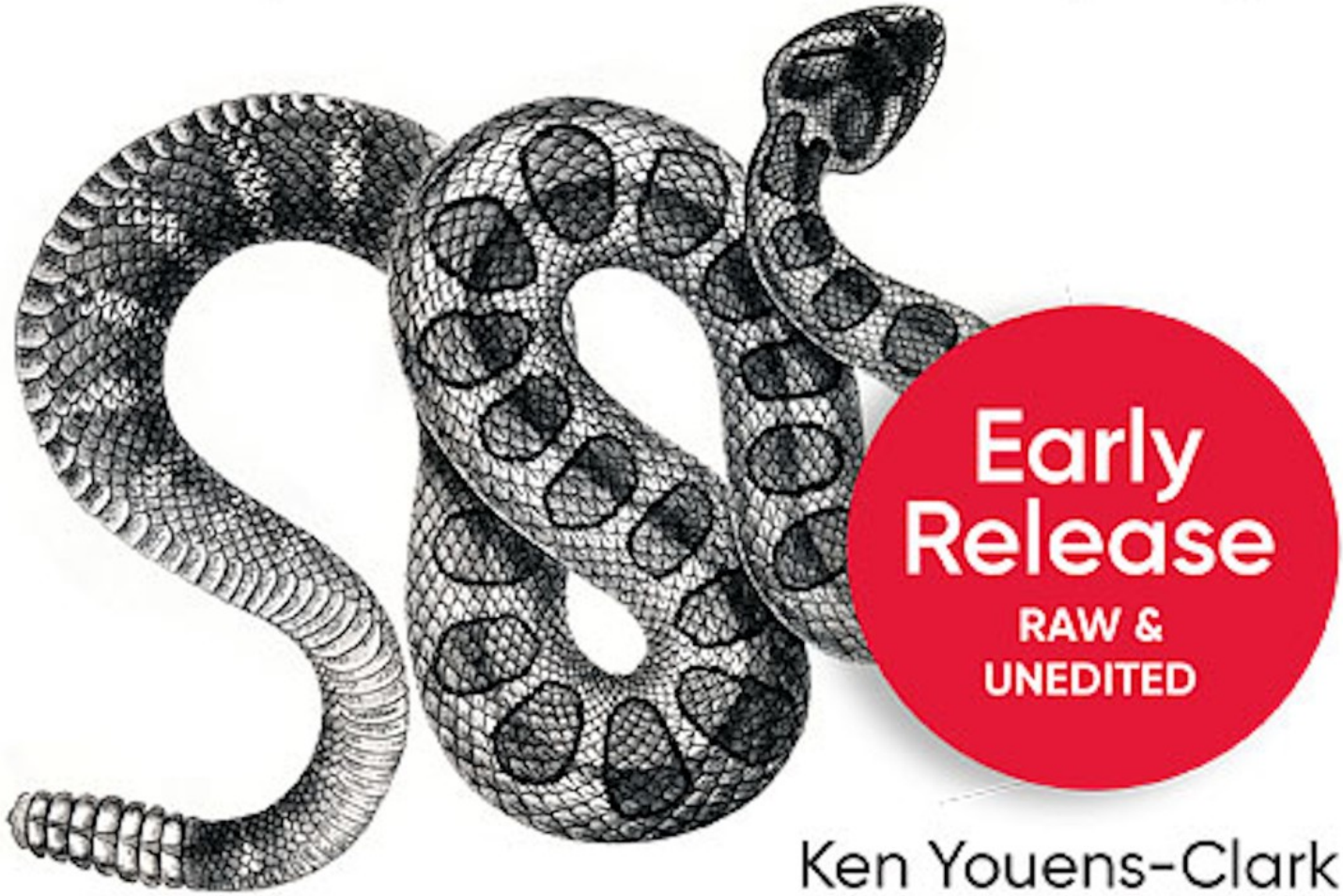
O'REILLY®

Reproducible Bioinformatics with Python

How to Write Flexible, Documented, Tested Python Code for Research Computing



Ken Youens-Clark

Reproducible Bioinformatics with Python

How to Write, Document, and Test Programs for Biology

Ken Youens-Clark

Reproducible Bioinformatics with Python

by Ken Youens-Clark

Copyright © 2021 Ken Youens-Clark. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc. , 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

Editors: Michelle Smith and Corbin Collins

Production Editor: FILL IN PRODUCTION EDITOR

Copyeditor: FILL IN COPYEDITOR

Proofreader: FILL IN PROOFREADER

Indexer: FILL IN INDEXER

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

February 2022: First Edition

Revision History for the First Edition

- 2021-01-26: First Early Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098100889 for release details.

The O’Reilly logo is a registered trademark of O’Reilly Media, Inc. Reproducible Bioinformatics with Python, the cover image, and related trade dress are trademarks of O’Reilly Media, Inc.

The views expressed in this work are those of the author(s), and do not represent the publisher’s views. While the publisher and the author(s) have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author(s) disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-10088-9

[FILL IN]

Preface

Programming is a force multiplier. We can write computer programs to free ourselves from tedious manual tasks and to accelerate research. Programming in *any* language will likely improve your productivity, but each language has different learning curves and tools that improve or impede the process of coding.

There is an old adage in business that says you have three choices:

1. Fast
2. Good
3. Cheap

Pick any two.

When it comes to programming languages, Python hits a sweet spot in that it's *fast* because it's fairly easy to learn and to write a working prototype of an idea — it's pretty much always the first language I'll use to write any program. I find Python to be *cheap* because my programs will usually run well enough on commodity hardware like my laptop or a tiny AWS instance. However, I would contend that it's not necessarily easy to make *good* programs using Python because the language itself is fairly lax. For instance, it allows one to mix characters and numbers in operations that will crash the program.

This book has been written for the aspiring bioinformatics programmer who wants to learn Python's best practices and tools such as the following:

- Since Python 3.6, you can add type hints to indicate, for instance, that a variable should be a *type* like a number or a list, and you can use the `mypy` tool to ensure the types are used correctly.

- Testing frameworks like `pytest` can exercise your code with both good and bad data to ensure that it reacts in some predictable way.
- Tools like `pylint` and `flake8` can find potential errors and stylistic problems that would make your programs more difficult to understand.
- The `argparse` module can document and validate the arguments to your programs.
- The Python ecosystem allows you to leverage hundreds of existing modules like Biopython to shorten programs and make them more reliable.

Using these practices will improve your programs, but combining them all will improve your code in compounding ways. This book is not a textbook on bioinformatics per se. The focus is on what Python offers that makes it suitable for writing scientific programs that are *reproducible*. That is, I'll show you how to design and test programs that will always produce the same outputs given the same inputs. Bioinformatics is saturated with poorly written, undocumented programs, and my goal is to reverse this trend, one program at a time.

The criteria for program reproducibility includes:

- *Parameters*: All program parameters can be set as runtime arguments. This means no hard-coded values which would require changing the source code to change the program's behavior.
- *Documentation*: A program should respond to a `--help` argument by printing the parameters and usage.
- *Testing*: You should be able to run a test suite that proves the code meets some specification.

You might expect that this would logically lead to programs which are perhaps correct, but, alas:

> *Program testing can be used to show the presence of bugs, but never to show their absence!*
>
> —Edsger Dijkstra

Bioinformatics is composed mostly of scientists who've learned programming or programmers who've learned biology (or someone like me who had to learn both). No matter how you've come to this, I want to show you the practical programming techniques that will help you write reproducible programs. I'll start with how to write programs that document and validate their arguments. Then I'll show how to write and run tests to ensure the programs actually do what they purport.

For instance, the first chapter shows you how to report the tetranucleotide frequency from a string of DNA. Sounds pretty simple, right? It's a trivial idea, but I'll take about 40 pages to show how to structure, document, and test this program. I'll spend a lot of time on how to write and test the 7 different versions of the program so that I can explore many aspects of Python data structures, syntax, modules, and tools.

Who Should Read This?

You should read this book if you care about the craft of programming, if you want to learn how to write programs that produce their own documentation, validate their parameters, fail gracefully, and work reliably. Testing is a key skill both for understanding your own code and for verifying its correctness. I'll show you how to use the tests I've written as well as how to write tests for your own programs.

To get the most out of this book, you should already have a solid understanding of Python. I will build on the skills I taught in *Tiny*

Python Projects (Manning, 2020) where I show how to use Python data structures like strings, lists, tuples, dictionaries, sets, and named tuples. You need not be an expert in Python, but I definitely will push you to understand some advanced concepts such as types, regular expressions, and ideas about higher-order functions, all of which I introduce in that book along with testing and how to use tools like `pylint`, `flake8`, `yapf`, and `pytest` to check style, syntax, and correctness. One notable difference is that I will consistently use type annotations for all code in this book and will use the `mypy` tool to ensure the correct use of types.

Programming Style: Why I Avoid OOP and Exceptions

I tend to avoid object-oriented programming (OOP). If you don't know what OOP means, that's OK. Python itself is an OO language, and almost every element from a string to a set is technically an object with internal state and methods. You will encounter enough objects to get a feel for what OOP means, but the programs I present will mostly avoid using objects to represent ideas in my programs.

That said, Chapter 1 shows how to use a `class` to represent a complex data structure. The `class` allows me to define a data structure with type annotations so that I can verify that I'm using the data types correctly. It does help to understand a bit about OOP. For instance, classes define the attributes of an object, and classes can inherit attributes from parent classes, but this essentially describes the limits of how and why I use OOP in Python. If you don't entirely follow that right now, don't worry. You'll understand it once you see it.

Instead of object-oriented code, I demonstrate programs composed almost entirely of *functions*. These functions are also *pure* in that they will only act on the values given to them. That is, pure functions never rely on some hidden, mutable state like global variables, and

they will always return the same values given the same arguments. Additionally, every function will have an associated test that I can run to verify it behaves predictably. It's my opinion that this leads to shorter programs that are more transparent and easier to test than solutions written using OOP. You may disagree and are of course welcome to write your own solutions using whatever style of programming you prefer so long as they pass the tests. The Functional Programming HOWTO documentation makes a good case for why Python is suited for functional programming (FP).

Finally, the programs in this book also avoid the use of exceptions which I think is appropriate for short programs you write for personal use. Managing exceptions so that they don't interrupt the flow of a program adds another level of complexity that I feel detracts from one's ability to understand a program. I'm generally unhappy with how to write functions in Python that return errors. Many people would raise an exception and let a `try`/`catch` block handle the mistakes. If I feel an exception is warranted, I will often choose to *not* catch it, instead letting the program crash. In this respect, I'm following an idea from the creator of the Erlang language who said:

> *The Erlang "way" is to write the happy path and not write twisty little passages full of error-correcting code.*
>
> —Joe Armstrong, Let It Crash

If you choose to write programs and modules for public release, you will need to learn much more about exceptions and error handling which is beyond the scope of this book.

Structure

The book is divided into two main parts, the first of which tackles some of the programming challenges found at the Rosalind.info[1] website. In the second part, I'll show some of my own more

complicated programs that demonstrate other patterns or concepts I feel are important to bioinformatics.

In the the first part, each chapter describes a coding challenge and provides a test suite for you to determine when you've written a working program. The Rosalind challenges are so popular that it's probable some readers will have already tackled them. If you haven't, I encourage you to try to write them, using the tests I provide, *before* you read my solutions. You'll get much more out of the material if you do this.

I also follow a theme-and-variations approach to the Rosalind solutions. If you've already written solutions, I encourage you to try writing alternate solutions using different techniques that I'll introduce. Although the "Zen of Python" says "There should be one — and preferably only one — obvious way to do it," I believe you can learn quite a bit by attempting many different approaches to a problem. I will often show many solutions in order to explore different aspects of Python syntax and data structures.

Test-Driven Development

> *More than the act of testing, the act of designing tests is one of the best bug preventers known. The thinking that must be done to create a useful test can discover and eliminate bugs before they are coded — indeed, test-design thinking can discover and eliminate bugs at every stage in the creation of software, from conception to specification, to design, coding, and the rest.*
>
> —Boris Beizer, _Software Testing Techniques_ (Thomson

Underlying all my experimentation will be test suites that I'll constantly run to ensure the programs continue to work correctly. Whenever I have the opportunity, I try to teach *test-driven development* (TDD), an idea explained in a book by that title written by Kent Beck (Addison-Wesley, 2002). TDD advocates writing tests

for code *before* writing the code. The typical cycle involves the following:

1. Add a test
2. Run all tests and see if the new test fails
3. Write the code
4. Run tests
5. Refactor code
6. Repeat

In the GitHub repository, you'll find the tests for each program you'll write. I'll explain how to run and write tests, and I hope by the end of the material you'll believe in the common sense and basic decency of using TDD. I hope that thinking about tests first will start to change the way you understand and explore coding.

Using the Command Line and Installing Python

My experience in bioinformatics has been always centered around the Unix command line. Much of my day-to-day work has been on some flavor of Linux server stitching together existing command-line programs using shell, Perl, and Python. While I might write and debug a program or a pipeline on my laptop, I will often deploy my tools to a high-performance compute (HPC) cluster where a scheduler will run my programs asynchronously, often in the middle of the night or over a weekend and without any supervision or intervention by me. Additionally, all my work building databases and websites and administering servers is done entirely from the command line, so I feel strongly that you need to master this environment in order to be successful in bioinformatics.

I used a Mac to write and test all the material, and macOS has the Terminal app you can use for a command line. I have also tested all

the programs using various Linux distributions, and the GitHub repository includes instructions on how to use a Linux virtual machine with Docker. Additionally I tested all the programs on Windows 10 using the Ubuntu distribution Windows Subsystem for Linux version 1. I *highly* recommend WSL for Windows users so as to have a true Unix command line, but Windows shells like `cmd.exe`, PowerShell, and Git Bash can work sufficiently well with some modifications as noted in the text.

While the programs are intended to *run* on the command line, I would encourage you to explore integrated development environments (IDEs) like VS Code, PyCharm, or Spyder to help you write, run, and test your programs. These tools integrate text editors, help documentation, and terminals. While I wrote all the programs, tests, and even this book using the `vim` editor in a terminal, most people would probably prefer to use at least a more modern text editor like Sublime, TextMate, Notepad++, or VS Code.

I wrote and tested all the examples using Python versions 3.8.6 and 3.9.1. Some examples use Python syntax that was not present in version 3.6, so I would recommend you not use that version. Python version 2.x is no longer supported and should not be used. I tend to get the latest version of Python 3 from the Python download page, but I've also had success using the Anaconda Python distribution. You may have a package manager like `apt` on Ubuntu or `brew` on Mac that can install a recent version, or you may choose to build from source. Whatever your platform and installation method, I would recommend you try to use the most recent version as the language continues to change, mostly for the better.

Note that I've chosen to present the programs as command-line programs and not as Jupyter Notebooks for several reasons. I really like Notebooks for data exploration, but the source code for Notebooks is stored in JSON (JavaScript Object Notation) and not as line-oriented text. This makes it very difficult to use tools like `diff` to find the differences between two Notebooks. Notebooks cannot be

parameterized, meaning I cannot pass in arguments from outside the program to change the behavior but instead have to change the source code itself. This makes the programs inflexible and automated testing impossible. While I encourage you to explore Notebooks, especially as an interactive way to run Python, I will focus on how to write well-behaved, flexible, tested command-line programs.

Getting the Code and Tests

All the code and tests are available from the book's GitHub repository. You can use the program `git` (which you may need to install) to copy that code to your computer with the following command. This will create a new directory called *biofx_python* on your computer with the contents of the repository:

```
$ git clone https://github.com/kyclark/biofx_python
```

If you enjoy using an IDE, it's possible there is a facility for cloning the repository through that interface as shown in Figure P-1. Because there are so many different IDEs that can help you manage projects and write code in many way, they all work differently. I will mostly show how to use the command line to accomplish most tasks:

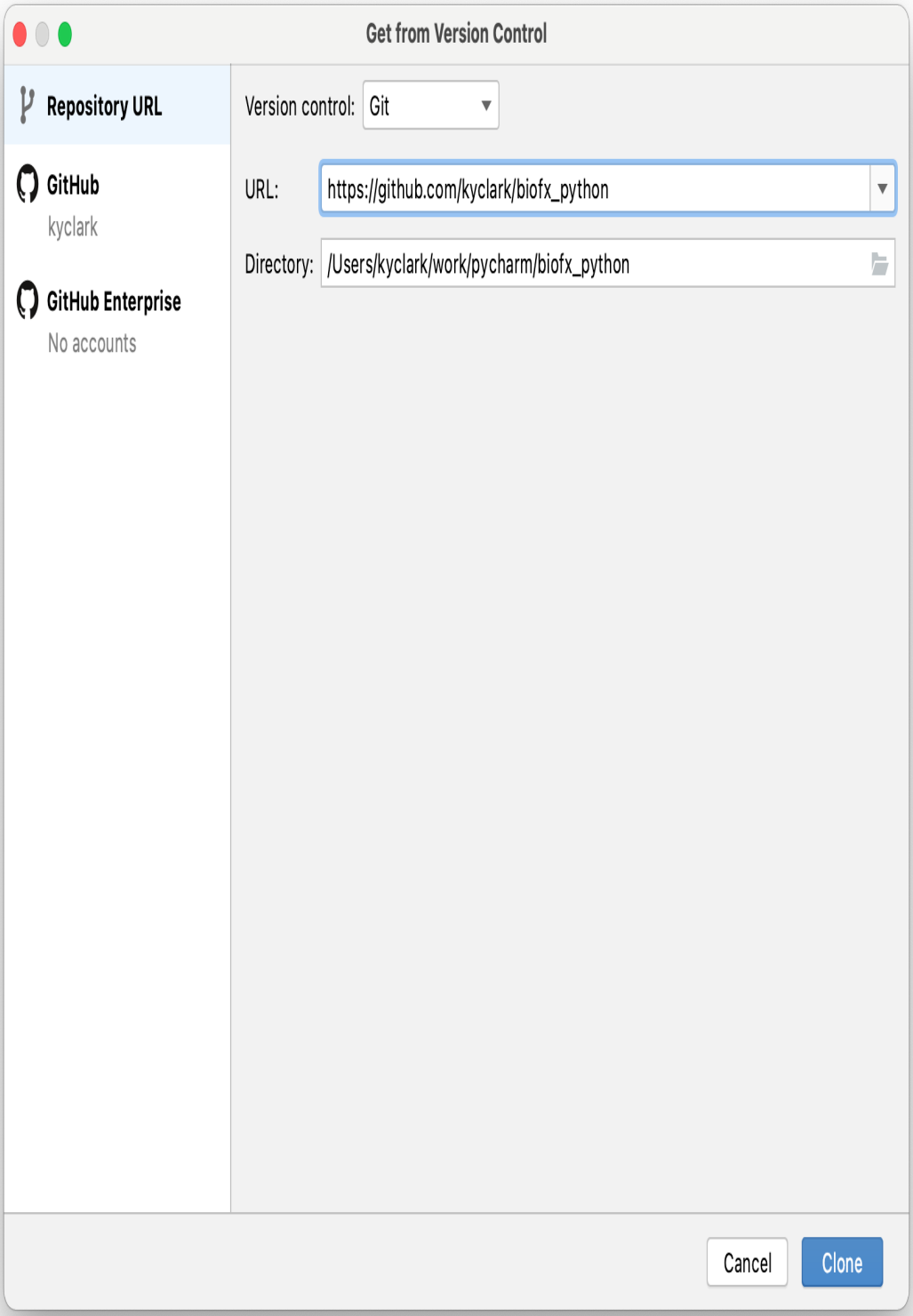

Get from Version Control
Repository URL
GitHub
kyclark
GitHub Enterprise
No accounts
Version control:
Git
URL:
https://github.com/kyclark/biofx_python
Directory:
/Users/kyclark/work/pycharm/biofx_python
Cancel
Clone

Figure P-1. The PyCharm tool can directly clone the GitHub repository for you.

> **NOTE**
>
> Some tools like PyCharm may automatically try to create a *virtual environment* inside the project directory. This is a way to insulate the version of Python and modules from other projects on your computer. Whether or not you use virtual environments is a personal preference. It is not a requirement to use them.

You may prefer to make a copy of the code into your own repository so that you can track your changes and share your solutions with others. This is called *forking* because you're breaking off from my code and adding your own programs to the repository.

To fork, do the following:

1. Create an account on GitHub.com.
2. Go to *https://github.com/kyclark/biofx_python*.
3. Click the Fork button in the upper-right corner (see Figure P-2) to make a copy of the repository into your account.

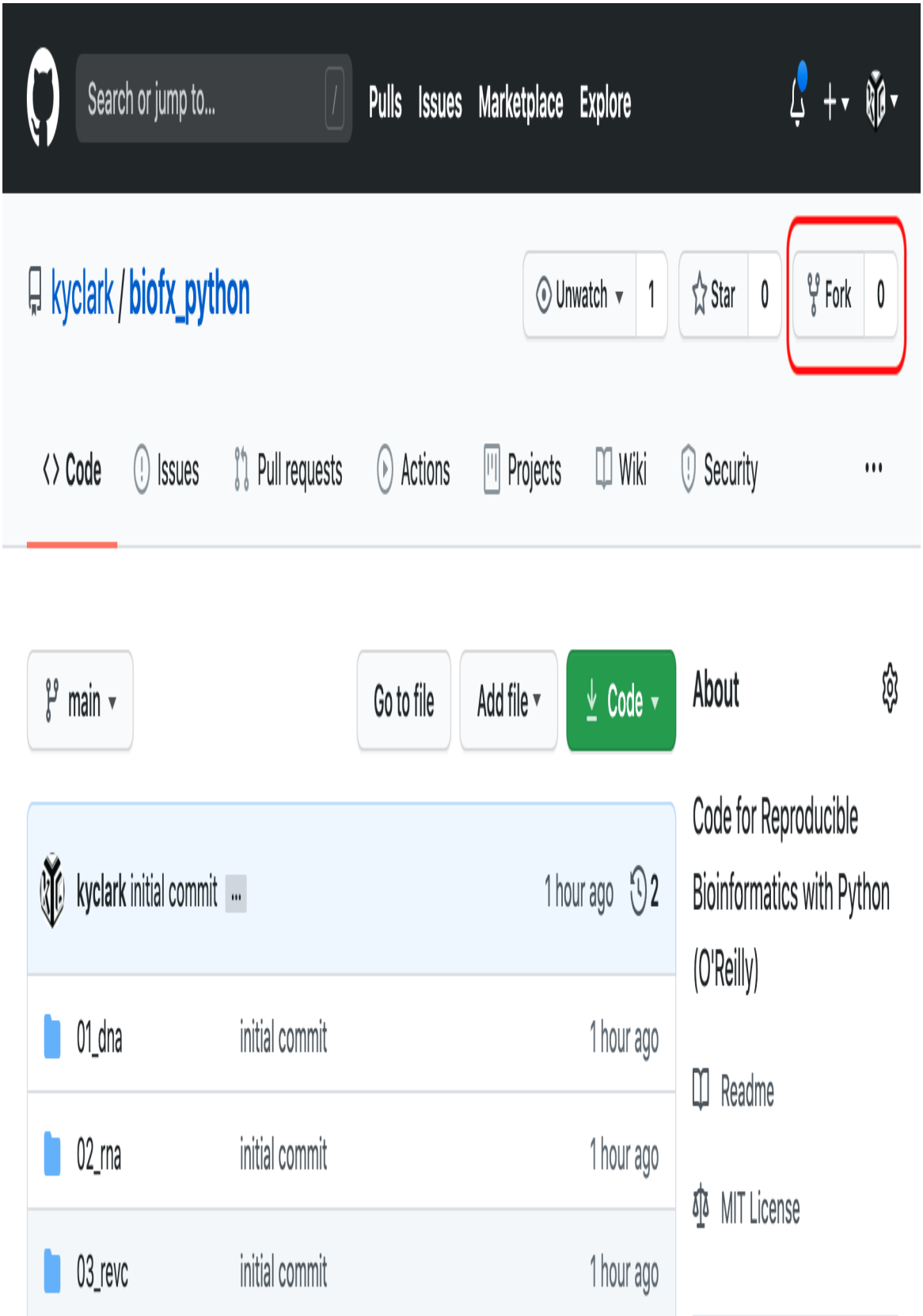

Search or jump to...
/
Pulls
Issues
Marketplace
Explore
kyclark / biofx_python
Unwatch 1
Star 0
Fork 0
Code
Issues
Pull requests
Actions
Projects
Wiki
Security
main
Go to file
Add file
Code
About
kyclark initial commit ...
1 hour ago
2
01_dna initial commit 1 hour ago
02_rna initial commit 1 hour ago
03_revc initial commit 1 hour ago
Code for Reproducible Bioinformatics with Python (O'Reilly)
Readme
MIT License

Figure P-2. The Fork button on my GitHub repository will make a copy of the code into your account.

Now that you have a copy of all my code in your own repository, you can use Git to copy that code to your computer. Be sure to replace `YOUR_GITHUB_ID` with your actual GitHub ID:

```
$ git clone https://github.com/YOUR_GITHUB_ID/biofx_python
```

If you have trouble installing Python or a command-line, I recommend using the Repl.it website as shown in Figure P-3. This is a free website that offers virtual machines preconfigured with various environments and languages. You should fork my repo into your own account so that you can configure Repl.it to interact with your own GitHub repository.

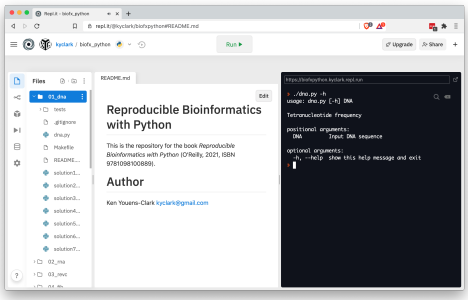


Figure P-3. The Repl.it website can import your repository and run it on a virtual machine all from a browser.

I may update the repo after you make your copy. If you would like to be able to get those updates, you will need to configure Git to set my repository as an *upstream* source. To do so, after you have cloned your repository to your computer, go into your *biofx_python* directory:

```
$ cd biofx_python
```

Then execute this command:

```
$ git remote add upstream https://github.com/kyclark/biofx_python.git
```

Whenever you would like to update your repository from mine, you can execute this command:

```
$ git pull upstream main
```

Installing Modules

You will need to install several Python modules and tools. I've included a *requirements.txt* file in the top level of the repository. This file lists all the modules needed to run the programs in the book. Some IDEs may detect this file and offer to install these for you, or you can use the following command:

```
$ python3 -m pip install -r requirements.txt
```

Alternately, this command should work:

```
$ pip3 install -r requirements.txt
```

If, for example, you want to write the exercises on Repl.it, you will need to run this command to set up your environment, as none of the modules may be installed.

Sometimes `pylint` may complain about some of the variable names in the programs, and `mypy` will raise some issues when I include modules that do not have type annotations. To silence these errors, you can create initialization files in your home directory that these programs will use to customize their behavior. In the root of the source repository, there are files called *pylintrc* and *mypy.ini* that you should copy to your home directory like so:

```
$ cp pylintrc ~/.pylintrc
$ cp mypy.ini ~/.mypy.ini
```

Alternately, you can generate a new *pylintrc* with the following command:

```
$ cd ~
$ pylint --generate-rcfile > .pylintrc
```

Feel free to customize these files to suit your own tastes.

Installing the new.py Program

I wrote a Python program called `new.py` that creates Python programs. So meta, I know. I wrote this for myself and then gave it to my students because I think it's quite difficult to start writing a program from an empty screen. The `new.py` program will create a new, well-structured Python program that uses the `argparse` module to interpret command-line arguments. It should have been installed in the preceding section when you installed the module dependencies. If not, you can use the `pip` module like so:

```
$ python3 -m pip install new-py
```

You should now be able to execute **`new. py`** and see something like this:

```
$ new.py
usage: new.py [-h] [-n NAME] [-e EMAIL] [-p PURPOSE] [-t] [-f] [--
version]
              program
new.py: error: the following arguments are required: program
```

Each exercise will suggest that you use `new.py` to start writing your new programs. For instance, in Chapter 1 you will create a program called `dna.py` in the *01_dna* directory like so:

```
$ cd 01_dna/
$ new.py dna.py
Done, see new script "dna.py".
```

If you then execute `./dna.py --help`, you will see that it generates help documentation on how to use the program. You should open the `dna.py` program in your editor, modify the arguments, and add your own code to satisfy the requirements of the program and the tests.

Note that it's never a requirement that you use `new.py` to start. I only offer this as an aid to getting started. This is, in fact, how I start every one of my own programs, but, while I find it useful, you may prefer to go a different route. As long as your programs pass the test suite, you are welcome to write your programs however you please.

Why Did I Write This Book?

Richard Hamming spent decades as a mathematician and researcher at Bell Labs. He was known for seeking out people he didn't know and asking them about their research. Then he would ask them what they thought were the biggest, most pressing, and unanswered questions in their field. If their answers for both of these weren't the same, he'd ask "So, why aren't you working on that?"

I feel that one of the most pressing problems in bioinformatics is that much of the software is poorly written and lacks proper documentation and testing, if at all. I want to show you that it's *less* difficult to use the techniques I show to write your programs using types and tests and linters and formatters because it will prove easier over time to add new features and release more and better software. You will have the confidence to know for certain when your program is correct, for at least some measure of correct.

To that end, I will demonstrate basic software development best practices. Though I'm using Python as the medium, the principles apply to any language from C to R to JavaScript. The most important thing you can learn from this book is the craft of developing, testing, documenting, releasing, and supporting software so that together we can all advance scientific research computing.

My career in bioinformatics was a product of wandering and happy accidents. I studied English literature and music in college, and then started playing with databases, HTML, and eventually learned programming on the job in the mid-1990s. By 2001, I'd become a decent Perl hacker, and I managed to get a job as a web developer for Dr. Lincoln Stein, an author of several Perl modules and books, at Cold Spring Harbor Laboratory (CSHL). He and my boss, Dr. Doreen Ware, patiently spoon-fed me enough biology to understand the programs they wanted to be written for a comparative plant genomics database called Gramene.org. I spent 13 years working on that project, learning a decent amount of science while continuing to learn more about programming and computer science.

Lincoln was passionate about sharing everything starting with all our data and code and continuing to the knowledge of how to program. He started the *Programming for Biology* course at CSHL, a two-week intensive crash course to teach basic Unix command-line skills, how to write simple programs in Perl, and the basics of this developing field of bioinformatics. The course is still being taught, although using Python, and I've had several opportunities to act as a teaching assistant. I've always found it rewarding to help someone learn a skill they will use to further their research.

It was during my tenure at CSHL that I met Bonnie Hurwitz who eventually left to pursue her Ph.D. at the University of Arizona (UA). When she started her new lab at UA, I was her first hire. Bonnie and I worked together for several years, and teaching became one of my favorite parts of the job. As with Lincoln's course, we taught basic programming skills to scientists who wanted to add more computational approaches to their research.

Some of the materials I wrote for these classes became the foundation for my first book, *Tiny Python Projects*, where I try to teach the essential elements of Python language syntax as well as how to use tests to ensure that programs are correct and reproducible — elements crucial to scientific programming. This

book picks up from there and focuses on the elements of Python that will help you write programs for biology.

Acknowledgments

I want to thank the many people who have reviewed this book including my editor, Corbin Collins, my technical reviewers Al Scherer, Brad Fulton, Bill Lubanovic, Dr. Rangarajan Janani, and Joshua Orvis, and the many other people who provided much appreciated feedback including Mark Henderson, Marc Bañuls Tornero, and Dr. Scott Cain.

In my professional career, I've been extremely fortunate to have had many wonderful bosses, supervisors, and colleagues who've helped me grow and pushed me to be better. Eric Thorsen was the first person to see I had the potential to learn how to code and helped me learn various languages and databases as well as important lessons about sales and support. Steve Reppucci was my boss at boston.com, and he provided a much deeper understanding of Perl and Unix and how to be an honest and thoughtful team leader. Dr. Lincoln Stein at Cold Spring Harbor took a chance to hire someone who had no knowledge of biology to work in his lab, and he pushed me to create programs I didn't imagine I could. Dr. Doreen Ware patiently taught me biology and pushed me to assume leadership roles and learn to write papers. Dr. Bonnie Hurwitz supported me through many years of learning high-performance computing, more programming languages, mentoring, teaching, and writing which ultimately lead to my first book. In every position, I also had many colleagues who taught me as much about programming as about being human, and I thank everyone who has helped me learn.

In my personal life, I would be nowhere without my family who has loved and supported me. My parents have shown great support throughout my life, and I surely wouldn't be the person I am without them. Lori Kindler and I have been married 25 years, and I can't

imagine a life without her. Together we generated three offspring who have been an incredible source of delight and challenge.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`

Shows commands or other text that should be typed literally by the user.

`Constant width italic`

Shows text that should be replaced with user-supplied values or by values determined by context.

> **TIP**
>
> This element signifies a tip or suggestion.

> **NOTE**
>
> This element signifies a general note.

> **WARNING**
>
> This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/kyclark/biofx_python*.

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Book Title* by Some Author (O'Reilly). Copyright 2012 Some Copyright Holder, 978-0-596-xxxx-x."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

NOTE

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and information about our books and courses, visit *http://oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

---

1 Named for Rosalind Franklin, who should have received a Nobel for her contributions to the discovering the structure of DNA.

About the Author

Ken Youens-Clark has been programming for about 25 years. After a wandering undergraduate education at the University of North Texas that started in music and ended in English literature, he learned programming on the job using various and sundry languages. Eventually he ended up in a bioinformatics lab and thought it seemed way cooler than anything he'd done before, so he stuck with that. Ken lives in Tucson, AZ, where he earned his MS in Biosystems Engineering in 2019 from the University of Arizona. When he's not coding, he enjoys cooking, playing music, riding bicycles, and being with his wife and three very interesting children.

Part I. The Rosalind.info Challenges

The chapters in this part explore the elements of Python's syntax and tooling that will enable you to write well-structured, documented, tested, and reproducible programs. I'll solve the first 14 challenges from the Rosalind.info website. These problems are short and focused and so present many different solutions that will help us explore Python. I'll also teach you how to write a program, step-by-step, using tests to guide you and to let you know when you're done.

Chapter 1. Tetranucleotide Frequency: Counting Things

Counting the bases in DNA is perhaps the “Hello, World!” of bioinformatics. The Rosalind DNA challenge describes a program that will take a sequence of DNA and will print a count of how many As, Cs, Gs, and Ts are found. There are surprisingly many ways to count things in Python, and I’ll explore what the language has to offer. I’ll also demonstrate how to write a well-structured, documented program that validates its arguments as well as how to write and run tests to ensure the program works correctly.

In this chapter, you’ll learn:

- How to start a new program using `new.py`
- How to define and validate command-line arguments using `argparse`
- How to run a test suite using `pytest`
- How to iterate the characters of a string
- Ways to count elements in a collection
- How to format strings

Getting Started

Before you start, be sure you have read “Getting the Code and Tests” in the preface. Once you have a local copy of the code repository, change into the *01_dna* directory:

```
$ cd 01_dna
```

Here you'll find several *solution*.py* programs along with tests and input data to see if the programs work correctly. To get an idea of how your program should work, start by copying the first solution to a program called `dna.py`:

```
$ cp solution1_iter.py dna.py
```

Now run the program with no arguments or with the `-h` or `--help` flags. It will print usage documentation (note that *usage* is the first word of the output):

```
$ ./dna.py
usage: dna.py [-h] DNA
dna.py: error: the following arguments are required: DNA
```

> **NOTE**
>
> If you get an error like "permission denied," you may need to run **`chmod +x dna.py`** to change the mode of the program by adding the executable bit.

This is one of the first elements of reproducibility. *Programs should provide documentation on how they work.* While it's common to have something like a README file or even a paper to describe a program, the program itself must provide documentation on its parameters and outputs. I'll show you how to use the `argparse` module to define and validate the arguments as well as to generate the documentation, meaning that there is no possibility that the usage statement generated by the program could be incorrect. Contrast this with how README files and change logs and the like can quickly fall out of sync with a program's development, and I hope you'll appreciate that this sort of documentation is quite effective.

You can see from the usage line that the program expects something like *DNA* as an argument, so let's give it a sequence. As described on the Rosalind page, the program prints the counts for each of the

bases A, C, G, and T, in order and separated by a single space each:

```
$ ./dna.py ACCGGGTTTT
1 2 3 4
```

When you go to solve a challenge on the Rosalind.info website, the input for your program will be provided as a downloaded file; therefore, I'll write the program so that it will also read the contents of a file. I can use the `cat` program (for *concatenate*) to print the contents of one of the files in the *tests/inputs* directory:

```
$ cat tests/inputs/input2.txt
AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGTCTGATAGCAGC
```

This is the same sequence shown in the example on the website. Accordingly, I know that the output of the program should be this:

```
$ ./dna.py tests/inputs/input2.txt
20 12 17 21
```

Throughout the book, I'll use the `pytest` tool to run the tests that ensure programs work as expected. When I run the command **`pytest`**, it will recursively search the current directory for tests and functions that look like tests. Note that you may need to run **`python3 -m pytest`** or **`pytest.exe`** if you are on Windows. Run this now, and you should see something like the following to indicate that the program passes all four of the tests found in the *tests/dna_test.py* file:

```
$ pytest
============================= test session starts
============================
...
collected 4 items

tests/dna_test.py ....
```

```
[100%]

============================ 4 passed in 0.41s
============================
```

> **TIP**
>
> A key element to testing software is that you *run your program with known inputs and verify that it produces correct output*. While that may seem like an obvious idea, I've had to object to "testing" schemes that simply ran programs but never verified that they behaved correctly.

Creating the Program Using new.py

If you copied one of the solutions, as shown in the preceding section, then delete that program so you can start from scratch:

```
$ rm dna.py
```

Without looking at my solutions yet, I want you to try to solve this problem. If you think you have all the information you need, feel free to jump ahead and write your own version of `dna.py` using `pytest` to run the provided tests. Keep reading if you want to go step-by-step with me to learn how to write a well-structured Python program and tests.

Every program in this book will accept some command-line argument(s) and create some output like some text on the command line or new files. I'll always use the `new.py` program described in the preface to create a basic, well-structured program. *This is not a requirement.* You can write your programs however you like, starting from whatever point you want, but your programs are expected to have the same features such as generating usage statements and properly validating arguments.

Create your `dna.py` program in the *01_dna* directory as this contains the test files for the program. Here is how I will start the `dna.py`

program. The `--purpose` argument will be used in the program's documentation:

```
$ new.py --purpose 'Tetranucleotide frequency' dna.py
Done, see new script "dna.py."
```

If you run the new `dna.py` program, you will see that it defines many different types of arguments common to command-line programs:

```
$ ./dna.py --help
usage: dna.py [-h] [-a str] [-i int] [-f FILE] [-o] str

Tetranucleotide frequency ❶

positional arguments:
  str                   A positional argument ❷

optional arguments:
  -h, --help            show this help message and exit ❸
  -a str, --arg str     A named string argument (default: ) ❹
  -i int, --int int     A named integer argument (default: 0) ❺
  -f FILE, --file FILE  A readable file (default: None) ❻
  -o, --on              A boolean flag (default: False) ❼
```

❶ The `--purpose` from `new.py` is used here to describe the program.

❷ The program accepts a single positional string argument.

❸ The `-h` and `--help` flags are automatically added by argparse and will trigger the usage.

❹ This is a named option with short (`-a`) and long (`--arg`) names for a string value.

❺ This is a named option with short (`-i`) and long (`--int`) names for an integer value.

❻ This is a named option with short (`-f`) and long (`--file`) names for a file argument.

❼ This is a boolean flag that will be `True` when either `-o` or `--on` is present and `False` when they are absent.

This program only needs the `str` positional argument, and you can use *DNA* for the `metavar` value to give some indication to the user as to the meaning of the argument. Delete all the other parameters. Note that you never define the `-h` and `--help` flags as `argparse` uses those internally to respond to usage requests. See if you can modify your program until it will produce usage that follows. If you can't produce the usage just yet, don't worry. I'll show this in the next section.

```
$ ./dna.py -h
usage: dna.py [-h] DNA

Tetranucleotide frequency

positional arguments:
  DNA         Input DNA sequence

optional arguments:
  -h, --help  show this help message and exit
```

If you can manage to get this working, I'd like to point out that this program will accept exactly one positional argument. If you try running it with 0 or 2 or more arguments, you'll see that the program will complain about any additional arguments:

```
$ ./dna.py AACC GGTT
usage: dna.py [-h] DNA
dna.py: error: unrecognized arguments: GGTT
```

Likewise, the program will reject any unknown flags or options. With very few lines of code, you have built a documented program that validates the arguments to the program. That's a very basic and important step toward reproducibility.

Using argparse

The program created by `new.py` uses the `argparse` module to define the program's parameters, validate that the arguments are correct, and to create usage documentation for the user. The `argparse` module is a standard Python module which means it's always present. There are other modules that can do these things, and you are free to use any code you like to handle this aspect of your program. Just be sure your programs can pass the tests.

I wrote a version of `new.py` for *Tiny Python Projects* you can find in the *bin* directory of that project's GitHub repo that is somewhat simpler than the version I describe in the preface as that was written more for a beginning programmer. Let's start by looking at what the `dna.py` program would look like after adding the parameter:

```
#!/usr/bin/env python3 ❶
""" Tetranucleotide frequency """ ❷

import argparse ❸


# --------------------------------------------------
def get_args(): ❹
    """ Get command-line arguments """ ❺

    parser = argparse.ArgumentParser( ❻
        description='Tetranucleotide frequency',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('dna', metavar='DNA', help='Input DNA
sequence') ❼

    return parser.parse_args() ❽


# --------------------------------------------------
def main(): ❾
    """ Make a jazz noise here """

    args = get_args() ❿
```

```
    print(args.dna)    ❿


# --------------------------------------------------
if __name__ == '__main__': ⓬
    main()
```

❶ The colloquial *shebang* (`#!`) tells the operating system to use the `env` command (*environment*) to find `python3` to execute the rest of the program.

❷ This is a *docstring* (documentation string) for the program or module as a whole.

❸ I import the `argparse` module to handle command-line arguments.

❹ I always define a `get_args()` function to handle the `argparse` code.

❺ This is a docstring for a function.

❻ The `parser` object is used to define the program's parameters.

❼ Define a `dna` argument which will be positional because the name *dna* **does not** start with a dash. The *metavar* is a short description of the argument that will appear in the short usage. No other arguments are needed.

❽ The function returns the results of parsing the arguments. The help flags or any problems with the arguments will cause `argparse` to print a usage statement/error messages and to exit the program.

❾ All programs in the book will always start in the `main()` function.

❿

The first step in `main()` will always be to call `get_args()`. If this call succeeds, then the arguments must have been valid.

11. The *DNA* value is available in the `args.dna` attribute as this is the name of the argument.

12. This is a common idiom in Python programs to detect when the program is being executed (as opposed to being imported) and to execute the `main()` function.

> **NOTE**
>
> The *shebang* line is used by the Unix shells when the program is invoked as a program like `./dna.py`. It does not work on Windows where you are required to type `python.exe dna.py` to run the program.

While this code works completely adequately, the `return` from the `get_args()` is a *dynamically generated* `argparse.Namespace` object. That is, I am using code like `parser.add_argument()` to modify the structure of this object *at runtime*, so Python is unable to know positively *at compile time* what attributes will be available in the parsed arguments or what their types would be. While it may be obvious to you that there can only be a single, required string argument, but there simply is not enough information in the code for Python to discern this.

To see why this could be a problem, alter the `main()` function to introduce a type error. That is, let's intentionally misuse the *type* of the `args.dna` value. Unless otherwise stated, all argument values returned from the command line by `argparse` are strings. If you try to divide the string `args.dna` by the integer value 2, Python will raise an exception and crash the program:

```
def main():
    args = get_args()
```

```
    print(args.dna / 2) ❶
```

❶ Dividing the string value by an integer will produce an exception.

If you run the program, it crashes as expected:

```
$ ./dna.py ACGT
Traceback (most recent call last):
  File "./dna.py", line 30, in <module>
    main()
  File "./dna.py", line 25, in main
    print(args.dna / 2)
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

Our big squishy brains know that this is an inevitable error waiting to happen, but Python can't see the problem. What I need is a *static* definition of the arguments that cannot be modified when the program is run. Read on to see how type annotations and other tools can detect these sorts of bugs.

Tools for Finding Errors in Your Code

The goal here is to write correct, reproducible programs in Python, so are there ways to spot and avoid problems like misusing a `str` in an numeric operation? The `python3` interpreter found no problems that prevented me from running the code. That is, the program is *syntactically* correct, so the code in the preceding section produces a *runtime error* because the error happens only when I execute the program. Years back I worked in a group where we joked, "If it compiles, ship it!" This is clearly a myopic approach when coding in Python.

I can use tools like linters and type checkers to find some kinds of problems in code. *Linters* are tools that check for program style and many kinds of errors beyond bad syntax. The `pylint` tool is a

popular Python linter that I use almost every day. Can it find this problem? Apparently not as it gives the biggest of thumbs-up:

```
$ pylint dna.py

--------------------------------------------------------------------
Your code has been rated at 10.00/10 (previous run: 9.78/10, +0.22)
```

The `flake8` tool is another linter that I often use in combination with `pylint` as it will report different kinds of errors. When I run `flake8 dna.py`, I get no output, which means it found no errors to report.

The `mypy` tool is a static *type checker* for Python, meaning it is designed to find misused types such as trying to divide a string by a number. Neither `pylint` nor `flake8` are designed to catch type errors, so I cannot be legitimately surprised they missed the bug.

So what does `mypy` have to say?

```
$ mypy dna.py
Success: no issues found in 1 source file
```

Well, that's just a little disappointing; however, you need to understand that `mypy` is failing to report a problem *because there is no type information*. That is, `mypy` has no information to say that dividing `args.dna` by 2 is wrong. I'll fix that.

USING PYTHON'S INTERACTIVE INTERPRETER

In the next section, I want to show you how to use Python's interactive interpreter `python3` to run short pieces of code. This kind of interface is sometimes called a REPL which stands for *Read-Evaluate-Print-Loop*, and I pronounce this so that it sort of rhymes with *pebble*. Each time you enter code in the REPL, Python will immediately read and evaluate the expressions, print the results, and then loop back to wait for more input.

Using a REPL may be new to you, so let's take a moment to introduce the idea. While I'll demonstrate using `python3`, you may prefer to use `idle3`, `ipython`, a Jupyter Notebook, or a Python console inside an IDE (integrated development environment) like VS Code as in Figure 1-1.

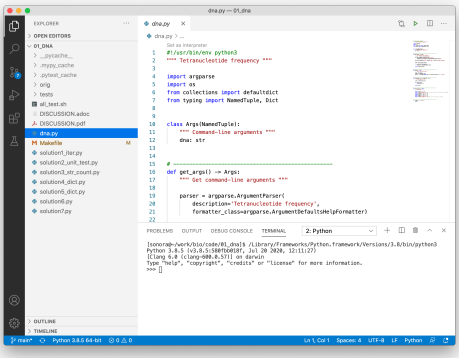


Figure 1-1. You can run an interactive Python interpreter inside VS Code.

Regardless of your choice, I *highly* encourage you to type all the examples yourself. You will learn so much from interacting with the Python interpreter. To start the REPL, type **`python3`** (or possibly just **`python`** on your computer if that points to the latest version. Here is what it looks like on my computer:

```
$ python3
Python 3.9.1 (v3.9.1:1e5d33e9b9, Dec  7 2020, 12:10:52)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more
```

```
information.
>>>
```

The standard `python3` REPL uses `>>>` as a prompt. Be sure you don't type the `>>>` prompt, only the text that follows. For example, if I demonstrate adding two numbers like this:

```
>>> 3 + 5
8
```

You should only type **`3 + 5<Enter>`**. If you include the leading prompt, you will get an error:

```
>>> >>> 3 + 5
  File "<stdin>", line 1
    >>> 3 + 5
    ^
SyntaxError: invalid syntax
```

I especially like to use the REPL to read documentation. For instance, type `help(str)` to read about Python's string class. Inside the help documentation, you can move forward with the *F* key, `Ctrl-F`, or the Space bar, and you can move backward with the *B* key or `Ctrl-B`. Search by pressing the `/` key followed by a string and then Enter. To leave the help, press *Q*. To quit the REPL, type `exit()` or `Ctrl-D`. OK, that's enough to get started.

Introducing Named Tuples

All of the programs in this book will use a named tuple data structure to return the arguments from `get_args()`. *Tuples* are essentially immutable lists, and they are often used to represent record-type data structures in Python. There's quite a bit to unpack with all that, so let's step back to lists.

To start, *lists* are ordered sequences of items. The items can be heterogeneous; in theory, this means all the items can be of different types, but, in practice, mixing types is often a bad idea. I'll use the `python3` REPL to demonstrate some aspects of lists. I recommend you use `help(list)` to read the documentation.

Use empty square brackets (`[]`) to create an empty list that will hold the sequences:

```
>>> seqs = []
```

The `list()` function will also create a new, empty list:

```
>>> seqs = list()
```

Verify that this is a list by using the `type()` function to return the variable's type:

```
>>> type(seqs)
<class 'list'>
```

Lists have methods that will add values to the end of the list like `list.append()` to add one value:

```
>>> seqs.append('ACT')
>>> seqs
['ACT']
```

and `list.extend()` to add multiple values:

```
>>> seqs.extend(['GCA', 'TTT'])
>>> seqs
['ACT', 'GCA', 'TTT']
```

If you type the variable by itself in the REPL, it will be stringified into a textual representation:

```
>>> seqs
['ACT', 'GCA', 'TTT']
```

Which is basically the same thing that happens when you `print()` a variable:

```
>>> print(seqs)
['ACT', 'GCA', 'TTT']
```

You can modify any of the values *in-place* using the index. Remember that all indexing in Python is zero-based, so 0 is the first element. Change the first sequence to be *TCA*:

```
>>> seqs[0] = 'TCA'
```

Verify that it was changed:

```
>>> seqs
['TCA', 'GCA', 'TTT']
```

As with lists, tuples are ordered sequences of possibly heterogeneous objects. Whenever you put commas between items in a series, you are in effect creating a tuple:

```
>>> seqs = 'TCA', 'GCA', 'TTT'
>>> type(seqs)
<class 'tuple'>
```

It's typical to place parentheses around tuple values to make this more explicit:

```
>>> seqs = ('TCA', 'GCA', 'TTT')
>>> type(seqs)
<class 'tuple'>
```

Unlike lists, tuples cannot be changed once they are created. If you read `help(tuple)`, you will see that a tuple is a *built-in immutable*

sequence, so I cannot add values:

```
>>> seqs.append('GGT')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
```

or modify existing values:

```
>>> seqs[0] = 'TCA'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

It's fairly common in Python to use tuples to represent records. For instance, I might represent a `Sequence` having a unique ID and a string of bases:

```
>>> seq = ('CAM_0231669729', 'GTGTTTATTCAATGCTAG')
```

While it's possible to use indexing to get the values from a tuple just as with lists, that's awkward and error-prone. Named tuples allow me to assign names to the tuple fields, which makes them much more ergonomic to use. To use named tuples, I can import the `namedtuple()` function from the `collections` module:

```
>>> from collections import namedtuple
```

As shown in Figure 1-2, I use the `namedtuple()` function to create the idea of a `Sequence` that has fields for the `id` and the `seq`:

```
>>> Sequence = namedtuple('Sequence', ['id', 'seq'])
```

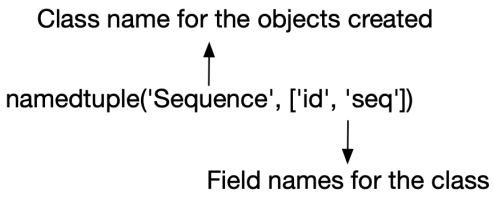


Figure 1-2. The `namedtuple()` function generates a way to make objects of the class `Sequence` that have the fields `id` and `seq`.

What exactly is `Sequence` here?

```
>>> type(Sequence)
<class 'type'>
```

I've just created a new type. You might call the `Sequence()` function a *factory* because it's a function used to generate new objects of the class `Sequence`. It's a common naming convention for these factory functions and class names to be TitleCased to set them apart.

Just as I can use the `list()` function to create a new list, I can use the `Sequence()` function to create a new `Sequence` object. I can pass the `id` and `seq` values *positionally* so as to match the order they are defined in the class:

```
>>> seq1 = Sequence('CAM_0231669729', 'GTGTTTATTCAATGCTAG')
>>> type(seq1)
<class '__main__.Sequence'>
```

Or I can use the field names and pass them as key/value pairs in any order I like:

```
>>> seq2 = Sequence(seq='GTGTTTATTCAATGCTAG', id='CAM_0231669729')
>>> seq2
Sequence(id='CAM_0231669729', seq='GTGTTTATTCAATGCTAG')
```

While it's possible to use indexes to access the ID and sequence:

```
>>> 'ID = ' + seq1[0]
'ID = CAM_0231669729'
```

```
>>> 'seq = ' + seq1[1]
'seq = GTGTTTATTCAATGCTAG'
```

The whole point of named tuples is to use the field names:

```
>>> 'ID = ' + seq1.id
'ID = CAM_0231669729'
>>> 'seq = ' + seq1.seq
'seq = GTGTTTATTCAATGCTAG'
```

The record's values remain immutable:

```
>>> seq1.id = 'XXX'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

I often want a guarantee that a value cannot be accidentally changed in my code. Python doesn't have a way to declare that a variable is *constant* or immutable. Tuples are by default immutable, and I think it makes sense to represent the arguments to a program using a data structure that cannot be altered. The inputs are sacrosanct and should (almost) never be modified.

Adding Types to Named Tuples

As nice as `namedtuple()` is, I can make it even better by importing the `NamedTuple` class from the `typing` module to use as the base class for the `Sequence`. Additionally, I can assign *types* to the fields using this syntax. Note the need to use an empty line in the REPL to indicate that the block is complete:

```
>>> from typing import NamedTuple
>>> class Sequence(NamedTuple):
...     id: str
...     seq: str
...
```

This is a different way of defining a class as shown with `namedtuple()`, and `Sequence` is still a new type:

```
>>> type(Sequence)
<class 'type'>
```

The code to instantiate a new `Sequence` object is the same:

```
>>> seq3 = Sequence('CAM_0231669729', 'GTGTTTATTCAATGCTAG')
>>> type(seq3)
<class '__main__.Sequence'>
```

I can still access the fields by names:

```
>>> seq3.id, seq3.seq
('CAM_0231669729', 'GTGTTTATTCAATGCTAG')
```

Since I defined that both fields have `str` types, you might assume this would *not* work:

```
>>> seq4 = Sequence(id='CAM_0231669729', seq=3.14)
```

I'm sorry to tell you that Python itself ignores the type information. You can see that the `seq` field should be a `str` but is actually a `float`:

```
>>> seq4
Sequence(id='CAM_0231669729', seq=3.14)
>>> type(seq4.seq)
<class 'float'>
```

So how does this help us? It doesn't help me in the REPL, but adding types to the source code will allow type checking tools like `mypy` to find such errors.

Representing the Arguments with a NamedTuple

I want the data structure that represents the program's arguments to include type information. As with the `Sequence` class, I can define a class which is derived from the `NamedTuple` type where I can *statically define the data structure with types*. I like to call this class `Args`, but you can call it whatever you like. I know this probably seems like driving a finishing nail with a sledgehammer, but, trust me, this kind of detail will pay off in the future.

Here is the kind of code produced by editing the latest `new.py` that uses the `NamedTuple` class from the `typing` module:

```
#!/usr/bin/env python3
"""Tetranucleotide frequency"""

import argparse
from typing import NamedTuple ❶


class Args(NamedTuple): ❷
    """ Command-line arguments """
    dna: str ❸


# --------------------------------------------------
def get_args() -> Args: ❹
    """ Get command-line arguments """

    parser = argparse.ArgumentParser(
        description='Tetranucleotide frequency',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('dna', metavar='DNA', help='Input DNA
sequence')

    args = parser.parse_args() ❺

    return Args(args.dna) ❻


# --------------------------------------------------
def main() -> None: ❼
    """ Make a jazz noise here """
```

```
    args = get_args()
    print(args.dna / 2) ❽


# --------------------------------------------------
if __name__ == '__main__':
    main()
```

❶ Import the `NamedTuple` class from the `typing` module.

❷ Define a `class` for the arguments which is based on the `NamedTuple` class. See following note.

❸ The class has a single field called `dna` that has the type `str`.

❹ The type annotation on the `get_args()` function shows that it returns an object of the type `Args`.

❺ Parse the arguments as before.

❻ Return a new `Args` object that contains the single value from `args.dna`.

❼ The `main()` function has no `return` statement, so it returns the default `None` value.

❽ This is the type error from the earlier program.

NOTE

If you run `pylint` on this program, you may encounter the errors “Inheriting *NamedTuple*, which is not a class. (inherit-non-class)” and “Too few public methods (0/2) (too-few-public-methods).” You can disable these warnings by adding the “inherit-non-class” and “too-few-public-methods” to the “disable” section of your `pylintrc` file or use the `pylintrc` file included in the root of the GitHub repository.

If you run this program, you'll see it still creates the same uncaught exception. Both `flake8` and `pylint` will continue to report that the program looks fine, but see what `mypy` tells me now:

```
$ mypy dna.py
dna.py:32: error: Unsupported operand types for / ("str" and "int")
Found 1 error in 1 file (checked 1 source file)
```

The error message shows that on line 32 of the `dna.py` there is a problem with the operands (arguments to the `/` operator[1]). I'm mixing string and integer values.

Without the type annotations, `mypy` would be unable to find a bug. Without this warning from `mypy`, I'd have to run my program being sure to exercise the branch of code that contains the error. In this case, it's all rather obvious and trivial, but in a much larger program of hundreds or thousands of lines of code (LOC) with many functions and logical branches (like `if`/`else`), I might not stumble upon this error. I rely on types and programs like `mypy` (and `pylint` and `flake8` and so on) to correct these kinds of errors rather relying solely on tests or, worse, waiting for users to report bugs.

Reading Input from the Command Line or a File

When you attempt to prove your program works on the Rosalind.info website, you will download a data file containing the input to your program. Usually, this data will be much larger than the sample data described in the problem. For instance, the example DNA string for this problem is 70 bases long, but the one I downloaded for one of my attempts was 910 bases.

Let's make the program read input both from the command line and a text file so that you don't have to copy and paste the contents from a downloaded file. This is a common pattern I use, and I prefer to handle this option inside the `get_args()` function since this pertains to processing the command-line arguments.

First, correct the program so that it prints the `args.dna` value without the division:

```
def main() -> None:
    args = get_args()
    print(args.dna) ❶
```

❶ Remove the division type error.

Check that it works:

```
$ ./dna.py ACGT
ACGT
```

For this next part, you need to bring in the `os` module to interact with your operating system. Add `import os` to the other `import` statements at the top, then add these two lines to your `get_args()` function:

```
def get_args() -> Args:
    """ Get command-line arguments """

    parser = argparse.ArgumentParser(
        description='Tetranucleotide frequency',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('dna', metavar='DNA', help='Input DNA
sequence')

    args = parser.parse_args()

    if os.path.isfile(args.dna): ❶
        args.dna = open(args.dna).read().rstrip() ❷

    return Args(args.dna)
```

❶ Check if the `dna` value is a file.

❷ Call `open()` to open a filehandle, then chain the `fh.read()` method to return a string, then chain the `str.rstrip()` method to remove trailing whitespace.

> **WARNING**
>
> The `fh.read()` function will read an *entire* file into a variable. In this case, the input file is small and so this should be fine, but it's very common in bioinformatics to process files which are gigabytes in size. Using `read()` on a large file could crash your program or even your entire computer. Later I will show you how to read a file line-by-line to avoid this.

Now run your program with a string value to ensure it works:

```
$ ./dna.py ACGT
ACGT
```

and then use a text file as the argument:

```
$ ./dna.py tests/inputs/input2.txt
AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGTCTGATAGCAGC
```

Now you have a flexible program that reads input from two sources. Run `mypy dna.py` to make sure there are no problems.

Testing Your Program

You know from the Rosalind description that given the input *ACGT*, the program should print `1 1 1 1` since that is the number of As, Cs, Gs, and Ts, respectively. In the *01_dna/tests* directory, there is a file called *dna_test.py* that contains tests for the `dna.py` program. I wrote these tests for you so you can see what it's like to develop a program using a method to tell you with some certainty when your program is correct. The tests are really basic — given an input string,

the program should print the correct counts for the 4 nucleotides. When the program reports the correct numbers, then it works.

Inside the *01_dna* directory, I'd like you to run `pytest` (or `python3 -m pytest` or `pytest.exe` on Windows). Pytest will recursively search for all files named *test*.py_ or *_test.py*. It will then look inside these file for any functions that have names starting with *test_*.

When you run `pytest`, you will see a lot of output, most of which are failing tests. To understand why these tests are failing, let's look at the *tests/dna_test.py* module:

```
""" Tests for dna.py """ ❶

import os ❷
import platform ❸
from subprocess import getstatusoutput, getoutput ❹

PRG = './dna.py' ❺
RUN = f'python {PRG}' if platform.system() == 'Windows' else PRG ❻
TEST1 = ('./tests/inputs/input1.txt', '1 2 3 4') ❼
TEST2 = ('./tests/inputs/input2.txt', '20 12 17 21')
TEST3 = ('./tests/inputs/input3.txt', '196 231 237 246')
```

❶ This is the docstring for the module.

❷ The standard `os` module will interact with the operating system.

❸ The `platform` module is used to determine if this is being run on Windows.

❹ From the `subprocess` module I import two functions to run the `dna.py` program and capture the output and status from the subprocess.

❺ These following lines are global variables for the program. I tend to avoid globals except in my tests. Here I want to define some

values that I'll use in the functions. I like to use UPPERCASE_NAMES to highlight the global visibility.

❻ The `RUN` variable determines how to run the `dna.py` program. On Windows, the `python` command must be used to run a Python program, but on Unix platforms the `dna.py` program can be directly executed.

❼ The `TEST*` variables are tuples that define a file containing a string of DNA and the expected output from the program for that string.

The `pytest` module will run the *test_* functions in the order in which they are defined in the test file. I often structure my tests so that they progress from the simplest cases to more complex, so there's usually no point in continuing after a failure. For instance, the first test is always that the program to test actually exists. If it doesn't, then there's no point in running more tests. I recommend you run `pytest` with the `-x` flag to indicate that `pytest` should stop on the first failing test along with the `-v` flag for verbose output.

Let's look at the first test. The function name must start with *test_* so that `pytest` will find it, so this one is called `test_exists()`. In the body of the function, I use one or more `assert` statements to check if some condition is *truthy* [2]. Here I assert that the program `dna.py` exists. This is why your program must exist in this directory or it wouldn't be found by the test.

```
def test_exists(): ❶
    """ Program exists """

    assert os.path.exists(PRG) ❷
```

❶ The function name must start with *test_* to be found by `pytest`.

❷

The `os.path.exists()` function returns `True` if the given argument is a file. If it returns `False`, then the assertion fails and so this test would fail.

The next test I write is always to check that the program will produce a usage statement for the `-h` and `--help` flags. In this case, I'm also checking that it will do so when run with no arguments. The `subprocess.getoutput()` function will run the `dna.py` with three arguments starting with the empty string and then the short and long help flags. In each case, I want to see that the program prints out text that starts with the word *usage*. It's not a perfect test. It doesn't check the contents of the documentation, only that the output appears to be something that might be a usage statement. I don't feel that every test needs to be completely exhaustive.

