.args` attribute tuple. Then the code can make choices within the exception handler to determine the source of the exception.

This solution is brittle since it depends on the arguments passed at the point where `ValueError` was raised, and those might change sometime later.

A better design is to create an exception specific to the needs of the program, narrowing the scope of the exceptions to handle. You create a custom exception handler in the module that defines the `calculate()` function like this:

```
class OutsideRangeException(ValueError):
    Pass
def calculate(parameter):
    if not 0 < parameter <= 100:
        raise OutsideRangeException("parameter range exceeded", parameter)

    # continue with function
```

The code above creates a new exception class named `OutsideRangeException` that inherits from the `ValueError` exception, which inherits from the parent `Exception` class.

This new exception class is used in the `calculate()` function and raised if the `parameter` value is outside the defined range of acceptable values.

Now the abbreviated program code looks like this:

```
class OutsideRangeException(ValueError):
    Pass

def prompt_for_user_data():
    # initialization and gather user input
    try:
        calculated_result = calculate(parameter)
    except OutsideRangeException as e:
        print(e)
        # restart code to gather user input
def calculate(parameter):
    if not 0 < parameter <= 100:
        raise OutsideRangeException("parameter range exceeded", parameter)

    # continue with function
```

If the `parameter` value is outside the acceptable range, the `prompt_for_user_data()` function can catch that specific exception and handle it. If a `ValueError` exception is raised by the `calculate()` function, that exception will flow upward to a handler catching that specific exception (or just the base class `Exception`).

A complete example program using the logging module that demonstrates this is in `CH_06/example_03.py`.

6.5 Summary

In this chapter, you've learned these things about exceptions:

- What exceptions are
- Why they occur
- How to handle exceptions
- How to raise your own exceptions
- How to create custom exceptions

Understanding how to handle and use exceptions is essential to a developer. They are messages our programs receive from the real world about the things happening to our programs, and the result of the actions our programs take. A goal of development is to create something useful in the world. Exceptions and how we handle them are tools allowing you to develop successful programs that will be used and well-received in that world.

7

Your First Web Server

This chapter covers

- The Project Application
- What a Web Server Does
- The Flask Microframework
- Running the Server

7.1 The Project Application

You've covered a lot of material in the previous chapters about being a developer. Now it's time to start developing something that puts those topics to use. Choosing an application to present is tricky because the domain of possibilities is so large. The project you'll be creating is a small but well-featured blogging platform we'll be referring to as MyBlog. The MyBlog application will be available as a web-based Python application.

The MyBlog web application will provide tools so users can join the blogging community and create blog posts. Registered users can post content using markdown for styling. All users will be able to view the posted content, and registered users will be able to comment on it. Administration users will be able to mark any content or comments inactive as they see fit. Registered users will be able to mark as deleted any content they've created

7.1.1 Web Application Advantages

The project choice to create a web application is based on a few considerations. First and foremost, creating a useful web application builds on the topics covered in previous chapters quite well. Pulling together the topics of development tools, naming and namespaces, API use and creation, and class design will play into the big picture of the application.

Other types of applications also offer some of these opportunities to express what you've learned but can be challenging to share and distribute to others. For example, creating a

desktop GUI application offers interesting challenges to a developer. However, distributing a GUI application for widespread use can be difficult. It's certainly possible to do so with Python, but the steps necessary are outside of the scope of what I want to present in this book.

A web server has some advantages in terms of distributing an application. The web server itself and the features and services are centrally located and are not running on many and varied computer installations. Having the server centralized like this means that changes and updates to the application happen in one place. Restarting the server, or pushing changes out interactively, makes the changes and updates available immediately to all users.

Making changes to the server application this way is very different from updating applications installed locally and running entirely on computers outside of your control. The changes and updates you want to make available to your application users are entirely dependent on those users updating the application and staying in sync with your work.

Another advantage of a web server-based application is making a user interface available. A web application takes advantage of something installed on just about every computer in existence, a web browser. Modern web browsers provide a powerful platform on which to build user interfaces. Data can be formatted and presented in almost infinite ways. Images and multimedia are also well supported. Users can interact with applications hosted on browsers using interface elements like buttons, lists, drop-down lists, forms and drag and drop.

7.1.2 Web Application Challenges

Using a web browser as an application platform isn't challenges; there certainly are. Creating a web application means you'll be working in multiple technical domains. Along with Python, you'll be creating HTML, CSS, and JavaScript code files. Additionally, desktop-based applications offer more direct access to the computer hardware and the tremendous computing power personal computers bring to bear.

However, with continuing advancements in browsers, new and expanding web technologies, and ever-increasing Internet speed that's widely available, the performance gap between desktop and web-based applications is narrowing. In addition, web-based systems have grown to have widespread acceptance as a method of delivering applications. This acceptance makes creating them a valid path for both personal and professional development.

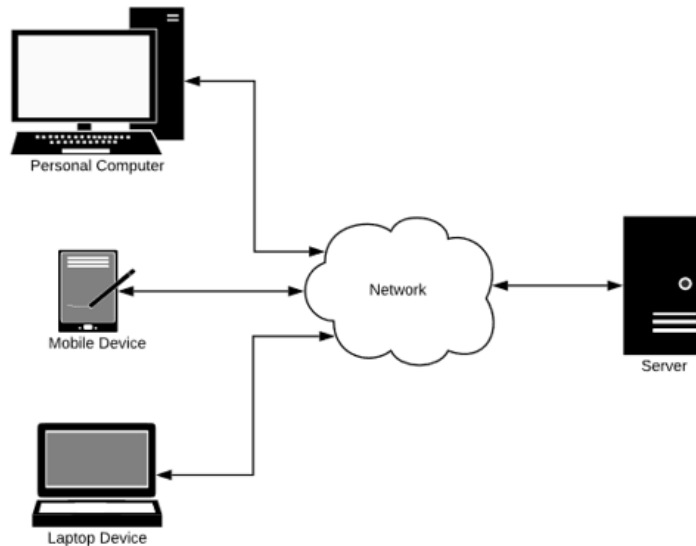
There are existing blogging platforms available to use or download and run yourself. The MyBlog application won't be in competition with them. What the application offers isn't groundbreaking technology; blogging software is well-understood. This is one of the MyBlog's advantages, already knowing what a blog application is intended to supply. The goal isn't to create a groundbreaking blog but to see the big picture of the application's intent and think like a developer to pull the necessary parts together to paint that picture into existence.

7.2 Servers

The MyBlog web application is a subset of what a server application in general provides. One definition of a server is an application running on a computer, or computers, providing

functionality to other applications across a network. This arrangement of multiple applications accessing the functionality of a central server is known as the client-server model.

We all use social networks and work with the programs that run on our desktop or mobile devices. Those tools are client applications using the functionality of many servers. If you play any multiplayer games, the game application uses a server's functionality to coordinate all the players' actions in the game. A diagram of a server connected to several clients is shown below:



7.2.1 Request / Response Model

One common server implementation is the request-response model. The client application makes a request to the server, which processes the request and returns a response. In this kind of application, the server takes no action unless requested to do so.

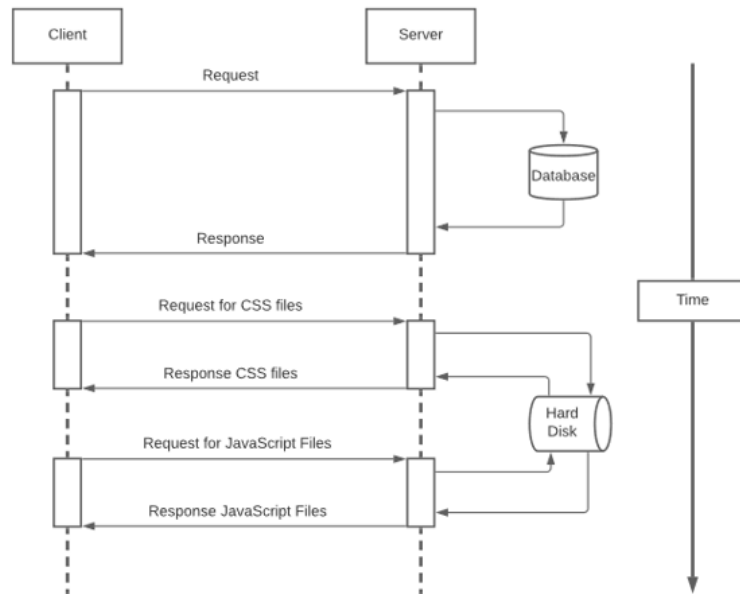
In a web application the client browser makes an HTTP request to the server for a response. This response is very often a stream of text representing an HTML page. When the client browser receives the response, it will render the HTML to the browser window. Depending on the HTML page it might also make additional requests to the server for information like CSS (Cascading Style Sheet) and JavaScript files.

The diagram below represents a simplified view of the request/response communication occurring between a client and server over time. Initially the client makes a request to the server, which might need to retrieve data from a database in order to compose the response. When the response is created it is transmitted back to the client.

In this example the response is an HTML page the client application will render in the browser window. Part of the HTML data includes links to CSS and JavaScript files, which

generates additional requests to the server. The server retrieves the requested files from the server's hard disk and sends them as a response to the client.

The request/response model is the primary means the MyBlog web application will use to get data from the server and build and present the application information to the users.



7.3 Web Servers

A web server is an application responding to HTTP requests from a client application. A web browser is a client application that makes requests to the web server and interprets the responses and displays them on the screen. Often what's sent to the client's web browser are HTML documents the browser interprets and renders as web pages. HTML documents are the content the client requested.

There are many other interactions between a client's browser and a web server. The browser can request the server to send pictures, audio and video content, even downloading other applications to the client computer.

HTML documents can contain links to CSS and JavaScript files. When the HTML received from a web server is rendered by the web browser, the embedded links to those files generate additional HTTP requests to the web server. The web server responds by sending the requested content.

CSS files contain styling information applied to the content in the HTML document displayed on the screen. The CSS code modifies a web page's look and feel and is the presentation layer to the HTML's content.

JavaScript files contain code that runs in the client's browser. Once downloaded, the web browser will start to execute the JavaScript code. This code can be connected to on-screen button clicks, updates to the display, and just about any action the user can make on a web page is handled by JavaScript code.

JavaScript code can also make HTTP requests to web servers for text and data. These requests can be initiated by user actions or programmatically and can change and update web pages dynamically.

HTTP REQUESTS

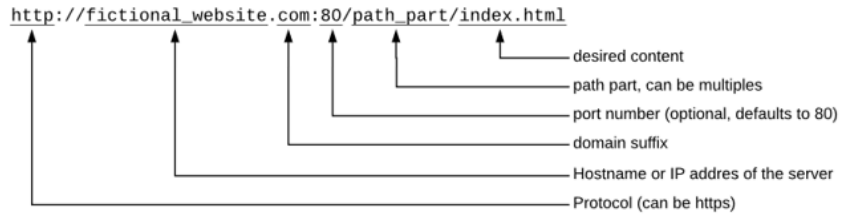
The HTTP protocol definition is not the intent of this book and is beyond its scope, but some basic information is useful. Here is an example HTTP request to a web server:

```
GET /path_part/index.html HTTP/1.1
Host: fictional_website.com:80
Accept: image/gif, image/jpeg, */*
<CR-LF>
```

The line numbers in the example above are not part of the request but were added to reference the lines in the explanation of the protocol:

1. GET /path_part/index.html HTTP/1.1 – This is the start of the request to the web server. The word GET indicates the HTTP method to use, in this case to retrieve the document located at /path_part/index.html using HTTP protocol version 1.1
2. Host: fictional_website.com:80 – Indicates the host, domain suffix, and port number where the request is sent. The port number (80) is optional, and if absent, defaults to 80.
3. Accept: image/gif, image/jpeg, */* – Is header information, which is optional. In this example, it indicates to the server the kinds of responses the client can accept. There can be many headers, each containing additional information from the client that can be useful to the web server.
4. <CR-LF> - Indicates carriage return/line feed characters or a blank line, which is a required part of the HTTP protocol and ends the list of headers and tells the server to start processing the request.

The first two lines of the request create a URL, or Uniform Resource Locator, which uniquely identifies what the client is requesting. In the example above, the URL is:



The server receives this request and takes one of the following steps:

- Maps the request to a file in the server's control and returns it to the client.
- Maps the request to a handler (program code) and returns the output of the handler to the client
- Determines the request can't be answered and returns an error message

The use of the '/' slash character is very much like that used as the path separator for directories and files on a file system. This pattern is a useful way to present a logical path hierarchy of the resources and content originating at the root domain `fictional_website.com`.

By allowing for multi-part paths, a logical hierarchy is created. The hierarchy can be navigated by a browser application to access different parts of the web server. The end point of a URL can be an actual file resource the web server provides, but it doesn't have to be. The logical path created has no relation to the actual file path to a resource on the server's file system.

7.4 Flask

You're going to build the MyBlog application using Flask, a lightweight web application framework for Python. Flask provides the mechanisms and plumbing necessary for Python to be used as a web application server to create useful applications that perform and scale well. Flask includes the ability to answer HTTP requests for URL resources and connect those requests to Python code that builds the response.

Flask (<https://palletsprojects.com/p/flask/>) is not part of the standard library modules that come with Python but is available as a third-party module hosted by the Python Package Index (<https://pypi.org/>). Like other modules available to Python, this makes it installable using the pip utility.

7.4.1 Why Flask

Python is in the fortunate position of being popular as a language to build web applications. Because of this, there are many tools and frameworks Python can use to create web applications, Flask being one of them. There is Django, Bottle, Pyramid, Turbogears, CherryPy, and more. All of the frameworks are useful; some have a more particular use case than others, some are faster than others, and some are more specialized for creating certain kinds of web applications and services.

Flask lives in the middle ground and is popular because it's small and has a minimal initial learning curve and is still more than capable as your skills and needs grow. There are many modules available that integrate with Flask that you'll use as the MyBlog application grows. These modules will give the MyBlog application access to databases, authentication, authorization, and form creation. Part of the beauty of Flask is not having to learn or use these expanded capabilities until they're needed, and you're ready to create new features with them.

As a developer myself, I've worked with some of the other web application frameworks available to Python. Flask is the one I've used the most and am most familiar with. That influences the choice to use it because I can write about it with confidence that I can present Flask to your best advantage and not miss details I might otherwise miss if I were to choose a framework not as familiar to me.

7.4.2 Your First Web Server

Now that you know where you're headed let's get started. The first server comes right from the Flask website quick start example and is as good as anywhere to begin getting familiar with Flask. The code for the server is available in the repository as `examples/CH_07/examples/01/app.py`:

```
from flask import Flask      #A
app = Flask(__name__)        #B
@app.route("/")              #C
def home():                  #D
    return "Hello World!"
```

#A Import the Flask system into the application

#B Create a Flask instance object passing it the name of the current file

#C This decorator connects the home function to the "/" application route

#D The home function will run when the user navigates to the "/" route

After running the install steps for the chapter examples and starting your Python virtual environment, the application is run by opening a terminal window and navigating to the `examples/CH_7/examples/01` directory. Enter the following command for Mac and Linux:

```
export FLASK_ENV=development
export FLASK_APP=app.py
```

For Windows users, enter the command:

```
set FLASK_ENV=development
set FLASK_APP=app.py
```

Once done, enter the command `flask run`, and the application will output the following text to the terminal window:

```
* Serving Flask app "app.py" (lazy loading)
* Environment: development
* Debug mode: on
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 325-409-845
```

The messages above might look ominous, but it's just informing you the web server is running in development mode, which isn't suitable for production. You'll also notice the last line informs you that to stop the application you'll need to press the CTRL-C keys. Notice also the terminal cursor doesn't return. This is because the `flask run` command is running the `app` instance in an infinite loop waiting to receive and process requests.

A server is intended to run long term and in fact would never stop unless instructed to do so. The server you've just started is in an idle state waiting to receive and process HTTP requests. The Flask development server defaults to running at IP address 127.0.0.1 on port 5000. The IP address 127.0.0.1 is known as localhost and is the loopback interface of your computer. You can create servers on this address even if you didn't have a network card installed on your computer. The port value of 5000 is just an unused port number out of the 65535 available on network interfaces. Both of these values can be configured, but the defaults are fine for now.

To interact with the server you need to open a web browser on your computer and navigate to <http://127.0.0.1:5000> as the URL and click enter. The browser will respond by printing "Hello World!" in the content window. You'll also see a log message in the terminal window where the server is running indicating the request was received and processed correctly, indicated by the 200 at the end of the log message. The 200 value is the HTTP status code for "Ok", which means the requested was handled successfully.

CONNECTING ROUTES

One of the important things to notice about the application is how Python code is connected to a valid URL route the server will respond to. The `@app.route("/")` line of code is a decorator provided by the Flask app instance and applied to the `home()` Python function. The decorator is how the `home()` function is registered with Flask and connected to the URL `/` route, and will be called when a user browses to `http:127.0.0.1:5000`. Because the route is defined the server will respond with the results of running the `home()` function, returning the "Hello World" string.

SERVING FOREVER

Once the server is running it will continue responding to requests until it's stopped, essentially running forever waiting for HTTP requests to respond to. There is no explicit loop in the application code, so how is the server running forever? The loop is part of the functionality in the Flask `app` instance. When the `flask run` command is invoked at the terminal command line it looks for an object named `app` and if found starts the server event loop.

The event loop is where the server waits for events to process. The events are data showing up on the network socket at port 5000. Unlike what you might think of as an infinite

loop in application code, the server is idle while waiting for events and using very little CPU time.

UNDEFINED ROUTES

If you go back to the browser and modify the URL to be <http://127.0.0.1:5000/something> and hit the enter key the browser will respond with a "Not Found" error. Looking at the log messages in the terminal window you'll see a message the request was received, but the server responded with a 404 status code. The HTTP protocol status code 404 essentially equates to "Page Not Found".

This makes sense if you look at your first web server application code. At the moment the only URL supported is the home route `/`, there is nothing defined to handle the `/something` route. The server didn't crash because it didn't have the route defined, instead the server handled it as an error and informed the browser about the error.

The ability to handle errors and continue to function is an important part of the server's design and implementation. As you develop the MyBlog application you'll use the errors handled and returned by the Flask server to help determine where problems exist in the application and where to go to resolve them.

7.4.3 Serving Content

Getting your first web server coded and running is a big step. There's a remarkable amount of functionality implemented and executed by the very small amount of code in `app.py`. The `home()` function shows how you can map Python code to a URL the web server will support. You can add new functions and map them to additional routes and the web server would provide additional pages the browser could navigate to.

To create a proper web page you could replace the "Hello World!" string returned by the `home()` function with a string containing HTML code. By doing this the browser would receive the HTML and render it in the browser window. However, useful and well-designed web pages are created with HTML code that can run to hundreds, even thousands, of lines of code. Embedding strings of HTML code directly into your web server would make it difficult to maintain and wouldn't take advantage of features available to you through Flask.

DYNAMIC CONTENT

The content served to the browser by the `home()` function is the string "Hello World!", which is returned to the browser every time the page is accessed or refreshed. Because `home()` is a Python function it could have returned anything, including information and data that is generated dynamically. The function could have returned the result of the `random()` function and the browser would have rendered a random value every time the page was accessed. The `home()` function could have returned the results of a calculation, data retrieved from a database or the return value of some other HTTP web based service.

Creating and returning dynamic information is one of the cornerstones of creating useful web applications, the MyBlog project being one of them. How do you merge dynamic information with HTML content that can be meaningfully rendered by browsers? Flask includes access to a template language called Jinja2. A template can be thought of as a

document that will be combined with data to produce a completed end result document. Here's an example using Python f-string formatting to illustrate the idea:

```
name = "Joe"
result = f"My name is {name}"
print(result)
My name is Joe
```

Here the variable `name` is set to the string `"Joe"` and the Python format string `f" My name is {name}"` acts as the template. The `result` variable is created, and then printed, and `"My name is Joe"` is output. Python's f-string formatting is like a small templating language, it takes in data in the form of the `name` variable and creates the resulting string output. Jinja2 works much like this as well as having many other features.

By using a templating language, you can place your HTML code in a template file and then have Jinja2 substitute your dynamic information and data into the right places in that template.

USING A TEMPLATE LANGUAGE

Let's modify the previous web server code to use Jinja2 templates and pass dynamic data to the template to render in the browser window. The modified code is found in `examples/CH_07/examples/02/app.py`:

```
from flask import Flask, render_template    #A
from datetime import datetime              #B

app = Flask(__name__)

@app.route("/")
def home():
    return render_template("index.html", now=datetime.now())    #C
```

#A Import the Flask function `render_template` in order to use Jinja2

#B Import the `datetime` functionality to generate dynamic data

#C Use the `render_template` function to connect the `index.html` template file with the `now` data element

The first parameter to the `render_template` function is the string `"index.html"`. This is the filename of a template file containing Jinja2 instructions. Everything else passed to `render_template` is a named parameter. In the example above the named parameter is `now` and it is the value returned by `datetime.now()`, the current timestamp.

By default, Flask initially searches for template files in a directory called `"templates"`. The template directory should exist in the same directory as the `app.py` file, so create that now. Inside the `"templates"` directory you'll need to create a file named `"index.html"`. In the example application the `"index.html"` file looks like this:

```

<!DOCTYPE html>      #A
<html>               #A

<head>              #A
  <!-- Required meta tags -->      #A
  <meta charset="utf-8">      #A
  <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
    #A
  <title>Your First Web Server</title>      #A
</head>             #A

<body>
  <h1>Current time: {{now}}</h1>      #B
</body>

</html>

```

#A HTML 5 boilerplate code

#B Insert the current datetime in the {{now}} Jinja output expression

The HTML 5 code above presents a complete web page the browser can render in its window. The interesting thing in the file is the `<h1>` tag inside the body of the document:

```
<h1>Current time: {{now}}</h1>
```

The `{{now}}` part of the line is Jinja2 template syntax and will be replaced by the value of the `render_template` parameter "now" that has a value of the current timestamp.

The `render_template` function uses the Jinja2 templating engine to parse the template file and substitute elements that follow the Jinja2 syntax rules with data. Jinja2 also has the capability to do much more processing than just substitution, and you'll get to that shortly.

Once you've created the "index.html" the directory structure should look something like this:

```

├─ app.py
├─ templates
│   └─ index.html

```

If you run the `app.py` file now you should get a browser page that looks something like this:

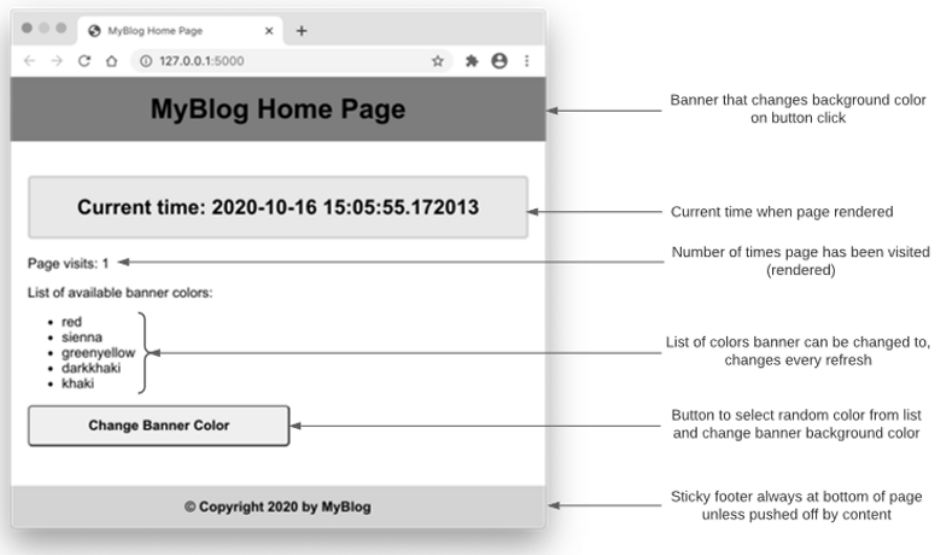


Every time the page is refreshed the timestamp will update. This demonstrates the `home()` function is running and the `"index.html"` template is being rendered with a new `datetime.now()` value every time the page is refreshed.

7.4.4 More Jinja2 Features

The previous example is a functioning web application, but it only shows off a small portion of what Jinja2 can do. Let's expand on the example web application to demonstrate more capabilities of Flask and Jinja2 that you'll make use of in the MyBlog project.

The updated web application will have a banner and a sticky footer. It will include the current time as it does now. It will also include a page visits counter that will increment every time the page is refreshed. There will also be a list of colors the banner can be changed to, and a button that will change the banner background color randomly. The updated application will look like this:



The application gets extra functionality to provide the random list of colors and the page visits counter. The server also provides styling information to the browser in the form of CSS files and client side (browser) interactivity as a JavaScript file.

The page counts ability is provided by a Python class using a class level variable. By using a class variable, the state of the page count is accessible by any instance of the class. Any change to the variable is visible to any instance that uses it. The `PageVisit` class has a simple purpose and interface:

```
class PageVisit:
    COUNT = 0

    def counts(self):
        PageVisit.COUNT += 1
        return PageVisit.COUNT
```

The `PageVisit` class maintains the class variable `COUNT` available to all instance of the class. It also provides the `counts()` method to increment the value of `COUNT` and return it to the caller. Every invocation of the `counts()` method will increment the `COUNT` value by one.

The `PageVisit` class exists because the web server can be handling many users at once who might be making requests for the page and a consistent value of `COUNT` has to be maintained across all of them.

The list of banner background colors is used by both the template file that renders the HTML of the page and the JavaScript functionality that changes the banner background colors on button clicks. This means the colors list has to be made available to the template and the JavaScript engine running on the browser.

To manage the list of colors the class `BannerColors` is created. This class encapsulates the primary list of colors as a class level variable and provides a method generating a

random subset of those colors as a list to use. Like `PageVisit`, the class `BannerColors` has a simple purpose and interface:

```
class BannerColors:
    COLORS = [
        "lightcoral", "salmon", "red", "firebrick", "pink",
        "gold", "yellow", "khaki", "darkkhaki", "violet",
        "blue", "purple", "indigo", "greenyellow", "lime",
        "green", "olive", "darkcyan", "aqua", "skyblue",
        "tan", "sienna", "gray", "silver"
    ]

    def get_colors(self):
        return sample(BannerColors.COLORS, 5)
```

The `BannerColors` class maintains the class variable `COLORS` that is a list of valid CSS color name strings. This creates the palette of banner colors that can be displayed on the page. The `get_colors()` method returns a random subset of five of those colors as a list using the `Random` module, and its `sample` function, from the Python standard library. Every time `get_colors()` is invoked it returns a random subset of colors as a list from the `COLORS` class variable list.

The `PageVisit` and `BannerColors` classes are added to the `app.py` file in the `examples/CH_07/examples/03` directory and integrated into the `home()` function that renders the web page:

```
@app.route("/")
def home():
    banner_colors = BannerColors().get_colors()    #A
    return render_template("index.html", data={    #B
        "now": datetime.now(),                    #B
        "page_visit": PageVisit(),                #B
        "banner_colors": {                        #B
            "display": banner_colors,              #B
            "js": json.dumps(banner_colors)        #B
        }
    })
```

#A Gets a random list of five colors and assigns it to the variable `banner_colors`

#B Creates a dictionary of information to pass to the template as the data named variable

The `BannerColors` class is instantiated right away and its `get_colors()` method is called storing the results in the variable `banner_colors`. This variable is used later to create the data passed to the template for rendering.

The `render_template` function is called with the name of the template to render, `"index.html"`, and the `data` variable. The `data` variable is a dictionary containing key/value pairs to pass information used in the template:

- `now` – is the value returned by `datetime.now()`
- `page_visit` – is an instance of `PageVisit()`
- `banner_colors` – is another dictionary

- `display` – is the previously create `banner_colors` list variable
- `js` – is the result of JSON stringifying the `banner_colors` list

The `banner_colors` dictionary inside the `data` dictionary contains two variations of the `banner_colors` list variable. You'll see how this is used when we get to the updated `"index.html"` template.

All of the work to this point adds functionality to the home function that's run when a user browses to the home page of the application. This new functionality passes new data to the `"index.html"` template and is rendered as a complete HTML page by Jinja2.

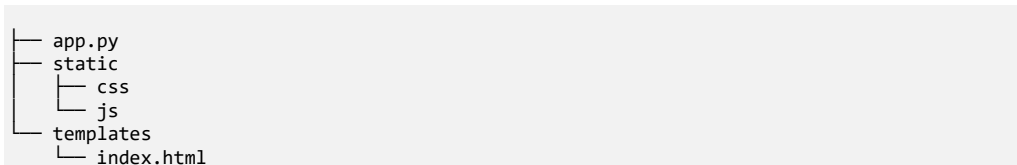
TEMPLATE INHERITANCE

Before you get into the updated `"index.html"` template, take a look at the screenshot of the web application presented earlier. The page has a banner and a footer section. Very often these kinds of visual and informative features are common to every page of a web application. HTML boilerplate code is also common to every page.

Based on the "Don't Repeat Yourself" principle (DRY), it would be useful to pull the common elements of an HTML page together rather than copying those pieces to every page. Even worse would be maintaining all those copies of the common elements as the web application changes and evolves over time. The Jinja2 templating engine provides for this through the use of template inheritance, which is conceptually like class inheritance in Python.

Before getting started using template inheritance you'll need to expand the directory structure of the web application. Because you're going to be serving static CSS and JavaScript files to the application, those files need to live somewhere the web server can access them. By default Flask looks for static files in a directory named `static`, which is a sibling of the `templates` directory.

Create the `static` directory at the same level as the `"templates"` directory. To help keep things organized in the `static` directory, create `"CSS"` and `"JS"` subdirectories as well to place the CSS and JavaScript files into. These files will give the web application it's presentation and interactivity. Your directory structure should look like this:



PARENT TEMPLATE

You can copy the `"index.html"` file to create a new file named `"base.html"` in the `templates` directory. This is the parent template containing all the common features presented on the web pages of the application. Modify the new `"base.html"` template file to look like this:

```

<!DOCTYPE html>
<html>
<head>
    {% block head %}      #A
    <!-- Required meta tags -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
    <title>Your Second Web Server</title>
    {% block styles %}    #B
    <link rel="stylesheet" type="text/css" href="{{ url_for('static',
        filename='css/myblog.css') }}">      #B
    {% endblock %}      #B
    {% endblock %}      #A
</head>

<body>
    <div id="header">
        <h1>MyBlog Home Page</h1>
    </div>
    <div id="content">
        {% block content %}{% endblock %}      #C
    </div>
    <div id="footer">
        {% block footer %}      #D
        <h4>&copy; Copyright 2020 by MyBlog</h4>      #D
        {% endblock %}      #D
    </div>
</body>

{% block scripts %}{% endblock %}      #E

</html>

```

#A Creates a template section named “head”, which will be referred to by child templates

#B Creates an inner template section named “styles”, which will be referred to by child templates to insert CSS file references

#C Creates an empty template section named “content”, which contain the content of the page and is provided by the child templates

#D Creates a template section named “footer”, which can be referred to by child templates

#E Creates an empty template section named “scripts”, which child templates can use to insert JavaScript file references

In this template there is HTML code you've seen before, mixed with Jinja2 template code. The template code that begins with `{% block head %}` and ends with `{% endblock %}` creates a template section named `head` that can be referenced by a child template by referring to the block name. Blocks like this can be referenced even from other files.

The block named `styles` contains a stylesheet link. Inside the `href` portion of the link is another Jinja2 template construct, `{{url_for('static', filename='css/blog.css')}}`. This is an expression substitution, in this case executing the Python `url_for` function.

It's generally a bad idea to hard code URL paths within a web application, and the `url_for` function helps avoid this. By passing a known endpoint as the first parameter, and the relative filepath as the second, the function can create a URL to the desired file that's valid for the Flask application. When the template is rendered a valid URL to the `blog.css` file will exist in the stylesheet link rendered by the browser.

The empty block sections named `content` and `scripts` respectively create references that will be used by the `"index.html"` file inheriting from the `"base.html"` template. The `"index.html"` file will use the references to inject content into the page and include a page specific JavaScript file named `"index.js"` containing interactivity code.

CHILD TEMPLATE

Now that you have a base template it's time to inherit from it by modifying the `"index.html"` template:

```
{% extends "base.html" %}      #A

{% block content %}            #B
<h2>Current time: {{ data["now"] }}</h2>      #B
<p>Page visits: {{ data["page_visit"].counts() }}</p>      #B
<p>List of available banner colors:</p>      #B
<ul>                            #B
    {% for banner_color in data["banner_colors"]["display"] %}      #B
    <li>{{ banner_color }}</li>      #B
    {% endfor %}      #B
</ul>                            #B
<div id="color-change">      #B
    <button class="change-banner-color">      #B
        Change Banner Color      #B
    </button>      #B
</div>                            #B
{% endblock %}      #B

{% block styles %}      #C
{{ super() }}      #C
<link rel="stylesheet" type="text/css" href="{{ url_for('static', filename='css/index.css') }}">      #C
{% endblock %}      #C

{% block scripts %}      #D
{{ super() }}      #D
<script>      #D
    const banner_colors = {{ data["banner_colors"]["js"]| safe }};      #D
</script>      #D
<script src="{{ url_for('static', filename='js/index.js') }}"></script>      #D
{% endblock %}      #D
```

#A Makes this child template inherit from the `base.html` parent template

#B Creates the content to render to the page, which will replace the empty content block in the parent template

#C Adds specific CSS style information for this child template to the styles block. The `{{ super() }}` expression calls the parent styles block first and then adds the content of this block

#D Adds specific JavaScript file references for this child template to the scripts block. The `{{ super() }}` expression calls the parent styles block first and then adds the content of this block

Like Python classes, the child template references the parent template to inherit from by referring to it in the template code at the first line. The `{% extends "base.html" %}` template code informs Jinja2 that `"index.html"` is inheriting from `"base.html"`. The template engine knows how to find the `"base.html"` template file in the same way that it found the `"index.html"` template, by looking in the templates directory.

The content provided by the "index.html" file begins with the `{% block content %}` start marker and ends with an `{% endblock %}` marker. When the complete page is rendered by Jinja2 the content will be placed on the HTML page at the position of the content block reference in the "base.html" parent template file.

Inside the content section is where the data passed to the template by the `render_template` function is used. The `{{data["now"]}}` Jinja2 expression gets the current timestamp. The `{{data["page_visit"].counts()}}` expression gets the `PageVisit` instance and call its `counts()` method to get the current page visit counts.

The Jinja2 language provides a mechanism to create repeating data in the rendered template by using a for loop. Mimicking Python, the loop construct is a For-In loop iterating over the contents of the `data["banner_colors"]["display"]` list. Each item in the list is used to create an html list element with the code `<li>{{banner_color}}</li>`. The for loop ends with the `{% endfor %}` marker.

The block named `styles` references the same block in the "base.html" template. Recall that the `stylesheet` block in the parent template wasn't empty, it had a `stylesheet` link to pull in presentation information common to all web pages. The `{{super()}}` expression renders the parent `stylesheet` block before including the information defined in the child "index.html" template.

The `scripts` named block does a couple things. It uses the `{{super()}}` expression to render anything defined by the parent template (nothing currently). It then builds some JavaScript code directly to define a variable called `banner_colors` which is initialized with the JSON formatted string of banner colors supplied by `{{data["banner_colors"]["js"] | safe}}`. The `| safe` part of the syntax prevents Jinja2 from translating symbols that could be dangerous. Lastly a script tag referencing an external JavaScript file using the same `url_for()` mechanism to create a valid relative URL for the web application is included.

PRESENTATION

The presentation of the web page is controlled by the "index.css" file, which contains CSS code applying particular style information to the HTML elements created by the Flask `render_template` function that is presented by the browser. There are two CSS files connected to the application, "myblog.css" and "index.css".

The "myblog.css" file applies to the parent template file "base.html":

```

html, body {
  height: 100%;
}

body {
  display: flex;
  flex-direction: column;
  margin: 0px;
  font-family: Arial, Helvetica, sans-serif;
}

#header h1 {
  margin: 0px;
  background-color: darkcyan;
  height: 75px;
  text-align: center;
  line-height: 75px;
}

#content {
  flex: 1 0 auto;
  margin: 20px;
}

#content h2 {
  border: 3px solid lightgray;
  border-radius: 5px;
  padding: 20px;
  text-align: center;
  background-color: bisque;
}

#footer {
  flex-shrink: 0;
}

#footer h4 {
  margin: 0px;
  background-color: lightgrey;
  height: 50px;
  text-align: center;
  line-height: 50px;
}

```

Though this book isn't about Cascading Stylesheets, it's worthwhile going over some of the above code to get a feel for how CSS code affects the presentation of the web application. Keep in mind the spacing and indentation of the CSS code is a convention for readability and is not part of the required syntax.

CSS code is about using and creating selectors to attach specific style information to so the browser can render the HTML elements with the intended look and feel. For example, the code `#content h2 {...}` attaches style rules to the HTML header `<h2>` element contained in the `<div id="content">` element. This narrows where the style will be applied on the page, in this case the `<h2>` tag within the `<div id="content">` will have a rounded border with an internal padding of 20 pixels, the header text will be centered and have a background

color of “bisque”. The rest of the selectors apply style rules to other parts of the “base.html” page.

The “index.css” file applies rules to the “index.html” child template page:

```
#color-change button {
  background-color: lightgrey;
  border-radius: 5px;
  border: 1px solid grey;
  display: inline-block;
  cursor: pointer;
  color: black;
  font-family: Arial;
  font-size: 16px;
  font-weight: bold;
  padding: 13px 69px;
  text-decoration: none;
  text-shadow: 0px 0px 0px lightskyblue;
}

#color-change button:hover {
  background-color: darkgrey;
}

#color-change button:active {
  position: relative;
  top: 1px;
}
```

The selectors in the code above apply styles to HTML elements that are created by the “index.html” page, essentially giving some style and CSS interactivity to the color change button.

INTERACTIVITY

This book isn’t about JavaScript, and its use will be kept to a minimum, but interesting web applications will include some JavaScript code. Once the HTML of page is rendered it is sent to the browser as a response. The browser will then render the HTML visually in the browser window. It will also parse and compile the JavaScript sent in the response and pulled in from the external file:

```
window.addEventListener('load', function (event) {    #A
  let banner = document.querySelector("#header h1");    #B
  window.addEventListener('click', function (event) {    #C
    // is this the click event we're looking for?    #C
    if (event.target.matches('.change-banner-color')) {    #D
      let color = banner_colors[Math.floor(Math.random() * banner_colors.length)];
      #E
      banner.style.backgroundColor = color;    #F
    }
  })
});
```

#A Wait for the page to be loaded before executing the nested code

#B Get a reference to the banner element

#C Add a click event handler for the banner color changing button

#D Check to see if the click event originated from the button

```
#E Select a random color from the banner_colors list
#F Change the banner background color
```

The code above is vanilla JavaScript code (no frameworks or jQuery involved) to add an action if the displayed button is clicked. The code creates an anonymous function to run when the page is finished loading. The anonymous function creates a reference to the banner element and then adds another anonymous function to listen for the "click" event.

Inside the "click" event handler a conditional checks if the event was generated by the change banner color button. If so, a random color is selected from the `banner_colors` list and is used to change the background color of the banner.

7.5 Running the Web Server

The updated app exists in the `examples/CH_07/examples/03` directory and the directory structure looks like this:

```
├── app.py
├── static
│   ├── css
│   │   ├── index.css
│   │   └── myblog.css
│   └── js
│       └── index.js
└── templates
    ├── base.html
    └── index.html
```

In the terminal move to the directory and set the environment variable `FLASK_APP` to point to your application by entering this at your command line for Mac and Linux:

```
export FLASK_ENV=development
export FLASK_APP=app.py
```

And this for Windows users:

```
set FLASK_ENV=development
set FLASK_APP=app.py
```

Then run the web server by entering `flask run` at the terminal command line. You should see the server start up and be able to navigate to `http:127.0.0.1:5000` to see the application.

When you run the web application with the `flask run` command the server starts and runs with the Flask built-in web server. The built-in web server is suitable for development and trying things out, but it is not suitable for production use.

For production use you'll need to use a WSGI server. WSGI stands for Web Server Gateway Interface, and a WSGI server is an application that provides a simple calling convention to forward requests from a web server to a Python web application. The web server built into Flask is a WSGI server that provides this calling convention.

The WSGI standard exists to abstract away the complexities of interfacing your Python web application to a web server and the world. So long as you're building your application to

the WSGI interface standard, which Flask and just about all other Python web frameworks do, your application can provide request/response handling.

Two of the most common production grade WSGI servers are uWSGI and Gunicorn. The uWSGI application is a high-performance application written in C/C++. Gunicorn' Green Unicorn' is also a high-performance WSGI compliant web server application. Both are production ready.

7.5.1 Gunicorn

To run your application using Gunicorn you need to install it using this command from your Python virtual environment:

```
pip install gunicorn
```

In one of the example application directories in the terminal enter this command:

```
gunicorn -w 4 app:app
```

This tells Gunicorn to start four worker instances of your application, which it finds with the "app:app" part of the command. The first part is the name of the Python file "app.py" and the second part ":app" refers to the application instance created with the `app = Flask(__name__)` part of the code. Naming the Flask application instance "app" is common for a WSGI application.

Running multiple instances of the application with Gunicorn workers allows your application to scale up to handle hundreds, even thousands of requests per second. This is dependent on the workload each request makes of the application and how much time it takes before a response is generated.

According to the Gunicorn documentation the recommended number of workers for an application running on a single production server is $(2 \times \text{number_of_cores}) + 1$. The formula is loosely based on the idea that for any given core one worker will be performing IO (Input/Output) operations and the other worker will be performing CPU operations.

7.5.2 Commercial Hosting

When you want to make your web application available for public use, you might need to do so using a commercial hosting service. There are many services available to host your application. They will offer options like Apache or Nginx for web serving, and uWSGI and Gunicorn for WSGI interfacing with your Python based web application. It's also possible to deploy your application using Docker containers.

I'm sure there are more options and configurations than listed here. Which to choose depends on you, your goals for the application and costs.

Because of the wide array of choices available and combinations afforded by those choices, I'm not going to spend time defining how to deploy a Flask based Python application to specific examples. My reasons for doing so are twofold:

- It's unlikely I would hit upon a combination of choices that would suit your deployment use case perfectly
- Deploying an application is a big topic worthy of a book of its own, and doesn't

directly correlate to becoming a well-grounded Python developer

7.6 Summary

You've learned a great deal in this chapter:

- You've been introduced to the main project application, MyBlog, and how you're going to implement it as a web application.
- You've seen in general what a server does and what a web server in particular does.
- You've also created your first three web applications with Flask with steadily growing complexity and abilities.

The next chapter will begin to build the groundwork for the MyBlog application that will grow through the rest of the book to become full featured. Along the way you'll learn how to handle the development of a larger application, how to integrate it with a persistent database and the ways in which what you've learned so far can make the project manageable to build and enjoyable to achieve.

8

Getting started with MyBlog

This chapter covers

- Application Styling
- Integrating Bootstrap Styling
- Creating a Scalable MyBlog
- Using Blueprint Namespaces
- Application Configuration
- Integrating the Flask DebugToolbar
- Configuring Logging Information

Creating a web application pulls together many concepts and technologies. To create an engaging application it's necessary to think about the look and feel, or style. For a web application this is largely provided by the user of CSS styling applied to the HTML content.

Integrating good styling practice raises the bar of complexity an application encapsulates. To help maintain growing complexity it's necessary to think about project structure and the use of namespaces. Project structure and namespaces helps the project grow and scale in a way that keeps the complexity manageable.

This chapter lays the foundation for the MyBlog application so that it can grow and evolve in a way that will help you maintain clarity about the goals of the application and stay ahead of complexity.

8.1 Handling Application Styling

Creating an application with interesting and useful features is necessary to keep users actively engaged in any application you create. The feature set is essential, but it's not the only thing needed to capture and keep users' attention. The way your application looks is a critical factor your users will expect in a modern computer system.

Look at any popular cell phone app, and you'll know this is true. The best apps have both useful feature sets and an engaging visual experience for their users. Even apps with compelling features will be hard-pressed to find users willing to accept and use them if the app looks clunky and unpolished.

The browser's CSS styling code controls a web application's visual look to determine how to render the HTML code to the screen. The first web application from Chapter 7 used hand-crafted CSS code to apply a distinct look and feel.

Continuing to hand-code CSS styles for the MyBlog application is possible, but continuing on that path has drawbacks:

1. It takes effort to create appealing CSS styles
2. As an application grows, it becomes challenging to maintain styling consistency
3. Normalizing styles across multiple browsers is tricky
4. Phones and tablets are becoming the primary interface between an application and your users; making a web application responsive to those devices is vital

8.1.1 Creating Appealing Styles

The first web application from Chapter 7 had a single page with relatively simple styles applied to it consisting of two files, `myblog.css` and `index.css`, each of which was about a page of text long. Continuing to create custom styles leads to many CSS files containing hundreds, possibly thousands of lines of code. It becomes apparent very quickly that doing this is additional hard work.

8.1.2 Styling Consistency

A web application with even moderate complexity will have multiple pages associated with it that users will navigate around. Making sure the buttons, lists, panels, and other visual elements look the same across all those pages is important to the cohesive picture you're trying to paint across the entire application.

Making those visual elements have the same style, even in different use cases within the application, is challenging. Maintaining that consistent style as your application scales upward with more features and pages compounds that challenge.

8.1.3 Normalizing Styles

Like myself, you probably use a single web browser most of the time. Even if you use one at home and another at work, you might be unaware of point 3 above, the need to normalize styles across browsers. When you're building an application and generating HTML code without any styles applied to it, the browser will render the HTML using its default styles. Any publicly available web application has style information applied to it.

Each browser applies its default style to headers, paragraphs, fonts, and spacing between elements. If you could suppress the CSS when navigating to web applications you're familiar with, you'd see how the pages are rendered with the browser's default style. If you were to do the same thing using multiple browsers like Chrome, Firefox, Safari, Edge, and others across multiple operating systems, you'd see the pages rendered differently between those browsers. Sometimes subtly, sometimes dramatically.

Giving any web application you create a consistent look across the different browsers, and operating systems means creating CSS code to override those browser's default styling.

8.1.4 Responsive Design

We've already talked about the advantages of providing application features using web browsers as the platform to deliver the experience. One of the implications is that web browsers are everywhere, on mobile phones, tablets, laptops, and desktops. They are even integrated into cars and household appliances. These devices can and do have varied presentation capabilities, including screen size, screen resolution, speed, and accessibility.

Because the internet is widely available, this means your web application can run on any of these devices that can access the URL where your application is available. Attempting to style your application for all of these devices is impossible; there are just too many of them. Also, new ones with new capabilities are continually being introduced.

Since it is impossible to code for so many devices, it's necessary to use responsive design principles. Responsive design means using fluid, proportion-based grids, media queries, and flexible images. Using these tools creates a design layout that automatically adjusts to the screen size of the device.

8.2 Integrating Bootstrap

I wouldn't present the styling speedbumps involved in developing the MyBlog web application without proposing a solution. The solution I've chosen is the Bootstrap CSS framework created by Twitter and located here: <https://getbootstrap.com/>. The Bootstrap framework addresses the problems raised above and frees you from solving many styling issues, and instead lets you focus on the application design and implementation.

Adopting Bootstrap relieves you from having to create the CSS style code that creates the MyBlog presentation. Using Bootstrap gives the MyBlog application an attractive, consistent user interface normalizes that interface across browsers and operating systems, and resolves many responsive design issues.

The use of Bootstrap means you'll still need to add CSS class names to the MyBlog web application's HTML elements, but the reduction in the amount of custom CSS code required is well worth it.

To use it well, you'll have to learn its conventions and use cases. Using Bootstrap presents a learning curve in addition to what you're already learning about Python and web development. The effort to climb that learning curve is worth it for what Bootstrap brings to the process.

Using Bootstrap to style your web-based applications also means your applications will have a Bootstrap "look" to them. This "look" can be advantageous because the visual presentation is attractive, well known, and understood. The choice to use Bootstrap is a good one because this book focuses on becoming an accomplished Python developer.

Using Bootstrap doesn't exclude you from completely customizing your design, and it provides an excellent jumping-off point if you want to pursue the possibilities offered by customizing your application design.

8.2.1 Bootstrap Version

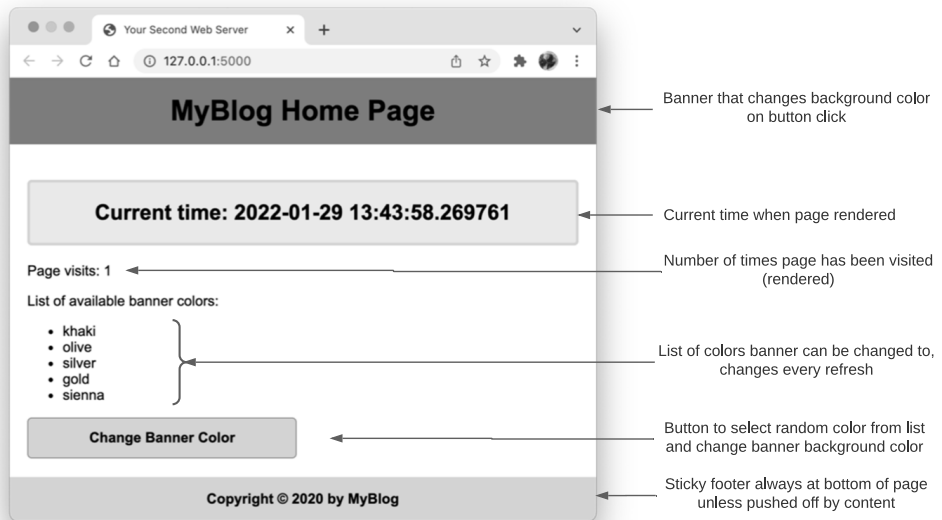
There are two Flask extensions that integrate Bootstrap into Flask applications, Flask-Bootstrap4 4.0.2, last updated in 2018, Flask-Bootstrap 3.3.7.1, last updated in 2017. The MyBlog application will integrate Bootstrap directly rather than using either of these extensions. You'll be using the Bootstrap 5 Beta version (as of this writing) for a couple of reasons. Version 5 focuses more on modern browsers, making the CSS code more future-friendly, more straightforward to implement, and possibly faster to render.

Version 5 no longer supports Microsoft Internet Explorer (IE) browsers. Dropping the support eliminates legacy code that's hard to support and held back the development of interesting features on modern browsers.

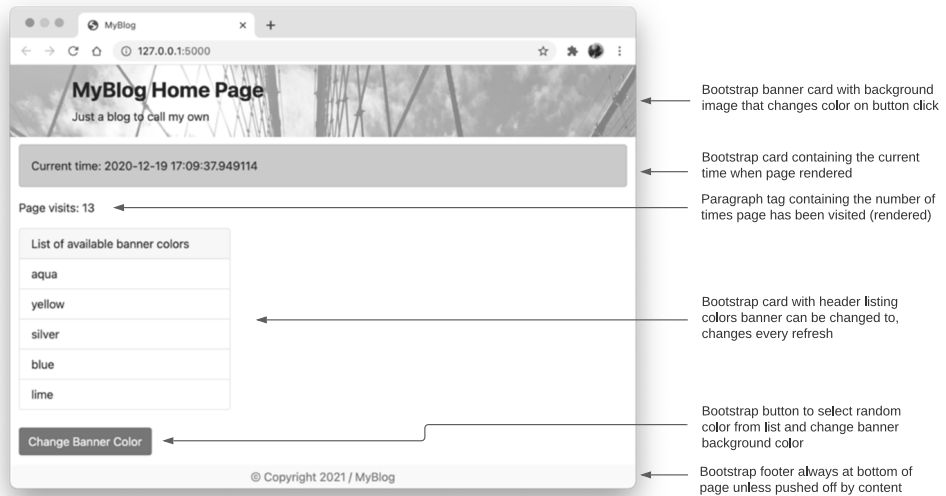
The framework also drops any dependency on the jQuery JavaScript library and instead uses straight JavaScript. There is absolutely nothing wrong with jQuery; it's a powerful library to access and manipulate HTML elements. However, since its creation, the JavaScript browser's support has become much more consistent and powerful, making dependence on jQuery a choice rather than a necessity. Using Bootstrap version 5 reduces dependencies in the MyBlog application and makes developing the app less opinionated.