```
def test_usage():
    """ Prints usage with no args or for help """

    for arg in ['', '-h', '--help']: ❶
        out = getoutput(f'{RUN} {arg}') ❷
        assert out.lower().startswith('usage:') ❸
```

❶ Use three arguments: the empty string and the short and long flags for help.

❷ Run the program with the argument and capture the output into the `out` variable.

❸ Assert that the lowercased output of the program starts with the text *usage:*.

At this point in testing, I know that I have a program with the correct name that can be run to produce documentation. This means that the program is at least syntactically correct, which is a decent place

to start testing. If your program has typographical errors, then you'll be forced to correct those to even get to this point.

Running the Program to Test the Output

Now I need to see if the program does what it's actually supposed to do. There are many ways to test programs, and I like to use two basic approaches I might call *inside-out* and *outside-in*. The inside-out approach starts at the level of testing individual functions inside a program. This is often called *unit* testing as functions might be considered a basic unit of computing, and I'll get to unit testing in the solutions section. I'll start with the outside-in approach. This means that I will run the program from the command line just as the user will run it. This is a holistic approach to check if the pieces of the code can work together to create the correct output, and so it's sometimes called an *integration* test.

The first such test will pass the DNA as a command-line argument and check if the program produces the right counts formatted in the correct string:

```
def test_arg():
    """ Uses command-line arg """

    for file, expected in [TEST1, TEST2, TEST3]: ❶
        dna = open(file).read() ❷
        retval, out = getstatusoutput(f'{RUN} {dna}') ❸
        assert retval == 0 ❹
        assert out == expected ❺
```

❶ Unpack the tuples into the `file` name containing a string of DNA and the `expected` value from the program when run with this input.

❷ Open the file and read the `dna` from the contents.

❸

Run the program with the given DNA string using the `subprocess.getstatusoutput()` function that will give me both the return value from the program and the text output (also called `STDOUT` which is pronounced *standard out*).

❹ Assert that the return value is 0 which indicates success (or 0 errors).

❺ Assert that the output from the program is the string of numbers expected.

> **NOTE**
>
> Command-line programs usually indicate an error to the operating system by returning a non-zero value. If the program runs successfully, it ought to return a 0. Sometimes that non-zero value may correlate to some internal error code, but often it just means that something went wrong. The programs I write will, likewise, always strive to report 0 for successful runs and some non-zero value when there are errors.

The next test is almost identical, but this time I'll pass the filenames as the argument to the program to verify that it correctly reads the DNA from a file:

```
def test_file():
    """ Uses file arg """

    for file, expected in [TEST1, TEST2, TEST3]:
        retval, out = getstatusoutput(f'{RUN} {file}') ❶
        assert retval == 0
        assert out == expected
```

❶ The only difference from the first test is that I pass the filename instead of the contents of the file.

Now that you've looked at the tests, go back and run the tests again. This time, use `pytest -xv` where the `-v` flag is for verbose output. Since both `-x` and `-v` are short flags, you can combine them like `-xv` or `-vx`. Read the output closely and notice that it's trying to tell you that the program is printing the DNA sequence but that the test is expecting a sequence of numbers:

```
$ pytest -xv
============================= test session starts
==============================
...

tests/dna_test.py::test_exists PASSED
[ 25%]
tests/dna_test.py::test_usage PASSED
[ 50%]
tests/dna_test.py::test_arg FAILED
[ 75%]

================================== FAILURES
==================================
_____________________________________ test_arg
_____________________________________

    def test_arg():
        """ Uses command-line arg """

        for file, expected in [TEST1, TEST2, TEST3]:
            dna = open(file).read()
            retval, out = getstatusoutput(f'{RUN} {dna}')
            assert retval == 0
>           assert out == expected ❶
E           AssertionError: assert 'ACCGGGTTTT' == '1 2 3 4' ❷
E             - 1 2 3 4
E             + ACCGGGTTTT

tests/dna_test.py:36: AssertionError
=========================== short test summary info
============================
FAILED tests/dna_test.py::test_arg - AssertionError: assert
'ACCGGGTTTT' == '...
!!!!!!!!!!!!!!!!!!!!!!!! stopping after 1 failures
!!!!!!!!!!!!!!!!!!!!!!!!
```

```
========================= 1 failed, 2 passed in 0.35s
=========================
```

❶ The `>` at the beginning of this line shows that this is the source of the error.

❷ The output from the program was the string `ACCGGGTTTT` but the expected value was `1 2 3 4`. Since these are not equal, an `AssertionError` exception is raised.

Let's fix that. If you think you know how to finish the program, please jump right into your own solution. First, perhaps try running your program to verify that it will report the correct number of As:

```
$ ./dna.py A
1 0 0 0
```

And then Cs:

```
$ ./dna.py C
0 1 0 0
```

and so forth with Gs and Ts. Then run `pytest` to see if it passes all the tests.

After you have a working version, consider trying to find as many different ways as you can to get the same answer. This is called *refactoring* a program. You need to start with something that works correctly, and then you try to improve it. The improvements can be measured in many ways. Perhaps you find a way to write the same idea using less code, or maybe you find a solution that runs faster. No matter what metric you're using, keep running `pytest` to ensure the program is correct.

Solution 1: Iterating and Counting the Characters in a String

If you don't know where to start, I'll work through the first solution with you. Let's consider how you might travel through all the bases in the DNA string. In the parlance of Python, I want to *iterate* through the characters of a string. A `for` loop will do that. I can create a variable called `dna` by assigning some value to it in the REPL:

```
>>> dna = 'ACGT'
```

Note that any value enclosed in quotes, whether single- or double-quotes, is a string. Even a single character in Python is considered a string. I will often use the `type()` function to verify the type of a variable, and here I see that `dna` is of the class `str` (string):

```
>>> type(dna)
<class 'str'>
```

> **TIP**
>
> Read `help(str)` in the REPL to see all the wonderful things you can do with strings. This data type is especially important in genomics where so much of the data is strings.

Let's take a look at that `for` loop I promised you. Python sees a string as an ordered sequence of characters, and so a `for` loop will visit each character from beginning to end:

```
>>> for base in dna: ❶
...     print(base)  ❷
...
A
C
G
T
```

❶ Each character in the `dna` string will be copied into the `base` variable. You could call this `char` or `c` for *character* or whatever else you like.

❷ Each call to `print()` will end with a new line, so you'll see each base on a separate line.

> **NOTE**
>
> The `...` you see are line continuations. The REPL is showing that what's been entered so far is not a complete expression. You need to enter a blank line to let the REPL know that you're done with the code block.

Later you will see that `for` loops can be used with lists and dictionaries and sets and lines in a file — basically any iterable data structure. It's very nice.

Counting the Nucleotides

Now that you know how to visit each base in the sequence, you need to count each base rather than printing it. That means you'll need some variables to keep track of the numbers for each of the four nucleotides. One way to do this is to create four variables that hold integer counts, one for each base. I will initialize four variables for counting by setting their initial values to 0:

```
>>> count_a = 0
>>> count_c = 0
>>> count_g = 0
>>> count_t = 0
```

I could write this in one line by using the tuple unpacking syntax that I showed earlier:

```
>>> count_a, count_c, count_g, count_t = 0, 0, 0, 0
```

VARIABLE NAMING CONVENTIONS

I could have named my variables like `countA` or `CountA` or `COUNTA` or `count_A` or any number of ways, but I always stick to the suggested naming conventions in the Style Guide for Python Code also known as *PEP8* which says that function and variable names "should be lowercase, with words separated by underscores as necessary to improve readability."

Either way, I need to look at each base and determine which variable to *increment*, making it increase by 1. For instance, if the current `base` is a *C*, then I should increment the `count_c` variable. I could write this:

```
for base in dna:
    if base == 'C': ❶
        count_c = count_c + 1 ❷
```

❶ The == operator is used to compare two values for equality. Here I want to know if the current `base` is equal to the string *C*.

❷ Set `count_c` equal to 1 greater than the current value.

NOTE

The == operator is used to compare two values for equality. It works to compare two strings or two numbers. I showed earlier that division with / will raise an exception if you mix strings and numbers. What happens if you == with mixed types, for example `'3' == 3`? Is this a safe operator to use without first comparing the types?

As shown in Figure 1-3, a shorter way to increment a variable uses the += operator to add whatever is on the right-hand side (often

noted as RHS in programming language syntax) of the expression to whatever is on the left-hand side (or LHS):

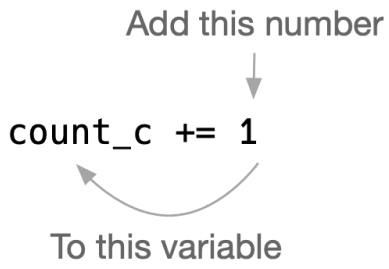


Figure 1-3. The += operator will add the value on the right-hand side to the variable on the left-hand side.

Since I have four nucleotides to check, I need a way to combine three more `if` expressions. The syntax in Python for this is to use `elif` for *else if* and `else` for any final or default case. Here is a block of code you can enter in your program or the REPL:

```
dna = 'ACCGGGTTTT'
count_a, count_c, count_g, count_t = 0, 0, 0, 0
for base in dna:
    if base == 'A':
        count_a += 1
    elif base == 'C':
        count_c += 1
    elif base == 'G':
        count_g += 1
    elif base == 'T':
        count_t += 1
```

You should end up with counts of 1, 2, 3, and 4 for each of the sorted bases:

```
>>> count_a, count_c, count_g, count_t
(1, 2, 3, 4)
```

Now I need to report the outcome to the user:

```
>>> print(count_a, count_c, count_g, count_t)
1 2 3 4
```

That is the exact output the program expects. Notice that `print()` will accept multiple values to print, and it inserts a space between each value. If you read `help(print)` in the REPL, you'll find that you can change this with the `sep` argument:

```
>>> print(count_a, count_c, count_g, count_t, sep='::')
1::2::3::4
```

The `print()` function will also put a newline at the end of the output, and this can likewise be changed using the `end` option:

```
>>> print(count_a, count_c, count_g, count_t, end='\n-30-\n')
1 2 3 4
-30-
```

Writing and Verifying a Solution

Using the preceding code, you should be able to create a program that passes all the tests. As you write, I would encourage you to regularly run `pylint`, `flake8`, and `mypy` to check your source code for potential errors. I would even go further and suggest you install the `pytest` extensions for these so that you can routinely incorporate such tests:

```
$ python3 -m pip install pytest-pylint pytest-flake8 pytest-mypy
```

Alternatively, I've placed a *requirements.txt* file in the root directory of the GitHub repo that lists various dependencies I'll use throughout the chapters. You can install all these modules with the following command:

```
$ python3 -m pip install -r requirements.txt
```

With those extensions, you can run the following command to run not only the tests defined in the *tests/dna_test.py* file but also tests for linting and type checking using these tools:

```
$ pytest -xv --pylint --flake8 --mypy tests/dna_test.py
============================= test session starts
==============================
...

tests/dna_test.py::.../dna_test.py::PYLINT SKIPPED
[ 12%]
tests/dna_test.py::FLAKE8 SKIPPED
[ 25%]
tests/dna_test.py::mypy PASSED
[ 37%]
tests/dna_test.py::test_exists PASSED
[ 50%]
tests/dna_test.py::test_usage PASSED
[ 62%]
tests/dna_test.py::test_arg PASSED
[ 75%]
tests/dna_test.py::test_file PASSED
[ 87%]
::mypy PASSED
[100%]
==================================== mypy
====================================

Success: no issues found in 1 source file
======================== 6 passed, 2 skipped in 0.50s
========================
```

NOTE

Some tests are SKIPPED when a cached version indicates nothing has changed since the last test. Run `pytest` with the `---cache-clear` option to force the tests to run. Also, you may find you fail linting tests if your code is not properly formatted or indented. You can automatically format your code using `yapf` or `black`. Most IDEs and editors will provide an auto-format option.

That's a lot to type, so I've created a shortcut for you in the form of a *Makefile* in the directory:

```
$ cat Makefile
.PHONY: test
```

```
test:
        python3 -m pytest -xv --flake8 --pylint --pylint-
rcfile=../pylintrc \
    --mypy dna.py tests/dna_test.py

all:
        ../bin/all_test.py dna.py
```

You can read the appendix on Makefiles to learn more about them. For now, it's enough to understand that, if you have `make` installed on your system, you can use the command `make test` to run the commands in the `test` target of the *Makefile*. If you don't have `make` installed or you don't want to use it, that's fine, too, but I suggest you explore how Makefiles can be used to document and automate processes.

There are many ways to write a passing version of `dna.py`, and I'd like to encourage you to keep exploring before you read the solutions. More than anything, I want to get you used to the idea of changing your program and then running the tests to see if it works. This is the cycle of *test-driven development* where I first create some metric to decide when the program works correctly. In this instance, that is the *dna_test.py* program that is run by `pytest`.

The tests ensure I don't stray from the goal, and they also let me know when I've met the requirements of the program. They are the specifications (also called *specs*) made incarnate as a program that I can execute. How else would I ever know when a program worked or was finished?

> *Without requirements or design, programming is the art of adding bugs to an empty text file. - Louis Srygley*

Testing is essential to creating *reproducible programs*. Unless you can absolutely and automatically prove the correctness and predictability of your program when run with both good and bad data, then you're not writing good software.

Solutions

The program I wrote in the introduction is the *solution1_iter.py* version in the GitHub repo, so I won't bother reviewing that version. I would like to should you several alternate solutions that progress from the simpler to more complex ideas. Please do not take this to mean they progress from worse to better. All versions pass the tests, so they are all equally valid. The point is to explore what Python has to offer for solving common problems. Note I will omit code they all have in common such as the `get_args()` function.

Solution 2: Creating a count Function and Adding a Unit Test

The first variation I'd like to show will move all the code in the `main()` function that does the counting into a `count()` function. You can define this function anywhere in your program, but I generally like `get_args()` first and `main()` second and then other functions after that but *before* they the final couplet that call `main()`.

For the following function, you will need to also import the `typing.Tuple` value:

```
def count(dna: str) -> Tuple[int, int, int, int]: ❶
    """ Count bases in DNA """

    count_a, count_c, count_g, count_t = 0, 0, 0, 0 ❷
    for base in dna:
        if base == 'A':
            count_a += 1
        elif base == 'C':
            count_c += 1
        elif base == 'G':
            count_g += 1
        elif base == 'T':
            count_t += 1

    return (count_a, count_c, count_g, count_t) ❸
```

❶ The types clearly document that the function takes a string and returns a tuple containing four integer values.

❷ This is the code from `main()` that did the counting.

❸ I return a tuple of the four counts.

There are many reasons to move this code into a function. To start, this is really a *unit* of computation — given a string of DNA, return the tetranucleotide frequency — so it makes sense to encapsulate it. This will make the `main()` shorter and more readable, and it allows me to write a unit test for the function. Since the function is called `count()`, I like to call the unit test `test_count()`. I have placed this function inside the `dna.py` program just after the `count()` function rather than in the `dna_test.py` program just as a matter of convenience. For short programs, I tend to put my functions and unit tests together in the source code, but, as projects grow larger, I will segregate unit tests into a separate module:

```
def test_count() -> None: ❶
    """ Test count """

    assert count('') == (0, 0, 0, 0) ❷
    assert count('123XYZ') == (0, 0, 0, 0)
    assert count('A') == (1, 0, 0, 0) ❸
    assert count('C') == (0, 1, 0, 0)
    assert count('G') == (0, 0, 1, 0)
    assert count('T') == (0, 0, 0, 1)
    assert count('ACCGGGTTTT') == (1, 2, 3, 4)
```

❶ The function name must start with *test*_ to be found by `pytest`. The types here show that the test accepts no arguments and, because it has no `return` statement, returns the default `None` value.

❷

I like to test functions with both expected and unexpected values to ensure they return something reasonable. The empty string should return all zeros.

❸ The rest of the tests ensure that each base is reported in the correct position.

To verify that my function works, I can use `pytest` on the `dna.py` program:

```
$ pytest -xv dna.py
========================== test session starts
==========================
...

dna.py::test_count PASSED
[100%]

============================ 1 passed in 0.01s
============================
```

The first test passes the empty string expects to get zeros for the counts. This is a judgment call, honestly. You might decide your program ought to complain to the user that there's no input. That is, it's possible to run the program using the empty string as the input, and this version will report the following:

```
$ ./dna.py ""
0 0 0 0
```

Likewise, if you passed an empty file, you'd get the same answer. The `touch` command is used to create an empty file:

```
$ touch empty
$ ./dna.py empty
0 0 0 0
```

On Unix systems, `/dev/null` is a special filehandle that returns nothing[3]:

```
$ ./dna.py /dev/null
0 0 0 0
```

You may instead feel that no input is an error and report it as such. The important thing about the test is that it forces me to think about it. Should the `count()` function raise an exception if it's given an empty string? I personally try to avoid exceptions, and later I'll explore various options for reporting errors.

Now that I have a unit test in the `dna.py` code, I can run `pytest` on that file to see if it passes:

```
$ pytest -v dna.py
============================= test session starts
==============================
...

dna.py::test_count PASSED
[100%]

============================== 1 passed in 0.03s
==============================
```

When I'm writing code, I like to write functions that do just one limited thing with as few parameters as possible. Then I like to write a *test_* plus the function name, usually right after the function in the source code. If I find I have lots of these kinds of unit tests, I might decide to move them to a separate file and ask `pytest` to execute that file.

To use this new function, modify the `main()` like so:

```
def main() -> None:
    args = get_args()
    count_a, count_c, count_g, count_t = count(args.dna) ❶
    print('{} {} {} {}'.format(count_a, count_c, count_g, count_t)) ❷
```

❶ Unpack the four values returned from `count()` into separate variables.

❷ Use `str.format()` to create the output string.

Let's focus for a moment on Python's `str.format()`. As shown in Figure 1-4, the string `'{} {} {} {}'` is a template for the output I want to generate, and I'm calling the `str.format()` function *directly on a string literal*. This is a common idiom in Python that you'll also see with the `str.join()` function. It's important to remember that, in Python, even a literal string (one that literally exists inside your source code in quotes) is an *object* upon which you can call *methods*.

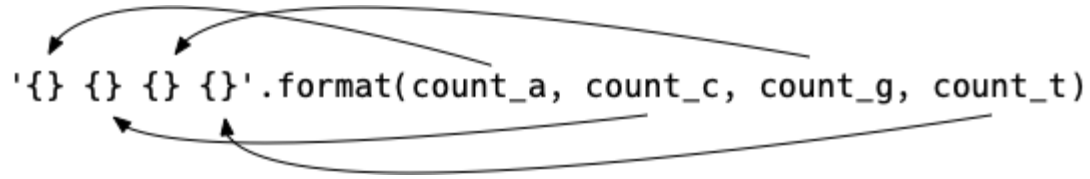


Figure 1-4. The str.format() function uses a template containing curly brackets to define placeholders that are filled in with the values of the arguments.

Every `{}` in the string template is a placeholder for some value that is provided as an argument to the function. When using this function, you need to ensure that you have the same number of placeholders as arguments. The arguments are inserted in the order in which they are provided. I'll have much more to say about the `str.format()` function later.

I'm not required to unpack the tuple returned by `count()` function. I can pass the entire tuple as the argument to the `str.format()` function if I *splat* it by adding an asterisk (`*`) to the front. This tells Python to expand the tuple into its values:

```
def main() -> None:
    args = get_args()
    counts = count(args.dna) ❶
    print('{} {} {} {}'.format(*counts)) ❷
```

❶ The `counts` variable is a 4-tuple of the integer base counts.

❷ The `*counts` syntax will expand the tuple into the four values needed by the format string; otherwise, the tuple would be interpreted as a single value.

Previously I use the `counts` variable only once, so I could skip the assignment and shrink this to one line:

```
def main() -> None:
    args = get_args()
    print('{} {} {} {}'.format(*count(args.dna))) ❶
```

❶ Pass the return from `count()` directly to the `str.format()` method.

The first solution is arguably easier to read and understand, and that is more important than writing clever code. Still, it's good to know about tuple unpacking and splatting variables as I'll use these in ideas in later programs.

Solution 3: Using str.count

The previous `count()` function turns out to be quite verbose. I can write the function using a single line of code using the `str.count()` method. This function will count the number of times one string is found inside another string. Let me show you in the REPL:

```
>>> seq = 'ACCGGGTTTT'
>>> seq.count('A')
1
>>> seq.count('C')
2
```

If a string is not found, it will report 0, making this safe to count all 4 nucleotides even when the input sequence is missing one or more base:

```
>>> 'AAA'.count('T')
0
```

Here is a new version of the `count()` function using this idea:

```
def count(dna: str) -> Tuple[int, int, int, int]: ❶
    """ Count bases in DNA """

    return (dna.count('A'), dna.count('C'), dna.count('G'),
dna.count('T')) ❷
```

❶ The signature is the same as before.

❷ Call the `dna.count()` method for each of the four bases.

This code is much more succinct, and I can use the same unit test to verify that it's correct. This is a key point: *functions should act like black boxes*. That is, I do not know or care what happens inside the box. Something goes in, an answer comes out, and I only really care that the answer is correct. I am free to change what happens inside the box so long as the contract to the outside — the parameters and return — stays the same.

Here's another way to create the output string in the `main()` function using Python's f-string syntax:

```
def main() -> None:
    args = get_args()
    count_a, count_c, count_g, count_t = count(args.dna) ❶
    print(f'{count_a} {count_c} {count_g} {count_t}') ❷
```

❶ Unpack the tuple into each of the four counts.

❷ Use an f-string to perform variable interpolation.

> **NOTE**
>
> It's called an *f-string* because the `f` precedes the quotes. I use the mnemonic *format* to remind me this is to format a string. Python also has a *raw* string that is preceded with an `r`, and I'll discuss that later. All strings in Python — bare, f-, or r-strings — can be enclosed in single- or double-quotes. It makes no difference.

With f-strings, the `{}` placeholders can perform *variable interpolation*, which is a 50-cent word that means turning a variable into its contents. Those curlies can even execute code. For instance, the `len()` function will return the length of a string and can be run inside the brackets:

```
>>> seq = 'ACGT'
>>> f'The sequence "{seq}" has {len(seq)} bases.'
'The sequence "ACGT" has 4 bases.'
```

I usually find f-strings easier to read over the equivalent code using `str.format()`. Which you choose is mostly a stylistic decision. I would recommend whichever makes your code more readable.

Solution 4: Using a Dictionary to Count all the Characters

So far I've discussed Python's strings, lists, and tuples. This next solution introduces dictionaries which are key/value stores. I'd like to show a version of the `count()` function that internally uses dictionaries so that I can hit on some important points to understand:

```
def count(dna: str) -> Tuple[int, int, int, int]: ❶
    """ Count bases in DNA """
```

```
    counts = {} ❷
    for base in dna: ❸
        if base not in counts: ❹
            counts[base] = 0 ❺
        counts[base] += 1 ❻

    return (counts.get('A', 0), ❼
            counts.get('C', 0),
            counts.get('G', 0),
            counts.get('T', 0))
```

❶ Internally I'll use a dictionary, but nothing changes about the function signature.

❷ Initialize an empty dictionary to hold the `counts`.

❸ Use a `for` loop to iterate through the sequence.

❹ Check if the base does not yet exist in the dictionary.

❺ Initialize the value for this base to 0.

❻ Increment the count for this base by 1.

❼ Use the `dict.get()` method to get each base's count or the default of 0.

Again, the contract for this function — the type signature — hasn't changed. It's still a string in and 4-tuple of integers out. Inside the function I'm going to use a dictionary that I'll initialize using the empty curly brackets:

```
>>> counts = {}
```

Or I could use the `dict()` function:

```
>>> counts = dict()
```

Neither is preferable. I can use the `type()` function to check that this is a dictionary:

```
>>> type(counts)
<class 'dict'>
```

The `isinstance()` function is another way to check the type of a variable:

```
>>> isinstance(counts, dict)
True
```

My goal is to create a dictionary that has each base as a *key* and the number of times it occurs as a *value*. For example, given the sequence `ACCGGGTTT`, I want `counts` to look like this:

```
>>> counts
{'A': 1, 'C': 2, 'G': 3, 'T': 4}
```

I can access any of the values using square brackets and a key name like so:

```
>>> counts['G']
3
```

Python will raise a `KeyError` exception if you attempt to access a dictionary key that doesn't exist:

```
>>> counts['N']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'N'
```

Use the `in` keyword to see if a key exists in a dictionary:

```
>>> 'N' in counts
False
```

```
>>> 'T' in counts
True
```

As I am iterating through each of the bases in the sequence, I need to see if a base exists in the `counts` dictionary. If it does not, I need to initialize it to 0. Then I can safely use the `+=` assignment to increment the count for a base by 1:

```
>>> seq = 'ACCGGGTTTT'
>>> counts = {}
>>> for base in seq:
...     if not base in counts:
...         counts[base] = 0
...     counts[base] += 1
...
>>> counts
{'A': 1, 'C': 2, 'G': 3, 'T': 4}
```

Finally, I want to return a 4-tuple of the counts for each of the bases. You might think this would work:

```
>>> counts['A'], counts['C'], counts['G'], counts['T']
(1, 2, 3, 4)
```

Ask yourself what would happen if one of the bases was missing from the sequence. Would this pass the unit test I wrote? Definitely not. It would fail on the very first test using an empty string because it would generate a `KeyError` exception.

The safe way to ask a dictionary for a value is to use the `dict.get()` method. If the key does not exist, then `None` will be returned:

```
>>> counts.get('T')
4
>>> counts.get('N')
```

PYTHON'S NONE VALUE

That second call looks like it does nothing because the REPL doesn't show `None`. I'll use the `type()` function to check the return. The `NoneType` is the type for the `None` value:

```
>>> type(counts.get('N'))
<class 'NoneType'>
```

While I can use the == operator to see if the return value is `None`:

```
>>> counts.get('N') == None
True
```

PEP8 recommends "Comparisons to singletons like `None` should always be done with `is` or `is not`, never the equality operators." The following is the prescribed method to check is a value is `None`:

```
>>> counts.get('N') is None
True
```

The `dict.get()` method accepts an optional second argument that is the default value to return when the key does not exist, so this is the safest way to return a 4-tuple of the base counts:

```
>>> counts.get('A', 0), counts.get('C', 0), counts.get('G', 0),
counts.get('T', 0)
(1, 2, 3, 4)
```

TIP

No matter what you write inside your `count()` function, ensure that it will pass the `test_count()` unit test.

Solution 5: Count Only the Desired Bases

The previous solution will count every character in the input sequence, but what if I only want to count the 4 nucleotides? In this solution, I will initialize a dictionary with values of 0 for the wanted bases. I'll need to also bring in `typing.Dict` to run this code:

```
def count(dna: str) -> Dict[str, int]: ❶
    """ Count bases in DNA """

    counts = {'A': 0, 'C': 0, 'G': 0, 'T': 0} ❷
    for base in dna: ❸
        if base in counts: ❹
            counts[base] += 1 ❺

    return counts ❻
```

❶ The signature now indicates I'll be returning a dictionary that has strings for the keys and integers for the values.

❷ Initialize the `counts` dictionary with the four bases as keys and values of 0.

❸ Iterate through the bases.

❹ Check if the base is found as a key in the `counts` dictionary.

❺ If so, increment the `counts` for this base by 1.

❻ Return the `counts` dictionary.

Since the `count()` function is now returning a dictionary rather than a tuple, the `test_count()` needs to change:

```
def test_count() -> None:
    """ Test count """

    assert count('') == {'A': 0, 'C': 0, 'G': 0, 'T': 0} ❶
```

```
    assert count('123XYZ') == {'A': 0, 'C': 0, 'G': 0, 'T': 0} ❷
    assert count('A') == {'A': 1, 'C': 0, 'G': 0, 'T': 0}
    assert count('C') == {'A': 0, 'C': 1, 'G': 0, 'T': 0}
    assert count('G') == {'A': 0, 'C': 0, 'G': 1, 'T': 0}
    assert count('T') == {'A': 0, 'C': 0, 'G': 0, 'T': 1}
    assert count('ACCGGGTTTT') == {'A': 1, 'C': 2, 'G': 3, 'T': 4}
```

❶ The returned dictionary will always have the keys A, C, G, and T. Even for the empty string, these keys will be present and set to 0.

❷ All the other tests have the same inputs, but now I check that the answer comes back as a dictionary.

When writing these tests, note that the order of the keys in the dictionaries is not important. The two dictionaries in the following code have the same content even though they were defined differently:

```
>>> counts1 = {'A': 1, 'C': 2, 'G': 3, 'T': 4}
>>> counts2 = {'T': 4, 'G': 3, 'C': 2, 'A': 1}
>>> counts1 == counts2
True
```

NOTE

I would point out that the `test_count()` function here not only tests the function to ensure it's correct, but it also acts as a piece of documentation. Reading these tests helps me clearly see the structure of the possible inputs and expected outputs from the function.

Here's how I need to change the `main()` function to use the returned dictionary:

```
def main() -> None:
    args = get_args()
    counts = count(args.dna) ❶
    print('{} {} {} {}'.format(counts['A'], counts['C'], counts['G'],
```

❷
```
                                counts['T']))
```

❶ `counts` is now a dictionary.

❷ Use the `str.format()` method to create the output using the values from the dictionary.

Solution 6: Using collections.defaultdict

I can rid my code of all the previous efforts to initialize dictionaries and check for keys and such by using the `defaultdict()` function from the `collections` module:

```
>>> from collections import defaultdict
```

When I use the `defaultdict()` function to create a new dictionary, I tell it the default type for the *values*. I no longer have to check for a key before using it because the `defaultdict` type will automatically create any key I reference using a representative value of the default type. For the case of counting the nucleotides, I want to use the `int` type:

```
>>> counts = defaultdict(int)
```

The default `int` value will be 0. Any reference to a non-existent key will cause it to be created with a value of 0:

```
>>> counts['A']
0
```

Which means I can instantiate and increment any base in one step:

```
>>> counts['C'] += 1
>>> counts
defaultdict(<class 'int'>, {'A': 0, 'C': 1})
```

Here is how I could rewrite the `count()` function using this idea:

```
def count(dna: str) -> Dict[str, int]:
    """ Count bases in DNA """

    counts: Dict[str, int] = defaultdict(int) ❶

    for base in dna:
        counts[base] += 1 ❷

    return counts
```

❶ The `counts` will be a defaultdict with integer values. The type annotation here is required by `mypy` so that it can be sure that the returned value is correct.

❷ I can safely increment the `counts` for this base.

The `test_count()` function looks quite different. I can see at a glance that the answers are very different from the previous versions:

```
def test_count() -> None:
    """ Test count """

    assert count('') == {} ❶
    assert count('123XYZ') == {'1': 1, '2': 1, '3': 1, 'X': 1, 'Y': 1,
'Z': 1} ❷
    assert count('A') == {'A': 1} ❸
    assert count('C') == {'C': 1}
    assert count('G') == {'G': 1}
    assert count('T') == {'T': 1}
    assert count('ACCGGGTTTT') == {'A': 1, 'C': 2, 'G': 3, 'T': 4}
```

❶ Given an empty string, an empty dictionary will be returned.

❷ Notice that every character in the string is a key in the dictionary.

❸ Only “A” is present with a count of 1.

Given the fact that the returned dictionary may not contain all the bases, the code in `main()` needs to use the `count.get()` method to retrieve each base's frequency:

```
def main() -> None:
    args = get_args()
    counts = count(args.dna) ❶
    print(counts.get('A', 0), counts.get('C', 0), counts.get('G', 0),
❷
          counts.get('T', 0))
```

❶ The `counts` will be a dictionary that may not contain all of the nucleotides.

❷ It's safest to use the `dict.get()` method with a default value of 0.

Solution 7: Using collections.Counter

I don't actually like the last three solutions all that much, but I needed to step through how to use a dictionary both manually and with `defaultdict()` so that you can appreciate the simplicity of using `collections.Counter()`:

```
>>> from collections import Counter
>>> Counter('ACCGGGTTT')
Counter({'G': 3, 'T': 3, 'C': 2, 'A': 1})
```

The best code is code you never write, and `Counter()` is a prepackaged function that will return a dictionary with the frequency of the items contained in the iterable you pass it. You might also hear this termed a *bag* or a *multiset*. Here the iterable is a string composed of characters, and so I get back the same dictionary as in the last two solutions but *having written no code*.

> *Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.*
>
> —Antoine de Saint-Exupéry

It's so simple that you could pretty much eschew the `count()` and `test_count()` functions and integrate it directly into your `main()`:

```
def main() -> None:
    args = get_args()
    counts = Counter(args.dna) ❶
    print(counts.get('A', 0), counts.get('C', 0), counts.get('G', 0),
❷
          counts.get('T', 0))
```

❶ The `counts` will be a dictionary containing the frequencies of the characters in `args.dna`.

❷ It is still safest to use `dict.get()` as I cannot be certain that all the bases are present.

I could argue that this code belongs in a `count()` function and keep the tests, but the `Counter()` function is already tested and has a well-defined interface. I think it makes more sense to use this function in-line.

Going Further

The solution only handle DNA provided as UPPERCASE TEXT. It's not uncommon to see DNA provided as lowercase letters. For instance, in plant genomics, it's common to use lowercase bases to denote regions of repetitive DNA. How could you handle both upper- and lower-case input?

If you choose to add the feature to your solution, do the following:

- Add a new input file that mixes case.
- Add a test to *tests/dna_test.py* that uses this new file and specifies the expected counts insensitive to case.
- Run the new test and ensure your program fails.

- Alter the program until it will pass the new test and all of the previous tests.

The solutions that used dictionaries to count all available characters would appear to be more flexible. That is, some of the tests only account for the bases A, C, G, and T, but if the input sequence were encoded using IUPAC codes to represent possible ambiguity in sequencing, then the program would have to be entirely rewritten. A program hard-coded to look only at the 4 nucleotides would also be useless on protein sequences that use a different alphabet. Consider writing a version of the program that will print two columns of output for each base that is found in the first column and the frequency in the second. Allow the user to sort ascending or descending by either column.

Review

This was kind of a monster chapter. The following chapters will be a bit shorter as I'll build upon many of the foundational ideas I've covered here. Here's what I covered:

- You can use the `new.py` program to create the basic structure of a Python program that accepts and validates command-line arguments using `argparse`.
- The `pytest` module will run all functions with names like *test*_ and will report the results of how many tests pass.
- Unit tests are for functions, and integration tests check if a program works as a whole.
- Programs like `pylint`, `flake8`, and `mypy` can find various kinds errors in your code. You can also have `pytest` automatically run tests to see if your code passes these checks.

- Complicated commands can be stored as a target in a *Makefile* and executed using the `make` command.
- There are many ways to count all the characters in a string. The `collections.Counter()` is perhaps the simplest method to create a dictionary of letter frequencies.
- You can annotate variables and functions with types and use `mypy` to ensure the types are used correctly.
- The Python REPL is an interactive tool for executing code examples and reading documentation.
- The Python community generally follows style guidelines such as PEP8. Tools like `yapf` and `black` can automatically format code according to these suggestions, and tools like `pylint` and `flake8` will report deviations from the guidelines.
- Python strings, lists, tuples, and dictionaries are very powerful data structures each with useful methods and copious documentation.
- You can create a custom, immutable, typed `class` derived from named tuples.

You may be wondering which is the "best" of the seven solutions. As with many things in life, it depends. Some programs are shorter to write and easier to understand but may fare poorly when confronting large data sets. In Chapter 2, I'll show you how to *benchmark* programs, pitting them against each other in multiple runs using large inputs to determine which perform the best.

---

1 The `/` operator here is a *binary* function (*bi* meaning *two* and *nary* meaning *number*) that is written as a punctuation mark sitting *between* its two arguments. I could also `import operator` to use `operator.truediv()` function, but the `/` syntax looks like the math we grew up writing in school and so is generally easy for most people to understand.

2 Boolean types are `True` or `False`, but many other data types are *truthy* or conversely *falsey*. The empty `str` (`""`) is falsey, so any non-empty string is truthy. The number `0` is falsey, so any non-zero value is truthy. An empty `list`, `set`, or `dict` is falsey, so any non-empty one of those is truthy.

3 You can also redirect output you don't want to see to /dev/null and it will be thrown away. For instance, in bash I can execute `some_command 2>/dev/null` to ignore all the errors from filehandle 2 which is called `STDERR` and pronounced *standard error.*

Chapter 2. Transcribing DNA into mRNA: Mutating Strings, Reading and Writing Files

To express the proteins necessary to sustain life, regions of DNA must be transcribed into a form of RNA called *messenger RNA* (mRNA). While there are many fascinating biochemical differences between DNA and RNA, for our purposes the only difference is that all the characters "T" representing the base *thymine* in a sequence of DNA need to be changed to the letter "U" for *uracil*. As described on the Rosalind RNA page, the program I'll show you how to write will accept a string of DNA like "ACGT" and print the transcribed RNA "ACGU." I can use Python's `str.replace()` function to accomplish this in one line:

```
>>> 'GATGGAACTTGACTACGTAAATT'.replace('T', 'U')
'GAUGGAACUUGACUACGUAAAUU'
```

You already saw in the Chapter 1 how to write a program to accept DNA from the command line or a file and print a result, so you won't be learning much if you do that again. I'll make this program more interesting by tackling a very common pattern found in bioinformatics. Namely, I'll show how to process one or more input files and place the results in an output directory. For instance, it's pretty common to get the results of a sequencing run back as a directory of files that need to be quality checked and filtered with the cleaned sequences going into some new directory for your analysis. Here the input files contain DNA sequences, one-per-line, and I'll write the RNA sequences into like-named files in an output directory.

In this chapter, you will learn:

- How to write a program to require one or more file inputs
- How to create directories
- How to read and write files
- How to modify strings

Getting Started

It might help to try running one of the solutions first to see how your program should work. Start by changing into the *02_rna* directory and copying the first solution to the program `rna.py`:

```
$ cd 02_rna
$ cp solution1_str_replace.py rna.py
```

Request the usage for the program using the `-h` flag:

```
$ ./rna.py -h
usage: rna.py [-h] [-o DIR] FILE [FILE ...] ❶

Transcribe DNA into RNA

positional arguments: ❷
  FILE                  Input DNA file

optional arguments:
  -h, --help            show this help message and exit
  -o DIR, --out_dir DIR
                        Output directory (default: out) ❸
```

❶ The arguments surrounded by square brackets (`[]`) are optional. The `[FILE ...]` syntax means that this argument can be repeated.

❷ The input FILE argument(s) will be positional.

❸ The output directory is an option and has the default value of *out*.

The goal of the program is to process files each containing one more sequences of DNA. Here is the first test input file:

```
$ cat tests/inputs/input1.txt
GATGGAACTTGACTACGTAAATT
```

Run the `rna.py` with this input file and note the output:

```
$ ./rna.py tests/inputs/input1.txt
Done, wrote 1 sequence in 1 file to directory "out".
```

Now there should be an *out* directory containing a file called *input1.txt*:

```
$ ls out/
input1.txt
```

The contents of that file should match the input DNA sequence but with all the *T*s changed to *U*s:

```
$ cat out/input1.txt
GAUGGAACUUGACUACGUAAAUU
```

You should run the program with multiple inputs and verify that you get multiple files in the output directory. Here I will use all the test input files with an output directory called *rna*. Notice how the summary text uses the correct singular/plurals for *sequence(s)* and *file(s)*:

```
$ ./rna.py --out_dir rna tests/inputs/*
Done, wrote 5 sequences in 3 files to directory "rna".
```

Defining the Program's Parameters

As you can see from the preceding usage, your program should accept the following parameters:

- One or more positional arguments which must be readable text files each containing strings of DNA to transcribe
- An optional `-o` or `--out_dir` argument that names an output directory to write the sequences of RNA. The default should be *out*.

You are free to write and structure your programs however you like (so long as they pass the tests), but I will always start a program using `new.py` and the structure I showed in the first chapter. The `--force` flag indicates that the existing `rna.py` should be overwritten:

```
$ new.py --force -p "Transcribe DNA to RNA" rna.py
Done, see new script "rna.py".
```

Defining an Optional Parameter

Modify the `get_args()` function to accept the parameters described in the previous section. To start, define the `out_dir` parameter. I would suggest you modify the `-a|--arg` option generated by `new.py` to this:

```
parser.add_argument('-o', ❶
                    '--out_dir', ❷
                    help='Output directory', ❸
                    metavar='DIR', ❹
                    type=str, ❺
                    default='out') ❻
```

❶ This is the short flag name. Short flags start with a single dash and are followed by a single character.

❷ This is the long flag name. Long flags start with two dashes and are followed by a more memorable string than the short flag. This will also be the name `argparse` will use to access the value.

❸

This will be incorporated into the usage statement to describe the argument.

❹ The metavar is a short description also shown in the usage.

❺ The default type of all arguments is `str` (string), so this is technically superfluous but still not a bad idea to document.

❻ The default value will be the string *out*. If you do not specify a `default` attribute when defining an argument, the default value will be `None`.

Defining One or More Required Positional Parameters

For the `FILE` value(s), I can modify the default `-f|--file` parameter to look like this:

```
parser.add_argument('file', ❶
                    help='Input DNA file(s)', ❷
                    metavar='FILE', ❸
                    nargs='+', ❹
                    type=argparse.FileType('rt')) ❺
```

❶ Remove the `-f` short flag and the two dashes from `--file` so that this becomes a *positional* argument called *file*. Optional parameters start with dashes, and positional ones do not.

❷ The help string indicates the argument should be one or more files containing DNA.

❸ This string is printed in the short usage to indicate the argument is a file.

❹ This indicates the number of arguments. The `+` indicates one or more values are required.

❺ This is the actual type that `argparse` will enforce. I am requiring any value to be a readable text (`rt`) file.

Using nargs to Define the Number of Arguments

I use `nargs` to describe the *number of arguments* to the program. There are three possible values you can use for this, as shown in Table 2-1.

Table 2-1. Possible values for `nargs`

Symbol	Meaning
`?`	Zero or one
`*`	Zero or more
`+`	One or more

When you use `+` with `nargs`, then `argparse` will provide the arguments as a list. Even if there is just one argument, you will get a list containing one element. You will never have an empty list because at least one argument is required.

Using argparse.FileType to Validate File Arguments

The `argparse.FileType()` is incredibly powerful, and using it can save you loads of time in validating file inputs. When you define a parameter with this type, `argparse` will print an error message and halt the execution of the program if any of the arguments is not a file. For instance, I would assume there is no file in your *02_dna* directory called *blargh*. Notice the result when I pass that value:

```
$ ./rna.py blargh
usage: rna.py [-h] [-o DIR] FILE [FILE ...]
rna.py: error: argument FILE: can't open 'blargh': [Errno 2] \
No such file or directory: 'blargh'
```

It's not obvious here, but the program never made it out of the `get_args()` function because `argparse` did the following:

- Detected that *blargh* is not a valid file
- Printed the short usage statement
- Printed a useful error message
- Exited the program with a non-zero value

This is how a well-written program ought to work, detecting and rejecting bad arguments as soon as possible and notifying the user of the problems. It's crucial in workflows that programs with fatal errors should fail loudly and signal the problem to the operating system with a non-zero exit value. All this happened without my writing anything more than a good description of the kind of argument I wanted. Again, the best code is code you never write.

Because I am using the file `type`, the elements of the list will not be strings representing the filenames but will instead be open filehandles. A *filehandle* is a mechanism to read and write the contents of a file. I used a filehandle in the last chapter when the DNA argument was a filename.

NOTE

The order that you define these parameters in your source code does not matter in this instance. You can define options before or after positional parameters. The order only matters when you have multiple positional arguments — the first parameter will be for the first positional argument, the second parameter for the second positional argument, and so forth.

Defining the Args Class

Finally, I need a way to define the `Args` class that will represent the arguments:

```
from typing import NamedTuple, List, TextIO ❶

class Args(NamedTuple):
    """ Command-line arguments """
    file: List[TextIO] ❷
    out_dir: str ❸
```

❶ I'll need two new imports from the `typing` module, `List` to describe a list and `TextIO` for an open filehandle.

❷ The `file` attribute will be a list of open filehandles.

❸ The `out_dir` attribute will be a string.

I can use this class to create the return value from `get_args()`. The following syntax uses positional notation such that the `file` is the first field and the `out_dir` is the second. When there are one or two fields, I will tend to use the positional notation:

```
return Args(args.file, args.out_dir)
```

Explicitly using the field names is safer and arguably easier to read, and it will become vital when I have more fields:

```
return Args(files=args.file, out_dir=args.out_dir)
```

Now I have all the code to define, document, and validate the inputs. Next I'll show how the rest of the program should work.

Outlining the Program Using Pseudocode

I'll sketch out the basics of the program's logic in the `main()` function using a mix of code and pseudocode to generally describe how to handle the input and output files. Whenever you get stuck writing a new program, this approach can help you see *what* needs to be done. Then you can figure out *how* to do it:

```
def main() -> None:
    args = get_args()

    if not os.path.isdir(args.out_dir): ❶
        os.makedirs(args.out_dir) ❷

    num_files, num_seqs = 0, 0 ❸
    for fh in args.files: ❹
        # open an output file in the output directory ❺
        # for each line/sequence from the input file:
            # write the transcribed sequence to the output file
            # update the number of sequences processed
        # update the number of files processed

    print('Done.') ❻
```

❶ The `os.path.isdir()` will report if the output directory exists.

❷ The `os.makedirs()` function will create a directory path.

❸ Initialize variables for the number of files and sequences written to use in the feedback you provide when the program exits.

❹ Use a `for` loop to iterate the list of filehandles in `args.files`. The iterator variable `fh` helps remind me of the type.

❺ Pseudocode describing the steps you need to do with each filehandle.

❻ Print a summary for the user to let them know what happened.

If you think you know how to finish the program, feel free to proceed at your own pace. Be sure to run `pytest` (or `make test`) to ensure your code is correct. If you need a little more guidance on how to read and write files, keep reading.

Iterating the Input Files

I'll tackle the pseudocode from the previous section. Remember that `args.files` is a `List[TextIO]` meaning that it is a list of filehandles. I can use a `for` loop to visit each element in any iterable such a list:

```
for fh in args.files:
```

I'd like to stress here that I choose an iterator variable called `fh` because each value is a filehandle. I sometimes see people who always use an iterator variable name like `i` or `x` with a `for` loop, but those are not descriptive variable names[1].

I'll concede that it's very common to use the variables like `n` or `i` (to mean *integer*) when iterating numbers like so:

```
for i in range(10):
```

I will sometimes use `x` and `xs` (pronounced *exes*) to stand for *one* and *many* of some generic value:

```
for x in xs:
```

Otherwise, it's very important to use variable names that accurately describe the thing they represent.

Creating the Output Filenames

Per the pseudocode, the first goal is to open an output file. For that, I need a filename that combines the name of the output directory with the *basename* of the input file. That is, if the input file is

dna/input1.txt and the output directory is *rna*, then the output file path should be *rna/input1.txt*.

The `os` module is used to interact with the operating system (like Windows, macOS, or Linux), and the `os.path` module has many handy functions I can use like `os.path.dirname()` to get the name of the directory from a file path and the `os.path.basename()` function to get the file's name (see Figure 2-1):

```
>>> import os
>>> os.path.dirname('./tests/inputs/input1.txt')
'./tests/inputs'
>>> os.path.basename('./tests/inputs/input1.txt')
'input1.txt'
```

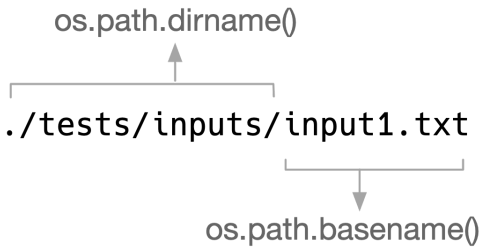


Figure 2-1. The os.path module contains useful functions like `dirname()` and `basename()` to extract parts from a file's path.

The new sequences will be written to an output file in `args.out_dir`. I suggest you use the `os.path.join()` function with the basename of the input file's basename to create the output filename, as shown in Figure 2-2. This will ensure that the output filename works both on Unix and Windows which use different path dividers, the slash (/) or backslash \, respectively.

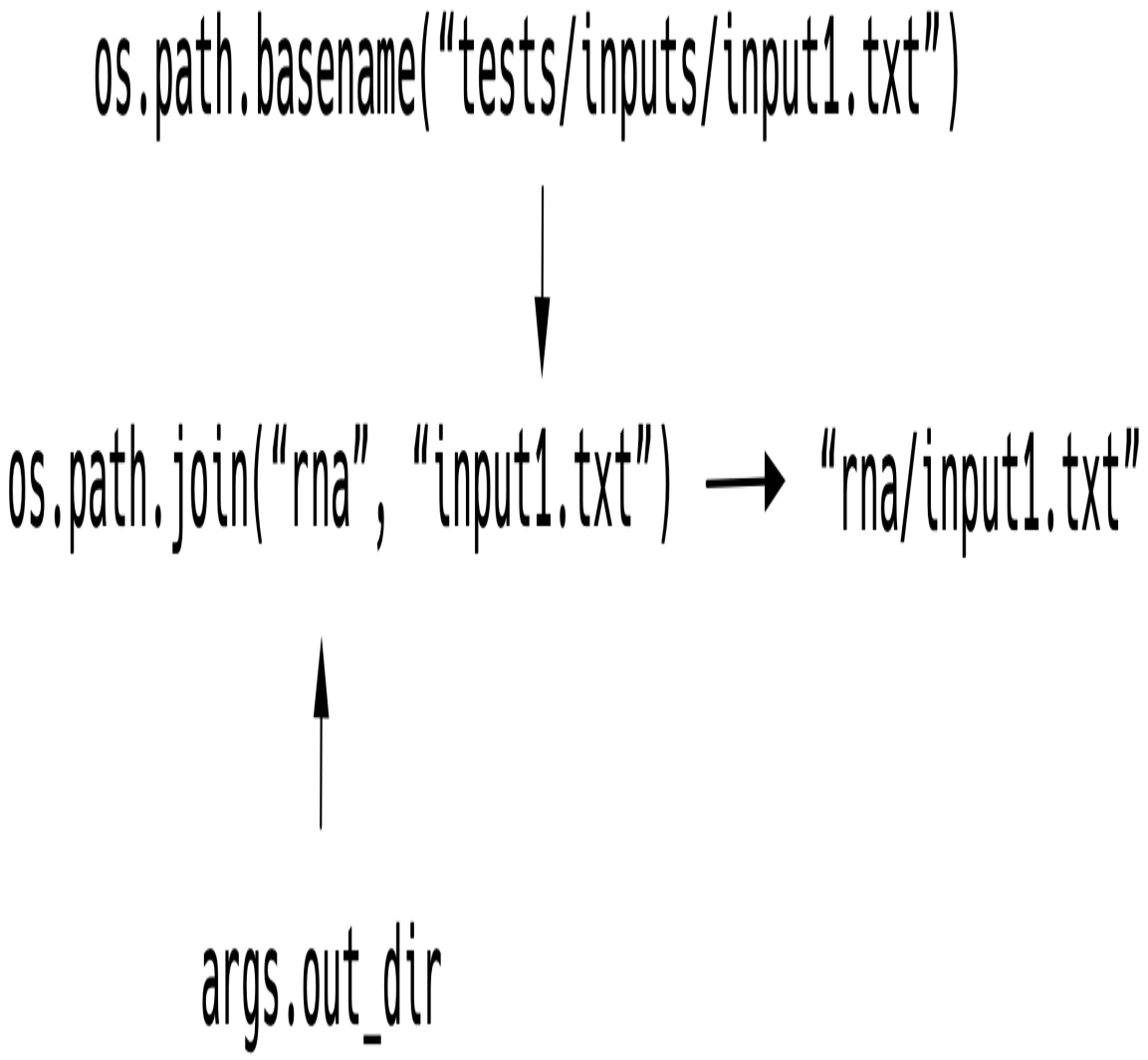


Figure 2-2. The os.path.join() will create the output path by combining the output directory with the basename of the input file.

You can get the file's path from the `fh.name` attribute of the filehandle:

```
for fh in args.files:
    out_file = os.path.join(args.out_dir, os.path.basename(fh.name))
    print(fh.name, '->', out_file)
```

Run your program to verify it looks like this:

```
$ ./rna.py tests/inputs/*
tests/inputs/input1.txt -> out/input1.txt
tests/inputs/input2.txt -> out/input2.txt
tests/inputs/input3.txt -> out/input3.txt
```

I'm taking baby steps towards what the program is supposed to do. It's very important to write just one or two lines of code and then run your program to see if it's correct. I often see students try to write many lines of code — whole programs, even — before they attempt to run them. That never ever works out well.

Opening the Output Files

Using this output filename, you need to `open()` the filehandle. I used this function in the first chapter to read DNA from an input file. By default, `open()` will only allow me to read a file, but I need to write a file. I can indicate that I want to open the file for writing by passing an optional second argument of the string `w` for *write*.

> **WARNING**
>
> When you open an existing file with a mode of `w`, the file will be *overwritten* meaning all data will be immediately and permanently lost. If needed, you can use the `os.path.isfile()` function to check if you're opening an existing file.

As shown in Table 2-2, you can also use the values `r` for *read* (the default) and `a` to *append* which allows you to open for writing more content at the end of an existing file.

Table 2-2. File writing modes

Mode	Meaning
`w`	Write
`r`	Read
`a`	Append

Table 2-3 shows that you can also read and write either text or raw bytes using the modes `t` and `b`, respectively:

Table 2-3. File content modes

Mode	Meaning
`t`	Text
`b`	Bytes

You can combine these like `rb` to *read bytes* and `wt` to *write text*, which is what I want here:

```
  for fh in args.files:
      out_file = os.path.join(args.out_dir, os.path.basename(fh.name))
      out_fh = open(out_file, 'wt') ❶
```

❶ Note that I named my variable `out_fh` to remind me this is the output filehandle.

Writing the Output Sequences

Looking at the pseudocode again, I have two levels of iterating — one for each filehandle of input, and then one for each line of DNA in

the filehandles. To read each line from an open filehandle, I can use another `for` loop:

```
for fh in args.files:
    for dna in fh:
```

The *input2.txt* file has two sequences, each ending with a newline:

```
$ cat tests/inputs/input2.txt
TTAGCCCAGACTAGGACTTT
AACTAGTCAAAGTACACC
```

To start, I'll show you how to print each sequence to the console, then I'll demonstrate how to use `print()` to write content to a filehandle. Chapter 1 mentions that the `print()` function will automatically append a newline (`\n` on Unix platforms and `\r\n` on Windows) unless I tell it not to. So as to avoid having two newlines from the following code, one from the sequence and one from `print()`, I can either use the `str.rstrip()` function to remove the newline from the sequence like this:

```
>>> fh = open('./tests/inputs/input2.txt')
>>> for dna in fh:
...     print(dna.rstrip()) ❶
...
TTAGCCCAGACTAGGACTTT
AACTAGTCAAAGTACACC
```

❶ Use `dna.rstrip()` to remove the trailing newline.

or use the `end` option to `print()`:

```
>>> fh = open('./tests/inputs/input2.txt')
>>> for dna in fh:
...     print(dna, end='') ❶
...
TTAGCCCAGACTAGGACTTT
AACTAGTCAAAGTACACC
```

❶ Use the empty string at the end instead of a newline.

The goal is to transcribe each DNA sequence to RNA and write the result to the `out_fh`. In the introduction of this chapter, I suggested you could use the `str.replace()` function. If you read `help(str.replace)` in the REPL, you'll see that it will "Return a copy with all occurrences of substring old replaced by new":

```
>>> dna = 'ACTG'
>>> dna.replace('T', 'U')
'ACUG'
```

There are other ways to change the *T*s to *U*s that I will explore on the flip side. I'd like to point out that strings in Python are immutable, meaning they cannot be changed in-place. That is, I could check to see if the letter *T* is in the DNA and then use the `str.index()` function to find the location and try to overwrite it with the letter *U*. This will raise an exception:

```
>>> dna = 'ACTG'
>>> if 'T' in dna:
...     dna[dna.index('T')] = 'U'
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: 'str' object does not support item assignment
```

Instead, I'll use `str.replace()` to create a new string. Note that this never alters the original string and only ever returns a new string:

```
>>> dna.replace('T', 'U')
'ACUG'
>>> dna
'ACTG'
```

I need to write this new string into the `out_fh` output filehandle. I have two options. First, I can use the `print()` function's `file` option

to describe *where* to print the string. Consult the `help(print)` documentation:

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

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current
sys.stdout. ❶
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

❶ This is the option I need to print the string to the open filehandle.

I need to use the `out_fh` filehandle as the `file` argument. I want to point out that the default `file` value is `sys.stdout`. On the command line, `STDOUT` (pronounced *standard out*) is the standard place for program output to appear which is usually the console.

Another option is to use the `out_fh.write()` method of the filehandle itself, but note that this function *does not* append a newline. It's up to you to decide when to add newlines. In the case of reading these sequences that are terminated with newlines, they are not needed.

Printing the Status Report

I almost always like to print something when my programs have finished running so I at least know they got to the end. It may be something as simple as "Done!" Here, though, I'd like to know how many sequences in how many files were processed. I also want to know where I can find the output, something that's especially helpful if I forget the name of the default output directory.

The tests expect that you will use proper grammar[2] to describe the numbers, for example, *1 sequence* and *1 file*:

```
$ ./rna.py tests/inputs/input1.txt
Done, wrote 1 sequence in 1 file to directory "out".
```

Or *3 sequences* and *2 files*:

```
$ ./rna.py --out_dir rna tests/inputs/input[12].txt
Done, wrote 3 sequences in 2 files to directory "rna".
```

Using the Test Suite

You can run `pytest -xv` to run *tests/rna_test.py*. A passing test suite looks like this:

```
$ pytest -xv
======================== test session starts ========================
...

tests/rna_test.py::test_exists PASSED                          [ 14%] ❶
tests/rna_test.py::test_usage PASSED                           [ 28%] ❷
tests/rna_test.py::test_no_args PASSED                         [ 42%] ❸
tests/rna_test.py::test_bad_file PASSED                        [ 57%] ❹
tests/rna_test.py::test_good_input1 PASSED                     [ 71%] ❺
tests/rna_test.py::test_good_input2 PASSED                     [ 85%]
tests/rna_test.py::test_good_multiple_inputs PASSED            [100%]

======================== 7 passed in 0.37s ========================
```

❶ The `rna.py` program exists.

❷ The program prints a usage statement when requested.

❸ The program exits with an error when given no arguments.

❹ The program prints an error message when given a bad file argument.

❺ The next tests all verify the program works properly given good inputs.

Generally speaking, I usually write tests that first try to break a program before giving it good input. For instance, I want the program to fail when given no files or when given non-existent files. Just as the best detectives can think like criminals, I imagine all the worst things that can happen to my programs and test that they behave predictably under those circumstances.

The first two tests are exactly as from Chapter 1. The third test is the `test_no_args()` function:

```
def test_no_args():
    """ Dies on no args """

    retval, out = getstatusoutput(RUN) ❶
    assert retval != 0 ❷
    assert out.lower().startswith('usage:') ❸
```

❶ Run the program with no arguments and capture the return value and the standard out.

❷ Check that the return value is not zero.

❸ Make sure the output looks like a usage statement.

When you write tests for your programs, it's important to check that the program reports a non-zero exit status for errors. This program requires at least one input file, so running with no arguments should produce a fatal error.

Contrast this with the return value from `test_usage()`:

```
def test_usage():
    """ Usage """

    for flag in ['-h', '--help']:
        retval, out = getstatusoutput(f'{RUN} {flag}') ❶
        assert retval == 0 ❷
        assert out.lower().startswith('usage:') ❸
```

❶ Run the program with both the short and long flags for help.

❷ Ensure that this does not result in an error value.

❸ Check that the lowercase output starts with *usage:*.

Likewise, when I pass a non-existent file, I expect a non-zero exit value along with the usage and the error message. Note that the error specifically mentions the offending value, here the bad filename. You should strive to create feedback that lets the user know exactly what the problem is and how to fix it:

```
def test_bad_file():
    """ Die on missing input """

    bad = random_filename() ❶
    retval, out = getstatusoutput(f'{RUN} {bad}') ❷
    assert retval != 0 ❸
    assert re.match('usage:', out, re.IGNORECASE) ❹
    assert re.search(f"No such file or directory: '{bad}'", out) ❺
```

❶ This is a function I wrote to generate a string of random characters.

❷ Run the program with this non-existent file.

❸ Make sure the exit value is not 0.

❹ Use a regular expression (*regex*) to look for the usage in the output.

❺ Use another regex to look for the error message describing the bad input filename.

I haven't introduced regular expressions yet, but they will become central to solutions I write later. To see why they are useful, look at

the output from the program when run with a bad file input:

```
$ ./rna.py dKej82
usage: rna.py [-h] [-o DIR] FILE [FILE ...]
rna.py: error: argument FILE: can't open 'dKej82': \
[Errno 2] No such file or directory: 'dKej82'
```

Using the `re.match()` function, I am looking for a pattern of text starting at the beginning of the `out` text. Using the `re.search()` function, I am looking for another pattern that occurs somewhere inside the `out` text. I'll have much more to say about regexes later. For now it's enough to point out that they are very useful.

I'll show one last test that verifies the program runs correctly when provided good input. There are many ways to write such a test, so don't get the impression this is canon:

```
def test_good_input1():
    """ Runs on good input """

    out_dir = 'out' ❶
    try: ❷
        if os.path.isdir(out_dir): ❸
            shutil.rmtree(out_dir) ❹

        retval, out = getstatusoutput(f'{RUN} {INPUT1}') ❺
        assert retval == 0
        assert out == 'Done, wrote 1 sequence in 1 file to directory
"out".'
        assert os.path.isdir(out_dir) ❻
        out_file = os.path.join(out_dir, 'input1.txt')
        assert os.path.isfile(out_file) ❼
        assert open(out_file).read().rstrip() ==
'GAUGGAACUUGACUACGUAAAUU' ❽

    finally: ❾
        if os.path.isdir(out_dir): ❿
            shutil.rmtree(out_dir)
```

❶ This is the default output directory name.

❷ The `try`/`finally` blocks help to ensure cleanup when tests fail.

❸ See if the output directory has been leftover from a previous run.

❹ Use the `shutil.rmtree()` function to remove the directory and its contents.

❺ Run the program with a known good input file.

❻ Make sure the expected output directory was created.

❼ Make sure the expected output file was created.

❽ Make sure the contents of the output file are correct.

❾ Even if something fails in the `try` block, this `finally` block will be run.

❿ Cleanup the testing environment.

I want to stress how important it is to check every aspect of what your program is supposed to do. Here, the program should process some number of input files, create an output directory, and then place the processed data into files in the output directory. I'm testing every one of those requirements using known input to verify that the expected output is created.

There are a couple of other tests I won't cover here as they are similar what I've already shown, but I would encourage you to read the entire *tests/rna_test.py* program. The first input file has one sequence. The second input file has two sequences, and I use that to test that two sequences are written to the output file. The third input file has two very long sequences. By using these inputs individually and together, I try to test every aspect of my program that I can imagine.

You can run all these tests using `pytest`. I also urge you to use `pylint`, `flake8`, and `mypy` to check your program. The `make test` shortcut can do this for you as it will execute `pytest` with the additional arguments to run those tools. Your goal should be a completely clean test suite.

> **NOTE**
>
> You may find that `pylint` will complain about variable names like `fh` being too short or not being *snake_case* where lowercase words are joined with underscores. I have included a *pylintrc* configuration file in the top level of the GitHub repository. Copy this to the file *.pylintrc* in your home directory to silence these errors.

You should have enough information and tests to help you finish this program. You'll get the most benefit from this book if you try to write working programs on your own before you look at my solutions. Once you have one working version, try to find other ways to solve it. If you know about regular expressions, that's a great solution. If you don't, I will demonstrate a version that uses them.

Solutions

The following two solutions differ only in how I substitute the *T*s for *U*s. The first uses the `str.replace()` method, and the second introduces regular expressions and uses the Python `re.sub()` function.

Solution 1: Using str.replace

Here is the entirety of one solution that uses the `str.replace()` method I discussed in the introduction to this chapter:

```
#!/usr/bin/env python3
""" Transcribe DNA into RNA """

import argparse
import os
from typing import NamedTuple, List, TextIO


class Args(NamedTuple):
    """ Command-line arguments """
    file: List[TextIO]
    out_dir: str


# --------------------------------------------------
def get_args() -> Args:
    """ Get command-line arguments """

    parser = argparse.ArgumentParser(
        description='Transcribe DNA into RNA',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file',
                        help='Input DNA file',
                        metavar='FILE',
                        type=argparse.FileType('rt'),
                        nargs='+')

    parser.add_argument('-o',
                        '--out_dir',
                        help='Output directory',
                        metavar='DIR',
                        type=str,
                        default='out')

    args = parser.parse_args()

    return Args(args.file, args.out_dir)


# --------------------------------------------------
def main() -> None:
    """ Make a jazz noise here """

    args = get_args()
```

```
    if not os.path.isdir(args.out_dir):
        os.makedirs(args.out_dir)

    num_files, num_seqs = 0, 0 ❶
    for fh in args.files: ❷
        num_files += 1 ❸
        out_file = os.path.join(args.out_dir,
os.path.basename(fh.name))
        out_fh = open(out_file, 'wt') ❹

        for dna in fh: ❺
            num_seqs += 1 ❻
            out_fh.write(dna.replace('T', 'U')) ❼

        out_fh.close() ❽

    print(f'Done, wrote {num_seqs} sequence{"" if num_seqs == 1 else
"s"} '
          f'in {num_files} file{"" if num_files == 1 else "s"} '
          f'to directory "{args.out_dir}".') ❾


# --------------------------------------------------
if __name__ == '__main__':
    main()
```

❶ Initialize the counters for files and sequences.

❷ Iterate the filehandles.

❸ Increment the counter for files.

❹ Open the output file for this input file.

❺ Iterate the sequences in the input file.

❻ Increment the counter for sequences.

❼ Write the transcribe sequence to the output file.

❽ Close the output filehandle.

❾ Print the status. Note that I'm relying on Python's implicit concatenation of adjacent strings to create one output string.

Solution 2: Using re.sub

I suggested you might explore how to use regular expressions to solve this. Regexes are a language for describing patterns of text. They have been around for decades, long before Python was even invented. Though they may seem somewhat daunting at first, regexes are well worth the effort to learn[3].

To use regular expressions in Python, I must import the `re` module:

```
>>> import re
```

You saw earlier that I used the `re.search()` function to look for a pattern of text inside another string. For this program, the pattern I am looking for is the letter *T* which I can write as a literal string:

```
>>> re.search('T', 'ACGT') ❶
<re.Match object; span=(3, 4), match='T'> ❷
```

❶ Search for the pattern "T" inside the string "ACGT."

❷ Because the *T* was found, the return value is a match object showing the location of the found pattern. A failed search would return `None`.

The `span=(3, 4)` reports the start and stop indexes of *T*. I can use these positions to extract the substring using a slice:

```
>>> 'ACGT'[3:4]
'T'
```

Instead of just finding the *T*, I want to replace the string "T" with "U." As shown in Figure 2-3, the `re.sub()` (for *substitute*) function will do this.

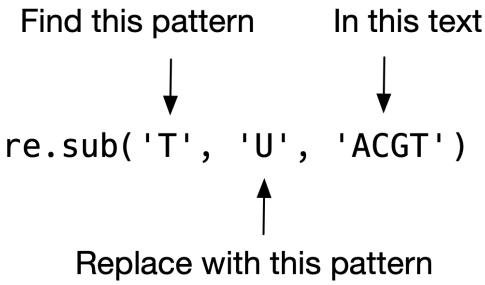


Figure 2-3. The `re.sub()` function will return a new string where all instances of a pattern have been replaced with a new string.

The result is a new string where the *T*s have all be replaced with *U*s:

```
>>> re.sub('T', 'U', 'ACGT') ❶
'ACGU' ❷
```

❶ Replace every "T" with "U" in the string "ACGT."

❷ The result is a new string with the substitutions.

To use this version, I can modify the inner `for` loop as shown. Note that I have chosen to use the `str.strip()` method to remove the newline terminating the input DNA string because `print()` will add a newline:

```
for dna in fh:
    num_seqs += 1
    print(re.sub('T', 'U', dna.rstrip()), file=out_fh) ❶
```

❶ Remove the newline from `dna`, substitute all the *T*s for *U*s, and print the resulting string to the output filehandle.

Benchmarking

You might be curious to know which solution is faster. Comparing the relative runtimes of programs is called *benchmarking*, and I'll show you a simple way to compare these two solutions using some basic `bash` commands. I'll use the *./tests/inputs/input3.txt* file as it is the largest test file. I can write a `for` loop in `bash` with almost the same syntax as Python. Note that I am using newlines in this command to make it more readable, and `bash` notes the line continuation with `>`. You can substitute semicolons (`;`) to write this on one line:

```
$ for py in ./solution*
> do echo $py && time $py ./tests/inputs/input3.txt
> done
./solution1_str_replace.py
Done, wrote 2 sequences in 1 file to directory "out".

real    0m1.539s
user    0m0.046s
sys         0m0.036s
./solution2_re_sub.py
Done, wrote 2 sequences in 1 file to directory "out".

real    0m0.179s
user    0m0.035s
sys         0m0.013s
```

It would appear the second solution using regular expressions is faster, but I really don't have enough data. I need a more substantial input file. In the *02_rna* directory, you'll find a program called `genseq.py` I wrote that will generate 1,000 sequences of 1,000,000 bases in a file called *seq.txt*. You can, of course, modify the parameters:

```
$ ./genseq.py --help
usage: genseq.py [-h] [-l int] [-n int] [-o FILE]

Generate long sequence

optional arguments:
  -h, --help            show this help message and exit
  -l int, --len int     Sequence length (default: 1000000)
```

```
  -n int, --num int     Number of sequences (default: 100)
  -o FILE, --outfile FILE
                        Output file (default: seq.txt)
```

The file *seq.txt* that is generated using the defaults is about 95 MB0. Here's how the programs do with a much more realistic input file:

```
$ for py in ./solution*; do echo $py && time $py seq.txt; done
./solution1_str_replace.py
Done, wrote 100 sequences in 1 file to directory "out".

real    0m0.456s
user    0m0.372s
sys     0m0.064s
./solution2_re_sub.py
Done, wrote 100 sequences in 1 file to directory "out".

real    0m3.100s
user    0m2.700s
sys     0m0.385s
```

It would now appear that the first solution is clearly faster. For what it's worth, I came up with several other solutions all of which fared much, much worse than these two solutions. I thought I was creating more and more clever solutions which would ultimately lead to the best performance. My pride was sorely abused when what I thought was my best program turned out to be orders of magnitude slower than these two first two. When you have assumptions, you should, as the saying goes, "Trust, but verify."

Going Further

Modify your program to print the length of the sequences to the output file rather than the transcribed RNA. Have the final status report the maximum, minimum, and average sequence lengths.

Review

- The `argparse.FileType` option will validate file arguments.
- The `nargs` option to `argparse` allows you to define the number of valid arguments for parameter.
- The `os.path.isdir()` function can detect if a directory exists.
- The `os.makedirs()` function will create a directory structure.
- The `open()` function by default allows only reading files. The `w` option must be used to write to the filehandle, and the `a` option is for appending values to an existing file.
- File handles can be opened with the `t` option for *text* (the default) or `b` for *bytes* such as when reading image files.
- Strings are immutable there are many methods to alter strings into new strings including `str.replace()` and `re.sub()`.

---

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

2 Sorry, but I can't stop being the English major.

3 *Mastering Regular Expressions* by Jeffrey Friedl (O'Reilly, 2006) is one of the best books I've found.

Chapter 3. Complementing a Strand of DNA: String Manipulation

The Rosalind reverse complement challenge explains that the bases of DNA form pairs of A-T and G-C. Additionally, DNA has directionality and is usually read from the 5'-end (*five-prime end*) towards the 3'-end (*three-prime end*). As shown in Figure 3-1, the complement of the DNA string “AAAACCCGGT” is “TTTTGGGCCA.” I then reverse this string (reading from the 3'-end) to get “ACCGGGTTTT” as the reverse complement.

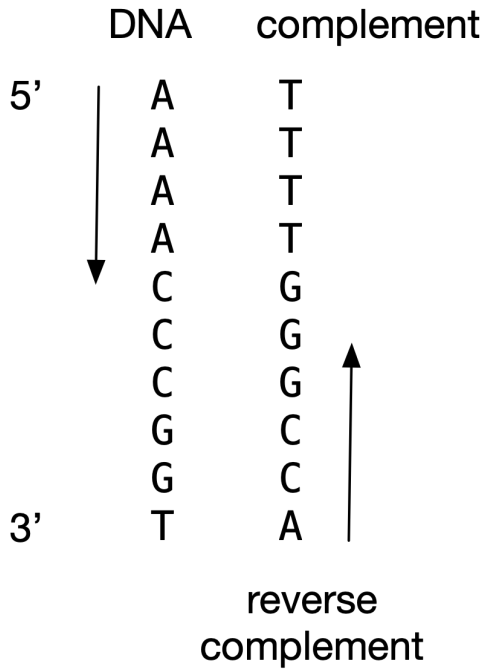


Figure 3-1. The reverse complement of DNA is the complement read from the opposite direction.

Although you can find many existing tools to generate the reverse complement of DNA — and I’ll drop a spoiler alert that the final

solution will use a function from the Biopython library — the point of writing our own algorithm is to explore Python. In this chapter, you will learn:

- How to create a decision tree using `if`/`elif` statements
- How to use a dictionary as a lookup table
- How to dynamically generate a list or a string
- How to use the `reversed()` function which is an example of an iterator
- How Python treats strings and lists similarly
- How to use a list comprehension to generate a list
- How to use `str.maketrans()` and `str.translate()` to transform a string
- How to use the Biopython's `Bio.Seq` module
- That the real treasure is the friends you make along the way

Getting Started

The code and tests for this program are in the *03_revc* directory. To get a feel for how the program will work, change into that directory and copy the first solution to a program called `revc.py`:

```
$ cd 03_revc
$ cp solution1_for_loop.py revc.py
```

Run the program with `--help` to read the usage:

```
$ ./revc.py --help
usage: revc.py [-h] DNA

Print the reverse complement of DNA
```

```
positional arguments:
  DNA         Input sequence or file

optional arguments:
  -h, --help  show this help message and exit
```

You see the program wants “DNA” and will print the reverse complement, so I’ll give it a string:

```
$ ./revc.py AAAACCCGGT
ACCGGGTTTT
```

As the help indicates, the program will also accept a file as input. The first test input has the same string:

```
$ cat tests/inputs/input1.txt
AAAACCCGGT
```

So the output should be the same:

```
$ ./revc.py tests/inputs/input1.txt
ACCGGGTTTT
```

Note that I want to make the specs for the program just a little harder as the tests will pass both upper- and lowercase input. The output should respect the case of the input:

```
$ ./revc.py aaaaCCCGGT
ACCGGGtttt
```

Run `pytest` or `make test` to see what kinds of tests the program should pass. When you’re satisfied you have a feel for what the program should do, start anew:

```
$ new.py -f -p 'Print the reverse complement of DNA' revc.py
Done, see new script "revc.py".
```

Edit the `get_args()` function until the program will print the preceding usage. Then modify your program so that it will echo back input either from the command line or from an input file:

```
$ ./revc.py AAAACCCGGT
AAAACCCGGT
$ ./revc.py tests/inputs/input1.txt
AAAACCCGGT
```

If you run the test suite, you should find your program passes the first three tests:

```
$ pytest -xv
============================= test session starts
==============================
...

tests/revc_test.py::test_exists PASSED
[ 14%]
tests/revc_test.py::test_usage PASSED
[ 28%]
tests/revc_test.py::test_no_args PASSED
[ 42%]
tests/revc_test.py::test_uppercase FAILED
[ 57%]

=================================== FAILURES
===================================
_________________________________ test_uppercase
__________________________________

    def test_uppercase():
        """ Runs on uppercase input """

        rv, out = getstatusoutput(f'{RUN} AAAACCCGGT')
        assert rv == 0
>       assert out == 'ACCGGGTTTT'
E       AssertionError: assert 'AAAACCCGGT' == 'ACCGGGTTTT'
E         - ACCGGGTTTT
E         + AAAACCCGGT

tests/revc_test.py:47: AssertionError
```

```
=========================== short test summary info
============================
FAILED tests/revc_test.py::test_uppercase - AssertionError: assert
'AAAACCCGG...
!!!!!!!!!!!!!!!!!!!!!!!! stopping after 1 failures
!!!!!!!!!!!!!!!!!!!!!!!!
======================== 1 failed, 3 passed in 0.33s
=========================
```

The program is being passed the input string “AAAACCCGGT,” and the test expects it to print “ACCGGGTTTT.” Since the program is echoing the input, this test fails. If you think you know how to write a program to satisfy these tests, have at it. If not, I’ll show you how to create the reverse complement of DNA, starting with a simple approach and working into more elegant solutions.

Iterating over a Reversed String

The order of reversing and then complementing the DNA or vice verse doesn’t matter as you will get the same answer. I’ll start with how you can reverse a string. It so happens that Python has a built-in `reversed()` function, so let’s try that:

```
>>> dna = 'AAAACCCGGT'
>>> reversed(dna)
<reversed object at 0x7ffc4c9013a0>
```

Surprise! You were probably expecting to see the string “TGGCCCAAAA.” If you read `help(reversed)` in the REPL, you’ll see that this function will “Return a reverse iterator over the values of the given sequence.”

What is an *iterator*? The Functional Programming HOWTO is a good primer on iterator which is describes as an “an object representing a stream of data.” I’ve mentioned iterables are anything that Python can travel over from end to end such as a string or a list. An iterator is something that will generate values until it is exhausted. Just as I can start with the first character of a string (or the first element of a

list or the first line of a file) and read until the end of the string (or list or file), an iterator can be iterated from the first value it produces until it finishes.

In this case, the `reversed()` function is returning a promise to produce the reversed values as soon as it appears that you really need it. This is an example of a *lazy* function because it waits until forced to do any work. One way to force `reversed()` to produce the values is to use a function that will consume the values. For instance, if the only goal were just to reverse the string, then I could use the `str.join()` function. I always feel the syntax is backward on this function, but normally you will use the `str.join()` method invoked on a string literal that is the element used to join the sequence.

Here is another way to generate the reversed DNA by using the empty string (`''`) to join the list elements back into a new string:

```
>>> ''.join(reversed(dna)) ❶
'TGGCCCAAAA'
```

❶ Use the empty string to join the reversed characters of the DNA.

In the REPL, I can coerce the lazy values from `reversed()` by using the `list()` function:

```
>>> list(reversed(dna))
['T', 'G', 'G', 'C', 'C', 'C', 'A', 'A', 'A', 'A']
```

Wait, what happened? The `dna` variable is a string, but I got back a list — and not just because I used the `list()` function. The documentation for `reversed()` shows that the function takes a *sequence*, which means basically any data structure or function that returns one thing followed by another. In a list or iterator context, Python treats strings as lists of characters:

```
>>> list(dna)
['A', 'A', 'A', 'A', 'C', 'C', 'C', 'G', 'G', 'T']
```

A longer way to build up the reversed DNA is to use a `for` loop to iterate over the reversed bases and append them to a string. First, I'll declare a `rev` variable, and I'll append each base in reverse order using the `+=` operator:

```
>>> rev = '' ❶
>>> for base in reversed(dna): ❷
...     rev += base ❸
...
>>> rev
'TGGCCCAAAA'
```

❶ Initialize the `rev` variable with the empty string.

❷ Iterate through the reversed bases of DNA.

❸ Append the current base to the `rev` variable.

Since I still need to complement the bases, I'm not quite done.

Creating a Decision Tree

There are a total of eight complements: A to T and G to C, both upper- and lowercase, and then vice versa. I also need to handle the case of a character *not* being A, C, G, or T. I can use `if`/`elif` statements to create a decision tree. I'll change my variable to `revc` since it's now the reverse complement, and I'll figure out the correct complement for each base:

```
revc = '' ❶
for base in reversed(dna): ❷
    if base == 'A': ❸
        revc += 'T' ❹
    elif base == 'T':
```

```
            revc += 'A'
        elif base == 'G':
            revc += 'C'
        elif base == 'C':
            revc += 'G'
        elif base == 'a':
            revc += 't'
        elif base == 't':
            revc += 'a'
        elif base == 'g':
            revc += 'c'
        elif base == 'c':
            revc += 'g'
        else: ❺
            revc += base
```

❶ Initialize a variable to hold the reverse complement string.

❷ Iterate through the reversed bases in the DNA.

❸ Test each upper- and lowercase base.

❹ Append the complementing base to the variable.

❺ If the base doesn't match any of these tests, use the base as-is.

If you inspect the `revc` variable, it appears to be correct:

```
>>> revc
'ACCGGGTTTT'
```

You should be able to incorporate these ideas into a program that will pass the test suite. To understand what exactly is expected of your program, take a look at the *tests/revc_test.py* file. After you pass the `test_uppercase()` function, see what is expected by the `test_lowercase()`:

```
def test_lowercase():
    """ Runs on lowercase input """
```

```
    rv, out = getstatusoutput(f'{RUN} aaaaCCCGGT') ❶
    assert rv == 0 ❷
    assert out == 'ACCGGGtttt' ❸
```

❶ Run the program using lower- and uppercase DNA.

❷ The exit value should be 0.

❸ The output from the program should be the indicated string.

The next tests pass files as input:

```
def test_input1():
    """ Runs on file input """

    file, expected = TEST1 ❶
    rv, out = getstatusoutput(f'{RUN} {file}') ❷
    assert rv == 0 ❸
    assert out == open(expected).read().rstrip() ❹
```

❶ The `TEST1` tuple is a file of input and a file of expected output.

❷ Run the program with the filename.

❸ Make sure the exit value is 0.

❹ Open and read the expected file and compare that to the output.

It's equally as important to read and understand the testing code as it is to learn how to write the solutions. When you write your own programs, you may find you can copy many of the ideas from these tests and save yourself lots of time.

Refactoring

While the algorithm in the preceding section will produce the correct answer, it is not an elegant solution. Still, it's a place to start that passes the tests. Now that you perhaps have a better idea of the challenge, it's time to refactor the program. You'll see that some of the solutions I present are 1-2 lines of code, so I challenge you to explore other options:

- Use a dictionary as a lookup table instead of the `if`/`elif` chain
- Rewrite the `for` loop as a list comprehension
- Use the `str.translate()` method to complement the bases
- Create a `Bio.Seq` object and find the method that will do this for you

There's really no hurry to read ahead. Take your time to try other solutions. I haven't introduced all these ideas yet, so I encourage you to research any unknowns and see if you can figure them out on your own.

I remember one of my teachers in music school shared this quote with me:

> *Then said a teacher, Speak to us of Teaching.*
>
> *And he said:*
>
> *No man can reveal to you aught but that which already lies half asleep in the dawning of your knowledge.*
>
> *The teacher who walks in the shadow of the temple, among his followers, gives not of his wisdom but rather of his faith and his lovingness.*
>
> *If he is indeed wise he does not bid you enter the house of his wisdom, but rather leads you to the threshold of your own mind.*
>
> —Kahlil Gibran

Solutions

All of the solutions share the same `get_args()` function as follows:

```
class Args(NamedTuple): ❶
    """ Command-line arguments """
    dna: str


# --------------------------------------------------
def get_args() -> Args:
    """ Get command-line arguments """

    parser = argparse.ArgumentParser(
        description='Print the reverse complement of DNA',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('dna', metavar='DNA', help='Input sequence or
file')

    args = parser.parse_args()

    if os.path.isfile(args.dna): ❷
        args.dna = open(args.dna).read().rstrip()

    return Args(args.dna) ❸
```

❶ The only argument to the program is a string of DNA.

❷ Handle the case when reading a file input.

❸ Return an `Args` object in compliance with the function signature.

Solution 1: Using a for Loop and Decision Tree

Here is my first solution using the `if`/`else` decision tree:

```
def main() -> None:
    args = get_args()
    revc = '' ❶
```

```
for base in reversed(args.dna): ❷
    if base == 'A': ❸
        revc += 'T'
    elif base == 'T':
        revc += 'A'
    elif base == 'G':
        revc += 'C'
    elif base == 'C':
        revc += 'G'
    elif base == 'a':
        revc += 't'
    elif base == 't':
        revc += 'a'
    elif base == 'g':
        revc += 'c'
    elif base == 'c':
        revc += 'g'
    else:
        revc += base

print(revc) ❹
```

❶ Initialize a variable to hold the reverse complement.

❷ Iterate through the reversed bases of the DNA argument.

❸ Create an `if`/`elif` decision tree to determine each base's complement.

❹ Print the result.

Solution 2: Using a Dictionary Lookup

I mentioned that the `if`/`else` chain is something you should try to replace. That is 18 lines of code (LOC) that could be represented more easily using a dictionary lookup:

```
>>> trans = {
...     'A': 'T', 'C': 'G', 'G': 'C', 'T': 'A',
```

```
...     'a': 't', 'c': 'g', 'g': 'c', 't': 'a'
... }
```

If I use a `for` loop to iterate through a string of DNA, I can use the `dict.get()` method to safely request each base in a string of DNA to create the complement (return to Figure 3-1). Note that I will use the `base` as the optional second argument to `dict.get()`. If the base doesn't exist in the lookup table, then I'll default to using the base as-is just like the `else` case from the first solution:

```
>>> for base in 'AAAACCCGGT':
...     print(base, trans.get(base, base))
...
A T
A T
A T
A T
C G
C G
C G
G C
G C
T A
```

I can create a `complement` variable to hold the new string I generate:

```
>>> complement = ''
>>> for base in 'AAAACCCGGT':
...     complement += trans.get(base, base)
...
>>> complement
'TTTTGGGCCA'
```

You saw before that using the `reversed()` function on a string will return a list of the characters of the string in reverse order:

```
>>> list(reversed(complement))
['A', 'C', 'C', 'G', 'G', 'G', 'T', 'T', 'T', 'T']
```

I can use the `str.join()` function to create a new string from a list:

```
>>> ''.join(reversed(complement))
'ACCGGGTTTT'
```

When I put all these ideas together, the `main()` function becomes significantly shorter. It also becomes easier to expand because adding a new branch to the decision tree only requires adding a new key/value pair to the dictionary:

```
def main() -> None:
    args = get_args()
    trans = { ❶
        'A': 'T', 'C': 'G', 'G': 'C', 'T': 'A',
        'a': 't', 'c': 'g', 'g': 'c', 't': 'a'
    }

    complement = '' ❷
    for base in args.dna: ❸
        complement += trans.get(base, base) ❹

    print(''.join(reversed(complement))) ❺
```

❶ This is a dictionary showing how to translate one base to its complement.

❷ Initialize a variable to hold the DNA complement.

❸ Iterate through each base in DNA string.

❹ Append the translation of the base or the base itself to the complement.

❺ Reverse the complement and join the results on an empty string.

Python strings and lists are somewhat interchangeable. See how I can change the `complement` variable to a list and nothing else in the program changes:

```
def main() -> None:
    args = get_args()
    trans = {
        'A': 'T', 'C': 'G', 'G': 'C', 'T': 'A',
        'a': 't', 'c': 'g', 'g': 'c', 't': 'a'
    }

    complement = [] ❶
    for base in args.dna:
        complement += trans.get(base, base)

    print(''.join(reversed(complement)))
```

❶ Initialize the complement to an empty list instead of a string.

I am here highlighting that the `+=` operator works with both strings and list to append a new value at the end. There is also a `list.append()` method which does the same:

```
for base in args.dna:
    complement.append(trans.get(base, base))
```

The `reversed()` function works just as well on a list as it does a string. It's somewhat remarkable to me that using two different types for the `complement` results in so few changes to the code.

Solution 3: Using a List Comprehension

I suggested that you use a list comprehension without telling you what that is. If you've never used one before, it's essentially a way to write a `for` loop inside the square brackets (`[]`) used to create a new list (see Figure 3-2). When the goal of a `for` loop is to build up a new string or list, it makes much more sense to use a list comprehension.

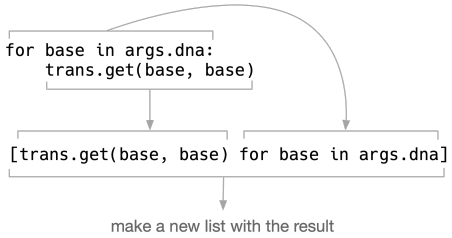


Figure 3-2. A list comprehension uses a for loop to generate a new list.

This shortens the three lines to initialize a `complement` and loop through the DNA down to one line:

```
def main() -> None:
    args = get_args()
    trans = {
        'A': 'T', 'C': 'G', 'G': 'C', 'T': 'A',
        'a': 't', 'c': 'g', 'g': 'c', 't': 'a'
    }

    complement = [trans.get(base, base) for base in args.dna] ❶
    print(''.join(reversed(complement)))
```

❶ Replace the `for` loop with a list comprehension.

Since the `complement` variable is only used once, I might even shorten this further by using the list comprehension directly:

```
print(''.join(reversed([trans.get(base, base) for base in args.dna])))
```

This is acceptable because the line is shorter than the maximum of 79 characters recommended by PEP8, but it's definitely not as readable as the longer version. You should use whatever version you feel is most immediately understandable.

Solution 4: Using str.translate

In Chapter 2, I used the `str.replace()` method to substitute all the *T*s to *U*s when transcribing DNA to RNA. Could I use that here? Let's try. I'll start by initializing the DNA string and replacing the *A*s with

*T*s. Remember that strings are *immutable*, meaning I can't change a string in-place but rather must overwrite the string with a new value:

```
>>> dna = 'AAAACCCGGT'
>>> dna = dna.replace('A', 'T')
```

Now let's look at the DNA:

```
>>> dna
'TTTTCCCGGT'
```

Can you see where this has started to go wrong? I'll complement the *T*s to *A*s now, and see if you can spot the problem:

```
>>> dna = dna.replace('T', 'A')
>>> dna
'AAAACCCGGA'
```

As shown in Figure 3-3, all the *A*s that turned into *T*s in the first move were just changed back to *A*s. Clearly, that way lies madness.

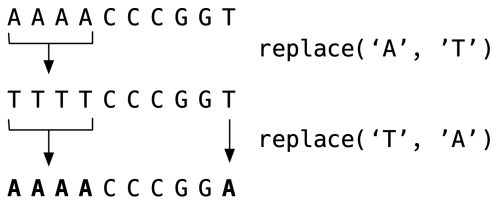


Figure 3-3. Iteratively using `str.replace()` leads to double replacements of values and the wrong answer.

Fortunately, Python has the `str.translate()` function for exactly this purpose. If you read `help(str.translate)`, you will find the function requires a table "which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None." The `trans` dictionary table will serve, but first it must be passed to the `str.maketrans()` function to transform the complement table into a form that uses the *ordinal* values of the keys:

```
>>> trans = {
...     'A': 'T', 'C': 'G', 'G': 'C', 'T': 'A',
...     'a': 't', 'c': 'g', 'g': 'c', 't': 'a'
... }
>>> str.maketrans(trans)
{65: 'T', 67: 'G', 71: 'C', 84: 'A', 97: 't', 99: 'g', 103: 'c', 116:
'a'}
```

You can see that the string key *A* was turned into the integer value 65 which is the same value returned by the `ord()` function:

```
>>> ord('A')
65
```

This value represents the ordinal position of the character *A* in the ASCII (American Standard Code for Information Interchange, pronounced *as-key*) table. That is, *A* is the 65th character in the table. The `chr()` function will reverse this process, providing the character represented by an ordinal value:

```
>>> chr(65)
'A'
```

The `str.translate()` function requires the complement table to have ordinal values for the keys, which is what I get from `str.maketrans()`:

```
>>> 'AAAACCCGGT'.translate(str.maketrans(trans))
'TTTTGGGCCA'
```

Finally, I need to reverse the complement. Here is a solution that incorporates all these ideas:

```
def main() -> None:
    args = get_args()

    trans = str.maketrans({ ❶
        'A': 'T', 'C': 'G', 'G': 'C', 'T': 'A',
        'a': 't', 'c': 'g', 'g': 'c', 't': 'a'
```

```
    })
    print(''.join(reversed(args.dna.translate(trans)))) ❷
```

❶ Create the translation table needed for the `str.translate()` function.

❷ Complement the DNA using the trans table, reverse, and join for a new string.

But, wait — there's more! There's another even shorter way to write this idea. If you read the `help(str.translate)` documentation:

> *If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y.*

So I can remove the `trans` dictionary and write the entire solution like this:

```
def main() -> None:
    args = get_args()
    trans = str.maketrans('ACGTacgt', 'TGCAtgca') ❶
    print(''.join(reversed(args.seq.translate(trans)))) ❷
```

❶ Make the translation table using two strings of equal lengths.

❷ Create the reverse complement.

If you really wanted to ruin someone's day — and in all likelihood, that person will be future you — this could be condensed into a single line of code.

Solution 5: Using Bio.Seq

I told you at the beginning that the final solution would involve an existing function[1]. There are many Python programmers working in bioinformatics who have created a set of modules under the name of Biopython. They have written and tested many incredibly useful algorithms, and it rarely makes sense to write your own code when you can use someone else's.

Be sure that you have first installed `biopython` by running the following:

```
$ python3 -m pip install biopython
```

I could import the entire module using `import Bio`, but it makes much more sense to only import the code I need. Here I only need the `Seq` class:

```
>>> from Bio import Seq
```

Now I can use the `Seq.reverse_complement()` function:

```
>>> Seq.reverse_complement('AAAACCCGGT')
'ACCGGGTTTT'
```

This final solution is the version I would recommend as it is the shortest and also uses existing, well-tested and documented modules that are almost ubiquitous in bioinformatics with Python:

```
def main() -> None:
    args = get_args()
    print(Seq.reverse_complement(args.dna)) ❶
```

❶ Use the `Bio.Seq.reverse_complement()` function.

When you run `mypy` on this solution (you *are* running `mypy` on every one of your programs, right?), you may get the following error:

```
================================== FAILURES
==================================
_____________________________________ revc.py
_____________________________________
6: error: Skipping analyzing 'Bio': found module but no type hints or
library stubs
6: note: See
https://mypy.readthedocs.io/en/latest/running_mypy.html#missing-
imports
==================================== mypy
====================================
Found 1 error in 1 file (checked 2 source files)
mypy.ini: No [mypy] section in config file

============================ short test summary info
============================
FAILED revc.py::mypy
!!!!!!!!!!!!!!!!!!!!!!!!! stopping after 1 failures
!!!!!!!!!!!!!!!!!!!!!!!!!
========================= 1 failed, 1 skipped in 0.20s
=========================
```

To silence this error, you can tell `mypy` to ignore imported files that are missing type annotations. In the root directory of the GitHub repository, you will find a file called *mypy.ini* with the following contents:

```
$ cat mypy.ini
[mypy]
ignore_missing_imports = True
```

Adding a *mypy.ini* file to any working directory allows you to make changes to the defaults that `mypy` uses *when you run it in the same directory*. If you would like to make this a global change so that `mypy` will use this no matter what directory you are in, then put this same content into *$HOME/.mypy.ini*.

Review

Manually creating the reverse complement of DNA is something of a rite of passage. Here's what I showed:

- You can write a decision tree algorithm using a series of `if`/`else` statements or by using a dictionary as a lookup table.
- Strings and lists are very similar. Both can be iterated using `for` loop, and the `+=` operator can be used to append to both.
- A list comprehension uses a `for` loop to iterate a sequence and generate a new list.
- The `reversed()` function is a lazy function that will return an iterator of the elements of a sequence in reverse order.
- You must use the `list()` function in the REPL to coerce lazy functions, iterators, and generators to generate their values.
- The `str.maketrans()` and `str.translate()` functions can be used to perform string substitution and generate a new string.
- The `ord()` function will return the ordinal value of a character, and conversely the `chr()` function will return the character for a given ordinal value.
- Biopython is a collection of modules and functions specific to bioinformatics. The preferred way to create the reverse complement of DNA is to use the `Bio.Seq.reverse_complement()` function.

---

1 This is kind of like how my high school calculus teacher spent a week teaching us how to perform manual derivatives, then showed us how it could be done in 20 seconds by pulling down the exponent and yada yada yada.

Chapter 4. Creating the Fibonacci Sequence: Writing, Testing, and Benchmarking Algorithms

Writing an implementation of the Fibonacci sequence is another step along the hero's journey to becoming a coder. The Rosalind description notes that the genesis for the sequence was a mathematical simulation of breeding rabbits that relies on some important (and unrealistic) assumptions:

- The first month starts with a pair of newborn rabbits
- Rabbits can reproduce after one month.
- Every month, every rabbit of reproductive age mates with another rabbit of reproductive age.
- Exactly one month after two rabbits mate, they produce a litter of the same size.
- Rabbits are immortal and never stop mating.

The sequence always begins with the numbers 0 and 1. The subsequent numbers can be generated *ad infinitum* by adding the two immediately previous values in the list as shown in Figure 4-1.

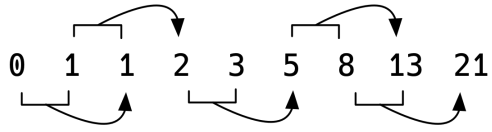


Figure 4-1. The first eight numbers of the Fibonacci sequence. After the initial 0 and 1, subsequent numbers are created by adding the two previous numbers.

If you search the internet for solutions, you'll find dozens of different ways to generate the sequence. I want to focus on three fairly different approaches. The first solution uses a very *imperative* approach where the algorithm strictly defines every step. The next solution uses a *generator* function, and the last will focus on a *recursive* solution. Recursion, while interesting, slows exponentially as I try to generate more of the sequence, but it turns out the performance problems can be solved using caching.

You will learn:

- How to manually validate arguments and throw errors
- How to use a list as a stack
- How to write a generator function
- How to write a recursive function
- Why recursive functions can be slow and how fix this with memoization
- How to use function decorators

Getting Started

The code and tests for this chapter are found in the *04_fib* directory. Start by copying the first solution to `fib.py`:

```
$ cd 04_fib/
$ cp solution1_list.py fib.py
```

Ask for the usage to see how the parameters are defined. You can use *n* and *k*, but I chose to use the names *generations* and *litter*:

```
$ ./fib.py -h
usage: fib.py [-h] generations litter

Calculate Fibonacci
```

```
positional arguments:
  generations  Number of generations
  litter       Size of litter per generation

optional arguments:
  -h, --help   show this help message and exit
```

This will be the first program to accept arguments that are not strings. The Rosalind challenge indicates that the program should accept two positive integer values:

- $n \leq 40$ representing the number of generations
- $k \leq 5$ representing the litter size produced by mate pairs

Try to pass non-integer values and notice how the program fails:

```
$ ./fib.py foo
usage: fib.py [-h] generations litter
fib.py: error: argument generations: invalid int value: 'foo'
```

You can't tell, but, in addition to printing the brief usage and a helpful error message, the program also generated a non-zero exit value. On the Unix command line, an exit value of *0* indicates success. I think of this as "zero errors."

In the `bash` shell, I can inspect the `$?` variable to look at the exit status of the most recent process. For instance, the command `echo Hello` should exit with a value of 0, and indeed it does:

```
$ echo Hello
Hello
$ echo $?
0
```

Try the previously failing command again, and then inspect `$?`:

```
$ ./fib.py foo
usage: fib.py [-h] generations litter
```

```
fib.py: error: argument generations: invalid int value: 'foo'
$ echo $?
2
```

That the exit status *2* is not as important as the fact that the value is not zero. This is a well-behaved program because rejects an invalid argument, prints a useful error message, and exits with a non-zero status. If this program were part of a chain of data processing steps (such as a Makefile, see appendix), the non-zero exit value would cause the entire process to stop which is a good thing. Programs which silently accept invalid values and fail softly or not at all can lead to unreliable and unreproducible results. It's vitally important that programs properly validate arguments and fail very convincingly when they cannot proceed.

The program is very strict even about the type of number it accepts. The values must be integers. It will also repel any floating-point values:

```
$ ./fib.py 5 3.2
usage: fib.py [-h] generations litter
fib.py: error: argument litter: invalid int value: '3.2'
```

NOTE

All command-line arguments to the program are technically received as strings. Even though "5" on the command line looks like the *number* 5, it's really the *character* "5". I am relying on `argparse` in this situation to attempt to convert the value from a string to an integer. When that fails, `argparse` generates these useful error messages.

Additionally, the program rejects values for the generations and litter parameters that are not in the allowed ranges. Notice that the error message includes the name of the argument and the offending value to provide sufficient feedback to the user so they can fix it:

```
$ ./fib.py -3 2
usage: fib.py [-h] generations litter
fib.py: error: generations "-3" must be between 1 and 40 ❶
$ ./fib.py 5 10
usage: fib.py [-h] generations litter
fib.py: error: litter "10" must be between 1 and 5 ❷
```

❶ The generations argument of *-3* value is not in the stated range of values.

❷ The litter argument of *10* value is too high.

Look at the first part of the solution to see how to make this work:

```
#!/usr/bin/env python3
""" Calculate Fibonacci """

import argparse
from typing import NamedTuple


class Args(NamedTuple):
    """ Command-line arguments """
    generations: int ❶
    litter: int ❷


# --------------------------------------------------
def get_args() -> Args:
    """ Get command-line arguments """

    parser = argparse.ArgumentParser(
        description='Calculate Fibonacci',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('gen', ❸
                        metavar='generations',
                        type=int, ❹
                        help='Number of generations')

    parser.add_argument('litter', ❺
                        metavar='litter',
```

```
                        type=int,
                        help='Size of litter per generation')

    args = parser.parse_args() ❻

    if not 1 <= args.gen <= 40: ❼
        parser.error(f'generations "{args.gen}" must be between 1 and
40') ❽

    if not 1 <= args.litter <= 5: ❾
        parser.error(f'litter "{args.litter}" must be between 1 and
5') ❿

    return Args(generations=args.gen, litter=args.litter) ⓫
```

❶ The `generations` field must be an `int`.

❷ The `litter` field must also be an `int`.

❸ The `gen` positional parameter is defined first, so it will receive the first positional value.

❹ The `type=int` indicates the required class of the value. Notice that `int` is a bareword indicating the class itself, not the name of the class.

❺ The `litter` positional parameter is defined second, so it will receive the second positional value.

❻ Attempt to parse the arguments. Any failure will result in error messages and the program exiting with a non-zero value.

❼ The `args.gen` value is now an actual `int` value, so I can perform numeric comparisons on it. Check if it is in the acceptable range.

❽ Use the `parser.error()` function to generate an error and exit the program.

❾ Likewise check the value of the `args.litter` argument.

- Generate an error that includes information the user needs to fix the problem.

- If the program makes it to this point, then the arguments are valid integer values in the accepted range, so return the `Args` using named fields.

I could check that the generattions and litter values are in the correct ranges in the `main()` function, but I prefer to do as much argument validation inside the `get_args()` function as possible so that I can use the `parser.error()` function to generate useful messages and exit the program with a non-zero value.

Remove the `fib.py` program and start anew either with `new.py` or your preferred method for creating a program:

```
$ new.py -fp 'Calculate Fibonacci' fib.py
Done, see new script "fib.py".
```

You can replace the `get_args()` with the preceding code, then modify your `main()` like so:

```
def main() -> None:
    args = get_args()
    print(f'generations = {args.generations}')
    print(f'litter = {args.litter}')
```

Run your program with invalid inputs and verify that you see the kinds of error messages shown. Try your program with acceptable values and verify you see this kind of output:

```
$ ./fib.py 1 2
generations = 1
litter = 2
```

Run `pytest` to see what your program passes and fails. You ought to pass the first four tests and fail the fifth:

```
$ pytest -xv
========================== test session starts
==========================
...

tests/fib_test.py::test_exists PASSED                                  [
14%]
tests/fib_test.py::test_usage PASSED                                   [
28%]
tests/fib_test.py::test_bad_generations PASSED                         [
42%]
tests/fib_test.py::test_bad_litter PASSED                              [
57%]
tests/fib_test.py::test_1 FAILED                                       [
71%] ❶

============================== FAILURES
==============================
_________________________________ test_1
_________________________________

    def test_1():
        """runs on good input"""

        rv, out = getstatusoutput(f'{RUN} 5 3') ❷
        assert rv == 0
>       assert out == '19' ❸
E       AssertionError: assert 'generations = 5\nlitter = 3' == '19' ❹
E         - 19    ❺
E         + generations = 5 ❻
E         + litter = 3

tests/fib_test.py:60: AssertionError
======================== short test summary info
========================
FAILED tests/fib_test.py::test_1 - AssertionError: assert
'generations...
!!!!!!!!!!!!!!!!!!!! stopping after 1 failures
!!!!!!!!!!!!!!!!!!!!
====================== 1 failed, 4 passed in 0.38s
======================
```

❶ The first failing test. Testing halts here because of the `-x` flag.

❷ The program is run with *5* for the number of generations and *3* for the litter size.

❸ The output should be *19*.

❹ This shows the two strings being compared are not equal.

❺ The expected value was *19*.

❻ This is the ouput that was received.

The output from `pytest` is trying very hard to point out exactly what went wrong. It shows how the program was run and what was expected versus what was actually produced. The program is supposed to print 19, which is the fifth number of the Fibonacci sequence when using a litter size of 3. If you want to finish the program on your own, please jump right in. You should use `pytest` to verify that you are passing all the tests. Also run `make test` to also check your program using `pylint`, `flake8`, and `mypy`. If you want some guidance, I'll cover the first approach I described.

An Imperative Approach

Figure 4-2 depicts the growth of the Fibonacci sequence.

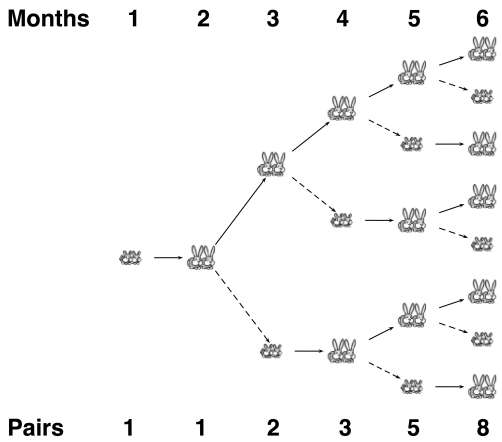


Figure 4-2. A visualization of the growth of the Fibonacci sequence as mating pairs of rabbits using a litter size of 1. The smaller rabbits indicate non-breeding pairs who must mature into larger, breeding pairs.

You can see that, to generate any number (after the first two), I need to know the two previous numbers. I can use this formula to describe the value of any position *n* of the Fibonacci sequence (F):

$F_n = F_{n-1} + F_{n-2}$

What kind of a data structure in Python would allow us to keep a sequence of numbers in order and refer to them by their position? A list. I'll start off with $F_1 = 0$ and $F_2 = 1$:

```
>>> fib = [0, 1]
```

The F_3 value is $F_2 + F_1 = 0 + 1 = 1$. When generating the next number, I'll always be referencing the *last two* elements of the sequence. It will be easiest to use negative indexing to indicate a position from the *end* of the list. The last value in a list is always `-1`:

```
>>> fib[-1]
1
```

The penultimate value is at `-2`:

```
>>> fib[-2]
0
```

I need to multiply this value by the litter size to calculate the number of offspring that generation created. To start, I'll consider a litter size of 1:

```
>>> litter = 1
>>> fib[-2] * litter
0
```

I want to add these two numbers together and append the result to the list:

```
>>> fib.append((fib[-2] * litter) + fib[-1])
>>> fib
[0, 1, 1]
```

If I do this again, I can see that the correct sequence is emerging:

```
>>> fib.append((fib[-2] * litter) + fib[-1])
>>> fib
[0, 1, 1, 2]
```

I need to repeat this action *generations* times. (Technically it will actually be *generations — 1* times because Python uses 0-based indexing.) I can use Python's `range()` function to generate a list of numbers starting from 0 up to but not including the end value:

```
>>> fib = [0, 1]
>>> litter = 1
>>> generations = 5
>>> for i in range(generations - 1):
...     fib.append((fib[-2] * litter) + fib[-1])
...
>>> fib
[0, 1, 1, 2, 3, 5]
```

This should be enough for you to create a solution that passes the tests. In the next part, I'll cover two other solutions that highlight some very interesting parts of Python.

Solutions

As with previous solutions, all the following share the same `get_args()` which will be omitted.

Solution 1: An Imperative Solution Using a List as a Stack

Here is how I wrote my imperative solution. I'm using a list as a kind of *stack* to keep track of past values. I don't need all the values, just the last two, but it's pretty easy to keep growing the list and referring to the last two:

```
def main() -> None:
    args = get_args()

    fib = [0, 1] ❶
    for _ in range(args.generations - 1): ❷
        fib.append((fib[-2] * args.litter) + fib[-1]) ❸

    print(fib[-1]) ❹
```

❶ Start with 0 and 1.

❷ Use the `range()` function to create the right number of loops.

❸ Append the next value to the sequence.

❹ Print the last number of the sequence.

NOTE

I used the _ variable name in the `for` loop to indicate that I don't intend to use the variable. The underscore is a valid Python identifier, and it's also a convention to use this to indicate a *throwaway* value. Linting tools, for instance, might see that I've assigned a variable some value but never used itwhich would normally indicate a possible error. The underscore variable will be ignored and is used to indicate that I intend to ignore that value. In this case, I'm using the `range()` function purely for the side effect of creating the number of loops needed.

This is considered an *imperative* solution because the code directly encodes every instruction of the algorithm. When you read the recursive solution, you will see that the algorithm can be written in a more declarative manner which also has unintended consequences that I must handle.

A slight variation on this would be to place this code inside a function I'll call `fib()`. Note that it's possible in Python to declare a function inside another function as here I'll create `fib()` inside the `main()`. The reason I do this is so I can reference the `args.litter` parameter, creating a *closure* because the function is capturing the runtime value of the litter size:

```
def main() -> None:
    args = get_args()

    def fib(n: int) -> int: ❶
        nums = [0, 1] ❷
        for i in range(n - 1):
            nums.append((nums[-2] * args.litter) + nums[-1]) ❸
        return nums[-1] ❹

    print(fib(args.generations)) ❺
```

❶ Create a function called `fib()` that accepts an integer parameter `n` and returns an integer.

❷ This is the same code as before. Note this list is called `nums` so it doesn't clash with the function name.

❸ The function references the `args.litter` parameter and so creates a closure.

❹ Use `return` to send the final value back to the caller.

❺ Call the `fib()` function with the `args.generations` parameter.

The scope of the `fib()` function in the preceding example is limited to the `main()` function. *Scope* refers the part of the program where a particular function name or variable is visible or legal.

I don't have to use a closure. Here is how I can express the same idea with a standard function:

```
def main() -> None:
    args = get_args()

    print(fib(args.generations, args.litter)) ❶


def fib(n: int, litter: int) -> int: ❷
    nums = [0, 1]
    for i in range(n - 1):
        nums.append((nums[-2] * litter) + nums[-1])

    return nums[-1]
```

❶ The `fib()` function must be called with two arguments.

❷ The function requires both the number of generations and the litter size. The function body is essentially the same.

In the preceding code, you see that I must pass two arguments to `fib()` whereas the closure required only one argument because the

`litter` was captured. Binding values and reducing the number of parameters is one valid reaons for creating closures.

Another reason to write a closure is to limit the scope of a function. The closure definition of `fib()` is valid only inside the `main()` function, but the preceding version is visible throughout the program. Hiding a function inside another function makes it harder to test. In this case, the `fib()` function is almost the entire program, so the tests have already been written in *tests/fib_test.py*. Later, I'll show you how to write and run tests for functions.

Solution 2: Creating a Generator Function

In the previous solution, I generated the Fibonacci sequence up to the value requested and then stopped; however, the sequence is infinite. Could I create a function that could generate *all* the numbers of the sequence? Technically, yes, but it would never finish, what with being infinite and all.

Python has a way to suspend a function that generates a possibly infinite sequence. I can use the `yield` statement to return a value from a function, temporarily leaving the function only to resume at the same state when the next value is requested. This kind of function is called a *generator*, and here is how I can use it to generate the sequence:

```
def fib(k: int) -> Generator[int, None, None]: ❶
    x, y = 0, 1 ❷
    yield x ❸

    while True: ❹
        yield y ❺
        x, y = y * k, x + y ❻
```

❶ The type signature indicates the function takes the parameter `k` (litter size) which must be an `int`. It returns a special type of

function of the type `Generator` which yields `int` values and has no *send* or *return* types.

❷ I only ever need to track the last two generations which I initialize to 0 and 1.

❸ Yield the 0.

❹ Create an infinite loop.

❺ Yield the last generation.

❻ Set `x` (2 generations back) to the current generation times the litter size. Set the `y` (1 generation back) to the sum of the two current generations.

A generator acts like an iterator, producing values as requested by the code until it is exhausted. The type signature for `Generator` looks a little complicated since it defines types for the yield, send, and return. I don't need to dive into it further here, but I recommend you read the docs on the `typing` module. Since this generator will only generate yield values, the send and return types are `None`. Otherwise, this code does exactly what the first version of the program did only inside a fancy-pants generator function. See Figure 4-3 to consider how the function works for two different litter sizes.

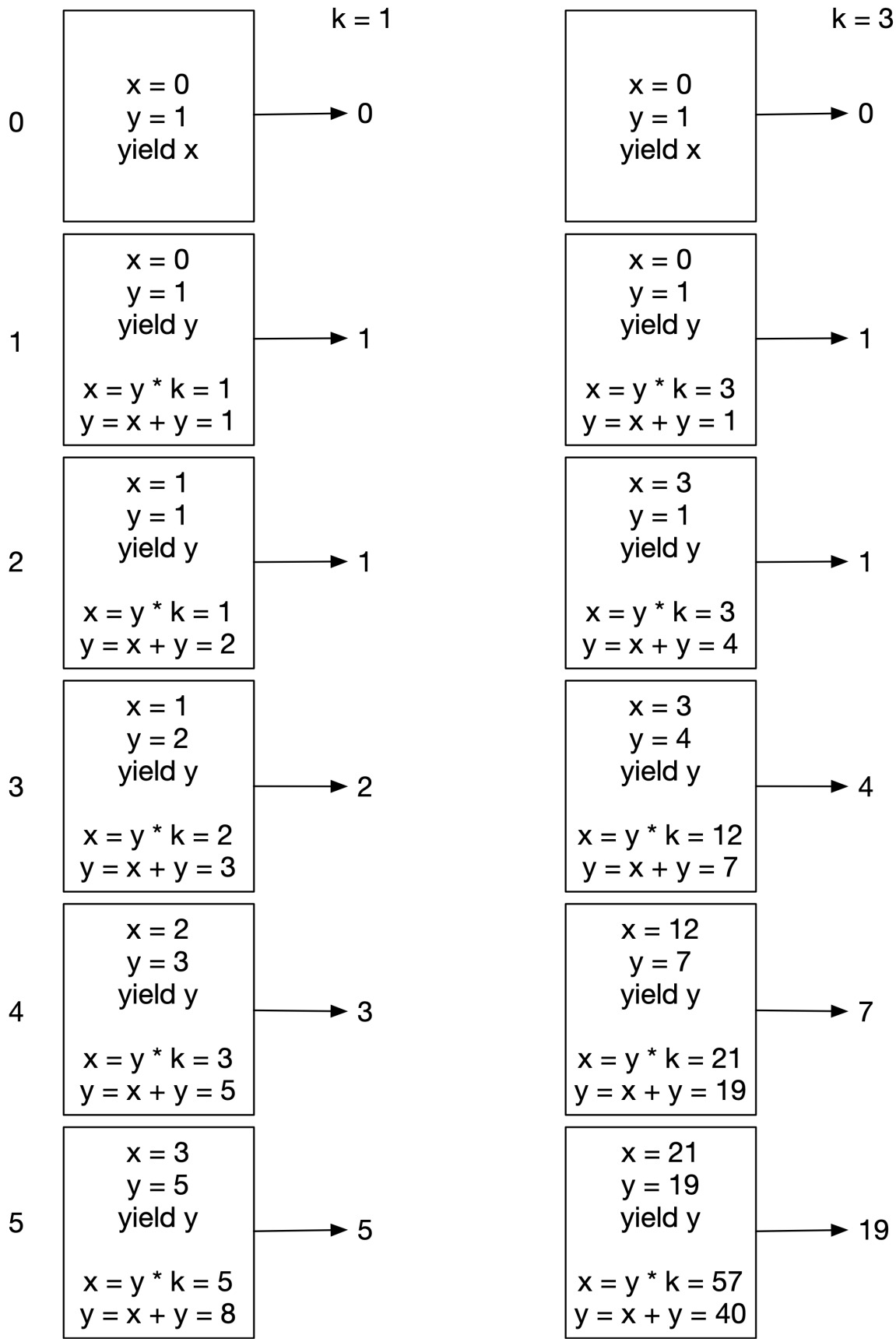

k = 1
k = 3
0
x = 0
y = 1
yield x
0
0
1
x = 0
y = 1
yield y
x = y * k = 1
y = x + y = 1
1
x = 0
y = 1
yield y
x = y * k = 3
y = x + y = 1
1
2
x = 1
y = 1
yield y
x = y * k = 1
y = x + y = 2
1
x = 3
y = 1
yield y
x = y * k = 3
y = x + y = 4
1
3
x = 1
y = 2
yield y
x = y * k = 2
y = x + y = 3
2
x = 3
y = 4
yield y
x = y * k = 12
y = x + y = 7
4
4
x = 2
y = 3
yield y
x = y * k = 3
y = x + y = 5
3
x = 12
y = 7
yield y
x = y * k = 21
y = x + y = 19
7
5
x = 3
y = 5
yield y
x = y * k = 5
y = x + y = 8
5
x = 21
y = 19
yield y
x = y * k = 57
y = x + y = 40
19

Figure 4-3. A depiction of how the `fib()` generator's state changes over time (n=5) for two litter sizes (k=1 and k=3).

Here's how to use it:

```
def main() -> None:
    args = get_args()
    gen = fib(args.litter) ❶
    seq = [next(gen) for _ in range(args.generations + 1)] ❷
    print(seq[-1]) ❸
```

❶ The `fib()` function takes the litter size as an argument and returns a generator.

❷ Use the `next()` function to retrieve the next value from a generator. Use a list comprehension to do this the correct number of times to generate the sequence up to the request value.

❸ Print the last number in the sequence.

> **NOTE**
>
> The `range()` is different because the first version already had the 0 and 1 in place. Here I have to call the generator two extra times to produce those values.

Although I prefer the list comprehension, I don't need the entire list. I only care about the final value, so I could I could have written it like so:

```
def main() -> None:
    args = get_args()
    gen = fib(args.litter)
    answer = 0 ❶
    for _ in range(args.generations + 1): ❷
        answer = next(gen) ❸
    print(answer) ❹
```

❶ Initialize the answer to 0.

❷ Create the correct number of loops.

❸ Get the value for the current generation.

❹ Print the answer.

As it happens, it's quite common to call a function repeatedly to generate a list, and so there is a function to do this for us. The `itertools.islice()` function will "Make an iterator that returns selected elements from the iterable." Here is how I can use it:

```
def main() -> None:
    args = get_args()
    seq = list(islice(fib(args.litter), args.generations + 1)) ❶
    print(seq[-1]) ❷
```

❶ The first argument to `islice()` is the function that will be called, and the second argument is the number of times to call it. The function is lazy, so I use `list()` to coerce the values.

❷ Print the last value.

Since I only use the `seq` variable one time, I could eschew that assignment. If benchmarking proved the following to be the best-performing version, I might be willing to write a one-liner:

```
def main() -> None:
    args = get_args()
    print(list(islice(fib(args.litter), args.generations + 1))[-1])
```

Clever code is fun but can become unreadable. You have been warned.

It's such a fine line between stupid and clever.

—David St. Hubbins and Nigel Tufnel

Generators are cool but more complex than generating a list. They are the appropriate way to generate a very large or potentially infinite sequence of values because they are lazy, only computing the next value when your code actually requires it.

Solution 3: Using Recursion and Memoization

While there are many more fun ways to write an algorithm to produce an infinite series of numbers, I'll show just one more using recursion which is when a function calls itself:

```
def main() -> None:
    args = get_args()

    def fib(n: int) -> int: ❶
        return 1 if n in (1, 2) \ ❷
            else fib(n - 2) * args.litter + fib(n - 1) ❸

    print(fib(args.generations)) ❹
```

❶ Define a function called `fib()` that takes the number of the generation wanted as an `int` and returns an `int`.

❷ If the generation is 1 or 2, return 1. This is the all-important base case that does not make a recursive call.

❸ For all other cases, call the `fib()` function twice, once for two generations back and another for the previous generation. Factor in the litter size as before.

❹ Print the results of the `fib()` function for the given generations.

NOTE

Here's another instance where I define a `fib()` function as a closure *inside* the `main()` function so as to use the `args.litter` value inside the `fib()` function. The reason is to close around `args.litter`, effectively binding that value to the function. If I had defined the function outside the `main()` function, I would have to pass the `args.litter` argument on the recursive calls.

This is a really elegant solution that gets taught in pretty much every introductory computer science class. It's fun to study, but it turns out to be wicked slow because I end up calling the function so many times. That is, `fib(5)` needs to call `fib(4)` and `fib(3)` to add those values. In turn, `fib(4)` needs to call `fib(3)` and `fib(2)` and so on. Figure 4-4 shows that `fib(5)` results in 14 function calls to produce five distinct values. For instance, `fib(2)` is calculated three times, but we only need to calculate it once.

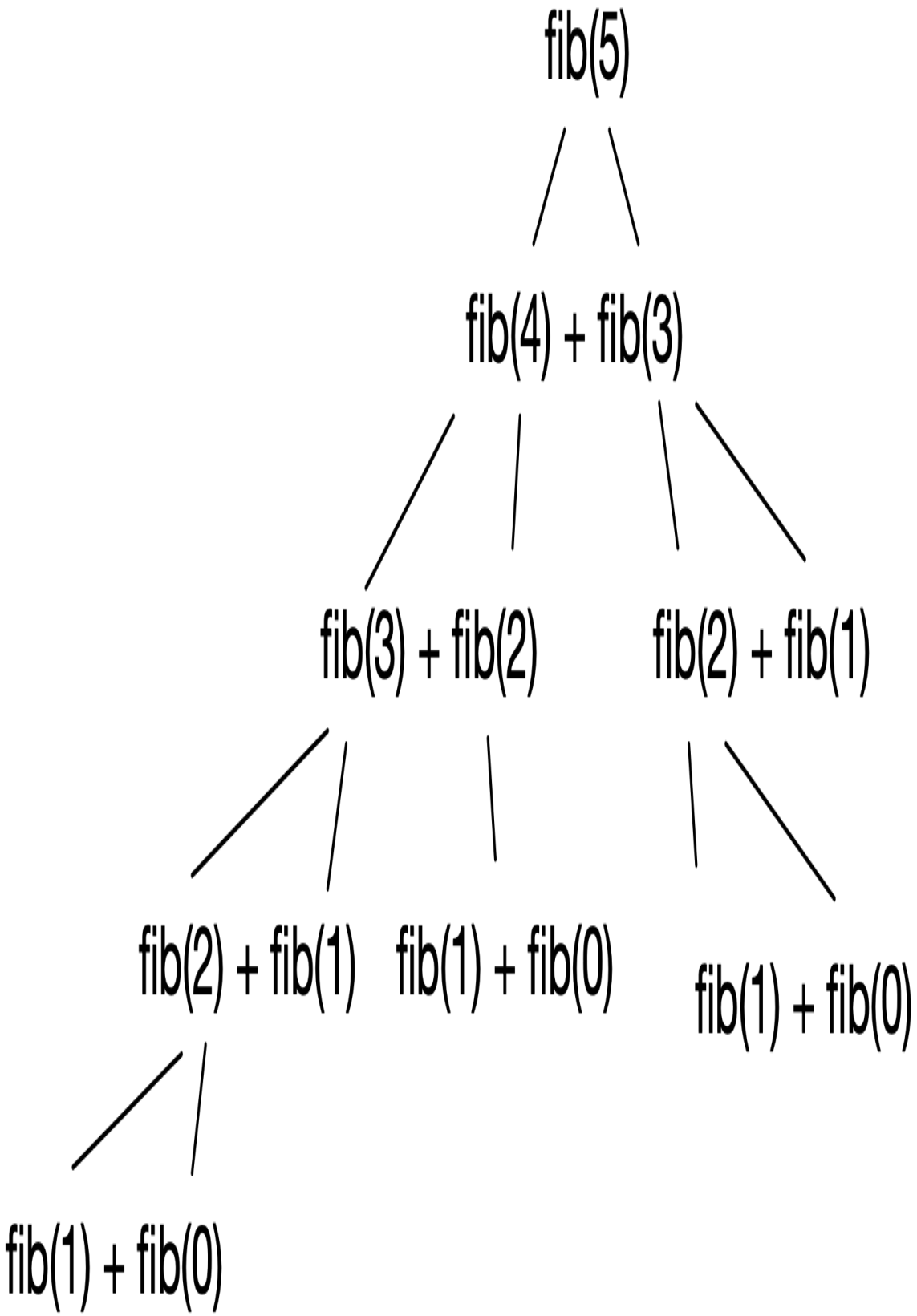

fib(5)
fib(4) + fib(3)
fib(3) + fib(2)
fib(2) + fib(1)
fib(2) + fib(1)
fib(1) + fib(0)
fib(1) + fib(0)
fib(1) + fib(0)

Figure 4-4. The call stack for `fib(5)` results in many recursive calls to the function that increase approximately exponentially as the input value increases.

To illustrate the problem, I'll take a sampling of how long this program takes to finish up to the maximum *n* of 40. Again I'll use a `for` loop in `bash` to show you how I would commonly benchmark such a program on the command line:

```
$ for n in 10 20 30 40;
> do echo "==> $n <==" && time ./solution3_recursion.py $n 1
> done
==> 10 <==
55

real    0m0.045s
user    0m0.032s
sys     0m0.011s
==> 20 <==
6765

real    0m0.041s
user    0m0.031s
sys     0m0.009s
==> 30 <==
832040

real    0m0.292s
user    0m0.281s
sys     0m0.009s
==> 40 <==
102334155

real    0m31.629s
user    0m31.505s
sys     0m0.043s
```

The jump from 0.29s for *n* = 30 to 31s for *n* = 40 is huge. Imagine going to 50 and beyond. I need to either find a way to speed this up or abandon all hope for recursion. The solution is to cache previously calculated results. This is called *memoization*, and there are many

ways to implement this. The following is one method. Note you will need to import `typing.Callable`:

```
def memoize(f: Callable) -> Callable: ❶
    """ Memoize a function """

    cache = {} ❷

    def memo(x): ❸
        if x not in cache: ❹
            cache[x] = f(x) ❺
        return cache[x] ❻

    return memo ❼
```

❶ Define a function that takes a function (something that is *callable*) and returns a function.

❷ Use a dictionary to store cached values.

❸ Define `memo()` as a closure around the cache. The function will take some parameter `x` when called.

❹ See if the argument value is in the cache.

❺ If not, call the function with the argument and set the cache for that argument value to the result.

❻ Return the cached value for the argument.

❼ Return the new function.

Note that the `memoize()` function returns a new function. In Python, functions are considered *first-class objects* meaning they can be used like other kinds of variables — you can pass them as arguments and overwrite their definitions. The `memoize()` function is an example of a *higher-order function* (HOF) because it takes other

functions as arguments. I'll be using other HOFs like `filter()` and `map()` throughout the book.

To use the `memoize()` function, I will define `fib()` and then *redefine it* with the memoized version. If you run this, you will see an almost instantaneous result no matter how high *n* goes.

```
def main() -> None:
    args = get_args()

    def fib(n: int) -> int:
        return 1 if n in (1, 2) else fib(n - 2) * args.litter + fib(n
- 1)

    fib = memoize(fib) ❶

    print(fib(args.generations))
```

❶ Overwrite the existing `fib()` definition with the memoized function.

A preferred method accomplish this uses a *decorator*, which is a function that modifies another function:

```
def main() -> None:
    args = get_args()

    @memoize ❶
    def fib(n: int) -> int:
        return 1 if n in (1, 2) else fib(n - 2) * args.litter + fib(n
- 1)

    print(fib(args.generations))
```

❶ Decorate the `fib()` function with the `memoize()` function.

As fun as writing memoization functions is, it again turns out that this is such a common practice that others have already solved it for us. I

can remove the `memoize()` function and rather import the `functools.lru_cache` (least-recently-used cache) function:

```
from functools import lru_cache
```

Decorate the `fib()` function with the `lru_cache()` function to get memoization with minimal distraction:

```
def main() -> None:
    args = get_args()

    @lru_cache() ❶
    def fib(n: int) -> int:
        return 1 if n in (1, 2) else fib(n - 2) * args.litter + fib(n
- 1)

    print(fib(args.generations))
```

❶ Memoize the `fib()` function via decoration with the `lru_cache()` function. Note that Python 3.6 requires the parentheses, but 3.8 and later versions do not.

Benchmarking the Solutions

Which is the fastest solution? I've shown you how to use a `for` loop in bash with the `time` command to compare the run times of commands:

```
$ for py in ./solution1_list.py ./solution2_generator_islice.py \
./solution3_recursion_lru_cache.py; do echo $py && time $py 40 5; done
./solution1_list.py
148277527396903091

real    0m0.070s
user    0m0.043s
sys     0m0.016s
./solution2_generator_islice.py
148277527396903091
```

```
real    0m0.049s
user    0m0.033s
sys     0m0.013s
./solution3_recursion_lru_cache.py
148277527396903091

real    0m0.041s
user    0m0.030s
sys     0m0.010s
```

It would appear that the recursive solution using LRU caching is the fastest, but again I have very little data — just one run per program. Also, I have to eyeball this data and figure out which is the fastest.

There's a better way. I have installed a tool called `hyperfine` to run each command many times and compare the results:

```
$ hyperfine -L prg
./solution1_list.py,./solution2_generator_islice.py,\
./solution3_recursion_lru_cache.py '{prg} 40 5' --prepare 'rm -rf
__pycache__'
Benchmark #1: ./solution1_list.py 40 5
  Time (mean ± σ):      38.1 ms ±   1.1 ms    [User: 28.3 ms, System:
8.2 ms]
  Range (min … max):    36.6 ms …  42.8 ms    60 runs

Benchmark #2: ./solution2_generator_islice.py 40 5
  Time (mean ± σ):      38.0 ms ±   0.6 ms    [User: 28.2 ms, System:
8.1 ms]
  Range (min … max):    36.7 ms …  39.2 ms    66 runs

Benchmark #3: ./solution3_recursion_lru_cache.py 40 5
  Time (mean ± σ):      37.9 ms ±   0.6 ms    [User: 28.1 ms, System:
8.1 ms]
  Range (min … max):    36.6 ms …  39.4 ms    65 runs

Summary
  './solution3_recursion_lru_cache.py 40 5' ran
    1.00 ± 0.02 times faster than './solution2_generator_islice.py 40
5'
    1.01 ± 0.03 times faster than './solution1_list.py 40 5'
```

It appears that `hyperfine` ran each command 60-66 times, averaged the results, and found `solution3_recursion_lru_cache.py` program is perhaps slightly faster. Another benchmarking tool you might find useful is `bench`, but you can search for other benchmarking tools on the internet that might suit your tastes more. Whatever tool you use, benchmarking along with testing is vital to challenging assumptions about your code.

> **NOTE**
>
> I used the `--prepare` option to tell `hyperfine` to remove the `__pycache__` directory before running the commands. This is a directory created by Python to cache *bytecode* of the program. If a program's source code hasn't changed since the last time it was run, then Python can skip compilation and used the bytecode version that exists in the `__pycache__` directory. I needed to remove this as `hyperfine` detected statistical outliers when running the commands, probably due to caching effects.

Testing the Good, the Bad, and the Ugly

For every challenge, I hope you spend part of your time reading through the tests. Learning how to design and write tests is as important as anything else I'm showing you. As I mentioned before, my first tests check that the expected program exists and will produce usage statements on request. After that, I usually give invalid inputs to ensure the program fails. I would like to highlight the tests for bad *n* and *k* parameters. They are essentially the same, so I'll just show the first one as it demonstrates how to randomly select an invalid integer value — one that is possibly negative or too high:

```
def test_bad_n():
    """ Dies when n is bad """

    n = random.choice(list(range(-10, 0)) + list(range(41, 50))) ❶
    k = random.randint(1, 5) ❷
    rv, out = getstatusoutput(f'{RUN} {n} {k}') ❸
```

```
    assert rv != 0 ❹
    assert out.lower().startswith('usage:') ❺
    assert re.search(f'n "{n}" must be between 1 and 40', out) ❻
```

❶ Join the two lists of invalid number ranges and randomly select one value.

❷ Select a random integer in the range from 1 to 5 (both bounds inclusive).

❸ Run the program with the arguments and capture the output.

❹ Make sure the program reported a failure (non-zero exit value).

❺ Check the output begins with a usage statement.

❻ Look for an error message describing the problem with the *n* argument.

I often like to use randomly selected invalid values when testing. This partially comes from writing tests for students so that they won't write programs that fail on a single bad input, but I also find it helps me to not accidentally code for a specific input value. I haven't yet covered the `random` module, but it gives you a way to make pseudo-random choices. First, you need to import the module:

```
>>> import random
```

For instance, you can use `random.randint()` to select a single integer from a given range:

```
>>> random.randint(1, 5)
2
>>> random.randint(1, 5)
5
```

Or use the `random.choice()` function to randomly select a single value from some sequence. Here I wanted to construct a discontiguous range of negative numbers separated from a range of positive numbers:

```
>>> random.choice(list(range(-10, 0)) + list(range(41, 50)))
46
>>> random.choice(list(range(-10, 0)) + list(range(41, 50)))
-1
```

The tests that follow all provide good inputs to the program, for example:

```
def test_2():
    """ Runs on good input """

    rv, out = getstatusoutput(f'{RUN} 30 4') ❶
    assert rv == 0 ❷
    assert out == '436390025825' ❸
```

❶ These are values I was given while attempting to solve the Rosalind challenge.

❷ The program should not fail on this input.

❸ This is the correct answer per Rosalind.

Testing, like documentation, is a love letter to your future self. As tedious as testing may seem, you'll appreciate failing tests when you try to add a feature and end up accidentally breaking something else. Assiduously writing and running tests can prevent you from deploying broken programs.

Running the Test Suite on All the Solutions

You've seen that in each chapter I write multiple solutions to explore various ways to solve the problems. I completely rely on my tests to ensure my programs are correct. You might be curious to see how I've automated the process of testing every single solution. Look at the *Makefile* and see the `all` target:

```
$ cat Makefile
.PHONY: test

test:
        python3 -m pytest -xv --flake8 --pylint --mypy fib.py
tests/fib_test.py

all:
    ../bin/all_test.py fib.py
```

> **WARNING**
>
> The `all_test.py` program will overwrite the `fib.py` program with each of the solutions before running the test suite. This could overwrite your solution. Be sure you commit your version to Git or at least make a copy before you run `make all` or you could lose your work.

The following is the `all_test.py` program that is run by the `all` target. I'll break it into two parts starting with the first part up to `get_args()`. Most of this should be familiar by now:

```
#!/usr/bin/env python3
""" Run the test suite on all solution*.py """

import argparse
import os
import re
import shutil
import sys
from subprocess import getstatusoutput
from functools import partial
from typing import NamedTuple
```

```
class Args(NamedTuple):
    """ Command-line arguments """
    program: str ❶
    quiet: bool ❷


# --------------------------------------------------
def get_args() -> Args:
    """ Get command-line arguments """

    parser = argparse.ArgumentParser(
        description='Run the test suite on all solution*.py',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('program', metavar='prg', help='Program to
test') ❸

    parser.add_argument('-q', '--quiet', action='store_true', help='Be
quiet') ❹

    args = parser.parse_args()

    return Args(args.program, args.quiet)
```

❶ The name of the program to test which in this case is `fib.py`.

❷ A Boolean value of `True` or `False` to create more or less output

❸ The default type is `str`

❹ The `action='store_true'` makes this a Boolean flag. If the flag is present, the value will be `True` and `False` otherwise.

The `main()` function is where the testing happens:

```
def main() -> None:
    args = get_args()
    cwd = os.getcwd() ❶
    solutions = list( ❷
        filter(partial(re.match, r'solution.*\.py'), os.listdir(cwd)))
```

❸

```
    for solution in sorted(solutions): ❹
        print(f'==> {solution} <==')
        shutil.copyfile(solution, os.path.join(cwd, args.program)) ❺
        subprocess.run(['chmod', '+x', args.program], check=True) ❻
        rv, out = getstatusoutput('make test') ❼
        if rv != 0: ❽
            sys.exit(out) ❾

        if not args.quiet: ❿
            print(out)

    print('Done.') ⓫
```

❶ Get the current working directory which will be the *04_fib* directory if you are in that directory when running the command.

❷ Find all the `solution*.py` files in the current directory.

❸ Both `filter()` and `partial()` are HOFs I'll explain next.

❹ The filenames will be in random order, so iterate through the sorted files.

❺ Copy the `solution*.py` file to the testing filename.

❻ Make the program executable.

❼ Run the `make test` command, capture the return value and output.

❽ See if the return value is not zero.

❾ Exit this program while printing the output from the testing and returning a non-zero value.

❿ Unless the program is supposed to be quiet, print the testing output.

- Let the user know the program finishes normally.

I haven't covered `filter()` or `partial()` yet, but these are both HOFs. I'll explain how and why I'm using them. To start, the `os.listdir()` function will return the entire contents of a directory including files and directories:

```
>>> import os
>>> files = os.listdir()
```

There's a lot there, so I'll import the `pprint()` function from the `pprint` module to *pretty-print* this:

```
>>> from pprint import pprint
>>> pprint(files)
['solution3_recursion_memoize_decorator.py',
 'solution2_generator_for_loop.py',
 '.pytest_cache',
 'Makefile',
 'solution2_generator_islice.py',
 'tests',
 '__pycache__',
 'fib.py',
 'README.md',
 'solution3_recursion_memoize.py',
 'bench.html',
 'solution2_generator.py',
 '.mypy_cache',
 '.gitignore',
 'solution1_list.py',
 'solution3_recursion_lru_cache.py',
 'solution3_recursion.py']
```

I want to filter those to names that start with *solution* and end with *.py*. On the command line, I would match this pattern using a *file glob* like `solution*.py` where the `*` means *zero or more of any character* and the `.` is a literal dot. A regular expression version of this pattern is the slightly more complicated `solution.*\.py` because `.` (dot) is a regex metacharacter representing *any character* and `*`

(star or asterisk) means *zero or more* (see Figure 4-5). To indicate a literal dot, I need to escape it with a backslash (`\.`). Note that it's prudent to use the r-string (*raw* string) to enclose this pattern.

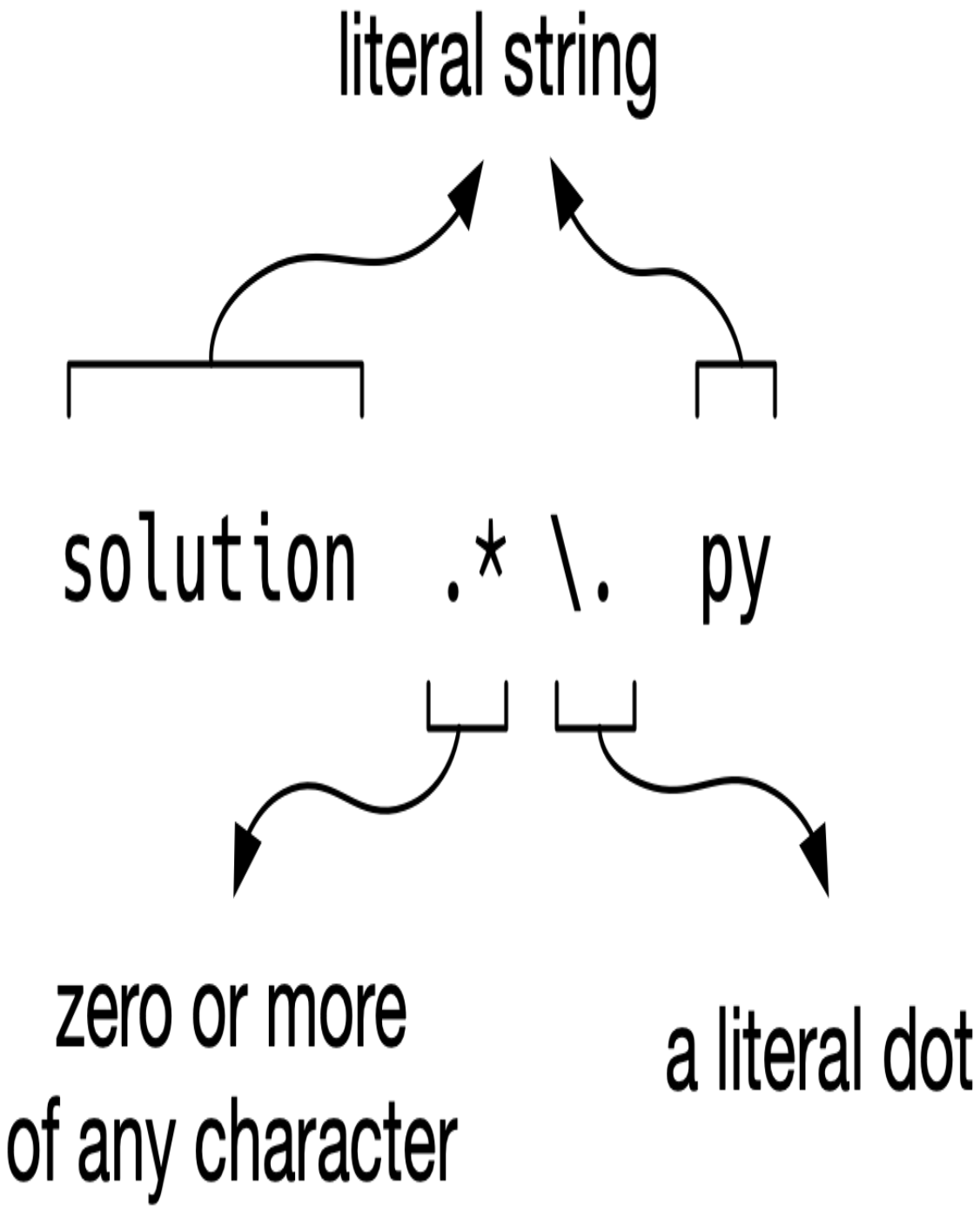


Figure 4-5. A regular expression to find files matching the file glob "solution.py":*

When the match is successful, an `re.Match` object is returned:

```
>>> import re
>>> re.match(r'solution.*\.py', 'solution1.py')
<re.Match object; span=(0, 12), match='solution1.py'>
```

When a match fails, the `None` value is returned. I have to use `type()` here because the `None` value is not displayed in the REPL:

```
>>> type(re.match(r'solution.*\.py', 'fib.py'))
<class 'NoneType'>
```

I want to apply this match to all the files returned by `os.listdir()`. I can use `filter()` and the `lambda` keyword to create an *anonymous* function. Each filename in `files` is passed as the `name` argument used in the match. The `filter()` will only return elements that return a truthy value from the given function, so those filenames that return `None` when they fail to match are excluded:

```
>>> pprint(list(filter(lambda name: re.match(r'solution.*\.py', name),
files)))
['solution3_recursion_memoize_decorator.py',
 'solution2_generator_for_loop.py',
 'solution2_generator_islice.py',
 'solution3_recursion_memoize.py',
 'solution2_generator.py',
 'solution1_list.py',
 'solution3_recursion_lru_cache.py',
 'solution3_recursion.py']
```

You see that `re.match()` function takes two arguments — a pattern and a string to match. The `partial()` function allows me to *partially apply* the function, and the result is a new function. For example, the `operator.add()` function expects two values and returns their sum:

```
>>> import operator
>>> operator.add(1, 2)
3
```

I can create a function that adds one to any value like so:

```
>>> from functools import partial
>>> succ = partial(op.add, 1)
```

The `succ()` function requires one argument, and will return the successor:

```
>>> succ(3)
4
>>> succ(succ(3))
5
```

Likewise, I can create a function `f()` that partially applies the `re.match()` function with its first argument, a regular expression pattern:

```
>>> f = partial(re.match, r'solution.*\.py')
```

The `f()` function is waiting for a string to apply the match:

```
>>> type(f('solution1.py'))
<class 're.Match'>
>>> type(f('fib.py'))
<class 'NoneType'>
```

If you call it without an argument, you will get an exception:

```
>>> f()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: match() missing 1 required positional argument: 'string'
```

I can replace the `lambda` with the partially applied function as the first argument to `filter()`:

```
>>> pprint(list(filter(f, files)))
['solution3_recursion_memoize_decorator.py',
```

```
    'solution2_generator_for_loop.py',
    'solution2_generator_islice.py',
    'solution3_recursion_memoize.py',
    'solution2_generator.py',
    'solution1_list.py',
    'solution3_recursion_lru_cache.py',
    'solution3_recursion.py']
```

My programming style leans heavily on purely functional programming ideas. I find this style to be like playing with Legos — small, well-defined, and tested functions can be composed into larger programs that work well.

Going Further

There are many different styles of programming such as procedural, functional, object-oriented, and so forth. Even within an object-oriented language like Python, I can use very different approaches to writing code. The first solution could be considered a *dynamic programming* approach because you try to solve the larger problem by first solving smaller solutions. If you find recursive functions interesting, the Towers of Hanoi problem is another classic exercise. Purely functional languages like Haskell mostly avoid constructs like `for` loops and rely heavily on these kinds of solutions. Both spoken and programming languages shape the way we think, and I encourage you to try solving this problem using other languages you know to see how you might write different solutions.

Review

- Inside the `get_args()` function, you can perform manual validation of arguments and use the `parser.error()` function to manually generate `argparse` errors.
- A list can be used as a stack by pushing and popping elements.

- Using `yield` in a function turns it into a generator. When the function yields a value, the value is returned and the state of the function is perserved until the next value is requested. Generators can be used to create a potentially infiinite stream of values.
- A recursive function calls itself, and the recursion can cause serious performance issues. One solution is to use memoization to cache values and avoid recomputation.
- Higher-order functions are functions that take other functions as arguments.
- Python's function decorators apply HOF to other functions.
- Benchmarking is an important technique for determining the best-performing algorithm. The `hyperfine` and `bench` tools allow one to compare program runtimes of commands over many iterations.
- The `random` module offers many functions for the pseudo-random selection of values.

Chapter 5. Computing GC Content: Parsing FASTA and Analyzing Sequences

In Chapter 1, you counted all the bases in a string of DNA. In this exercise, you need to count the Gs and Cs in a sequence and divide by the length of the sequence to determine the GC content as described on the Rosalind GC page. GC content is informative in several ways. A higher GC content level indicates a relatively higher melting temperature in molecular biology, and DNA sequences that encode proteins tend to be found in GC-rich regions. There are many ways to solve this problem, and they all start with using Biopython to parse a FASTA file, a key file format in bioinformatics. I'll show you how to use the `Bio.SeqIO` module to iterate over the sequences in the file to identify the sequence with the highest GC content.

You will learn:

- How to parse FASTA format using `Bio.SeqIO`
- Several ways to express the notion of a `for` loop using list comprehensions, `filter()`, and `map()`
- How to address runtime challenges such as memory allocation when parsing large files
- More about the `sorted()` function
- How to include formatting instructions in format strings
- How to use the `sum()` function to add a list of numbers

- How to use regular expressions to count the occurences of a pattern in a string

Getting Started

All the code and tests for this program are in the *05_gc* directory. While I'd like to name this program `gc.py`, it turns out that this conflicts with a very important Python module called gc.py which is used for garbage collection such as freeing memory. Instead, I'll use `cgc.py` for *calculate GC*.

Start by copying the first solution and asking for the usage:

```
$ cp solution1_list.py cgc.py
$ ./cgc.py -h
usage: cgc.py [-h] FILE

Compute GC content

positional arguments:
  FILE        Input sequence file

optional arguments:
  -h, --help  show this help message and exit
```

As in Chapter 2, this program expects a file as input and will reject invalid or unreadable files. To illustrate this second point, create an empty file using `touch` and then use `chmod` (change mode) to set the permissions to `000` (all read/write/execute bits off):

```
$ touch cant-touch-this
$ chmod 000 cant-touch-this
```

Notice that the error message specifically tells me that I lack permission to read the file:

```
$ ./cgc.py cant-touch-this
usage: cgc.py [-h] FILE
```

```
cgc.py: error: argument FILE: can't open 'cant-touch-this': \
[Errno 13] Permission denied: 'cant-touch-this'
```

Now run the program with valid input and observe that the program prints the ID of the record having the highest percentage of GC:

```
$ ./cgc.py tests/inputs/1.fa
Rosalind_0808 60.919540
```

To get started, remove this program and start over:

```
$ new.py -fp 'Compute GC content' cgc.py
Done, see new script "cgc.py".
```

The following shows how to modify the first part of the program to accept a single positional argument that is a valid, readable file:

```
class Args(NamedTuple):
    """ Command-line arguments """
    file: TextIO ❶


# --------------------------------------------------
def get_args() -> Args:
    """ Get command-line arguments """

    parser = argparse.ArgumentParser(
        description='Compute GC content',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file', ❷
                        metavar='FILE',
                        type=argparse.FileType('rt'),
                        help='Input sequence file')

    args = parser.parse_args()

    return Args(args.file)
```

❶ The only attribute of the `Args` class is a filehandle.

❷ Create a positional *file* argument that must be a readable text file.

Your program should now recreate the preceding usage. Modify your `main()` to this:

```
def main() -> None:
    args = get_args()
    print(args.file.name)
```

Verify that it will print the name of the input file:

```
$ ./cgc.py tests/inputs/1.fa
tests/inputs/1.fa
```

Finally, run `pytest` to see how you're faring. You should pass the first three and fail on the fourth:

```
$ pytest -xv
============================= test session starts
==============================
....

tests/cgc_test.py::test_exists PASSED
[ 20%]
tests/cgc_test.py::test_usage PASSED
[ 40%]
tests/cgc_test.py::test_bad_input PASSED
[ 60%]
tests/cgc_test.py::test_good_input1 FAILED
[ 80%]

================================== FAILURES
==================================
_______________________________ test_good_input1
_______________________________

    def test_good_input1():
        """ Works on good input """

        rv, out = getstatusoutput(f'{RUN} {SAMPLE1}') ❶
```

```
        assert rv == 0
>       assert out == 'Rosalind_0808 60.919540' ❷
E       AssertionError: assert './tests/inputs/1.fa' == 'Rosalind_0808
60.919540'
E         - Rosalind_0808 60.919540 ❸
E         + ./tests/inputs/1.fa ❹

tests/cgc_test.py:48: AssertionError
========================== short test summary info
==========================
FAILED tests/cgc_test.py::test_good_input1 - AssertionError: assert
'./tes...
!!!!!!!!!!!!!!!!!!!!!! stopping after 1 failures
!!!!!!!!!!!!!!!!!!!!!!
======================== 1 failed, 3 passed in 0.34s
========================
```

❶ The test is running the program using the first input file.

❷ The output is expected to be the given string.

❸ This is the expected string.

❹ This is the string that was printed.

So far you have created a syntactically correct, well-structured, and documented program that validates a file input, all by doing relatively little work. Next you need to figure out how to find the sequence with the highest GC content.

Parsing FASTA using Biopython

You've already validated that the input to this program will be a file. Now let's talk about the content of that file. It should be sequence data in FASTA format which is a common way to represent biological sequences. Let's look at the first file to understand the format:

```
$ cat tests/inputs/1.fa
>Rosalind_6404 ❶
CCTGCGGAAGATCGGCACTAGAATAGCCAGAACCGTTTCTCTGAGGCTTCCGGCCTTCCC ❷
TCCCACTAATAATTCTGAGG
>Rosalind_5959
CCATCGGTAGCGCATCCTTAGTCCAATTAAGTCCCTATCCAGGCGCTCCGCCGAAGGTCT
ATATCCATTTGTCAGCAGACACGC
>Rosalind_0808
CCACCCTCGTGGTATGGCTAGGCATTCAGGAACCGGAGAACGCTTCAGACCAGCCCGGAC
TGGGAACCTGCGGGCAGTAGGTGGAAT
```

❶ A FASTA record starts with a `>` at the beginning of a line. The sequence ID is any following text up to a space.

❷ A sequence can be any length and can span multiple lines or be placed on a single line.

> **NOTE**
>
> The header of a FASTA file can get very ugly, very quickly. I would encourage you to find real sequence from from NCBI or look at the files in the *17_synth/tests/inputs* directory for more examples.

While it would be fun (for certain values of fun) to teach you how to manually parse this file, I'll go straight to using Biopython's `Bio.SeqIO` module:

```
>>> from Bio import SeqIO
>>> recs = SeqIO.parse('tests/inputs/1.fa', 'fasta') ❶
```

❶ The first argument is the name of the input file. As this function can parse many different record formats, the second argument is the format of the data.

What is the `type()` of `recs`?

```
>>> type(recs)
<class 'Bio.SeqIO.FastaIO.FastaIterator'>
```

I've shown iterators a couple of times now, even creating one in Chapter 4. In that exercise, I used the `next()` function to retrieve the next value from the Fibonacci sequence generator. I'll retrieve the first record and inspect its `type()`:

```
>>> rec = next(recs)
>>> type(rec)
<class 'Bio.SeqRecord.SeqRecord'>
```

To learn more about a sequence record, I highly recommend you read the SeqRecord documentation in addition to the documentation in the REPL using `help(rec)`. The data from the FASTA record must be parsed, which means discerning some meaning of the data from its syntax and structure. If you evaluate `rec` in the REPL, you'll see something that looks like a dictionary. This output is the same as `repr(seq)`, which is used to "return the canonical string representation of the object":

```
SeqRecord(

seq=Seq('CCTGCGGAAGATCGGCACTAGAATAGCCAGAACCGTTTCTCTGAGGCTTCCGGC...AGG'
), ❶
  id='Rosalind_6404', ❷
  name='Rosalind_6404', ❸
  description='Rosalind_6404',
  dbxrefs=[])
```

❶ The multiple lines of the sequence are concatenated into a single sequence represented by a `Seq` object.

❷ The *ID* of a FASTA record is all the characters in the header starting *after* the `>` and continuing up to the first space.

❸

> The `SeqRecord` object is meant to also handle data with more fields such as *name*, *description*, and database cross-references (*dbxrefs*). Since those fields are lacking in this FASTA record, the ID is duplicated for name and description, and the dbxrefs is the empty list.

If you `print()` the sequence, this information will be *stringified* so it's a little easier to read. This output is the same as `str(rec)`, which is meant to provide a useful string representation of an object:

```
>>> print(rec)
ID: Rosalind_6404
Name: Rosalind_6404
Description: Rosalind_6404
Number of features: 0
Seq('CCTGCGGAAGATCGGCACTAGAATAGCCAGAACCGTTTCTCTGAGGCTTCCGGC...AGG')
```

The most salient feature for the program is the record's sequence. You might expect this would be a `str`, but it's actually another object:

```
>>> type(rec.seq)
<class 'Bio.Seq.Seq'>
```

Use `help(rec.seq)` to see what attributes and methods the `Seq` object offers. I only want the DNA sequence itself which I can get by coercing the sequence to a string using the `str()` function:

```
>>> str(rec.seq)
'CCTGCGGAAGATCGGCACTAGAATAGCCAGAACCGTTTCTCTGAGGCTTCCGGCCTTCCCTCCCACTAA
TAATTCTGAGG'
```

Note this was the same class I used as the last solution of Chapter 3 to create a reverse complement. I can use it here like so:

```
>>> rec.seq.reverse_complement()
Seq('CCTCAGAATTATTAGTGGGAGGGAAGGCCGGAAGCCTCAGAGAAACGGTTCTGG...AGG')
```

The `Seq` object has many other useful methods, and I encourage you to explore the documentation as these can save you a lot of time[1]. At this point, you may feel you have enough information to finish the challenge. You need to iterate through all the sequences, determine what percentage of the bases are *G* or *C*, and return the ID and GC content of the record with the maximum value. I would challenge you to write a solution on your own. If you need more help, I'll show you one approach, and then I'll cover several variations in the solutions.

Iterating the Sequences Using a for Loop

Because I am using `argparse` to validate the file input argument, the `file` argument will be an open filehandle. Earlier I showed how the first argument to `SeqIO.parse()` was a filename, and it also works using a filehandle. Remember that the function also needs to know the format of the input file as the second argument:

```
>>> from Bio import SeqIO
>>> recs = SeqIO.parse(open('./tests/inputs/1.fa'), 'fasta')
```

I can use a `for` loop to iterate through each record to print the ID and the first 10 bases of the sequence:

```
>>> for rec in recs:
...     print(rec.id, rec.seq[:10])
...
Rosalind_6404 CCTGCGGAAG
Rosalind_5959 CCATCGGTAG
Rosalind_0808 CCACCCTCGT
```

Take a moment to run those lines again and notice that nothing will be printed:

```
>>> for rec in recs:
...     print(rec.id, rec.seq[:10])
...
```

Earlier I showed that `recs` is a `Bio.SeqIO.FastaIO.FastaIterator`, and, like all iterators, it will produce values until they exhausted. If you want to loop through the records again, you will need to recreate the `recs` object using the `SeqIO.parse()` function.

For the moment, assume the sequence is this:

```
>>> seq = 'CCACCCTCGTGGTATGGCT'
```

I need to find how many Cs and Gs occur in that string. I can use another `for` loop to iterate each base of the sequence and increment a counter whenever the base is a *G* or a *C*.

```
gc = 0 ❶
for base in seq: ❷
    if base in ('G', 'C'): ❸
        gc += 1 ❹
```

❶ Initialize a variable for the counts of the G/C bases.

❷ Iterate each base (character) in the sequence.

❸ See if the base is in the tuple containing *G* or *C*.

❹ Increment the GC counter.

To find the percentage of GC content, divide the GC count by the length of the sequence:

```
>>> gc
12
>>> len(seq)
19
>>> gc / len(seq)
0.631578947368421
```

The output from the program should be the ID of the sequence with the highest GC, a single space, and the GC content truncated to six significant digits. The easiest way to format the number is to learn more about `str.format()`. The `help` doesn't have much in the way of documentation, so I recommend you read PEP 3101 on advanced string formatting.

I've shown how I can use `{}` as placeholders for interpolating variables either using `str.format()` or f-strings. I can add formatting instructions after a colon (`:`) in the curly brackets. The syntax looks like that used with the `printf()` function in C-like languages, so `{:0.6f}` is a floating point number to 6 places:

```
>>> '{:0.6f}'.format(gc * 100 / len(seq))
'63.157895'
```

Or execute the code directly inside an f-string:

```
>>> f'{gc * 100 / len(seq):0.06f}'
'63.157895'
```

To figure out the sequence with the maximum GC, you have a couple of options, both of which I'll demonstrate in the solutions:

- Make a list of all the IDs and GC content (a list of tuples would serve well). Sort the list by GC content and take the maximum value.
- Keep track of the ID and GC content of the maximum value. Overwrite this when a new maximum is found.

I think that should be enough for you to finish a solution. You can do this. Fear is the mind-killer. Keep going until you pass *all* the tests, including those for linting and type checking. Your test output should look something like this:

```
$ make test
python3 -m pytest -xv --disable-pytest-warnings --flake8 --pylint --
mypy \
  cgc.py tests/cgc_test.py
============================= test session starts
==============================
...

cgc.py::FLAKE8 SKIPPED
[ 10%]
cgc.py::mypy PASSED
[ 20%]
tests/cgc_test.py::FLAKE8 SKIPPED
[ 30%]
tests/cgc_test.py::mypy PASSED
[ 40%]
tests/cgc_test.py::test_exists PASSED
[ 50%]
tests/cgc_test.py::test_usage PASSED
[ 60%]
tests/cgc_test.py::test_bad_input PASSED
[ 70%]
tests/cgc_test.py::test_good_input1 PASSED
[ 80%]
tests/cgc_test.py::test_good_input2 PASSED
[ 90%]
::mypy PASSED
[100%]
=================================== mypy
====================================

Success: no issues found in 2 source files
======================== 8 passed, 2 skipped in 1.60s
========================
```

Solutions

As before, all the solutions share the same `get_args()`, so only the differences will be shown.

Solution 1: Using a List

Let's look at my first solution. I always try to start with the most obvious and simple way, and you'll find this is often the most verbose. Once you understand the logic, I hope you'll be able to follow more powerful and terse ways to express the same ideas. For this first solution, be sure to also import `List` and `Tuple` from the `typing` module:

```
def main() -> None:
    args = get_args()
    seqs: List[Tuple[float, str]] = [] ❶

    for rec in SeqIO.parse(args.file, 'fasta'): ❷
        gc = 0 ❸
        for base in rec.seq.upper(): ❹
            if base in ('C', 'G'): ❺
                gc += 1 ❻
        pct = (gc * 100) / len(rec.seq) ❼
        seqs.append((pct, rec.id)) ❽

    high = max(seqs) ❾
    print(f'{high[1]} {high[0]:0.6f}') ❿
```

❶ Initialize an empty list to hold the GC content and sequences IDs as tuples.

❷ Iterate through each record in the input file.

❸ Initialize a GC counter.

❹ Iterate through each base uppercased sequence, guarding against possible mixed-case input.

❺ Check if the base is a *C* or *G*.

❻ Increment the GC counter.

❼ Calculate the GC content.

❽ Append a new tuple of the GC content and the sequence ID.

❾ Take the maximum value.

❿ Print the sequence ID and GC content of the highest value.

> **NOTE**
>
> The type annotation `List[Tuple[float, str]]` on the `seqs` variable provides not only a way to programmatically check the code using tools like `mypy` but also an added layer of documentation. The reader of this code doesn't have to jump ahead to see what kind of data will be added to the list because it has been explicitly described using types.

In this solution, I decided to make a list of all the IDs and GC percentages mostly so that I could show you how to create a list of tuples. Then I wanted to point out a few magical properties of Python's sorting. Let's start with the `sorted()` function, which works on strings as you might imagine:

```
>>> sorted(['McClintock', 'Curie', 'Doudna', 'Charpentier'])
['Charpentier', 'Curie', 'Doudna', 'McClintock']
```

When all the values are numbers, then they will be sorted numerically, so I've got that going for me, which is nice:

```
>>> sorted([2, 10, 1])
[1, 2, 10]
```

Note that those same values *as strings* will sort in lexicographic order:

```
>>> sorted(['2', '10', '1'])
['1', '10', '2']
```

PYTHON LISTS SHOULD BE HOMOGENEOUS

Comparing different types such as strings and integers will cause an exception:

```
>>> sorted([2, '10', 1])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'str' and 'int'
```

While it's an acceptable practice in Python to mix types in a list, it's likely to end in tears. This reminds me of a joke by Henny Youngman: A man goes to see his doctor. He says, "Doc, it hurts when I do this." The doctor says, "Then don't do that."

Essentially Python is a bit like the doctor here in that, yeah, mixing types in lists will lead to runtime exceptions if you try to sort them, so just don't do that. To avoid this, always use a type declaration to describe your data:

```
seqs: List[Tuple[float, str]] = []
```

Try adding this line:

```
seqs.append('foo')
```

Then check the program using `mypy` to see how clearly the error message shows how this violates the description of the data:

```
$ mypy solution1_list.py
solution1_list.py:38: error: Argument 1 to "append" of "list" has
incompatible type "str"; expected "Tuple[float, str]"
Found 1 error in 1 file (checked 1 source file)
```

Another option is to use the `numpy` module to create an *array* which is like a Python list but all the values are required (and coerced) to be of a common type. Arrays in `numpy` are both faster

(due to memory management) and safer than Python lists. Note that mixing strings and numbers would result in a list of strings:

```
>>> import numpy as np
>>> nums = np.array([2, '10', 1])
>>> nums
array(['2', '10', '1'], dtype='<U21')
```

Although this is not correct — I want them to be integers — it is at least safe. I can use the `int()` function in a list comprehension to coerce all the values to integers:

```
>>> [int(n) for n in [2, '10', 1]]
[2, 10, 1]
```

Or the same thing expressed using the higher-order function `map()` which takes a function like `int()` as the first argument and applies it to all the elements in the sequence to return a new list of elements transformed by that function:

```
>>> list(map(int, [2, '10', 1]))
[2, 10, 1]
```

Which makes for good sorting:

```
>>> sorted(map(int, [2, '10', 1]))
[1, 2, 10]
```

Now consider a list of tuples where the first element is a `float` and the second element is a `str`. How will `sorted()` handle this? By first sorting all the data by the first elements *numerically* and then the second elements *lexicographically*:

```
>>> sorted([(0.2, 'foo'), (.01, 'baz'), (.01, 'bar')])
[(0.01, 'bar'), (0.01, 'baz'), (0.2, 'foo')]
```

Structuring `seqs` as `List[Tuple[float, str]]` takes advantage of this built-in behavior of `sorted()`, allowing me to quickly sort the sequences by GC content and select the highest value:

```
>>> high = sorted(seqs)[-1]
```

This is the same as finding the highest value, which the `max()` function can do more easily:

```
>>> high = max(seqs)
```

The `high` is a tuple where the 1st position is the sequence ID and the 0th position is the GC content which needs to be formatted:

```
print(f'{high[1]} {high[0]:0.6f}')
```

Solution 2: Type Annotations and Unit Tests

Hidden inside the `for` loop is a kernel of code to compute GC that needs to be extracted into a function with a test. The following the ideas of test-driven development (TDD), I will first define a `find_gc()` function:

```
def find_gc(seq: str) -> float: ❶
    """ Calculate GC content """

    return 0. ❷
```

❶ The function accepts a `str` and returns a `float`.

❷ For now, return 0. Note the trailing `.` tells Python this is a `float`. This is shorthand for `0.0`.

Next, I'll define a function that will serve as a unit test. Since I'm using `pytest`, this function's name must start with `test_`. Since I'm testing the `find_gc()` function, I like to name the function

`test_find_gc`. I will use a series of `assert` statements to test if the function returns the expected result for a given input. Note how this test function serves both as a formal test and as an additional piece of documentation as the reader can clearly see the inputs and outputs:

```
def test_find_gc():
    """ Test find_gc """

    assert find_gc('') == 0. ❶
    assert find_gc('C') == 100. ❷
    assert find_gc('G') == 100. ❸
    assert find_gc('CGCCG') == 100. ❹
    assert find_gc('ATTAA') == 0.
    assert find_gc('ACGT') == 50.
```

❶ If a function accepts a `str`, I always start by testing with the empty string to make sure it returns something useful.

❷ A single *C* should be 100% GC.

❸ Same for a single *G*.

❹ Various other tests mixing bases at various percentages.

It's rarely possible to exhaustively check every possible input to a function, so I often rely on spot-checking. Note that the `hypothesis` module can generate random values for testing. Presumably, the `find_gc()` function is simple enough that these tests are sufficient. My goal in writing functions is to make them as simple as possible, but no simpler.

> *There are two ways to write code: write code so simple there are obviously no bugs in it, or write code so complex that there are no obvious bugs in it.*
>
> —Tony Hoare

The `find_gc()` and `test_find_gc()` functions are inside the `cgc.py` program, not in the *tests/cgc_test.py* module. To execute the unit test, I run `pytest` on the source code *expecting the test to fail*:

```
$ pytest -v cgc.py
============================= test session starts
============================
...

cgc.py::test_find_gc FAILED
[100%] ❶

================================== FAILURES
==================================
_____________________________________ test_gc
_____________________________________

    def test_find_gc():
        """ Test find_gc """

        assert find_gc('') == 0. ❷
>       assert find_gc('C') == 100. ❸
E       assert 0 == 100.0
E         +0
E         -100.0

cgc.py:74: AssertionError
========================== short test summary info
==========================
FAILED cgc.py::test_gc - assert 0 == 100.0
============================== 1 failed in 0.32s
==============================
```

❶ The unit test fails as expected.

❷ The first test passes because it was expecting 0.

❸ This test fails because it should have return 100.

Now I have established a baseline from which I can proceed. I know that my code fails to meet some expectation as formally defined

using a test. To fix this, I move all the relevant code from the `main()` into the function:

```
def find_gc(seq: str) -> float:
    """ Calculate GC content """

    if not seq: ❶
        return 0 ❷

    gc = 0 ❸
    for base in seq.upper():
        if base in ('C', 'G'):
            gc += 1

    return (gc * 100) / len(seq)
```

❶ This guards against trying to divide by 0 when the sequence is the empty string.

❷ If there is no sequence, the GC content is 0.

❸ This is the same code as before.

Run `pytest` again to check that the function works:

```
$ pytest -v cgc.py
============================= test session starts
==============================
...

cgc.py::test_gc PASSED
[100%]

============================== 1 passed in 0.30s
==============================
```

This is test-driven development:

- Define a function to test

- Write the test
- Ensure the function fails the test
- Make the function work
- Ensure the function passes the test (and all your previous tests still pass)

If you later encounter sequences that trigger bugs in your code, you fix the code and add those as more tests. You shouldn't have to worry about weird cases like the `find_gc()` function receiving a `None` or a list of integers *because I used type annotations*. Testing is useful. Type annotations are useful. Combining tests and types leads to code that is easier to verify and comprehend.

I want to make one other addition to this solution by adding a custom type to document the tuple holding the GC content and sequence ID. I'll call it `MySeq` just to avoid any confusion with the `Bio.Seq` class. Add this below the `Args` definition:

```
class MySeq(NamedTuple):
    """ Sequence """
    gc: float ❶
    name: str ❷
```

❶ The GC content is a percentage.

❷ I would prefer to use the field name *id*, but that conflicts with the `id()` *identity* function which is built-in to Python.

Here is how it can be incorporated into the code:

```
def main() -> None:
    args = get_args()
    seqs: List[MySeq] = [] ❶

    for rec in SeqIO.parse(args.file, 'fasta'):
        seqs.append(MySeq(find_gc(rec.seq), rec.id)) ❷
```

```
    high = sorted(seqs)[-1] ❸
    print(f'{high.name} {high.gc:0.6f}') ❹
```

❶ Use `MySeq` as a type annotation.

❷ Create `MySeq` using the return from the `find_gc()` function and the record ID.

❸ This still works because `MySeq` is a tuple.

❹ Use the field access rather than the index position of the tuple.

This version of the program is arguably easier to read. You can and should create as many custom types as you want to better document and test your code.

Solution 3: Keeping a Running Max Variable

The previous solution works well, but it's a bit verbose and needlessly keeps track of *all* the sequences when I actually only care about the maximum value. Given how small the test inputs are, this will never be a problem, but bioinformatics is always about scaling up. A solution that tries to store all the sequences will eventually choke. Consider processing 1 million sequences, or 1 billion, or 100 billion. Eventually, I'd run out of memory.

Here's a solution that would scale to any number of sequences as it only ever allocates a single tuple to remember the highest value:

```
def main():
    args = get_args()
    high = MySeq(0., '') ❶

    for rec in SeqIO.parse(args.file, 'fasta'):
        pct = find_gc(rec.seq) ❷
        if pct > high.gc: ❸
            high = MySeq(pct, rec.id) ❹
```

```
    print(f'{high.name} {high.gc:0.6f}') ❺
```

❶ Initialize a variable to remember the highest value. Type annotation is superfluous as `mypy` will expect this variable to remain this type forever.

❷ Calculate the GC content.

❸ See if the percent GC is greater than the highest value.

❹ If so, overwrite the highest value using this percent GC and sequence ID.

❺ Print the highest value.

For this solution, I also took a slightly different approach to compute the GC:

```
def find_gc(seq: str) -> float:
    """ Calculate GC content """

    return (seq.upper().count('C') + ❶
            seq.upper().count('G')) * 100 / len(seq) if seq else 0 ❷
```

❶ Use the `str.count()` method to find the Cs and Gs in the sequence.

❷ Since there are two conditions for the state of the sequence — the empty string or not — I prefer to write a single `return` using an if-expression.

I'll benchmark the last solution against this one. First I need to generate an input file with a significant number of sequences, say 10K. In the *05_gc* directory, you'll find a `genseq.py` file similar to the one I used in the *02_rna* directory. This one generates a FASTA file:

```
$ ./genseq.py -h
usage: genseq.py [-h] [-l int] [-n int] [-s sigma] [-o FILE]

Generate long sequence

optional arguments:
  -h, --help            show this help message and exit
  -l int, --len int     Average sequence length (default: 500)
  -n int, --num int     Number of sequences (default: 1000)
  -s sigma, --sigma sigma
                        Sigma/STD (default: 0.1)
  -o FILE, --outfile FILE
                        Output file (default: seqs.fa)
```

Here's how I'll generate an input file:

```
$ ./genseq.py -n 10000 -o 10K.fa
Wrote 10,000 sequences of avg length 500 to "10K.fa".
```

I can use that with `hyperfine` to compare these two implementations:

```
$ hyperfine -L prg ./solution2_unit_test.py,./solution3_max_var.py
'{prg} 10K.fa'
Benchmark #1: ./solution2_unit_test.py 10K.fa
  Time (mean ± σ):      1.546 s ±  0.035 s    [User: 2.117 s, System:
0.147 s]
  Range (min … max):    1.511 s …  1.625 s    10 runs

Benchmark #2: ./solution3_max_var.py 10K.fa
  Time (mean ± σ):     368.7 ms ±   3.0 ms    [User: 957.7 ms, System:
137.1 ms]
  Range (min … max):   364.9 ms … 374.7 ms    10 runs

Summary
  './solution3_max_var.py 10K.fa' ran
    4.19 ± 0.10 times faster than './solution2_unit_test.py 10K.fa'
```

It would appear that the third solution is about four times faster than the second running on 10K sequences. You can try generating more and longer sequences for your own benchmarking. I would definitely

recommend you create a file with at least 1 million sequences and compare your first solution with this version.

Solution 4: Using a List Comprehension with a Guard

Figure 5-1 shows that another way to find all the Cs and Gs in the sequence is to use a list comprehension and the compound `if` comparison from the first solution which is called a *guard*:

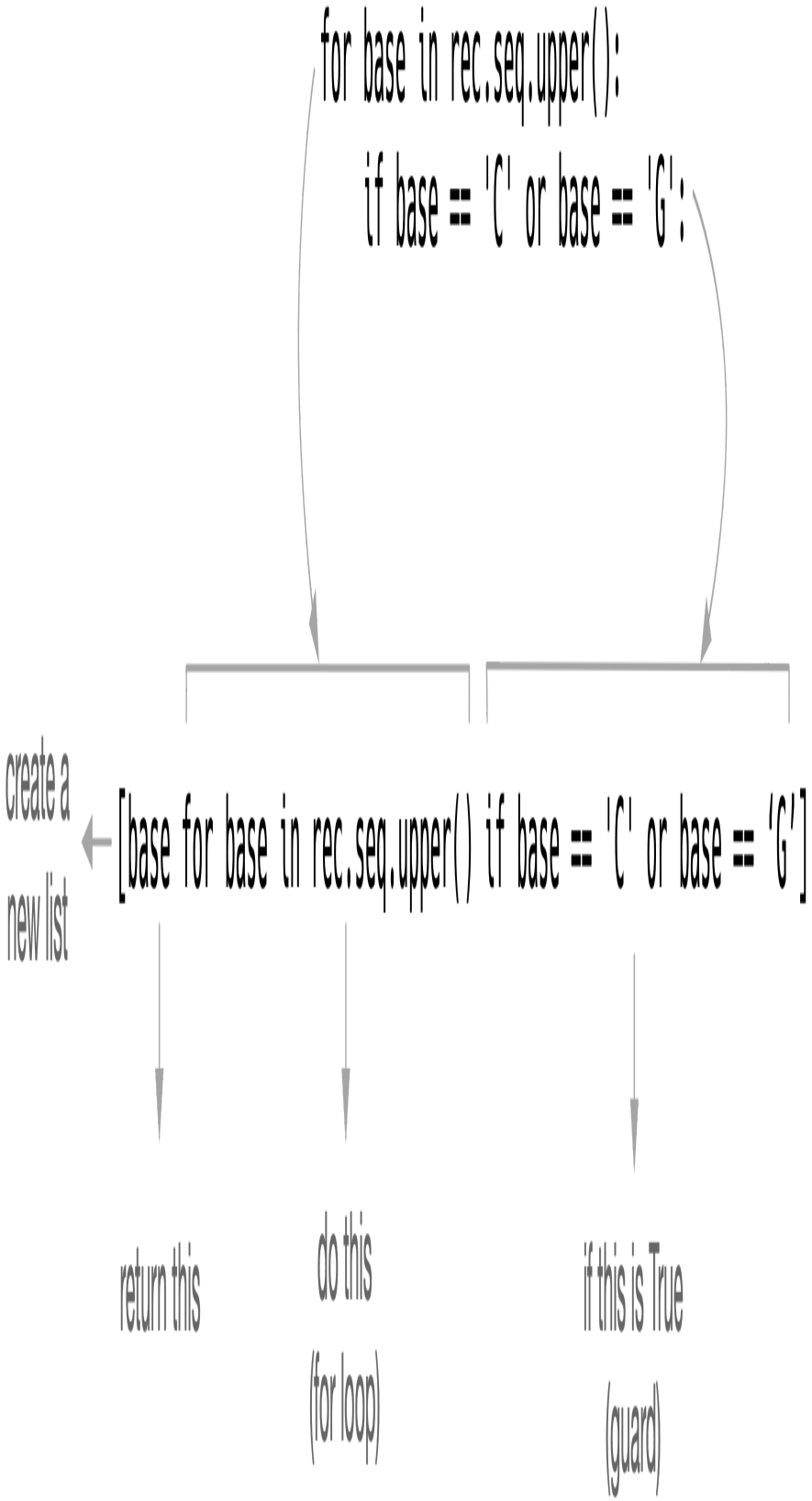

for base in rec.seq.upper():
if base == 'C' or base == 'G':
create a new list
[base for base in rec.seq.upper() if base == 'C' or base == 'G']
return this
do this (for loop)
if this is True (guard)

Figure 5-1. A list comprehension with a guard will select only those elements returning a truthy value for the `if` expression.

The list comprehension only yields those elements passing the guard test which I can further shorten by checking if the `base` is in the string `'CG'`:

```
>>> gc = [base for base in 'CCACCCTCGTGGTATGGCT' if base in 'CG']
>>> gc
['C', 'C', 'C', 'C', 'C', 'C', 'G', 'G', 'G', 'G', 'G', 'C']
```

Since the result is a new list, I can use the `len()` function to find how many Cs and Gs are present:

```
>>> len(gc)
12
```

I incorporate this idea into the `find_gc()` function:

```
def find_gc(seq: str) -> float:
    """ Calculate GC content """

    if not seq:
        return 0

    gc = len([base for base in seq.upper() if base in 'CG']) ❶
    return (gc * 100) / len(seq)
```

❶ Another way to count the Cs and Gs is to select them using a list comprehension with a guard.

Solution 5: Using the filter Function

The idea of list comprehension with a guard can be expressed with the higher-order function `filter()`. Earlier in the chapter I used the `map()` function to apply the `int()` function to all the elements of a list to produce a new list of integers. The `filter()` function works

similarly, accepting a function as the first argument and an iterable as the second. It's different, though, as only those elements returning a truthy value when the function is applied will be returned. As this is a lazy function, I will need to coerce with `list()` in the REPL:

```
>>> list(filter(lambda base: base in 'CG', 'CCACCCTCGTGGTATGGCT'))
['C', 'C', 'C', 'C', 'C', 'C', 'G', 'G', 'G', 'G', 'G', 'C']
```

So here's another way to express the same idea from the last solution:

```
def find_gc(seq: str) -> float:
    """ Calculate GC content """

    if not seq:
        return 0

    gc = len(list(filter(lambda base: base in 'CG', seq.upper()))) ❶
    return (gc * 100) / len(seq)
```

❶ Use a `filter()` to select only those bases matching C or G.

Solution 6: Using the map Function and Summing Booleans

The `map()` function is a favorite of mine, so I want to show another way to use it. I could use `map()` to turn each base into a 1 if it's a C or G and 0 otherwise:

```
>>> seq = 'CCACCCTCGTGGTATGGCT'
>>> list(map(lambda base: 1 if base in 'CG' else 0, seq))
[1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0]
```

Counting the Cs and Gs would then be a matter of summing this list which I can do using the `sum()` function:

```
>>> sum(map(lambda base: 1 if base in 'CG' else 0, seq))
12
```

BOOLEANS ARE REALLY INTEGERS

Python can use Boolean algebra to combine values like `True`/`False` using `and`/`or`:

```
>>> True and False
False
>>> True or False
True
```

What do you think happens if you use `+` instead of `and`?

```
>>> True + True
2
>>> True + False
1
```

It turns out that Boolean values are actually integers under the hood where `True` is 1 and `False` is 0.

I can shorten my `map()` to return the result of the comparison which is a `bool` (but secretly an `int`) and `sum()` that instead:

```
>>> sum(map(lambda base: base in 'CG', seq))
12
```

Here is how I could incorporate this idea:

```
def find_gc(seq: str) -> float:
    """ Calculate GC content """

    if not seq:
        return 0

    gc = sum(map(lambda base: base in 'CG', seq.upper())) ❶
    return (gc * 100) / len(seq)
```

❶ Transform the sequence into Boolean values based on their comparison to the bases *C* or *G*, then sum the `True` (1) values to get a count.

Solution 7: Using Regular Expressions to Find Patterns

So far I've been showing you multiple ways to manually iterate a sequence of characters in a string to pick out those matching *C* or *G*. This is pattern matching, and it's precisely what regular expressions will do. The cost to you is learning another domain-specific language (DSL), but this is well worth the effort as regexes are widely used outside of Python. Begin by importing the `re` module:

```
>>> import re
```

You should read `help(re)` as this is a fantastically useful module. I want to use the `re.findall()` function to find all occurrences of a pattern in a string. I can create a *character class* pattern for the regex engine by using square brackets to enclose any characters I want to include. The class `[GC]` means *match either G or C*:

```
>>> re.findall('[GC]', 'CCACCCTCGTGGTATGGCT')
['C', 'C', 'C', 'C', 'C', 'C', 'G', 'G', 'G', 'G', 'G', 'C']
```

As before, I can use the `len()` function to find how many Cs and Gs there are. The following code shows how I would incorporate this into my function. Note that I use if-expression to return 0 if the sequence is the empty string so I can avoid division when `len(seq)` is 0:

```
def find_gc(seq: str) -> float:
    """ Calculate GC content """

    return len(re.findall('[GC]', seq.upper()) * 100) / len(seq) if
seq else 0
```

Note that it's important to change how this function is called from the `main()` so as to explicitly coerce the `rec.seq` value (which is a `Seq` object) to a string by using `str()`:

```
def main() -> None:
    args = get_args()
    high = MySeq(0., '')

    for rec in SeqIO.parse(args.file, 'fasta'):
        pct = find_gc(str(rec.seq)) ❶
        if pct > high.gc:
            high = MySeq(pct, rec.id)

    print(f'{high.name} {high.gc:0.6f}')
```

❶ Coerce the sequence to a string value or the `Seq` object will be passed.

Solution 8: A More Complex find_gc Function

In this final solution, I'll move almost all of the code from `main()` into the `find_gc()` function. I want the function to accept a `SeqRecord` object rather than a string of the sequence, and I want it to return the `MySeq` tuple:

First I'll change the test:

```
def test_find_gc() -> None:
    """ Test find_gc """

    assert find_gc(SeqRecord(Seq(''), id='123')) == (0.0, '123')
    assert find_gc(SeqRecord(Seq('C'), id='ABC')) == (100.0, 'ABC')
    assert find_gc(SeqRecord(Seq('G'), id='XYZ')) == (100.0, 'XYZ')
    assert find_gc(SeqRecord(Seq('ACTG'), id='ABC')) == (50.0, 'ABC')
    assert find_gc(SeqRecord(Seq('GGCC'), id='XYZ')) == (100.0, 'XYZ')
```

These are essentially the same tests as before, but I'm now passing `SeqRecord` objects. To make this work in the REPL, you will need to import a couple of classes:

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> seq = SeqRecord(Seq('ACTG'), id='ABC')
```

If you look at the object, it looks similar enough to the data I've been reading from input files because I only care about the *seq* field:

```
SeqRecord(seq=Seq('ACTG'),
  id='ABC',
  name='<unknown name>',
  description='<unknown description>',i
  dbxrefs=[])
```

If you run `pytest`, your `test_find_gc()` function should fail because you haven't yet changed your `find_gc()` function. Here's how I wrote it:

```
def find_gc(rec: SeqRecord) -> MySeq: ❶
    """ Return the GC content, record ID for a sequence """

    pct = 0. ❷
    if seq := str(rec.seq): ❸
        gc = len(re.findall('[GC]', seq.upper())) ❹
        pct = (gc * 100) / len(seq)

    return MySeq(pct, rec.id) ❺
```

❶ The function accepts a `SeqRecord` and return a `MySeq`.

❷ Initialize this to a floating point 0.

❸ This syntax is new as of Python 3.8 and allows variable assignment (first) and testing (second) in one line using `:=` assignment.

❹ This is the same code as before.

❺ Return a `MySeq` object.

NOTE

The *walrus operator* `:=` was proposed in PEP 572 which notes that the `=` operator allows one to name the result of an expression only in a statement form, “making it unavailable in list comprehensions and other expression contexts.” This new operator combines two actions, that of *assigning* the value of an expression to a variable and then *evaluating* that variable. In the preceding code, the `seq` is assigned the value of the stringified sequence. If that evaluates to something truthy such as a non-empty string, then the following block of code will be executed.

This radically changes the `main()` function. The `for` loop can incorporate a `map()` function to turn each `SeqRecord` into a `MySeq`:

```
def main() -> None:
    args = get_args()
    high = MySeq(0., '') ❶
    for seq in map(find_gc, SeqIO.parse(args.file, 'fasta')): ❷
        if seq.gc > high.gc: ❸
            high = seq ❹

    print(f'{high.name} {high.gc:0.6f}') ❺
```

❶ Initialize the high variable.

❷ Use `map()` to turn each `SeqRecord` into a `MySeq`.

❸ Compare the current sequence’s GC content against the running high.

❹ Overwrite the value.

❺ Print the results.

The point of the expanded `find_gc()` function was to hide more of the guts of the program so I can write a more expressive program.

You may disagree, but I think this is the most readable version of the program.

Benchmarking

So which one is the winner? There is a `bench.sh` program that will run `hyperfine` on all the *solution*.py* with the *seqs.fa* file. Here is the result:

```
Summary
  './solution3_max_var.py seqs.fa' ran
    2.15 ± 0.03 times faster than './solution8_list_comp_map.py
seqs.fa'
    3.88 ± 0.05 times faster than './solution7_re.py seqs.fa'
    5.38 ± 0.11 times faster than './solution2_unit_test.py seqs.fa'
    5.45 ± 0.18 times faster than './solution4_list_comp.py seqs.fa'
    5.46 ± 0.14 times faster than './solution1_list.py seqs.fa'
    6.22 ± 0.08 times faster than './solution6_map.py seqs.fa'
    6.29 ± 0.14 times faster than './solution5_filter.py seqs.fa'
```

Going Further

Try writing your own FASTA parser. Create a new directory called *faparser*:

```
$ mkdir faparser
```

Change into that directory, and run `new.py` with the `-t|--write_test` option:

```
$ cd faparser/
$ new.py -t faparser.py
Done, see new script "faparser.py".
```

You should now have a structure that includes a *tests* directory along with the a starting test file:

```
$ tree
.
├── Makefile
├── faparser.py
└── tests
    └── faparser_test.py

1 directory, 3 files
```

You can run `make test` or `pytest` to verify that everything at least runs. Copy the *tests/inputs* directory from the *05_gc* to the new *tests* directory so you have some test input files. Now consider how you want your new program to work. I would imagine it would take one (or more) readable text files as inputs, so you could define your arguments accordingly. Then what will your program do with the data? Do you want it to print, for instance, the IDs and the length of each sequence? Now write the tests and codes to manually parse the input FASTA files and print the output. Challenge yourself.

Review

- The `Bio.SeqIO.parse()` function will parse FASTA-formatted sequences files into records which give us access to the record's ID and sequence.
- You can use several constructs to visits all the elements of iterables including `for` loops, list comprehensions, and the `filter()` and `map()` functions.
- Input files can get very large in bioinformatics, so it's often important to avoid writing algorithms that read in entire files such as sequence files.
- The `sorted()` function will sort homogeneous lists of strings and numbers lexicographically and numerically, respectively. It can also sort homegeneous lists of tuples using each position of the tuples in order.

- Formatting templates for strings can include `printf()`-like instructions to control how output values are presented.
- The `sum()` function will add a list of numbers.
- Regular expressions can find patterns of text in a strings.

---

1 “Weeks of coding can save you hours of planning.” — Unknown

Chapter 6. Finding the Hamming Distance: Counting Point Mutations

The Hamming distance (named after the same Richard Hamming mentioned in the preface) is the number of edits required to change one string into another. It's one metric for gauging sequence similarity. I have written a couple of other metrics for this, starting in Chapter 1 by counting all the bases and continuing in Chapter 5 by calculation GC content. While GC content can be practically informative as coding regions tend to be GC-rich, tetranucleotide frequency falls pretty short of being useful. For example, the sequences *AAACCCGGGTTT* and *CGACGATATGTC* are wildly different yet produce the same signature:

```
$ ./dna.py AAACCCGGGTTT
3 3 3 3
$ ./dna.py CGACGATATGTC
3 3 3 3
```

Taken alone, tetranucleotide frequency makes these sequences seem identical, but it's quite obvious that they would produce entirely different protein sequences and so would likely be functionally unlike. Figure 6-1 depicts an alignment of the two sequences indicating that only 3 of the 12 bases are shared, meaning they are only 25% similar.

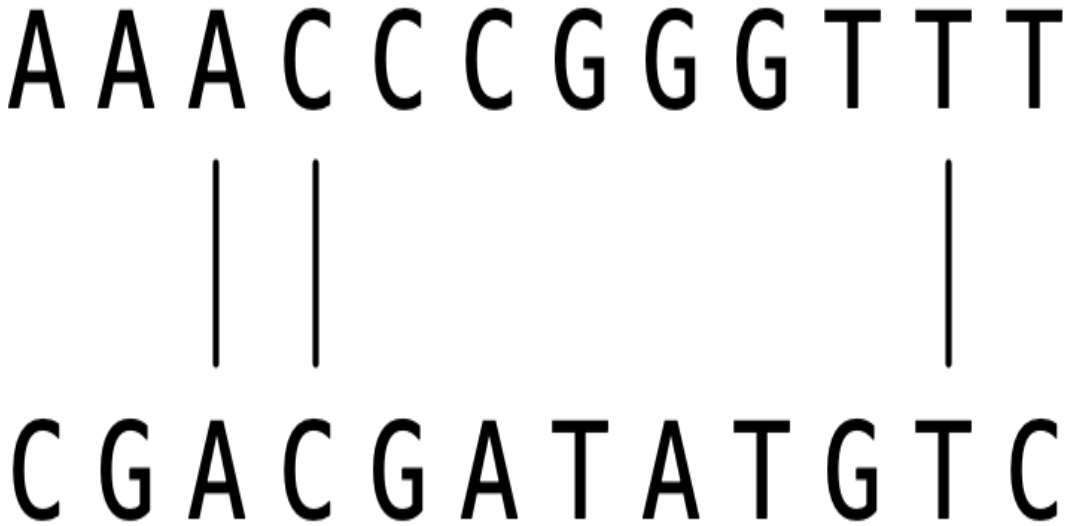


Figure 6-1. An alignment of two sequences with vertical bars showing matching bases.

By contrast, the Hamming distance between these sequences is 9, meaning 9 bases would need to be changed to transform one sequence to the other. The Hamming distance is basically equivalent in bioinformatics to single-nucleotide polymorphisms (SNPs, pronounced *snips*) or single-nucleotide variations (SNVs, pronounced *snivs*). Note that this algorithm falls far short of something like sequence alignment that can identify insertions and deletions as it only accounts for the change of one base to another value. For instance, Figure 6-2 shows that the sequences *AAACCCGGGTTT* and *AACCCGGGTTTA* are 92% similar when aligned (on the left) as they differ by a single base. The Hamming distance (on the right), though, shows only 4 bases are in common which means they are only 33% similar.



Figure 6-2. The alignment of these sequences shows them to be nearly identical while the Hamming distance finds they're only 33% similar.

This program will always compare strings strictly from their beginnings which limits the practical application to real-world bioinformatics. Still, it turns out that this naïve algorithm is a useful metric for sequence similarity, and writing the implementation presents many interesting solutions in Python.

In this chapter, you will learn:

- How to use the `abs()` and `min()` functions
- How to combine the elements from two lists of possibly unequal lengths
- How to use a list comprehension with a guard and how this relates to `filter()`
- How to write `map()` using `lambda` or existing functions
- How to use functions from the `operator` module
- How to use the `itertools.starmap()` function

Getting Started

You should work in the *06_hamm* directory of the repository. I suggest you start by getting a feel for how the solutions work, so copy one of them to the `hamm.py` program and request the help:

```
$ cp solution1_abs_iterate.py hamm.py
$ ./hamm.py -h
usage: hamm.py [-h] FILE

Hamming distance

positional arguments:
  FILE        File input

optional arguments:
  -h, --help  show this help message and exit
```

The program wants a file as input, so you can safely assume that it's using the `argparse.FileType`. The structure of this file should look like the inputs shown on the Rosalind page, two sequences separated by a newline:

```
$ cat tests/inputs/1.txt
GAGCCTACTAACGGGAT
CATCGTAATGACGGCCT
```

Run the program with this input and verify the answer is 7:

```
$ ./hamm.py tests/inputs/1.txt
7
```

Run the tests (either with `pytest` or `make test`) to see a passing suite. Once you feel you understand what's expected, remove this file and start from scratch:

```
$ new.py -fp 'Hamming distance' hamm.py
Done, see new script "hamm.py".
```

Define the parameters so that the program requires a single, positional file argument:

```
import argparse
from typing import NamedTuple, TextIO


class Args(NamedTuple):
    """ Command-line arguments """
    file: TextIO


# --------------------------------------------------
def get_args() -> Args:
    """ Get command-line arguments """

    parser = argparse.ArgumentParser(
        description='Hamming distance',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file',
                        help='Input file',
                        metavar='FILE',
                        type=argparse.FileType('rt'))

    args = parser.parse_args()

    return Args(args.file)
```

Change the `main()` to print the contents of the input file. Remember that `args.file` is an open filehandle, so I can call `args.file.read()` to read the entire contents. Since the file ends in a newline, I'll tell `print()` to leave off its own newline:

```
def main() -> None:
    args = get_args()
    print(args.file.read(), end='')
```

By this point, you should have a program that prints the usage, validates the file input, and prints the file contents:

```
$ ./hamm.py tests/inputs/1.txt
GAGCCTACTAACGGGAT
CATCGTAATGACGGCCT
```

If you run `pytest`, you should find that the program passes the first three tests. It should fail the `test_input1` with a message like the following:

```
=================================== FAILURES
===================================
_____________________________________ test_input1
_____________________________________

    def test_input1() -> None:
        """ Test with input1 """

>       run(INPUT1, '7')

tests/hamm_test.py:58:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _

file = './tests/inputs/1.txt', expected = '7'

def run(file: str, expected: str) -> None:
        """ Run with input """

        rv, out = getstatusoutput(f'{RUN} {file}') ❶
        assert rv == 0
>       assert out.rstrip() == expected ❷
E       AssertionError: assert 'GAGCCTACTAAC...GTAATGACGGCCT' == '7' ❸
E         - 7
E         + GAGCCTACTAACGGGAT
E         + CATCGTAATGACGGCCT
```

❶ The program is being run with the input file *./tests/inputs/1.txt*.

❷ This is the line in the test where it failed.

❸ Specifically the program printed the file contents but should have printed *7*.

Reading the Input

To start, I want to separate the two sequences into two variables. This is one approach:

```
def main():
    args = get_args()
    seq1, seq2 = args.file.read().splitlines()[:2] ❶
    print(seq1, seq2)
```

❶ Read the contents, split on newlines into a list, slice the first two, unpack into two variables.

To understand how this works, consider reading the first input file:

```
>>> fh = open('./tests/inputs/1.txt')
>>> contents = fh.read()
>>> contents
'GAGCCTACTAACGGGAT\nCATCGTAATGACGGCCT\n'
```

Note that the filehandle has been exhausted from the call to `fh.read()`:

```
>>> fh.read()
''
```

This is why I save the contents to a variable. To read the handle again, I can `seek()` to move the stream position back to the beginning:

```
>>> fh.seek(0)
0
>>> fh.read()
'GAGCCTACTAACGGGAT\nCATCGTAATGACGGCCT\n'
```

The `str.splitlines()` function will break contents on newlines into a list, and I can use slice notation to ensure I'm getting only the first

two elements from the list:

```
>>> contents.splitlines()[:2]
['GAGCCTACTAACGGGAT', 'CATCGTAATGACGGCCT']
```

Then unpack these two values into two variables:

```
>>> seq1, seq2 = contents.splitlines()[:2]
>>> seq1, seq2
('GAGCCTACTAACGGGAT', 'CATCGTAATGACGGCCT')
```

Many problems lie in wait. For instance, what if the input file is empty? Everything is OK up to the call to `str.splitlines()` as this will return an empty list:

```
>>> open('./tests/inputs/empty.txt').read().splitlines()
[]
```

Python slices using ranges will return an empty list for a non-existent slice range:

```
>>> open('./tests/inputs/empty.txt').read().splitlines()[:2]
[]
```

but this raises an exception when I try to assign an empty list to two variables:

```
>>> seq1, seq2 = open('./tests/inputs/empty.txt').read().splitlines()
[:2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not enough values to unpack (expected 2, got 0)
```

The code works well enough on the first input file:

```
$ ./hamm.py tests/inputs/1.txt
GAGCCTACTAACGGGAT CATCGTAATGACGGCCT
```

but the uncaught exception rears its ugly head on the empty file:

```
$ ./hamm.py tests/inputs/empty.txt
Traceback (most recent call last):
  File "./hamm.py", line 48, in <module>
    main()
  File "./hamm.py", line 37, in main
    seq1, seq2 = args.file.read().splitlines()[:2]
ValueError: not enough values to unpack (expected 2, got 0)
```

While you might understand the error message, assume you've given this program to a colleague who does not know Python. Would they understand how to fix the problem? Technically the error message is explicit — it wanted to unpack two values but got none — but this would be well nigh unintelligible to anyone but a fairly intermediate Python programmer.

NOTE

The root of the problem here is that I chained several operations that had underlying assumptions about the data that are violated when using the empty file. Nothing in Python forces one to confront the fact that the read operation might fail, that splitlines might return nothing, that assigning a list slice might fail. All these complexities fall to the programmer's shoulders in Python, which is why I have to use extreme caution. Try this kind of operation in Haskell or Rust for a comparison. While the code may be initially harder to write, those languages force you to programmatically deal with all the possible failure cases.

Instead, I want to ensure that the input file contains exactly two lines. I'll do this by reading the file's lines into a list and checking that the list contains exactly two elements. I'll generate a more useful error message using the `sys.exit()` function, which means I'll need to add `import sys`:

```
def main():
    args = get_args()
```

```
    lines = args.file.read().splitlines() ❶

    if len(lines) != 2: ❷
        sys.exit(f'Input file "{args.file.name}" must have two lines')
❸

    seq1, seq2 = lines ❹
    print(seq1, seq2) ❺
```

❶ Read the input file and split the content on newlines into a list.

❷ Check that there are exactly two lines.

❸ Print an error message and exit the program with a non-zero value.

❹ Unpack the sequences.

❺ Print the sequences

Now observe the difference given an empty file. I should treat my users with respect and consideration by providing *useful* error messages that help them solve their own problems.

```
$ ./hamm.py tests/inputs/empty.txt
Input file "tests/inputs/empty.txt" must have two lines.
```

I would challenge you to now create an implementation of the Hamming distance using the Feynman algorithm:

1. Write down the problem.
2. Think real hard.
3. Write down the solution.

It’s perfectly fine to spend a few hours or days working on the problem. There’s no hurry to find a solution. Think about how, for

instance, you might iterate over the positions of two strings and compare characters to identify those that are not equal. If you want a boost in the right direction, keep reading, and I'll introduce some ideas that will form the first solution.

Iterating the Characters of Two Strings

At this point, you have safely read two sequences as input from a file. Now to find their Hamming distance. To start, consider these two sequences:

```
>>> seq1, seq2 = 'AC', 'ACGT'
```

The distance is 2 because you would either need to add *GT* to the first sequence or remove *GT* from the second sequence. I would suggest that the baseline distance is the difference in their lengths. Note that the Rosalind challenge assumes two strings of equal lengths, but I want to use this exercise to consider strings of different lengths.

Depending on the order you do the subtraction, you might end up with a negative number:

```
>>> len(seq1) - len(seq2)
-2
```

Use the `abs()` function to get the absolute value:

```
>>> distance = abs(len(seq1) - len(seq2))
>>> distance
2
```

Now I will consider how to iterate the characters they have in common. I can use the `min()` function to find the length of the shorter sequence:

```
>>> min(len(seq1), len(seq2))
2
```

and use this with the `range()` to get the indexes of the common characters:

```
>>> for i in range(min(len(seq1), len(seq2))):
...     print(seq1[i], seq2[i])
...
A A
C C
```

When these two characters are *not* equal, the `distance` variable should be incremented. I believe this should be enough information for you to craft a solution that passes the test suite. Once you have a working solution, explore some other ways you might write your algorithm, and keep checking your work using the test suite. In addition to running the tests via `pytest`, be sure to use the `make test` option to verify that your code also passes the various linting and type checking tests.

Solutions

This section works through eight variations on how to find the Hamming distance starting with an entirely manual calculation that takes several lines of code and ending with a solution that combines several functions in a single line.

Solution 1: Iterate and Count

The first solution follows from the suggestions in the introduction:

```
def main():
    args = get_args()
    lines = args.file.read().splitlines()

    if len(lines) != 2:
```

```
        sys.exit(f'Input file "{args.file.name}" must have two
lines.')

    seq1, seq2 = lines
    l1, l2 = len(seq1), len(seq2) ❶
    distance = abs(l1 - l2) ❷

    for i in range(min(l1, l2)): ❸
        if seq1[i] != seq2[i]: ❹
            distance += 1 ❺

    print(distance) ❻
```

❶ Since I'll use the lengths more than once, store into variables.

❷ The base distance is the difference between the two lengths.

❸ Use the shorter length to find the indexes in common.

❹ Check the letters at each position.

❺ Increment the distance by 1.

❻ Print the distance.

This solution is very explicit, laying out every individual step needed to compare all the characters of two strings. The following solutions will start to shorten many of the steps, so be sure you are comfortable with exactly what I've shown here.

Solution 2: Creating a Unit Test

The first solution leaves me feeling vaguely uncomfortable because the code to calculate the Hamming distance should be in its own function with tests. I'll start by imagining the inputs and output of my function:

```
def hamming(seq1: str, seq2: str) -> int: ❶
    """ Calculate Hamming distance """

    return 0 ❷
```

❶ The function will accept two strings as positional arguments and will return an integer.

❷ To start, the function will always return 0.

> **NOTE**
>
> I want to stress the fact that the function *does not print* the answer but rather *returns it as a result*. If you wrote this function `print()` the distance, you would not be able to write a unit test. You would have to rely entirely on the integration test that looks to see if the program prints the correct answer. As much as possible, I would encourage you to write *pure* functions that act *only* on the arguments and have no side effects. Printing is a side effect, and, while the program does need to print the answer eventually, the function's job is *only* to return an integer when given two strings.

I've already shown a few test cases I can encode:

```
def test_hamming() -> None:
    """ Test hamming """

    assert hamming('', '') == 0 ❶
    assert hamming('AC', 'ACGT') == 2 ❷
    assert hamming('GAGCCTACTAACGGGAT', 'CATCGTAATGACGGCCT') == 7 ❸
```

❶ I always think it's good practice to send empty strings for string inputs.

❷ The difference is due only to length.

❸ The example from the documentation.

I know this may seem a bit extreme because this function is essentially the entire program. I'm almost duplicating the integration test, I know, but I'm using this to point out best practices for writing programs. The `hamming()` function is a good unit of code, and it really does belong in a function with a test. In a much larger program, this would be one of perhaps dozens to hundreds of other functions, and each should be *encapsulated*, *documented*, and *tested*.

Following test-driven principles, run `pytest` on the program to ensure that the test fails:

```
$ pytest -v hamm.py
========================== test session starts
==========================
...

hamm.py::test_hamming FAILED
[100%]

=============================== FAILURES
===============================
_______________________________ test_hamming
_______________________________

    def test_hamming() -> None:
        """ Test hamming """

        assert hamming('', '') == 0
>       assert hamming('AC', 'ACGT') == 2
E       assert 0 == 2
E         +0
E         -2

hamm.py:69: AssertionError
======================== short test summary info
========================
FAILED hamm.py::test_hamming - assert 0 == 2
============================ 1 failed in 0.13s
============================
```

Now copy the code from `main()` to fix the function:

```
def hamming(seq1: str, seq2: str) -> int:
    """ Calculate Hamming distance """

    l1, l2 = len(seq1), len(seq2)
    distance = abs(l1 - l2)

    for i in range(min(l1, l2)):
        if seq1[i] != seq2[i]:
            distance += 1

    return distance
```

Verify your function is correct:

```
$ pytest -v hamm.py
========================== test session starts
==========================
...

hamm.py::test_hamming PASSED
[100%]

=========================== 1 passed in 0.02s
===========================
```

You can incorporate it into your `main()` function like so:

```
def main():
    args = get_args()
    lines = args.file.read().splitlines()

    if len(lines) != 2:
        sys.exit(f'Input file "{args.file.name}" must have two
lines.')

    seq1, seq2 = lines ❶
    print(hamming(seq1, seq2)) ❷
```

❶ Unpack the two sequences.

❷ Print the return value from the function.

This hides the complexity of the program into a named, documented, tested unit, shortening the main body of the program, and improving the readability. TDD FTW.

Solution 3: Using the zip Function

The following solution uses the `zip()` function to combine the elements from two sequences. The result is a list of tuples containing the characters from each position (see Figure 6-3). Note that `zip()` is another lazy function, so I'll use `list()` to coerce the values in the REPL:

```
>>> list(zip('ABC', '123'))
[('A', '1'), ('B', '2'), ('C', '3')]
```

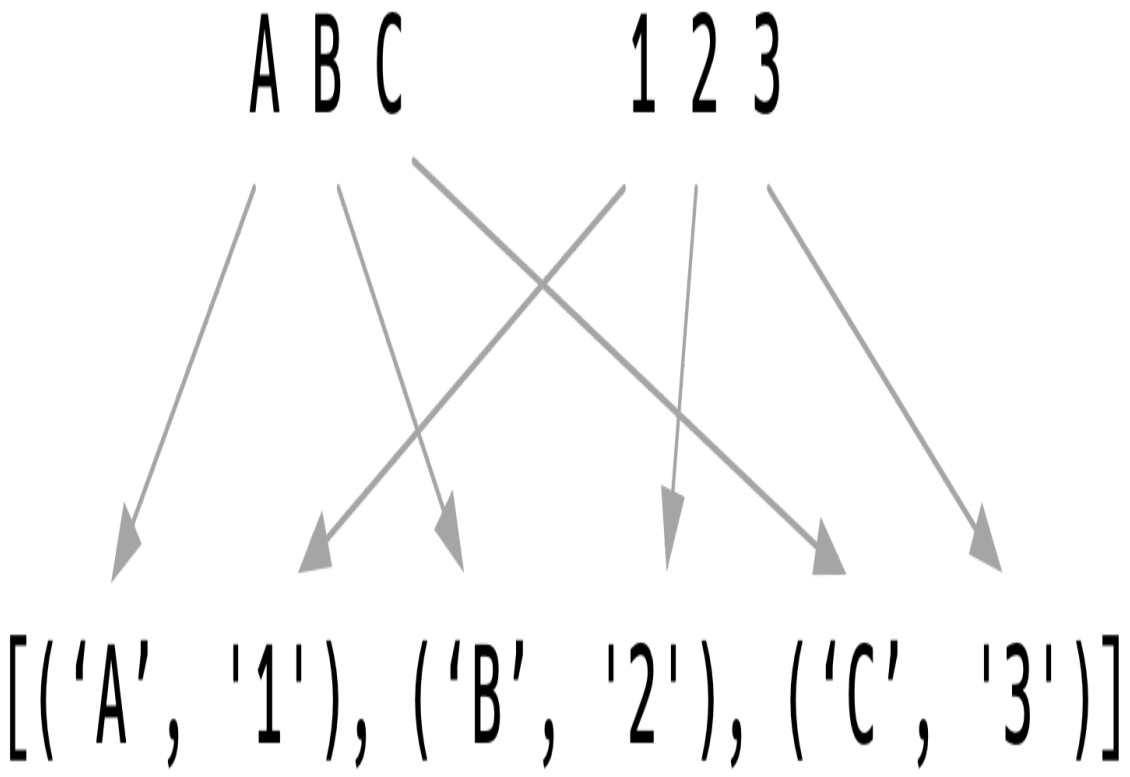


Figure 6-3. The tuples are composed of characters in common positions.

If I use the *AC* and *ACGT* sequences, you'll notice that `zip()` stops with the shorter sequence, as shown in Figure 6-4:

```
>>> list(zip('AC', 'ACGT'))
[('A', 'A'), ('C', 'C')]
```

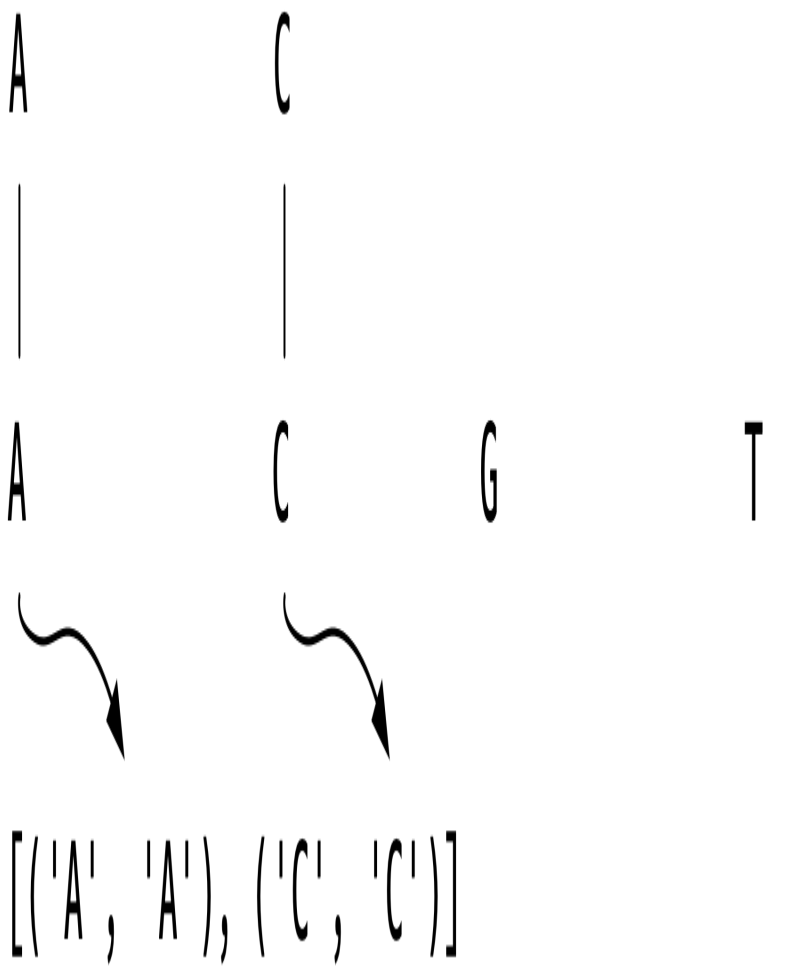


Figure 6-4. The zip function will stop at the shortest sequence.

I can use a `for` loop to iterate over each pair. So far in my `for` loops, I used a single variable to represent each element in a list like this:

```
>>> for tup in zip('AC', 'ACGT'):
...     print(tup)
...
```

```
('A', 'A')
('C', 'C')
```

Python allows me to unpack those tuple values into separate variables:

```
>>> for char1, char2 in zip('AC', 'ACGT'):
...     print(char1, char2)
...
A A
C C
```

The `zip()` function obviates a couple of lines from the first implementation:

```
def hamming(seq1: str, seq2: str) -> int:
    """ Calculate Hamming distance """

    distance = abs(len(seq1) - len(seq2)) ❶

    for char1, char2 in zip(seq1, seq2): ❷
        if char1 != char2: ❸
            distance += 1 ❹

    return distance
```

❶ Start with the absolute difference of the lengths.

❷ Use `zip()` to pair up the characters of the two strings.

❸ Check if the two characters are not equal.

❹ Increment the distance.

Solution 4: Using the zip_longest Function

The next solution imports the `zip_longest()` function from the `itertools` module. As shown in Figure 6-5, this version of the

function will insert a `None` value when a sequence has been exhausted:

```
>>> from itertools import zip_longest
>>> list(zip_longest('AC', 'ACGT'))
[('A', 'A'), ('C', 'C'), (None, 'G'), (None, 'T')]
```

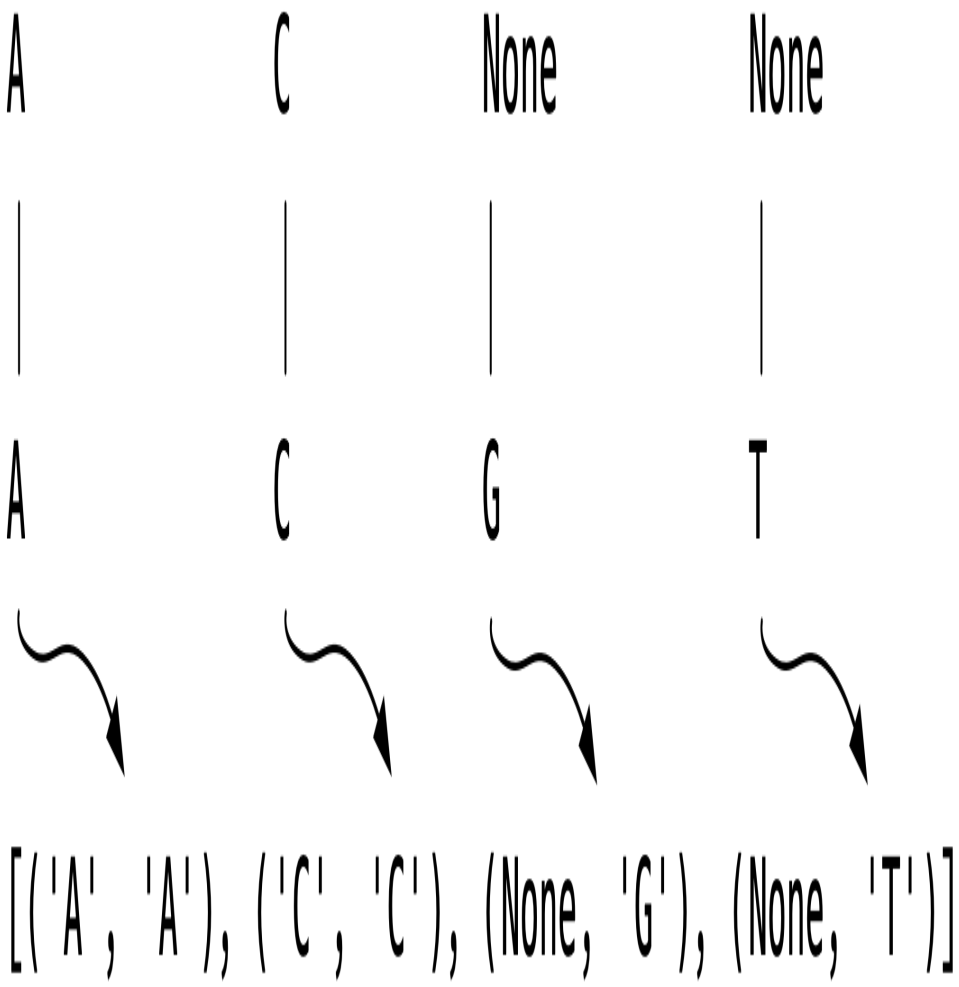


Figure 6-5. The zip_longest function will stop at the longest sequence.

I no longer need to start by subtracting the lengths of the sequences. Instead, I'll initialize a distance variable to 0 and then use `zip()` to create tuples of bases to compare:

```
def hamming(seq1: str, seq2: str) -> int:
    """ Calculate Hamming distance """

    distance = 0 ❶
    for char1, char2 in zip_longest(seq1, seq2): ❷
        if char1 != char2: ❸
            distance += 1 ❹

    return distance
```

❶ Initialize the distance to 0.

❷ Zip to the longest sequence.

❸ Compare the characters.

❹ Increment the counter.

Solution 5: Using a List Comprehension

All the solutions up to this point have used a `for` loop. I hope you're starting to anticipate that I'm going to show you how to convert this into a list comprehension next. When the goal is to create a new list or reduce a list of values to some answer, it's often shorter and preferable to use a list comprehension.

The first version is going to use an if-expression to return a 1 if the two characters are the same or a 0 if they are not:

```
>>> seq1, seq2, = 'GAGCCTACTAACGGGAT', 'CATCGTAATGACGGCCT'
>>> [1 if c1 != c2 else 0 for c1, c2 in zip_longest(seq1, seq2)]
[1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0]
```

The Hamming distance, then, is the `sum()` of these:

```
>>> sum([1 if c1 != c2 else 0 for c1, c2 in zip_longest(seq1, seq2)])
7
```

Another way to express this idea is to only produce the 1s by using a guard:

```
>>> [1 for c1, c2 in zip_longest(seq1, seq2) if c1 != c2]
[1, 1, 1, 1, 1, 1, 1]
>>> sum([1 for c1, c2 in zip_longest(seq1, seq2) if c1 != c2])
7
```

Or use the Boolean/integer coercion I showed in Chapter 5:

```
>>> sum([c1 != c2 for c1, c2 in zip_longest(seq1, seq2)])
7
```

Any of these ideas will reduce the function to a single line of code that passes the tests:

```
def hamming(seq1: str, seq2: str) -> int:
    """ Calculate Hamming distance """

    return sum([c1 != c2 for c1, c2 in zip_longest(seq1, seq2)])
```

Solution 6: Using the filter Function

Whenever you write a list comprehension with a guard, you can rewrite the same idea using `filter()`. The syntax is a little ugly because Python doesn't allow the unpacking of the tuples from `zip_longest()` into separate variables. That is, I would *like* to write this:

```
>>> list(filter(lambda char1, char2: char1 != char2, zip_longest(seq1,
seq2)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: <lambda>() missing 1 required positional argument: 'char2'
```

Since that's not possible, I will usually call the `lambda` variable `tup` or `t` to remind me this is a tuple. I will use the positional tuple notation to compare the element in the 0th position to the element in 1st

position. The `filter()` will only produce those tuples where the elements are different:

```
>>> seq1, seq2 = 'AC', 'ACGT'
>>> list(filter(lambda t: t[0] != t[1], zip_longest(seq1, seq2)))
[(None, 'G'), (None, 'T')]
```

The Hamming distance then is the length of this list. Note that the `len()` function will not prompt the `filter()` to produce values:

```
>>> len(filter(lambda t: t[0] != t[1], zip_longest(seq1, seq2)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'filter' has no len()
```

This is one of those instances where the code must use `list()` to coerce the values. Here is how I can incorporate these ideas:

```
def hamming(seq1: str, seq2: str) -> int:
    """ Calculate Hamming distance """

    distance = filter(lambda t: t[0] != t[1], zip_longest(seq1, seq2))
❶
    return len(list((distance))) ❷
```

❶ Use a `filter()` to find tuple pairs of different characters.

❷ Return the length of the resulting list.

Solution 7: Using the map Function with zip_longest

This solution uses `map()` instead of `filter()` only to show you that the same inability to unpack the tuples also applies. I'd like to use `map()` to produce a list of Boolean values indicating whether the character pairs match or not:

```
>>> seq1, seq2 = 'AC', 'ACGT'
>>> list(map(lambda t: t[0] != t[1], zip_longest(seq1, seq2)))
[False, False, True, True]
```

The `lambda` is identical to the one to `filter()` that was used as the *predicate* to determine which elements are allowed to pass. Here the code *transforms* the elements into the result of applying the `lambda` function to the arguments. It might help to remember that `map()` will always return the same number of elements it consumes, but `filter()` may return fewer or none at all.

I can `sum()` these Booleans to get the number of mismatched pairs:

```
>>> seq1, seq2, = 'GAGCCTACTAACGGGAT', 'CATCGTAATGACGGCCT'
>>> sum(map(lambda t: t[0] != t[1], zip_longest(seq1, seq2)))
7
```

Here is the function with this idea:

```
def hamming(seq1: str, seq2: str) -> int:
    """ Calculate Hamming distance """

    return sum(map(lambda t: t[0] != t[1], zip_longest(seq1, seq2)))
```

Even though these functions have gone from 10 or more lines of code to a single line, it still makes sense for this to be a function with a descriptive name and tests. Eventually, you'll start creating modules of reusable code to share across your projects.