8.2.2 Previous Example Now With Bootstrap!

You've read quite a bit about using the Bootstrap framework and why you'd want to do so; let's get to using it. The last example web application in Chapter 7 used hand-coded CSS for styling. As a refresher, that version of the MyBlog application looks like this:



Your initial goal is to replace all of the hand-coded CSS style information with that supplied by Bootstrap. It will be the same application with the same content and functionality, but styled entirely with Bootstrap CSS style classes to look like this:



The first step on your way to replacing the hand-coded CSS styles with Bootstrap is making the Bootstrap framework available to the MyBlog web application. It's possible to incorporate the Bootstrap framework using a CDN (Content Delivery Network), a good solution. However, it's unknown how long a particular version of a file delivered by a CDN network will be available. Since this book will have some longevity, you'll download the Bootstrap files and serve them with the MyBlog web application.

The file `bootstrap.min.css` goes in the `static/css` directory, and the file `bootstrap.bundle.min.js` goes in the `static/js` directory. The Bootstrap files are already part of the example code for Chapter 8 and will be going forward.

BASE.HTML

You're changing just the style of the application, so the `app.py` file doesn't need to change. Because Bootstrap will be used for all of the MyBlog application you'll add it to the `base.html` template, making it available to any template that inherits from it. You'll also need to update the hand-coded style information in the template file with Bootstrap style classes. The `base.html` template file is updated as follows:

```

<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>MyBlog</title>
  {% block styles %}
  <style>      #A
    :root {      #A
      --background-url: url({{url_for("static", filename=
        "images/myblog_banner_50.png")}});      #A
    }      #A
  </style>      #A
  <link rel="stylesheet" type="text/css" href="{{ url_for('static',
    filename='css/bootstrap.min.css') }}">      #B
  <link rel="stylesheet" type="text/css" href="{{ url_for('static',
    filename='css/base.css') }}">
  {% endblock styles %}
</head>
<body class="d-flex flex-column h-100">      #C
  <div class="banner card">      #C
    <div class="card-body">      #C
      <div class="col-md-4 offset-md-1">      #C
        <h2 class="card-title fw-bold">MyBlog Home Page</h2>      #C
        <p class="card-text">Just a blog to call my own</p>      #C
      </div>      #C
    </div>      #C
  </div>      #C
  <main class="flex-shrink-0">      #D
    {% block content %}{% endblock %}
  </main>
  <footer class="footer mt-auto py-1 bg-light">      #E
    <div class="container text-center">
      <span class="text-muted">&copy; Copyright 2021 / MyBlog</span>
    </div>
  </footer>
  {% block scripts %}
  <script src="{{ url_for('static', filename='js/bootstrap.bundle.min.js') }}"></script>
  #F
  {% endblock %}
</body>
</html>

```

#A Creates the CSS variable background-url referencing the banner image, done here because it uses url_for() in a template processed by Jinja2

#B Includes the Bootstrap minimized CSS file from the static directory

#C Creates the banner section content and its styles

#D Creates the content section enclosing style, which is provided by child templates

#E Creates the Bootstrap footer that sticks to the bottom of the page

#F Includes the Bootstrap bundled, minimized JavaScript code

The template code forms the content and style basis of the MyBlog application. Every template that inherits from this one will have these content and style elements. The `base.html` template gives a foundation for the look and feel of the entire application as well as easing the work of doing so.

BASE.CSS

The `base.html` template has its own `base.css` file that overrides some of the Bootstrap styling for the banner. It also has a CSS media query that makes the application responsive in a specific way for smaller devices. We'll go over the `base.css` file after presenting it:

```
.banner.card {      #A
  border: 0;        #A
  border-radius: 0;  #A
  background-clip: none;  #A
}                  #A

.banner {          #B
  display: none;    #B
}                  #B

@media (min-width: 768px) {  #C
  .banner {        #C
    display: block;  #C
    background: var(--background-url) no-repeat center center / cover;  #C
  }                #C
}                    #C
```

#A Modifies the Bootstrap card style to remove the border, make the radius 0 and removes the background image clipping

#B Sets the default visibility of the banner to none, or invisible

#C Uses a CSS media query to override the `.banner` setting to be visible if the screen size is greater than 768 pixels

The interesting part of the above `base.css` file is the `.banner` and `@media` sections. The first sets the `.banner display` value to `none`, preventing it from being rendered to the display. The `@media` section takes advantage of the cascading nature of CSS to affect how the banner is displayed. Remember, the last defined style overrides any previously defined style. The `@media` section acts as a conditional statement in the CSS. If the screen size is greater than 768 pixels, then set the display value to `block`, meaning it will be rendered to the display. The background portion defines how to display the background image and gets that image from the CSS variable `background-url` defined in the `base.html` template.

If the screen size is less than 768 pixels, the previous `.banner` definition stands, and the banner image isn't rendered to the display. The `@media` query gives MyBlog control over displaying the banner image, giving small screen devices more display real estate to show MyBlog content. You'll see an example of how the media query affects the display after going over the changes to the `index.html` file.

INDEX.HTML

As before, the `index.html` holds the content of the home page. The content is the same, but like the `base.html` file, the styling information is updated to use Bootstrap. Here's the updated `index.html` file:

```
{% extends "base.html" %}

{% block content %}      #A
    <div class="container-fluid">      #A
        <div class="card bg-warning mb-3 font-weight-bold" style="margin-top: 10px;">      #A
            <div class="card-body">      #A
                Current time: {{ data["now"] }}      #A
            </div>      #A
        </div>      #A
        <p>Page visits: {{ data["page_visit"].counts() }}</p>      #A
        <div class="card" style="width: 18rem;">      #A
            <div class="card-header">      #A
                List of available banner colors      #A
            </div>      #A
            <ul class="list-group list-group-flush">      #A
                {% for banner_color in data["banner_colors"] %}      #A
                    <li class="list-group-item">{{ banner_color }}</li>      #A
                {% endfor %}      #A
            </ul>      #A
        </div>      #A
        <br />      #A
        <button id="change-banner-color" type="button" class="btn btn-primary">      #A
            Change Banner Color      #A
        </button>      #A
    </div>
{% endblock %}      #A

{% block scripts %}
    {{ super() }}
    <script>
        const banner_colors = {{ data["banner_colors"] | tojson | safe }};
    </script>      #B
    <script src="{{ url_for('static', filename='js/index.js') }}"></script>
{% endblock %}
```

#A Create the home page content, styled using a Bootstrap responsive container and card

#B Convert the passed template parameter `data["banner_colors"]` template parameter to JSON so it can be used by the page's JavaScript

The home page content changes are all about styling the content presentation and not about the content itself.

When the changes are complete, the directory structure should be that of `examples/CH_08/examples/01:`

```
.
├── app.py
├── static
│   ├── css
│   │   ├── base.css
│   │   └── bootstrap.min.css
│   ├── images
│   │   ├── myblog_banner.png
│   │   └── myblog_banner_50.png
│   └── js
│       ├── bootstrap.bundle.min.js
│       └── index.js
└── templates
    ├── base.html
    └── index.html
```

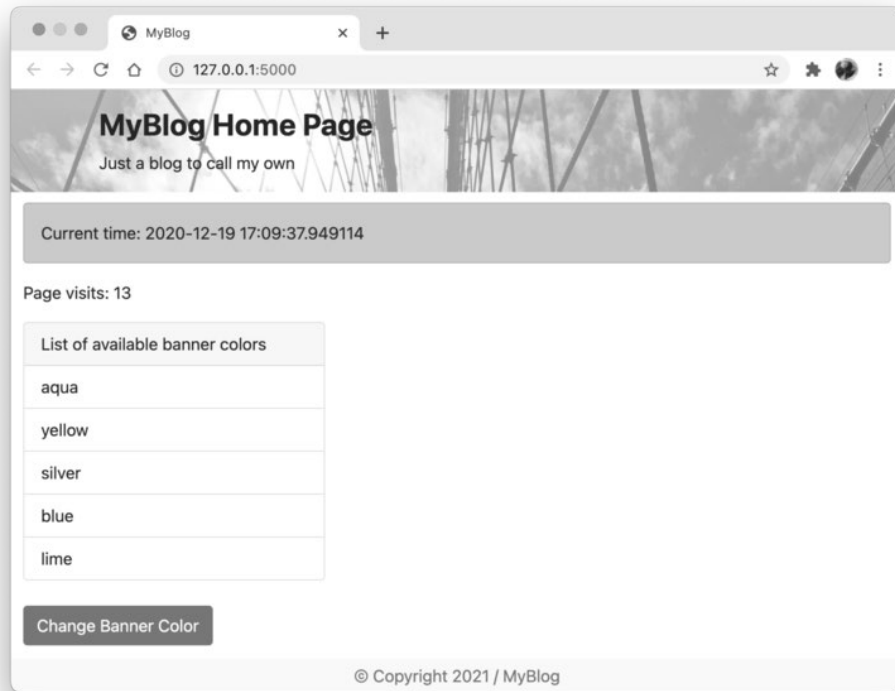
From that directory, if you execute the following commands from a Mac or Linux terminal:

```
export FLASK_ENV=development
export FLASK_APP=app.py
flask run
```

Or for a Windows system terminal:

```
set FLASK_ENV=development
set FLASK_APP=app.py
flask run
```

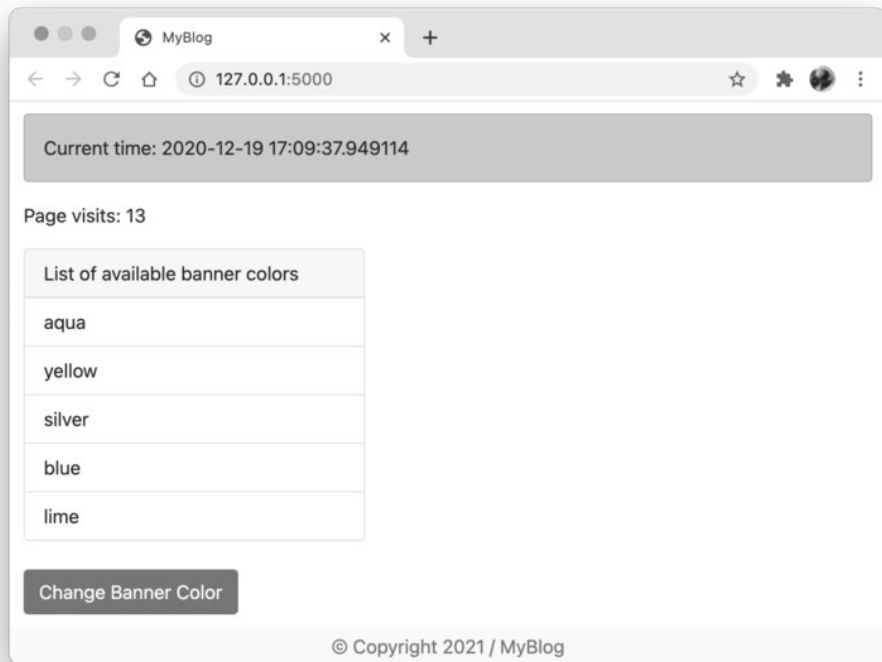
The MyBlog web server will run, and you can navigate with your browser to 127.0.0.1:5000 and see the application:



The application shows the modified banner containing the image at the top of the display. It also shows the Bootstrap sticky footer at the bottom of the page. These elements come from the `base.html` template and will be present on every page that inherits from it.

The current timestamp is displayed in a Bootstrap card, and the list of available colors is contained in another card. Additionally, the button that changes the color of the banner background is styled as a Bootstrap button.

The above display is rendered if the browser screen size is greater than 768 pixels, most likely correct for a desktop or laptop computer. If you resize the browser window to be narrower, eventually you'll cross the 768 pixel boundary, and the web application display will change to this:



The absence of the banner text and image demonstrates the conditional implied by the `@media` query in the `base.css` file. The conditional becomes false, and the previous definition of the `.banner` CSS class became active, setting the display value to `none`. Using the media query gives the web application more vertical screen real estate for smaller devices like tablets and mobile phones.

8.3 Helping MyBlog Grow

At this point in the development, the MyBlog app has expanded on the basic Flask example application often found in Flask documentation:

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, World!'
```

Everything added extends what this basic example can do with the addition of functionality, styling, and the use of Jinja2 templates. Continuing to extend this basic

framework's capabilities is possible, but doing so would hinder developing a fully-featured and extensible application.

All new functionality would have to be included in the `app.py` file as a long list of functions decorated with `@app.route(...)` to connect that functionality to the Flask application. Doing so breaks the concept of single responsibility and makes naming the web application URL endpoint function handlers awkward.

Harder would be the mental demand of working in many technical domains in a single large file. The `app.py` file would contain all of the parts necessary for a full-featured blogging application; authentication, authorization, database access, user management, and presenting the blogging content.

You've seen how to break up functionality along logical or complex boundaries by using modules to create namespaces and create namespaced containers for functionality. The same approach will be used in the MyBlog application. However, one thing that needs attention to make that possible as you move forward, the Flask app instance.

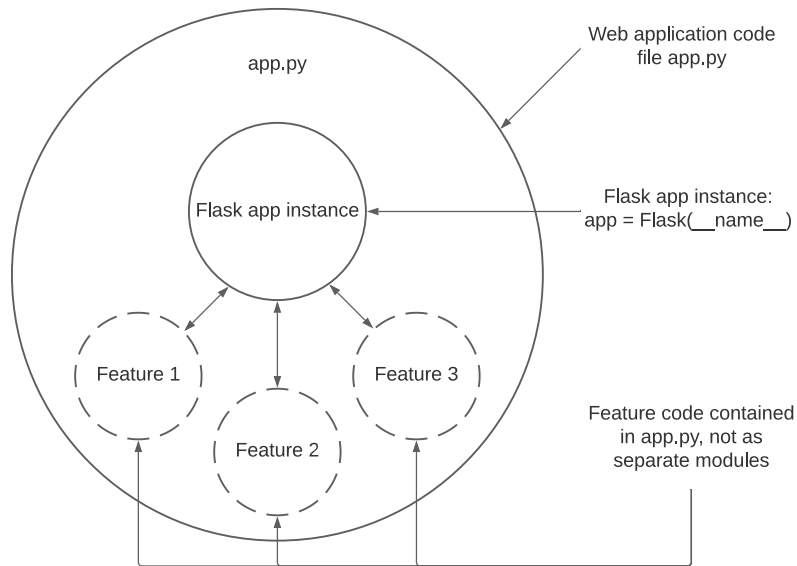
8.3.1 The Flask app Instance

In the current version of the `app.py` program file, the Flask module is imported, and the Flask application instance variable `app` is created directly. Creating the Flask app instance in the root application file works fine for a sample application whose purpose is to build a working example webserver quickly.

Why is this structure a problem when you want to use modules to namespace functionality in the web application? It's the `app` instance variable. Any feature or functionality you'd want to add to the web application with a URL endpoint function like `hello_world()` will need access to the `app` instance variable.

With the current MyBlog application setup creating modules to contain features and functionality becomes difficult because those modules will need access to the `app` instance. That presents a problem, the `app.py` code could import modules to access additional functionality, but those same modules would need access to the `app` instance. If those modules import the `app` instance from `app.py` to gain access, it creates a circular reference problem Python won't allow.

Adding features and functionality to the current implementation of MyBlog leads to a structure something like this:



RESOLVING THE APP INSTANCE PROBLEM

To resolve the problem of modules accessing the Flask app instance, you'll need to change the application's structure. The Flask app instance is the central hub around which the features of a Flask based application revolve. Besides the `@app.route(...)` functionality seen so far, there is more utilized in later chapters.

Given its central role, how do you gain access to it when needed? You'll be following a two-step process to resolve the problem: putting the bulk of the application code into a Python package and creating a factory function to instantiate the Flask app instance.

The use and creation of packages were presented in earlier chapters and used to create module namespaces. As a refresher, creating a package means creating a directory and adding an `__init__.py` file to the directory. The existence of this file lets Python import modules from the package. Adding packages can continue to any reasonable depth by creating packages within packages to create meaningful namespace hierarchies.

In earlier chapters, the `__init__.py` file need only exist to make a directory a Python package. Part of the activity of importing a module from a package also includes executing any Python code in the `__init__.py` file. The `__init__.py` file in many packages doesn't contain any code, but it's possible to add Python code to it.

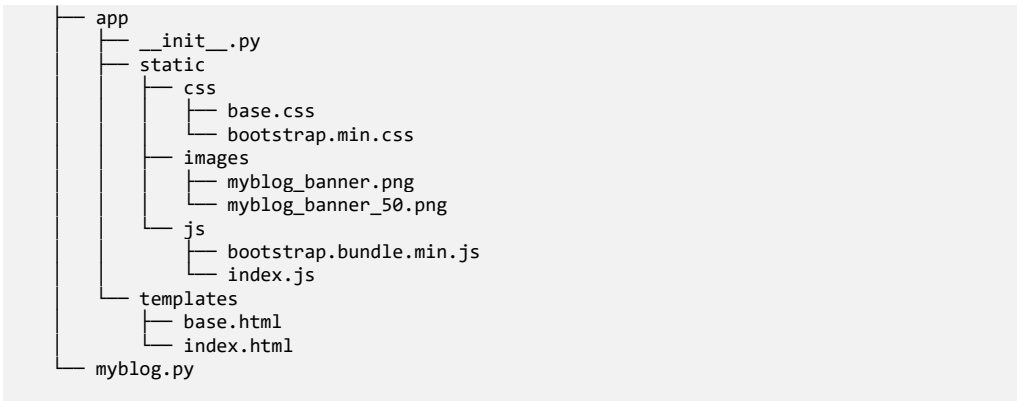
Any module in the package automatically has access to the code and variables in the package `__init__.py` file, and the `__init__.py` file has access to the package's sibling modules. Packages will be useful when creating the application factory function to create the Flask app instance.

MYBLOG RESTRUCTURING

You're at an excellent point to restructure the file layout of the MyBlog application to create a meaningful hierarchy. Having an intentional file structure helps keep using the files contained in the directory structure related and useful to your projects.

The first thing to do is rename the `app.py` file to `myblog.py`. Then create a directory named `app`, which is the root package directory for the MyBlog application. Move the `static` and `templates` directories into the `app` directory.

Inside the `app` directory, create an `__init__.py` file, which turns the `app` directory into a Python package. The directory structure should now look like this:



APPLICATION FACTORY

The newly renamed `myblog.py` file creates the Flask app instance directly and then uses it to connect URL endpoints to functionality. One goal is to get more control over creating the app instance and make using it with external modules easier. To do this, you're going to implement an application factory function called `create_app()` inside the `app` package `__init__.py` file. Edit the `app/__init__.py` file and add this code:

```

from flask import Flask, render_template
from datetime import datetime
from random import sample

class PageVisit:    #A
    COUNT = 0
    def counts(self):
        PageVisit.COUNT += 1
        return PageVisit.COUNT

class BannerColors:    #A
    COLORS = [
        "lightcoral", "salmon", "red", "firebrick", "pink",
        "gold", "yellow", "khaki", "darkkhaki", "violet",
        "blue", "purple", "indigo", "greenyellow", "lime",
        "green", "olive", "darkcyan", "aqua", "skyblue",
        "tan", "sienna", "gray", "silver"
    ]
    def get_colors(self):
        return sample(BannerColors.COLORS, 5)

def create_app():
    app = Flask(__name__)    #B

    with app.app_context():    #C

        @app.route("/")
        def home():
            return render_template("index.html", data={
                "now": datetime.now(),
                "page_visit": PageVisit(),
                "banner_colors": BannerColors().get_colors()
            })

    return app    #D

```

#A The support classes used by the home() function

#B Creates the Flask app instance inside the application factory create_app() function

#C Begin a context manager to initialize the rest of the app

#D Return the app instance to the caller

The code above is basically everything currently in the `myblog.py` file placed inside the app package `__init__.py` file. Because this replicates almost all of the code in `myblog.py`, that file needs to be updated:

```

from app import create_app

app = create_app()

```

All that `myblog.py` does now is import the application factory `create_app` and then calls it to create the Flask app instance. Change your working directory to `examples/CH_08/examples/02` and execute the commands to run the application from a Mac or Linux terminal:

```

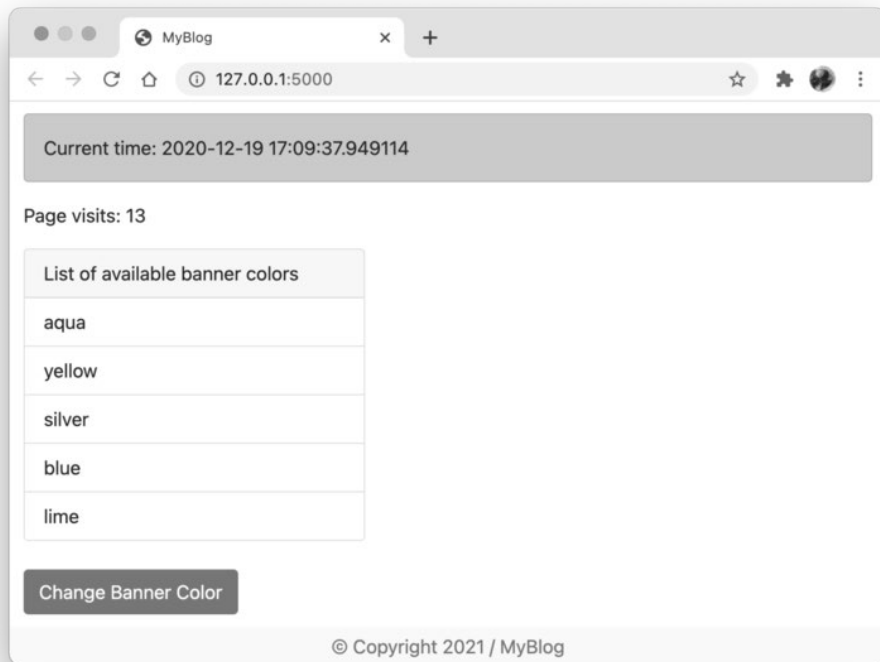
export FLASK_ENV=development
export FLASK_APP=app.py
flask run

```

Or for a Windows system terminal:

```
set FLASK_ENV=development
set FLASK_APP=app.py
flask run
```

Navigating to 127.0.0.1:5000 in your browser will show the same application view as before, but using the new MyBlog application structure:



The next step is to move the `home()` function to an external module where other features can be added. To do this, you'll make use of the Flask Blueprints capability.

8.4 MyBlog Namespaces

Any interesting application will have many features that necessitate interaction between those features, which adds complexity. The MyBlog web application is no different and will acquire functionality focused on particular application parts. Rather than roll all the other functional areas of code into one big Python file, it's better to break the work into modules so you can work in one technical domain at a time.

Just like you've seen in previous chapters where Python packages and modules contain high-level namespaces, the same idea will be applied here. In a web application, this separation of concerns not only includes creating modules to manage and namespace functionality to create, but those modules also namespace the URL endpoints they are intended to handle.

8.4.1 Flask Blueprints

The Flask web framework provides a feature called Blueprints to implement modules that service URL endpoints or routes. In the MyBlog application, when the `@app.route("/")` decorator is applied to a function, the function is registered with the server, so it will be called when the URL endpoint path "/" is called. As the decorator name implies, the "/" URL path parameter passed in is the application's route being defined.

Flask Blueprints lets you separate specific functionality into modules. A Blueprint is useful when creating logically distinct features like authentication, authorization, and other parts of a web application. In chapter 9, you'll be adding user authentication to the MyBlog application. Authentication secures access to almost all other pages and features MyBlog provides to users.

Getting authentication working and contained in one place is valuable, but this functionality could be used in an entirely different web application with little trouble once complete. You could make the authentication functionality available to other projects by placing it in a shared library for all your application projects to use. You could also put it in a repository like GitHub, where a team of developers could access it. Lastly, despite years of effort to create plug-and-play software components and methodologies to automate code reuse, one of the most commonly used patterns to reuse code is merely copying and pasting.

8.4.2 Add Blueprints To MyBlog

The MyBlog application creates the app instance inside the application factory `create_app()` function in the `app` package. The home page is also contained with the `create_app()` function. A little further in this chapter, you're going to add an about page to the application. Let's create a namespace for the home page and future about page where they both can live. We'll call that namespace "intro" as the home and about page will be introductory to the MyBlog application.

To do that, you'll create an `intro` package within the `app` package. As a first step, create an `intro` subdirectory within the `app` directory. In that subdirectory, create an empty `__init__.py` file, which makes the `intro` directory a Python package.

THE CONTENT

There's a bit of a chicken and egg problem presenting how a Blueprint is created and used, but let's start with the `intro` namespace's functionality. In the `intro` subdirectory, create an `intro.py` file with this content:

```

from flask import render_template
from datetime import datetime
from random import sample
from . import intro_bp    #A

class PageVisit:
    COUNT = 0

    def counts(self):
        PageVisit.COUNT += 1
        return PageVisit.COUNT

class BannerColors:
    COLORS = [
        "lightcoral", "salmon", "red", "firebrick", "pink",
        "gold", "yellow", "khaki", "darkkhaki", "violet",
        "blue", "purple", "indigo", "greenyellow", "lime",
        "green", "olive", "darkcyan", "aqua", "skyblue",
        "tan", "sienna", "gray", "silver"
    ]

    def get_colors(self):
        return sample(BannerColors.COLORS, 5)

@intro_bp.route("/")    #B
def home():
    return render_template("index.html", data={
        "now": datetime.now(),
        "page_visit": PageVisit(),
        "banner_colors": BannerColors().get_colors()
    })

```

#A Import the as yet undefined intro blueprint instance intro_bp

#B Notice how the home function is decorated with the intro_bp blueprint instance route function rather than @app.route

THE INTRO PACKAGE

The `intro.py` file contains Blueprint functionality and uses the as yet undefined `intro_bp` Blueprint instance. The `intro_bp` instance naming is a convention; adding `_bp` to the end of the instance name indicates a Flask Blueprint. The next thing to do is edit the empty `app/intro/__init__.py` file and create that `intro_bp` instance:

```

from flask import Blueprint    #A

intro_bp = Blueprint('intro_bp', __name__,    #B
    static_folder="static",    #B
    static_url_path="/intro/static",    #B
    template_folder="templates"    #B
)    #B

from app.intro import intro    #C

```

#A Import the Blueprint class from the flask module

#B Create a Blueprint instance, initializing its name, file name, and paths to the static and template files

#C Import the intro module functionality

Because the `__init__.py` file is inside a package, the code will be run every time anything within the package a module is imported, including the just created `intro.py` module. The first thing the code does is import the Blueprint class from the flask module. It then creates the `intro_bp` instance by instantiating the Blueprint class with some parameters.

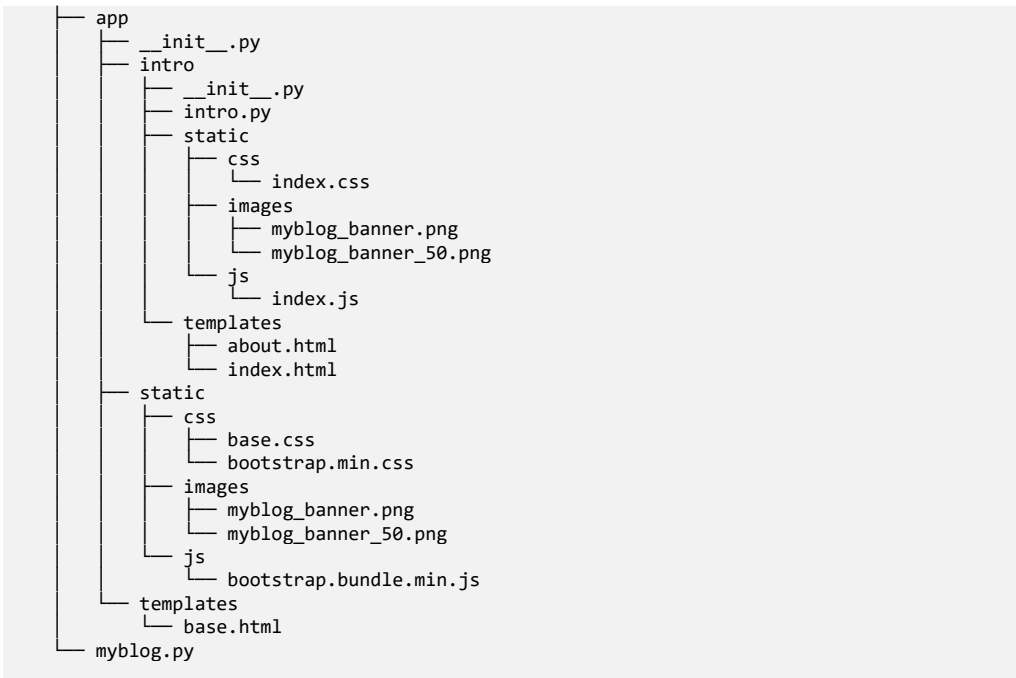
The parameters give the Blueprint a name, pass it the Python filename, and set the `static_folder` and `template_folder` parameters with path strings relative to the Blueprint location. Relative to the Blueprint location means the `intro_bp` Blueprint instance expects to find the templates it will render and the static assets those templates might require on a path relative to where the instance exists.

The `static_url_path` parameter is set to ensure the blueprint relative path doesn't conflict with the root static folder. The value assigned to the parameter is the absolute path from the root directory to the Blueprints static directory.

This means the `index.html` template that provides the home page's content needs to move somewhere the `intro_bp` Blueprint can find. It also means the static assets (CSS files, JavaScript, and images) need to move as well.

UPDATED MYBLOG DIRECTORY

The result of the directory restructuring to put all the files related to the intro Blueprint exists in the `examples/CH_08/examples/03` directory from the code repository:



In the above directory structure, there are two pairs of template and static directories. The first being the original at the project root from the previous example, and the second is

the set under the `intro` package. The second set is what the `intro_bp` Blueprint instance will use when looking for templates and static files because of the values passed to the `static_folder` and `template_folder` parameters when the instance was created.

If those two parameters had not been set, the `intro_bp` instance would have looked for templates and static assets in the app root `static` and `templates` directories under the app directory. Having the `templates` and `static` folders under the Blueprint packages you create makes the Blueprint more portable to other projects and self-contained.

One additional change has to be made to the `index.html` file so it can access the static assets referenced by the HTML code. The `<script>...</script>` tags that references the JavaScript that reacts to the button clicks on the home page needs to be updated in this way:

```
<script src="{{ url_for('.static', filename='js/index.js') }}"></script>
```

The only change is the addition of the single `"."` character in front of `"static"` in the `url_for(...)` statement. The `url_for(...)` statement will resolve this path to the `static` directory relative to the `intro_bp` Blueprint, and the `index.js` file will be pulled in correctly.

APP PACKAGE CHANGES

Before you move on to create the new about page, take a look at the `app/__init__.py` file. All of the code related to the home page has just been moved to the `intro` module, so the file needs to be updated:

```
from flask import Flask

def create_app():
    app = Flask(__name__)
    with app.app_context():
        from . import intro    #A
        app.register_blueprint(intro.intro_bp)    #B
    return app
```

#A Import the `intro` module containing the `intro` Blueprint

#B Register the `intro` blueprint with the app

The code above shows the application factory `create_app()` function simplified to remove the home page's code, which is now part of the `intro` Blueprint.

This is all needed in the MyBlog application to get things rolling when the `flask run` command is called. By default, the `flask run` command looks in the application (which was in the `FLASK_ENV` environment variable) for a Flask instance named `app`. Finding the app instance will start to run the application, serving any URL endpoints that have been configured and registered with the app.

8.4.3 Create the About Page

To demonstrate how the `intro_bp` Blueprint instance can contain features and functionality, you'll add an about page to the MyBlog application. To do so, add a new handler function to the `intro.py` file and register it with the instance by decorating it with a route:

```
@intro_bp.route("/about")    #A
def about():
    return render_template("about.html")    #B
```

#A Decorate the about() function and register it as the handler for the new route "/"about" using the intro_bp Blueprint instance

#B Retrieve and render the about.html template file from the intro_bp relative templates directory

The code above creates a URL endpoint route to "/"about" and registers the about() function as the handler when that route is navigated to by a browser. To render an about HTML page, an about.html template is created in the app/intro/templates directory that's relative to the intro_bp Blueprint:

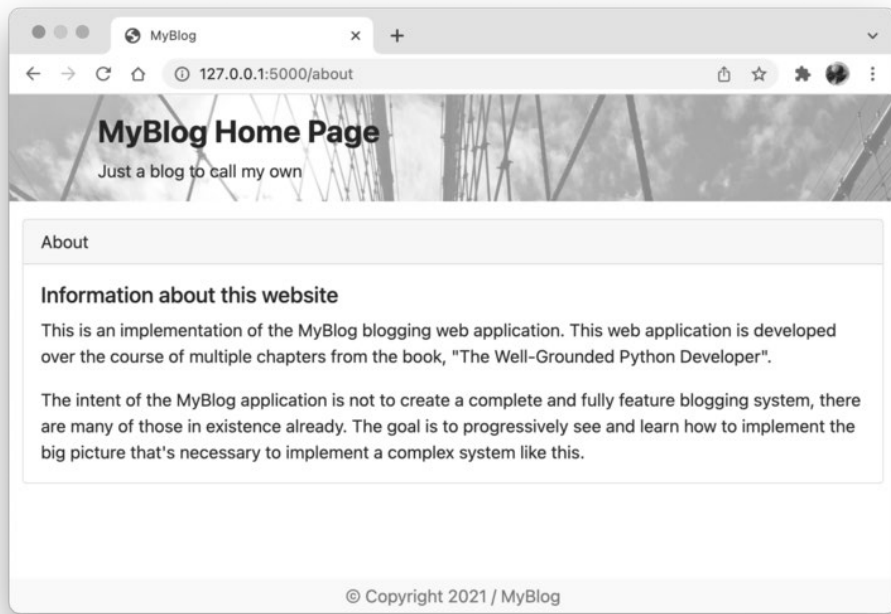
```
{% extends "base.html" %}    #A

{% block content %}    #B
    <div class="container-fluid mt-3">
        <div class="card">
            <div class="card-header">
                About
            </div>
            <div class="card-body">
                <h5 class="card-title">Information about this website</h5>
                <p class="card-text">
                    This is an implementation of the MyBlog blogging
                    web application. This web application is developed over
                    the course of multiple chapters from the book,
                    "The Well-Grounded Python Developer".
                </p>
                <p class="card-text">
                    The intent of the MyBlog application is not to create
                    a complete and fully feature blogging system, there are
                    many of those in existence already. The goal is to
                    progressively see and learn how to implement the big
                    picture that's necessary to implement a complex system
                    like this.
                </p>
            </div>
        </div>
    </div>
{% endblock %}
```

#A Like index.html the about page inherits from base.html and gets the same features it provides.

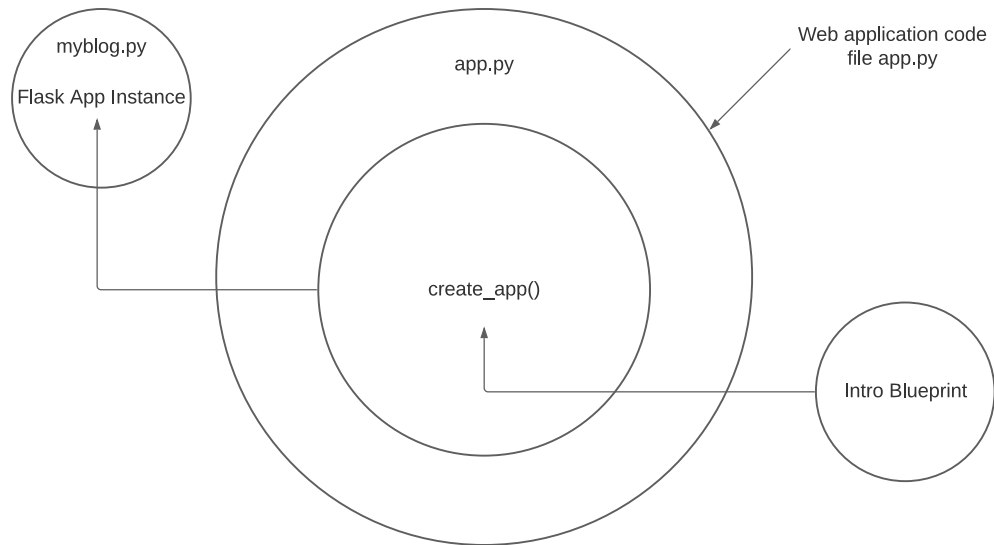
#B The about page replaces the content block with Bootstrap styled text information about the MyBlog application

If you run the MyBlog application and navigate to "127.0.0.1:5000/about" in a browser, the server will respond by rendering the about.html file to the display window. The about page uses Bootstrap styles in the same manner as the home page did:



8.4.4 Refactored App Instance

At this point, the MyBlog application, and Flask app instance, in particular, has been refactored to support a growing web application better:



The diagram shows the `create_app()` function connects to the functionality contained in the external `intro` module, which has access to the app instance inside that function's scope. The app instance is returned by the function inside the scope of the `myblog.py` code. The reference to the app instance is held by `myblog.py` for the life of the application.

8.5 MyBlog Navigation

The MyBlog application now has two pages, the home and about pages. You can navigate to them directly by entering the URL into the browser, but that's not very convenient. Web sites provide clickable links to navigate the system, and you'll add that navigation using Bootstrap.

The MyBlog site navigation is provided by a Bootstrap navbar added to the `base.html` template, so it's rendered on any page inheriting from it, essentially sitewide. The Bootstrap navbar is visually attractive and responsive to device size. It contracts to be a dropdown list for small devices.

You can also add some interactive touches by making the currently active page's menu item visibly highlighted. This means if the about page is being viewed, the about menu item is highlighted. You'll add this by making use of features in Jinja2.

8.5.1 Creating Navigation Information

There are two parts to add the navbar and make it interactive. The first thing to do is edit the `base.html` template file in the root templates folder and add this code at the top of the file:

```
{# configure the navigation items to build in the navbar #}    #A
{% set    #B
    nav_items = [    #B
        {"name": "Home", "link": "intro_bp.home"},    #B
        {"name": "About", "link": "intro_bp.about"}    #B
    ]    #B
}%}    #B
```

#A This is a Jinja2 comment in the template

#B This creates a variable named `nav_items` containing a list of dictionaries with navigation information in each dictionary item

The `{% set ... %}` block allows you to create a variable just as you would in Python code used by other parts of the template. The `nav_items` list variable holds the navigation information necessary to build the Bootstrap navbar links.

Look at the link information in the `nav_items` structure. The home page link is `"intro_bp.home"`, not just `"home"`. This would appear in an HTML link in a template something like this:

```
<a href="{{url_for('intro_bp.home')}}">Home</a>
```

The `url_for` function knows how to find the page relative to the `intro` Blueprint and uses the `intro` Blueprint instance `intro_bp`. It then finds the URL endpoint `home` relative to that. You'll see how this is used when rendering the navbar below.

8.5.2 Displaying the Navigation Information

The second part of creating the navbar is further down in the `base.html` template file. Just above the `{% block content %}{% endblock %}` insert this code:

```

<nav class="navbar navbar-expand-lg navbar-dark bg-primary">    #A
  <a class="navbar-brand ml-2" href="{{url_for('intro_bp.home')}}">    #B
    MyBlog    #B
  </a>    #B
  <button class="navbar-toggler"
    type="button"
    data-bs-toggle="collapse"
    data-bs-target="#navbarSupportedContent"
    aria-controls="navbarSupportedContent"
    aria-expanded="false"
    aria-label="Toggle navigation">
    <span class="navbar-toggler-icon"></span>
  </button>
  <div class="collapse navbar-collapse"
    id="navbarSupportedContent">
    <div class="navbar-nav">
      {% for nav_item in nav_items %}    #C
        {% if request.endpoint == nav_item["link"] %}    #D
          <a class="nav-link ml-2 active"    #E
            aria-current="page"    #E
            href="{{url_for(nav_item['link'])}}">    #E
            {{nav_item["name"]}}    #E
          </a>    #E
        {% else %}
          <a class="nav-link ml-2"    #F
            href="{{url_for(nav_item['link'])}}">    #F
            {{nav_item["name"]}}    #F
          </a>    #F
        {% endif %}
      {% endfor %}
    </div>
  </div>
</nav>

```

#A This begins the Bootstrap navbar styling section and sets the color and style of the navbar

#B Creates the “MyBlog” brand as a clickable link to the home page

#C Iterates over the `nav_items` variable

#D Compares the current page to the current `nav_item` link

#E Output a highlighted link if the comparison was true

#F Output a normal link if the comparison was false

There's quite a bit going on here. Much of the code is about getting Bootstrap style classes in the right places with the correct context and information. This is a lot of styling information to work with and learn when using Bootstrap. However, compared to providing the same styling functionality with hand-written CSS code, it's minuscule.

An interesting part of the template code is the for loop and the if statement within it. The for loop iterates over the previously created `nav_items` list pulling out one `nav_item` at a time. The if statement uses the built-in object and attribute `request.endpoint` to determine if the page being currently built is the same as the `nav_item["link"]` value.

If the current page is equal to the `nav_item` link, the navbar menu item is an html link containing the class "active" and the html attribute `aria-current=" page"`. The Bootstrap "active" class adds visual highlighting to the menu item. The HTML attribute `aria-current=" page"` helps users navigating to the page using a screen reader for accessibility.

If the current page is not equal to the `nav_item` link, the navbar menu item is rendered in its default state with no highlight applied.

8.5.3 MyBlog's Current Look

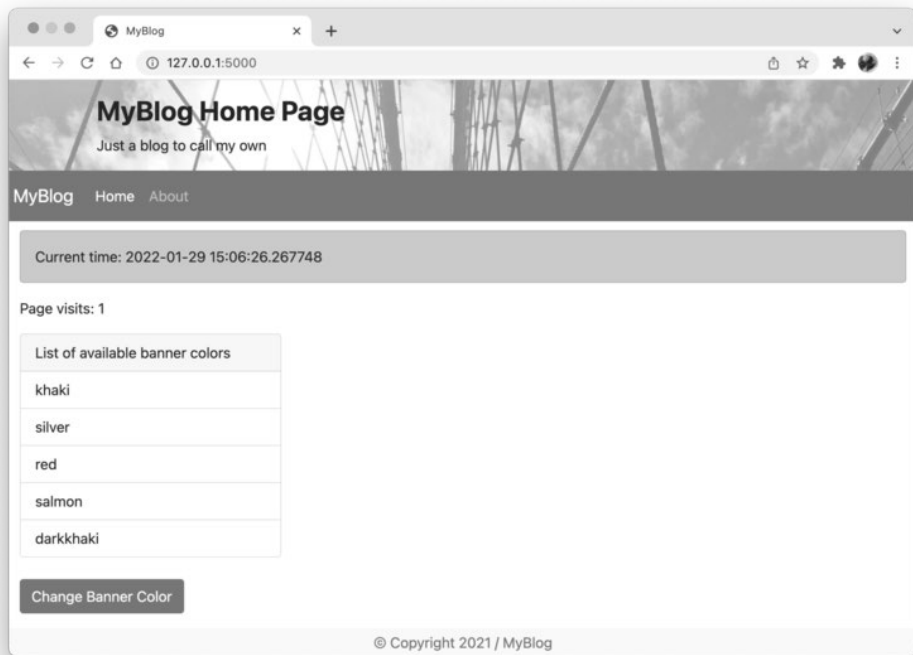
If you change your working directory to `examples/CH_08/examples/04`, there's a fully functional example program that implements what we've gone through. Use the commands you've seen before to run the application for Mac and Linux users:

```
export FLASK_ENV=development
export FLASK_APP=myblog.py
flask run
```

And for Windows users:

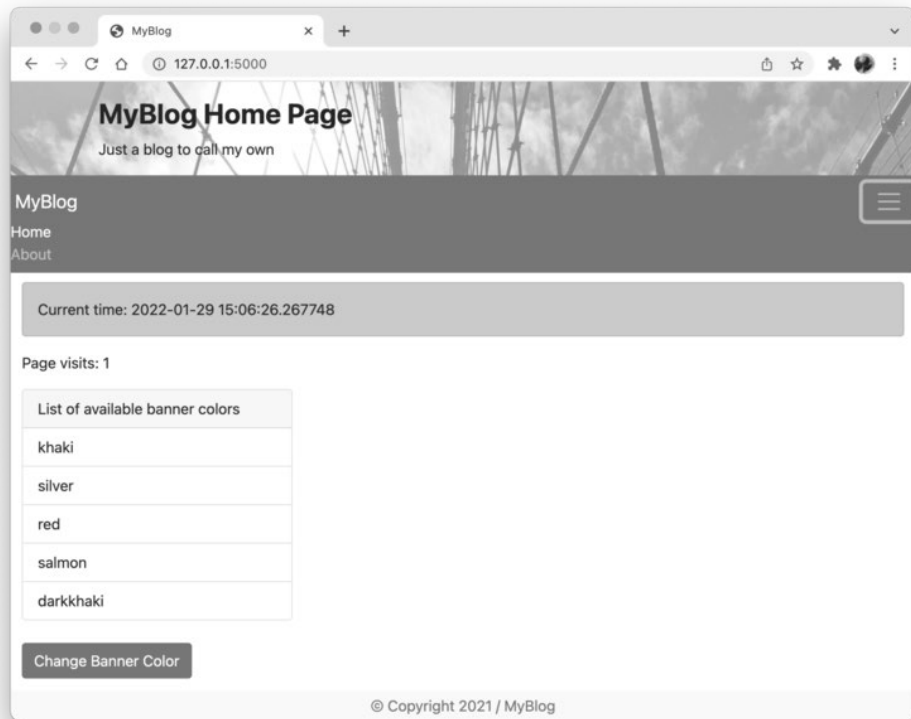
```
set FLASK_ENV=development
set FLASK_APP=myblog.py
flask run
```

When the application is running, use your browser to navigate to `127.0.0.1:5000`, and you'll see the application:

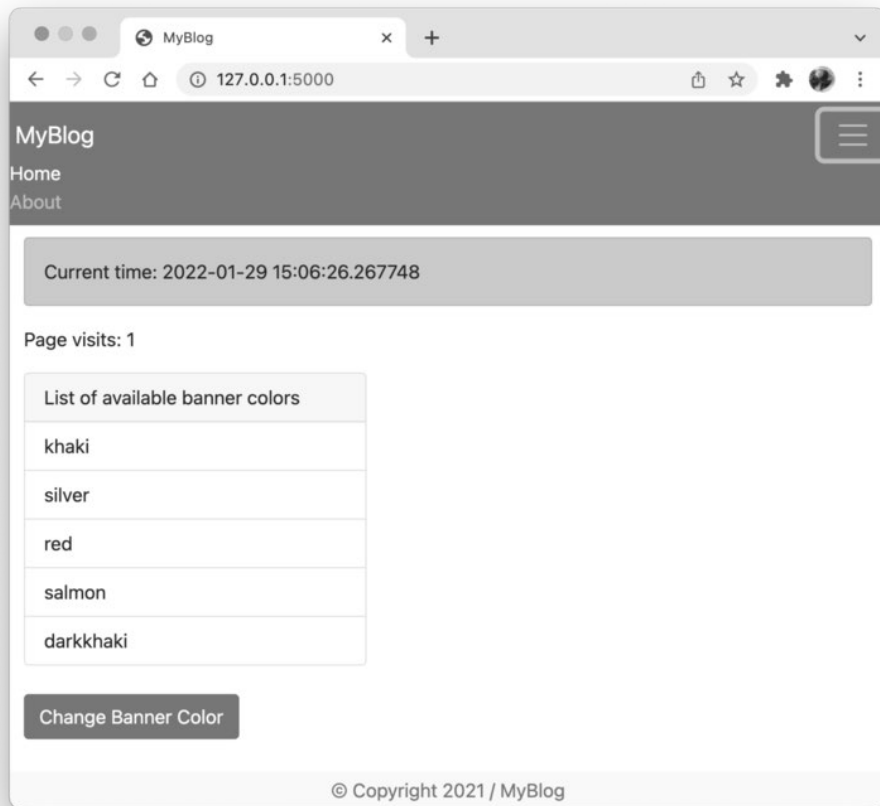


The image shows the current MyBlog application with the newly added Bootstrap navbar directly below the banner. This is the expanded view with the home page menu item highlighted by the `base.html` template code shown previously.

If you start to minimize the browser window horizontally, you'll see the navbar navigation will be reduced to a dropdown button on the left side of the navbar. If you click that button, the navigation menu items will appear, allowing access to the Home and About links:



If you continue to minimize the browser, the media query in the `base.css` file will become active, and the banner will be removed, providing more screen real estate for smaller devices:



8.6 Application Configuration

The MyBlog application has come a long way since its first incarnation, and there's still some work to do to improve it further. The development you're embarking on targets making the configuration, sustainability, and maintainability of the application better and easier.

8.6.1 Configuration Files

The MyBlog application will eventually rely on functionality that needs configuration data. Sending an email to an SMTP (Simple Mail Transfer Protocol) email server, accessing a database to store users and blog posts, and security data for the Flask application itself.

In addition to the configuration data needed by the MyBlog feature set, you'll want to set up distinct data for the development and production environments.

The configuration data creates an environment in which the MyBlog application will run. A development environment often provides additional debugging services and access to the inner workings of the application. If you develop an application in a team, different configuration data is needed to create a staging environment. A staging environment is where the team can test the various parts of the application before being pushed into production. A production environment removes the debug information and restricts access to only the feature set intended to be publicly available.

8.6.2 Private Information

Any application whose feature set includes accessing systems or services requiring usernames, passwords, or API keys needs to keep that information private. For example, the MyBlog application will automatically send emails to users and contact the system administrator using email.

The MyBlog application won't handle sending an email directly but will instead use an external SMTP email server. The app will need to authenticate the email server using a username and password and possibly an API key. While the MyBlog application will need access to this information, you don't want this information to become available in the public domain.

It would be easy to embed this information directly in the code, but that pretty much assures it would become public. Unintentionally making private data public could happen as easily as embedding private information in the code and then checking that code into a shared repository like GitHub. Services like GitHub provide a valuable resource to developers, but they don't automatically protect you from publishing things you didn't intend to.

As a developer, you'll want to protect yourself and your employer so the services your application uses don't get abused. Besides being good practice, if the services cost money, failing to protect them could cost more than you intended.

One way to separate private information from an application but still have access to it is to store the information in separate files the application reads at runtime. These files aren't checked in to a repository, so they are at less risk of becoming public. Additionally, multiple versions can be maintained for the different environments the application would run in, like development, staging, and production.

8.7 Flask Debug Toolbar

To introduce configuration files more interesting, you'll be using them to add the Flask DebugToolbar (<https://github.com/jazzband/django-debug-toolbar>) to the MyBlog application when it's running in development mode. The toolbar is useful when developing a Flask application. It shows internal information right in the browser application window that would otherwise only be available by debugging the application itself.

The configuration files you'll add will control information the DebugToolbar requires to run and dynamically installs the toolbar only in development environments. The DebugToolbar module needs to exist in the currently active Python virtual environment:

```
pip install flask-debugtoolbar==0.11.0
```

8.7.1 FlaskDynaConf

The configuration information is stored in TOML files, which are human-readable and supports data types for the information contained within them. To gain access to the configuration TOML files, you'll need to install the dynaconf module into the current Python virtual environment:

```
pip install dynaconf==3.1.2
```

The module contains a class specific to Flask that will automatically add configuration information to the Flask configuration system.

The Flask DebugToolbar requires the flask app to have a `SECRET_KEY` value. The `SECRET_KEY` value is used for security aspects of a Flask application used in later chapters. In the configuration file, it looks like this:

```
secret_key=" <random string of characters>"
```

Following my own advice, the actual secret key used during the example code development isn't revealed. The secret key's value should be a cryptographically strong string of characters that isn't publicly known. You can generate a `SECRET_KEY` value with the following code:

```
python
>>> import secrets
>>> print(secrets.token_hex(24))
b3a40bcc3bcc5894c390681396ec04687ad869c6546cdff9    #A
```

#A Sample output

The `secrets` module provides cryptographically strong random values better suited to manage private information. The printed results are copied between the quotes as the `secret_key` value in a file named `.secrets.toml`:

```
[default]
secret_key=" <random string of characters>"
```

The file structure contains sections, indicated by the `[default]` line, and a set of key/value pairs of data. Because the `secret_key` is defined within the `[default]` section, that value is available in any other section unless it is overridden explicitly by another `secret_key` key/value pair.

To complete the configuration to add the Flask DebugToolbar requires the creation of another TOML file, `settings.toml`:

```
[default]      #A

[development]   #B
flask_debug = true      #C
extensions = ["flask_debugtoolbar:DebugToolbarExtension"]      #D
debug_tb_enabled = true      #D
debug_tb_intercept_redirects = false      #D

[production]    #E
flask_debug = false    #F
```

#A No default information is currently defined

#B Begin the definition of configuration information for development environments

#C Enable the Flask debug mode

#D Import the Flask DebugToolbar, enable it, and disable redirect intercepts

#E Begin the definition of the configuration for production environments

#F Disable the Flask debug mode

The configuration is separated into three sections, `[default]` (currently empty), `[development]`, and `[production]`, and could be divided into as many as needed. Notice values have data types, and the `extension` key's value is a list of strings. The `flask_debug` key/value pair has a value of `true`, which is a Boolean.

Each section (aside from `[default]`) is keyed to the Flask environment variable initialized before running the application:

```
export FLASK_ENV=development
```

Only the information under the `[default]` and the indicated environment will be read from the `settings.toml` file at run time. Both the `.secrets.toml` and `settings.toml` files should be excluded from the project repository.

CONFIGURING MyBLOG

To use the configuration information, you need to make changes to the MyBlog application. Because the configuration is central to the application, make these changes to the `app/__init__.py` file:

```
import os
import sys
from flask import Flask
from dynaconf import FlaskDynaconf      #A

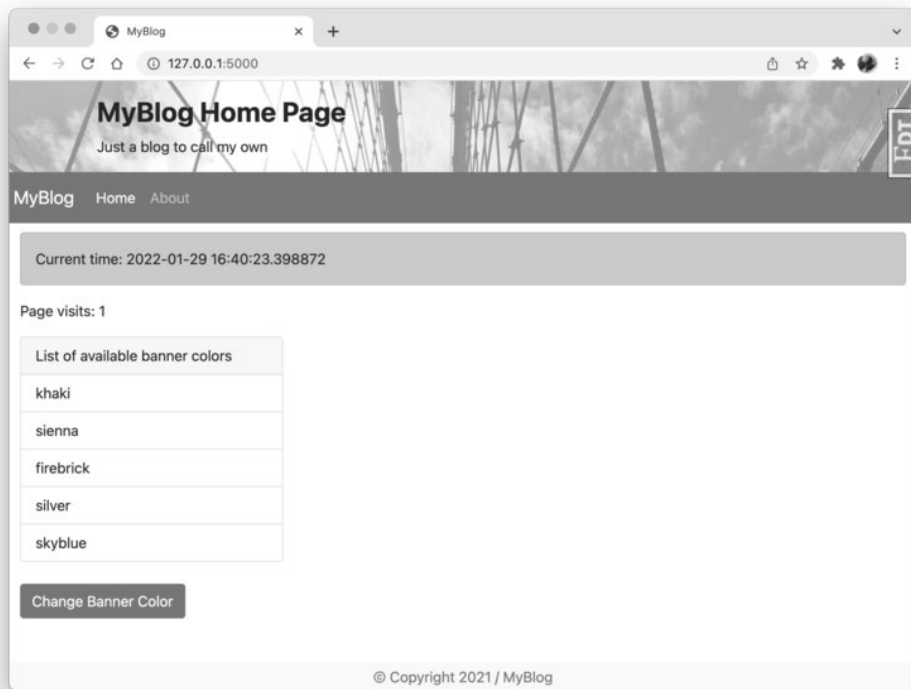
def create_app():
    app = Flask(__name__)
    dynaconf = FlaskDynaconf(extensions_list=True)      #B

    with app.app_context():
        os.environ["ROOT_PATH_FOR_DYNACONF"] = app.root_path      #C
        dynaconf.init_app(app)      #D
        app.config["SECRET_KEY"] = bytearray(app.config["SECRET_KEY"], "UTF-8")      #E
        from . import intro
        app.register_blueprint(intro.intro_bp)
    return app
```

#A Import the Flask specific dynaconf class

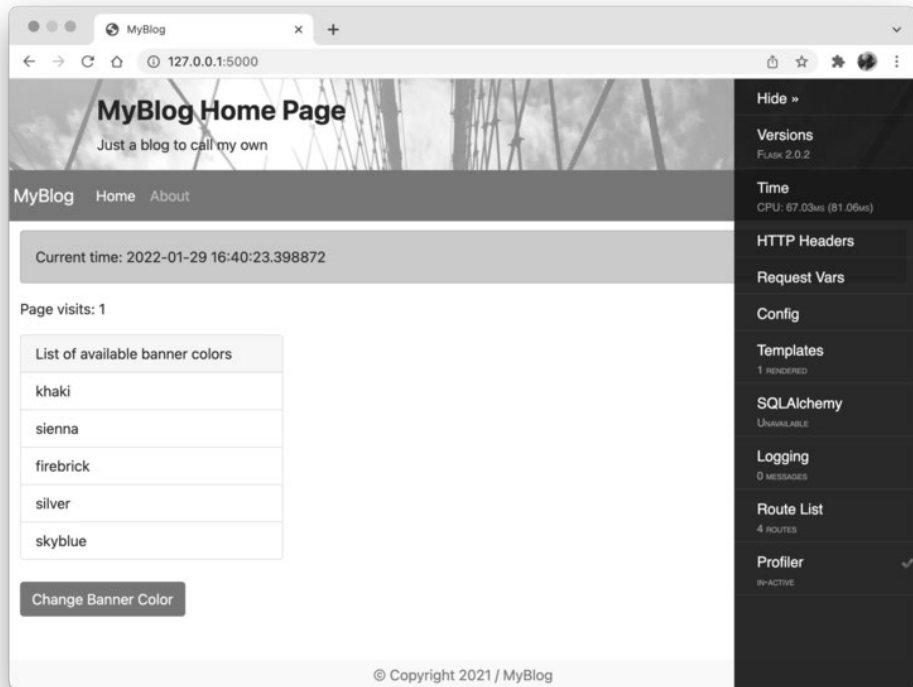
#B Create an instance of the `FlaskDynaconf` class, activating the module loader feature
 #C Inform dynaconf where to look for configuration *.toml files
 #D Configure the Flask app based on the dynaconf read configuration files
 #E Translate the `SECRET_KEY` string into a bytearray as recommended by the Flask documentation

The `FlaskDynaconf` class searches for configuration files based on file naming patterns and directory structures and finds both the `.secrets.toml` and `settings.toml` files. It parses them and configures the Flask `app.config` object accordingly.

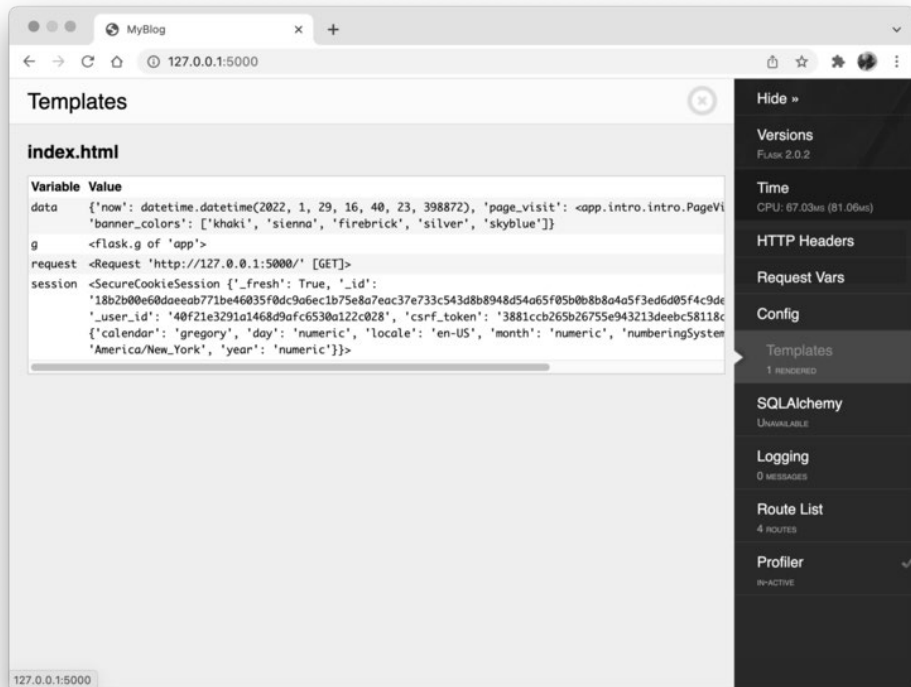


Running the MyBlog application in `examples/CH_08/examples/05` in the development environment renders the home page in the browser like this:

The initial view doesn't show much except for the small tab in the upper right corner of the window labeled "FDT" for Flask Debug Toolbar. Clicking on this tab opens the toolbar and presents this view:



This screen shows the Debug Toolbar menu providing access to the information available. Clicking on the Templates menu item loads the working area of the display with information about the home page template:



Not all the menu options provide useful information. The Logging selection doesn't show anything currently but will after adding logging configuration in the next section.

8.8 Logging Information

Flask uses the Python logging module to log information to standard output (STDOUT), as you've seen when running the MyBlog application from the terminal command line.

Almost any coding work a developer does leads to inserting print statements into the code to get information out of a running application. Logging is a simple way to get a snapshot of what's happening at that point of execution in the application. The same facility is available when developing a web application through the logging system. Logging information to STDOUT is beneficial in a long-running server application.

There are other advantages the logging system has over print statements; it supports severity levels for the logged information. It can also provide a standardized format that's chronologically arranged and useful when looking for events or the order of operations for a sequence of events.

The Python logging system supports multiple paths, or handlers, for logged messages with different endpoints. These endpoints can be as simple as logging to STDOUT or, more

sophisticated, like sending an email or text message to a service in response to a logged message.

For the MyBlog application, the logging configuration is relatively simple: sending logs to STDOUT with a particular message format.

8.8.1 Configuration

The Python logging module supports configuration from a dictionary, and that's what MyBlog will use. Configuring the logging system is another responsibility you'll add to the app package's `create_app()` function.

As mentioned, the logging system uses logging levels, ranging from `NOTSET` to `CRITICAL`. Besides providing some context about the logged message, the logging system's level acts as a simple filter. If the logging system's level is set to `INFO` for a particular logger, only messages with a level equal to `INFO` or greater will be logged.

The `DEBUG` and `INFO` levels are of interest for the development and production environments the MyBlog application runs in. In development, the level can be set to `DEBUG`, and all logged messages with a level of `DEBUG` or greater will be logged. In a production environment, the level can be set to `INFO`, and all log messages with a level of `INFO` or greater will be logged, and `DEBUG` level messages will be ignored.

By differentiating the two environment's logging levels, you can add `DEBUG` level messages into an application during development work and leave them in place. In a production environment, those messages won't be logged.

The Python dictionary to configure logging is created by reading a logging configuration YAML file with a support function in `app/__init__.py`:

```
def _configure_logging(app, dynaconf):
    logging_config_path = Path(app.root_path).parent / "logging_config.yaml"
    with open(logging_config_path, "r") as fh:
        logging_config = yaml.safe_load(fh.read())
        env_logging_level = dynaconf.settings.get("logging_level", "INFO").upper()
        logging_level = logging.INFO if env_logging_level == "INFO" else logging.DEBUG
        logging_config["handlers"]["console"]["level"] = logging_level
        logging_config["loggers"][""]["level"] = logging_level
        logging.config.dictConfig(logging_config)
```

The `_configure_logging()` function has a leading underbar character, a convention indicating it intends to be private to the module. The leading underbar is only a convention and does not add any privacy protections to the function.

The function creates an `env_logging_level` variable based on the environment string passed to the function. The variable is used to create the configuration dictionary to control the logging level console STDOUT handler.

The logging configuration information is in a file named `logging_config.yaml` at the project directory level:

```

version: 1
disable_existing_loggers: true      #A
formatters:
  default:
    format: '[%(asctime)s.%(msecs)03d] %(levelname)s in %(module)s: %(message)s'
    datefmt: '%Y-%m-%d %H:%M:%S'
handlers:
  console:      #B
    level: DEBUG      #B
    class: logging.StreamHandler      #B
    formatter: default      #B
    stream: ext://sys.stdout      #B
loggers:
  '':
    level: DEBUG
    handlers: [console]
    propagate: false

```

#A Disable any existing loggers created by Flask

#B Configure only one handler to send errors to STDOUT

The structure creates the formatters used by the handlers, which the loggers use. The formatters change the logging message default format used by the MyBlog application.

Add a `logging_level` configuration key/value to the `settings.toml` file to both the development and production sections:

```

[development]
...
logging_level = "DEBUG"

[production]
...
logging_level = "INFO"

```

Two changes need to be made to the `create_app()` function in `app/__init__.py` to use the dictionary returned by the `_configure_logging()` function. Add these two lines at the bottom of the import section:

```

import logging
import logging.config

```

and add this line to the `create_app()` function immediately below the code that converts the `app.config[SECRET_KEY]` to a bytearray:

```

_configure_logging(app, dynaconf)      #A

```

#A Calling the function configures logging for the MyBlog application

You can now add `DEBUG` level log messages where needed to help develop the MyBlog application. The home and about pages can be modified to demonstrate this:

```
@intro_bp.route("/")
def home():
    logger.debug("rendering home page")    #A
    return render_template("index.html", data={
        "now": datetime.now(),
        "page_visit": PageVisit(),
        "banner_colors": BannerColors().get_colors()
    })

@intro_bp.route("/about")
def about():
    logger.debug("rendering about page")    #B
    return render_template("about.html")
```

#A Send a **DEBUG** level message to the logging system that the home page has been rendered

#B Send a **DEBUG** level message to the logging system that the home page has been rendered

When starting and running the MyBlog application, a **DEBUG** log message will be present in the logging output whenever the application home or pages are accessed. If you run the code in `examples/CH_09/examples/07` and navigate to the about and home pages, the log output will look similar to this:

```
[2021-01-08 15:29:39,030] WARNING in _internal: * Debugger is active!
[2021-01-08 15:29:39,055] INFO in _internal: * Debugger PIN: 107-111-649
[2021-01-08 15:29:39,104] INFO in myblog: MyBlog is running
[2021-02-03 13:56:57.535] DEBUG in intro: rendering about page
[2021-01-08 15:29:55,707] DEBUG in intro: rendering home page
```

When the MyBlog application runs in a production environment, any **DEBUG** messages generated by the app won't be present in the logging output. Suppressing **DEBUG** messages is useful to keep from cluttering the logging output with development information.

8.9 Adding a favicon

A favicon is a graphic image used as a shortcut to represent a website. Supporting a favicon gives the MyBlog application some additional professionalism, so we'll add one.

The code in `examples/CH_09/examples/07` includes two versions of the MyBlog brand graphic; one in the `.ico` format (icon) and one in the `.svg` format (Scalable Vector Graphics). The first will give a browser window tab holding the MyBlog application the small brand icon. The second appears alongside the "MyBlog" text in the navigation bar.

The `favicon.ico` file has to be served from the MyBlog application when requested by the browser. The browser expects to find it in the root folder, which won't be part of any Flask blueprint in MyBlog. The `favicon.ico` is made available by adding a route to it directly in the `app/__init__.py` file, inside the `with app.app_context():` context manager:

```
@app.route('/favicon.ico')
def favicon():
    return send_from_directory(
        os.path.join(app.root_path, 'static'),
        'favicon.ico',
        mimetype="image/vnd.microsoft.icon"
    )
```

The code registers the URL route where the browser is looking for the `favicon()` function. Flask uses the `send_from_directory()` function to get the file's path, the file name, and mime type and return it to the browser.

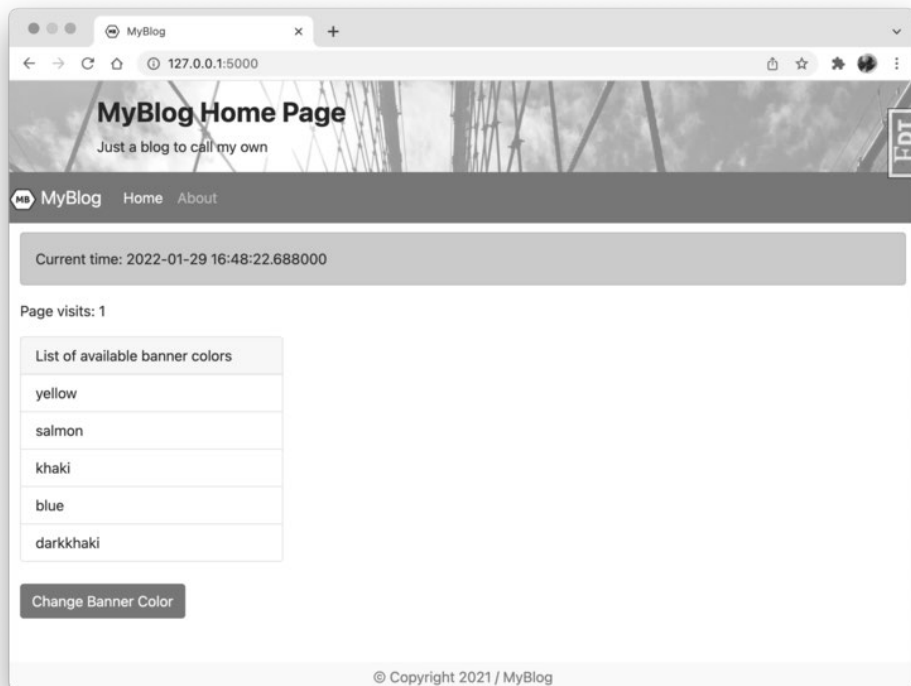
You'll also add the MyBlog brand svg image in the navigation bar built in the `base.html` template. The image is added immediately before the "MyBlog" text in the navbar brand section of the template:

```

```

This HTML code adds an image link that finds the `myblog_brand_icon.svg` file in the application root static folder and scales it appropriately for display in the navigation bar.

Running the MyBlog application in `examples/CH_09/examples/07` presents a browser display like this:



The screenshot shows the `favicon.ico` file displayed in the browser tab containing the MyBlog application. The brand svg file is displayed immediately to the left of the "MyBlog"

text in the navigation bar. Both the brand graphic and name are clickable links returning the user to the application's home page.

8.10 Summary

You've covered a lot of material that helps bring the initial draft of the MyBlog application to a great place:

- You've integrated Bootstrap into the MyBlog application
- You've refactored the MyBlog application for further growth
- Created namespaces within the app using Flask Blueprints
- You've made configuration information external to the app and able to be kept private
- Added the Flask DebugToolbar when working on the app in development environments
- You've added environment aware configured logging

Having Bootstrap integrated into the application goes a long way towards making MyBlog look polished and professional while at the same time reducing the development workload of creating that look and feel.

The refactoring work done to MyBlog gives you a good foundation from which to grow the application further. Adding new features will be simpler because backtracking to address initial assumptions will be avoided.

By adding external configuration files to MyBlog, you've enabled creating multiple runtime environments and moved sensitive information to a more protected space. The use of configuration files also let you add the Flask DebugToolbar and logging levels to the MyBlog application.

The next chapter will build on this excellent foundation by adding user authorization and creation, allowing the MyBlog application to secure pages to only logged in users.

9

Authenticating the user

This chapter covers

- Flask Sessions
- Remembering the User
- Letting Users Log In
- Registering New Users
- Allowing Users to Reset Forgotten Passwords
- Allowing Existing Users to Change Their Passwords
- Securing Routes in the Application to Logged in Users

The MyBlog web application supports many users so they can post interesting content that will be read by the community. That community can read and comment on the content posted by other users. However, it's unlikely users want the content they created edited or deleted by a user other than themselves, or the MyBlog administrators and editors.

To control who can do what on the MyBlog site we'll need to identify who users are and what they have permission to do. Identifying users on a web application is called authenticating a user so the application has a way to ensure a user who is who they say they are. Deciding what an authenticated can do while using a web application is called authorization, and lets the application know what an authenticated user has permission to do.

Providing authentication and authorization to the MyBlog application is the intent of this chapter. Do so with a web application presents some unique challenges.

9.1 The HTTP Protocol is Stateless

The MyBlog web application follows the request/response model supported by HTTP. The user creates an HTTP GET request from the browser, and the server responds by sending the

requested HTML, CSS, JavaScript, and image files back. Nothing in that transaction implies the server has any prior knowledge about the requests it receives. The HTTP protocol is stateless, meaning each request is complete and independent from any previous request, and the server maintains no memory of past, present, or future request/response transactions.

In this model, the request has to contain all the information necessary so the server can build the appropriate response. Even if the same request is made to the server multiple times, the server will rebuild the same response each time (barring any response caching).

If servers using the HTTP protocol have no memory, how do retail shopping sites know it's you when you purchase things? How does the web application running on that site remember the shopping cart you've created across multiple request/response transactions? And most importantly, how does a shopping site ensure the credit card belongs to the user making the purchase request? Answering these questions implies adding state information to the transactions between the client's browser and the web application server.

9.1.1 Sessions

A session allows the server to relate information it has about a user to incoming requests. The session commonly establishes the relationship with a cryptographically strong unique id value generated by the server and saved as a cookie in the client browser. This is known as a session cookie, though the client's actual cookie name can vary depending on the server framework in use.

This session cookie is shared with the server when a request from the client is sent. The session usually contains a unique identifier that the server has encrypted. The unique identifier encryption uses a key-value stored on the server to prevent the session from being modified on the transaction's client-side.

When a request with a session arrives at the server, the server can decrypt the unique identifier and relate it to a specific user and any user information the server is maintaining. For example, a user's name, shopping cart, and more. By layering a session on the HTTP protocol, the server can keep state information from one request to the next.

FLASK SESSIONS

Flask supports the use of sessions and makes them available to the MyBlog application. A session cookie doesn't exist between a client and the server until the server explicitly creates it. You can create a session by adding information to it in any URL route handler code you develop. When that URL is accessed, the session is created and added to the client as a cookie in the response.

Flask uses the `SECRET_KEY` to cryptographically sign the session cookie when created. By doing this, the cookie can be viewed on the client-side but can't be modified unless the `SECRET_KEY` is available. As mentioned in the previous chapter, when the `SECRET_KEY` was added to enable the Flask DebugToolbar, it's essential the `SECRET_KEY` be cryptographically strong and kept private.

By default, session cookies exist until the client browser is closed. This can be changed by modifying the session's permanent attribute, which is a Python `datetime.timedelta()` value. You could make a session exist for a year by doing this in the server code:

```
session.permanent = True      #A
app.permanent_session_lifetime = datetime.timedelta(days=365)    #B
```

#A Marks the session as permanent

#B Uses the Flask app instance to set the lifetime of the permanent session

Once a session exists, you can use it to maintain information across the request/response transaction. Using a session cookie for information storage is convenient but has limitations. A cookie has a memory size limit imposed on it by the browser, which can vary from browser to browser. The memory constraint on cookie size is one issue, but another is data size going across the internet for every request/response message. The information stored in a cookie is sent back and forth with every transaction between the client and the browser. Even in the age of widely available high-speed internet access, that's still a concern, especially for mobile devices.

The solution to both these concerns is to use the session cookie to store the unique user identifier value and use that on the server-side to retrieve all the other information necessary to build the correct response to the request.

The MyBlog application will provide different access depending on whether or not the user is known and logged into the system and the user's role. Knowing who the user is and keeping track of their role means the system must remember that user's information. We'll get started adding that ability to the MyBlog application.

9.2 Remembering the User

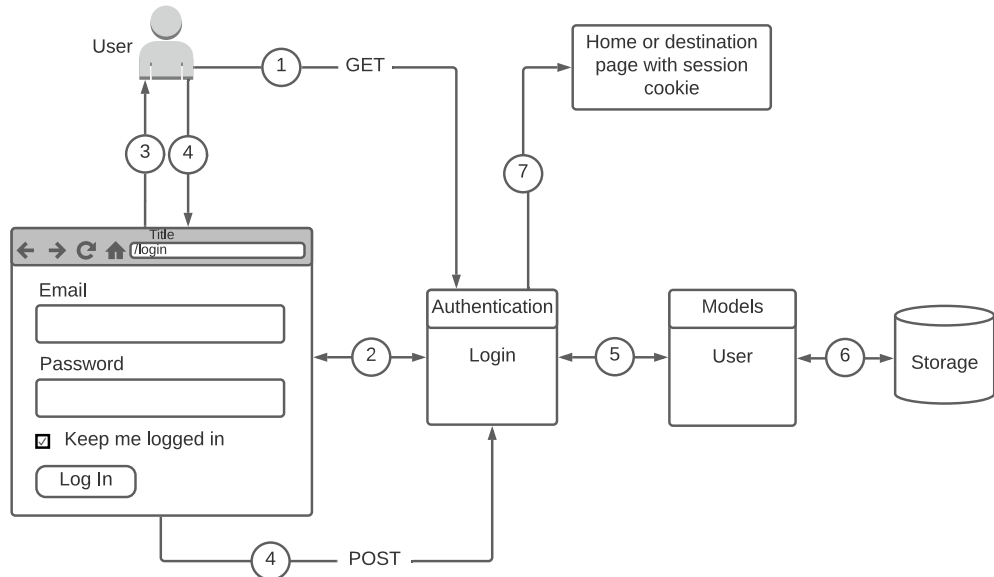
Remembering who the user is gives the MyBlog application ways to control what features are available to users. Viewing blog entries and comments are open to anyone, but the ability to add a new blog entry or comment on an existing one is reserved for known users. Users with the editor role can edit any user's posted blog content or comment

The user information described above can be stored on the server and retrieved using the session cookie's unique user identifier. A unique user identifier value has to be created and stored and then used to authenticate that a user is who they say they are. For a website identifying a user is performed by logging a user in through an authentication system.

9.2.1 Authenticating Users

The MyBlog application uses the Flask-Login extension as an installable module from the Python Package Index (<https://pypi.org/>). The Flask-Login extension gives the MyBlog application session management abilities and tools to log users in and out and handle the somewhat difficult "remember me" functionality. It also handles adding protection to URL endpoints so only authenticated users can access them.

The login process follows the common email/password pattern to authenticate users. The user's email address is valid choice as a unique identifier because it is already unique and likely well remembered. From a high-level view, the login system you're going to create looks like this:



The user login process follows this sequence of steps:

1. The user makes a GET request from their browser to the authentication login URL endpoint
2. The authentication login handler responds to the GET request by returning the rendered login HTML page
3. The user fills out the login page form fields and submits the form
4. The form is submitted to the authentication login system using a POST request
5. The login system tries to find a user with a matching email and password using the models supported by the application
6. The User model tries to find a user with a matching email and password in the application storage system
7. If a matching user is found, the user is directed to the home page or the original destination page the user wanted to view

Because powerful computer CPU and GPU hardware are readily available, the ability to crack user passwords has become far more accessible for hackers to implement. The MyBlog application uses the Flask-Bcrypt extension to hash the passwords stored at the server. The bcrypt functionality creates a computationally expensive hash making it resistant to brute force attacks even with increasing computer power. Plain-text passwords should never be stored in a database and should always be cryptographically hashed first.

The diagram above shows a storage mechanism accessed by step six. The Flask-Login extension requires this storage to persist users to retrieve and identify them later. To do this,

you'll be using SQLAlchemy to manage the user data in a SQLite database. This chapter focuses primarily on authenticating users and will defer more detailed information about SQLAlchemy and utilizing a database till the next chapter.

To install all the modules necessary to run the example applications for this chapter, run the following command from within a Python virtual environment:

```
pip install -r requirements.txt
```

This makes the modules available to you in the code you'll create to add authentication.

LOGINMANAGER

The first thing to do is add the modules necessary for authentication, password encryption, and user persistence modules to the `app/__init__.py` module. Adding them to the import section at the top of the module makes the functionality available to the `create_app()` application factory function:

```
import os
import yaml
from pathlib import Path
from flask import Flask, send_from_directory
from dynaconf import FlaskDynaconf
from Flask-sqlalchemy import SQLAlchemy      #A
from Flask-migrate import Migrate           #B
from Flask-Login import LoginManager        #C
from Flask-bcrypt import Bcrypt             #D
import logging
import logging.config
```

#A Import the SQLAlchemy functionality to manage the data persistence

#B Import the data migration functionality (detailed in the next chapter)

#C Import the LoginManager to handle user authentication

#D Import the Bcrypt module to cryptographically encrypt user passwords

Then right above the `create_app()` function add new global instance variables for the new functionality:

```
login_manager = LoginManager()      #A
login_manager.login_view = "auth_bp.login"  #B
Flask-bcrypt = Bcrypt()             #C
db = SQLAlchemy()                   #D
migrate = Migrate()                 #E
```

#A Create an uninitialized instance of the LoginManager class

#B Point the LoginManager instance to the Blueprint view to be created later

#C Create an uninitialized instance of the Bcrypt() class

#D Create an uninitialized instance of the SQLAlchemy class

#E Create an uninitialized instance of the Migrate class

Inside the scope of the `create_app()` function in the initialize plugins section initialize the new instance variables you've just created with the `app` instance variable:

```
os.environ["ROOT_PATH_FOR_DYNACONF"] = app.root_path
dynaconf.init_app(app)
login_manager.init_app(app)
Flask-bcrypt.init_app(app)
db.init_app(app)
migrate.init_app(app, db))    #A
```

#A The migrate instance initialization requires both the app and db instances as parameters

In the import routes section, import an auth module you'll create shortly:

```
from . import intro
from . import auth
```

In the register blueprints section register the auth blueprint:

```
app.register_blueprint(intro.intro_bp)
app.register_blueprint(auth.auth_bp)
```

Add this new section just above the `return app` line at the bottom of the `create_app()` function:

```
db.create_all()    #A

from .models import Role    #B
Role.initialize_role_table()    #B

@app.context_processor    #C
def inject_permissions():    #C
    return dict(Permissions=Role.Permissions)    #C
```

#A Creates the SQLite database if it doesn't already exist

#B Initializes the Role lookup table with data

#C Makes the Role Permissions available within the context of a Jinja template

The new code above uses functionality that's defined in sections coming up in this chapter. This work initializes the authentication, encryption, and database systems whenever the app package is accessed or imported in the same manner that dynaconf was. The next step is to create the auth Blueprint that handles the user authentication functionality.

AUTH BLUEPRINT

Like the intro Blueprint, the auth Blueprint is a Python package containing distinct functionality. Create a directory named `auth` under the app package and create an `__init__.py` file inside of the auth directory. The `__init__.py` file creates and initializes the `auth_bp` Blueprint instance:

```

from flask import Blueprint

auth_bp = Blueprint(
    "auth_bp", __name__,
    static_folder="static",
    static_url_path="/auth/static",
    template_folder="templates"
)

from . import auth

```

This code creates the `auth_bp` Blueprint instance whenever the `auth` package is accessed or imported. The actual authentication functionality is contained in the `auth.py` file also created under the `app/auth` directory:

```

from logging import getLogger
from flask import render_template, redirect, url_for, request
from . import auth_bp      #A
from ..models import db_session_manager, User      #B
from .. import login_manager      #C
from .forms import LoginForm      #D
from Flask-Login import login_user, logout_user, current_user
from werkzeug.urls import url_parse

logger = getLogger(__name__)

@login_manager.user_loader      #E
def load_user(user_id):      #E
    with db_session_manager() as db_session:      #E
        return db_session.query(User).get(user_id)      #E

@auth_bp.route("/login", methods=["GET", "POST"])      #F
def login():
    form = LoginForm()
    if form.validate_on_submit():
        with db_session_manager() as db_session:      #G
            user = db_session.query(User).filter(User.email ==
            form.email.data).one_or_none()      #H
            if user is None or not user.verify_password(form.password.data):      #I
                flash("Invalid email or password", "warning")      #I
                return redirect(url_for("auth_bp.login"))      #I
            login_user(user, remember=form.remember_me.data)      #J
            next = request.args.get("next")      #J
            if not next or url_parse(next).netloc != "":      #J
                next = url_for("intro_bp.home")      #J
            return redirect(next)      #J
    return render_template("login.html", form=form)

```

#A Import the `auth_bp` Blueprint instance from the package

#B Import the models, which will be created next

#C Import the `login_manager` instance from the package

#D Import the `LoginForm`, which will be created in the next section

#E Function called every time the `login_manager` needs to determine if the user exists

#F The login function registered with the `auth_bp` Blueprint for the `/login` route

#G Use the `db_session_manager` context manager to enclose the use of the database session

#H Get a user from the database based on the form email value,

#I If no user was found, or the password doesn't verify, warn the user and redirect to the login

#J Set the logged in user and create a session cookie

The `auth.py` module creates a route called `"/login"` associated with the `auth_bp` Blueprint instance and associates it with `login()` handler function. The `login()` handler function has two purposes. When it is called because of an HTTP GET request, it returns the rendered `login.html` template created in the next section. If the function is called as the result of an HTTP POST request, it will process the contents of the form parameters in the `login.html` template.

The `if form.validate_on_submit()` code determines if the HTTP request method is a GET or a POST and branches accordingly. If the method is a POST, it will validate the form parameters against a set of rules. If the form parameters are valid, the function takes the following actions:

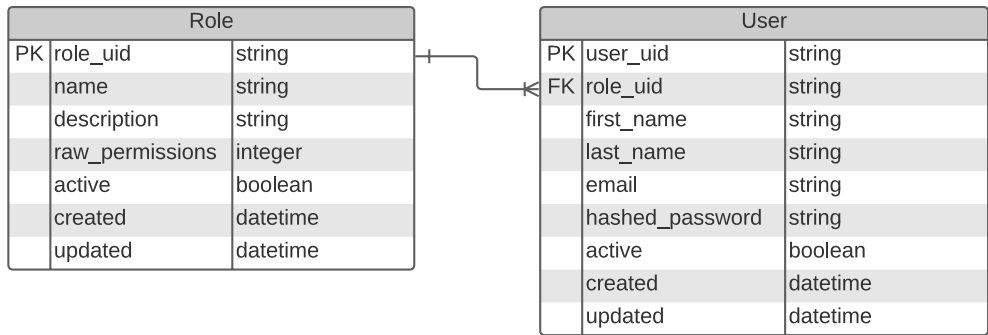
- Get a user from the database using the form email parameter
- If the user does not exist or the form password parameter is not valid
- Flash a warning message to the user and re-render the login screen
- If the user does exist and the form password parameter is valid
- Update the login manager system about the user and create a session cookie to remember them
- Get the page the user was trying to navigate to when presented with the login action
- Validate the request for that page
- Redirect the user to that page

USER AND ROLE MODELS

One of the goals of adding a login mechanism to the MyBlog application was to layer state information on top of the HTTP protocol to remember the user. Using the Flask-Login system, you're getting closer to that, but we need to have a unique identifier to save in the session cookie. The identifier can be used to retrieve information about the user.

Both of those things require us to define and implement a user data structure. Additionally, we want to give users a role to control what they can do in the MyBlog application. We will create Python SQLAlchemy classes called `User` and `Role` that will be stored in a SQLite database. An in-depth discussion about databases and accessing them with SQLAlchemy will be presented in the next chapter. For now, here is an ERD (Entity Relationship Diagram) showing the `User` and `Role` models you're about to create:

Role / User Entity Relationship Diagram



The diagram shows a one-to-many relationship between the roles and users; a role can be used by many users, but a user can have only one role.

The MyBlog application will store and present information it remembers, like users, their roles, blog content, and comments. Each item will need to be defined and implemented as you create more features the application supports. Because everything the MyBlog application can present is stored in a database, you'll start by using a database term, models.

The `app/models.py` module holds all of the database models that define and implement everything the MyBlog application stores. Since you're at the point of needing a `User` model to enable users to login into the system, let's create the `app/models.py` file now. The first thing to do in the `app/models.py` file is to import the modules needed:

```
from contextlib import contextmanager
from enum import Flag, auto
from Flask-bcrypt import (
    generate_password_hash,
    check_password_hash
)
from . import db
from Flask-Login import UserMixin
from uuid import uuid4
from datetime import datetime
```

The import statements give the `app/models.py` module access to functionality needed to create the `User` and `Role` classes. The `User` class is used with the Flask-Login extension to authenticate a user of the MyBlog application. Authenticating a user means identifying and verifying the user is known to the MyBlog application and can access features available to known users.

We'll begin by creating the `User` class. The `User` contains information about the user; their name, email address, password, and the unique id associated with the user will be stored in the HTTP session cookie.

When creating a database table, a common practice is to use an auto-incrementing integer value as the unique id associated with each record in the table. Instead of this, the MyBlog application will use UUID values for this unique id, called the primary key, for records in the table. The pros and cons for taking this approach is covered in the next chapter.

To make that simpler when creating the `User` class, a small function is created to supply UUID string values when `User` records are created and inserted into the database:

```
def get_uuid():  
    return uuid4().hex
```

The `get_uuid()` function uses the imported `uuid4()` function to create UUID values and then returns the hex string version of that value. Returning the hex string version makes the unique id value a little shorter.

The next thing to add to `app/models.py` is the definition of the `User` class. This class uses multiple inheritance to get some built-in functionality from the modules imported at the top of `app/models.py`. The first is the `UserMixin` class which gives child classes methods the Flask-Login system expects to access `User` information.

The second is the `db.Model` class, which comes from the `db` instance variable created and initialized in the `app` module. The `db.Model` class gives a child class the SQLAlchemy functionality needed to interact with the database and define the columns in a table associated with the attributes of the class:

```

class User(UserMixin, db.Model):      #A
    __tablename__ = "user"           #B
    user_uid = db.Column(db.String, primary_key=True, default=get_uuid)      #C
    role_uid = db.Column(db.Integer, db.ForeignKey("role.role_uid"), nullable=False)  #D
    first_name = db.Column(db.String, nullable=False)      #E
    last_name = db.Column(db.String, nullable=False)      #E
    email = db.Column(db.String, nullable=False, unique=True, index=True)      #E
    hashed_password = db.Column("password", db.String, nullable=False)      #E
    active = db.Column(db.Boolean, nullable=False, default=False)      #E
    created = db.Column(db.DateTime, nullable=False, default=datetime.utcnow)      #F
    updated = db.Column(db.DateTime, nullable=False, default=datetime.utcnow,
        onupdate=datetime.utcnow)      #F

    def get_id(self):
        return self.user_uid

    @property
    def password(self):
        raise AttributeError("user password can't be read")

    @password.setter
    def password(self, password):
        self.hashed_password = generate_password_hash(password)

    def verify_password(self, password):
        return check_password_hash(self.hashed_password, password)

    def __repr__(self):
        return f"""
        user_uid: {self.user_uid}
        name: {self.first_name} {self.last_name}
        email: {self.email}
        active: {'True' if self.active else 'False'}
        role_uid: {self.role.role_uid}
        name: {self.role.name}
        description: {self.role.description}
        permissions: {self.role.permissions}
        """

```

#A The User class multiply inherits from the UserMixin and db.Model classes

#B Define the table name in the database where records of this class will be stored

#C Define the unique id value for User records using the get_uuid function

#D Define the role_uid as a relationship to the Role class (yet to be defined)

#E Define other User attributes

#F Define record auditing timestamp attributes

The User class inherits from the imported UserMixin and db.Model classes. This means the User class IS-A UserMixin class and IS-A db.model class and has access to both classes' methods and attributes.

By inheriting from the UserMixin class, the User class gets methods the LoginManager() instance created in the app package needs to function. These methods use attributes defined in the User class to authenticate a user.

The User class also inherits from the db.Model and is how the SQLAlchemy functionality is added to the class giving it access to the database. The User model defines the structure

of a single row of data in a table named "user", where each defined attribute is a column in a database table record.

The `get_id()` method overrides the method of the same name provided by the `UserMixin` class. The default `get_id()` method returns a `self.id` value, but since the `User` class defines the unique id attribute name as `user_uid`, it's necessary to override the default behavior. The `get_id()` method is used whenever the `LoginManager` instance needs to determine if the unique identifier stored in the session cookie relates to a real user in the system.

Note the pair of `password()` methods that create a write-only attribute on the `User` class. Since it's not useful to read the password, the method decorated with `@property` raises an attribute error. The `password()` method decorated with `@password.setter` creates the write behavior. The method intercepts setting the password attribute and generates a cryptographically strong hash of the password stored in the `hashed_password` class attribute. Even though `hashed_password` is how the attribute is named, the corresponding database column is named "password".

The `verify_password()` method is used in the `auth.py` module to determine if the password retrieved from the login form template matches the hashed version stored for the user.

The `User` class also has a relationship to the `Role` class that provides the authorization permissions assigned to a user. The `Role` class associates permissions with a user to authorize what a known user can do. In the MyBlog application, there are three roles defined, registered user, editor, and administrator. A registered user can view and create content and comment on the content created by other users. An editor has the permissions of a user and can edit content and comment of any user and delete either when necessary. The administrator has all the permissions of an editor and user, can change user permissions, and access administrator-only features.

Let's define the `Role` class:

```
class Role(db.Model):      #A
    class Permissions(Flag): #B
        REGISTERED = auto() #B
        EDITOR = auto() #B
        ADMINISTRATOR = auto() #B

    __tablename__ = "role" #C
    role_uid = db.Column(db.String, primary_key=True, default=get_uuid) #D
    name = db.Column(db.String, nullable=False, unique=True) #E
    description = db.Column(db.String, nullable=False) #E
    raw_permissions = db.Column(db.Integer) #E
    users = db.relationship("User", backref="role", lazy="joined") #E
    active = db.Column(db.Boolean, nullable=False, default=True) #E
    created = db.Column(db.DateTime, nullable=False, default=datetime.utcnow) #F
    updated = db.Column(db.DateTime, nullable=False, default=datetime.utcnow,
                        onupdate=datetime.utcnow) #F

    @property
    def permissions(self):
        return Role.Permissions(self.raw_permissions)
```

```
#A The Role class inherits from the db.Model class
#B Create the internally defined Permissions class
#C Define the table name in the database where records for this class will be stored
#D Define the unique id value for Role records using the get_uuid function
#E Define other Role attributes
#F Define record auditing timestamp attributes
```

Like the `User` class, the `Role` class inherits from `db.Model` so it can interface with the database functionality.

The internally defined `Permissions` class is worth mentioning. Embedding the `Permissions` class this way places it within the `Role` class's namespace, but what does it do?

Because it inherits from the `enum.Flag` class, it creates a mechanism to contain bitmasks for each of the permission attributes; `REGISTERED`, `EDITOR`, and `ADMINISTRATOR`. A bitmask is a way to store multiple Boolean values in a single integer. The `Permissions` class definition creates a structure equivalent to this:

```
Permissions(Flag):
    REGISTERED = 1      # binary value 001
    EDITOR = 2          # binary value 010
    ADMINISTRATOR = 4   # binary value 100
```

Each attribute exists as a distinct single bit within an integer value. This also means the values can be combined using bitwise Boolean operations without conflict to contain multiple attribute values. For example, a variable with a binary value of 111 indicates the `REGISTERED`, `EDITOR`, and `ADMINISTRATOR` flags are set.

The Python `Flag` class lets you use binary logical operations with the attributes. For example, this Python statement is True:

```
7 == (
    Permissions.REGISTERED |
    Permissions.EDITOR |
    Permissions.ADMINISTRATOR
).value
```

This is because 7 is represented in binary as 111, and the logical OR operator `"|"` creates the bitmask returned by the value attributes as 7.

Using the `Permissions` class will be useful when identifying what a user is authorized to do. It's also useful when initializing the database role lookup table. There isn't a way to store a `Flag` bitmask in the database directly, so it's stored as the equivalent integer in the database. The `permissions` attribute defined by the `@property` decorator translates the database integer value back into a `Flag` representation for use within the MyBlog application.

The "role" table is a lookup table and isn't modified after the MyBlog application is running. To provide `Role` information it has to be initialized with data that can be associated with users as they are registered. To initialize the "role" table add this code to the end of the `Role` class definition:

```

@staticmethod
def initialize_role_table():
    roles = [ #A
        { #A
            "name": "user", #A
            "description": "registered user permission", #A
            "raw_permissions": Role.Permissions.REGISTERED.value #A
        }, #A
        { #A
            "name": "editor", #A
            "description": "user has ability to edit all content and comments", #A
            "raw_permissions": (Role.Permissions.REGISTERED |
Role.Permissions.EDITOR).value #A
        }, #A
        { #A
            "name": "admin", #A
            "description": "administrator user with access to all of the application",
#A
            "raw_permissions": (Role.Permissions.REGISTERED | Role.Permissions.EDITOR |
Role.Permissions.ADMINISTRATOR).value #A
        } #A
    ] #A
    with db_session_manager() as db_session:
        for r in roles:
            role = db_session.query(Role).filter(Role.name ==
r.get("name")).one_or_none()

            if role is None: #B
                role = Role( #B
                    name=r.get("name"), #B
                    description=r.get("description"), #B
                    raw_permissions=r.get("raw_permissions") #B
                ) #B
            else: #C
                role.description = r.get("description") #C
                role.raw_permissions = r.get("raw_permissions") #C

            db_session.add(role)
    db_session.commit()

```

#A Define the list of role attributes to configure

#B If the role name doesn't exist, create and add it

#C If the roles name does exist, update it with the configuration data

The `initialize_role_table()` method is within the role class's namespace, but because it's a `@staticmethod`, it can be called without having an instance of the `Role` class. This function is called by the `create_app()` application factory after the database is created to initialize the "role" lookup table with data.

All of the mechanics are in place so a user can login. All that remains is creating the `login.html` form to let the user identify themselves to the system.

9.2.2 User Login

The most straightforward thing to do is code up an HTML form to gather user input for the email and a password. With the inclusion of a submit button, the form contents would be

sent to the MyBlog server as an HTTP POST request, and you could process the form information.

Because users can make unintentional and intentional errors when entering form data, it's necessary to validate the form input information. Are the email and password within required length restrictions? Are the email and password present at all? Does the email address conform to a standardized format? Implementing these kinds of validation steps is extra work that's difficult to get right. Fortunately, there's another Flask extension that dramatically simplifies form handling, Flask-WTF.

FLASK-WTF

The Flask-WTF extension integrates the more generalized WTForms package into Flask. Using the extension allows you to bind MyBlog server code to HTML form elements and automate handling those elements when the corresponding form is received by a handler using the HTTP POST method.

To create the login form and its validation, add a new file to the `app/auth` package named `forms.py`:

```
from flask-wtf import FlaskForm    #A
from wtforms import PasswordField, BooleanField, SubmitField    #B
from wtforms.fields.html5 import EmailField    #B
from wtforms.validators import DataRequired, Length, Email, EqualTo    #C

class LoginForm(FlaskForm):    #D
    email = EmailField(    #E
        "Email",    #E
        validators=[DataRequired(), Length(    #E
            min=4,    #E
            max=128,    #E
            message="Email must be between 4 and 128 characters long"    #E
        ), Email()],    #E
        render_kw={"placeholder": " "}    #E
    )    #E
    password = PasswordField(    #F
        "Password",    #F
        validators=[DataRequired(), Length(    #F
            min=3,    #F
            max=64,    #F
            message="Password must be between 3 and 64 characters long"    #F
        )],    #F
        render_kw={"placeholder": ""}    #F
    )    #F
    remember_me = BooleanField(" Keep me logged in")    #G
    cancel = SubmitField(    #H
        label="Cancel",    #H
        render_kw={"formnovalidate": True},    #H
    )    #H
    submit = SubmitField("Log In")    #I
```

#A Import the `FlaskForm` class

#B Import the field type classes to create in the form

#C Import the field validation classes used to validate the form elements

#D Create the `LoginForm`, inheriting from the base `FlaskForm` class

#E Create the email form element and validators

#F Create the password form element and validators

```
#G Create the remember me form element
#H Create the form cancel button
#I Create the form submit button
```

The `LoginForm` class defines an object to create when the form is rendered containing all HTML elements in the form. The elements have a required first parameter, the element's name used for labels, the element form name, and the id value. The `validators` parameter defines a list of validation steps the element must pass for the form to be considered valid. The `render_kw` parameter is optional and defines additional HTML form attributes to be rendered with the element.

The `email` element is an instance of the `EmailField` class. The class constructor has a parameter "Email" used as-is for any label rendered with the element and converted to lowercase when used for name and id values. The `validators` define that an entry is required, the length must be between 4 and 128 characters long inclusive, and the element value must be in a valid email format.

The `password` element is an instance of the `PasswordField` class. The `PasswordField` renders the element as an HTML type of password, so the user-entered text is shown as a sequence of asterisk characters. The class constructor has a parameter "Password" used in the same manner as the email element. The `validators` define that a password is required, and the length must be between 3 and 64 characters long inclusive.

Notice the `render_kw={"placeholder": " "}` parameters on both the email and password attributes. These are necessary to make the Bootstrap input elements visual functionality work as intended.

The `remember_me` element is an instance of the `BooleanField` class. This will be rendered as an HTML checkbox with a label of "Keep me logged in".

The `cancel` element is an instance of the `SubmitField` class and creates a cancel button that will take the user to the home screen when clicked.

The `submit` element is also an instance of the `SubmitField` class and creates a submit button when the form is rendered.

With the `forms.py` module in place, you'll need to create an HTML page where the `LoginForm` class will render the contained elements.

LOGIN FORM

The `login.html` file is created in the `app/auth/templates` directory where the `auth_bp` Blueprint can access it.