Solution 8: Using the starmap and operator.ne Functions

I confess that I showed the last few solutions solely to build up to this last solution. Let me start by showing how I can assign a `lambda` to a variable:

```
>>> not_same = lambda t: t[0] != t[1]
```

This is not recommended syntax, and `pylint` will definitely fail your code on this and recommend a `def` instead:

```
def not_same(t):
    return t[0] != t[1]
```

Both will create a function called `not_same()` that will accept a tuple and return whether the two elements are the same:

```
>>> not_same(('A', 'A'))
False
>>> not_same(('A', 'T'))
True
```

If, however, I wrote the function to accept two positional arguments, the same error I saw before crops up:

```
>>> not_same = lambda a, b: a != b
>>> list(map(not_same, zip_longest(seq1, seq2)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: <lambda>() missing 1 required positional argument: 'b'
```

What I need is a version of `map()` that can splat the incoming tuple (as I first showed in Chapter 1) by adding `*` (star, asterisk, or splat) to the tuple to expand it into its elements, which is exactly what the function `itertools.starmap()` does:

```
>>> from itertools import zip_longest, starmap
>>> seq1, seq2 = 'AC', 'ACGT'
>>> list(starmap(not_same, zip_longest(seq1, seq2)))
[False, False, True, True]
```

But, wait, there's more! I don't even need to write my own `not_same()` function because I already have `operator.ne()` (not equal) which I usually write using the `!=` operator:

```
>>> import operator
>>> operator.ne('A', 'A')
False
>>> operator.ne('A', 'T')
True
```

An *operator* is a special binary function (accepting two arguments) where the function name is usually some symbol like `+` that sits between the arguments. In the case of `+`, Python has to decide if this means `operator.add()`:

```
>>> 1 + 2
3
>>> operator.add(1, 2)
3
```

or `operator.concat()`:

```
>>> 'AC' + 'GT'
'ACGT'
>>> operator.concat('AC', 'GT')
'ACGT'
```

The point is that I already have an existing function that expects two arguments and returns whether they are equal, and I can use `starmap()` to properly expand the tuples into the needed arguments:

```
>>> seq1, seq2 = 'AC', 'ACGT'
>>> list(starmap(operator.ne, zip_longest(seq1, seq2)))
[False, False, True, True]
```

As before, the Hamming distance is the `sum()` of the unmatched pairs:

```
>>> seq1, seq2, = 'GAGCCTACTAACGGGAT', 'CATCGTAATGACGGCCT'
>>> sum(starmap(operator.ne, zip_longest(seq1, seq2)))
7
```

To see it in action:

```
def hamming(seq1: str, seq2: str) -> int:
    """ Calculate Hamming distance """

    return sum(starmap(operator.ne, zip_longest(seq1, seq2))) ❶
```

❶ Zip the sequences, transform the tuples to boolean comparisons, and sum.

Note that this final solution relies entirely on fitting four functions that I didn't write. The best code is code you don't write (or test or document).

Going Further

- Without looking at the source code for `zip_longest()`, write your own version. Be sure to start with a test, then write the function that satisfies the test.
- Expand your program to handle more than two input sequences. Start by creating a new input file and adding a sequence. Have your program print the Hamming distance between every pair of sequences. That means the program will print *n* choose *k* numbers which is *n*! / *k*!(*n* - *k*)!. For three sequences, your program will print 3!/(2!(3-2)!) = 6/2 = 3 distance pairs.
- Try writing a sequence alignment algorithm that will show there is, for instance, just on difference between the sequences *AAACCCGGGTTT* and *AACCCGGGTTTA*.

Review

This was a rather deep rabbit hole to go down just to find the Hamming distance, but it highlights lots of interesting bits about Python functions:

- The built-in `zip()` function will combine two or more lists into a list of tuples grouping elements at common positions. It stops at the shortest sequence, so use `itertools.zip_longest()` function if you want to go to the longest sequence.
- A list comprehension with a guard will only produce elements that return a truthy value for the guard. This guard predicate serves the same purpose as the `lambda` in the `filter()` function.
- Both `map()` and `filter()` apply a function to some iterable of values. The `map()` will return a new sequence transformed by the function while `filter()` will only return those elements that return a truthy value when the function is applied.
- The function to `map()` and `filter()` can be an anonymous function created by `lambda` or an existing function.
- The `operator` module contains many functions like `ne()` (not equal) that can be used with `map()` and `filter()`.
- The `functools.starmap()` function works just like `map()` but will splat the function's incoming values to expand them into a list of values.

Chapter 7. Translating RNA into Protein: More Functional Programming

According to the Central Dogma of molecular biology, *DNA makes RNA, and RNA makes protein*. You've handled transcribing DNA to RNA, so it's time to translate RNA into protein sequences. As described on the Rosalind page, I need to write a program that accepts a string of RNA and produces an amino acid sequence. I will show several solutions using lists, `for` loops, list comprehensions, dictionaries, and higher-order functions, but I confess I'll end with a Biopython function. Still, it'll will be tons of fun.

Mostly I'm going to focus on how to write, test, and compose small functions to create solutions. What you'll learn:

- How to extract codons/k-mers from a sequence using string slices
- How to use a dictionary as a lookup table
- How to translate a `for` loop into a list comprehension and a `map()` expression
- How to use the `takewhile()` and `partial()` functions
- How to use the `Bio.Seq` module to translate RNA to proteins

Getting Started

You will need to work in the *07_prot* directory for this exercise. I recommend you begin by copying the first solution to `prot.py` and asking for the usage:

```
$ cp solution1_for.py prot.py
$ ./prot.py -h
usage: prot.py [-h] RNA

Translate RNA to proteins

positional arguments:
  RNA         RNA sequence

optional arguments:
  -h, --help  show this help message and exit
```

The program requires *RNA* as a single positional argument. I'll use the example string from the Rosalind page:

```
$ ./prot.py AUGGCCAUGGCGCCCAGAACUGAGAUCAAUAGUACCCGUAUUAACGGGUGA
MAMAPRTEINSTRING
```

I recommend you run `make test` to ensure the program works properly. When you feel you have a decent idea of how the program works, start from scratch:

```
$ new.py -fp 'Translate RNA to proteins' prot.py
Done, see new script "prot.py".
```

Modify your arguments until the program will produce the correct usage, then modify your `main()` to print the incoming RNA string:

```
def main() -> None:
    args = get_args()
    print(args.rna)
```

Verify it works:

```
$ ./prot.py AUGGCCAUGGCGCCCAGAACUGAGAUCAAUAGUACCCGUAUUAACGGGUGA
AUGGCCAUGGCGCCCAGAACUGAGAUCAAUAGUACCCGUAUUAACGGGUGA
```

Run `pytest` to see how you fare. As usual, the first two tests should be fine, and you should fail the third where the output should be the

protein translation. If you think you can figure this out, go ahead with your own solution. It's perfectly fine to struggle. There's no hurry, so take a few days if you need. Be sure to incorporate naps and walks (diffuse thinking time) in addition to your focused coding time. If you need some help, read on.

K-mers and Codons

So far you've seen many examples of how to iterate over the characters of a string such as the bases of DNA. Here I need to group the bases of RNA into threes to read the codons matching the translation shown in the Table 7-1:

Table 7-1. The RNA codon table describes how 3-mers of RNA encode the 22 amino acids.

AAA K	AAC N	AAG K	AAU N	ACA T
ACC T	ACG T	ACU T	AGA R	AGC S
AGG R	AGU S	AUA I	AUC I	AUG M
AUU I	CAA Q	CAC H	CAG Q	CAU H
CCA P	CCC P	CCG P	CCU P	CGA R
CGC R	CGG R	CGU R	CUA L	CUC L
CUG L	CUU L	GAA E	GAC D	GAG E
GAU D	GCA A	GCC A	GCG A	GCU A
GGA G	GGC G	GGG G	GGU G	GUA V
GUC V	GUG V	GUU V	UAC Y	UAU Y
UCA S	UCC S	UCG S	UCU S	UGC C
UGG W	UGU C	UUA L	UUC F	UUG L
UUU F	UAA Stop	UAG Stop	UGA Stop	

Given some string of RNA:

```
>>> rna = 'AUGGCCAUGGCGCCCAGAACUGAGAUCAAUAGUACCCGUAUUAACGGGUGA'
```

I want to read the first 3 bases *AUG*. As shown in Figure 7-1, I can use a string slice to manually grab the characters from index 0 to 3 (remembering that the upper bound is not inclusive):

```
>>> rna[0:3]
'AUG'
```



Figure 7-1. Extracting codons from RNA using string slices.

The next codon can be found by adding 3 to the start and stop positions:

```
>>> rna[3:6]
'GCC'
```

Can you see a pattern emerging? For the first number, I need to start at 0 and add 3. For the second number, I need to add 3 to the first number (see Figure 7-2).

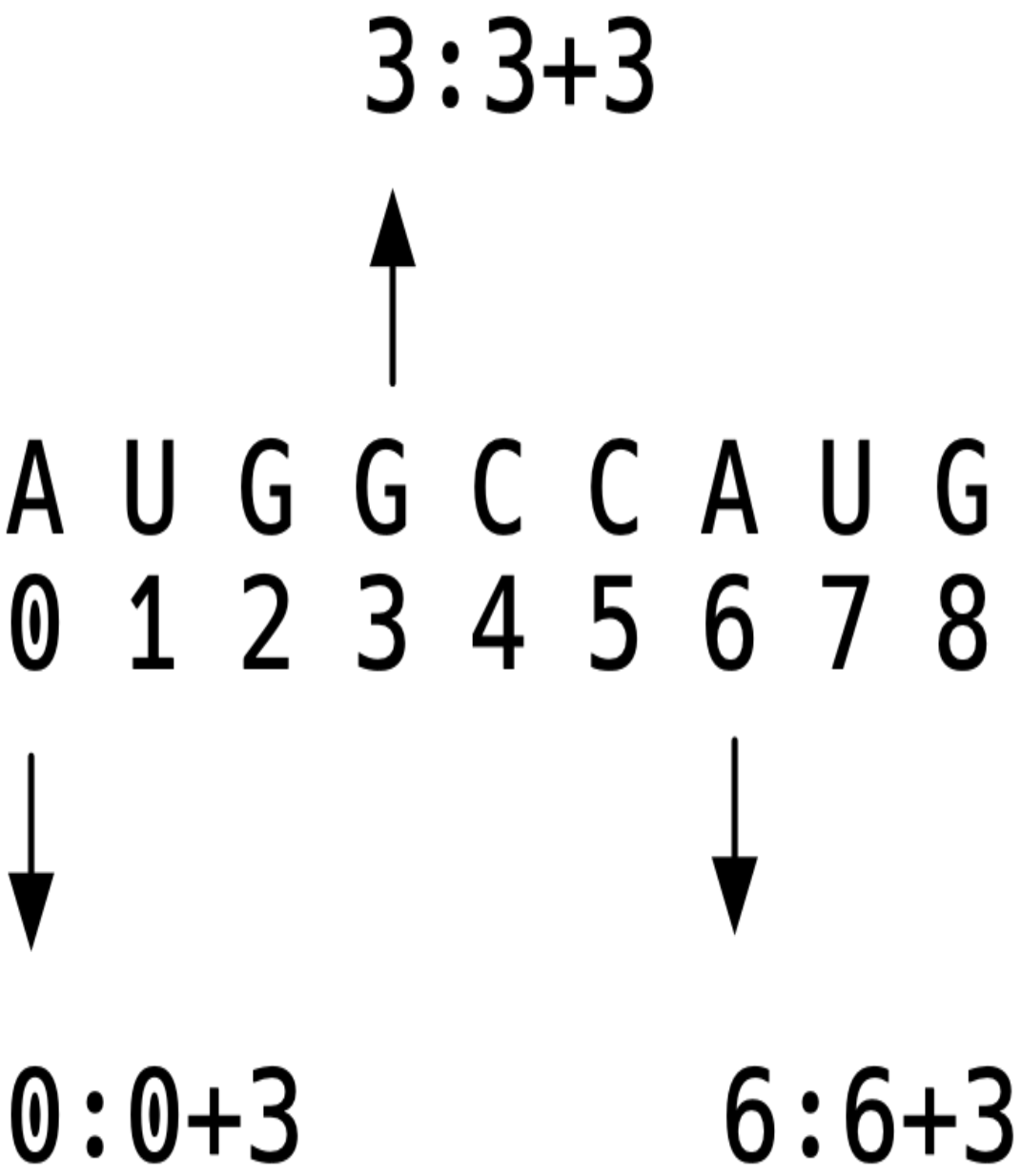


Figure 7-2. Each slice is a function of the start positions of the codons which can be found using the `range()` function.

I can handle the first part using the `range()` function which can take 1, 2, or 3 arguments. Given just one argument, it will produce all the numbers from 0 up to (but not including) the given value. Note this is a lazy function which I'll coerce with `list()`:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Given two arguments, `range()` will assume the first is the start and the second is the stop:

```
>>> list(range(5, 10))
[5, 6, 7, 8, 9]
```

A third argument will be interpreted as the step size. In our case, I want to start counting at 0 and go up to the length of the RNA, stepping by 3. These are the starting positions for the codons:

```
>>> list(range(0, len(rna), 3))
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48]
```

I can use a list comprehension to generate the start and stop values as tuples. The stop positions are the 3 more than the start positions. I'll show just the first 5:

```
>>> [(n, n + 3) for n in range(0, len(rna), 3)][:5]
[(0, 3), (3, 6), (6, 9), (9, 12), (12, 15)]
```

I can use those values to take slices of the RNA:

```
>>> [rna[n:n + 3] for n in range(0, len(rna), 3)][:5]
['AUG', 'GCC', 'AUG', 'GCG', 'CCC']
```

The codons are subsequences of the RNA, and they are similar to *k-mers*. The *k* is the size, here 3, and *mer* is *a share* as in the word *polymer*. It's common to refer to k-mers by their size, so here I might call these *3-mers*. K-mers overlap by one character, so the window shifts right by one base. Figure 7-3 shows the first 7 k-mers found in the first 9 bases of the input RNA.

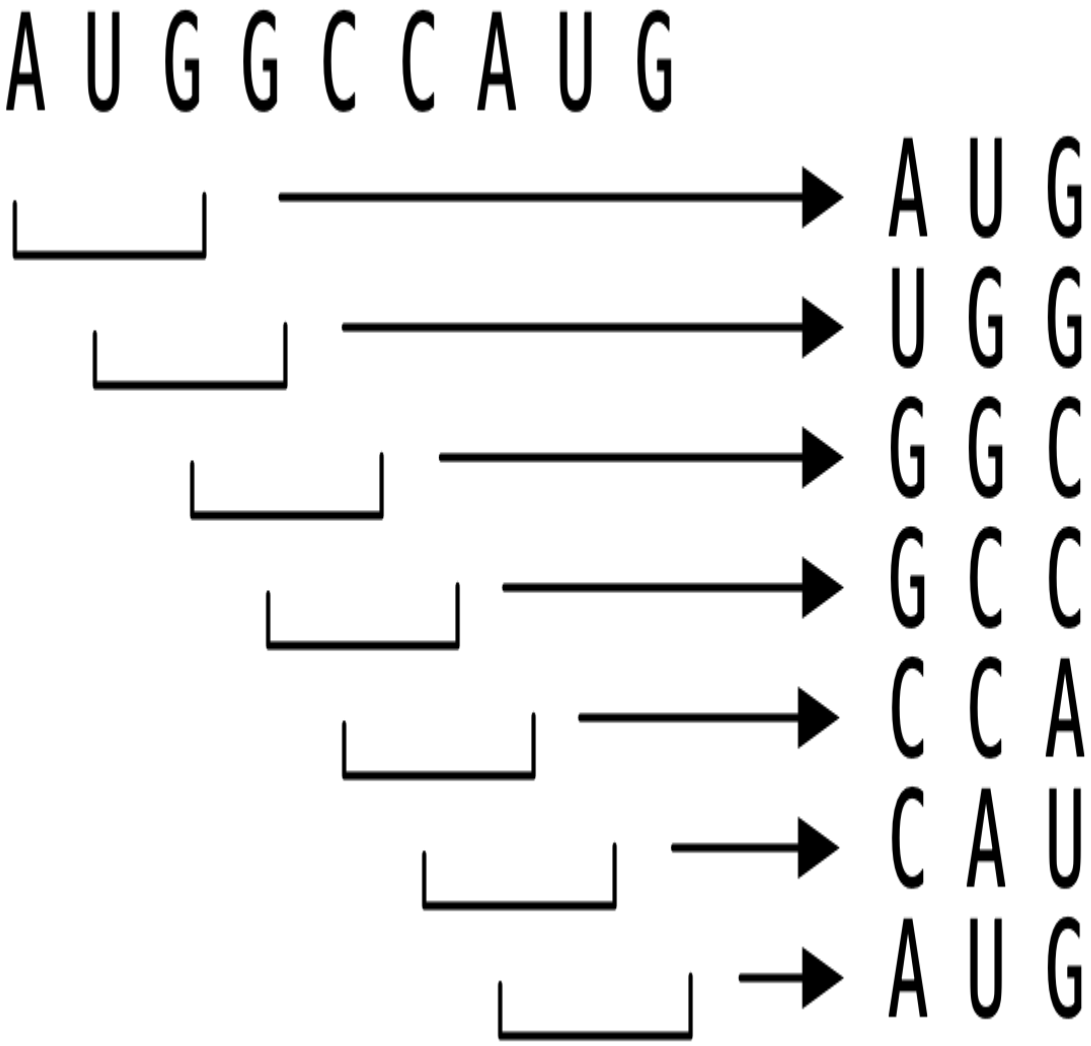


Figure 7-3. All the 3-mers in the first 9 bases of the RNA sequence.

The number *n* of *k*-mers in any sequence *s* is:

n = len(*s*) - *k* + 1

The length of this RNA sequence is 51, so it contains 49 3-mers:

n = 51 - 3 + 1 = 49

Codons do not overlap and so shift 3 (see Figure 7-4), leaving 17 codons:

```
>>> len([rna[n:n + 3] for n in range(0, len(rna), 3)])
17
```

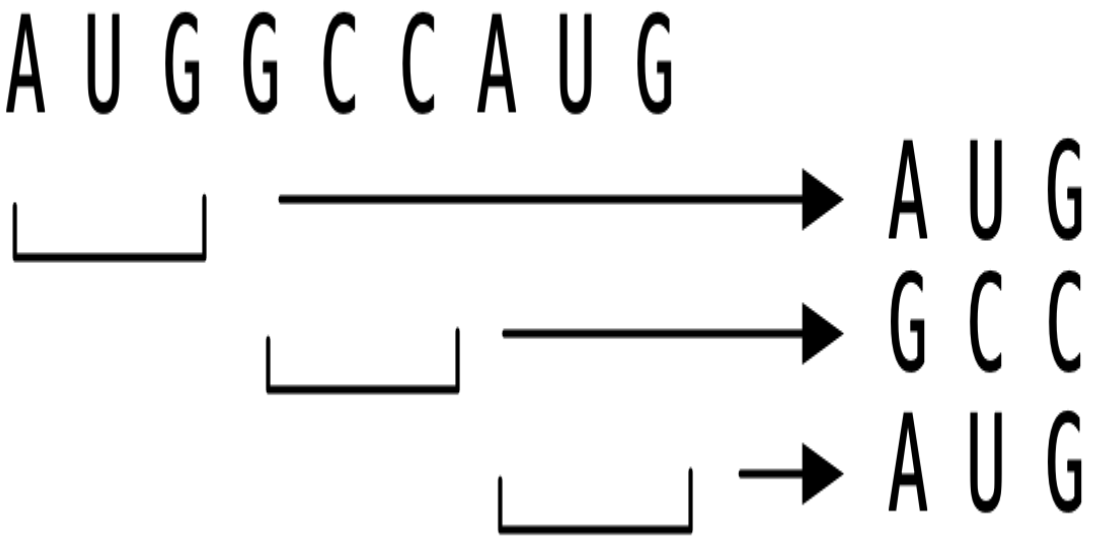


Figure 7-4. Codons are non-overlapping k-mers.

Translating Codons

Now that you know how to extract the codons from the RNA, let's consider how to translate a codon into a protein. The Rosalind page provides the following translation table:

```
UUU F      CUU L      AUU I      GUU V
UUC F      CUC L      AUC I      GUC V
UUA L      CUA L      AUA I      GUA V
UUG L      CUG L      AUG M      GUG V
UCU S      CCU P      ACU T      GCU A
UCC S      CCC P      ACC T      GCC A
UCA S      CCA P      ACA T      GCA A
UCG S      CCG P      ACG T      GCG A
UAU Y      CAU H      AAU N      GAU D
UAC Y      CAC H      AAC N      GAC D
UAA Stop   CAA Q      AAA K      GAA E
UAG Stop   CAG Q      AAG K      GAG E
UGU C      CGU R      AGU S      GGU G
UGC C      CGC R      AGC S      GGC G
```

```
UGA Stop   CGA R      AGA R      GGA G
UGG W      CGG R      AGG R      GGG G
```

A dictionary would be a natural data structure to look up a string like *AUG* to find that it translates to the protein *M*. I will leave it to you to incorporate this data into your program. For what it's worth, I changed *Stop* to `*` in my dictionary to indicate the stop codon. I called my dictionary `codon_to_aa`, and I can use it like so:

```
>>> rna = 'AUGGCCAUGGCGCCCAGAACUGAGAUCAAUAGUACCCGUAUUAACGGGUGA'
>>> aa = []
>>> for codon in [rna[n:n + 3] for n in range(0, len(rna), 3)]:
...     aa.append(codon_to_aa[codon])
...
>>> aa
['M', 'A', 'M', 'A', 'P', 'R', 'T', 'E', 'I', 'N', 'S', 'T', 'R', 'I',
 'N', 'G', '*']
```

The `*` codon indicates where the translation should end and should not be included in the output. Your algorithm should consider that the stop codon may occur before the end of the RNA string and should halt accordingly. This should be enough hints for you to create a solution that passes the tests. Be sure to run `pytest` and `make test` to ensure your program is logically and stylistically correct.

Solutions

In this section, I will show five variations on how to translate RNA into protein moving from wholly manual solutions where I encode the RNA codon table with a dictionary and moving to a single line of code that uses a function from Biopython.

Solution 1: Using a for Loop

Here is the whole of my first solution that uses a `for` loop to iterate the codons to translate them via a dictionary:

```
#!/usr/bin/env python3
""" Translate RNA to proteins """

import argparse
from typing import NamedTuple


class Args(NamedTuple):
    """ Command-line arguments """
    rna: str


# --------------------------------------------------
def get_args() -> Args:
    """Get command-line arguments"""

    parser = argparse.ArgumentParser(
        description='Translate RNA to proteins',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('rna', type=str, metavar='RNA', help='RNA
sequence')

    args = parser.parse_args()

    return Args(args.rna)


# --------------------------------------------------
def main() -> None:
    """Make a jazz noise here"""

    args = get_args()
    rna = args.rna.upper() ❶
    codon_to_aa = { ❷
        'AAA': 'K', 'AAC': 'N', 'AAG': 'K', 'AAU': 'N', 'ACA': 'T',
        'ACC': 'T', 'ACG': 'T', 'ACU': 'T', 'AGA': 'R', 'AGC': 'S',
        'AGG': 'R', 'AGU': 'S', 'AUA': 'I', 'AUC': 'I', 'AUG': 'M',
        'AUU': 'I', 'CAA': 'Q', 'CAC': 'H', 'CAG': 'Q', 'CAU': 'H',
        'CCA': 'P', 'CCC': 'P', 'CCG': 'P', 'CCU': 'P', 'CGA': 'R',
        'CGC': 'R', 'CGG': 'R', 'CGU': 'R', 'CUA': 'L', 'CUC': 'L',
        'CUG': 'L', 'CUU': 'L', 'GAA': 'E', 'GAC': 'D', 'GAG': 'E',
        'GAU': 'D', 'GCA': 'A', 'GCC': 'A', 'GCG': 'A', 'GCU': 'A',
        'GGA': 'G', 'GGC': 'G', 'GGG': 'G', 'GGU': 'G', 'GUA': 'V',
        'GUC': 'V', 'GUG': 'V', 'GUU': 'V', 'UAC': 'Y', 'UAU': 'Y',
```

```
        'UCA': 'S', 'UCC': 'S', 'UCG': 'S', 'UCU': 'S', 'UGC': 'C',
        'UGG': 'W', 'UGU': 'C', 'UUA': 'L', 'UUC': 'F', 'UUG': 'L',
        'UUU': 'F', 'UAA': '*', 'UAG': '*', 'UGA': '*',
    }

    # Method 1: for loop
    k = 3 ❸
    protein = '' ❹
    for codon in [rna[i:i + k] for i in range(0, len(rna), k)]: ❺
        aa = codon_to_aa.get(codon, '-') ❻
        if aa == '*': ❼
            break ❽
        protein += aa ❾

    print(protein) ❿


if __name__ == '__main__':
    main()
```

❶ Copy the incoming RNA, forcing to uppercase.

❷ Create a codon/AA lookup table using a dictionary.

❸ Establish the size of *k* for finding k-mers.

❹ Initialize the protein sequence to the empty string.

❺ Iterate through the codons of the RNA.

❻ Lookup the amino acid for this codon.

❼ Check if this is the stop codon.

❽ Break out of the `for` loop.

❾ Append the amino acid to the protein sequence.

❿ Print the protein sequence.

Solution 2: Adding Unit Tests

The first solution works adequately well, and, for such a short program, has a decent organization. The problem is that short programs usually become long programs. It's common to make functions longer and longer, so I'd like to show how I can break up the code in `main()` into a couple of smaller functions with tests. Generally speaking, I like to see a function fit into 50 lines or fewer on the high end. As for how short a function can be, I'm not opposed to a single line of code.

My first intuition is to extract the code that finds the codons and make that into a function with a unit test. I can start by defining a placeholder for the function with a type signature that helps me think about what the function accepts as arguments and will return as a result:

```
def codons(seq: str, k: int) -> List[str]: ❶
    """ Extract k-sized codons from a sequence """

    return [] ❷
```

❶ The function will accept a sequence string and a *k* integer value and will return a list of strings.

❷ For now, just return an empty list.

Next, I define a `test_codons()` function to imagine how it might work. Whenever I have a string as a function parameter, I try passing the empty string. Whenever I have an integer as a function parameter, I try passing 0. Then I try other possible values and imagine what the function ought to do:

```
def test_codons() -> None:
    """ Test codons """

    assert codons('', 0) == []
```

```
    assert codons('', 1) == []
    assert codons('A', 1) == ['A']
    assert codons('A', 2) == ['A']
    assert codons('ABC', 3) == ['ABC']
    assert codons('ABCDE', 3) == ['ABC', 'DE']
    assert codons('ABCDEF', 3) == ['ABC', 'DEF']
```

Now to write the function that will satisfy these tests. If I move the relevant code from the `main()` into the `codons()` function, I get this:

```
def codons(seq: str, k: int) -> List[str]:
    """ Extract k-sized codons from a sequence """

    ret = []
    for codon in [seq[i:i + k] for i in range(0, len(seq), k)]:
        ret.append(codon)

    return ret
```

When I try running `pytest` on this program, I get the following error:

```
    def codons(seq: str, k: int) -> List[str]:
        """ Extract k-sized codons from a sequence """

        ret = []
>       for codon in [seq[i:i + k] for i in range(0, len(seq), k)]: ❶
E       ValueError: range() arg 3 must not be zero ❷

solution2_unit.py:119: ValueError
```

❶ When `k` is 0, the `range()` function raises an exception.

❷ In the error message, *arg 3* means `k`.

My function failed my first test using the empty string and 0 as values. The problem is that the `range()` function is unhappy when `k` is 0 (*arg 3 must not be zero*). While it's possible to use "`k: int`" in the function signature to indicate that this value should be an integer,

it's not so easy to say it must be a *positive* integer. How can I fix this? I could actually use an `assert` statement inside the function:

```
def codons(seq: str, k: int) -> List[str]:
    """ Extract k-sized codons from a sequence """

    assert k > 0 ❶

    ret = []
    for codon in [seq[i:i + k] for i in range(0, len(seq), k)]:
        ret.append(codon)

    return ret
```

❶ An `assert` statement will raise an exception if the condition is not met.

The problem with this fix is that I've now introduced exception handling into my program. If you run `pytest` on this, you'll see it still fails the test. I would need to change the test to indicate that, given a *k* of 0, the function *should raise an exception*. I would also have to change any code that calls this function to accept the possibility that the function might raise an exception.

To my mind, exceptions create unnecessary complexity. I might choose to raise an exception if, deep in the bowels of a program, I discover a runtime condition that prevents further execution. I would prefer to *not* catch the exception and rather let the program crash.

You might think "Well, I'll just be sure to never pass a non-positive value," but you can't rely upon yourself to remember this. You're essentially like the doctor who says "Then don't do that" (Chapter 5). *Trusting yourself to not make mistakes is not a solution.* It's much wiser to encode how to handle unexpected or invalid argruments. I think the simplest solution here is to return the empty list if *k* is not a positive number:

```
def codons(seq: str, k: int) -> List[str]:
    """ Extract k-sized codons from a sequence """

    if k < 1:
        return []

    ret = []
    for codon in [seq[i:i + k] for i in range(0, len(seq), k)]:
        ret.append(codon)

    return ret
```

Since the `for` loop is being used to build up a return list, it would be stylistically better to use a list comprehension. I can also squash the code to a single line using an if-expression to check the value of *k*:

```
def codons(seq: str, k: int) -> List[str]:
    """ Extract k-sized codons from a sequence """

    return [] if k < 1 else [seq[i:i + k] for i in range(0, len(seq),
k)]
```

This is a nice little function that is documented and tested and which will make the rest of the code more readable:

```
def main() -> None:
    args = get_args()
    rna = args.rna.upper()
    codon_to_aa = {
        'AAA': 'K', 'AAC': 'N', 'AAG': 'K', 'AAU': 'N', 'ACA': 'T',
        'ACC': 'T', 'ACG': 'T', 'ACU': 'T', 'AGA': 'R', 'AGC': 'S',
        'AGG': 'R', 'AGU': 'S', 'AUA': 'I', 'AUC': 'I', 'AUG': 'M',
        'AUU': 'I', 'CAA': 'Q', 'CAC': 'H', 'CAG': 'Q', 'CAU': 'H',
        'CCA': 'P', 'CCC': 'P', 'CCG': 'P', 'CCU': 'P', 'CGA': 'R',
        'CGC': 'R', 'CGG': 'R', 'CGU': 'R', 'CUA': 'L', 'CUC': 'L',
        'CUG': 'L', 'CUU': 'L', 'GAA': 'E', 'GAC': 'D', 'GAG': 'E',
        'GAU': 'D', 'GCA': 'A', 'GCC': 'A', 'GCG': 'A', 'GCU': 'A',
        'GGA': 'G', 'GGC': 'G', 'GGG': 'G', 'GGU': 'G', 'GUA': 'V',
        'GUC': 'V', 'GUG': 'V', 'GUU': 'V', 'UAC': 'Y', 'UAU': 'Y',
        'UCA': 'S', 'UCC': 'S', 'UCG': 'S', 'UCU': 'S', 'UGC': 'C',
        'UGG': 'W', 'UGU': 'C', 'UUA': 'L', 'UUC': 'F', 'UUG': 'L',
        'UUU': 'F', 'UAA': '*', 'UAG': '*', 'UGA': '*',
```

```
}

protein = ''
for codon in codons(rna, 3): ❶
    aa = codon_to_aa.get(codon, '-')
    if aa == '*':
        break
    protein += aa

print(protein)
```

❶ The complexities of codon finding are hidden in a function.

Further, this function (and its test) would now be easier to incorporate into another program. The simplest case would be to copy and paste these lines, but a better solution would be to share the function. Let me demonstrate using the REPL. If you have the `codons()` function in your `prot.py` program, then import the function:

```
>>> from prot import codons
```

Now you can execute the `codons()` function:

```
>>> codons('AAACCCGGGTTT', 3)
['AAA', 'CCC', 'GGG', 'TTT']
```

Or you can import the entire `prot` module and call the function like so:

```
>>> import prot
>>> prot.codons('AAACCCGGGTTT', 3)
['AAA', 'CCC', 'GGG', 'TTT']
```

Python *programs* are also *modules* of reusable code. Sometimes you execute a source code file and it becomes a program, but there's not a big distinction in Python between a program and a

module. This is the meaning of the couplet at the end of all the programs:

```
if __name__ == '__main__': ❶
    main() ❷
```

❶ When a Python program is being *executed* as a program, the value of `__name__` is "*_main_*".

❷ Call the `main()` function to start the program.

> **NOTE**
>
> When a Python module is being *imported* by another piece of code, then the `__name__` is the name of the module, for example, *prot* in the case of `prot.py`. If you simply called `main()` at the end of the program without checking the `__name__`, then it would be executed whenever your module is imported, which is not good.

As you write more and more Python, you'll likely find you're solving some of the same problems repeatedly. It would be far better to share common solutions by writing functions that you share across projects rather than copy-pasting bits of code. Python makes it pretty easy to put reusable functions into modules and import them into other programs.

Solution 3: Another Function and a List Comprehension

The `codons()` function is tidy and useful and makes the `main()` function easier to understand; however, all the code that's left in `main()` is concerned with translating the protein. I'd like to hide this away in a `translate()` function, and here is the test I'd like to use:

```
def test_translate() -> None:
    """ Test translate """

    assert translate('') == '' ❶
    assert translate('AUG') == 'M' ❷
    assert translate('AUGCCGUAAUCU') == 'MP' ❸
    assert translate('AUGGCCAUGGCGCCCAGAACUGAGAU' ❹
                     'CAAUAGUACCCGUAUUAACGGGUGA') ==
'MAMAPRTEINSTRING' ❺
```

❶ I usually test strings parameters with the empty string.

❷ Test for a single amino acid.

❸ Test using a stop codon before the end of the sequence.

❹ Notice that adjacent string literals are joined into a single string. This is a useful way to break long lines in source code.

❺ Test using the example from Rosalind.

I move all the code from `main()` into this, changing the `for` loop to a list comprehension and using a list slice to truncate the protein at the stop codon:

```
def translate(rna: str) -> str:
    """ Translate codon sequence """

    codon_to_aa = {
        'AAA': 'K', 'AAC': 'N', 'AAG': 'K', 'AAU': 'N', 'ACA': 'T',
        'ACC': 'T', 'ACG': 'T', 'ACU': 'T', 'AGA': 'R', 'AGC': 'S',
        'AGG': 'R', 'AGU': 'S', 'AUA': 'I', 'AUC': 'I', 'AUG': 'M',
        'AUU': 'I', 'CAA': 'Q', 'CAC': 'H', 'CAG': 'Q', 'CAU': 'H',
        'CCA': 'P', 'CCC': 'P', 'CCG': 'P', 'CCU': 'P', 'CGA': 'R',
        'CGC': 'R', 'CGG': 'R', 'CGU': 'R', 'CUA': 'L', 'CUC': 'L',
        'CUG': 'L', 'CUU': 'L', 'GAA': 'E', 'GAC': 'D', 'GAG': 'E',
        'GAU': 'D', 'GCA': 'A', 'GCC': 'A', 'GCG': 'A', 'GCU': 'A',
        'GGA': 'G', 'GGC': 'G', 'GGG': 'G', 'GGU': 'G', 'GUA': 'V',
        'GUC': 'V', 'GUG': 'V', 'GUU': 'V', 'UAC': 'Y', 'UAU': 'Y',
        'UCA': 'S', 'UCC': 'S', 'UCG': 'S', 'UCU': 'S', 'UGC': 'C',
```

```
        'UGG': 'W', 'UGU': 'C', 'UUA': 'L', 'UUC': 'F', 'UUG': 'L',
        'UUU': 'F', 'UAA': '*', 'UAG': '*', 'UGA': '*',
    }

    aa = [codon_to_aa.get(codon, '-') for codon in codons(rna, 3)] ❶
    if '*' in aa: ❷
        aa = aa[:aa.index('*')] ❸

    return ''.join(aa) ❹
```

❶ Use a list comprehension to translate the list of codons to a list of amino acids.

❷ See if the stop (*) codon is present in the list of AAs.

❸ Overwrite the AAs using a list slice up to the index of the stop codon.

❹ Join the amino acids on the empty string and return the new protein sequence.

To understand this, consider the following RNA sequence:

```
>>> rna = 'AUGCCGUAAUCU'
```

I can use the `codons()` function to get the codons:

```
>>> from solution3_list_comp_slice import codons, translate
>>> codons(rna, 3)
['AUG', 'CCG', 'UAA', 'UCU']
```

And use a list comprehension to turn those into amino acids:

```
>>> codon_to_aa = {
...     'AAA': 'K', 'AAC': 'N', 'AAG': 'K', 'AAU': 'N', 'ACA': 'T',
...     'ACC': 'T', 'ACG': 'T', 'ACU': 'T', 'AGA': 'R', 'AGC': 'S',
...     'AGG': 'R', 'AGU': 'S', 'AUA': 'I', 'AUC': 'I', 'AUG': 'M',
...     'AUU': 'I', 'CAA': 'Q', 'CAC': 'H', 'CAG': 'Q', 'CAU': 'H',
...     'CCA': 'P', 'CCC': 'P', 'CCG': 'P', 'CCU': 'P', 'CGA': 'R',
```

```
...     'CGC': 'R', 'CGG': 'R', 'CGU': 'R', 'CUA': 'L', 'CUC': 'L',
...     'CUG': 'L', 'CUU': 'L', 'GAA': 'E', 'GAC': 'D', 'GAG': 'E',
...     'GAU': 'D', 'GCA': 'A', 'GCC': 'A', 'GCG': 'A', 'GCU': 'A',
...     'GGA': 'G', 'GGC': 'G', 'GGG': 'G', 'GGU': 'G', 'GUA': 'V',
...     'GUC': 'V', 'GUG': 'V', 'GUU': 'V', 'UAC': 'Y', 'UAU': 'Y',
...     'UCA': 'S', 'UCC': 'S', 'UCG': 'S', 'UCU': 'S', 'UGC': 'C',
...     'UGG': 'W', 'UGU': 'C', 'UUA': 'L', 'UUC': 'F', 'UUG': 'L',
...     'UUU': 'F', 'UAA': '*', 'UAG': '*', 'UGA': '*',
... }
>>> aa = [codon_to_aa.get(c, '-') for c in codons(rna, 3)]
>>> aa
['M', 'P', '*', 'S']
```

I can see the stop codon is present:

```
>>> '*' in aa
True
```

and so the sequence needs to be truncated at index 2:

```
>>> aa.index('*')
2
```

I can use a list slice to select up to the position of the stop codon. If no *start* position is supplied, then Python assumes 0 or the start of the list (or string):

```
>>> aa = aa[:aa.index('*')]
>>> aa
['M', 'P']
```

Finally, this list needs to be joined on the empty string:

```
>>> ''.join(aa)
'MP'
```

The `main()` incorporates the new function and makes for a very readable program:

```
def main() -> None:
    args = get_args()
    print(translate(args.rna.upper()))
```

This is another instance where the unit test almost duplicates the integration test. The latter is still important as it ensures that the program works, produces documentation, handles the arguments, and so forth. As overengineered as this solution may seem, I want you to focus on how to break programs into smaller *functions* that you can understand, test, compose, and share.

Solution 4: Functional Programming with the map, partial, and takewhile Functions

For this next solution, I'd want to highlight how to rewrite some of the logic using three higher-order functions, `map()`, `partial()`, and `takewhile()`. Figure 7-5 shows how the list comprehension can be rewritten as a `map()`.

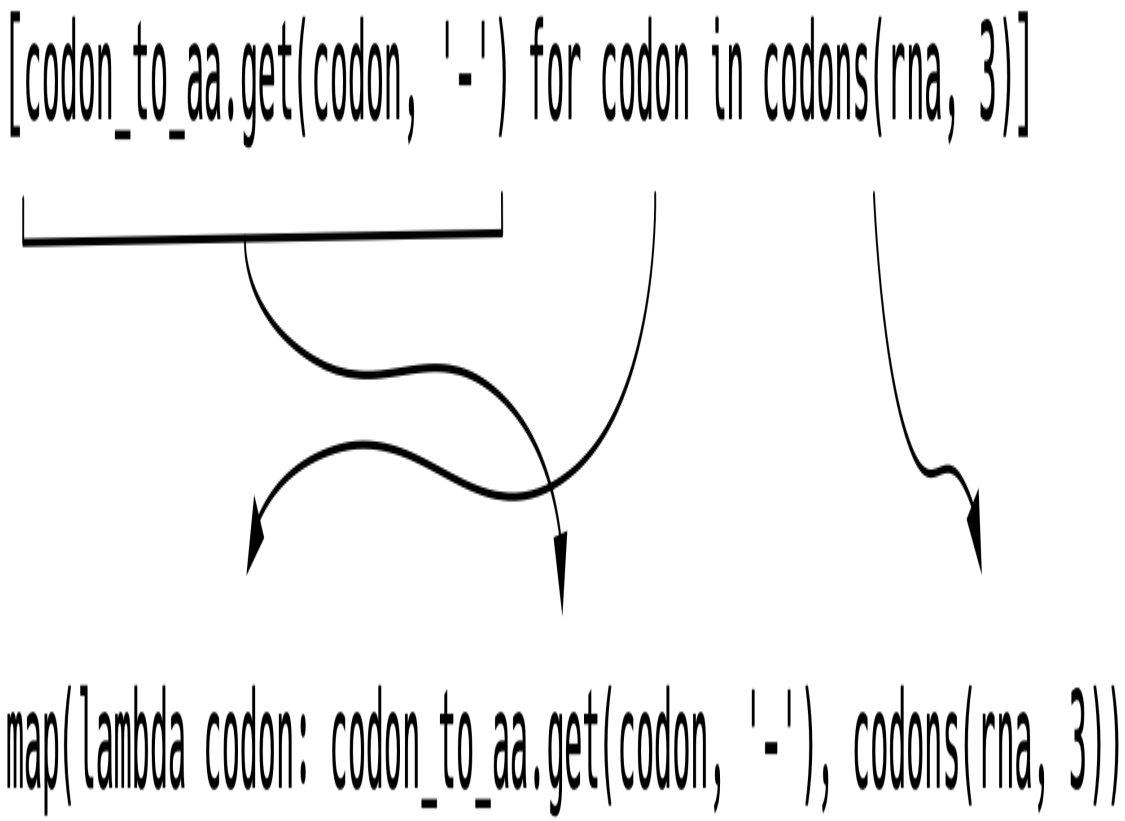


Figure 7-5. A list comprehension can be rewritten as a `map()`.

I can use the `map()` to get the AA sequence:

```
>>> aa = list(map(lambda codon: codon_to_aa.get(codon, '-'),
codons(rna, 3)))
>>> aa
['M', 'P', '*', 'S']
```

The code to find the stop codon and slice the list can be rewritten using the `itertools.takewhile()` function:

```
>>> from itertools import takewhile
```

As the name implies, this function will *take* elements from a sequence *while* the predicate is met. Once the predicate fails, the

function stops accepting values. Here the condition is that the residue is not * (stop):

```
>>> list(takewhile(lambda residue: residue != '*', aa))
['M', 'P']
```

If you like using these kinds of HOFs, you can take this even further by using the `functools.partial()` function I showed in Chapter 4. Again, I want to partially apply the `operator.ne()` (not equal) function:

```
>>> from functools import partial
>>> import operator
>>> not_stop = partial(operator.ne, '*')
```

Remember, `not_stop()` is a function that needs one more string value before it can return a value:

```
>>> not_stop('F')
True
>>> not_stop('*')
False
```

I can use it like so:

```
>>> list(takewhile(not_stop, aa))
['M', 'P']
```

Here is how I would write the `translate()` function with these ideas:

```
def translate(rna: str) -> str:
    """ Translate codon sequence """

    codon_to_aa = {
        'AAA': 'K', 'AAC': 'N', 'AAG': 'K', 'AAU': 'N', 'ACA': 'T',
        'ACC': 'T', 'ACG': 'T', 'ACU': 'T', 'AGA': 'R', 'AGC': 'S',
        'AGG': 'R', 'AGU': 'S', 'AUA': 'I', 'AUC': 'I', 'AUG': 'M',
        'AUU': 'I', 'CAA': 'Q', 'CAC': 'H', 'CAG': 'Q', 'CAU': 'H',
        'CCA': 'P', 'CCC': 'P', 'CCG': 'P', 'CCU': 'P', 'CGA': 'R',
```

```
        'CGC': 'R', 'CGG': 'R', 'CGU': 'R', 'CUA': 'L', 'CUC': 'L',
        'CUG': 'L', 'CUU': 'L', 'GAA': 'E', 'GAC': 'D', 'GAG': 'E',
        'GAU': 'D', 'GCA': 'A', 'GCC': 'A', 'GCG': 'A', 'GCU': 'A',
        'GGA': 'G', 'GGC': 'G', 'GGG': 'G', 'GGU': 'G', 'GUA': 'V',
        'GUC': 'V', 'GUG': 'V', 'GUU': 'V', 'UAC': 'Y', 'UAU': 'Y',
        'UCA': 'S', 'UCC': 'S', 'UCG': 'S', 'UCU': 'S', 'UGC': 'C',
        'UGG': 'W', 'UGU': 'C', 'UUA': 'L', 'UUC': 'F', 'UUG': 'L',
        'UUU': 'F', 'UAA': '*', 'UAG': '*', 'UGA': '*',
    }

    aa = map(lambda codon: codon_to_aa.get(codon, '-'), codons(rna,
3))
    return ''.join(takewhile(partial(operator.ne, '*'), aa))
```

Solution 5: Using Bio.Seq

As promised, the last solution uses Biopython. In Chapter 3, I used the `Bio.Seq.reverse_complement()` function, and here I can use `Bio.Seq.translate()`. First, import the `Bio.Seq` class:

```
>>> from Bio import Seq
```

Then call the translate function:

```
>>> rna = 'AUGGCCAUGGCGCCCAGAACUGAGAUCAAUAGUACCCGUAUUAACGGGUGA'
>>> Seq.translate(rna)
'MAMAPRTEINSTRING*'
```

As with my dictionary, the stop codon is represented by `*`. Note that the function does not stop translating at the stop codon by default:

```
>>> Seq.translate('AUGCCGUAAUCU')
'MP*S'
```

If you read `help(Seq.translate)` in the REPL, you'll find the `to_stop` option to fix this:

```
>>> Seq.translate('AUGCCGUAAUCU', to_stop=True)
'MP'
```

Here is how I put it all together:

```
def main() -> None:
    args = get_args()
    print(Seq.translate(args.rna, to_stop=True))
```

This is the solution I would recommend, again because it relies on the widely used Biopython module. While it was fun and enlightening to explore how to manually code a solution, it's far better practice to use code that's already been written and tested by a dedicated team of developers.

Going Further

- Add a `--frame-shift` argument that defaults to 0 and allows values 0-2 (inclusive). Use the frame shift to start reading the RNA from an alternate position.

Review

The focus of this chapter was really on how to write, test, and compose *functions* alone to solve the problem at hand. I wrote functions to find codons in a sequence and translate RNA. Then I showed how to use higher-order functions to compose various functions, and finally, I used an off-the-shelf function to write the entire body of the program.

- K-mers are all the *k*-length subsequences of a sequence.
- Codons are 3-mers that do not overlap.
- Dictionaries are useful as lookup tables such as translating a codon to an amino acid.
- A `for` loop, a list comprehension, and `map()` are all methods for transforming one sequence into another.

- The `takewhile()` function is similar to the `filter()` function in accepting values from a sequence based on a predicate or test of the values.
- The `partial()` function allows one to partially apply the arguments to a function.
- The `Bio.Seq.translate()` function will translate an RNA sequence into a protein sequence.

Chapter 8. Find a Motif in DNA: Exploring Sequence Similarity

In this Rosalind challenge, I'll be searching for any places where one sequence may occur inside another sequence. A shared subsequence might represent a conserved element such as a marker, gene, or regulatory sequence. Conserved sequences between two organisms might suggest some inherited or convergent trait. I'll explore how to write a solution using the `str` (string) class in Python and will compare strings to lists. Then I'll explore how to express these ideas using higher-order functions and will continue the discussion of k-mers I started in Chapter 7. Finally, I'll show how regular expressions can find patterns and will point out problems with finding overlapping matches.

In this chapter, I'll demonstrate:

- How to use `str.find()`, `str.index()`, and string slices
- How to use sets to create unique collections of elements
- How to combine higher-order functions
- How to find subsequences using k-mers
- How to find possibly overlapping sequences using regular expressions

Getting Started

The code and tests for this chapter are in *08_subs*. I suggest you start by copying the first solution to the program `subs.py` and requesting help:

```
$ cd 08_subs/
$ cp solution1_str_find.py subs.py
$ ./subs.py -h
usage: subs.py [-h] seq subseq

Find subsequences

positional arguments:
  seq         Sequence
  subseq      Sub-sequence

optional arguments:
  -h, --help  show this help message and exit
```

The program should report the starting locations where the subsequence can be found in the sequence. As shown in Figure 8-1, the the subsequence *ATAT* can be found at positions 2, 4, and 10 in the sequence *GATATATGCATATACTT*:

```
$ ./subs.py GATATATGCATATACTT ATAT
2 4 10
```

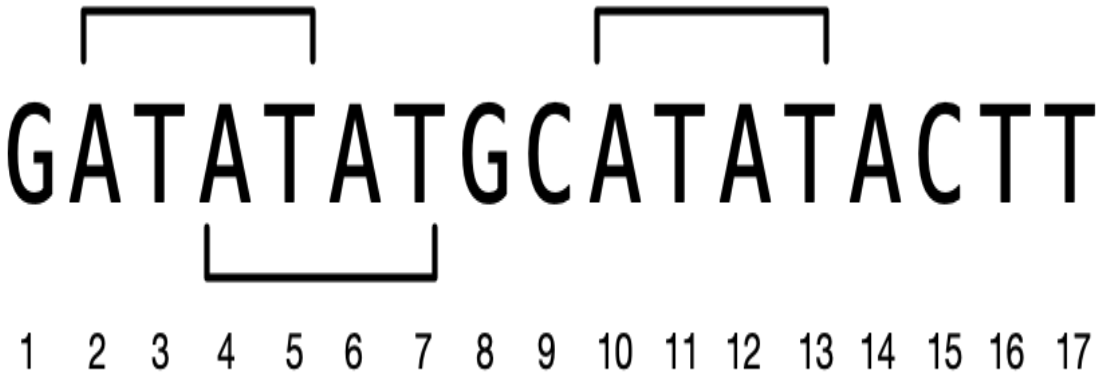


Figure 8-1. The subsequence ATAT can be found at positions 2, 4, and 10.

Run the tests to see if you understand what all will be expected, then start your program from scratch:

```
$ new.py -fp 'Find subsequences' subs.py
Done, see new script "subs.py".
```

Modify the parameters so that your program will print the correct usage, and then change your `main()` to print the sequence and subsequence:

```
def main() -> None:
    args = get_args()
    print(f'sequence = {args.seq}')
    print(f'subsequence = {args.subseq}')
```

Run the program with the expected inputs and verify it prints the arguments correctly:

```
$ ./subs.py GATATATGCATATACTT ATAT
sequence = GATATATGCATATACTT
subsequence = ATAT
```

Now you have a program that should pass the first two tests. If you think you can finish this on your own, please proceed; otherwise, I'll show you one way to find the location of one string inside another.

Finding Subsequences

I'll start by defining the following sequence and subsequence:

```
>>> seq = 'GATATATGCATATACTT'
>>> subseq = 'ATAT'
```

I can use `in` to find if one sequence is a subset of another. This also works for membership in lists or sets or keys in a dictionary:

```
>>> subseq in seq
True
```

That's good information, but it doesn't tell me *where* the string can be found. Luckily there's the `str.find()` function that says `subseq` can be found starting at index 1 (which is actually the 2nd character):

```
>>> seq.find(subseq)
1
```

I know from the Rosalind description that the answer should be 2, 4, and 10. I just found 2, so how can I find the next? I can't just call the same function again because I'll get the same answer. I need to look farther into the sequence. Maybe `help(str.find)` could be of some use?

```
>>> help(str.find)
find(...)
    S.find(sub[, start[, end]]) -> int

    Return the lowest index in S where substring sub is found,
    such that sub is contained within S[start:end].  Optional
    arguments start and end are interpreted as in slice notation.

    Return -1 on failure.
```

It appears I can specify a *start* position. I'll use 1 greater than the position where the first subsequence was found (which was 1 so starting at 2)

```
>>> seq.find(subseq, 2)
3
```

Great. That's the next answer — well, 4 is the next answer, but you know what I mean. I'll try that again, this time starting at 4:

```
>>> seq.find(subseq, 4)
9
```

That was the last value I expected. What happens if I try using a start of 10? As the documentation shows, this will return -1 to indicate the subsequence cannot be found:

```
>>> seq.find(subseq, 10)
-1
```

Can you think of a way to iterate through the sequence, remembering the last position where the subsequence was found until it cannot be found?

Another option would be to use `str.index()` but only if the subsequence is present:

```
>>> if subseq in seq:
...     seq.index(subseq)
...
1
```

To find the next occurrence, you could slice the sequence using the last known position. You'll have to add this position to the starting position, but you're essentially doing the same operation of moving further into the sequence to find if the subsequence is present and where:

```
>>> if subseq in seq[2:]:
...     seq.index(subseq[2:])
...
1
```

If you read `help(str.index)`, you'll see that, like `str.find()`, the function takes a second optional *start* position of the index to start looking:

```
>>> if subseq in seq[2:]:
...     seq.index(subseq, 2)
...
3
```

A different approach would be to use k-mers. If the subsequence is present in the sequence, then it is by definition a k-mer where *k* is the length of the subsequence. Try extracting all the k-mers from the sequence and comparing them to the subsequence.

Finally, since I'm looking for a pattern of text, I could use a regular expression. In Chapter 5, I showed the `re.findall()` function to find all the Gs and Cs in DNA. I can similarly use this method to find all the subsequences in the sequence:

```
>>> import re
>>> re.findall(subseq, seq)
['ATAT', 'ATAT']
```

That seems to have a couple of problems. One, it only returned two of the subsequences when I know there are three. The other problem is that this provides no information about *where* the matches are found. Fear not, the `re.finditer()` function solves this second problem:

```
>>> list(re.finditer(subseq, seq))
[<re.Match object; span=(1, 5), match='ATAT'>,
 <re.Match object; span=(9, 13), match='ATAT'>]
```

Now it's apparent that it finds the first and last subsequences. Why doesn't it find the second instance? It turns out regular expressions don't handle overlapping patterns very well, but some additions to the search pattern can fix this. I'll leave it to you and some internet searching to see if you can figure out a solution.

I've presented four different options on how to solve this problem. See if you can write solutions using each approach. The point is to explore the corners of Python, storing away tasty bits and tricks that might prove decisive in some future program you write. It's OK to spend hours or days working this out. Keep at it until you have solutions that pass both `pytest` and `make test`.

Solutions

All of the solutions will share the same start, so I'll present that here:

```
#!/usr/bin/env python3
""" Find subsequences """

import argparse
from typing import NamedTuple


class Args(NamedTuple): ❶
    """ Command-line arguments """
    seq: str
    subseq: str


# --------------------------------------------------
def get_args() -> Args: ❷
    """ Get command-line arguments """

    parser = argparse.ArgumentParser(
        description='Find subsequences',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('seq', metavar='seq', help='Sequence')

    parser.add_argument('subseq', metavar='subseq', help='Sub-
sequence')

    args = parser.parse_args()

    return Args(args.seq, args.subseq) ❸
```

❶ The `Args` will have two fields for `seq` and `subseq`.

❷ The function returns an `Args` object.

❸ Package and return the arguments using `Args`.

Solution 1: Using the str.find Method

Here is my first solution using the `str.find()` method:

```
def main() -> None:
    args = get_args()
```

```
last = 0 ❶
found = [] ❷
while True: ❸
    pos = args.seq.find(args.subseq, last) ❹
    if pos == -1: ❺
        break ❻
    found.append(pos + 1) ❼
    last = pos + 1 ❽

print(*found) ❾
```

❶ Initialize the *last* position to 0, the start of the sequence.

❷ Initialize a list to hold all the positions where the subsequence is found.

❸ Create an infinite loop using `while`.

❹ Use `str.find()` to look for the subsequence using the last known position.

❺ Check if the return is -1 meaning the subsequence is not found.

❻ Break out of the loop.

❼ Append 1 greater than the index to the list of found positions.

❽ Update the last known position with 1 greater than the found index.

❾ Print the found positions using `*` to expand the list into the elements. The function will use a space to separate multiple values.

This solution turns on keeping track of the *last* place the subsequence was found. I initialize this to 0:

```
>>> last = 0
```

I use this with `str.find()` to look for the subsequence starting at the last known position:

```
>>> seq = 'GATATATGCATATACTT'
>>> subseq = 'ATAT'
>>> pos = seq.find(subseq, last)
>>> pos
1
```

When the function returns a value greater than 0, I update the last position to one greater to search starting at the next character:

```
>>> last = pos + 1
>>> pos = seq.find(subseq, last)
>>> pos
3
```

Another call to the function finds the last instance:

```
>>> last = pos + 1
>>> pos = seq.find(subseq, last)
>>> pos
9
```

And finally, the function returns -1 to indicate that the pattern can no longer be found:

```
>>> last = pos + 1
>>> pos = seq.find(subseq, last)
>>> pos
-1
```

This solution would be immediately understandable to someone with a background in the C programming language. It's a very *imperative* approach with lots of detailed logic for updating the state of the algorithm. *State* is how data in our programs change over time. For instance, properly updating and using the last known position is key

to making this approach work. Later approaches use far less explicit coding.

Solution 2: Using the str.index Method

This next solution is a variation that slices the sequence using the last known position:

```
def main() -> None:
    args = get_args()
    seq, subseq = args.seq, args.subseq ❶
    found = []
    last = 0
    while subseq in seq[last:]: ❷
        last = seq.index(subseq, last) + 1 ❸
        found.append(last) ❹

    print(' '.join(map(str, found))) ❺
```

❶ Unpack the sequence and subsequence.

❷ Ask if the subsequence appears in a slice of the sequence starting at the last found position. The `while` loop will execute as long as this condition is true.

❸ Use `str.index()` to get the starting position of the subsequence. The `last` variable gets updated by adding 1 to the sub-sequence index to create the next starting position.

❹ Append this position to the list of found positions.

❺ Use `map()` to coerce all the found integer positions to strings, then join them on spaces to print.

Here again, I rely on tracking the last place a subsequence was found. I start at 0 or the beginning of the string:

```
>>> last = 0
>>> if subseq in seq[last:]:
...     last = seq.index(subseq, last) + 1
...
>>> last
2
```

The `while True` loop in the first solution is a common way to start an infinite loop. Here, the `while` loop will only execute as long as the subsequence is found in the slice of the sequence, meaning the algorithm doesn't need to decide when to break out of the loop:

```
>>> last = 0
>>> found = []
>>> while subseq in seq[last:]:
...     last = seq.index(subseq, last) + 1
...     found.append(last)
...
>>> found
[2, 4, 10]
```

The found positions, in this case, are a list of integer values. In the first solution, I used `*found` to splat the list and relied on `print()` to coerce the values to strings and join them on spaces. If instead I were to try to create a new string from `found` using `str.join()`, I would run into problems. The `str.join()` function joins many *strings* into a single string and so raises an exception when you give it non-string values:

```
>>> ' '.join(found)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sequence item 0: expected str instance, int found
```

I could use a list comprehension to turn each number `n` into a string using the `str()` function:

```
>>> ' '.join([str(n) for n in found])
'2 4 10'
```

This can also be written using a `map()`:

```
>>> ' '.join(map(lambda n: str(n), found))
'2 4 10'
```

I can leave out the `lambda` entirely because the `str()` function expects a single argument, and `map()` will naturally supply each value from `found` as the argument to `str()`. This is my preferred way to turn a list of integers into a list of strings:

```
>>> ' '.join(map(str, found))
'2 4 10'
```

Solution 3: A Purely Functional Approach

This next solution combines many of the preceding ideas using a purely functional approach. To start, consider the `while` loops in the first two solutions used to append non-negative values to the *found* list. Does that sound like something a list comprehension could do? The range of values to iterate include all the positions from 0 to the end of the sequence minus the length of the subsequence:

```
>>> r = range(len(seq) - len(subseq))
>>> list(r)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

A list comprehension can use these values with `str.find()` to search for the subsequence in the sequence starting at these positions:

```
>>> [seq.find(subseq, n) for n in r]
[1, 1, 3, 3, 9, 9, 9, 9, 9, 9, -1, -1, -1]
```

I only want the non-negative values, and `filter()` can remove these:

```
>>> list(filter(lambda n: n >= 0, [seq.find(subseq, n) for n in r]))
[1, 1, 3, 3, 9, 9, 9, 9, 9, 9]
```

Which could also be written like so:

```
>>> list(filter(lambda n: 0 <= n, [seq.find(subseq, n) for n in r]))
[1, 1, 3, 3, 9, 9, 9, 9, 9, 9]
```

I show you this because I'd like to use `partial()` with the `operator.le()` (less than or equal) function because I don't really like `lambda` expressions:

```
>>> from functools import partial
>>> import operator
>>> ok = partial(operator.le, 0)
>>> list(filter(ok, [seq.find(subseq, n) for n in r]))
[1, 1, 3, 3, 9, 9, 9, 9, 9, 9]
```

I'd like to change the list comprehension to a `map()`:

```
>>> list(filter(ok, map(lambda n: seq.find(subseq, n), r)))
[1, 1, 3, 3, 9, 9, 9, 9, 9, 9]
```

but again I want to get rid of the `lambda` by using `partial()`:

```
>>> find = partial(seq.find, subseq)
>>> list(filter(ok, map(find, r)))
[1, 1, 3, 3, 9, 9, 9, 9, 9, 9]
```

I can use `set()` to get a distinct list:

```
>>> set(filter(ok, map(find, r)))
{1, 3, 9}
```

These are almost the correct values, but they are the *index* positions which are zero-based. I need the values one greater, so I can make a function to add 1 and apply this using a `map()`:

```
>>> add1 = partial(operator.add, 1)
>>> list(map(add1, set(filter(ok, map(find, r)))))
[2, 4, 10]
```

In these limited examples, the results are properly sorted; however, one can never rely on the order of values from a set. I must use the `sorted()` function to ensure they are properly sorted numerically:

```
>>> sorted(map(add1, set(filter(ok, map(find, r)))))
[2, 4, 10]
```

Finally, I need to `print()` these values which still exist as a list of integers:

```
>>> print(sorted(map(add1, set(filter(ok, map(find, r))))))
[2, 4, 10]
```

That's almost right. As in the first solution, I need to splat the results to get `print()` to see the individual elements:

```
>>> print(*sorted(map(add1, set(filter(ok, map(find, r))))))
2 4 10
```

That's a lot of closing parentheses. This code is starting to look a little like Lisp. If you combine all these ideas, you wind up with the same answer as the imperative solution but now by combining only functions:

```
def main() -> None:
    args = get_args()
    seq, subseq = args.seq, args.subseq ❶
    r = list(range(len(seq) - len(subseq))) ❷
    ok = partial(operator.le, 0) ❸
    find = partial(seq.find, subseq) ❹
    add1 = partial(operator.add, 1) ❺
    print(*sorted(map(add1, set(filter(ok, map(find, r)))))) ❻
```

❶ Unpack the sequence and subsequence.

❷ Generate a range of numbers up to the length of the sequence less the length of the subsequence.

❸ Create a partial `ok()` function that will return `True` if a given number is greater than or equal to 0.

❹ Create a partial `find()` function that will look for the subsequence in the sequence when provided with a *start* parameter.

❺ Create a partial `add1()` function that will return 1 greater than the argument.

❻ Apply all the numbers from the range to the `find()` function, filter out negative values, unique the results, add 1 to the values, and sort the numbers before printing.

This solution uses only *pure* functions and would be fairly easy to understand for a person with a background in the Haskell programming language. If it seems like a jumble of confusion to you, I'd encourage you to spend some time working in the REPL with each piece until you understand how all these functions fit together perfectly.

Solution 4: Using kmers

I mentioned that you might try finding the answer using k-mers, which I showed in Chapter 7. If the subsequence exists in the sequence, then it must be a k-mer where *k* equals the length of the subsequence:

```
>>> seq = 'GATATATGCATATACTT'
>>> subseq = 'ATAT'
>>> k = len(subseq)
>>> k
4
```

Here are all the 4-mers in the sequence:

```
>>> kmers = [seq[i:i + k] for i in range(len(seq) - k + 1)]
>>> kmers
['GATA', 'ATAT', 'TATA', 'ATAT', 'TATG', 'ATGC', 'TGCA', 'GCAT',
'CATA',
 'ATAT', 'TATA', 'ATAC', 'TACT', 'ACTT']
```

Here are the 4-mers that are the same as the subsequence I'm looking for:

```
>>> list(filter(lambda s: s == subseq, kmers))
['ATAT', 'ATAT', 'ATAT']
```

I need to know the positions as well as the k-mers. The `enumerate()` function will return both the index and value of all the elements in a sequence. Here are the first four:

```
>>> kmers = list(enumerate([seq[i:i + k] for i in range(len(seq) - k +
1)]))
>>> kmers[:4]
[(0, 'GATA'), (1, 'ATAT'), (2, 'TATA'), (3, 'ATAT')]
```

I can use this with `filter()`, but now the `lambda` is receiving a tuple of the index and value so I will need to look at the second field (which is in index 1):

```
>>> list(filter(lambda t: t[1] == subseq, kmers))
[(1, 'ATAT'), (3, 'ATAT'), (9, 'ATAT')]
```

I only care about getting the *index* for the matching k-mers. I could rewrite this using a `map()` with an if-expression to return the index position when it's a match and `None` otherwise:

```
>>> list(map(lambda t: t[0] if t[1] == subseq else None, kmers))
[None, 1, None, 3, None, None, None, None, None, 9, None, None, None,
None]
```

I find using the tuple difficult to understand, so I'll import the `starmap` function so I can unpack the tuple into good names. I can also add 1 to the index (`i`):

```
>>> from itertools import starmap
>>> list(starmap(lambda i, kmer: i + 1 if kmer == subseq else None,
kmers))
[None, 2, None, 4, None, None, None, None, None, 10, None, None, None,
None]
```

This probably seems like an odd choice until I show you that `filter()`, if passed `None` for the `lambda`, will use the truthiness of each value so that `None` values will be excluded. Because this line of code is getting rather long, I'll break the function `f` to `map()` into a separate line:

```
>>> f = lambda i, kmer: i + 1 if kmer == subseq else None
>>> list(filter(None, starmap(f, kmers)))
[2, 4, 10]
```

I can also express a k-mer solution using imperative techniques:

```
def main() -> None:
    args = get_args()
    seq, subseq = args.seq, args.subseq
    k = len(subseq) ❶
    kmers = [seq[i:i + k] for i in range(len(seq) - k + 1)] ❷
    found = [i + 1 for i, kmer in enumerate(kmers) if kmer == subseq]
❸
    print(*found) ❹
```

❶ When looking for k-mers, *k* is the length of the subsequence.

❷ Use a list comprehension to generate all the k-mers from the sequence.

❸ Iterate through the index and value of all the k-mers where the k-mer is equal to the subsequence. Return 1 greater than the index

position.

❹ Print the found positions.

Or I can use purely functional ideas. Note that `mypy` insisted on a type annotation for the `found` variable:

```
def main() -> None:
    args = get_args()
    seq, subseq = args.seq, args.subseq
    k = len(subseq)
    kmers = enumerate(seq[i:i + k] for i in range(len(seq) - k + 1)) ❶
    found: Iterator[int] = filter( ❷
        None, starmap(lambda i, kmer: i + 1 if kmer == subseq else
None, kmers))
    print(*found) ❸
```

❶ Generate an enumerated list of the kmers.

❷ Select the positions of those k-mers equal to the subsequence.

❸ Print the results.

Whichever solution you prefer, the interesting point is that k-mers can prove extremely useful in many situations such as partial sequence comparison. While alignment tools like BLAST can never (likely) be replaced in bioinformatics, k-mer comparisons have proven far faster and extremely accurate.

Solution 5: Finding Overlapping Patterns Using Regular Expressions

To this point, I've been writing fairly complex solutions to find a pattern of characters inside a string. This is precisely the domain of regular expressions, and so it's actually a bit silly to write manual solutions. I showed in the introduction to this chapter that the

`re.finditer()` function does not find overlapping matches and so returns just 2 hits when I know there are 3:

```
>>> import re
>>> list(re.finditer(subseq, seq))
[<re.Match object; span=(1, 5), match='ATAT'>,
 <re.Match object; span=(9, 13), match='ATAT'>]
```

> **NOTE**
>
> I'm going to show you that the solution is quite simple, but I want to stress that I did not know the solution until I researched the answer for this chapter. Further, I found the answer by searching the internet and reading several answers. The key to finding the answer was knowing what search terms to use — something like *regex overlapping patterns* turns up several useful results. The point of this aside is that no one knows all the answers, and you will constantly be searching for solutions to problems you never knew even existed. It's not what you know that's important but what you can learn.

The problem turns out to be that the regex engine *consumes* strings as they match. Once the engine matches the first *ATAT*, it starts searching at the end of the match. The solution is to wrap the search pattern in a *look-ahead assertion* so that the engine won't consume the matched string. Note that this is a *positive* look-ahead, and there are also *negative* look-ahead assertion as well as both positive and negative look-behind assertions.

That is, if the subsequence is *ATAT*, then I want the pattern to be `?=(ATAT)`. The problem now is that the regex engine won't *save* the match — I've just told it to look for this pattern but haven't told it to do anything with the text that is found. I need to further wrap the assertion in parentheses to create a *capture group*:

```
>>> list(re.finditer('(?=(ATAT))', 'GATATATGCATATACTT'))
[<re.Match object; span=(1, 1), match=''>,
 <re.Match object; span=(3, 3), match=''>,
 <re.Match object; span=(9, 9), match=''>]
```

I can use a list comprehension over this iterator to call the `match.start()` function on each of the `re.Match` objects, adding 1 to correct the position:

```
>>> [match.start() + 1 for match in re.finditer(f'(?=({subseq}))',
seq)]
[2, 4, 10]
```

Here is the final solution that I would suggest as the best way to solve this problem:

```
def main() -> None:
    args = get_args()
    seq, subseq = args.seq, args.subseq
    print(*[m.start() + 1 for m in re.finditer(f'(?=({subseq}))',
seq)])
```

Going Further

Expand the program to look for a subsequence *pattern* in a sequence. For example, you might search for simple sequence repeats (AKA *SSRs* or _microsattelites) such as “GA(26)” which would means “GA” repeated 26 times or “(GA)15GT(GA)2” which means “GA repeated 15 times followed by GT followed by GA repeated 2 times.”

Review

- The `str.find()` and `str.index()` methods can determine is a subsequence is present in a given string.
- Sets can be used to create unique collections of elements.
- K-mers are by definition subsequences and are relatively quick to extract and compare.

- Regular expressions can find overlapping sequences by using a look-ahead assertion combined with a capture group.

Chapter 9. Overlap Graphs: Sequence Assembly Using Shared K-mers

A *graph* is a structure used to represent pair-wise relationships between objects. As described in the Rosalind GRPH challenge, the goal is to find pairs of sequences that can be joined using some overlap from the end of one sequence to the beginning of another. The practical application of this would be to join short DNA reads into longer contiguous sequences (*contigs*) or even whole genomes. To begin, I'll only be concerned about joining two sequences, but a second version of the program will use a graph structure that can join any number of sequences to approximate a complete assembly. In this implementation, the overlapping regions used to join sequences are required to be exact matches. Real-world assemblers must allow for variation in the size and composition of the overlapping sequences.

You will learn:

- How to use k-mers to create overlap graphs
- How to log runtime messages to a file
- How to use `defaultdict()`
- How to use set intersection to find common elements between collections
- How to use `product()` to create all combinations of elements
- How to use the `starfilter()` function
- How to use Graphviz to model and visualize graph structures

Getting Started

The code and tests for this exercise are in the *09_grph* directory. Start by copying one of the solutions to the program `grph.py` and requesting the usage:

```
$ cd 09_grph/
$ cp solution1.py grph.py
$ ./grph.py -h
usage: grph.py [-h] [-k size] [-d] FILE

Overlap Graphs

positional arguments:
  FILE                  FASTA file ❶

optional arguments:
  -h, --help            show this help message and exit
  -k size, --overlap size
                        Size of overlap (default: 3) ❷
  -d, --debug           Debug (default: False) ❸
```

❶ The positional parameter is a required FASTA-formatted file of sequences.

❷ The *k* option controls the length of the overlapping strings and defaults to 3.

❸ This is a *flag* or Boolean parameter. When present, the value will be `True` and `False` otherwise.

The sample input shown on the Rosalind page is also the contents of the first sample input file:

```
$ cat tests/inputs/1.fa
>Rosalind_0498
AAATAAA
>Rosalind_2391
AAATTTT
```

```
>Rosalind_2323
TTTTCCC
>Rosalind_0442
AAATCCC
>Rosalind_5013
GGGTGGG
```

The Rosalind problem always assumes an overlapping window of three bases. I see no reason for this parameter to be hard-coded, so my version includes a *k* parameter to indicate the size of the overlap window. When *k* is the default value of 3, for instance, three pairs of the sequences can be joined:

```
$ ./grph.py tests/inputs/1.fa
Rosalind_2391 Rosalind_2323
Rosalind_0498 Rosalind_2391
Rosalind_0498 Rosalind_0442
```

Figure 9-1 shows how these sequences overlap by three common bases:

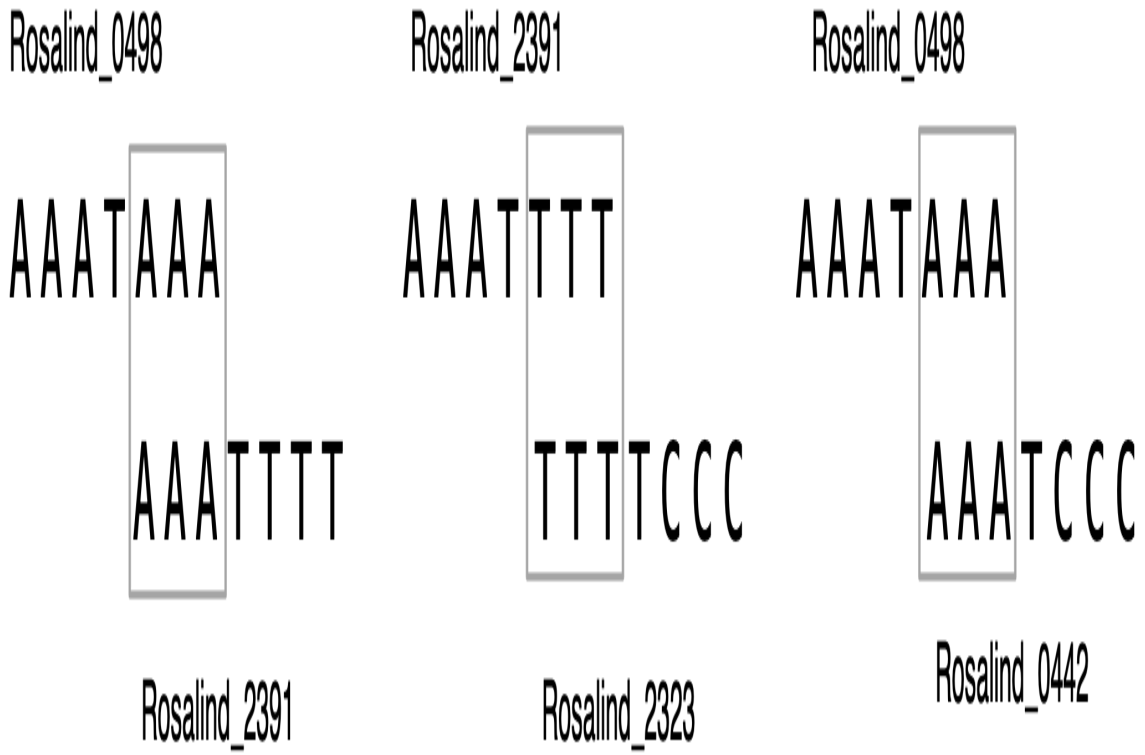


Figure 9-1. Three pairs of sequences form overlap graphs when joining on 3-mers.

As shown in Figure 9-2, only one of these pairs can be joined when the overlap window increases to four bases:

```
$ ./solution1.py -k 4 tests/inputs/1.fa
Rosalind_2391 Rosalind_2323
```

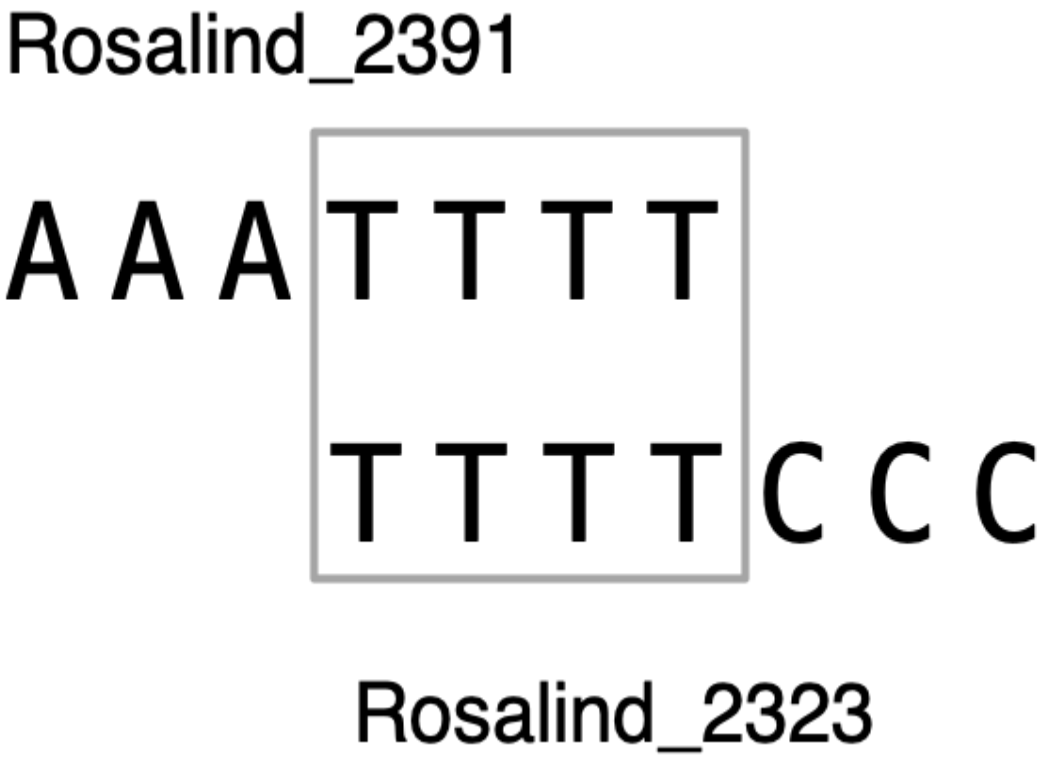


Figure 9-2. Only one pair of sequences form overlap graphs when joining on 4-mers.

Finally, the `--debug` option is a *flag*, a Boolean parameter that has a `True` value when the argument is present and `False` otherwise. When present, this option instructs the program to print runtime logging messages to a file called *.log* in the current working directory. This is not a requirement of the Rosalind challenge, but I think it's important for you to know how to log messages. To see it in action, run the program with the option:

```
$ ./grph.py --debug tests/inputs/1.fa
Rosalind_2391 Rosalind_2323
Rosalind_0498 Rosalind_2391
Rosalind_0498 Rosalind_0442
```

There should now be a *.log* file with the following contents, the meaning of which will become more apparent later:

```
$ cat .log
DEBUG:root:STARTS
defaultdict(<class 'list'>,
            {'AAA': ['Rosalind_0498', 'Rosalind_2391',
'Rosalind_0442'],
             'GGG': ['Rosalind_5013'],
             'TTT': ['Rosalind_2323']})
DEBUG:root:ENDS
defaultdict(<class 'list'>,
            {'AAA': ['Rosalind_0498'],
             'CCC': ['Rosalind_2323', 'Rosalind_0442'],
             'GGG': ['Rosalind_5013'],
             'TTT': ['Rosalind_2391']})
```

Once you understand how your program should work, start over with a new `grph.py` program:

```
$ new.py -fp 'Overlap Graphs' grph.py
Done, see new script "grph.py".
```

Here is how I define and validate the arguments:

```
from typing import List, NamedTuple, TextIO


class Args(NamedTuple): ❶
    """ Command-line arguments """
    file: TextIO
    k: int
    debug: bool


# --------------------------------------------------
def get_args() -> Args:
    """ Get command-line arguments """

    parser = argparse.ArgumentParser(
        description='Overlap Graphs',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file', ❷
                        metavar='FILE',
                        type=argparse.FileType('rt'),
```

```
                        help='FASTA file')

    parser.add_argument('-k', ❸
                        '--overlap',
                        help='Size of overlap',
                        metavar='size',
                        type=int,
                        default=3)

    parser.add_argument('-d', '--debug', help='Debug',
action='store_true') ❹

    args = parser.parse_args()

    if args.overlap < 1: ❺
        parser.error(f'-k "{args.overlap}" must be > 0') ❻

    return Args(args.file, args.overlap, args.debug) ❼
```

❶ The `Args` contains three fields: a *file* which is a filehandle, a *k* which should be a positive integer, and a *debug* flag which is a Boolean value.

❷ Use the `argparge.FileType` to ensure this is a readable text file.

❸ Define an integer argument that defaults to 3.

❹ Define a Boolean flag that will store a `True` value when present.

❺ Check if the *k* (overlap) value is negative.

❻ Use `parser.error()` to kill the program and generate a useful error message.

❼ Return the validated arguments.

I would like to stress how much is happening in these lines to ensure that the arguments to the program are correct. This work should happen *immediately* in the program logic. I've encountered too many

programs that, for instance, never validate a file argument and, deep in the bowels of the program, finally attempt to open a non-existent file and wind up throwing a cryptic exception that no mere mortal could debug. If you want *reproducible* programs, the first order of business is documenting and validating all the arguments.

Modify your `main()` to the following:

```
def main() -> None:
    args = get_args()
    print(args.file.name)
```

Run your program with the first test input file and verify you see this:

```
$ ./grph.py tests/inputs/1.fa
tests/inputs/1.fa
```

Try running your program with invalid values for *k* and the file input, then run `pytest` to verify your program passes the first four tests. The failing test expects 3 pairs of sequence IDs that can be joined but the program printed the name of the input file. Before I talk about how to create overlap graphs, I want to introduce *logging* as this can prove useful to debugging a program.

Managing Runtime Messages with STDOUT, STDERR, and Logging

I've shown how to `print()` strings and data structures to the console. In fact, you just did it by printing the input filename to verify that the program is working. Printing such messages while one is writing and debugging a program might be called *log-driven development*. This is a simple and effective way to debug a program going back decades[1].

By default, `print()` will emit messages to `STDOUT` (*standard out*), which Python represents using `sys.stdout`. I can use the `print()`

function's `file` option to change this to `STDERR` (*standard error*) by indicating `sys.stderr`. Consider the following Python program:

```
$ cat log.py
#!/usr/bin/env python3

import sys

print('This is STDOUT.') ❶
print('This is also STDOUT.', file=sys.stdout) ❷
print('This is STDERR.', file=sys.stderr) ❸
```

❶ The default `file` value is `STDOUT`.

❷ I can specify standard out using the `file` option.

❸ This will print messages to standard error.

When I run this, it would appear that all output is printed to standard out:

```
$ ./log.py
This is STDOUT.
This is also STDOUT.
This is STDERR.
```

In the `bash` shell, I can separate and capture the two streams, however, using file redirection with `>`. Standard out can be captured using the filehandle 1 and standard error using 2. If you run the following command, you should see no output on the console:

```
$ ./log.py 1>out 2>err
```

There should now be two new files, one called *out* with the two lines that were printed to standard out:

```
$ cat out
This is STDOUT.
This is also STDOUT.
```

and another called *err* with the one line printed to standard error:

```
$ cat err
This is STDERR.
```

Just knowing how to print to and capture these two filehandles may prove sufficient for most of your debugging efforts; however, there are times when you may want more levels of printing beyond two, and you may want to control where these messages are written from your code rather than by using shell redirection.

Enter *logging*, a way to control whether, when, how, and where runtime messages are printed. The Python `logging` module handles all of this for us, so I start by importing this module:

```
import logging
```

For this program, I'll print debugging messages to a file called *.log* (in the current working directory) if the `--debug` flag is present. Modify your `main()` to this:

```
def main() -> None:
    args = get_args()

    logging.basicConfig( ❶
        filename='.log', ❷
        filemode='w', ❸
        level=logging.DEBUG if args.debug else logging.CRITICAL) ❹

    logging.debug('input file = "%s"', args.file.name) ❺
```

❶ This will *globally* affect all subsequent calls to the `logging` module's functions.

❷ All output will be written to the file *.log* in the current working directory. I chose a file starting with a dot so that it will normally be hidden from view.

❸ The output file will be opened with the `w` (*write*) option, meaning it will be *overwritten* on each run. Use the `a` mode to *append*, but be warned that the file will grow for every run and will never be truncated or removed except by you.

❹ This sets the minimum logging level (see Table 9-1). Messages at any level below the set level will be ignored.

❺ Use the `logging.debug()` function to print a message to the log file when the logging level is set to *DEBUG*.

NOTE

In the previous example, I used the older `printf()` style of formatting for the call to `logging.debug()`. The placeholders are noted with symbols like `%s` for a string, and the values to substitute are passed as arguments. You can also use `str.format()` and f-strings for the log message, but `pylint` may suggest you use the `printf()` style.

DOTFILES

Files and directories with names starting with a dot are normally hidden when you use `ls`. You must use the `-a` option to `ls` to see *all* files. I'm naming the log file *.log* so I won't normally see it. You may also notice a *.gitignore* file in this directory. If you look at the contents of this file, it contains filenames and patterns of files and directories I do *not* want `git` to add to my repo. Included is the *.log* file. Whenever you want to be sure data like configuration files, passwords, large sequence files, etc., will not be included by `git add`, put their names (or file globs that would match them) to this file.

A key concept to logging is the notion of logging levels. As shown in Table 9-1, the *critical* level is the highest, and the *debug* level is the lowest. To learn more, I recommend you read the module documentation either with `help(logging)` in the REPL or the module's documentation. For this program, I'll only use the lowest debug setting. When the `--debug` flag is present, the logging level is set to `logging.DEBUG` and all messages to `logging.debug()` are printed in the log file. When the flag is absent, the logging level is set to `logging.CRITICAL` and only messages logged with `logging.critical()` will pass through.

Table 9-1. The logging levels available in Python's `logging` module.

Level	Numeric value
CRITICAL	50
ERROR	40
WARNING	30
INFO	20
DEBUG	10
NOTSET	0

To see this in action, run your program as follows. Note that you can place the `--debug` or `-d` flag *after* the positional argument, too, and `argparse` will correctly understand its meaning. Other argument parsers always require options to be placed *before* positional parameters:

```
$ ./grph.py --debug tests/inputs/1.fa
```

It would appear the program did nothing, but there should now be a *.log* file with the following contents:

```
$ cat .log
DEBUG:root:input file = "tests/inputs/1.fa"
```

Run the program again without the debug flag, and note that the *.log* file is empty as it was overwritten when opened but no content was ever logged. If you were to use the typical print-based debugging technique, then you'd have to find and remove (or comment out) all the `print()` statements in your program to turn off your debugging.

If, instead, you use `logging.debug()`, then you can debug your program while logging at the *debug* level and then deploy your program to only log *critical* messages. Further, you can write the log messages to various locations depending on the environment, and all of this happens *programmatically* inside your code rather than relying on shell redirection to put log messages into the right place.

There are no tests to ensure your program creates log files. This is only to show you how to use logging. Note that calls to functions like `logging.critical()` and `logging.debug()` are controlled by the *global* scope of the `logging` module. I don't generally like programs to be controlled by global settings, but this is one exception I'll make, mostly because I don't have a choice. I encourage you to liberally sprinkly `logging.debug()` calls throughout your code to see the kinds of output you can generate. Consider how you could use logging while writing a program on your laptop versus deploying it to an HPC to run unattended.

Finding Overlaps

The next order of business is to read the input FASTA file. I first showed how to do this in Chapter 5. Again I'll use the `Bio.SeqIO` module for this by adding the following import:

```
from Bio import SeqIO
```

I can modify `main()` to the following (omitting any logging calls):

```
def main() -> None:
    args = get_args()

    for rec in SeqIO.parse(args.file, 'fasta'):
        print(rec.id, rec.seq)
```

And then run this on the first input file to ensure the program works properly:

```
$ ./grph.py tests/inputs/1.fa
Rosalind_0498 AAATAAA
Rosalind_2391 AAATTTT
Rosalind_2323 TTTTCCC
Rosalind_0442 AAATCCC
Rosalind_5013 GGGTGGG
```

NOTE

In each exercise, I try to show how to write a program logically, step-by-step. I want you to learn to make very small changes to your program with some end goal in mind, then run your program to see the output. Also, run the tests to see what needs to be fixed. Taking small steps and running your program often are key elements to learning to code.

Now think about how you might get the first and last *k* bases from each sequence. Could you use the code for extracting k-mers that I first showed in Chapter 7? For instance, try to get your program to print this:

```
$ ./grph.py tests/inputs/1.fa
Rosalind_0498 AAATAAA first AAA last AAA
Rosalind_2391 AAATTTT first AAA last TTT
Rosalind_2323 TTTTCCC first TTT last CCC
Rosalind_0442 AAATCCC first AAA last CCC
Rosalind_5013 GGGTGGG first GGG last GGG
```

Think about which *first* strings match which *end* strings. For instance, sequence 0498 ends with *AAA*, and sequence 0442 starts with *AAA*. These are sequences that can be joined into an overlap graph.

Change the value of *k* to 4:

```
$ ./grph.py tests/inputs/1.fa -k 4
Rosalind_0498 AAATAAA first AAAT last TAAA
Rosalind_2391 AAATTTT first AAAT last TTTT
Rosalind_2323 TTTTCCC first TTTT last TCCC
Rosalind_0442 AAATCCC first AAAT last TCCC
Rosalind_5013 GGGTGGG first GGGT last TGGG
```

Now you can see that only two sequences, 2391 and 2323, can be joined by their overlapping sequence *TTTT*. Vary *k* from 1 to 10 and examining the first and last regions. Do you have enough information to write a solution? If not, let's keep thinking about this.

Grouping Sequences by the Overlap

The `for` loop reads the sequences individually. While reading any one sequence to find the starting and ending overlap regions, I necessarily do not have enough information to say which other sequences can be joined. I'm going to have to create some data structure to hold the overlapping regions for *all* the sequences. Only then can I go back and figure out which ones can be joined. This gets at a key element to sequence assemblers — most need prodigious amounts of memory to gather all the information needed from all the input sequences of which there may be millions to billions.

I chose to use two dictionaries, one for the *start* and one for the *end* regions. I decided the keys would be the *k*-length sequence like *AAA* when *k* is 3 and the values would be a list of the sequence IDs sharing this region. I can use string slices with the value *k* to extract these leading and trailing sequences:

```
>>> k = 3
>>> seq = 'AAATTTT'
>>> seq[:k] ❶
'AAA'
>>> seq[-k:] ❷
'TTT'
```

❶ A slice of the first *k* bases

❷ A slice of the last *k* bases using negative indexing to start from the end of the sequence.

These are k-mers, which I showed in the last chapter. They keep showing up, so it makes sense to write a `find_kmers()` function to extract k-mers from a sequence. I'll start by defining the function's signature:

```
def find_kmers(seq: str, k: int) -> List[str]: ❶
    """ Find k-mers in string """

    return [] ❷
```

❶ The function will accept a string (the sequence) and an integer value *k* and will return a list of strings.

❷ For now, return the empty list.

Now I write a test to imagine how I'd use this function:

```
def test_find_kmers():
    """Test find_kmers"""

    assert find_kmers('', 1) == [] ❶
    assert find_kmers('ACTG', 1) == ['A', 'C', 'T', 'G'] ❷
    assert find_kmers('ACTG', 2) == ['AC', 'CT', 'TG']
    assert find_kmers('ACTG', 3) == ['ACT', 'CTG']
    assert find_kmers('ACTG', 4) == ['ACTG']
    assert find_kmers('ACTG', 5) == [] ❸
```

❶ Pass the empty string as the sequence to ensure the function returns the empty list.

❷ Check all the values for *k* using a short sequence.

❸ There are no 5-mers for a string of length 4.

Try writing your version before you read ahead. Here is the function I wrote:

```
def find_kmers(seq: str, k: int) -> List[str]:
    """Find k-mers in string"""

    n = len(seq) - k + 1 ❶
    return [] if n < 1 else [seq[i:i + k] for i in range(n)] ❷
```

❶ Find the number *n* of *k*-length substrings in a string *seq*.

❷ If *n* is a negative number, return the empty list; otherwise, return the k-mers using a list comprehension.

Now I have a handy way to get the leading and trailing k-mers from a sequence:

```
>>> from grph import find_kmers
>>> kmers = find_kmers('AAATTTT', 3)
>>> kmers
['AAA', 'AAT', 'ATT', 'TTT', 'TTT']
>>> kmers[0] ❶
'AAA'
>>> kmers[-1] ❷
'TTT'
```

❶ The first element is the leading k-mer.

❷ The last element is the trailing k-mer.

The k-mers give me the overlap sequences I need for the keys of my dictionary. I want the values of the dictionaries to be a list of the sequence IDs that share these k-mers. The `collections.defaultdict()` I introduced in Chapter 1 is a good data structure to use for this. I need to import it and the `pprint.pformat()` function for logging purposes, so add the following:

```
from collections import defaultdict
from pprint import pformat
```

Here is how I can use these ideas:

```
def main() -> None:
    args = get_args()

    logging.basicConfig(
        filename='.log',
        filemode='w',
        level=logging.DEBUG if args.debug else logging.CRITICAL)

    start, end = defaultdict(list), defaultdict(list) ❶
    for rec in SeqIO.parse(args.file, 'fasta'): ❷
        if kmers := find_kmers(str(rec.seq), args.k): ❸
            start[kmers[0]].append(rec.id) ❹
            end[kmers[-1]].append(rec.id) ❺

    logging.debug(f'STARTS\n{pformat(start)}') ❻
    logging.debug(f'ENDS\n{pformat(end)}')
```

❶ Create dictionaries for the start and end regions that will have lists as the default values.

❷ Iterate the FASTA records.

❸ Coerce the `Seq` object to a string and find the k-mers. The `:=` syntax assigns the return to `kmers` and then the `if` evaluates if `kmers` is truthy. If the function returns no kmers, then the following block will not execute.

❹ Use the first k-mer as a key into the `start` dictionary and append this sequence ID to the list.

❺ Do likewise for the `end` dictionary using the last k-mer.

❻ Use the `pprint.pformat()` function to format the dictionaries for logging.

I've used the `pprint.pprint()` function in earlier chapters to print complex data structures in a prettier format than the default `print()` function. I can't use `pprint()` here because it would print to `STDOUT` (or `STDERR`). Instead, I need to format the data structure for the `logging.debug()` function to log.

Now run the program again with the first input and the debug flag, then inspect the log file:

```
$ ./grph.py tests/inputs/1.fa -d
$ cat .log
DEBUG:root:input file = "tests/inputs/1.fa"
DEBUG:root:STARTS ❶
defaultdict(<class 'list'>,
            {'AAA': ['Rosalind_0498', 'Rosalind_2391',
'Rosalind_0442'], ❷
             'GGG': ['Rosalind_5013'],
             'TTT': ['Rosalind_2323']})
DEBUG:root:ENDS ❸
defaultdict(<class 'list'>,
            {'AAA': ['Rosalind_0498'], ❹
             'CCC': ['Rosalind_2323', 'Rosalind_0442'],
             'GGG': ['Rosalind_5013'],
             'TTT': ['Rosalind_2391']})
```

❶ A dictionary of the various starting sequences and the IDs.

❷ Three sequences start with *AAA*: 0498, 2391, and 0442.

❸ A dictionary of the various ending sequences and the IDs.

❹ There is just one sequence ending with *AAA* which is 0498.

The correct pairs for this input file and the overlapping 3-mers are as follows:

- Rosalind_0498, Rosalind_2391: AAA
- Rosalind_0498, Rosalind_0442: AAA

- Rosalind_2391, Rosalind_2323: TTT

When you combine, for instance, the sequence ending with *AAA* (0498) with those starting with this sequence(0498, 2391, 0442), you wind up with the following pairs:

- Rosalind_0498, Rosalind_0498
- Rosalind_0498, Rosalind_2391
- Rosalind_0498, Rosalind_0442

Since I can't join a sequence to itself, the first pair is disqualified. Find the next *end* and *start* sequence in common, then iterate all the sequence pairs. I'll leave you to finish this exercise by finding all the start and end keys that are in common and then combining all the sequence IDs to print the pairs that can be joined. The pairs can be in any order and still pass the tests. I just want to wish you good luck. We're all counting on you.

Solutions

I have two variations to share with you. The first solves the Rosalind problem to show how to combine any two sequences. The second extends the graphs to create a full assembly of all the sequences.

Solution 1: Using Set Intersections to Find Overlaps

In the following solution, I use set intersections to find the k-mers shared between the start and end dictionaries:

```
def main() -> None:
    args = get_args()

    logging.basicConfig(
        filename='.log',
```

```
        filemode='w',
        level=logging.DEBUG if args.debug else logging.CRITICAL)

    start, end = defaultdict(list), defaultdict(list)
    for rec in SeqIO.parse(args.file, 'fasta'):
        if kmers := find_kmers(str(rec.seq), args.k):
            start[kmers[0]].append(rec.id)
            end[kmers[-1]].append(rec.id)

    logging.debug('STARTS\n{}'.format(pformat(start)))
    logging.debug('ENDS\n{}'.format(pformat(end)))

    for kmer in set(start).intersection(set(end)): ❶
        for pair in starfilter(op.ne, product(end[kmer],
start[kmer])): ❷
            print(*pair) ❸
```

❶ Find the keys in common between the `start` and `end` dictionaries.

❷ Iterate through the pairs of the ending and starting sequences that are not equal to each other.

❸ Print the pair of sequences.

The final 3 lines took me a few attempts to write. Given these dictionaries:

```
>>> from pprint import pprint
>>> from Bio import SeqIO
>>> from collections import defaultdict
>>> from grph import find_kmers
>>> k = 3
>>> start, end = defaultdict(list), defaultdict(list)
>>> for rec in SeqIO.parse('tests/inputs/1.fa', 'fasta'):
...     if kmers := find_kmers(str(rec.seq), k):
...         start[kmers[0]].append(rec.id)
...         end[kmers[-1]].append(rec.id)
...
>>> pprint(start)
{'AAA': ['Rosalind_0498', 'Rosalind_2391', 'Rosalind_0442'],
 'GGG': ['Rosalind_5013'],
```

```
 'TTT': ['Rosalind_2323']}
>>> pprint(end)
{'AAA': ['Rosalind_0498'],
 'CCC': ['Rosalind_2323', 'Rosalind_0442'],
 'GGG': ['Rosalind_5013'],
```

I started with this idea:

```
>>> for kmer in end: ❶
...     if kmer in start: ❷
...         for seq_id in end[kmer]: ❸
...             for other in start[kmer]: ❹
...                 if seq_id != other: ❺
...                     print(seq_id, other) ❻
...
Rosalind_0498 Rosalind_2391
Rosalind_0498 Rosalind_0442
Rosalind_2391 Rosalind_2323
```

❶ Start with each k-mer from the `end` dictionary. A `for` loop over a dictionary will iterate over the *keys* of the dictionary.

❷ See if this k-mer is in the `start` dictionary.

❸ Iterate through each ending sequence ID for this k-mer.

❹ Iterate through each starting sequence ID for this k-mer.

❺ Make sure the sequences are not the same.

❻ Print the sequence IDs.

While that works just fine, I let this sit for a while and came back to it, asking myself exactly what was I trying to do? The first two lines are trying to find the keys that are in common between the two dictionaries. Set intersection is an easier way to achieve this. If I use the `set()` function on a dictionary, it creates a set using the keys of the dictionary:

```
>>> set(start)
{'TTT', 'GGG', 'AAA'}
>>> set(end)
{'TTT', 'CCC', 'AAA', 'GGG'}
```

I can then call the `set.intersection()` function to find the keys in common:

```
>>> set(start).intersection(set(end))
{'TTT', 'GGG', 'AAA'}
```

In the preceding code, the next lines find all the combinations of the ending and starting sequence IDs. This is more easily done using the `itertools.product()` function Here I'll use the k-mer *AAA*:

```
>>> from itertools import product
>>> kmer = 'AAA'
>>> pairs = list(product(end[kmer], start[kmer]))
>>> pprint(pairs)
[('Rosalind_0498', 'Rosalind_0498'),
 ('Rosalind_0498', 'Rosalind_2391'),
 ('Rosalind_0498', 'Rosalind_0442')]
```

I want to exclude any pairs where the two values are the same. I could write a `filter()` for this:

```
>>> list(filter(lambda p: p[0] != p[1], pairs)) ❶
[('Rosalind_0498', 'Rosalind_2391'), ('Rosalind_0498',
'Rosalind_0442')]
```

❶ The `lambda` receives the pair `p` and checks that the 0th and 1st elements are not equal.

This works adequately, but I'm not satisfied with the code. I really hate that I can't unpack the tuple values in the `lambda` to `filter()`. Immediately I started thinking about how the `itertools.starmap()` function I used in Chapters 6 and 8 can do this, so I searched the

internet for *Python starfilter* and found the function `iteration_utilities.starfilter()`. I installed this module and imported the function:

```
>>> from iteration_utilities import starfilter
>>> list(starfilter(lambda a, b: a != b, pairs))
[('Rosalind_0498', 'Rosalind_2391'), ('Rosalind_0498',
'Rosalind_0442')]
```

This is an improvement, but I can make it cleaner by using the `operator.ne()` (not equal) function which will obviate the `lambda`:

```
>>> import operator as op
>>> list(starfilter(op.ne, pairs))
[('Rosalind_0498', 'Rosalind_2391'), ('Rosalind_0498',
'Rosalind_0442')]
```

Finally, I splat each of the pairs to make `print()` see the individual strings rather than the list container:

```
>>> for pair in starfilter(op.ne, pairs):
...     print(*pair)
...
Rosalind_0498 Rosalind_2391
Rosalind_0498 Rosalind_0442
```

I could have shortened this even more, but I fear this gets a little too dense:

```
>>> print('\n'.join(map(' '.join, starfilter(op.ne, pairs))))
Rosalind_0498 Rosalind_2391
Rosalind_0498 Rosalind_0442
```

Solution 2: Using a Graph to Find All Paths

This next solution approximates a full assembly of the sequences using a graph to link all the overlapping sequences. While not part of the original challenge, it is, nonetheless, interesting to contemplate

while also proving surprisingly simple to implement and even visualize. Since *GRPH* is the challenge name, it makes sense to investigate how to represent a graph in Python code.

I can manually align all the sequences as shown in Figure 9-3. This reveals a graph structure where sequence Rosalind_0498 can join to either Rosalind_2391 or Rosalind_0442 and there follows a chain from Rosalind_0498 to Rosalind_2391 to Rosalind_2323.

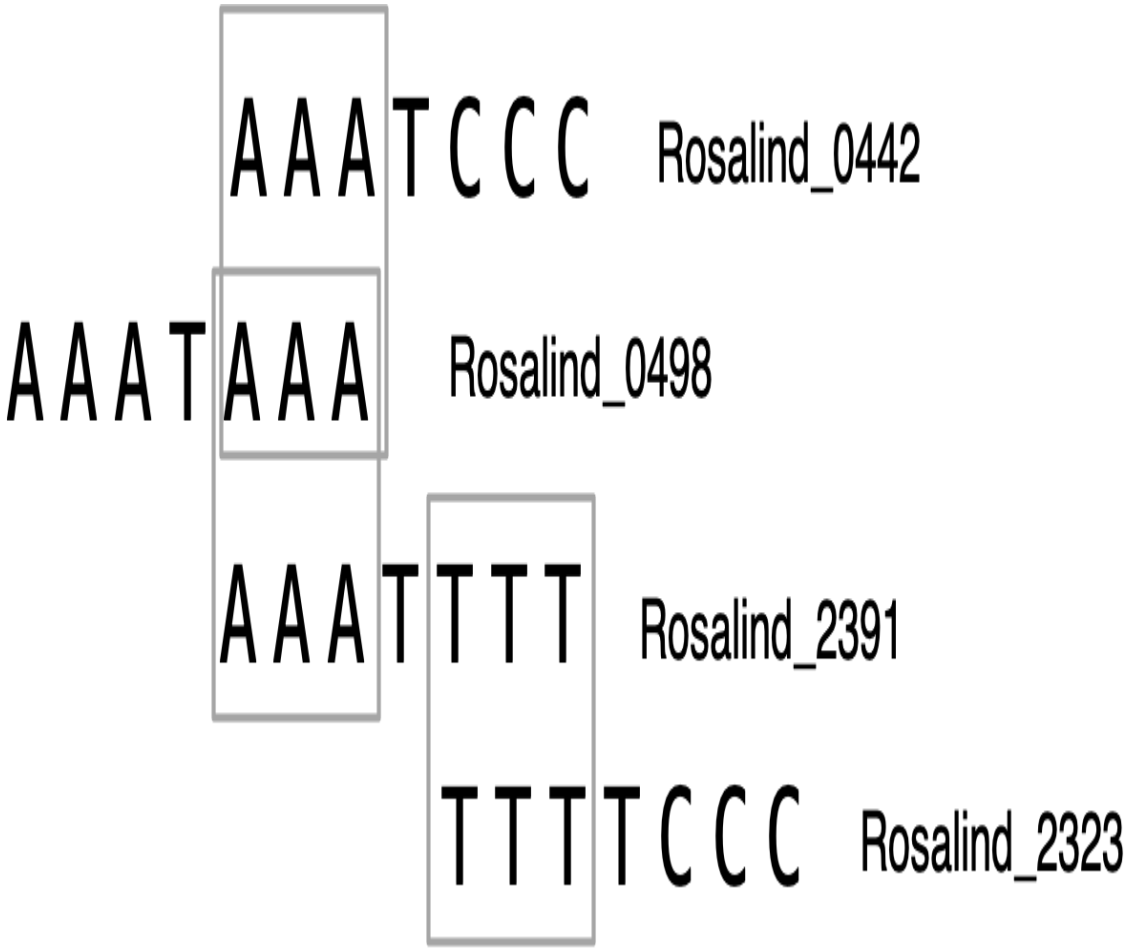


Figure 9-3. All the sequences in the first input file can be joined using 3-mers.

To encode this, I use the Graphviz tool both to represent and visualize a graph structure. Note that you will need to install the `graphviz` libraries for this to work. The output from this program will be a text file in the Dot language format which can be turned into a pictorial graph by the `dot` program. The second solution in the

repository has options to control the output filename and whether the image should be opened:

```
$ ./solution2_graph.py -h
usage: solution2_graph.py [-h] [-k size] [-o FILE] [-v] [-d] FILE

Overlap Graphs

positional arguments:
  FILE                  FASTA file

optional arguments:
  -h, --help            show this help message and exit
  -k size, --overlap size
                        Size of overlap (default: 3)
  -o FILE, --outfile FILE
                        Output filename (default: graph.txt) ❶
  -v, --view            View outfile (default: False) ❷
  -d, --debug           Debug (default: False)
```

❶ The default output filename is *graph.txt*. A *.pdf* file will also be generated automatically which is the visual rendition of the graph.

❷ This option controls whether the PDF should be opened automatically when the program finishes.

If you run this program on the first test input, you will see the same output as before so that it will pass the test suite:

```
$ ./solution2_graph.py tests/inputs/1.fa -o 1.txt
Rosalind_2391 Rosalind_2323
Rosalind_0498 Rosalind_2391
Rosalind_0498 Rosalind_0442
```

There should now also be a new output file called `1.txt` containing a graph structure encoded in the Dot language:

```
$ cat 1.txt
digraph {
```

```
        Rosalind_0498
        Rosalind_2391
        Rosalind_0498 -> Rosalind_2391
        Rosalind_0498
        Rosalind_0442
        Rosalind_0498 -> Rosalind_0442
        Rosalind_2391
        Rosalind_2323
        Rosalind_2391 -> Rosalind_2323
}
```

You can use the `dot` program to turn this into a visualization. Here is a command to save the graph to a PNG file:

```
$ dot -O -Tpng 1.txt
```

Figure 9-4 shows the resulting visualization of the graph joining all the sequences in the first FASTA file, recapitulating the manual alignment from Figure 9-3. If you run the program with the `-v|--view` flag, this image should be shown automatically. In graph terminology, each sequence is a *node*, and the relationship between two sequences is an *edge*.

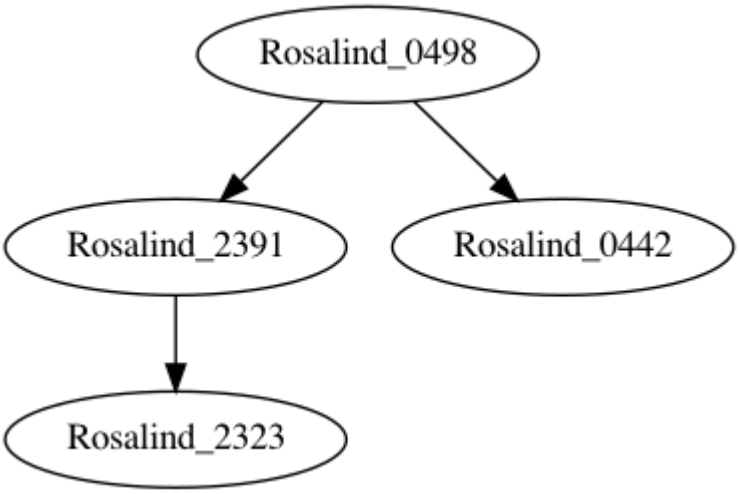


Figure 9-4. The output from the `dot` program showing an assembly of the sequences in the first input file when joined on 3-mers.

Graphs may or may not have directionality. Figure 9-4 includes arrows implying that there is a relationship that flows from one node

to another; therefore, this is a *directed graph*. The following shows how I create and visualize this graph. Note that I import `graphiz.Digraph` to create the directed graph and that this code omits the logging code that is part of the actual solution:

```
def main() -> None:
    args = get_args()
    start, end = defaultdict(list), defaultdict(list)
    for rec in SeqIO.parse(args.file, 'fasta'):
        if kmers := find_kmers(str(rec.seq), args.k):
            start[kmers[0]].append(rec.id)
            end[kmers[-1]].append(rec.id)

    dot = Digraph() ❶
    for kmer in set(start).intersection(set(end)): ❷
        for s1, s2 in starfilter(op.ne, product(end[kmer],
start[kmer])): ❸
            print(s1, s2) ❹
            dot.node(s1) ❺
            dot.node(s2)
            dot.edge(s1, s2) ❻

    args.outfile.close() ❼
    dot.render(args.outfile.name, view=args.view) ❽
```

❶ Create a directed graph.

❷ Iterate through the shared k-mers.

❸ Find sequence pairs sharing a k-mer, unpack the two sequence IDs into `s1` and `s2`.

❹ Print the output for the test.

❺ Add nodes for each sequence.

❻ Add an edge connecting the nodes.

❼

Close the output filehandle so that the graph can write to the filename.

8. Write the graph structure to the output filename. Use the `view` option to open the image depending on the `args.view` option.

These few lines of code have an outsized effect on the output of the program. For instance, Figure 9-5 shows that this program can essentially create a full assembly of the 100 sequences in the second input file.

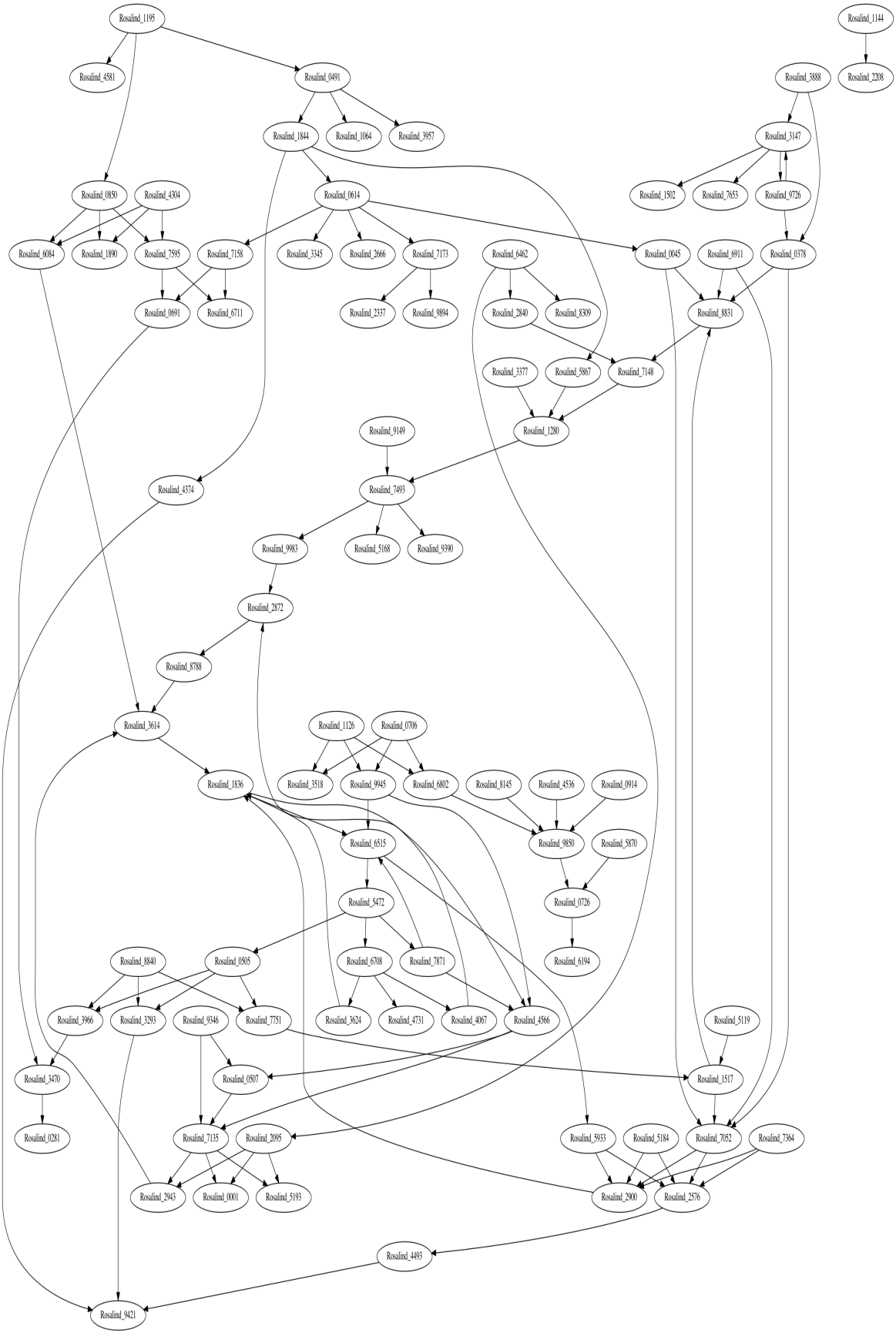

Rosalind_1195
Rosalind_1144
Rosalind_4581
Rosalind_0491
Rosalind_3888
Rosalind_2208
Rosalind_1844
Rosalind_1064
Rosalind_3957
Rosalind_3147
Rosalind_0850
Rosalind_4304
Rosalind_0614
Rosalind_1502
Rosalind_7653
Rosalind_9726
Rosalind_6084
Rosalind_1890
Rosalind_7595
Rosalind_7158
Rosalind_3345
Rosalind_2666
Rosalind_7173
Rosalind_6462
Rosalind_0045
Rosalind_6911
Rosalind_0378
Rosalind_0691
Rosalind_6711
Rosalind_2337
Rosalind_9894
Rosalind_2840
Rosalind_8309
Rosalind_8831
Rosalind_3377
Rosalind_5867
Rosalind_7148
Rosalind_1280
Rosalind_9149
Rosalind_4374
Rosalind_7493
Rosalind_9983
Rosalind_5168
Rosalind_9390
Rosalind_2872
Rosalind_8788
Rosalind_3614
Rosalind_1126
Rosalind_0706
Rosalind_1836
Rosalind_3518
Rosalind_9945
Rosalind_6802
Rosalind_8145
Rosalind_4536
Rosalind_0914
Rosalind_6515
Rosalind_9850
Rosalind_5870
Rosalind_5472
Rosalind_0726
Rosalind_8840
Rosalind_0505
Rosalind_6708
Rosalind_7871
Rosalind_6194
Rosalind_3966
Rosalind_3293
Rosalind_9346
Rosalind_7751
Rosalind_3624
Rosalind_4731
Rosalind_4067
Rosalind_4566
Rosalind_5119
Rosalind_3470
Rosalind_0507
Rosalind_1517
Rosalind_0281
Rosalind_7135
Rosalind_2095
Rosalind_5933
Rosalind_5184
Rosalind_7052
Rosalind_7364
Rosalind_2943
Rosalind_0001
Rosalind_5193
Rosalind_2900
Rosalind_2576
Rosalind_4493
Rosalind_9421

Figure 9-5. The graph of the second test input file.

This image speaks volumes for the complexity and completeness of the data; for instance, the sequence pair in the upper-right corner — Rosalind_1144 and Rosalind_2208 — cannot be joined to any other sequences. I would encourage you to try increasing *k* to 4 and inspecting the resulting graph to see a profoundly different result.

Graphs are truly powerful data structures. As noted in the introduction, graphs encode pair-wise relationships. It's amazing to see the assembly of the 100 sequences in Figure 9-5 emerge with so few lines of code that are noting the sequence pairs. While it's possible to abuse Python lists and dictionaries to represent graphs, the Graphviz tool makes this much simpler.

I used a directed graph for this exercise, but that wasn't necessarily required. This could have been an undirected graph, too, but I like the arrows. I would note that you might encounter the term *directed acyclic graph* (DAG) to indicate a directed graph that has no cycles which is when a node joins back to itself. An actual sequence assembly would disallow cycles, but real-world data might present sequences that create unresolvable paths. If you find these ideas interesting, you should investigate De Bruijn graphs which are often built from overlapping k-mers.

Going Further

Add a Hamming distance option that will allow the overlapping sequence to have the indicated edit distance. That is, a distance of 1 will allow for the overlap of sequences with a single base difference.

Review

- K-mers quickly find the first and last *k* bases of each sequence to identify overlapping regions.

- The `logging` module makes it easy to turn on and off the logging of runtime messages to a file.
- I used the `defaultdict(list)` to create a dictionary that would auto-vivify any key not present with a default value of the empty list.
- Set intersection can find common elements between collections such as the keys shared between two dictionaries.
- The `itertools.product()` function found all the possible pairs of sequences.
- The `iteration_utilities.starfilter()` function will splat the argument to the `lambda` for `filter()` just as the `itertools.starmap()` function does for `map()`.
- The Graphviz tool can efficiently represent and visualize complex graph structures.
- Graphs can be textually represented using the Dot language, and the `dot` program can generate visualizations of graphs in various formats.
- Overlap graphs can be used to create a complete assembly of two or more sequences.

---

1 Imagine debugging a program without even a console. In the 1950s, Claude Shannon was visiting Alan Turing at the latter's lab in England. During their conversation, a horn began sounding at regular intervals. Turing said this indicated his code was stuck in a loop. Without a monitor, this was how he monitored the progress of his programs.

Chapter 10. Finding the Longest Shared Subsequence: Finding K-mers, Writing and Functions, and Using Binary Search

---

As described in the Rosalind LCSM challenge, the goal of this exercise is to find the longest shared substring in a set of sequences. As in Chapter 8, I'm searching for a motif shared by all the sequences in a given file. In that previous challenge, however, I knew the identity of the motif. This time, I don't know *a priori* that any shared motif is present — much less the size or composition of it — so I'll just look for any length of sequence that is present in every sequence. This is a challenging exercise that brings together many ideas I've shown in earlier chapters. I'll use the solutions to explore algorithm design, functions, tests, and code organization.

You will learn:

- How to count k-mers to find shared subsequences
- How to use `itertools.chain()` to concatenate lists of lists
- How to use a binary search
- About *Big O* run times
- How to maximize a function
- How to use the `key` option with `min()` and `max()`

Getting Started

All the code and tests for this challenge are in the *10_lcsm* directory. Start by copying the first solution to the `lcsm.py` program and asking for help:

```
$ cp solution1_kmers_imperative.py lcsm.py
$ ./lcsm.py -h
usage: lcsm.py [-h] FILE

Longest Common Substring

positional arguments:
  FILE        Input FASTA

optional arguments:
  -h, --help  show this help message and exit
```

The only required argument is a single positional file of FASTA-formatted DNA sequences. As with other programs that accept files, the program will reject invalid or unreadable input. The following is the first input I'll use. The longest common subsequences in these sequences are *CA*, *TA*, and *AC* with the last shown in bold:

```
$ cat tests/inputs/1.fa
>Rosalind_1
GATTACA
>Rosalind_2
TAGACCA
>Rosalind_3
ATACA
```

Any of these answers is acceptable. Run the program with the first test input and note that it finds one of the acceptable 2-mers:

```
$ ./lcsm.py tests/inputs/1.fa
CA
```

The second test input is much larger, and you'll notice that the program takes significantly longer to find the answer. On my laptop, it takes almost 40 seconds. In the solutions, I'll show you a way to significantly decrease the run time using a binary search:

```
$ time ./lcsm.py tests/inputs/2.fa
GCCTTTTGATTTTAACGTTTATCGGGTGTAGTAAGATTGCGCGCTAATTCCAATAAACGTATGGAGGACA
TTCCCCGT

real    0m39.244s
user    0m33.708s
sys     0m6.202s
```

Although not a requirement of the challenge, I've included one input file that contains no shared subsequences for which the program should create a suitable response:

```
$ ./lcsm.py tests/inputs/none.fa
No common subsequence.
```

Start the `lcsm.py` program from scratch:

```
$ new.py -fp 'Longest Common Substring' lcsm.py
Done, see new script "lcsm.py".
```

Define the arguments like so:

```
class Args(NamedTuple): ❶
    file: TextIO


# --------------------------------------------------
def get_args() -> Args:
    """ Get command-line arguments """

    parser = argparse.ArgumentParser(
        description='Longest Common Substring',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file', ❷
```

```
                        help='Input FASTA',
                        metavar='FILE',
                        type=argparse.FileType('rt'))

    args = parser.parse_args()

    return Args(args.file) ❸
```

❶ The only input to this program is a FASTA-formatted file.

❷ Define a single *file* argument.

❸ Return the `Args` containing the open filehandle.

Then update the `main()` to print the incoming filename:

```
def main() -> None:
    args = get_args()
    print(args.file.name)
```

Verify that you see the correct usage and that the program correctly prints the filename:

```
$ ./lcsm.py tests/inputs/1.fa
tests/inputs/1.fa
```

At this point, your program should pass the first three tests. If you think you know how to complete the program, have at it. If you want a prod in the right direction, read on.

Finding the Shortest Sequence in a FASTA File

Reading a FASTA file should be familiar by now. I'll use `Bio.SeqIO.parse()` as before. My first idea on this problem was to find shared k-mers while maximizing for *k*. I decided to start with *k* equal to the length of the shortest sequence since the longest subsequence can be no longer than that. Finding the shortest

sequence requires that I first scan through *all* the records. To review how to do this, the `Bio.SeqIO.parse()` function returns an iterator that gives me access to each FASTA record:

```
>>> from Bio import SeqIO
>>> fh = open('./tests/inputs/1.fa')
>>> recs = SeqIO.parse(fh, 'fasta')
>>> type(recs)
<class 'Bio.SeqIO.FastaIO.FastaIterator'>
```

I can use the `next()` function I first showed in Chapter 4 to force the iterator to produce the next value, the type of which is a `SeqRecord`:

```
>>> rec = next(recs)
>>> type(rec)
<class 'Bio.SeqRecord.SeqRecord'>
```

In addition to the read itself, the FASTA record contains metadata such as the sequence ID, name, and such:

```
>>> rec
SeqRecord(seq=Seq('GATTACA'),
          id='Rosalind_1',
          name='Rosalind_1',
          description='Rosalind_1',
          dbxrefs=[])
```

The read information is wrapped in a `Seq` object which has many interesting and useful methods you can explore in the REPL using `help(rec.seq)`. I'm only interested in the raw sequence, so I can use the `str()` function to coerce it to a string:

```
>>> str(rec.seq)
'GATTACA'
```

I need all the sequences in a list so I can find the length of the shortest one. I can use a list comprehension to read the entire file into a list since I'll be using these many times:

```
>>> fh = open('./tests/inputs/1.fa') ❶
>>> seqs = [str(rec.seq) for rec in SeqIO.parse(fh, 'fasta')] ❷
>>> seqs
['GATTACA', 'TAGACCA', 'ATACA']
```

❶ Reopen the filehandle or the existing filehandle would continue from the second read.

❷ Create a list, coercing each record's sequence to a string.

The same idea can be expressed using the `map()` function:

```
>>> fh = open('./tests/inputs/1.fa')
>>> seqs = list(map(lambda rec: str(rec.seq), SeqIO.parse(fh,
'fasta')))
>>> seqs
['GATTACA', 'TAGACCA', 'ATACA']
```

To find the length of the shortest sequence, I need to find the lengths of all the sequences which I can do using a list comprehension:

```
>>> [len(seq) for seq in seqs]
[7, 7, 5]
```

I prefer the shorter way to write this using a `map()`:

```
>>> list(map(len, seqs))
[7, 7, 5]
```

Python has built-in `min()` and `max()` function that will return the minimum or maximum value from a list:

```
>>> min(map(len, seqs))
5
>>> max(map(len, seqs))
7
```

So the shortest sequence is equal to the minimum of the lengths:

```
>>> shortest = min(map(len, seqs))
>>> shortest
5
```

Extracting k-mers From a Sequence

The longest shared subsequence can be no longer than the shortest sequence and must be shared by all the reads. Therefore, my next step is to find all the k-mers in all the sequences, starting with *k* equal to the length of the shortest sequence (5). In Chapter 9, I wrote a `find_kmers()` function and test, so I'll copy that code into this program. Remember to import `typing.List` for this:

```
def find_kmers(seq: str, k: int) -> List[str]:
    """ Find k-mers in string """

    n = len(seq) - k + 1
    return [] if n < 1 else [seq[i:i + k] for i in range(n)]


def test_find_kmers() -> None:
    """ Test find_kmers """

    assert find_kmers('', 1) == []
    assert find_kmers('ACTG', 1) == ['A', 'C', 'T', 'G']
    assert find_kmers('ACTG', 2) == ['AC', 'CT', 'TG']
    assert find_kmers('ACTG', 3) == ['ACT', 'CTG']
    assert find_kmers('ACTG', 4) == ['ACTG']
    assert find_kmers('ACTG', 5) == []
```

One logical approach is to start with the maximum possible value of *k* and count down. So far I've only used the `range()` function to count up. Can I reverse the start and stop values to have it count down? Apparently not. If the start value is greater than the stop value, then `range()` will produce an empty list:

```
>>> list(range(shortest, 0))
[]
```

When reading codons in Chapter 7, I mentioned that the `range()` function accepts up to three arguments, the last of which is the *step*, which I used there to jump three bases at a time. Here I need to use a step of -1 to count down. Remember that the stop value is not included:

```
>>> list(range(shortest, 0, -1))
[5, 4, 3, 2, 1]
```

Another way to count backwards is to count up and reverse the results:

```
>>> list(reversed(range(1, shortest + 1)))
[5, 4, 3, 2, 1]
```

Either way, I want to iterate over decreasing values of *k* until I find a k-mer that is shared by all the sequences. A sequence might contain multiple copies of the same k-mer, so it's important to make the result unique by using the `set()` function:

```
>>> from lcsm import find_kmers
>>> from pprint import pprint
>>> for k in range(shortest, 0, -1):
...     print(f'==> {k} <==')
...     pprint([set(find_kmers(s, k)) for s in seqs])
...
==> 5 <==
[{'TTACA', 'GATTA', 'ATTAC'}, {'TAGAC', 'AGACC', 'GACCA'}, {'ATACA'}]
==> 4 <==
[{'ATTA', 'TTAC', 'TACA', 'GATT'},
 {'GACC', 'AGAC', 'TAGA', 'ACCA'},
 {'TACA', 'ATAC'}]
==> 3 <==
[{'ACA', 'TAC', 'GAT', 'ATT', 'TTA'},
 {'AGA', 'TAG', 'CCA', 'ACC', 'GAC'},
 {'ACA', 'ATA', 'TAC'}]
==> 2 <==
[{'AC', 'AT', 'CA', 'TA', 'TT', 'GA'},
 {'AC', 'CA', 'CC', 'TA', 'AG', 'GA'},
 {'AC', 'AT', 'CA', 'TA'}]
```

```
==> 1 <==
[{'G', 'C', 'T', 'A'}, {'G', 'C', 'T', 'A'}, {'C', 'T', 'A'}]
```

Can you see a way to use this idea to count all the k-mers for each value of *k*? Look for k-mers that have a frequency matching the number of sequences. If you find more than one, print any of them.

Solutions

The two variations for this program use the same basic logic to the longest shared subsequence. The first version proves to scale poorly as the input size increases because it uses a step-wise, linear approach to iterating over every possible *k* length of sequence. The second version introduces a binary search to find a good starting value for *k* and then initiates a hill-climbing search to discover a maximum value for *k*.

Solution 1: Counting Frequencies of k-mers

In the introduction, I got as far as finding all the k-mers in the sequences for values of *k* starting with the shortest sequence and moving down to 1. Here I'll start with *k* of 5, which was the shortest sequence in the first FASTA file:

```
>>> fh = open('./tests/inputs/1.fa')
>>> seqs = [str(rec.seq) for rec in SeqIO.parse(fh, 'fasta')]
>>> shortest = min(map(len, seqs))
>>> kmers = [set(find_kmers(seq, shortest)) for seq in seqs]
>>> kmers
[{'TTACA', 'GATTA', 'ATTAC'}, {'TAGAC', 'AGACC', 'GACCA'}, {'ATACA'}]
```

I need a way to count how many times each k-mer appears across all the sequences. One approach is to use `collections.Counter()` that I first showed in Chapter 1:

```
>>> from collections import Counter
>>> counts = Counter()
```

I can iterate over each set of k-mers from the sequences and use the `Counter.update()` method to add them:

```
>>> for group in kmers:
...     counts.update(group)
...
>>> pprint(counts)
Counter({'TTACA': 1,
         'GATTA': 1,
         'ATTAC': 1,
         'TAGAC': 1,
         'AGACC': 1,
         'GACCA': 1,
         'ATACA': 1})
```

Or I could concatenate the many lists of k-mers together into a single list using `itertools.chain()`:

```
>>> from itertools import chain
>>> list(chain.from_iterable(kmers))
['TTACA', 'GATTA', 'ATTAC', 'TAGAC', 'AGACC', 'GACCA', 'ATACA']
```

Using this as the input for the `Counter()` produces the same collection showing that each 5-mer is unique:

```
>>> counts = Counter(chain.from_iterable(kmers))
>>> pprint(counts)
Counter({'TTACA': 1,
         'GATTA': 1,
         'ATTAC': 1,
         'TAGAC': 1,
         'AGACC': 1,
         'GACCA': 1,
         'ATACA': 1})
```

The `Counter()` is a regular dictionary underneath, which means I have access to all the dictionary methods. I want to iterate through

the keys and values as pairs using the `dict.items()` method to find where the count of the kmer is equal to the number of sequences:

```
>>> n = len(seqs)
>>> candidates = []
>>> for kmer, count in counts.items():
...     if count == n:
...         candidates.append(kmer)
...
>>> candidates
[]
```

When *k* is 5, there are no candidate sequences, so I need to try with a smaller value. Since I know the right answer is 2, I'll re-run this code with *k* of 2 to produce this dictionary:

```
>>> k = 2
>>> kmers = [set(find_kmers(seq, k)) for seq in seqs]
>>> counts = Counter(chain.from_iterable(kmers))
>>> pprint(counts)
Counter({'CA': 3,
         'AC': 3,
         'TA': 3,
         'GA': 2,
         'AT': 2,
         'TT': 1,
         'AG': 1,
         'CC': 1})
```

From this, I find three candidate 2-mers have a frequency of 3 which equals the number of sequences:

```
>>> candidates = []
>>> for kmer, count in counts.items():
...     if count == n:
...         candidates.append(kmer)
...
>>> candidates
['CA', 'AC', 'TA']
```

It doesn't matter which of the candidates I choose, so I'll use the `random.choice()` function which returns one value from a list of choices:

```
>>> import random
>>> random.choice(candidates)
'AC'
```

I like where this is going, so I'd like to put it into a function so I can test it. Here is the stub for the function:

```
def common_kmers(seqs: List[str], k: int) -> List[str]:
    """ Find k-mers common to all sequences """

    kmers = [set(find_kmers(seq, k)) for seq in seqs]
    counts = Counter(chain.from_iterable(kmers))
    n = len(seqs) ❶
    return [seq for seq, freq in counts.items() if freq == n] ❷
```

❶ Find the number of sequences.

❷ Return the sequences having a frequency equal to the number of sequences.

This makes for a pretty readable `main()`:

```
import random
import sys


def main() -> None:
    args = get_args()
    seqs = [str(rec.seq) for rec in SeqIO.parse(args.file, 'fasta')] ❶
    shortest = min(map(len, seqs)) ❷

    for k in range(shortest, 0, -1): ❸
        if kmers := common_kmers(seqs, k): ❹
            print(random.choice(kmers)) ❺
            sys.exit(0) ❻
```

```
    print('No common subsequence.') ❼
```

❶ Read all the sequences into a list.

❷ Find the length of the shortest sequence.

❸ Count down from the shortest sequence.

❹ Find all the common k-mers using this value of *k*.

❺ If any k-mers are found, print a random selection.

❻ Exit the program using an exit value of 0 (no errors).

❼ If I make it to this point, inform the user there is no shared sequence.

> **NOTE**
>
> The program starts by reading all the sequences into memory. It's not uncommon to have FASTA inputs with perhaps 10s of million reads. I have several Makefile targets that will use the `genseq.py` program in the *10_lcsm* directory to generate large FASTA inputs with a common motif for you to test. This program reads all the sequences into memory and works adequately for this solution, but it's generally best practice to treat large files as *streams* of data to read rather than trying to fit them into memory.

Solution 2: Speeding Things Up with a Binary Search

As noted in the introduction, this solution grows much slower as the size of the inputs increases. One way to track the progress of the program is to put a `print(k)` statement at the beginning of the `for` loop. Run this with the second input file, and you'll see that it starts

counting down from 1000 and doesn't reach the correct value for *k* until 78.

Counting backward by 1 is taking too long. If your friend asked you to guess a number between 1 and 1,000, you wouldn't start at 1,000 and keep guessing 1 less while your friend said, "Too high." It's much faster (and better for your friendship) to guess 500. If your friend chose 453, they'd say, "Too high," so you'd be wise to choose 250. They reply, "Too low," so you keep splitting the differences between your last high and low guesses until you find the right answer. This is a *binary search*, and it's a great way to quickly find the location of a wanted value from a sorted list of values.

To understand this better, I've included a program in the *10_lcsm* directory called `binsearch.py`:

```
$ ./binsearch.py -h
usage: binsearch.py [-h] -n int -m int

Binary Search

optional arguments:
  -h, --help         show this help message and exit
  -n int, --num int  The number to guess (default: None)
  -m int, --max int  The maximum range (default: None)
```

The following is the relevant portion of the program. You can read the source code for the argument definitions if you like. The `binary_search()` function is recursive like one version of the Fibonacci sequence from Chapter 4. Note that the search values must be sorted for binary searches to work, which the `range()` function provides:

```
def main() -> None:
    args = get_args()
    nums = list(range(args.maximum + 1))
    pos = binary_search(args.num, nums, 0, args.maximum)
    print(f'Found {args.num}!' if pos > 0 else f'{args.num} not
present.')
```

```
def binary_search(x: int, xs: List[int], low: int, high: int) -> int:
    print(f'{low:4} {high:4}', file=sys.stderr)

    if high >= low: ❶
        mid = (high + low) // 2 ❷

        if xs[mid] == x: ❸
            return mid

        if xs[mid] > x: ❹
            return binary_search(x, xs, low, mid - 1) ❺

        return binary_search(x, xs, mid + 1, high) ❻

    return -1 ❼
```

❶ The base case to decide when to exit the recursion.

❷ The midpoint is halfway between the high and low using floor division.

❸ Return the midpoint if the element is in the middle.

❹ See if the value at the midpoint is greater than the desired.

❺ Search the lower values.

❻ Search the higher values.

❼ The value was not found.

I included some print messages so that, running with the previous numbers, you can see how the low and high finally converge on the target number in 10 steps:

```
$ ./binsearch.py -n 453 -m 1000
   0 1000
```

```
  0  499
250  499
375  499
438  499
438  467
453  467
453  459
453  455
453  453
Found 453!
```

It takes just eight iterations to determine the number is not present:

```
$ ./binsearch.py -n 453 -m 100
  0  100
 51  100
 76  100
 89  100
 95  100
 98  100
100  100
101  100
453 not present.
```

The binary search can tell me if a value occurs in a list of values, but this is not quite my problem. While I'm reasonably sure there will be at least a 2- or 1-mer in common in most datasets, I have included one file that has none:

```
$ cat tests/inputs/none.fa
>Rosalind_1
GGGGGGG
>Rosalind_2
AAAAAAAA
>Rosalind_3
CCCC
>Rosalind_4
TTTTTTTT
```

If there is an acceptable value for *k*, then I need to find the *maximum* value. I decided to use the binary search to find a starting point for a

hill-climbing search to find the maximum value. First I'll show the `main()`, and then I'll break down the other functions:

```
def main() -> None:
    args = get_args()
    seqs = [str(rec.seq) for rec in SeqIO.parse(args.file, 'fasta')] ❶
    shortest = min(map(len, seqs)) ❷
    common = partial(common_kmers, seqs) ❸
    start = binary_search(common, 1, shortest) ❹

    if start >= 0: ❺
        candidates = [] ❻
        for k in range(start, shortest + 1): ❼
            if kmers := common(k): ❽
                candidates.append(random.choice(kmers)) ❾
            else:
                break ❿

        print(max(candidates, key=len)) ⓫
    else:
        print('No common subsequence.') ⓬
```

❶ Get a list of the sequences as strings.

❷ Find the length of the shortest sequence.

❸ Partially apply the `common_kmers()` function with the `seqs` input.

❹ Use the binary search to find a starting point for the given function using 1 for the lowest value of *k* and the shortest sequence length for the maximum.

❺ Check that the binary search found something useful.

❻ Initialize a list of the candidate values.

❼ Start climbing hill using the binary search result.

❽ See if there are common k-mers using this value of *k*.

❾ If so, randomly choose one and add to the list of candidates.

❿ If there are no common k-mers, break out of the loop.

⓫ Choose the candidate sequence having the longest length.

⓬ Let the user know that there is no answer.

While there are many things to explain in the preceding code, I want to highlight the call to `max()`. I showed earlier that this function will return the maximum value from a list. Normally you might think to use this on a list of numbers:

```
>>> max([4, 2, 8, 1])
8
```

In the preceding code, I want to find the longest string in a list. I can `map()` the `len()` function to find their lengths:

```
>>> seqs = ['A', 'CC', 'GGGG', 'TTT']
>>> list(map(len, seqs))
[1, 2, 4, 3]
```

This shows that the third sequence *GGGG* is the longest. The `max()` function accepts an optional `key` argument which is a function to apply to each element before comparing. If I use the `len()` function, then `max()` correctly identifies the longest sequence:

```
>>> max(seqs, key=len)
'GGGG'
```

Let's take a look at how I modified the `binary_search()` function to suit my needs:

```
def binary_search(f: Callable, low: int, high: int) -> int: ❶
    """ Binary search """
```

```
    hi, lo = f(high), f(low) ❷
    mid = (high + low) // 2 ❸

    if hi and lo: ❹
        return high

    if not hi and lo: ❺
        return binary_search(f, low, mid)

    if hi and not lo: ❻
        return binary_search(f, mid, high)

    return -1 ❼
```

❶ The function takes another function `f` along with low and high values as arguments.

❷ Call the function with the highest and lowest values for *k*.

❸ Find the midpoint value of *k*.

❹ If there are values for both the high and low values, return the highest *k*.

❺ If the high *k* found nothing but the low value did, recursively call the function searching in the lower values of *k*.

❻ If the low *k* found nothing but the high value did, recursively call the function searching in the higher values of *k*.

❼ Return -1 to indicate the value was not found.

Here is the test I wrote for this:

```
def test_binary_search() -> None:
    """ Test binary_search """

    seqs1 = ['GATTACA', 'TAGACCA', 'ATACA'] ❶
```

```
    f1 = partial(common_kmers, seqs1) ❷
    assert binary_search(f1, 1, 5) == 2 ❸

    seqs2 = ['GATTACTA', 'TAGACTCA', 'ATACTA'] ❹
    f2 = partial(common_kmers, seqs2)
    assert binary_search(f2, 1, 6) == 3 ❺
```

❶ These are the sequences I've been using that have three shared 2-mers.

❷ Define a function to find the k-mers in the first set of sequences.

❸ The search finds *k* of 2 which is the right answer.

❹ The same sequences as before but now with a shared 3-mer.

❺ The search finds *k* of 3.

Unlike the previous binary search, my version won't (necessarily) return the *exact* answer, just a decent starting point. If there are no shared sequences for any size *k*, then I let the user know:

```
$ ./solution2_binary_search.py tests/inputs/none.fa
No common subsequence.
```

If there is a shared subsequence, this version runs significantly faster — perhaps as much as 28X faster.