```

{% extends "base.html" %}      #A
{% import "macros.jinja" as macros %}    #B

{% block content %}
<div class="login-wrapper mx-auto mt-3">
  <div class="container login">
    <form method="POST" novalidate>
      <div class="card">
        <h5 class="card-header">
          User Login
        </h5>
        <div class="card-body">
          <div class="card-text">
            {{form.csrf_token}}      #C
            <div class="form-floating mb-3">      #D
              {{form.email(class_="form-control")}}      #D
              {{form.email.label(class_="form-label")}}      #D
              {{macros.validation_errors(form.email.errors)}}      #D
            </div>      #D
            <div class="mb-3">      #E
              {{form.password(class_="form-control")}}      #E
              {{form.password.label(class_="form-label")}}      #E
              {{macros.validation_errors(form.password.errors)}}      #E
            </div>      #E
            <div class="mb-3">      #F
              {{form.remember_me}}      #F
              {{form.remember_me.label(class_="form-check-label")}}      #F
            </div>      #F
          </div>
          <div class="card-footer text-end">
            {{form.cancel(class="btn btn-warning me-2")}}      #G
            {{form.submit(class="btn btn-primary")}}      #H
          </div>
        </div>
      </form>
    </div>
  </div>
{% endblock %}

{% block styles %}
  {{ super() }}
  <link rel="stylesheet" type="text/css" href="{{ url_for('.static',
    filename='css/login.css') }}" />
{% endblock %}

```

#A The login.html template inherits from base.html so it gets all the MyBlog page elements

#B Import the macros.jinja macros file, which we'll get to next

#C Cross-Site Request Forgery protection token talked about at the end of this chapter

#D Create the email element on the page, passing the element Bootstrap class information

#E Create the password element on the page, passing the element Bootstrap class information

#F Create the remember me element on the page, passing the element Bootstrap class information

#G Create the cancel button, pass the element Bootstrap class information

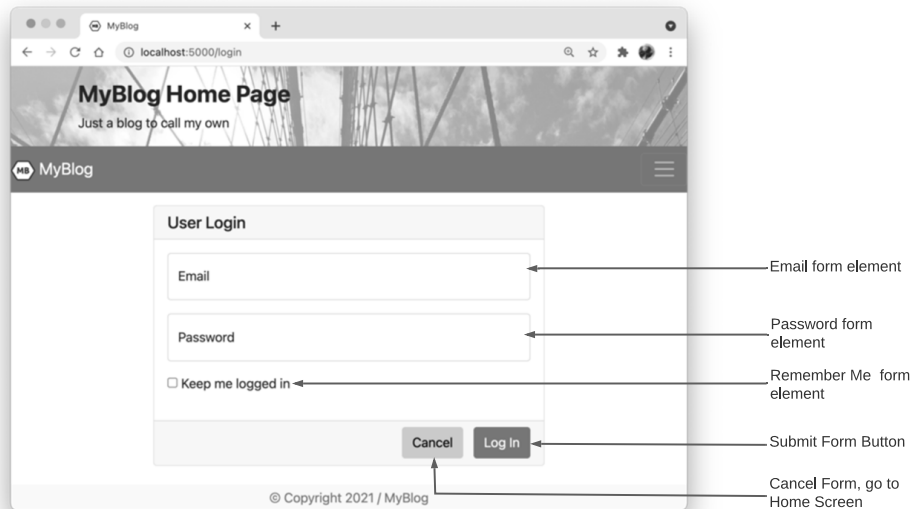
#H Create the submit button, pass the element Bootstrap class information

Recall back to the `auth.py` module and the `login()` handler assigned to the `"/login"` route. There are two lines of code of interest related to rendering the login form and connecting it to an instance of the `LoginForm` created in `forms.py`:

```
form = LoginForm()
:
return render_template("login.html", form=form)
```

These two lines of code are important when the `login()` handler is invoked by an HTTP GET or POST request. The first creates the `LoginForm` class instance variable. The second passes that `form` instance to `render_template` as the second parameter, giving the Jinja template engine access to the `form` when rendering the `login.html` template elements. This is how the `LoginForm` definition is connected to the `login.html` template and how the validation runs when the form is submitted to the `login()` handler when it receives an HTTP POST request.

Running the code in `CH_09/examples/01` and navigating to `127.0.0.1:5000/login` presents the login form to the user:



JINJA MACROS

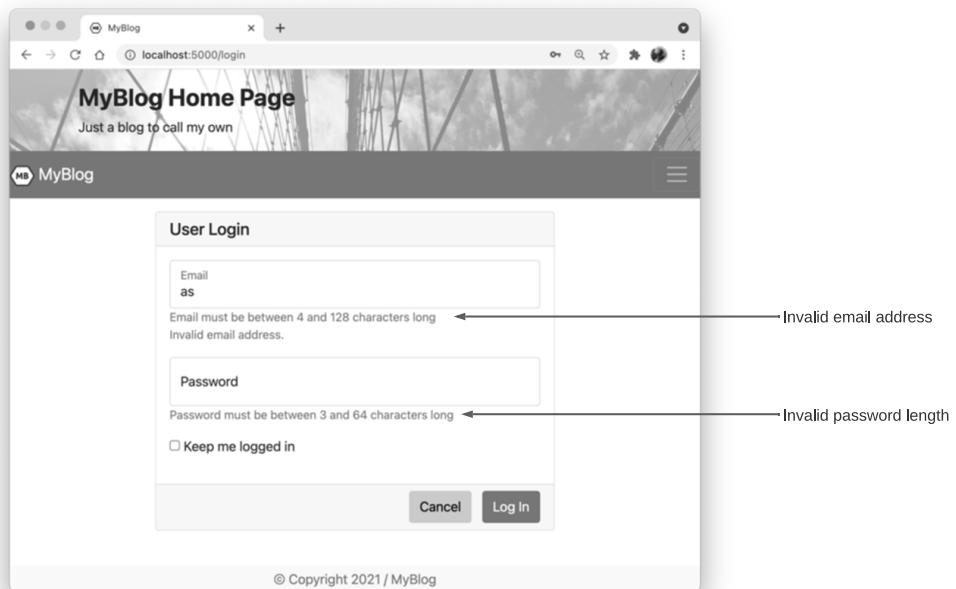
Notice this line in the `login.html` template that's part of the email and password definition sections:

```
{{macros.validation_errors(form.email.errors)}}
```

This references a Jinja macro in the `app/templates/macros.jinja` file imported at the top of the `login.html` template. The `validation_errors()` macro handles displaying any `LoginForm` validation errors to the user so they can be corrected:

```
{% macro validation_errors(errors) %}
    {% if errors %}
        {% for error in errors %}
            <div class="text-danger small">{{error}}</div>
        {% endfor %}
    {% endif %}
{% endmacro %}
```

A macro is a function definition in the Jinja template engine. The `validation_errors()` macro receives a list of `LoginForm` validation errors. It first checks if there are any errors at all, and if so, iterates over that list displaying the error message in small red text below the form field that failed validation. The results of entering an invalid email address and a password of only two characters renders the `login.html` to the user like this:



If you enter a valid email address, a password with an acceptable length, and click the submit button, you'll notice nothing happens. The `login.html` template is rendered again, and there is no information presented to the user about what, if anything, happened.

Looking back at the `login()` function in the `auth.py` file there is a conditional check in the code right after trying to find a user by their email address:

```
user = db_session.query(User).filter(User.email == form.email.data).one_or_none()
if user is None or not user.verify_password(form.password.data):
    flash("Invalid email or password", "warning")
    return redirect(url_for("auth_bp.login"))
```

If the user is not found, it has a value of `None`, and the code executes the `flash()` message function and redirects to the login route, and the `login.html` template is rendered again. No user was found because none have been created in the application yet. The intended code for this condition ran, and the user got redirected to the login screen. However, why isn't the `flash()` function doing anything to inform the user by displaying the invalid email or password message?

9.3 Flask Flash Messages

The Flask `flash()` function exists to provide feedback to users about events and activities in an application. When a message is created and sent to the `flash()` function, the message is appended to a list of messages available in the context of the next request, and only the next request, and therefore the next rendered template.

The template has to access the message list and add the messages to the rendered HTML to display the flash messages. A direct way to do this is to iterate over the flash messages list using a Jinja for loop and create an HTML unordered list of the messages as part of the rendered HTML. We're going to take advantage of Bootstrap to render the messages so they are displayed temporarily and don't disrupt the intended template flow.

9.3.1 Improving the Login Form

The Bootstrap framework provides a component called Toasts, which are lightweight alert messages that "float" above the content. Toasts have been made popular in both mobile and desktop operating systems. They are useful in the MyBlog application because they don't disrupt the template layout and are transient and disappear soon after the message is presented.

Since any URL endpoint handler can call the `flash()` function, it's useful to centralize where the flash messages are handled. The `base.html` template is ideal for this as it's intended to be inherited from by every template in the MyBlog system.

Creating a Bootstrap toast involves a significant amount of HTML code that would need to be added to the `base.html` template file. A better option is to pull the flash message handling out of the `base.html` template and create a Jinja macro. The `flask_flash_messages()` macro function is added to the `app/templates/macros.jinja` file:

```

{% macro flask_flash_messages() %}      #A
{% with messages = get_flashed_messages(with_categories=true) %}      #B
  {% if messages %}      #C
    <div aria-live="polite"
        aria-atomic="true"
        class="position-relative">
      <div class="toast-container position-absolute top-0 end-0 p-3"
          style="z-index: 2000; opacity: 1;">
        {% for category, message in messages %}      #D
          {% set category = "white" if category == "message" else category %}
          {% set text_color = "text-dark" if category in [
            "warning",
            "info",
            "light",
            "white",
          ] else "text-white"
          %}
          <div class="toast bg-{{category}}"
              role="alert"
              aria-live="assertive"
              aria-atomic="true">
            <div class="toast-header bg-{{category}} {{text_color}}">
              {% set toast_title = category if category in [
                "success", "danger", "warning", "info"
              ] else "message" %}
              <strong class="me-auto">MyBlog: {{toast_title.title()}}</strong>
              <button type="button"
                  class="btn-close"
                  data-bs-dismiss="toast"
                  aria-label="Close"></button>
            </div>
            <div class="toast-body {{text_color}}">
              {{message}}
            </div>
          </div>
        {% endfor %}
      </div>
    </div>
  {% endif %}
{% endwith %}
{% endmacro %}

```

#A Begin the definition of the `flask_flash_messages()` macro

#B Begin a with context block to get the flash messages

#C Are there any flash messages to process?

#D Begin the for loop to iterate over the list of flash messages

Most of the `flask_flash_messages()` macro concerns itself with generating the Bootstrap styling required by toast messages. The toast messages are added to the rendered template but aren't displayed to the user immediately. To do that requires JavaScript code to show the messages. The JavaScript code has to run every time a template that inherits from `base.html` is rendered, so create an `app/static/js/base.js` file:

```
(function() {
  var option = {
    animation: true,
    delay: 3000
  }
  var toastElements = [].slice.call(document.querySelectorAll('.toast'))
  toastElements.map(function (toastElement) {
    toast = new bootstrap.Toast(toastElement, option)
    toast.show()
  })
})();
```

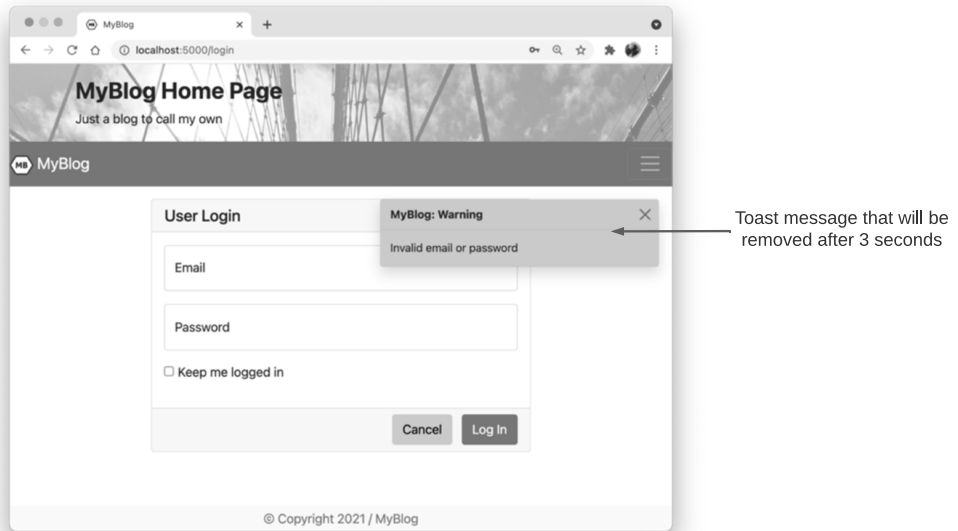
The code above creates a self-invoking anonymous JavaScript function, meaning it runs as soon as the browser JavaScript engine parses the code. This kind of function is useful when you want to run some code immediately and keep variables out of the global JavaScript scope. Because `base.js` is included at the end of the `base.html` template, the function runs after the HTML DOM elements have been created, including the toast elements.

The `option` variable is a JavaScript object literal and is something like a Python dictionary. It contains configuration information passed to the Bootstrap Toast class to animate the toast and remove it after 3000 milliseconds, or 3 seconds.

The function then creates the `toastElements` array variable containing all the toast HTML DOM elements on the page. An array in JavaScript is similar to a Python list.

JavaScript arrays have a method called `map` that applies a function to each element in the array. The anonymous function passed to the `map` creates a new Toast instance passing the `option` object and then calls the `show()` method to display the toast message in the browser window.

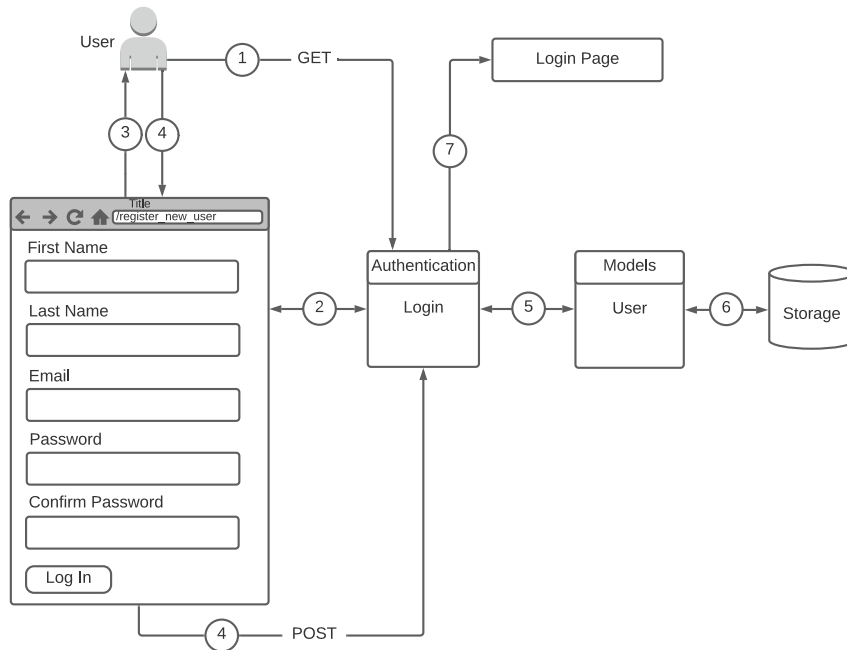
Suppose you run the application in `CH_09/examples/02` and enter a valid email address and password, but the values are unknown to the MyBlog application. In that case, the login page will be re-rendered with a toast message about the email or password being invalid:



Now that users can log into the MyBlog application, it's time to allow a new user to register with the application so they can login successfully.

9.4 Registering New Users

Registering new users on the MyBlog application also uses the Flask-Login extension. The register new user process follows a pattern similar to the login process, but instead of looking for a user, it creates and saves one to the database. From a high-level view the register new user system looks like this:



The register new user process follows this sequence of steps:

1. The user makes a GET request from their browser to the authentication register new user URL endpoint
2. The authentication register new user handler responds to the GET request by returning the rendered register new user HTML page
3. The user fills out the register new user page form fields and submits the form
4. The form is submitted to the authentication register new user system using a POST request
5. The register new user system creates a user from the form data using the models supported by the application
6. The User model saves the newly created user in the application storage system
7. The user is directed to the authentication login page to enter their login credentials

AUTH BLUEPRINT

The register new user handler is in the `app/auth.py` file. Like the login handler, a form is derived from the `FlaskForm` class in the `forms.py` module called `RegisterNewUserForm`. Add this class instance to the line of code in `app/auth.py` that imports the `LoginForm` class:

```
from .forms import LoginForm, RegisterNewUserForm
```

Add a new handler to the module at the bottom of the file:

```

@auth_bp.route("/register_new_user", methods=["GET", "POST"])    #A
def register_new_user():    #A
    if current_user.is_authenticated:    #B
        return redirect(url_for("intro_bp.home"))    #B
    form = RegisterNewUserForm()    #C
    if form.validate_on_submit():    #D
        with db_session_manager() as db_session:    #E
            user = User(    #E
                first_name=form.first_name.data,    #E
                last_name=form.last_name.data,    #E
                email=form.email.data,    #E
                password=form.password.data,    #E
                active=True    #E
            )    #E
            role_name = "admin" if user.email in current_app.config.get("ADMIN_USERS") else
            "user"    #F
            role = db_session.query(Role).filter(Role.name == role_name).one_or_none()
            #F
            role.users.append(user)    #F
            db_session.add(user)    #G
            db_session.commit()    #G
            logger.debug(f"new user {form.email.data} added")    #G
            return redirect(url_for("auth_bp.login"))    #G
    return render_template("register_new_user.html", form=form)

```

#A Mark the `register_new_user()` function as the handler for the `/register_new_user` route for both GET and POST HTTP requests

#B If the user is already authenticated redirect them to the home screen

#C Create an instance of the `RegisterNewUserForm()`

#D If the HTTP request is a POST, validate the incoming form data

#E Create new user initializing the attributes with form data

#F Get an appropriate role for the new user and set the user role

#G Add the newly created user to the database, log that a new user was created and redirect to the login page

#H If the HTTP request is a GET, render the empty `register_new_user.html` template, passing in the form instance for use in the template

9.4.1 New User Form

Repeating the pattern used for the login form, there is a Flask-WTF form and html template file created to complete the register new user functionality. Add the `RegisterNewUserForm` class definition to the `app/auth/forms.py` file:

```

from wtforms.validators import DataRequired, Length, Email, EqualTo #A
from wtforms import ValidationError #A
from ..models import User, db_session_manager #A

: intervening code

class RegisterNewUserForm(FlaskForm): #B
    first_name = StringField( #C
        "First Name", #C
        validators=[DataRequired()] #C
    ) #C
    last_name = StringField( #C
        "Last Name", #C
        validators=[DataRequired()] #C
    ) #C
    email = EmailField( #C
        "Email", #C
        validators=[DataRequired(), Length( #C
            min=4, #C
            max=128, #C
            message="Email must be between 4 and 128 characters long" #C
        ), Email()] #C
    ) #C
    password = PasswordField( #C
        "Password", #C
        validators=[DataRequired(), Length( #C
            min=3, #C
            max=64, #C
            message="Password must be between 3 and 64 characters long" #C
        ), #C
        EqualTo("confirm_password", message="Passwords must match") #C
    ] #C
    ) #C
    confirm_password = PasswordField( #C
        "Confirm Password", #C
        validators=[DataRequired(), Length( #C
            min=3, #C
            max=64, #C
            message="Password must be between 3 and 64 characters long" #C
        )] #C
    ) #C
    submit = SubmitField("Register") #D

```

#A New items to add to import section

#B Define the RegisterNewUserForm class

#C Create the first name, last name, email, password and confirm password form elements and validators

#D Create the form submit button

The code above creates the form that's passed to the register new user template to create the HTML DOM elements to render and apply validation rules when the form is submitted by a POST request. There is one additional method to add at the bottom of the RegisterNewUserForm class:

```

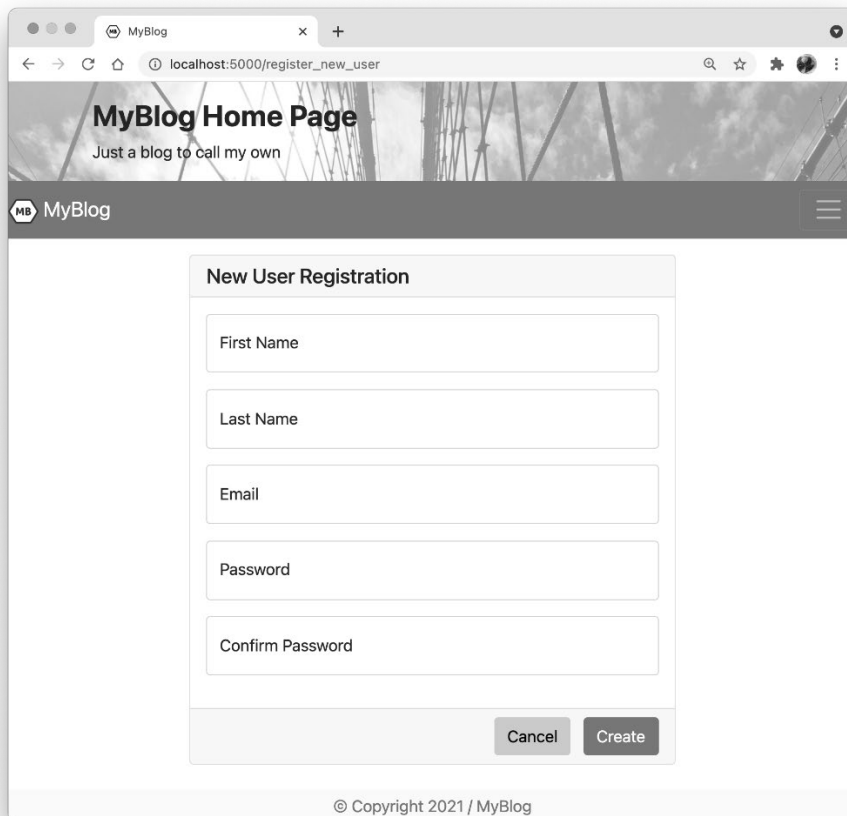
def validate_email(self, field):
    with db_session_manager() as db_session:
        user = db_session.query(User).filter(User.email == field.data).one_or_none()
        if user is not None:
            raise ValidationError("Email already registered")

```

The `validate_email()` method is a custom validation that ensures a new user isn't using an email address that already exists in the system. The `FlaskForm` class has the functionality to introspect classes that inherit from it. That introspection finds the `validate_email()` method and adds it to the validation for the "email" form field.

The `RegisterNewUserForm` class instance created in the handler is passed to the `register_new_user.html` template to render the user's form. This template is similar to the `login.html` template and isn't presented here. You can see the template by editing the `CH_09/examples/03/app/auth/templates/register_new_user.html` file.

If you move to the `CH_09/examples/03` directory and run the MyBlog application and navigate to the `127.0.0.1:5000/register_new_user` route, the form will be rendered in the browser:



The screenshot shows a web browser window with the address bar displaying `localhost:5000/register_new_user`. The page has a header with the text "MyBlog Home Page" and "Just a blog to call my own". Below the header is a navigation bar with the MyBlog logo and a hamburger menu icon. The main content area features a "New User Registration" form with the following fields: First Name, Last Name, Email, Password, and Confirm Password. At the bottom of the form are "Cancel" and "Create" buttons. The footer of the page reads "© Copyright 2021 / MyBlog".

The form provides fields for a new user to enter their first name, last name, email, a password and to confirm the password. When the register button is clicked, the form data is

sent to the server, and the email is checked to see if it already exists in the system. If the email is unknown in the MyBlog application, a new user is created and saved to the database.

9.4.2 Oh Yeah, Logging Out

Now that users can log in to the MyBlog application, we also need to provide a way to log out. Besides nice symmetry, being able to log out of an authenticated application is vital so users have control over who can access the application with their credentials.

For the MyBlog application, the logout functionality is created by adding another URL route to the auth module. When a user navigates to the logout route, no template is presented. The route handler just resets the session cookie and redirects the user to the application home page. Since the home page is available to any user, authenticated or not, this is a reasonable approach.

In the `app/auth.py` module, modify the `from Flask-Login` line like this:

```
from Flask-Login import login_user, logout_user, current_user
```

And add this to the bottom of the `app/auth.py` module:

```
@auth_bp.route("/logout")    #A
def logout():                #A
    logout_user()            #B
    flash("You've been logged out", "light")    #C
    return redirect(url_for("intro_bp.home"))    #D
```

#A Add a new route and handler for logging a user out of the system

#B Call the Flask-Login `logout_user()` function to log the user out

#C Flash a message to inform the user they've been logged out

#D Redirect the user to the application home page

9.4.3 Navigation

You've created a working authentication system, but currently, it's accessible primarily by entering the URL into the browser navigation bar. Let's add the login/logout URL routes to the Bootstrap navigation system.

The authentication system has two mutually exclusive states as a user; you can only be logged in or logged out. Because of this, the authentication system is represented in the navigation menu as a single item that will toggle between states depending on the user's current authentication status.

Keeping with the idea of single responsibility and not overcomplicating the `base.html` template, the login/logout menu functionality will exist as a Jinja macro in the `macros.jinja` file:

```

{% macro build_login_logout_items(current_user) %}      #A
  {% if not current_user.is_authenticated %}          #B
    {% if request.endpoint == "auth_bp.login" %}      #C
      <a class="nav-link ml-2 active" aria-current="page"
href="{{url_for('auth_bp.login')}}">                  #C
    {% else %}    #C
      <a class="nav-link ml-2" href="{{url_for('auth_bp.login')}}">    #C
    {% endif %}    #C
    Login
  </a>
  {% else %}    #D
    <a class="nav-link ml-2" href="{{url_for('auth_bp.logout')}}">    #E
    Logout    #E
  </a>    #E
  {% endif %}
{% endmacro %}

```

#A Begin the `build_login_logout_items` macro, passing in the current user from the `base.html` template

#B Is the current user unauthenticated?

#C Present the login menu item and route as highlighted or not depending on the current route

#D Otherwise, if the current user is authenticated

#E Present the logout menu item and route

The `build_login_logout_items()` macro has to be added to the `base.html` template so it can be rendered. Modify the section of code in the template that creates the navigation menu to add this functionality:

```

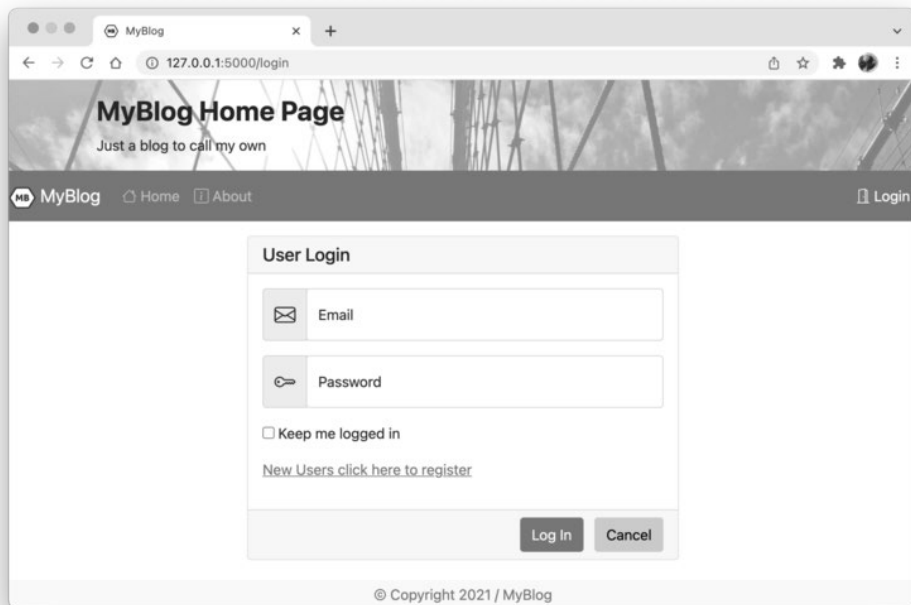
<div class="collapse navbar-collapse justify-content-between"    #A
  id="navbarSupportedContent">
  <div class="navbar-nav mr-auto">
    {{ macros.build_nav_item(nav_item) }}
  </div>
  <div class="navbar-nav">    #B
    {{ macros.build_login_logout_items(current_user) }}    #B
  </div>    #B
</div>

```

#A Right and left justify the two `navbar-nav` sections

#B Create the second `navbar-nav` section and call the macro to render the login/logout items

With the above changes in place, the MyBlog web application will display a highlighted login menu item when the login menu item is clicked and rendered. Run the application from the `CH_09/examples/04` directory:



9.5 Confirming New Users

When a potential user of the MyBlog application registers with the system it's useful to confirm they are who they say they are. This is often done by sending an email with a confirmation link to the email address they used to register with. Since the MyBlog application uses the email address as a unique identifier for the user, sending a confirmation email to that email closes the loop that the user intended to register for access to the MyBlog application.

We'll add the ability to send email to the MyBlog application so we can send these confirmation emails.

9.5.1 Sending Email

Like the choice to use SQLite as the database, we'll implement a straightforward email system that works for MyBlog, but may have to be re-considered if used in a larger system.

To send email we'll need access to an SMTP service. SMTP (Simple Mail Transport Protocol) is how email is sent and delivered across the internet. You can use a Google Gmail account (preferably one you create just for this purpose) to be the SMTP service MyBlog will use.

I won't go into detail about how to configure a Gmail account to make it available to MyBlog as an SMTP server. A quick internet search will reveal many links that describe the steps to do so.

Using a Gmail account works well but has limitations on the number of emails that can be sent per month. Currently that limit is 500 messages per day, which is more than enough for the purposes of the MyBlog application.

We'll also use a Python module named `yagmail`, which makes it easy to configure and send emails from an application using Google's Gmail.

When a new user registers with the MyBlog application on the New User Registration form we want to add two things:

- A Boolean field called `confirmed` on the User database model, initially set to `False`
- Send an email with a confirmation link to the registering users email address

Adding a `confirmed` field to the User model is simple enough and will be included in the example program that accompanies this section. Sending the confirmation link email needs to be thought about.

EMAILER

We could send the email directly from the `auth.py` module's `register_new_user()` function, and that would work fine. However, sending email via SMTP is a relatively slow process and doing so directly in the URL handler impacts the performance of the entire MyBlog application.

To improve the process we'll use a Python thread to decouple sending an email from Flask's URL processing. A Python thread is a section of the application program code that runs alongside the main application code and shares the same memory space. Python will context switch between the main application code thread and any other currently executing threads automatically.

Using a thread to send SMTP email lets the Flask URL handler continue and complete rather than waiting for email to be sent. This makes sending emails asynchronous from the MyBlog main application.

Rather than manage the thread manually, we'll use a module called `unsync`. The `unsync` module provides a decorator we place on a function to make that function run asynchronously. Depending on how the `unsync` decorator is used will determine if the decorated function runs as part of a Python `asyncio` application, a separate process using the `multiprocessing` module, or as a thread. MyBlog will use the last of these approaches.

We'll create a new module called `app/emailer.py` that has a single function, `send_mail()`:

```
from flask import current_app
import yagmail    #A
from unsync import unsync    #B

yag = yagmail.SMTP(    #C
    user=current_app.config.get("SMTP_USERNAME"),    #C
    password=current_app.config.get("SMTP_PASSWORD")    #C
)    #C

@unsync    #D
def send_mail(to, subject, contents):    #E
    yag.send(to=to, subject=subject, contents=contents)    #E
```

```
#A Import the yagmail module to send mail using a Gmail account
#B Import the unsync decorator from the unsync module
#C Configure yagmail with the username and password of the Gmail account to use for sending SMTP mail
#D Decorate the send_mail() function to make it run in a Python thread
#E Create the function the rest of the MyBlog application uses to send email
```

The code above creates a simple wrapper function called `send_mail()` around the call to send an email using `yagmail`. The `send_mail()` function is decorated with `@unsync` which will run it as a thread.

CONFIRMATION EMAIL

Now that MyBlog has a way to send emails, let's use it to send a confirmation email to newly registered users. The confirmation email will contain a link back to the MyBlog application. The link contains encrypted information that is sent along when the user clicks the link. When MyBlog handles a call to the link it decrypts the information to determine if the request was valid. If it is, the user is confirmed in the database.

The encoded information also contains a current timestamp. When the link is clicked and the application handles that request the timestamp is compared to the current time. If the application handles the request within a timeout period, the user is confirmed. However, if the user waited longer than the defined timeout period, the confirmation link is considered expired and the user isn't confirmed. The timeout value is set in the `settings.toml` file as 12 hours, which can be changed.

To send the new user confirmation email we'll add two function calls to the existing `register_new_user()` function handler. The first is a call to a new function `send_confirmation_email(user)` and the second is call to the Flask `flash()` function notifying the user with a toast message to check for the confirmation email:

```

@auth_bp.get("/register_new_user")
@auth_bp.post("/register_new_user")
def register_new_user():
    if current_user.is_authenticated:
        return redirect(url_for("intro_bp.home"))
    form = RegisterNewUserForm()
    if form.cancel.data:
        return redirect(url_for("intro_bp.home"))
    if form.validate_on_submit():
        with db_session_manager() as db_session:
            user = User(
                first_name=form.first_name.data,
                last_name=form.last_name.data,
                email=form.email.data,
                password=form.password.data,
                active=True
            )
            role_name = "admin" if user.email in current_app.config.get("ADMIN_USERS") else "user"
            role = db_session.query(Role).filter(Role.name == role_name).one_or_none()
            role.users.append(user)
            db_session.add(user)
            db_session.commit()
            send_confirmation_email(user)    #A
            timeout = current_app.config.get("CONFIRMATION_LINK_TIMEOUT")    #B
            flash((
                "Please click on the confirmation link just sent "    #B
                f"to your email address within {timeout} hours "    #B
                "to complete your registration"    ))    #B
            logger.debug(f"new user {form.email.data} added")
            return redirect(url_for("intro_bp.home"))
    return render_template("register_new_user.html", form=form)

```

#A Call to new `send_confirmation_email(user)` function to send email

#B Call Flask `flash()` functionality to notify the user to check their email within the confirmation link timeout

Let's take a look at the `send_confirmation_email()` function:

```

def send_confirmation_email(user):
    confirmation_token = user.confirmation_token()    #A
    confirmation_url = url_for(    #B
        "auth_bp.confirm",    #B
        confirmation_token=confirmation_token,    #B
        _external=True    #B
    )    #B
    timeout = current_app.config.get("CONFIRMATION_LINK_TIMEOUT")    #C
    to = user.email    #C
    subject = "Confirm Your Email"    #C
    contents = (    #C
        f""""Dear {user.first_name},    #C
        Welcome to MyBlog, please click the link to confirm your email within {timeout}
        hours:    #C
        {confirmation_url}    #C
        Thank you!    #C
        """"    #C
    )    #C
    send_mail(to=to, subject=subject, contents=contents)    #C

```

#A Call a new user method to construct a confirmation token

#B Construct a URL to insert in the email that when clicked will inform MyBlog the user has confirmed
#C Construct and send an email with the confirmation URL to the user

The `send_confirmation_email()` function calls a new method of the `User` model `confirmation_token()` to build a unique token with an expiration timeout. It then builds a URL to a new URL handler, `auth_bp.confirm`. The `_external=True` parameter causes `url_for()` to create a full URL that will work in the context when a user clicks the link from their email client.

Once the confirmation link is created an email is generated inline (rather than in a template) containing the confirmation link and sent to the new user. If the new user clicks on the confirmation link within the 12 hour time limit their account is confirmed.

USER CONFIRMATION TOKEN

Because the confirmation token is unique for each user, it's generated by a new method attached to the `User` model class:

```
def confirmation_token(self):
    serializer = URLSafeTimedSerializer(current_app.config["SECRET_KEY"])
    return serializer.dumps({"confirm": self.user_id})
```

This method uses the `URLSafeTimedSerializer()` function to create a serializing instance based on the Flask `SECRET_KEY` and includes the current timestamp. The serializer instance is used to create the unique token based on the new user's `user_id` value.

CONFIRM USER HANDLER

When a new user clicks on the confirmation link in their email, that action makes a request to a new URL handler in the auth module to confirm the token passed in the request is valid:

```
@auth_bp.get("/confirm/<confirmation_token>")    #A
@login_required    #B
def confirm(confirmation_token):
    if current_user.confirmed:    #C
        return redirect(url_for("intro_bp.home"))    #C
    try:
        # is the confirmation token confirmed?
        if current_user.confirm_token(confirmation_token):    #D
            with db_session_manager() as db_session:    #E
                current_user.confirmation = True    #E
                db_session.add(current_user)    #E
                db_session.commit()    #E
                flash("Thank you for confirming your account")
    # confirmation token bad or expired
    except Exception as e:    #F
        logger.exception(e)    #F
        flash(e.message)    #F
        return redirect(url_for("auth_bp.resend_confirmation"))    #F
    return redirect(url_for("intro_bp.home"))
```

#A Register new /confirm URL route with Blueprint
#B Confirming a token requires the user to login
#C If the user is already confirmed, redirect them to the home page
#D Confirm the token is valid
#E If the token was valid, set the current users confirmation status to True and save it in the database

#F If confirming the token raised an exception, log it, inform the user and redirect them to the resend confirmation page

USER CONFIRM TOKEN

```
def confirm_token(self, token):
    serializer = URLSafeTimedSerializer(current_app.config["SECRET_KEY"])
    with db_session_manager() as session:
        confirmation_link_timeout = current_app.config.get("CONFIRMATION_LINK_TIMEOUT")
        timeout = confirmation_link_timeout * 60 * 1000
        try:
            data = serializer.loads(token, max_age=timeout)
            if data.get("confirm") != self.user_uid:
                return False
            self.confirmed = True
            session.add(self)
            return True
        except (SignatureExpired, BadSignature) as e:
            return False
```

The `confirm_token()` URL handler creates a serializer instance just as the `confirmation_token()` creator method did. It then enters a database context manager, gets the confirmation timeout value and the confirmation data sent in the request.

The code then compares the "confirm" value of the data dictionary to the user's `user_id` value. If the values don't match the user who clicked the link isn't the user who sent the confirmation link.

The code to confirm the token is wrapped in an exception handler to return False if the token has expired or is invalid in some way.

9.6 Resetting User Passwords

We're at a point where new users can register and confirm their email address and existing users can login to use the MyBlog application.

Now we need to create a way for existing users to reset their password if they've forgotten it. In some ways a password reset request is similar to confirming a new user, it sends a link to the user's email. A handler exists for when the user clicks the link to present the user with the password reset form.

The link sent in the reset password email contains the encrypted `user_uid` value of the requesting user, along with an expiration timeout. The timeout value is set in the `settings.toml` file at ten minutes and is configurable:

```

@auth_bp.get("/request_reset_password")    #A
@auth_bp.post("/request_reset_password")   #A
def request_reset_password():
    if current_user.is_authenticated:
        return redirect("intro_bp.home")
    form = RequestResetPasswordForm()      #B
    if form.cancel.data:
        return redirect(url_for("intro_bp.home"))
    if form.validate_on_submit():
        with db_session_manager() as db_session:    #C
            user = (                                #C
                db_session.query(User)              #C
                .filter(User.email == form.email.data)  #C
                .one_or_none()                       #C
            )    #C
            if user is not None:
                send_password_reset(user)            #D
                timeout = current_app.config.get("PASSWORD_RESET_TIMEOUT")    #D
                flash(f"Check your email to reset your password within {timeout} minutes")
            #D
            return redirect(url_for("intro_bp.home"))
    return render_template("request_reset_password.html", form=form)    #E

```

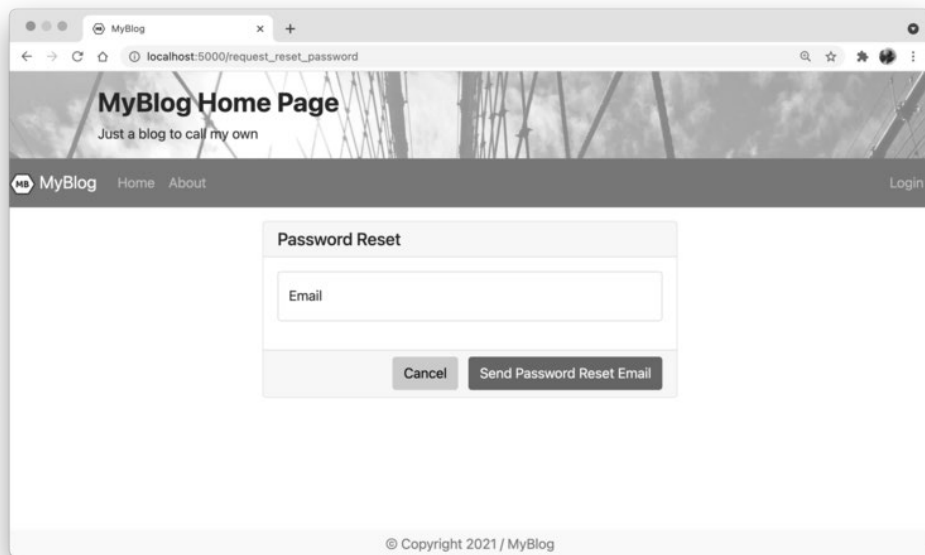
#A Register the `request_reset_password` function for both GET and POST HTTP methods

#B Create an instance of the request password form

#C Get the user associated with the email from the form

#D Send the password reset email and notify the current user to check for it within the timeout

For a HTTP GET request the handler presents this screen when running the application from the `CH_09/examples/05` directory:



The Password Reset form only present a single field for the user's email that will be used to generate the email with the password reset link.

If a user is found for the email entered in the form, that user is passed as a parameter to a new function `send_password_reset()`:

```
def send_password_reset(user):
    timeout = current_app.config.get("PASSWORD_RESET_TIMEOUT")    #A
    token = user.get_reset_token(timeout)    #A
    to = user.email    #B
    subject = "Password Reset"    #B
    contents = (    #B
        f""{user.first_name},    #B
        Click the following link to reset your password within {timeout} minutes:    #B
        {url_for('auth_bp.reset_password', token=token, _external=True)}    #B
        If you haven't requested a password reset ignore this email.    #B
        Sincerely,    #B
        MyBlog    #B
        ""    #B
    )
    send_mail(to=to, subject=subject, contents=contents)    #C
```

#A Create the encrypted reset token with the expiration timeout

#B Build the email content

#C Sent the email to the passed in user's email address

When the user clicks on the link in the reset password email a new function, `reset_password()`, handler is invoked:

```

@auth_bp.get("/reset_password/<token>")    #A
@auth_bp.post("/reset_password/<token>")    #A
def reset_password(token):
    if current_user.is_authenticated:
        return redirect("intro_bp.home")
    try:
        user_uid = User.verify_reset_token(token)    #B
        with db_session_manager() as db_session:    #C
            user = (
                db_session    #C
                .query(User)    #C
                .filter(User.user_uid == user_uid)    #C
                .one_or_none()    #C
            )    #C
            if user is None:
                flash("Reset token invalid")
                return redirect("intro_bp.home")
            form = ResetPasswordForm()
            if form.cancel.data:
                return redirect(url_for("intro_bp.home"))
            if form.validate_on_submit():
                user.password = form.password.data    #D
                db_session.commit()    #D
                flash("Your password has been reset")
                return redirect(url_for("intro_bp.home"))
    except Exception as e:
        flash(str(e))
        logger.exception(e)
        return redirect("intro_bp.home")
    return render_template("reset_password.html", form=form)

```

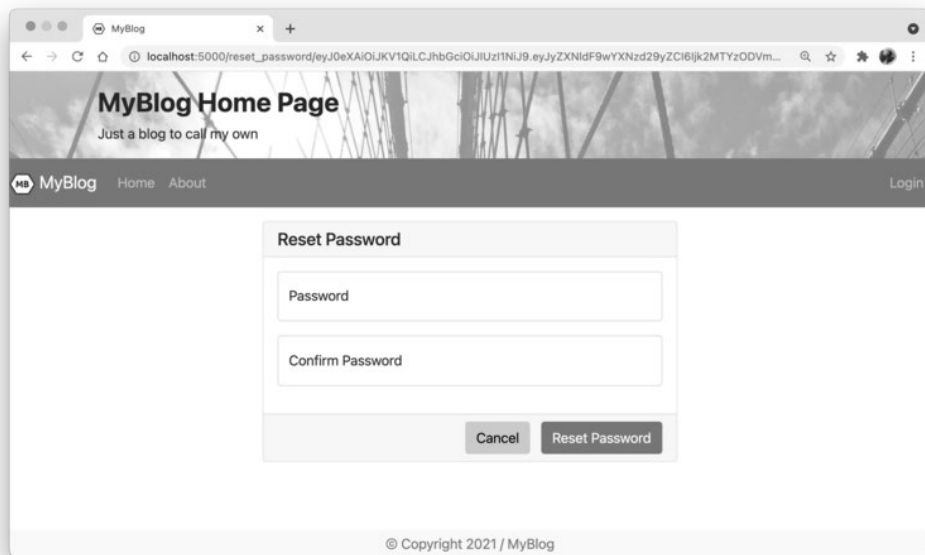
#A Register the request_reset_password function for both GET and POST HTTP methods

#B Get the user_uid from the encrypted token passed with the URL

#C Find a user matching the user_uid

#D Update and save the user's new password

When the handler is called with the HTTP GET method the form is presented in the user's browser:



The form lets the user enter and confirm a new password. When they click the “Reset Password” button the handler is called with the HTTP POST method. The `user_uid` is decrypted from the token passed with the URL and that user is searched for in the database. If the user is found and the form is valid, the user’s password is updated and saved in the database.

9.7 User Profile

The MyBlog application currently saves only a few pieces of information about registered users: first name, last name, email and password, whether they are confirmed and if they are active.

In addition to being able to reset their passwords if forgotten, users would also want to change their passwords. We’ll add a profile page that shows most of the user information and allows for password changes.

The profile is a form that presents and gathers information. It shows the user’s name and email and has input fields to enter and confirm a new password.

The screenshot shows a web browser window with the title "MyBlog". The address bar displays "localhost:5000/profile/9616385f354744a99be4a99219593809". The page header features the text "MyBlog Home Page" and "Just a blog to call my own". Below the header is a navigation bar with "MyBlog", "Home", and "About" links, and a user greeting "Welcome Doug" with a "Logout" link. The main content area displays a "User Profile" form. The form contains the following fields and buttons:

- First Name:** Doug
- Last Name:** Farrell
- Email:** (empty input field)
- Update Password:** (button)
- Confirm Updated Password:** (input field)
- Cancel** and **Update Password** (buttons at the bottom)

At the bottom of the page, there is a copyright notice: "© Copyright 2021 / MyBlog".

The form class to present the profile information is added to the `auth/forms.py` file:

```

class UserProfileForm(FlaskForm):
    first_name = StringField("First Name")
    last_name = StringField("Last Name")
    email = EmailField("Email")
    password = PasswordField(
        "Update Password",
        validators=[DataRequired(), Length(
            min=3,
            max=64,
            message="Password must be between 3 and 64 characters long"
        )],
       EqualTo("confirm_password", message="Passwords must match")
    )
    confirm_password = PasswordField(
        "Confirm Updated Password",
        validators=[DataRequired(), Length(
            min=3,
            max=64,
            message="Password must be between 3 and 64 characters long"
        )]
    )
    cancel = SubmitField(
        label="Cancel",
        render_kw={"formnovalidate": True},
    )
    submit = SubmitField(label="Okay")

```

To generate the HTML to display on the browser requires a new handler in the `auth/auth.py` module:

```

@auth_bp.get("/profile/<user_uid>")    #A
@auth_bp.post("/profile/<user_uid>")   #A
@login_required                        #B
def profile(user_uid):
    with db_session_manager() as db_session:    #C
        user = (                                #C
            db_session                          #C
                .query(User)                    #C
                .filter(User.user_uid == user_uid)    #C
                .one_or_none()                  #C
        )    #C
    if user is None:    #D
        flash("Unknown user")    #D
        abort(404)    #D
    if user.user_uid != current_user.user_uid:    #E
        flash("Can't view profile for other users")    #E
        return redirect("intro_bp.home")    #E
    form = UserProfileForm(obj=user)
    if form.cancel.data:
        return redirect(url_for("intro_bp.home"))
    if form.validate_on_submit():
        user.password = form.password.data    #F
        db_session.commit()    #F
        flash("Your password has been updated")
        return redirect(url_for("intro_bp.home"))
    return render_template("profile.html", form=form)

```

#A Register the request_reset_password function for both GET and POST HTTP methods
 #B To view a profile the user must be logged in (coming up in the next section)
 #C Get the user associated with the user_uid in the URL path
 #D If no user is found, notify the user and abort with a 404 error
 #E Prevent users from viewing profiles other than their own
 #F For a valid form submission, update the users password

The HTML template to use to render the profile form isn't show here but can be seen in the `CH_09/examples/05/auth/templates/profile.html` template file.

9.8 Security

One of the goals of building an authentication system is to provide security for an application's features and functions. That security includes what features and functions a user can perform while using the application. It also means protecting the application itself by maintaining control that only known users can access protected features and functions.

9.8.1 Protecting Routes

An authenticated user has a cryptographically secure session cookie that identifies them to the application. We can use the session and `Flask-Login` module to protect routes in the application so only users who are logged in and authenticated can navigate to those routes.

Currently, the MyBlog application only has two routes that aren't associated with authentication, the home page and the about page. To demonstrate how to protect a route, you'll temporarily create two new routes. Protecting a page is done by adding another decorator provided by the `Flask-Login` module to a URL route page handler. Add this to the import section of the `app/intro.py` module:

```
from Flask-Login import login_required    #A
```

#A Import the login_required decorator functionality from the Flask-Login module

Add a new route and handler to the `app/intro.py` module:

```
@intro_bp.route("/auth_required")    #A
@login_required    #B
def auth_required():
    return render_template("auth_required.html")
```

#A Add a new route for "/auth_required"

#B Decorate the auth required handler with the login_required functionality

The `auth_required()` handler has two decorators, `@intro_bp.route()` and `@login_required`. Stacking decorators this way is absolutely fine. The decorator functionality wraps around other decorator functionality, working from the inner level outward. In this case, the `@login_required` decorator must be placed after the `@intro_bp.route()` (or any Blueprint instance routing) to make sure the `@login_required` functionality wraps the `auth_required()` handler functionality.

With the `auth_required()` handler protected by the `@login_required` decorator, an unauthenticated user will be redirected to the login page and won't be able to access the protected `auth_required` page. This is useful when you want to allow only authenticated

users to see sensitive or private information or prevent access to forms that could change server data or functionality.

An example use case for this security is allowing only authenticated users to create and post blog content to the MyBlog application.

9.8.2 Authorizing Routes

Besides protecting URL routes in the MyBlog application so only authenticated users can access them, you'll also want to protect URL routes so only authenticated users with specific permissions can access them. This will be useful in later chapters when forms are created that only editors or administrators need to access.

To create this functionality, you'll need to create a decorator similar to `@login_required` but instead examines the user's authorization. To do this, create another module inside of the app package, `app/decorators.py`:

```
from functools import wraps
from enum import Flag
from flask import abort
from Flask-Login import current_user
from .models import Role

def authorization_required(permissions):    #A
    def wrapper(func):                    #B
        @wraps(func)                      #C
        def wrapped_function(*args, **kwargs):    #D
            if not current_user.role.permissions & permissions:    #E
                abort(403)    #F
            return func(*args, **kwargs)    #G
        return wrapped_function
    return wrapper
```

#A Create the decorator function expecting to be passed a permissions bitmask

#B Create the wrapper to receive the wrapped function

#C Use the `@wraps(func)` decorator to maintain the wrapped function signature

#D Create the wrapper to receive the wrapped function's parameters

#E Determine if the current user has the permissions necessary for this route

#F Abort with an HTTP 404 error code if the user doesn't have the required permissions

With the `authorization_required()` decorator function created, let's demonstrate its usage. Update the `app/intro.py` module and add this code to the bottom of the import section:

```
from ..decorators import authorization_required
from ..models import Role
```

With those lines added create a new URL route and handler:

```
@intro_bp.route("/admin_required")    #A
@login_required    #B
@authorization_required(Role.Permissions.ADMINISTRATOR)    #C
def admin_required():    #C
    return render_template("admin_required.html")    #C
```

#A Add a new route for `/admin_required`

#B Decorate the admin required handler with the `login_required` functionality

#C Decorate the admin required handler with the `authorization_required` functionality

With this route in place, you can run the application and try to navigate to the URL route http://127.0.0.1/admin_required. The system will generate a 404 error for all users except a user with administrator permissions. How to create an administrator user will be covered in a later chapter.

9.8.3 Protecting Forms

There's another protection for forms that we've glossed over. In both the `login.html` and `register_new_user.html` templates there's a field within the form context that looks like this:

```
<form action="" method="POST" novalidate>
    {{form.csrf_token}}
    <!--rest of the form -->
</form>
```

What is the `{{form.csrf_token}}` Jinja substitution element? If you view the source of either the `login` or `register_new_user` pages, you'll see an `<input...>` element that looks something like this:

```
<input id="csrf_token" name="csrf_token" type="hidden"
    value="IjE1NzU4NjE3OWNlMTUxYmM0Yzc3OTAyTOZiODk4NjRmNTdmZGM5OGUi.YEULPg.jVDKYLM3MMlpK
    K-BQSh2f1hwUfQ">
```

The element is a hidden (not shown on the browser page) input element with a strange-looking value. The value is generated by the MyBlog application server using the Flask `SECRET_KEY` configuration value along with the user session unique identifier. The purpose of this element is to prevent Cross-Site Request Forgery (CSRF) attacks. The `form.csrf_token` is intended to protect a request that would take action (like an HTTP POST). When the server receives a protected form it will validate both the session and `form.csrf_token` to ensure a malicious user hasn't altered it.

The protection above is provided automatically by using the Flask-WTF module. All you need to do is include the `{{form.csrf_token}}` in any form you want to protect.

9.9 Summary

This chapter has added a great deal of functionality to the MyBlog application and expanded your knowledge with practical tools and techniques used to create an application with growing complexity:

- You've learned about sessions and their importance in web applications
- You've created User and Role models to authenticate users
- An introduction to databases enabled saving users for later login
- You've built a system to register new users and allow those users to login
- You've created a mechanism to confirm new users

- You've created a system so users can reset forgotten passwords
- You've created a profile so existing users can change their passwords
- You've learned how to protect routes in the application to authenticated users
- You've created a Role permissions system to protect routes to particular users

By using the new modules you've learned about, Flask-Login, Flask-Bcrypt, Flask-WTF, and Flask-SQLAlchemy, you've created an effective authentication and authorization system. Ensuring user security is a decisive step forward towards having an application accepted by users. The flip side of that coin is protecting the application from inadvertent or unacceptable use.

The authorization system you've created is absolutely functional and useful to the MyBlog application. It is far from the last word in security. If your needs to secure a web application are more stringent you'll need to consider things like two-factor authentication, or perhaps use a third party application to host your authentication.

The next chapter will take the database information introduced here further. You'll dive deeper into designing database tables, the relationships between them, and how SQLAlchemy integrates the Python and database worlds.