```
$ hyperfine -L prg
./solution1_kmers_functional.py,./solution2_binary_search.py\
  '{prg} tests/inputs/2.fa'
Benchmark #1: ./solution1_kmers_functional.py tests/inputs/2.fa
  Time (mean ± σ):     40.686 s ±  0.443 s    [User: 35.208 s, System:
6.042 s]
  Range (min … max):   40.165 s … 41.349 s    10 runs

Benchmark #2: ./solution2_binary_search.py tests/inputs/2.fa
  Time (mean ± σ):      1.441 s ±  0.037 s    [User: 1.903 s, System:
0.255 s]
```

```
  Range (min … max):    1.378 s …  1.492 s    10 runs

Summary
  './solution2_binary_search.py tests/inputs/2.fa' ran
   28.24 ± 0.79 times faster than './solution1_kmers_functional.py
tests/inputs/2.fa'
```

When I was searching from the maximum *k* value and iterating down, I was performing a *linear* search through all the possible values. This means the time to search grows in proportion (linearly) to the number *n* of values. A binary search, by contrast, grows at a rate of log *n*. It's common to talk about the runtime growth of algorithms using *Big O* notation, so you might see binary search described as O(log *n*) whereas linear searching is O(*n*) — which is much, much worse.

Going Further

- As with the suggestion in Chapter 9, add a Hamming distance option that will allow for the indicated number of differences when deciding on a shared k-mer.

Review

- K-mers can be used to find conserved regions of sequences.
- Lists of lists can be combined into a single list using `itertools.chain()`.
- A binary search can be used on sorted values to find a value more quickly than searching through the list linearly.
- Hill-climbing is one way to maximize the input to a function.
- The `key` option for `min()` and `max()` is a function that is applied to the values before comparing them.

Chapter 11. Finding a Protein Motif: Fetching Data and Using Regular Expressions

We've spent quite a bit of time now looking for sequence motifs. As described on the Rosalind MPRT challenge, shared or conserved sequences in proteins imply shared functions. In this exercise, I need to identify protein sequences that contain the N-glycosylation motif. The input to the program is a list of protein IDs which will be used to download the sequences from the UniProt website. After demonstrating how to manually and programmatically download the data, I'll show how to find the motif using both a regular expression and by writing a manual solution.

You will learn:

- How to fetch data from the internet
- How to write a regular expression to find the N-glycosylation motif
- How to manually find the N-glycosylation motif

Getting Started

All the code and tests for this program are located in the *11_mprt* directory. To begin, copy the first solution to the program `mprt.py`:

```
$ cd 11_mprt
$ cp solution1_regex.py mprt.py
```

Inspect the usage:

```
$ ./mprt.py -h
usage: mprt.py [-h] [-d DIR] FILE

Find locations of N-glycosylation motif

positional arguments:
  FILE                  Input text file of UniProt IDs ❶

optional arguments:
  -h, --help            show this help message and exit
  -d DIR, --download_dir DIR ❷
                        Directory for downloads (default: fasta)
```

❶ The required positional argument is a file of protein IDs.

❷ The optional download directory name defaults to *fasta*.

The input file will list protein IDs, one-per-line. The proteins IDs provided in the example comprise the first test input file:

```
$ cat tests/inputs/1.txt
A2Z669
B5ZC00
P07204_TRBM_HUMAN
P20840_SAG1_YEAST
```

Run the program using this as the argument. The output of the program lists each protein ID containing the N-glycosylation motif and the locations where it can be found:

```
$ ./mprt.py tests/inputs/1.txt
B5ZC00
85 118 142 306 395
P07204_TRBM_HUMAN
47 115 116 382 409
P20840_SAG1_YEAST
79 109 135 248 306 348 364 402 485 501 614
```

The first time the program runs, it will fetch the sequence data for each protein ID and write FASTA files to the indicated download directory. All subsequent runs using these protein IDs will be faster as the cached data will be used unless you remove the download directory, for instance, by using the `make clean` target.

After running the preceding command, you should see that the default *fasta* directory has been created. Inside you should find four FASTA files. The headers for some of the records are quite long. I've broken them here so they won't wrap, but the actual headers must be on a single line:

```
$ head -2 fasta/*
==> fasta/A2Z669.fasta <==
>sp|A2Z669|CSPLT_ORYSI CASP-like protein 5A2 OS=Oryza sativa subsp.
 indica OX=39946 GN=OsI_33147 PE=3 SV=1
MRASRPVVHPVEAPPPAALAVAAAAVAVEAGVGAGGGAAAHGGENAQPRGVRMKDPPGAP

==> fasta/B5ZC00.fasta <==
>sp|B5ZC00|SYG_UREU1 Glycine--tRNA ligase OS=Ureaplasma urealyticum
 serovar 10 (strain ATCC 33699 / Western) OX=565575 GN=glyQS PE=3 SV=1
MKNKFKTQEELVNHLKTVGFVFANSEIYNGLANAWDYGPLGVLLKNNLKNLWWKEFVTKQ

==> fasta/P07204_TRBM_HUMAN.fasta <==
>sp|P07204|TRBM_HUMAN Thrombomodulin OS=Homo sapiens OX=9606 GN=THBD
PE=1 SV=2
MLGVLVLGALALAGLGFPAPAEPQPGGSQCVEHDCFALYPGPATFLNASQICDGLRGHLM

==> fasta/P20840_SAG1_YEAST.fasta <==
>sp|P20840|SAG1_YEAST Alpha-agglutinin OS=Saccharomyces cerevisiae
 (strain ATCC 204508 / S288c) OX=559292 GN=SAG1 PE=1 SV=2
MFTFLKIILWLFSLALASAININDITFSNLEITPLTANKQPDQGWTATFDFSIADASSIR
```

Run `make test` to see the kinds of tests your program should pass. When you're ready, start the program from scratch:

```
$ new.py -fp 'Find locations of N-glycosylation motif' mprt.py
Done, see new script "mprt.py".
```

You should define a positional file argument and an optional download directory as the arguments to the program:

```
class Args(NamedTuple):
    file: TextIO ❶
    download_dir: str ❷


def get_args() -> Args:
    """Get command-line arguments"""

    parser = argparse.ArgumentParser(
        description='Find location of N-glycosylation motif',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file',
                        help='Input text file of UniProt IDs',
                        metavar='FILE',
                        type=argparse.FileType('rt')) ❸

    parser.add_argument('-d',
                        '--download_dir',
                        help='Directory for downloads',
                        metavar='DIR',
                        type=str,
                        default='fasta') ❹

    args = parser.parse_args()

    return Args(args.file, args.download_dir)
```

❶ The *file* should be a filehandle.

❷ The *download_dir* will be a string.

❸ Ensure the argument is a readable text file.

❹ I always set reasonable default values for options.

Ensure your program can create the usage, then start by printing the protein IDs from the file. Each ID is terminated by the newline, so I'll

use the `str.rstrip()` (*right strip*) method to remove any whitespace from the right side:

```
def main() -> None:
    args = get_args()
    for prot_id in map(str.rstrip, args.file):
        print(prot_id)
```

Run the program and make sure you see the protein IDs:

```
$ ./mprt.py tests/inputs/1.txt
A2Z669
B5ZC00
P07204_TRBM_HUMAN
P20840_SAG1_YEAST
```

If you run `pytest`, you should pass the first three tests and fail the fourth. You are welcome to complete the program if you feel you have enough to go on. Keep reading if you want some more direction.

Downloading Sequences Files on the Command Line

The first order of business is fetching the protein sequences. The UniProt information for each protein is found by substituting the protein ID into the URL (uniform resource locator) *http://www.uniprot.org/uniprot/{uniprot_id}*. I'll change the program to print this string instead:

```
def main() -> None:
    args = get_args()
    for prot_id in map(str.rstrip, args.file):
        print(f'http://www.uniprot.org/uniprot/{prot_id}')
```

You should now see this output:

```
$ ./mprt.py tests/inputs/1.txt
http://www.uniprot.org/uniprot/A2Z669
http://www.uniprot.org/uniprot/B5ZC00
http://www.uniprot.org/uniprot/P07204_TRBM_HUMAN
http://www.uniprot.org/uniprot/P20840_SAG1_YEAST
```

Paste the first URL into your web browser and inspect the page. There is a wealth of data, all in a human-readable format. Scroll down to the sequence, and you should see 203 amino acids. It would be just awful to have to parse this page to extract the sequence. Luckily, I can append *.fasta* to the URL and get a FASTA file of the sequence.

Before I show you how to download the sequences using Python, I think you should know how to do this using command-line tools as you'll find this is a very useful skill. From the command line, you could use the `curl` command (which you may need to install) to download the sequence. By default, this will print the contents of the file to `STDOUT`:

```
$ curl https://www.uniprot.org/uniprot/A2Z669.fasta
>sp|A2Z669|CSPLT_ORYSI CASP-like protein 5A2 OS=Oryza sativa subsp.
 indica OX=39946 GN=OsI_33147 PE=3 SV=1
MRASRPVVHPVEAPPPAALAVAAAAVAVEAGVGAGGGAAAHGGENAQPRGVRMKDPPGAP
GTPGGLGLRLVQAFFAAAALAVMASTDDFPSVSAFCYLVAAAILQCLWSLSLAVVDIYAL
LVKRSLRNPQAVCIFTIGDGITGTLTLGAACASAGITVLIGNDLNICANNHCASFETATA
MAFISWFALAPSCVLNFWSMASR
```

You could either redirect this to a file:

```
$ curl https://www.uniprot.org/uniprot/A2Z669.fasta > A2Z669.fasta
```

Or use the `-o|--output` option to name the output file:

```
$ curl -o A2Z669.fasta https://www.uniprot.org/uniprot/A2Z669.fasta
```

You can also use the `wget` (*web get*, which may also need to be installed) command to download the sequence file like so.

```
$ wget https://www.uniprot.org/uniprot/A2Z669.fasta
```

Whichever tool you use, you should now have a file called *A2Z669.fasta* with the sequence data:

```
$ cat A2Z669.fasta
>sp|A2Z669|CSPLT_ORYSI CASP-like protein 5A2 OS=Oryza sativa subsp.
 indica OX=39946 GN=OsI_33147 PE=3 SV=1
MRASRPVVHPVEAPPPAALAVAAAAVAVEAGVGAGGGAAAHGGENAQPRGVRMKDPPGAP
GTPGGLGLRLVQAFFAAAALAVMASTDDFPSVSAFCYLVAAAILQCLWSLSLAVVDIYAL
LVKRSLRNPQAVCIFTIGDGITGTLTLGAACASAGITVLIGNDLNICANNHCASFETATA
MAFISWFALAPSCVLNFWSMASR
```

You could automate this by writing a `bash` script:

```
#!/usr/bin/env bash ❶

if [[ $# -ne 1 ]]; then ❷
    printf "usage: %s FILE\n" $(basename "$0") ❸
    exit 1 ❹
fi

OUT_DIR="fasta" ❺
[[ ! -d "$OUT_DIR" ]] && mkdir -p "$OUT_DIR" ❻

while read -r PROT_ID; do ❼
    echo "$PROT_ID"   ❽
    URL="https://www.uniprot.org/uniprot/${PROT_ID}" ❾
    OUT_FILE="$OUT_DIR/${PROT_ID}.fasta" ❿
    wget -q -o "$OUT_FILE" "$URL" ⓫
done < $1 ⓬

echo "Done, see output in \"$OUT_DIR\"." ⓭
```

❶ The shebang (`#!`) should use the `env` (environment) to find `bash`.

❷ Check the number of arguments (`$#`) is 1.

❸ Print a usage statement using the name of the basename of the program (`$0`).

❹ Exit with a non-zero value.

❺ Define the output directory to be *fasta*. Note that in `bash` you can have no spaces around the = for variable assignment.

❻ Create the output directory if it does not exist.

❼ Read each line from the file into the `PROT_ID` variable.

❽ Print the current protein ID so the user knows something is happening.

❾ Construct the URL by using variable interpolation inside double-quotes.

❿ Construct the output filename by combining the output directory and the protein ID.

⓫ Call `wget` with the `-q` (quiet) flag to fetch the URL into the output file.

⓬ This reads each line from the first positional argument (`$1`) which is the input filename.

⓭ Let the user know the program has finished and where to find the output.

I can run this like so:

```
$ ./fetch_fasta.sh tests/inputs/1.txt
A2Z669
B5ZC00
P07204_TRBM_HUMAN
P20840_SAG1_YEAST
Done, see output in "fasta".
```

Now there should be a *fasta* directory containing the four FASTA files. One way to write the `mprt.py` program would be to fetch all the input files first using something like this and then providing the FASTA files as arguments. This is a very common pattern in bioinformatics, and writing a shell script like this is a great way to document exactly how you retrieved the data for your analysis. Be sure you always commit programs like this to your source repository and consider adding a Makefile target with a name like *fasta* that is flush-left followed by a colon and the command on the next line indented with a single tab character:

```
fasta:
	./fetch_fasta.sh tests/inputs/1.txt
```

Now you should be able to `make fasta` to automate the process of getting your data. By writing the program to accept the input file as an argument rather than hard-coding it, I can use this program and multiple Makefile targets to automate the process of downloading many different datasets. Reproducibility FTW.

Downloading Sequences Files with Python

I'll translate the `bash` utility to Python now. As you can see from the preceding program, there are several steps involved to fetch each sequence file. I don't want this to be a part of the `main()` as it will clutter the program, so I'll write a function for this:

```
def fetch_fasta(fh: TextIO, fasta_dir: str) -> List[str]: ❶
    """ Fetch the FASTA files into the download directory """

    return [] ❷
```

❶ The function will accept a filehandle for the protein IDs and a download directory name and will return a list of the files that were downloaded. Be sure to add `typing.List` to your imports.

❷ For now return an empty list.

I want to call it like this:

```
def main() -> None:
    args = get_args()
    files = fetch_fasta(args.file, args.download_dir)
    print('\n'.join(files))
```

Run your program and ensure it compiles and prints nothing. Now add the following Python code to fetch the sequences. You'll need to import `os`, `sys`, and `requests`, a library for making web requests:

```
def fetch_fasta(fh: TextIO, fasta_dir: str) -> List[str]:
    """ Fetch the FASTA files into the download directory """

    if not os.path.isdir(fasta_dir): ❶
        os.makedirs(fasta_dir)

    files = [] ❷
    for prot_id in map(str.rstrip, fh): ❸
        fasta = os.path.join(fasta_dir, prot_id + '.fasta') ❹
        if not os.path.isfile(fasta): ❺
            url = f'http://www.uniprot.org/uniprot/{prot_id}.fasta' ❻
            response = requests.get(url) ❼
            if response.status_code == 200: ❽
                print(response.text, file=open(fasta, 'wt')) ❾
            else:
                print(f'Error fetching "{url}": "
{response.status_code}"',
                      file=sys.stderr) ❿
                continue ⓫

        files.append(fasta) ⓬

    return files ⓭
```

❶ Create the output directory if it does not exist.

❷ Initialize the return list of filenames.

❸ Read each protein ID from the file.

❹ Construct the output filename by combining the output directory plus the protein ID.

❺ Check if the file already exists.

❻ Construct the URL to the FASTA file.

❼ Make a *GET* request for the file.

❽ A response code of 200 indicates success.

❾ Write the text of the response to the output file.

❿ Print a warning to `STDERR` that the file could not be fetched.

⓫ Skip to the next iteration.

⓬ Append the file to the return list.

⓭ Returns the files that now exist locally.

That logic almost exactly mirrors that of the `bash` program. If you run your program again, there should be a *fasta* directory with the four files, and the program should print the names of the downloaded files:

```
$ ./mprt.py tests/inputs/1.txt
fasta/A2Z669.fasta
fasta/B5ZC00.fasta
fasta/P07204_TRBM_HUMAN.fasta
fasta/P20840_SAG1_YEAST.fasta
```

Writing a Regular Expression to Find the Motif

The Rosalind page notes:

> *To allow for the presence of its varying forms, a protein motif is represented by a shorthand as follows: `[XY]` means either X or Y and `{X}` means any amino acid except X. For example, the N-glycosylation motif is written as `N{P}[ST]{P}`.*

The Prosite website is a database of protein domains, families, and functional sites. The details for the for the N-glycosylation motif shows a similar convention for the *consensus pattern* of `N-{P}-[ST]-{P}`. Both patterns are extremely close to the regular expression shown in Figure 11-1.

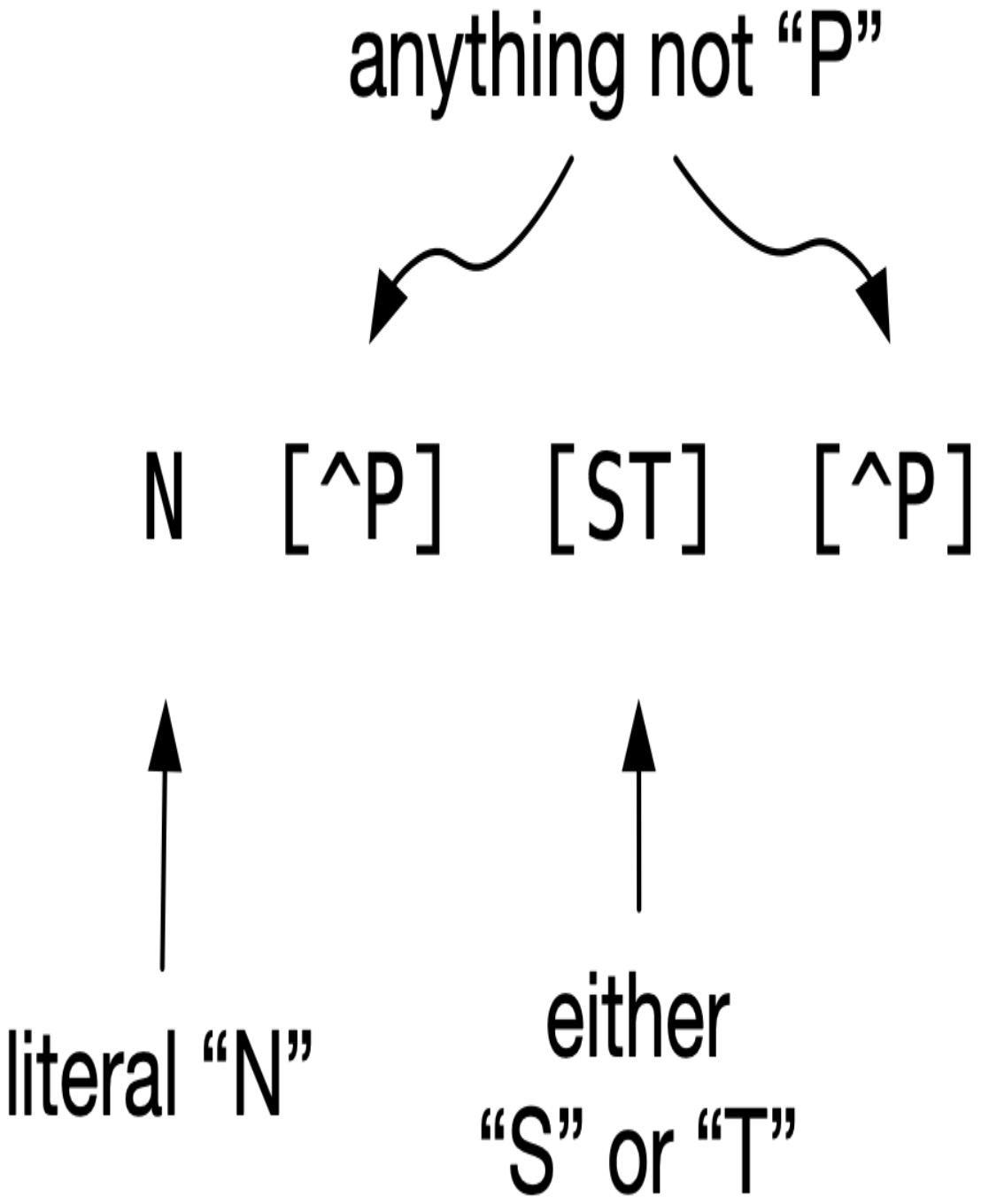


Figure 11-1. A regular expression for the N-glycosylation protein motif.

In a regex, the `N` indicates the literal character *N*. The `[ST]` is a character class representing either the character *S* or *T*. It's the same as the regex `[GC]` I wrote in Chapter 5 to find either *G* or *C*.

The `[^P]` is a *negated* character class, which means it will match any character that is *not P*.

Some people (OK, mostly just me) like to represent regexes using the notation of finite state machines (FSM) such as the one shown in Figure 11-2. Imagine the pattern entering on the left. It first needs to find the letter *N* to proceed to the next step. Next can be any character that is not the letter *P*. After that, the graph has two alternate paths through the letters *S* or *T* which must be followed again by a not-P character. If the pattern makes it to the double circle, the match was successful.

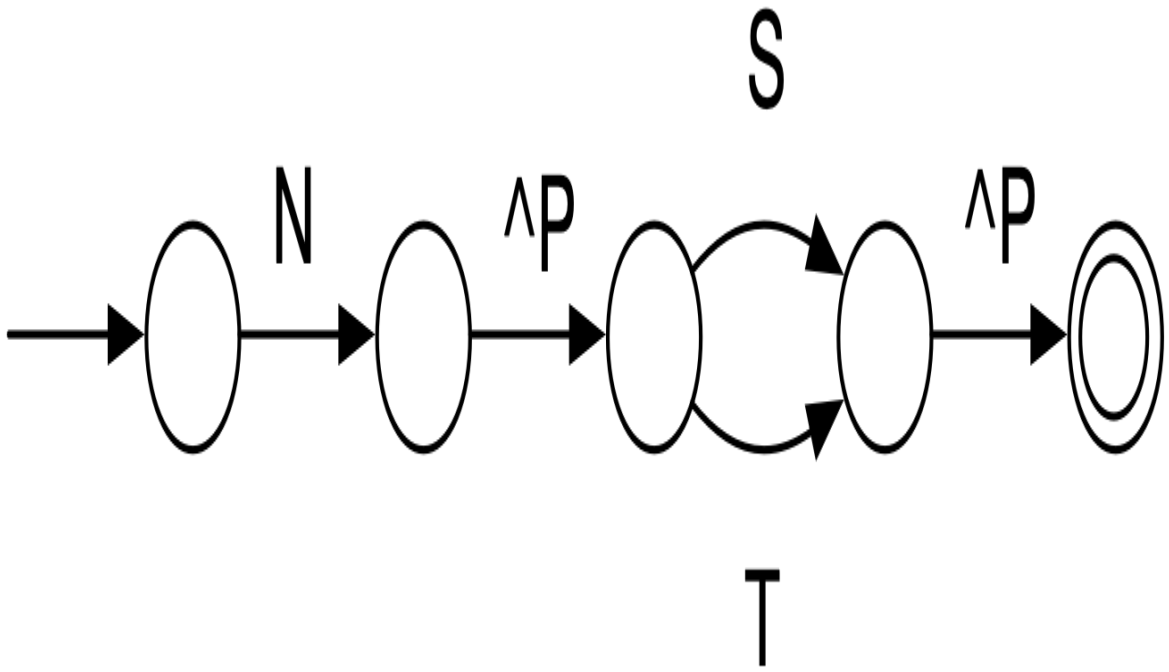


Figure 11-2. Graphical depiction of a finite state machine (FSM) to identify the N-glycosylation motif.

In Chapter 8, I pointed out a problem when using regular expressions to find overlapping text. There are no instances of this in the first test file, but another of the datasets I used to solve the problem did have two overlapping motifs. Let me demonstrate in the REPL:

```
>>> import re
>>> regex = re.compile('N[^P][ST][^P]')
```

> **NOTE**
>
> I'm using the `re.compile()` function here to force the regex engine to parse the pattern and create the necessary internal code to do the matching. This is similar to how compiled languages like C use source code that humans can edit and read into machine code that computers can directly execute. The time to make this transformation happens just once when you use `re.compile()` whereas functions like `re.search()` must recompile the regex on each call.

Here is the relevant portion of the protein sequence for *P07204_TRBM_HUMAN* that has the pattern starting at both the first and second positions (see Figure 11-3). The `re.findall()` function shows that only the pattern starting at the first position is found:

```
>>> seq = 'NNTSYS'
>>> regex.findall(seq)
['NNTS']
```



Figure 11-3. This sequence contains two copies of the motif that overlap.

As in Chapter 8, the solution is to wrap the regex in a look-ahead assertion which itself will need to be wrapped in capturing parentheses:

```
>>> regex = re.compile('(?=(N[^P][ST][^P]))')
>>> regex.findall(seq)
['NNTS', 'NTSY']
```

I need to know the positions of the matches which I can get from `re.finditer()`. This will return a list of `re.Match` objects, each of which has a `match.start()` function that will return the zero-offset index of the match's starting position. I need to add one to report the position using 1-based counting:

```
>>> [match.start() + 1 for match in regex.finditer(seq)]
[1, 2]
```

This should be enough for you to solve the rest of the problem. Keep hacking until you pass all the tests. Be sure to download a dataset from the Rosalind site and verify that the solution gives an answer that passes the test there, too. See if you can also write a version that doesn't use regular expressions. Go back and study the FSM model and think about how you express those ideas in Python.

Solutions

I will present two variations to solve this problem. The first uses a regular expression to find the motif, and the second imagines how to solve the problem in a horrible, desolate intellectual wasteland where regular expressions don't exist.

Solution 1: Using a Regular Expression

The following is my final solution using a regular expression. Be sure to import `re` and `Bio.SeqIO` for this:

```
def main():
    args = get_args()
    files = fetch_fasta(args.file, args.download_dir) ❶
    regex = re.compile('(?=(N[^P][ST][^P]))') ❷

    for file in files: ❸
        prot_id, _ = os.path.splitext(os.path.basename(file)) ❹
        if recs := list(SeqIO.parse(file, 'fasta')): ❺
            if matches := list(regex.finditer(str(recs[0].seq))): ❻
```

```
        print(prot_id) ❼
        print(*[match.start() + 1 for match in matches]) ❽
```

❶ Fetch the sequence files for the protein IDs in the given file. Put the files into the indicated download directory.

❷ Compile the regex for the N-glycosylation motif.

❸ Iterate through the files.

❹ Get the protein ID from the basename of the file and split off the file extension.

❺ Attempt to retrieve the sequence records from the FASTA file. There should only be one.

❻ Coerce the sequence of the first record from a `Seq` to a `str`, then try to find all the matches for the motif.

❼ Print the protein ID.

❽ Print all the matches, correcting to 1-based counting.

In this solution, I used the `os.path.basename()` and `os.path.splitext()` functions. I often use these, so I want to make sure you understand exactly what they do. I first introduced the `os.path.basename()` in Chapter 2. This function will return just the filename from a path that might include directories:

```
>>> import os
>>> basename = os.path.basename('fasta/B5ZC00.fasta')
>>> basename
'B5ZC00.fasta'
```

The `os.path.splitext()` function will break a filename into the part before the file extension and the extension:

```
>>> os.path.splitext(basename)
('B5ZC00', '.fasta')
```

> **NOTE**
>
> File extensions are nothing more than a human convention. They really mean nothing to the operating system. There are many conventions for FASTA extensions including *.fasta*, *.fa*, *.fna* (for nucleotides), and *.faa* (for amino acids). You can put whatever extension you like on a FASTA file or none at all, and just because a file has a FASTA-like extension does not necessarily mean it's *actually* a FASTA file. *Caveat emptor*.

In the preceding code, I don't need the extension, so I assign it to the variable _ (underscore), which is a convention indicating that I don't intend to use the value. I could also use a list slice to grab just the first element from the function:

```
>>> os.path.splitext(basename)[0]
'B5ZC00'
```

Solution 2: Writing a Manual Solution

If I were writing a program like this for production use, I would definitely use a regular expression to find the motif. In this context, though, I wanted to challenge myself to find a manual solution. As usual, I want to write a function to encapsulate this idea, so I stub it out:

```
def find_motif(text: str) -> List[int]: ❶
    """ Find a pattern in some text """

    return [] ❷
```

❶ The function will take some text and will return a list of integers where the motif can be found in the text.

❷ For now, return the empty list.

The biggest reason to have a function is to write a test where I encode examples I expect to match and fail:

```
def test_find_motif() -> None:
    """ Test find_pattern """

    assert find_motif('') == [] ❶
    assert find_motif('NPTX') == [] ❷
    assert find_motif('NXTP') == [] ❸
    assert find_motif('NXSX') == [0] ❹
    assert find_motif('ANXTX') == [1] ❺
    assert find_motif('NNTSYS') == [0, 1] ❻
    assert find_motif('XNNTSYS') == [1, 2] ❼
```

❶ Ensure the function does not do something silly like raise an exception when given the empty string.

❷ This should fail because it has a P in the second position.

❸ This should fail because it has a P in the fourth position.

❹ This should find the motif at the beginning of the string.

❺ This should find the motif not at the beginning of the string.

❻ This should find overlapping motifs at the beginning of the string.

❼ This should find overlapping motifs not at the beginning of the string.

I can add these functions to my `mprt.py` program and run `pytest` on that source code to ensure that the tests *do fail* as expected. Now I need to write the `find_motif()` code that will pass these tests. I decided I would again use k-mers, so I will bring in the `find_kmers()`

function (and test, of course, but I'll omit that here) from Chapters 9 and 10:

```
def find_kmers(seq: str, k: int) -> List[str]:
    """ Find k-mers in string """

    n = len(seq) - k + 1
    return [] if n < 1 else [seq[i:i + k] for i in range(n)]
```

Since the motif is four characters long, I can use this to find all the 4-mers in a sequence:

```
>>> from solution2_manual import find_kmers
>>> seq = 'NNTSYS'
>>> find_kmers(seq, 4)
['NNTS', 'NTSY', 'TSYS']
```

I will also need their positions. The `enumerate()` function I introduced in Chapter 8 will provide both the index and value of the items in a sequence:

```
>>> list(enumerate(find_kmers(seq, 4)))
[(0, 'NNTS'), (1, 'NTSY'), (2, 'TSYS')]
```

I can unpack each position and k-mer while iterating like so:

```
>>> for i, kmer in enumerate(find_kmers(seq, 4)):
...     print(i, kmer)
...
0 NNTS
1 NTSY
2 TSYS
```

Take the first k-mer, *NNTS*. One way to test for this pattern is to manually check each index:

```
>>> kmer = 'NNTS'
>>> kmer[0] == 'N' and kmer[1] != 'P' and kmer[2] in 'ST' and kmer[3]
```

```
!= 'P'
True
```

I know the first two k-mers should match, and this is borne out:

```
>>> for i, kmer in enumerate(find_kmers(seq, 4)):
...   kmer[0] == 'N' and kmer[1] != 'P' and kmer[2] in 'ST' and
kmer[3] != 'P'
...
True
True
False
```

While effective, this is tedious. I would at least like to hide this code in a function:

```
def is_match(s: str) -> bool:
    return s[0] == 'N' and s[1] != 'P' and s[2] in 'ST' and s[3] !=
'P'
```

That makes the code much more readable:

```
>>> for i, kmer in enumerate(find_kmers(seq, 4)):
...     print(i, kmer, is_match(kmer))
...
0 NNTS True
1 NTSY True
2 TSYS False
```

I only want the k-mers that match. I could write this using an if-expression with a guard, which I showed in Chapters 5 and 6:

```
>>> kmers = list(enumerate(find_kmers(seq, 4)))
>>> [i for i, kmer in kmers if is_match(kmer)]
[0, 1]
```

Or using the `starfilter()` function I showed in Chapter 9:

```
>>> from iteration_utilities import starfilter
>>> list(starfilter(lambda i, s: is_match(s), kmers))
```

```
[(0, 'NNTS'), (1, 'NTSY')]
```

I only want the first elements from each of the tuples, so I could use a `map()` to select those:

```
>>> matches = starfilter(lambda i, s: is_match(s), kmers)
>>> list(map(lambda t: t[0], matches))
[0, 1]
```

For what it's worth, Haskell uses tuples extensively and so includes two handy functions in the prelude, `fst()` to get the first element from a 2-tuple and `snd()` to get the second. Be sure to import `typing.Tuple` for this code:

```
def fst(t: Tuple[Any, Any]) -> Any:
    return t[0]

def snd(t: Tuple[Any, Any]) -> Any:
    return t[1]
```

With these functions, I could eliminate the `starfilter()` like this:

```
>>> list(map(fst, filter(lambda t: is_match(snd(t)), kmers)))
[0, 1]
```

Notice a very subtle bug if I try to use the `filter()`/`starmap()` technique I've shown a couple of times:

```
>>> from itertools import starmap
>>> list(filter(None, starmap(lambda i, s: i if is_match(s) else None,
kmers)))
[1]
```

It only returns the second match. Why is that? It's due to using `None` as the predicate to `filter()`. According to `help(filter)`, *If [the] function is None, return the items that are true*. In Chapter 1, I introduced the ideas of truthy and falsey. The Boolean values `True` and `False` are actually represented by the integer values 1 and 0,

respectively; hence, the actual number 0 (either `int` or `float`) is technically `False` which means that any non-zero number is not-`False` or, if you will, truthy. Python will evaluate many data types in a Boolean context to decide if they are truthy or falsey. The empty string, list, and dictionary are all falsey, and so any non-empty instances of these types are truthy.

In this case, using `None` as the predicate for `filter()` causes it to remove the number 0:

```
>>> list(filter(None, [1, 0, 2]))
[1, 2]
```

> **NOTE**
>
> I came to Python from Perl and JavaScript, both of which also silently coerce values given different contexts, so I was not so surprised by this behavior. If you come from a language like C or Haskell that have stricter types, this is probably very troubling. I often feel that Python is a very powerful language if you know exactly what you're doing at all times. This is a high bar, so it's extremely important when writing Python to use types and tests liberally.

In the end, I felt the list comprehension was the easiest to read. Here's how I wrote my function to manually identify the protein motif:

```
def find_motif(text: str) -> List[int]:
    """ Find a pattern in some text """

    def is_match(s: str) -> bool: ❶
        return s[0] == 'N' and s[1] != 'P' and s[2] in 'ST' and s[3]
!= 'P'

    kmers = list(enumerate(find_kmers(text, 4))) ❷
    return [i for i, kmer in kmers if is_match(kmer)] ❸
```

❶ Manually identify the N-glycosylation motif.

❷ Get the positions and values of the 4-mers from the text.

❸ Select those positions for the k-mers matching the motif.

Using this function is almost identical to how I used the regular expression, which is quite the point of hiding complexities behind functions:

```
def main() -> None:
    args = get_args()
    files = fetch_fasta(args.file, args.download_dir)

    for file in files:
        prot_id, _ = os.path.splitext(os.path.basename(file))
        if recs := list(SeqIO.parse(file, 'fasta')):
            if matches := find_motif(str(recs[0].seq)): ❶
                pos = map(lambda p: p + 1, matches) ❷
                print('\n'.join([prot_id, ' '.join(map(str, pos))])) ❸
```

❶ Try to find any matches to the motif.

❷ The matches are a list of 0-based indexes, so add 1 to each.

❸ Convert the integer values to strings and join them on spaces to print.

Although this works and was fun (YMMV) to write, I would not want to use or maintain this code. I hope it gives you a sense of how much work the regular expression is doing for us. Regexes allow me to describe *what* I want, not *how* to get it.

Going Further

- The Eukaryotic Linear Motifs database example, provides regexes to find motifs that define functional sites in proteins.

Write a program to search for any occurrence of any pattern in a given set of FASTA files.

Review

- You can use command-line utilities like `curl` and `wget` to fetch data from the internet. Sometimes it makes sense to write a shell script for such tasks, and sometimes it's better to encode this using a language like Python.
- A regular expression can find the N-glycosylation motif, but it's necessary to wrap it in a look-ahead assertion and capturing parentheses to find overlapping matches.
- It's possible to manually find the N-glycosylation motif, but it's not easy.
- The `os.path.splitext()` is useful when you need to separate a filename from the extension.
- File extensions are conventions and may be unreliable.

Chapter 12. Inferring mRNA from Protein: Products and Reductions of Lists

As described in the Rosalind MRNA challenge, the goal of this program is to reverse engineer all the possible mRNA strings that could produce a given protein sequence. I hope to show that I can turn the tables on regular expressions by trying to generate all the strings that could be matched by a particular pattern. I'll show how to create the products of numbers and lists as well as how to *reduce* any list of values to a single value.

You will learn:

- How to use `itertools.product()` to create combinations of values
- How to use the `functools.reduce()` function to create a `product()` function for multiplying numbers
- How to use Python's `%` modulo operator
- What monoids are
- How to reverse a dictionary by flipping the keys and values

Getting Started

You should work in the *12_mrna* directory of the repository. Begin by copying the first solution to the program `mrna.py`:

```
$ cd 12_mrna/
$ cp solution1_dict.py mrna.py
```

As usual, inspect the usage first:

```
$ ./mrna.py -h
usage: mrna.py [-h] [-m int] protein

Inferring mRNA from Protein

positional arguments:
  protein               Input protein or file ❶

optional arguments:
  -h, --help            show this help message and exit
  -m int, --modulo int  Modulo value (default: 1000000) ❷
```

❶ The required positional argument is a protein sequence or a file containing a protein sequence.

❷ The modulo option defaults to 1,000,000.

Run the program with the Rosalind example of *MA* and verify it produces the number 12:

```
$ ./mrna.py MA
12
```

The program will also read an input file for the sequence. The first input file has a sequence that is 998 residues long, and the result should be the *448832*:

```
$ ./mrna.py tests/inputs/1.txt
448832
```

Run the program with any other inputs you care to and also execute the tests. When you are satisfied you understand how the program should work, start over:

```
$ new.py -fp 'Infer mRNA from Protein' mrna.py
Done, see new script "mrna.py".
```

Define the parameters as described in the usage. The *protein* may be a string or a filename, but I chose to model the parameter as a string. If the user provides a file, I will read the contents and pass this to the program as I first demonstrated in Chapter 3:

```
class Args(NamedTuple):
    """ Command-line arguments """
    protein: str ❶
    modulo: int ❷


def get_args() -> Args:
    """ Get command-line arguments """

    parser = argparse.ArgumentParser(
        description='Infer mRNA from Protein',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('protein', ❶
                        metavar='protein',
                        type=str,
                        help='Input protein or file')

    parser.add_argument('-m', ❷
                        '--modulo',
                        metavar='int',
                        type=int,
                        default=1000000,
                        help='Modulo value')

    args = parser.parse_args()

    if os.path.isfile(args.protein): ❸
        args.protein = open(args.protein).read().rstrip()

    return Args(args.protein, args.modulo)
```

❶ The required protein argument should be a string which may be a filename.

❷ The modulo option is an integer that will default to 1,000,000.

❸ If the protein argument names an existing file, read the protein sequence from the file.

Change your `main()` to print the protein sequence:

```
def main() -> None:
    args = get_args()
    print(args.protein)
```

Verify that your program prints the protein both from the command line or a file:

```
$ ./mrna.py MA
MA
$ ./mrna.py tests/inputs/1.txt | wc -c
     998
```

Your program should pass the first two tests and fail the third.

Creating the Product of Lists

Your program should print the response *12* when the input is *MA*. What does this mean? It's the number of possible mRNA strings that could have produced this protein squence as shown in Figure 12.1. The amino acid *M* is encoded by one mRNA codon sequence (*AUG*), while *A* has four possible codons (*GCA*, *GCC*, *GCG*, *GCU*), and the stop codon has three (*UAA*, *UAG*, *UGA*). The product of these three groups is 1 × 4 × 3 = 12.

M	AUG	AUG, GCA, UAA AUG, GCA, UAG AUG, GCA, UGA AUG, GCC, UAA AUG, GCC, UAG AUG, GCC, UGA AUG, GCG, UAA AUG, GCG, UAG AUG, GCG, UGA AUG, GCU, UAA AUG, GCU, UAG AUG, GCU, UGA
A	GCA GCC GCG GCU	
Stop	UAA UAG UGA	

Figure 12-1. The product of all the codons that could produce the protein sequence MA.

I can produce this list in the REPL by using the `itertools.product()` function to create the Cartesian product of the codons:

```
>>> from itertools import product
>>> from pprint import pprint
>>> codons = [['AUG'], ['GCA', 'GCC', 'GCG', 'GCU'], ['UAA', 'UAG',
'UGA']]
>>> combos = list(product(*codons))
>>> pprint(combos)
[('AUG', 'GCA', 'UAA'),
 ('AUG', 'GCA', 'UAG'),
 ('AUG', 'GCA', 'UGA'),
 ('AUG', 'GCC', 'UAA'),
 ('AUG', 'GCC', 'UAG'),
 ('AUG', 'GCC', 'UGA'),
 ('AUG', 'GCG', 'UAA'),
 ('AUG', 'GCG', 'UAG'),
 ('AUG', 'GCG', 'UGA'),
 ('AUG', 'GCU', 'UAA'),
 ('AUG', 'GCU', 'UAG'),
 ('AUG', 'GCU', 'UGA')]
```

The length of this product is 12:

```
>>> len(combos)
12
```

As the length of the input protein sequence grows, the number of possible combinations will grow extremely large, so the problem expects the answer to be this raw number *modulo* 1,000,000. For example, the number of combinations that could produce the 998 aa protein in the first input file is approximately 8.98×10^{29}. The modulo operation returns the remainder when one number is divided by another. For example, 5 mod 2 = 1 because $5 = (2 \times 2) + 1$. Python has the `%` operator to compute the modulo:

```
>>> 5 % 2
1
```

The answer for that 998 aa protein is the more reasonable 448,832.

The `prot.py` program I showed in Chapter 7 has an RNA codon table you could use to find all the possible codons for each residue in a protein string. This should be enough information to create a working solution. Keep working until your program passes all the tests.

Solutions

I will present three solutions that mostly differ in the structure of a dictionary used to represent the RNA translation information.

Solution 1: Using a Dictionary for the RNA Codon Table

For my first solution, I used the RNA codon table from Chapter 7's solution to find the number of codons that encode each residue:

```
>>> c2aa = {
...     'AAA': 'K', 'AAC': 'N', 'AAG': 'K', 'AAU': 'N', 'ACA': 'T',
...     'ACC': 'T', 'ACG': 'T', 'ACU': 'T', 'AGA': 'R', 'AGC': 'S',
...     'AGG': 'R', 'AGU': 'S', 'AUA': 'I', 'AUC': 'I', 'AUG': 'M',
...     'AUU': 'I', 'CAA': 'Q', 'CAC': 'H', 'CAG': 'Q', 'CAU': 'H',
...     'CCA': 'P', 'CCC': 'P', 'CCG': 'P', 'CCU': 'P', 'CGA': 'R',
...     'CGC': 'R', 'CGG': 'R', 'CGU': 'R', 'CUA': 'L', 'CUC': 'L',
...     'CUG': 'L', 'CUU': 'L', 'GAA': 'E', 'GAC': 'D', 'GAG': 'E',
...     'GAU': 'D', 'GCA': 'A', 'GCC': 'A', 'GCG': 'A', 'GCU': 'A',
...     'GGA': 'G', 'GGC': 'G', 'GGG': 'G', 'GGU': 'G', 'GUA': 'V',
...     'GUC': 'V', 'GUG': 'V', 'GUU': 'V', 'UAC': 'Y', 'UAU': 'Y',
...     'UCA': 'S', 'UCC': 'S', 'UCG': 'S', 'UCU': 'S', 'UGC': 'C',
...     'UGG': 'W', 'UGU': 'C', 'UUA': 'L', 'UUC': 'F', 'UUG': 'L',
...     'UUU': 'F', 'UAA': '*', 'UAG': '*', 'UGA': '*',
... }
```

I want to iterate over each amino acid in the protein sequence *MA* plus the stop codon `*` to find all the encoding codons:

```
>>> protein = 'MA'
>>> for aa in protein + '*':
...     print(aa, [c for c, res in c2aa.items() if res == aa])
...
M ['AUG']
A ['GCA', 'GCC', 'GCG', 'GCU']
* ['UAA', 'UAG', 'UGA']
```

The list comprehension in the preceding code is really quite powerful, combining operations using `map()` and `filter()` which I find much less readable:

```
>>> for aa in protein + '*':
...   codons = map(lambda t: t[0], filter(lambda t: t[1] == aa,
c2aa.items()))
...   print(aa, list(codons))
...
M ['AUG']
A ['GCA', 'GCC', 'GCG', 'GCU']
* ['UAA', 'UAG', 'UGA']
```

In the introduction, I showed the actual combinations of the codons, but I only need the *number* of codons for a given residue:

```
>>> possible = [
...     len([c for c, res in codon_to_aa.items() if res == aa])
...     for aa in protein + '*'
... ]
>>>
>>> possible
[1, 4, 3]
```

The answer is the multiplication of these values. In Chapter 5, I showed that Python has a built-in `sum()` function that will return the addition of a list of numbers:

```
>>> sum(possible)
8
```

This might lead you to believe that there is also a `product()` function. Alas, there is not; however, this gives us the chance to talk about *reducing* a sequence of values to a single value. This is actually what the `sum()` fuction does with the numbers 1, 4, and 3. First I add 1 + 4 to get 5, then add 5 + 3 to get 8. If I change the `+` to `*`, then I get a product and the result is 12 as shown in Figure 12-2.

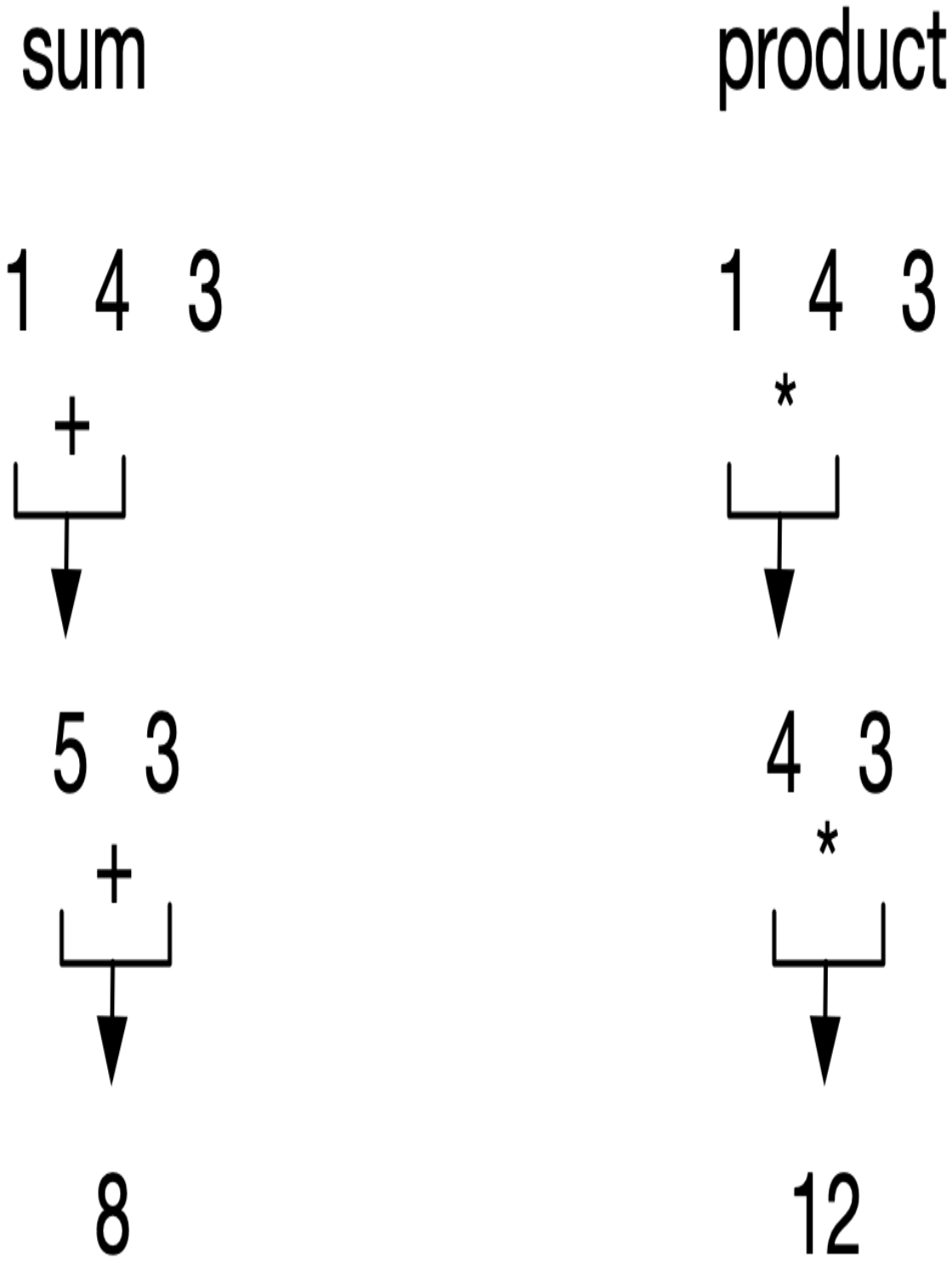


Figure 12-2. Reducing a list of numbers using addition and multiplication.

This is the idea behind reducing a list of values, and it's precisely what the `functools.reduce()` fuction helps us to do. This is another higher-order function like `filter()` and `map()` and others I've used throughout, but with an important difference that the `lambda` function will receive *two* arguments instead of one. The documentation actually shows how to write `sum()`:

```
reduce(...)
    reduce(function, sequence[, initial]) -> value

    Apply a function of two arguments cumulatively to the items of a
sequence,
    from left to right, so as to reduce the sequence to a single
value.
    For example, reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])
calculates
    ((((1+2)+3)+4)+5).  If initial is present, it is placed before
the items
    of the sequence in the calculation, and serves as a default when
the
    sequence is empty.
```

Here is how I could use this:

```
>>> from functools import reduce
>>> reduce(lambda x, y: x + y, possible)
8
```

To create a product, I can change the addition to multiplication:

```
>>> reduce(lambda x, y: x * y, possible)
12
```

MONOIDS

For what it's worth, a homogenous list of numbers or strings could be thought of as a *monoid* which is an algebraic structure with a single associative binary operation and an identity element. For a list of numbers under addition, the binary operation is `+`, and the identity element is 0. That is, the identity element is whatever value you could combine under the operation without changing a value. Since 0 + *n* = *n*, then 0 is the identity element for addition and is the appropriate value for the *initial* argument to `reduce()`:

```
>>> reduce(lambda x, y: x + y, [1, 2, 3, 4], 0)
10
```

It's also the appropriate return value when adding an empty list of numbers:

```
>>> reduce(lambda x, y: x + y, [], 0)
0
```

Under multiplication, the operator is `*` and the identity element is 1 because 1 * *n* = *n*:

```
>>> reduce(lambda x, y: x * y, [1, 2, 3, 4], 1)
24
```

The product of an empty list is 1, so this is the correct initial value for `reduce()`:

```
>>> reduce(lambda x, y: x * y, [], 1)
1
```

A list of strings under concatenation would also use `+`, and the result is a new string:

```
>>> reduce(lambda x, y: x + y, ['M', 'A'], '')
'MA'
```

The identity element for strings is the empty string:

```
>>> reduce(lambda x, y: x + y, [], '')
''
```

This is the same value returned when passing the empty list to `str.join()`:

```
>>> ''.join([])
''
```

Even lists themselves are monoids. They can be combined using the `+` operator or `operator.concat()`, and the identity value is the empty list:

```
>>> reduce(operator.concat, [['A'], ['list', 'of'], ['values']],
[])
['A', 'list', 'of', 'values']
```

Notice that these identity values are used as the default value for the `collections.defaultdict()` function. That is, a `defaultdict(int)` initializes to 0, `defaultdict(str)` uses the empty string, and `defaultdict(list)` uses the empty list.

Monoids such as these can be reduced to a single value by progressively applying the *associative binary operation*, hence the name of the `reduce()` function. If you find this kind of theory interesting, you should might be interested to learn more about the Haskell programming language and the field of *category theory*.

Note that Python has an `id()` function that will “return the identity of an object” which is a unique numerical representation of a

> value in Python, akin to a memory address, and so is not at all like monoidal identity.

I can use `reduce()` to write my own `product()` function:

```
def product(xs: List[int]) -> int: ❶
    """ Return the product """

    return reduce(lambda x, y: x * y, xs, 1) ❷
```

❶ Return the product of a list of integers.

❷ Use the `functools.reduce()` function to progressively multiply the values.

Here is the test I wrote:

```
def test_product() -> None:
    """ Test product """

    assert product([]) == 1
    assert product([4]) == 4
    assert product([1, 2, 3, 4]) == 24
```

The final answer is the products of these numbers modulo the given argument. Here is how I put it all together:

```
def main() -> None:
    args = get_args()
    codon_to_aa = { ❶
        'AAA': 'K', 'AAC': 'N', 'AAG': 'K', 'AAU': 'N', 'ACA': 'T',
        'ACC': 'T', 'ACG': 'T', 'ACU': 'T', 'AGA': 'R', 'AGC': 'S',
        'AGG': 'R', 'AGU': 'S', 'AUA': 'I', 'AUC': 'I', 'AUG': 'M',
        'AUU': 'I', 'CAA': 'Q', 'CAC': 'H', 'CAG': 'Q', 'CAU': 'H',
        'CCA': 'P', 'CCC': 'P', 'CCG': 'P', 'CCU': 'P', 'CGA': 'R',
        'CGC': 'R', 'CGG': 'R', 'CGU': 'R', 'CUA': 'L', 'CUC': 'L',
        'CUG': 'L', 'CUU': 'L', 'GAA': 'E', 'GAC': 'D', 'GAG': 'E',
        'GAU': 'D', 'GCA': 'A', 'GCC': 'A', 'GCG': 'A', 'GCU': 'A',
        'GGA': 'G', 'GGC': 'G', 'GGG': 'G', 'GGU': 'G', 'GUA': 'V',
```

```
        'GUC': 'V', 'GUG': 'V', 'GUU': 'V', 'UAC': 'Y', 'UAU': 'Y',
        'UCA': 'S', 'UCC': 'S', 'UCG': 'S', 'UCU': 'S', 'UGC': 'C',
        'UGG': 'W', 'UGU': 'C', 'UUA': 'L', 'UUC': 'F', 'UUG': 'L',
        'UUU': 'F', 'UAA': '*', 'UAG': '*', 'UGA': '*',
    }

    possible = [ ❷
        len([c for c, res in codon_to_aa.items() if res == aa])
        for aa in args.protein + '*'
    ]
    print(product(possible) % args.modulo) ❸
```

❶ A dictionary encoding RNA codons to amino acids.

❷ Iterate through the residues of the protein plus the stop codon, then find the number of codons matching the given amino acid.

❸ Find the products of the possibilities and *mod* by the given value.

Solution 2: Turn the Beat Around

For this next solution, I decided to reverse the keys and values of the RNA codons dictionary so that the amino acids which are unique form the keys and the values are the lists of codons. It's handy to know how to flip a dictionary like this, but it only works if the values are unique. For instance, I can create a lookup table to go from DNA bases like *A* or *T* to their names:

```
>>> base_to_name = dict(A='adenine', G='guanine', C='cytosine',
T='thymine')
>>> base_to_name['A']
'adenine'
```

If I wanted to turn that around so I could go from the name to the base, I can use the `dict.items()` to get the key/value pairs:

```
>>> list(base_to_name.items())
[('A', 'adenine'), ('G', 'guanine'), ('C', 'cytosine'), ('T',
'thymine')]
```

then `map()` those through `reversed()` to flip them, and finally pass the result to the `dict()` function to create a dictionary:

```
>>> dict(map(reversed, base_to_name.items()))
{'adenine': 'A', 'guanine': 'G', 'cytosine': 'C', 'thymine': 'T'}
```

If I try that on the RNA codons table from the first solution, however, I'll get this:

```
>>> pprint(dict(map(reversed, c2aa.items())))
{'*': 'UGA',
 'A': 'GCU',
 'C': 'UGU',
 'D': 'GAU',
 'E': 'GAG',
 'F': 'UUU',
 'G': 'GGU',
 'H': 'CAU',
 'I': 'AUU',
 'K': 'AAG',
 'L': 'UUG',
 'M': 'AUG',
 'N': 'AAU',
 'P': 'CCU',
 'Q': 'CAG',
 'R': 'CGU',
 'S': 'UCU',
 'T': 'ACU',
 'V': 'GUU',
 'W': 'UGG',
 'Y': 'UAU'}
```

You can see that I'm missing most of the codons. Only *M* and *W* actually have one codon. What happened to the rest? When creating the dictionary, Python overwrote any existing values for a key with the newest value. In the original table, for instance, *UUG* was the last value indicated for *L*, so that was the value that was left standing. Just remember this trick for reversing dictionary key/values and ensure that the values are unique

For what it's worth, if I really needed to do this, I would use the `collections.defaultdict()`:

```
>>> from collections import defaultdict
>>> aa2codon = defaultdict(list)
>>> for k, v in c2aa.items():
...     aa2codon[v].append(k)
...
>>> pprint(aa2codon)
defaultdict(<class 'list'>,
            {'*': ['UAA', 'UAG', 'UGA'],
             'A': ['GCA', 'GCC', 'GCG', 'GCU'],
             'C': ['UGC', 'UGU'],
             'D': ['GAC', 'GAU'],
             'E': ['GAA', 'GAG'],
             'F': ['UUC', 'UUU'],
             'G': ['GGA', 'GGC', 'GGG', 'GGU'],
             'H': ['CAC', 'CAU'],
             'I': ['AUA', 'AUC', 'AUU'],
             'K': ['AAA', 'AAG'],
             'L': ['CUA', 'CUC', 'CUG', 'CUU', 'UUA', 'UUG'],
             'M': ['AUG'],
             'N': ['AAC', 'AAU'],
             'P': ['CCA', 'CCC', 'CCG', 'CCU'],
             'Q': ['CAA', 'CAG'],
             'R': ['AGA', 'AGG', 'CGA', 'CGC', 'CGG', 'CGU'],
             'S': ['AGC', 'AGU', 'UCA', 'UCC', 'UCG', 'UCU'],
             'T': ['ACA', 'ACC', 'ACG', 'ACU'],
             'V': ['GUA', 'GUC', 'GUG', 'GUU'],
             'W': ['UGG'],
             'Y': ['UAC', 'UAU']})
```

This is the data structure I used in the following solution:

```
def main():
    args = get_args()
    aa_to_codon = { ❶
        'A': ['GCA', 'GCC', 'GCG', 'GCU'],
        'C': ['UGC', 'UGU'],
        'D': ['GAC', 'GAU'],
        'E': ['GAA', 'GAG'],
        'F': ['UUC', 'UUU']
        'G': ['GGA', 'GGC', 'GGG', 'GGU'],
```

```
        'H': ['CAC', 'CAU'],
        'I': ['AUA', 'AUC', 'AUU'],
        'K': ['AAA', 'AAG'],
        'L': ['CUA', 'CUC', 'CUG', 'CUU', 'UUA', 'UUG'],
        'M': ['AUG'],
        'N': ['AAC', 'AAU'],
        'P': ['CCA', 'CCC', 'CCG', 'CCU'],
        'Q': ['CAA', 'CAG'],
        'R': ['AGA', 'AGG', 'CGA', 'CGC', 'CGG', 'CGU'],
        'S': ['AGC', 'AGU', 'UCA', 'UCC', 'UCG', 'UCU'],
        'T': ['ACA', 'ACC', 'ACG', 'ACU'],
        'V': ['GUA', 'GUC', 'GUG', 'GUU'],
        'W': ['UGG'],
        'Y': ['UAC', 'UAU'],
        '*': ['UAA', 'UAG', 'UGA'],
    }

    possible = [len(aa_to_codon[aa]) for aa in args.protein + '*'] ❷
    print(product(possible) % args.modulo) ❸
```

❶ Represent the dictionary using the residues as the keys and the codons for the values.

❷ Find the number of codons encoding each amino acid in the protein sequence plus the stop codon.

❸ Calculate the product *mod* the modulo value.

Solution 3: Encode the Minimal Information

The previous solution encoded more information than was necessary to find the solution. Since I only need the *number* of codons that encode a given amino acid, not the actual list, I could instead create this lookup table:

```
>>> codons = {
...     'A': 4, 'C': 2, 'D': 2, 'E': 2, 'F': 2, 'G': 4, 'H': 2, 'I':
3,
...     'K': 2, 'L': 6, 'M': 1, 'N': 2, 'P': 4, 'Q': 2, 'R': 6, 'S':
6,
```

```
...     'T': 4, 'V': 4, 'W': 1, 'Y': 2, '*': 3,
... }
```

A list comprehension will return the numbers needed for the product:

```
>>> [codons.get(aa) for aa in 'MA*']
[1, 4, 3]
```

Which can be expressed more succinctly with a `map()`:

```
>>> list(map(codons.get, 'MA*'))
[1, 4, 3]
```

Leading to this code:

```
def main():
    args = get_args()
    codons = { ❶
        'A': 4, 'C': 2, 'D': 2, 'E': 2, 'F': 2, 'G': 4, 'H': 2, 'I':
3,
        'K': 2, 'L': 6, 'M': 1, 'N': 2, 'P': 4, 'Q': 2, 'R': 6, 'S':
6,
        'T': 4, 'V': 4, 'W': 1, 'Y': 2, '*': 3,
    }
    print(product(map(codons.get, args.protein + '*')) % args.modulo)
❷
```

❶ Encode the number of codons for each amino acid.

❷ Find the product of the codon combinations for the input sequence *mod* the modulo value.

Going Further

In a sense, I've reversed the idea of a regular expression match by creating all the possible strings for a match. That is, the 12 patterns that could produce the protein *MA* are as follows:

```
$ ./show_patterns.py MA
    1: AUGGCAUAA
    2: AUGGCAUAG
    3: AUGGCAUGA
    4: AUGGCCUAA
    5: AUGGCCUAG
    6: AUGGCCUGA
    7: AUGGCGUAA
    8: AUGGCGUAG
    9: AUGGCGUGA
   10: AUGGCUUAA
   11: AUGGCUUAG
   12: AUGGCUUGA
```

Essentially, I could try to use this information to possibly create a single unified regular expression. That might not be easy or even possible, but it's an idea that might help us to find a genomic source for a protein. For example, the first two sequences differ by their last base. The alternation between *A* and *G* can be expressed with the character class `[AG]`:

```
  AUGGCAUAA
+ AUGGCAUAG
  ---------
  AUGGCAUA[AG]
```

Could you write a tool that would combine many regular expression patterns into a single one?

Review

- The `itertools.product()` will create the Cartesian product of input iterables.
- `functools.reduce()` is a higher-order function that provides a way to combine progressive pairs of elements from an iterable.

- Python's `%` modulo operator will return the remainder after division.
- Homogeneous lists of numbers and strings can be reduced under monoidal operations like addition, multiplication, and concatenation to a single value
- A dictionary with unique values can be reversed by flipping the keys and values.

Chapter 13. Location Restriction Sites: Using, Testing, and Sharing Code

As described on the Rosalind REVP challenge, most restriction enzyme targets are in the form of *reverse palindromes*, meaning they are sequences equal to their reverse complement. For instance, the reverse complement of the DNA sequence *GCATGC* is the sequence itself:

```
>>> from Bio import Seq
>>> Seq.reverse_complement('GCATGC') == 'GCATGC'
True
```

Typically these restriction enzymes have a length between 4 and 12 nucleotides. The goal in this exercise is to find the locations in a DNA sequence of every putative restriction enzyme. The code to solve this problem could be massively complicated, but a clear understanding of some functional programming techniques helps to create a short, elegant solution. I will explore `map()`, `zip()`, and `enumerate()` as well as many small, tested functions.

You will learn:

- How to find a reverse palindrome
- How to create modules to share common functions
- About the `PYTHONPATH` environment variable

Getting Started

The code and tests for this exercise are in the *13_revp* directory. Start by copying a solution to the program `revp.py`:

```
$ cd 13_revp
$ cp solution1_zip_enumerate.py revp.py
```

Inspect the usage:

```
$ ./revp.py -h
usage: revp.py [-h] FILE

Locating Restriction Sites

positional arguments:
  FILE        Input FASTA file ❶

optional arguments:
  -h, --help  show this help message and exit
```

❶ The only parameter is a single, positional FASTA file.

Have a look at the first test input file. The contents are identical to the example on the Rosalind page:

```
$ cat tests/inputs/1.fa
>Rosalind_24
TCAATGCATGCGGGTCTATATGCAT
```

Run the program with this input and verify you see the expected output as shown on the Rosalind page:

```
$ ./revp.py tests/inputs/1.fa
5 4
7 4
17 4
18 4
21 4
4 6
```

```
6 6
20 6
```

Run the tests to verify the program passes, then start over:

```
$ new.py -fp 'Locating Restriction Sites' revp.py
Done, see new script "revp.py".
```

Here is a way to define the program's parameter:

```
class Args(NamedTuple):
    file: TextIO ❶


# --------------------------------------------------
def get_args() -> Args:
    """ Get command-line arguments """

    parser = argparse.ArgumentParser(
        description='Locating Restriction Sites',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file', ❷
                        help='Input FASTA file',
                        metavar='FILE',
                        type=argparse.FileType('rt'))

    args = parser.parse_args()

    return Args(args.file)
```

❶ The only parameter is a filehandle.

❷ Define a parameter that must be a readable text file.

Have the `main()` print the input filename for now:

```
def main() -> None:
    args = get_args()
    print(args.file.name)
```

Manually verify that the program will produce the correct usage, will reject bogus files, and will print a valid input's name:

```
$ ./revp.py tests/inputs/1.fa
tests/inputs/1.fa
```

Run `pytest` and you should find you pass the first two tests. Now you're ready to write the bones of the program.

Finding All Subsequences Using k-mers

The first step is to read the sequences from the FASTA input file. I know for this exercise that there will be just one record in the file, so it's safe to read the entire file into memory. Normally it would be wiser to treat the records as a lazy stream by relying on the iterator to retrieve data as needed:

```
>>> from Bio import SeqIO
>>> fh = open('tests/inputs/1.fa')
>>> seqs = [str(rec.seq) for rec in SeqIO.parse(fh, 'fasta')]
>>> seqs
['TCAATGCATGCGGGTCTATATGCAT']
```

I need to find all the sequences between 4 and 12 bases long. This sounds like yet another job for k-mers, so I'll once again bring in the `find_kmers()` function from Chapter 9:

```
>>> def find_kmers(seq, k):
...     n = len(seq) - k + 1
...     return [] if n < 1 else [seq[i:i + k] for i in range(n)]
...
```

I can use `range()` to generate all the numbers between 4 and 12, remembering that the end position is not included so I have to go up to 13. As there are many k-mers for each *k*, I'll print the value of *k* and how many k-mers are found:

```
>>> for k in range(4, 13):
...     print(k, len(find_kmers(seq, k)))
...
4 22
5 21
6 20
7 19
8 18
9 17
10 16
11 15
12 14
```

Finding All Reverse Complements

I showed many ways to find the reverse complement in Chapter 3 with the conclusion that `Bio.Seq.reverse_complement()` is probably the easiest method. Start by finding all the 12-mers:

```
>>> kmers = find_kmers(seq, 12)
>>> kmers
['TCAATGCATGCG', 'CAATGCATGCGG', 'AATGCATGCGGG', 'ATGCATGCGGGT',
 'TGCATGCGGGTC', 'GCATGCGGGTCT', 'CATGCGGGTCTA', 'ATGCGGGTCTAT',
 'TGCGGGTCTATA', 'GCGGGTCTATAT', 'CGGGTCTATATG', 'GGGTCTATATGC',
 'GGTCTATATGCA', 'GTCTATATGCAT']
```

To create a list of the reverse complements, you could use a list comprehension:

```
>>> from Bio import Seq
>>> revc = [Seq.reverse_complement(kmer) for kmer in kmers]
```

Or use `map()`:

```
>>> revc = list(map(Seq.reverse_complement, kmers))
```

Either way, you should have 12 reverse complements:

```
>>> revc
['CGCATGCATTGA', 'CCGCATGCATTG', 'CCCGCATGCATT', 'ACCCGCATGCAT',
```

```
'GACCCGCATGCA', 'AGACCCGCATGC', 'TAGACCCGCATG', 'ATAGACCCGCAT',
'TATAGACCCGCA', 'ATATAGACCCGC', 'CATATAGACCCG', 'GCATATAGACCC',
'TGCATATAGACC', 'ATGCATATAGAC']
```

Putting It All Together

You should have just about everything you need to complete this challenge. First, pair all the k-mers with their reverse complements, find those that are the same, and print their positions. You could iterate through them with a `for` loop, or you might consider using the `zip()` function that we first looked at in Chapter 6 to create the pairs. This is an interesting challenge, and I'm sure you can figure out a working solution before you read my versions.

Solutions

I will show three variations to find the restrictions sites which increasingly rely on functions to hide the complexities of the program.

Solution 1: Using the zip and enumerate Functions

In my first solution, I first use `zip()` to pair the k-mers and reverse complements. Assume *k* of 4:

```
>>> seq = 'TCAATGCATGCGGGTCTATATGCAT'
>>> kmers = find_kmers(seq, 4)
>>> revc = list(map(Seq.reverse_complement, kmers))
>>> pairs = list(zip(kmers, revc))
```

I also need to know the positions of the pairs which I can get from `enumerate()`. If I inspect the pairs, I see that some of them (4, 6, 16, 17, and 20) are the same:

```
>>> pprint(list(enumerate(pairs)))
[(0, ("TCAA", "TTGA")),
 (1, ("CAAT", "ATTG")),
 (2, ("AATG", "CATT")),
 (3, ("ATGC", "GCAT")),
 (4, ("TGCA", "TGCA")),
 (5, ("GCAT", "ATGC")),
 (6, ("CATG", "CATG")),
 (7, ("ATGC", "GCAT")),
 (8, ("TGCG", "CGCA")),
 (9, ("GCGG", "CCGC")),
 (10, ("CGGG", "CCCG")),
 (11, ("GGGT", "ACCC")),
 (12, ("GGTC", "GACC")),
 (13, ("GTCT", "AGAC")),
 (14, ("TCTA", "TAGA")),
 (15, ("CTAT", "ATAG")),
 (16, ("TATA", "TATA")),
 (17, ("ATAT", "ATAT")),
 (18, ("TATG", "CATA")),
 (19, ("ATGC", "GCAT")),
 (20, ("TGCA", "TGCA")),
 (21, ("GCAT", "ATGC"))]
```

I can use a list comprehension with a guard to find all the positions where the pairs are the same:

```
>>> [pos + 1 for pos, pair in enumerate(pairs) if pair[0] == pair[1]]
[5, 7, 17, 18, 21]
```

In Chapter 11, I introduced the functions `fst()` and `snd()` for getting the first or second elements from a 2-tuple. I'd like to use those here so I don't have to use indexing into the tuples. I also keep using the `find_kmers()` function from previous chapters. It seems like it's time to put these functions into a separate module so I can import them as needed rather than copying.

If you inspect the `common.py` module, you'll see these functions and their tests. I can run `pytest` to ensure they all pass:

```
$ pytest -v common.py
============================= test session starts
==============================
...

common.py::test_fst PASSED
[ 33%]
common.py::test_snd PASSED
[ 66%]
common.py::test_find_kmers PASSED
[100%]

============================== 3 passed in 0.01s
==============================
```

Because the `common.py` is in the current directory, I can import any functions I like from it:

```
>>> from common import fst, snd
>>> [pos + 1 for pos, pair in enumerate(pairs) if fst(pair) ==
snd(pair)]
[5, 7, 17, 18, 21]
```

PYTHONPATH

You can also place modules of reusable code into a directory that is shared across all your projects. You can use the `PYTHONPATH` environmental variable to indicate the location of additional directories where Python should look for modules. According to the `PYPATH` documentation, it will:

> *Augment the default search path for module files. The format is the same as the shell's PATH: one or more directory pathnames separated by os.pathsep (e.g. colons on Unix or semicolons on Windows). Non-existent directories are silently ignored.*

In the appendix on `$PATH`, I recommend that you install binaries and scripts to a location like `$HOME/.local/bin` and use something like `$HOME/.bashrc` to set your `PATH` to include this directory. I would likewise suggest you define a location for sharing common Python functions and modules and setting your `PYTHONPATH` to include this location.

Here is how I incorporated these ideas in the first solution:

```
def main() -> None:
    args = get_args()
    if seqs := [str(rec.seq) for rec in SeqIO.parse(args.file,
'fasta')]: ❶
        seq = seqs[0] ❷

        for k in range(4, 13): ❸
            kmers = find_kmers(seq, k) ❹
            revc = list(map(Seq.reverse_complement, kmers)) ❺

            for pos, pair in enumerate(zip(kmers, revc)): ❻
                if fst(pair) == snd(pair): ❼
                    print(pos + 1, k) ❽

    else:
        sys.exit(f'"{args.file.name}" contains no sequences.') ❾
```

❶ Use `:=` to assign the list comprehension to `seqs` and then have the `if` evaluate if the result is truthy. This ensures I have at least one sequence.

❷ Grab the first sequence.

❸ Iterate through all the values of *k*.

❹ Find the k-mers for this *k*.

❺ Find the reverse complements of the k-mers.

❻ Iterate through the positions and pairs of k-mer/revcomp pairs.

❼ Check if the first element of the pair is the same as the second element.

❽ Print the position plus 1 (to correct for 0-based indexing) and the size of the sequence *k*.

❾ It's not technically necessary to let the user know that there are no sequences, but this is a nice thing to do in case you happen to parse an empty file.

Solution 2: Using the operator.eq Function

Though I like the `fst()` and `snd()` functions and want to highlight how to share modules and functions, I'm duplicating the `operator.eq()` function. I first introduced this module in Chapter 6 to use the `operator.ne()` (not equal) function, and I've also used the `operator.le()` (less than or equal) and `operator.add()` functions elsewhere.

I can rewrite part of the preceding solution like so:

```
for pos, pair in enumerate(zip(kmers, revc)):
    if operator.eq(*pair): ❶
```

```
            print(pos + 1, k)
```

❶ Use the functional version of the == operator to compare the elements of the pair. Note the need to splat the pair to expand the tuple into its two values.

I prefer a list comprehension with a guard to condense this code:

```
def main() -> None:
    args = get_args()

    if seqs := [str(rec.seq) for rec in SeqIO.parse(args.file,
'fasta')]:
        seq = seqs[0]

        for k in range(4, 13):
            kmers = find_kmers(seq, k)
            revc = map(Seq.reverse_complement, kmers)
            pairs = enumerate(zip(kmers, revc))

            for pos in [pos + 1 for pos, pair in pairs if
operator.eq(*pair)]: ❶
                print(pos, k)

    else:
        sys.exit(f'"{args.file.name}" contains no sequences.')
```

❶ Use a guard for the equality comparison, and correct the position inside a list comprehension.

Solution 3: Writing a revp Function

In this final solution, it behooves me to write a `revp()` function and create a test. This will make the program more readable and will also make it easier to move this function into something like the `common.py` module for sharing in other projects.

As usual, I imagine the signature of my function:

```
def revp(seq: str, k: int) -> List[int]: ❶
    """ Return positions of reverse palindromes """

    return [] ❷
```

❶ I want to pass in a sequence and a size for *k* and get back a list of locations where reverse palindromes of the given size are found.

❷ For now return the empty list.

Here is the test I wrote. Note that I decided that the function should correct the indexes to 1-based counting:

```
def test_revp() -> None:
    """ Test revp """

    assert revp('CGCATGCATTGA', 4) == [3, 5]
    assert revp('CGCATGCATTGA', 5) == []
    assert revp('CGCATGCATTGA', 6) == [2, 4]
    assert revp('CGCATGCATTGA', 7) == []
    assert revp('CCCGCATGCATT', 4) == [5, 7]
    assert revp('CCCGCATGCATT', 5) == []
    assert revp('CCCGCATGCATT', 6) == [4, 6]
```

If I add these to my `revp.py` program and run `pytest revp.py`, I would see that the test fails as it should. Now I can fill in the code:

```
def revp(seq: str, k: int) -> List[int]:
    """ Return positions of reverse palindromes """

    kmers = find_kmers(seq, k)
    revc = map(Seq.reverse_complement, kmers)
    pairs = enumerate(zip(kmers, revc))
    return [pos + 1 for pos, pair in pairs if operator.eq(*pair)]
```

If I run `pytest` again, I should get a passing test. The `main()` becomes more readable:

```
def main() -> None:
    args = get_args()

    if seqs := [str(rec.seq) for rec in SeqIO.parse(args.file,
'fasta')]:
        seq = seqs[0]
        for k in range(4, 13): ❶
            for pos in revp(seq, k): ❷
                print(pos, k) ❸
    else:
        sys.exit(f'"{args.file.name}" contains no sequences.')
```

❶ Iterate through each value of *k*.

❷ Iterate through each reverse palindrome of size *k* found in the sequence.

❸ Print the position and size of the reverse palindrome.

Note that it's possible to use more than one iterator inside a list comprehension. I can collapse the two `for` loops into a single one like so:

```
for k, pos in [(k, pos) for k in range(4, 13) for pos in revp(seq,
k)]: ❶
    print(pos, k)
```

❶ First iterate the *k* values, then use those to iterate the `revp()` values, returning both as a tuple.

I would probably not use this construct. It reminds me of my old coworker, Joe, who would joke: *If it was hard to write, it should be hard to read!*

Testing the Program

I'd like to take a moment to look at the integration test in *tests/revp_test.py*. The first two tests are always the same, checking for the existence of the expected program and that the program will produce some *usage* statement when requested. For a program that accept files as inputs such as this one, I include a test that the program rejects an invalid file. I usually challenge other inputs like passing strings when integers are expected to ensure the arguments are rejected.

After I've checked that the arguments to the program are all validated, I start passing good input values to see that the program works as expected. This requires that I use valid, known input and verify that the program produces the correct, expected output. In this case, I encode the inputs and outputs using files in the *tests/inputs* directory. For instance, the expected output for the input file *1.fa* is found in *1.fa.out*:

```
$ ls tests/inputs/
1.fa      1.fa.out  2.fa      2.fa.out
```

The following is the first input:

```
$ cat tests/inputs/1.fa
>Rosalind_24
TCAATGCATGCGGGTCTATATGCAT
```

and the expected output:

```
$ cat tests/inputs/1.fa.out
5 4
7 4
17 4
18 4
21 4
4 6
6 6
20 6
```

The second input file is significantly larger than the first. This is common with the Rosalind problems, and so it would be ugly to try to include the input and output values as literal strings in the test program. The expected output for the second file is 70 lines long.

In *tests/revp_test.py*, I wrote a `run()` helper function that takes the name of the input file, reads the expected output filename, and runs the program with the input to check the output:

```
def run(file: str) -> None: ❶
    """ Run the test """

    expected_file = file + '.out' ❷
    assert os.path.isfile(expected_file) ❸

    rv, out = getstatusoutput(f'{PRG} {file}') ❹
    assert rv == 0 ❺

    expected = set(open(expected_file).read().splitlines()) ❻
    assert set(out.splitlines()) == expected ❼
```

❶ The function takes the name of the input file.

❷ The output file is the name of the input file plus *.out*.

❸ Make sure the output file exists.

❹ Run the program with the input file, get the return value and `STDOUT`.

❺ Make sure the program reported a successful run.

❻ Read the expected output file, breaking the contents on lines and creating a set of the resulting strings

❼ Break the output of the program on lines and create a set to compare to the expected results. Sets allow me to disregard the order of the lines.

This simplifies the tests. Note that the `INPUT*` variables are declared at the top of the test:

```
def test_ok1() -> None:
    run(INPUT1)


def test_ok2() -> None:
    run(INPUT2)
```

I would encourage you to spend some time reading the *_*test.py* files for every program. I hope that you will integrate testing into your development workflow, and I'm sure you can find ample code to copy from my tests which will save you time.

Going Further

- Write a program that can identify English palindromes such as "A man, a plan, a canal — Panama!" Start by creating a new repository. Find several interesting palindromes to use in your tests. Be sure to provide phrases that are not palindromes and verify that your algorithm rejects those, too. Release your code to the internet and reap the fame, glory, and profit from writing open-source software.

Review

- You can reuse functions by placing them into a module and importing them as needed.
- The `PYTHONPATH` environment variable indicates directories which Python should add when looking for modules of code.

Chapter 14. Finding Open Reading Frames

This is the final Rosalind challenge I'll tackle in this book. The goal is to find all the possible open reading frames (ORFs) in a sequence of DNA, considering both the forward and reverse complement and frame shifts. I like this problem as it brings together so many of the previous challenges from reading a FASTA file, taking a reverse complement, using string slices, finding k-mers, using multiple for loops/iterations, DNA translation, and regular expressions.

You will learn:

- How to truncate a sequence to a length evenly divisible by a codon size
- How to use the `str.find()` and `str.partition()` functions
- How to document a regular expression using code formatting, comments, and Python's implicit string concatenation

Getting Started

The code, tests, and solutions for this challenge are located in the *14_orf* directory. Start by copying the first solution to the program `orf.py`:

```
$ cd 14_orf/
$ cp solution1_iterate_set.py orf.py
```

If you request the usage, you'll see the program takes a single positional argument of a FASTA-formatted file of sequences:

```
$ ./orf.py -h
usage: orf.py [-h] FILE

Open Reading Frames

positional arguments:
  FILE        Input FASTA file

optional arguments:
  -h, --help  show this help message and exit
```

The first test input file has the same content as the example on the Rosalind page. Note that I've broken the sequence file here, but it's a single line in the input file:

```
$ cat tests/inputs/1.fa
>Rosalind_99
AGCCATGTAGCTAACTCAGGTTACATGGGGATGACCCCGCGACTTGGATTAGAGTCTCTTTTGGAATAAG
\
CCTGAATGATCCGAGTAGCATCTCAG
```

Run the program with this input file and note the output. The order of the ORFs is not important:

```
$ ./orf.py tests/inputs/1.fa
M
MGMTPRLGLESLLE
MLLGSFRLIPKETLIQVAGSSPCNLS
MTPRLGLESLLE
```

Run the test suite to ensure the program passes the tests. When you are satisfied with how your program should work, start over:

```
$ new.py -fp 'Open Reading Frames' orf.py
Done, see new script "orf.py".
```

At this point you probably need no help in defining a single positional file argument, but here is the code you can use:

```
class Args(NamedTuple):
    file: TextIO


# --------------------------------------------------
def get_args() -> Args:
    """ Get command-line arguments """

    parser = argparse.ArgumentParser(
        description='Open Reading Frames',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file', ❶
                        help='Input FASTA file',
                        metavar='FILE',
                        type=argparse.FileType('rt'))

    args = parser.parse_args()

    return Args(args.file)
```

❶ Define a positional argument that must be a readable text file.

Modify the `main()` to print the incoming filename:

```
def main() -> None:
    args = get_args()
    print(args.file.name)
```

Verify the program prints the usage, rejects bad files, and prints the filename for a valid argument:

```
$ ./orf.py tests/inputs/1.fa
tests/inputs/1.fa
```

At this point, your program should pass the first three tests. Next I'll talk about how to make the program find ORFs.

Translating Proteins Inside Each Frame Shift

It might be helpful to write a bit of pseudocode to help sketch out what needs to happen:

```
def main() -> None:
    args = get_args()

    # If there are DNA sequences in the file:
        # Transcribe the first sequence from DNA to RNA
        # Iterate using the forward and reverse complement of the RNA:
            # Iterate through 0, 1, 2 for frame shifts in this
sequence:
                # Translate the frame-shifted RNA into a protein
sequence
                # Try to find the ORFs in this protein sequence
    # Else
        # Let the user know the file has no sequences
```

I can steal code from Chapter 13 for reading the sequences using `Bio.SeqIO`:

```
def main() -> None:
    args = get_args()

    if seqs := [str(rec.seq) for rec in SeqIO.parse(args.file,
'fasta')]: ❶
        seq = seqs[0] ❷
        print(seq)
    else:
        sys.exit(f'"{args.file.name}" contains no sequences.') ❸
```

❶ Use a list comprehension to read all the sequences from the file. If none are returned, the `if` will evaluate an empty list which will be falsey.

❷ There should be just one sequence in the file, so grab the first one.

❸ The `sys.exit()` function will print the message to `STDERR` and then will exit the program with a non-zero value to indicate an

error.

I can run the program to verify this works:

```
$ ./orf.py tests/inputs/1.fa
AGCCATGTAGCTAACTCAGGTTACATGGGGATGACCCCGCGACTTGGATTAGAGTCTCTTTTGGA\
ATAAGCCTGAATGATCCGAGTAGCATCTCAG
```

There is an *empty.fa* file I can use to ensure the program prints an error:

```
$ ./solution1_iterate_set.py tests/inputs/empty.fa
"tests/inputs/empty.fa" contains no sequences.
```

I need to transcribe this to RNA, which entails changing all the Ts to Us. I'll let you use whatever solution from Chapter 2 you like so long as your program can now print this:

```
$ ./orf.py tests/inputs/1.fa
AGCCAUGUAGCUAACUCAGGUUACAUGGGGAUGACCCCGCGACUUGGAUUAGAGUCUCUUUUGGA\
AUAAGCCUGAAUGAUCCGAGUAGCAUCUCAG
```

Next, refer to Chapter 3 and have your program print both the forward and reverse complements of this sequence:

```
$ ./orf.py tests/inputs/1.fa
AGCCAUGUAGCUAACUCAGGUUACAUGGGGAUGACCCCGCGACUUGGAUUAGAGUCUCUUUUGGA\
AUAAGCCUGAAUGAUCCGAGUAGCAUCUCAG
CUGAGAUGCUACUCGGAUCAUUCAGGCUUAUUCCAAAAGAGACUCUAAUCCAAGUCGCGGGGUCA\
UCCCCAUGUAACCUGAGUUAGCUACAUGGCU
```

To implement frame shifts, you can read these sequences starting from the 0th, 1st, and 2nd characters using a string slice. Refer to Chapter 7 to find code that will translate this slice of RNA into a protein sequence. If you use Biopython to translate the RNA slice, you may encounter the warning *Partial codon, len(sequence) not a multiple of three. Explicitly trim the sequence or add trailing N before*

translation. This may become an error in the future. It may help to create a function to truncate a sequence to the nearest even division by a value:

```
def truncate(seq: str, k: int) -> str:
    """ Truncate a sequence to even division by k """

    return ''
```

Figure 14-1 shows the results of shifting through the string *0123456789* and truncating each result to a length that is evenly divisible by 3.

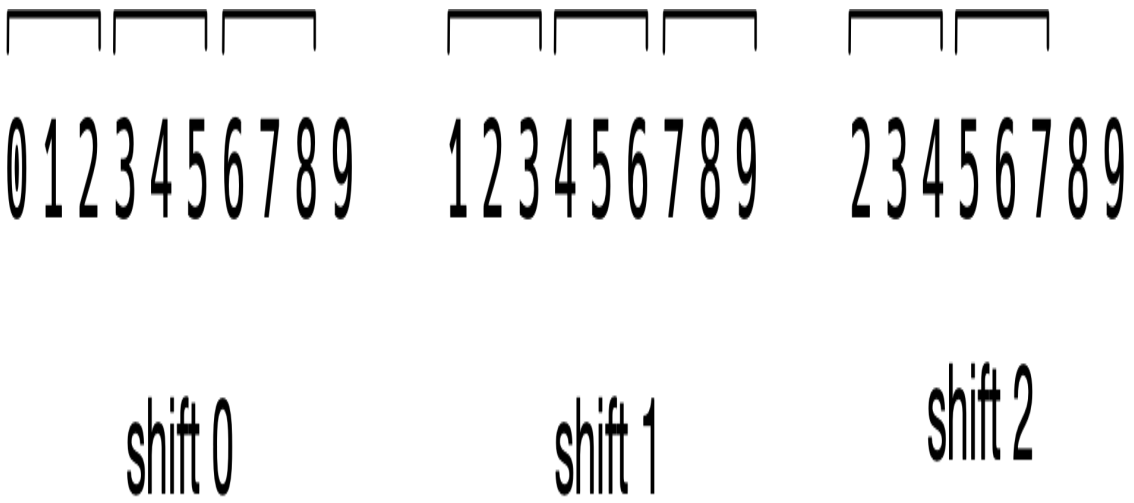


Figure 14-1. Truncating the various frame shifts to a length that is evenly divisible by the codon size 3.

Here is a test you could use:

```
def test_truncate() -> None:
    """ Test truncate """

    seq = '0123456789'
    assert truncate(seq, 3) == '012345678'
    assert truncate(seq[1:], 3) == '123456789'
    assert truncate(seq[2:], 3) == '234567'
```

Your program should be able to now print the protein translations for the three frame shifts for both the forward and reverse complements of the RNA like so:

```
$ ./orf.py tests/inputs/1.fa
SHVANSGYMGMTPRLGLESLLE*A*MIRVASQ
AM*LTQVTWG*PRDLD*SLFWNKPE*SE*HL
PCS*LRLHGDDPATWIRVSFGISLNDPSSIS
LRCYSDHSGLFQKRL*SKSRGHPHVT*VSYMA
*DATRIIQAYSKRDSNPSRGVIPM*PELATW
EMLLGSFRLIPKETLIQVAGSSPCNLS*LHG
```

Finding the ORFs in a Protein Sequence

Now that the program can find all the protein sequences from each frame shift of the RNA, it's time to look for the open reading frames in the proteins. Your code will need to consider every interval from each start codon *M* to the first subsequent stop codon. For example, Figure 14-2 shows that the amino acid sequence *MAMAPR** contains two start codons and one stop codon and so has two possible proteins of *MAMAPR* and *MAPR*.



Figure 14-2. The protein sequence MAMAPR has two overlapping open reading frames.*

I decided to write a function called `find_orfs()` that will accept an amino acid string and return a list of ORFs:

```
def find_orfs(aa: str) -> List[str]: ❶
    """ Find ORFs in AA sequence """
```

```
    return [] ❷
```

❶ The function accepts a string of amino acids and returns a list of possible protein strings.

❷ For now, return the empty list.

Here is a test for this function. If you can implement the `find_orfs()` that passes this test, then you should be able to pass the integration test:

```
def test_find_orfs() -> None:
    """ Test find_orfs """

    assert find_orfs('') == [] ❶
    assert find_orfs('M') == [] ❷
    assert find_orfs('*') == [] ❸
    assert find_orfs('M*') == ['M'] ❹
    assert find_orfs('MAMAPR*') == ['MAMAPR', 'MAPR'] ❺
    assert find_orfs('MAMAPR*M') == ['MAMAPR', 'MAPR'] ❻
    assert find_orfs('MAMAPR*MP*') == ['MAMAPR', 'MAPR', 'MP'] ❼
```

❶ The empty string should produce no ORFs.

❷ A single start codon with no stop codon should produce no ORFs.

❸ A single stop codon with no preceding start codon should produce no ORFs.

❹ The function should return the start codon even if there are no intervening bases before the stop codon.

❺ This sequence contains two ORFs.

❻ This sequence also contains only two ORFs.

❼ This sequence contains three ORFs in two separate sections.

Once you can find all the ORFs in each RNA sequence, you should collect them into a distinct list. I suggest you use a `set()` for this. Though my solution prints the ORFs in sorted order, this is not a requirement for the test.

Your solution will bring together many of the skills you've already learned. The skill of writing longer programs lies in composing smaller pieces that you understand and test. Keep plugging away at your program until you pass all the tests.

Solutions

I'll present three solutions to finding ORFs using two string functions and regular expressions.

Solution 1: Using the str.index Function

To start, here is how I wrote the `truncate()` function that will ameliorate the `Bio.Seq.translate()` function when I try to translate the various frame-shifted RNA sequences:

```
def truncate(seq: str, k: int) -> str:
    """ Truncate a sequence to even division by k """

    length = len(seq) ❶
    end = length - (length % k) ❷
    return seq[:end] ❸
```

❶ Find the length of the sequence.

❷ The end of the desired subsequence is the length minus the length modulo `k`.

❸ Return the subsequence.

Next, here is one way to write the `find_orfs()` that uses the `str.index()` function to find each starting *M* codon followed by a `*` stop codon:

```
def find_orfs(aa: str) -> List[str]:
    orfs = [] ❶
    while 'M' in aa: ❷
        start = aa.index('M') ❸
        if '*' in aa[start + 1:]: ❹
            stop = aa.index('*', start + 1) ❺
            orfs.append(''.join(aa[start:stop])) ❻
            aa = aa[start + 1:] ❼
        else:
            break ❽

    return orfs
```

❶ Initialize a list to hold the ORFs.

❷ Create a loop to iterate while there are start codons present.

❸ Use `str.index()` to find the location of the start codon.

❹ See if the stop codon is present after the start codon's position.

❺ Get the index of the stop codon after the start codon.

❻ Use a string slice to grab the protein.

❼ Set the amino acid string to the index after the position of the start codon to find the next start codon.

❽ Leave the `while` loop if there is no stop codon.

Here is how I incorporate these ideas into the program:

```
def main() -> None:
    args = get_args()
```

```
    if seqs := [str(rec.seq) for rec in SeqIO.parse(args.file,
'fasta')]: ❶
        rna = seqs[0].replace('T', 'U') ❷
        orfs = set() ❸

        for seq in [rna, Seq.reverse_complement(rna)]: ❹
            for i in range(3): ❺
                if prot := Seq.translate(truncate(seq[i:], 3),
to_stop=False): ❻
                    for orf in find_orfs(prot): ❼
                        orfs.add(orf) ❽

        print('\n'.join(sorted(orfs))) ❾
    else:
        sys.exit(f'"{args.file.name}" contains no sequences.') ❿
```

❶ Try to get sequences from the input file.

❷ Assume a single DNA sequence and transcribe it to RNA.

❸ Create an empty set to hold all the ORFs.

❹ Iterate through the forward and reverse complement of the RNA.

❺ Iterate through the frame shifts.

❻ Attempt to translate the truncated, frame-shifted RNA into a protein sequence.

❼ Iterate through each ORF found in the protein sequence.

❽ Add the ORF to the set to maintain a unique list.

❾ Print the sorted ORFs.

❿ Inform the user that the input file contains no sequences.

Solution 2: Using the str.partition Function

Here is another approach to writing the `find_orfs()` function that uses `str.partition()`. This function breaks a string into the part before some substring, the substring, and the part after. For instance, the string *MAMAPR*MP** can be partitioned on the stop codon (*):

```
>>> 'MAMAPR*MP*'.partition('*')
('MAMAPR', '*', 'MP*')
```

If the protein sequence does not contain a stop codon, the function returns empty strings for the 2nd and 3rd return values:

```
>>> 'M'.partition('*')
('M', '', '')
```

In this version, I use two infinite loops. The first tries to partition the given amino acid sequence on the stop codon. If this is not successful, then I break out of the first loop. Figure 14-3 shows that the protein sequence *MAMAPR*MP** contains two sections that have start and end codons:

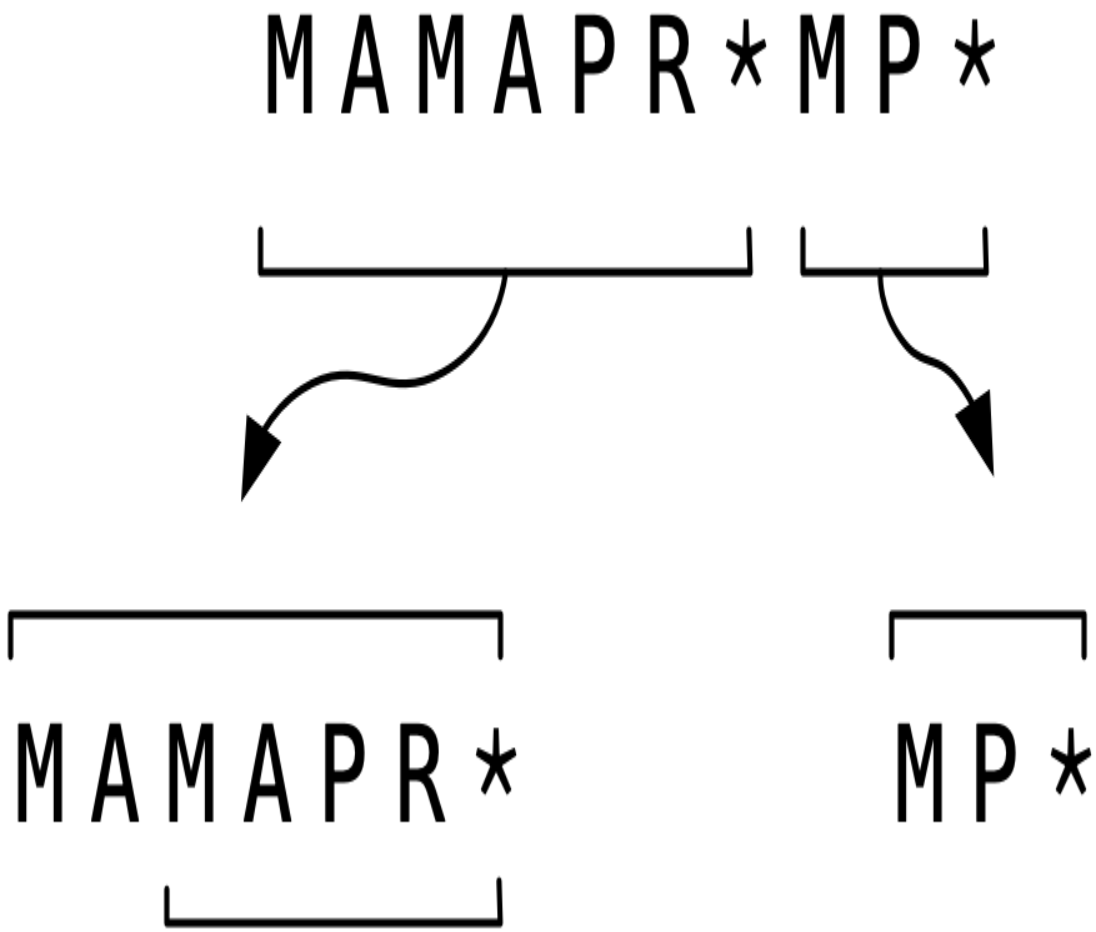


*Figure 14-3. The protein sequence MAMAPR*MP* has three ORFs in two sections.*

The second loop checks the first partition to find all the subsequences starting with the *M* start codon. So in the partition *MAMAPR*, it finds the two sequences *MAMAPR* and *MAPR*. The code then truncates the AA sequence to the last partition, *MP**, to repeat the operation until all ORFs have been found:

```
def find_orfs(aa: str) -> List[str]:
    """ Find ORFs in AA sequence """

    orfs = [] ❶
    while True: ❷
        first, middle, rest = aa.partition('*') ❸
        if middle == '': ❹
            break
```

```
        last = 0 ❺
        while True: ❻
            start = first.find('M', last) ❼
            if start == -1: ❽
                break
            orfs.append(first[start:]) ❾
            last = start + 1 ❿
        aa = rest ⓫

    return orfs ⓬
```

❶ Initialize a list for the ORFs to return.

❷ Create the first infinite loop.

❸ Partition the amino acid sequence on the stop codon.

❹ The middle will be empty if the stop codon is not present, so break from the outer loop.

❺ Set up a temporary variable to remember the last position of a start codon.

❻ Create a second infinite loop.

❼ Use the `str.find()` method to locate the index of the start codon.

❽ The value -1 indicates that the start codon is not present, so leave the inner loop.

❾ Add the substring from the start index to the list of ORFs.

❿ Move the last known position to after the current start position.

⓫ Truncate the protein sequence to the last part of the initial partition.

Solution 3: Using a Regular Expression

In this final solution, I'll once again point out that a regular expression is probably the most fitting solution to find a pattern of text. This pattern always starts with *M*, and I can use the `re.findall()` function to find the four Ms in this protein sequence:

```
>>> import re
>>> re.findall('M', 'MAMAPR*MP*M')
['M', 'M', 'M', 'M']
```

An ORF starts with an *M* and extends to the first stop codon. In between these, there can be zero or more not-stop codons which I can represent using a negated character class of `[^*]` that excludes the stop codon followed by a `*` to indicate that there can be *zero or more* of the preceding pattern:

```
>>> re.findall('M[^*]*', 'MAMAPR*MP*M')
['MAMAPR', 'MP', 'M']
```

I need to add the stop codon `*` to this pattern. The literal asterisk is a metacharacter, so I can either use a backslash to escape it:

```
>>> re.findall('M[^*]*\*', 'MAMAPR*MP*M')
['MAMAPR*', 'MP*']
```

or place it inside a character class where it has no meta meaning:

```
>>> re.findall('M[^*]*[*]', 'MAMAPR*MP*M')
['MAMAPR*', 'MP*']
```

I can represent this pattern using a finite state machine diagram, as shown in Figure 14-4.

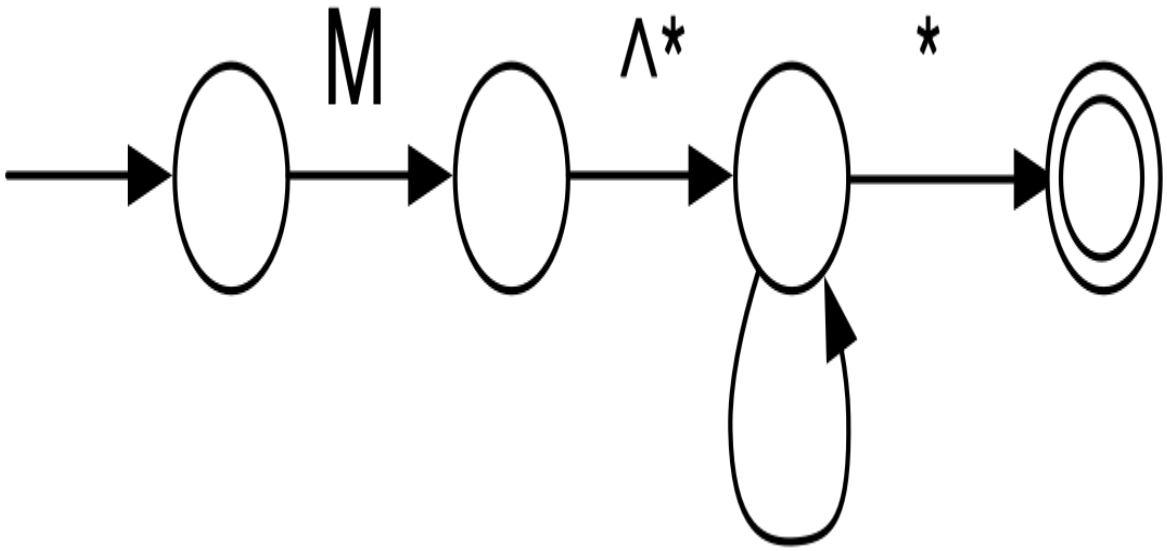


Figure 14-4. A finite state machine diagram of the regular expression to find an open reading frame.

I can see that this pattern is close to working, but it's only finding two of the three ORFs because the first one overlaps the second one. As in Chapters 8 and 11, I can wrap the pattern in a positive look-ahead assertion. Further, I will use parentheses to create a capture group around the ORF up to the stop codon:

```
>>> re.findall('(?=(M[^*]*)[*])', 'MAMAPR*MP*M')
['MAMAPR', 'MAPR', 'MP']
```

Here is one version of the `find_orfs()` that uses this pattern:

```
def find_orfs(aa: str) -> List[str]:
    """ Find ORFs in AA sequence """

    return re.findall('(?=(M[^*]*)[*])', aa)
```

While this passes `test_find_orfs()`, this is a complicated regex that I will have to relearn every time I come back to it. An alternate way to write this is to place each functional piece of the regex on a separate line followed by an end-of-line comment and rely on Python's implicit

string concatenation (first shown in Chapter 2) to join these into a single string. This is my preferred method to find the ORFs:

```
def find_orfs(aa: str) -> List[str]:
    """ Find ORFs in AA sequence """

    pattern = ( ❶
        '(?='    # start positive look-ahead to handle overlaps
        '('      # start a capture group
        'M'      # a literal M
        '[^*]*'  # zero or more of anything not the asterisk
        ')'      # end the capture group
        '[*]'    # a literal asterisk
        ')')     # end the look-ahead group

    return re.findall(pattern, aa) ❷
```

❶ The parentheses group the following lines such that Python will automatically join the strings into a single string. Be sure there are no commas or Python will create a tuple.

❷ Use the pattern with the `re.findall()` function.

This is a longer function, but it will be much easier to understand the next time I see it. One downside is that the `yapf` program I use to format my code will remove the vertical alignment of the comments, so I must manually format this section. Still, I think it's worth it to have more self-documenting code.

Going Further

Expand the program to process multiple input files, writing all the unique ORFs to an indicated output file.

Review

- The `Bio.Seq.translate()` function will print warnings if the input sequence is not evenly divisible by 3, so I wrote a `truncate()` function to trim the protein.
- The `str.find()` and `str.partition()` functions each present ways to find subsequences in a string.
- A regular expression remains my preferred method to find a pattern in some text.
- A complicated regex can be written over multiple lines with comments so that Python will implicitly concatenate them into a single string.

Part II. Other Programs

In the chapters in this part, I'll show you several programs I've written that capture patterns I use repeatedly.

Chapter 15. Seqmagique: Creating and Formatting Reports

Often in bioinformatics projects, you'll find yourself staring at a directory full of sequence files, probably in FASTA or FASTQ format. You'll probably want to start by getting an idea of the distribution of sequences in the files such as how many are in each file and the average, minimum, and maximum lengths of the sequences. You need to know if any files are corrupted — maybe they didn't transfer completely from your sequencing center — or if any samples have far fewer reads perhaps indicating a bad sequencing run that needs to be redone. In this chapter, I'd like to introduce techniques for checking your sequence files using hashes and the Seqmagick tool. Then I'll write a small utility to mimic part of Seqmagick to illustrate how to create formatted text tables. This program serves as a template for any program that needs to process all the records in a given set of files and produce a table of summary statistics.

You will learn:

- How to install the `seqmagick` tool
- How to use MD5 hashes
- How to use the `numpy` module
- How to mock a filehandle
- How to use the `tabulate` and `rich` modules to format output tables

Using Seqmagick to Analyze Sequence Files

The Seqmagick tool is a useful command-line utility for handling sequence files. Normally you could install this module with `pip` like so:

```
$ python3 -m pip install seqmagick
```

As of writing this book, the installation can be a bit tricky due to changes in the Biopython libraries. I've included directions in the *README.md* file in the repository detailing how to install a working version. In due time, `pip` will probably be able to install it as with other modules.

If you run **`seqmagick --help`**, you'll see the tool offers many options. I only want to focus on the `info` subcommand. I can run this on the test input FASTA files in the *15_seqmagique* directory like so:

```
$ cd 15_seqmagique
$ seqmagick info tests/inputs/*.fa
name                  alignment    min_len   max_len   avg_len
num_seqs
tests/inputs/1.fa     FALSE             50        50     50.00
1
tests/inputs/2.fa     FALSE             49        79     64.00
5
tests/inputs/empty.fa FALSE              0         0      0.00
0
```

In this exercise, you will create a program called `seqmagique.py` that will mimic this output. The point of the program is to provide a basic overview of the sequences in a given set of files so you can spot, for instance, a truncated or corrupted file. It really, really sucks to queue up an HPC job that runs over the weekend only to come back days later and realize that your whole analysis must be re-run because one or more of the input files was incomplete.

Copy the solution to `seqmagique.py` and request the usage:

```
$ cp solution1.py seqmagique.py
$ ./seqmagique.py -h
usage: seqmagique.py [-h] [-t table] FILE [FILE ...]

Argparse Python script

positional arguments:
  FILE                  Input FASTA file(s) ❶

optional arguments:
  -h, --help            show this help message and exit
  -t table, --tablefmt table ❷
                        Tabulate table style (default: plain)
```

❶ The program accepts one or more input files which should be in FASTA format.

❷ This option controls the format of the output table.

Run this program on the same files and note the output is almost identical except that I have omitted the *alignment* column:

```
$ ./seqmagique.py tests/inputs/*.fa
name                     min_len    max_len    avg_len    num_seqs
tests/inputs/1.fa             50         50      50.00           1
tests/inputs/2.fa             49         79      64.00           5
tests/inputs/empty.fa          0          0       0.00           0
```

The `--tablefmt` option controls how the output table is formatted. This is the first program you'll write that constrains the value to a given list. To see this in action, use a bogus value like *foo*:

```
$ ./seqmagique.py -t foo tests/inputs/1.fa
usage: seqmagique.py [-h] [-t table] FILE [FILE ...]
seqmagique.py: error: argument -t/--tablefmt: invalid choice: 'foo'
(choose from 'plain', 'simple', 'grid', 'pipe', 'orgtbl', 'rst',
 'mediawiki', 'latex', 'latex_raw', 'latex_booktabs')
```

Try a different table format such as *simple*:

```
$ ./seqmagique.py -t simple tests/inputs/*.fa
name                    min_len    max_len    avg_len    num_seqs
--------------------  ---------  ---------  ---------  ----------
tests/inputs/1.fa            50         50      50.00           1
tests/inputs/2.fa            49         79      64.00           5
tests/inputs/empty.fa         0          0       0.00           0
```

Run the program with other table styles and then try the test suite. Next, I'll talk about getting data for our program to analyze.

Check Files Using MD5 Hashes

The first step in most genomics projects will be transferring sequence files to some location where you can analyze them, and the first line of defense against data corruption is ensuring the files were copied completely. The source of the files may be a sequencing center or a public repository like GenBank or the Sequence Read Archive (SRA). The files may arrive on a thumb drive, or you may download them from the internet. If the latter, you may find that your connection drops, causing some files to be truncated or corrupted. How could you find these types of errors?

One way to check that your files are complete is to compare the file sizes locally with those on the server. For instance, you can use the `ls -l` command to view the *long* listing of files where the file size in bytes is shown. For large sequence files, this is going to be a very large number, and you will have to manually compare the file sizes from the source to the destination which is tedious and prone to error.

Another technique involves using a *hash* or *message digest* of the file, which is a signature of the file's contents generated by a one-way cryptographic algorithm that creates a unique output for every possible input. Although there are many tools you can use to create a hash, I'll focus on tools that use the MD5 algorithm. This algorithm was originally developed in the context of cryptography and security,

but researchers have since identified numerous flaws that now make it suitable only for purposes such as verifying data integrity.

On macOS, I can use the `md5` program to generate a 128-bit hash value from the contents of the first test input file like so:

```
$ md5 -r tests/inputs/1.fa
c383c386a44d83c37ae287f0aa5ae11d tests/inputs/1.fa
```

I can also use the `openssl` program:

```
$ openssl md5 tests/inputs/1.fa
MD5(tests/inputs/1.fa)= c383c386a44d83c37ae287f0aa5ae11d
```

On Linux, the program is called `md5sum`:

```
$ md5sum tests/inputs/1.fa
c383c386a44d83c37ae287f0aa5ae11d  tests/inputs/1.fa
```

If I change even one bit of the input file, a different hash value will be generated. Conversely, if I find another file that generates the same hash value, then the contents of the two files are identical. For instance, the *empty.fa* file is a zero-length file I created for testing, and it has the following hash value:

```
$ md5 -r tests/inputs/empty.fa
d41d8cd98f00b204e9800998ecf8427e tests/inputs/empty.fa
```

If I use the **`touch foo`** command to create another empty file, I'll find it has the same signature:

```
$ touch foo
$ md5 -r foo
d41d8cd98f00b204e9800998ecf8427e foo
```

It's common for data providers to create a file of the checksums so that you can verify that your copies of the data are complete. I

created a *tests/inputs/checksums.md5* like so:

```
$ cd tests/inputs
$ md5 -r *.fa > checksums.md5
```

It has the following contents:

```
$ cat checksums.md5
c383c386a44d83c37ae287f0aa5ae11d 1.fa
863ebc53e28fdfe6689278e40992db9d 2.fa
d41d8cd98f00b204e9800998ecf8427e empty.fa
```

The `md5sum` tool has a `--check` option that I can use to automatically verify that the files match the checksums found in a given file: The macOS `md5` tool does not have an option for this, but you can use **`brew install md5sha1sum`** to install a tool an equivalent `md5sum` tool that can do this:

```
$ md5sum --check checksums.md5
1.fa: OK
2.fa: OK
empty.fa: OK
```

MD5 checksums present more complete and easier ways to verify data integrity over manually checking file sizes. Although file digests are not directly part of this exercise, I feel it's important to understand how to verify that you have complete and uncorrupted data before beginning any analyses.

Getting Started

You should work in the *15_seqmagique* directory for this exercise. I'll start the program as usual:

```
$ new.py -fp 'Mimic seqmagick' seqmagique.py
Done, see new script "seqmagique.py".
```

First I need to make the program accept one or more text files as positional parameters. I also want to create an option to control the output table format. Here is the code for that:

```
import argparse
from typing import NamedTuple, TextIO, List


class Args(NamedTuple):
    """ Command-line arguments """
    file: List[TextIO]
    tablefmt: str


def get_args() -> Args:
    """Get command-line arguments"""

    parser = argparse.ArgumentParser(
        description='Argparse Python script',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file', ❶
                        metavar='FILE',
                        type=argparse.FileType('rt'),
                        nargs='+',
                        help='Input FASTA file(s)')

    parser.add_argument('-t',
                        '--tablefmt',
                        metavar='table',
                        type=str,
                        choices=[ ❷
                            'plain', 'simple', 'grid', 'pipe',
'orgtbl', 'rst',
                            'mediawiki', 'latex', 'latex_raw',
'latex_booktabs'
                        ],
                        default='plain',
                        help='Tabulate table style')

    args = parser.parse_args()

    return Args(args.file, args.tablefmt)
```

❶ Define a positional parameter for one or more readable text files.

❷ Define an option that uses `choices` to constrain the argument to a value in the list, making sure to define a reasonable `default` value.

Using `choices` for the `--tablefmt` really saves you quite a bit of work in validating user input. As shown in the introduction to this chapter, a bad value for the table format option will trigger a useful error message.

Modify the `main()` to print the input filenames:

```
def main() -> None:
    args = get_args()
    for fh in args.file:
        print(fh.name)
```

Verify this works:

```
$ ./seqmagique.py tests/inputs/*.fa
tests/inputs/1.fa
tests/inputs/2.fa
tests/inputs/empty.fa
```

The goal is to iterate through each file and print the following:

- `name`: The filename
- `min_len`: The length of the shortest sequence
- `max_len`: The length of the longest sequence
- `avg_len`: The average/mean length of all the sequences
- `num_seqs`: The number of sequences

If you would like to have some real input files for our program, you can use the `fastq-dump` tool from NCBI to download some reads

from the study "Planktonic microbial communities from North Pacific Subtropical Gyre":

```
$ fastq-dump --split-3 SAMN00000013 ❶
```

❶ The `--split-3` option will ensure that paired-end reads are correctly split into forward/reverse/unpaired reads. The `SAMN00000013` string is the accession of one of the samples from the experiment.

Formatting Text Tables Using tabulate

The output of the program will be a text table formatted using the `tabulate()` function from that module. Be sure to read the documentation:

```
>>> from tabulate import tabulate
>>> help(tabulate)
```

I need to define the headers for the table, and I decided to use the same ones as Seqmagick (minus the *alignment* column):

```
>>> hdr = ['name', 'min_len', 'max_len', 'avg_len', 'num_seqs']
```

The first test file, *tests/data/1.fa*, has just one sequence of 50 bases, and so the columns for this are as follows:

```
>>> f1 = ['tests/inputs/1.fa', 50, 50, 50.00, 1]
```

The second test file, *tests/data/2.fa*, has five sequences ranging from 49 bases to 79 with an average length of 64 bases:

```
>>> f2 = ['tests/inputs/2.fa', 49, 79, 64.00, 5]
```

The `tabulate()` function expects the table data to be passed positionally as a list of lists, and I can specify the `headers` as a keyword argument:

```
>>> print(tabulate([f1, f2], headers=hdr))
name                 min_len    max_len    avg_len    num_seqs
-----------------  ---------  ---------  ---------  ----------
tests/inputs/1.fa         50         50         50           1
tests/inputs/2.fa         49         79         64           5
```

Alternately, I can place the headers as the first row of data and indicate this is the location of the headers:

```
>>> print(tabulate([hdr, f1, f2], headers='firstrow'))
name                 min_len    max_len    avg_len    num_seqs
-----------------  ---------  ---------  ---------  ----------
tests/inputs/1.fa         50         50         50           1
tests/inputs/2.fa         49         79         64           5
```

Note that the default table style for the `tabulate()` function is *simple*, but the *plain* format is what I need to match Seqmagick's output. Use the `tablefmt` option to set this:

```
>>> print(tabulate([f1, f2], headers=hdr, tablefmt='plain'))
name                 min_len    max_len    avg_len    num_seqs
tests/inputs/1.fa         50         50         50           1
tests/inputs/2.fa         49         79         64           5
```

One other thing to note is that the *avg_len* column data is being shown as an integer value but should be formatted as a floating-point number to two decimal places. The `floatfmt` option controls this, using syntax similar to the f-string number formatting I've shown before:

```
>>> print(tabulate([f1, f2], headers=hdr, tablefmt='plain',
floatfmt='.2f'))
name                 min_len    max_len    avg_len    num_seqs
tests/inputs/1.fa         50         50      50.00           1
tests/inputs/2.fa         49         79      64.00           5
```

Your job is to process all the sequences in each file to find the statistics and print the final table. This should be enough for you to solve the problem. Don't read ahead until you can pass all the tests.

Solutions

I will show two solutions that both show the file statistics but which differ in the formatting of the output. The first solution uses the `tabulate()` function to create an ASCII text table while the second uses the `rich` module to create a fancier table sure to impress your labmates and PI.

Solution 1: Formatting with tabulate

For my solution, I first decided to write a `process()` function that would handle each input file. Whenever I approach a problem that needs to handle some list of items, I prefer to focus on how to handle just one of the items. That is, rather than trying to find all the statistics for all the files, I first want to figure out how to find this information for just one file.

My function needs to return the filename and the four metrics: minimum/maximum/average sequence lengths plus the number of sequences. A tuple or a dictionary would be fine data structures for this, but I really prefer to create a type based on a `NamedTuple` for this so that I have a statically typed container that `mypy` can validate:

```
class FastaInfo(NamedTuple):
    """ FASTA file information """
    filename: str
    min_len: int
    max_len: int
    avg_len: float
    num_seqs: int
```

Now I can define a function that returns this data structure. Note that I'm using the `numpy.mean()` function to get the average length. The `numpy` module offers many powerful mathematical operations to handle numeric data and is especially useful for multidimensional arrays and linear algebra functions. It's common to import the `numpy` module with the alias `np`:

```
import numpy as np
```

Here is how I incorporate this:

```
def process(fh: TextIO) -> FastaInfo: ❶
    """ Process a file """

    if lengths := [len(rec.seq) for rec in SeqIO.parse(fh, 'fasta')]:
❷
        return FastaInfo(filename=fh.name, ❸
                         min_len=min(lengths), ❹
                         max_len=max(lengths), ❺
                         avg_len=round(np.mean(lengths), 2), ❻
                         num_seqs=len(lengths)) ❼

    return FastaInfo(filename=fh.name, ❽
                     min_len=0,
                     max_len=0,
                     avg_len=0,
                     num_seqs=0)
```

❶ The function accepts a filehandle and returns a `FastaInfo` object.

❷ Use a list comprehension to read all the sequences from the filehandle. Use the `len()` function to return the length of each sequence.

❸ The name of the file is available through the `fh.name` attribute.

❹ The `min()` function will return the minimum value.

❺ The `max()` function will return the maximum value.

❻ The `np.mean()` function will return the mean from a list of values. The `round()` function is used to round this floating-point value to two significant digits.

❼ The number of sequences is the length of the list.

❽ If there are no sequences, return zeros for all the values.

As always, I want to write a unit test for this. Since the function reads a filehandle, it raises the question of how to write such a test. Should I rely on the files in the *tests/data* directory, or can I somehow fake a filehandle? It's common in testing vernacular to describe these kinds of fake objects and interfaces as *mocks*.

The first test file looks like this:

```
$ cat tests/inputs/1.fa
>SEQ0
GGATAAAGCGAGAGGCTGGATCATGCACCAACTGCGTGCAACGAAGGAAT
```

I can use the `io.StringIO()` function to create an object that behaves like a filehandle:

```
>>> f1 = '>SEQ0\nGGATAAAGCGAGAGGCTGGATCATGCACCAACTGCGTGCAACGAAGGAAT\n'
❶
>>> fh = io.StringIO(f1) ❷
>>> for line in fh: ❸
...     print(line, end='') ❹
...
>SEQ0
GGATAAAGCGAGAGGCTGGATCATGCACCAACTGCGTGCAACGAAGGAAT
```

❶ This is the data from the first input file.

❷ Create a mock filehandle.

❸ Iterate through the *lines* of the mock filehandle.

❹ Print the line which has a newline so use `end` to leave off an additional newline.

There's a slight problem, though, because the `process()` function calls the `fh.name` attribute to get the input filename, which will raise an exception:

```
>>> fh.name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: '_io.StringIO' object has no attribute 'name'
```

Luckily, there's another way to create a mock filehandle using Python's standard `unittest` module. While I favor the `pytest` module for almost everything I write, the `unittest` module has been around for a long time and is another capable framework for writing and running tests. In this case, I only need to import the `unittest.mock.mock_open()` function. Here is how I can create a mock filehandle with the data from the first test file. I use `read_data` to define the data that will be returned by the `fh.read()` method:

```
>>> from unittest.mock import mock_open
>>> fh = mock_open(read_data=f1)()
>>> fh.read()
'>SEQ0\nGGATAAAGCGAGAGGCTGGATCATGCACCAACTGCGTGCAACGAAGGAAT\n'
```

Note the lack of an exception when asking for the filehandle's name:

```
>>> fh.name
<MagicMock name='open().name' id='140349116126880'>
```

While I often place my unit tests in the same modules as the functions they test, in this instance, I'd rather put this into a separate `unit.py` module to keep the main program shorter. I wrote the test to handle an empty file, a file with one sequence, and a file with more than one sequence (which are also reflected in the three input test

files). Presumably, if the function works for these three cases, it should work for all others:

```
from unittest.mock import mock_open ❶
from seqmagique import process ❷


def test_process() -> None:
    """ Test process """

    empty = process(mock_open(read_data='')()) ❸
    assert empty.min_len == 0
    assert empty.max_len == 0
    assert empty.avg_len == 0
    assert empty.num_seqs == 0

    one = process(mock_open(read_data='>SEQ0\nAAA')()) ❹
    assert one.min_len == 3
    assert one.max_len == 3
    assert one.avg_len == 3
    assert one.num_seqs == 1

    two = process(mock_open(read_data='>SEQ0\nAAA\n>SEQ1\nCCCC')()) ❺
    assert two.min_len == 3
    assert two.max_len == 4
    assert two.avg_len == 3.5
    assert two.num_seqs == 2
```

❶ Import the `mock_open()` function.

❷ Import the `process()` function I'm testing.

❸ A mock empty filehandle that should have zeros for all the values.

❹ A single sequence with three bases.

❺ A filehandle with two sequences of three and four bases.

I can run this using `pytest` like so:

```
$ pytest -xv unit.py
============================= test session starts
==============================
...

unit.py::test_process PASSED
[100%]

============================== 1 passed in 2.55s
==============================
```

WHERE TO PLACE UNIT TESTS

Note that the preceding `unit.py` module imports the `process()` function from the `seqmagique.py` module, and so both modules need to be in the same directory. If I were to move `unit.py` to the *tests* directory, then `pytest` would break. I encourage you to try the following and note the errors. You should get a notice like *ModuleNotFoundError: No module named 'seqmagique'*:

```
$ cp unit.py tests
$ pytest -xv tests/unit.py
```

As noted in the `pytest` documentation, I must invoke `pytest` as follows to add the current directory to `sys.path` so that `seqmagique.py` can be found:

```
$ python3 -m pytest -xv tests/unit.py
```

Placing `unit.py` in the same directory as the code it's testing is more slightly convenient as I can run the shorter `pytest` command, but grouping all the tests into the *tests* directory is tidier. I normally prefer to have this module as *tests/unit_test.py* so that `pytest` would automatically discover it, and I would use a `make` target to run the longer invocation. Mostly I just want you to be aware of different ways to organize your code and tests and the larger implications these can have.

Here is how I use my `process()` function in the `main()`:

```
def main() -> None:
    args = get_args()
    data = [process(fh) for fh in args.file] ❶
    hdr = ['name', 'min_len', 'max_len', 'avg_len', 'num_seqs'] ❷
    print(tabulate(data, tablefmt=args.tablefmt, headers=hdr,
floatfmt='.2f')) ❸
```

❶ Process all the input files into a list of `FastaInfo` objects (tuples).

❷ Define the table headers.

❸ Use the `tabulate()` function to print a formatted output table.

To test this program, I run the program with the following inputs:

- The empty file
- The file with one sequence
- The file with two sequences
- All the input files

To start, I run all these with the default table style. Then I need to verify that all 10 of the table styles are created correctly. If I combine all the possible test inputs with all the table styles, it a high degree of *cyclomatic complexity* — the number of different ways that parameters can be combined.

To test this, I first need to manually verify that my program is working correctly. Then I need to generate sample outputs for each of the combinations I intend to test. I wrote the following `bash` script to create an *out* file for a given combination of an input file and possibly a table style:

```
$ cat mk-outs.sh
#!/usr/bin/env bash

PRG="./seqmagique.py" ❶
DIR="./tests/inputs" ❷
INPUT1="${DIR}/1.fa" ❸
INPUT2="${DIR}/2.fa"
EMPTY="${DIR}/empty.fa"

$PRG $INPUT1 > "${INPUT1}.out" ❹
$PRG $INPUT2 > "${INPUT2}.out"
$PRG $EMPTY > "${EMPTY}.out"
```

```
$PRG $INPUT1 $INPUT2 $EMPTY > "$DIR/all.fa.out"

STYLES="plain simple grid pipe orgtbl rst mediawiki latex latex_raw
latex_booktabs"

for FILE in $INPUT1 $INPUT2; do ❺
    for STYLE in $STYLES; do
        $PRG -t $STYLE $FILE > "$FILE.${STYLE}.out"
    done
done

echo Done.
```

❶ The program being tested

❷ The directory for the input files

❸ The input files

❹ Run the program using the three input files and the default table style.

❺ Run the program with two of the input files and all the table styles.

The test suite will run the program with a given file and will compare the output to one of the *out* files in the *tests/inputs* directory. For instance, the test defines the input and output files like so:

```
TEST1 = ('./tests/inputs/1.fa', './tests/inputs/1.fa.out')
```

I define a test runner function like so:

```
def run(input_file: str, expected_file: str) -> None:
    """ Runs on command-line input """

    expected = open(expected_file).read().rstrip() ❶
    rv, out = getstatusoutput(f'{RUN} {input_file}') ❷
```

```
    assert rv == 0 ❸
    assert out == expected ❹
```

❶ Read the expected output from the file.

❷ Run the program with the given input file using the default table style.

❸ Check that the return value is 0.

❹ Check that the output was the expected value.

I run it like so:

```
def test_input1() -> None:
    """ Runs on command-line input """

    run(*TEST1) ❶
```

❶ Splat the tuple to pass the two values positionally to the `run()` function.

Here is how I test the table styles:

```
def test_styles() -> None:
    """ Test table styles """

    styles = [ ❶
        'plain', 'simple', 'grid', 'pipe', 'orgtbl', 'rst',
'mediawiki',
        'latex', 'latex_raw', 'latex_booktabs'
    ]

    for file in [TEST1[0], TEST2[0]]: ❷
        for style in styles: ❸
            expected_file = file + '.' + style + '.out' ❹
            assert os.path.isfile(expected_file) ❺
            expected = open(expected_file).read().rstrip() ❻
            flag = '--tablefmt' if random.choice([0, 1]) else '-t' ❼
```

```
            rv, out = getstatusoutput(f'{RUN} {flag} {style} {file}')
❽

            assert rv == 0 ❾
            assert out == expected
```

❶ Define a list of all possible styles.

❷ Use the two non-empty files.

❸ Iterate through each style.

❹ The output file is the name of the input file plus the style and the extension *.out*.

❺ Check that the file exists.

❻ Read the expected value from the file.

❼ Randomly choose the short or long flag to test.

❽ Run the program with the flag option, style, and file.

❾ Ensure the program ran without error and produces the correct output.

If I make a change such that the program no longer creates the same output as before, these tests should catch it. This is a *regression* test where I am comparing how a program works now to how it previously worked. That is, a failure to produce the same output would be considered a regression. While my test suite isn't completely exhaustive, it covers enough combinations that I feel confident the program is correct.

Solution 2: Formatting with rich

In this second solution, I want to show a different way to create the output table using the `rich` module to track the processing of the input files and make a fancier output table. Figure 15-1 shows how the output looks different:

```
$ ./seqmagique_rich.py tests/inputs/*.fa
Working... ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 100% 0:00:00
```

Name	Min. Len	Max. Len	Avg. Len	Num. Seqs
tests/inputs/1.fa	50	50	50.0	1
tests/inputs/2.fa	49	79	64.0	5
tests/inputs/empty.fa	0	0	0	0

Figure 15-1. The progress indicator and output table using the `rich` module is fancier.

I still process the files in the same way, so the only difference is in creating the output. I first need to import the needed functions:

```
from rich import box
from rich.console import Console
from rich.progress import track
from rich.table import Table, Column
```

Here is how I use these:

```
def main() -> None:
    args = get_args()

    table = Table('Name', Column(header='Min. Len', justify='right'),
❷
                  Column(header='Max. Len', justify='right'),
                  Column(header='Avg. Len', justify='right'),
                  Column(header='Num. Seqs', justify='right'))

    for file in track([process(fh) for fh in args.file]): ❷
        table.add_row(file.filename, str(file.min_len),
```

```
            str(file.max_len), ❸
                               str(file.avg_len), str(file.num_seqs))

        Console().print(table) ❹
```

❶ Create the table to hold the data. The `'Name'` column is a standard, left-justified string field. All the others need to be right-justified and so require a custom `Column()` object.

❷ Iterate through each file's data using the `track` function to create a progress bar for the user.

❸ Add the file's statistics to the table. Note that all the values must be strings.

❹ Print the table to the console.

Going Further

The `seqmagick` tool has many other useful options. Implement your own version of as many as you can.

Review

- The `seqmagick` tool provides many nice options for examining sequence files.
- There are many ways to verify that your input files are complete and not corrupted from examining file sizes to using message digests such as MD5 hashes.
- The `tabulate` and `rich` modules can create text tables of data.
- The `numpy` module is useful for many mathematical operations.

- The `io.StrioIO()` and `unittest.mock.mock_open()` functions offer two ways to mock a filehandle for testing.
- Regression testing verifies that a program continues to work as it did before.

Chapter 16. FASTX Grep: Creating a Utility Program to Select Sequences

Once a colleague asked me to find all the sequences in a FASTQ file that had a description or name matching *LSU* (for *long subunit*) RNA. Although it's possible to solve this problem for FASTQ files by using the `grep` program[1] to find all the lines of a file matching some pattern, writing a solution in Python allows me to create a program that could be expanded to handle other formats like FASTA as well as to select records based on other criteria such as length or GC content. Additionally, I can add options to change the output sequence format and introduce conveniences for the user like guessing the input file's format based on the file extension.

You will learn:

- About the structure of a FASTQ file
- How to perform a case-insensitive regular expression match
- About DWIM (Do What I Mean) and DRY (Don't Repeat Yourself) ideas in code
- How to use bitwise operations to combine integers

Finding Lines in a File Using grep

The `grep` program can find all the lines in a file matching a given pattern. If I search for *LSU* in one of the FASTQ files, it finds two header lines containing this pattern:

```
$ grep LSU tests/inputs/lsu.fq
@ITSLSUmock2p.ITS_M01380:138:000000000-C9GKM:1:1101:14440:2042 2:N:0
@ITSLSUmock2p.ITS_M01384:138:000000000-C9GKM:1:1101:14440:2043 2:N:0
```

If the goal were only to find how many sequences contain this string, I could pipe this into `wc` (word count) to count the lines:

```
$ grep LSU tests/inputs/lsu.fq | wc -l
       2
```

Since my goal is to extract the sequence records where the header contains the substring *LSU*, I have to do a little more work. As long as the input files are in FASTQ format, I can still use `grep`, but this requires a better understanding of the format.

The Structure of a FASTQ Record

The FASTQ sequence format is a common way to receive sequence data from a sequencer as it includes both the sequence and the quality (the *Q* in *FASTQ*) scores for each base. That is, sequencers generally report both a base and a measure of certainty that the base is correct. Some sequencing technologies have trouble, for instance, with homopolymer runs like a poly(A) run of many *A*s where the sequencer may be unable to count the correct number. Many sequencers also lose confidence of base calls as the reads grow longer. Quality scores are an important means for rejecting or truncating low-quality reads.

> **NOTE**
>
> Depending on the sequencer, some bases can be hard to distinguish, and so the ambiguity may be reported using IUPAC codes I describe in Chapter 1 such as *R* for *A* or *G* or *N* for any base.

The FASTQ format is somewhat similar to the FASTA sequence used in so many of the problems in the Rosalind challenges. As a reminder, FASTA records start with a `>` symbol followed by a header line that identifies the sequence and may contain metadata. The sequence itself follows which can be split over multiple lines or be represented using one (possibly long) line of text.

In contrast, FASTQ records must always be exactly four lines. Here is a sample FASTQ record:

```
@Paramo13_ITS__M01380:138:000000000-C9GKM:1:1101:8813:1878 1:N:0 ❶
GGAAGGATCATTACAGAGAACATGCCCTTTGTGGTATATCTCCCACCCTTTGTTTACAATACTTTTTTTC
TT\ ❷
TTCCTTCCCCTTCTTCTAGTCCTCTTGCCTTTTCTCCTCTTTTCCTGCCTAAGTTTCCCTAACACTGTTT
TT\
TTTATGCTGTCTCATTTCTATACAATAGTTACAAACTTTCTACAACGTATCTCTTGTTTCTTGCTTCTAT
GA\
TGTTCTCAGCGACATGCGTTATGTAATGTGAATTTCAGAATTCATTGTATCATCCAATCTTT
+ ❸
GGG:@FGFGCFF<@FGGGGGG,C,66<C@F,,,,;,<,<<@,,6,9,69CC,<E@,
<,,6,,CC6,CC+66<\ ❹
6,,:,,,66:,:4?E,C,,,,,,,
<,,,,:B,,::,,5,9,5,55:,44,,,,,,55,55,49,6?,9ACA+\
4@>,C,3=,8,C,,,8,,6,3,,,33,,6@,,,,,445;3,,5,,,*3,7:;=A;,,54@;,,3,5,,,5
,,\
++++5+5++**)))8*0))/))+;++331+++3:+3++))1943+31++/7)9)))).//76
```

❶ The header line starts with the `@` symbol.

❷ The sequence cannot have line breaks. I'm using `\` to show line continuation of a long sequence.

❸ The third line begins with the `+` symbol and sometimes repeats the header.

❹ The fourth line contains the quality scores for each base in the sequence and also has no line breaks.

A FASTQ header has the same structure as a the header in a FASTA record with the exception that it starts with the `@` sign instead of the `>`. The sequence identifier is usually all the characters after `@` up to the first space.

The second line containing the sequence cannot contain any line breaks, and each base in the sequence has a corresponding quality value in the fourth line. The quality scores on the fourth line use the ASCII value of the characters to encode the certainty of the base call. These scores are represented using the printable characters from the ASCII table first introduced in Chapter 3.

The first 32 values in the ASCII table are unprintable control characters and the space. The printable characters start at 33 with punctuation followed by numbers. The first letter, *A*, is not found until 65, and uppercase characters precede lowercase. The following is the output from the `asciitbl.py` program included in the repository that shows the ordinal values of the 128 values from the ASCII table:

```
$ ./asciitbl.py
  0 NA      26 NA      52 4       78 N      104 h
  1 NA      27 NA      53 5       79 O      105 i
  2 NA      28 NA      54 6       80 P      106 j
  3 NA      29 NA      55 7       81 Q      107 k
  4 NA      30 NA      56 8       82 R      108 l
  5 NA      31 NA      57 9       83 S      109 m
  6 NA      32 SPACE   58 :       84 T      110 n
  7 NA      33 !       59 ;       85 U      111 o
  8 NA      34 "       60 <       86 V      112 p
  9 NA      35 #       61 =       87 W      113 q
 10 NA      36 $       62 >       88 X      114 r
 11 NA      37 %       63 ?       89 Y      115 s
 12 NA      38 &       64 @       90 Z      116 t
 13 NA      39 '       65 A       91 [      117 u
 14 NA      40 (       66 B       92 \      118 v
 15 NA      41 )       67 C       93 ]      119 w
 16 NA      42 *       68 D       94 ^      120 x
 17 NA      43 +       69 E       95 _      121 y
 18 NA      44 ,       70 F       96 `      122 z
 19 NA      45 -       71 G       97 a      123 {
 20 NA      46 .       72 H       98 b      124 |
```

```
 21 NA      47 /       73 I       99 c      125 }
 22 NA      48 0       74 J      100 d      126 ~
 23 NA      49 1       75 K      101 e      127 DEL
 24 NA      50 2       76 L      102 f
 25 NA      51 3       77 M      103 g
```

Look at the quality line from the preceding FASTQ record and see how the characters change from higher values like uppercase letters at the beginning to lower values like punctuation and numbers toward the end. Note the `@` and `+` symbols on the fourth line represent possible quality values and so would not be metacharacters denoting the beginning of a record or the separator line. For this reason, FASTQ records can't introduce newlines to break the sequence and quality lines seen in FASTA records because `@` and `+` might end up as the first character on a line making it impossible to find the start of a record. The only sane way to structure FASTQ is to always use four lines per record, no matter how long the lines become. Combine this with the utterly useless third line that often consists of a single `+` symbol and which sometimes needlessly recapitulates all the header information and you see why biologists should never be allowed to define a file format.

Because FASTQ records must be four lines long, I can use the `-A|--after-context` option for `grep` to specify the number of lines of trailing context after each match:

```
$ grep -A 4 LSU tests/inputs/lsu.fq | head -4
@ITSLSUmock2p.ITS_M01380:138:000000000-C9GKM:1:1101:14440:2042 2:N:0
CAAGTTACTTCCTCTAAATGACCAAGCCTAGTGTAGAACCATGTCGTCAGTGTCAGTCTGAGTGTAGATC
T\
CGGTGGTCGCCGTATCATTAAAAAAAAAAATGTAATACTACTAGTAATTATTAATATTATAATTTTGTCT
A\
TTAGCATCTTATTATAGATAGAAGATATTATTCATATTTCACTATCTTATACTGATATCAGCTTTATCAG
A\
TCACACTCTAGTGAAGATTGTTCTTAACTGAAATTTCCTTCTTCATACAGACACATTAATCTTACCTA
+
EFGGGGGGGGGCGGGGGFCFFFGGGGGFGGGGGGGGGGGFGGGGGGGFGFFFCFGGFFGGGGGGGGGFGG
G\
```

```
GFGGGDG<FD@4@CFFGGGGCFFAFEFEG+,9,,,,99,,,5,,49,4,8,4,444,4,4,,,,,,,,,
,\
,,,8,,,,63,,,,,,,,376,3,,,,,,,8,,,,,,,,,++++++++++++3++25+++0+*+0+*0+
*\
**))*0))1/+++**************.****.*******0*********/(,(/).)))1)).).).
```

This works as long as the substring of interest occurs only in the header which is the first line of a record. If `grep` managed to find a match in any other line in the record, it would print that line plus the following three yielding unusable garbage. Given that I would like to control exactly which parts of the record to search and the fact that the input files might be in FASTQ, FASTA, or any other number of formats, and it quickly becomes evident that `grep` won't take me very far.

Getting Started

First, I'll show you how my solution works, and then I'll challenge you to implement your own version. All the code and tests for this exercise are in the *16_fastx_grep* directory of the repository. Start by changing into this directory and copying the solution to `fastx_grep.py`:

```
$ cd 16_fastx_grep
$ cp solution.py fastx_grep.py
```

The `grep` program requires two positional arguments: a pattern and one or more files:

```
$ grep -h
usage: grep [-abcDEFGHhIiJLlmnOoqRSsUVvwxZ] [-A num] [-B num] [-
C[num]]
        [-e pattern] [-f file] [--binary-files=value] [--color=when]
        [--context[=num]] [--directories=action] [--label] [--line-
buffered]
        [--null] [pattern] [file ...]
```

Request help from the `fastx_grep.py` program and see that it has a similar interface in that it requires a pattern and one or more input files. Additionally, this program can parse different input file formats, produce various output formats, write the output to a file, and perform case-insensitive matching:

```
$ ./fastx_grep.py -h
usage: fastx_grep.py [-h] [-f str] [-O str] [-o FILE] [-i] [-v]
                     PATTERN FILE [FILE ...]

Grep through FASTX files

positional arguments:
  PATTERN               Search pattern
  FILE                  Input file(s)

optional arguments:
  -h, --help            show this help message and exit
  -f str, --format str  Input file format (default: )
  -O str, --outfmt str  Output file format (default: )
  -o FILE, --outfile FILE
                        Output file (default: <_io.TextIOWrapper
                        name='<stdout>' mode='w' encoding='utf-8'>)
  -i, --insensitive     Case-insensitive search (default: False)
  -v, --verbose         Be chatty (default: False)
```

This program has a more complicated set of arguments than many of the programs from Part 1. As usual, I like to use a `NamedTuple` to model the options:

```
from typing import List, NamedTuple, TextIO


class Args(NamedTuple):
    """ Command-line arguments """
    pattern: str ❶
    files: List[TextIO] ❷
    input_format: str ❸
    output_format: str ❹
    outfile: TextIO ❺
    insensitive: bool ❻
    verbose: bool ❼
```

❶ The regular expression to use

❷ One or more input files

❸ The format (e.g., FASTA, FASTQ) of the input file

❹ The format of the output file

❺ The name of the output file

❻ Whether to perform case-insensitive searching

❼ Whether to print the progress

Here is how I define the program's parameters:

```
  def get_args() -> Args:
      """ Get command-line arguments """

      parser = argparse.ArgumentParser(
          description='Grep through FASTX files',
          formatter_class=argparse.ArgumentDefaultsHelpFormatter)

      parser.add_argument('pattern', ❶
                          metavar='PATTERN',
                          type=str,
                          help='Search pattern')

      parser.add_argument('file',
                          metavar='FILE',
                          nargs='+',
                          type=argparse.FileType('rt'), ❷
                          help='Input file(s)')

      parser.add_argument('-f',
                          '--format',
                          help='Input file format',
                          metavar='str',
                          choices=['fasta', 'fastq'], ❸
                          default='')
```

```
    parser.add_argument('-O',
                        '--outfmt',
                        help='Output file format',
                        metavar='str',
                        choices=['fasta', 'fastq', 'fasta-2line'], ❹
                        default='')

    parser.add_argument('-o',
                        '--outfile',
                        help='Output file',
                        type=argparse.FileType('wt'), ❺
                        metavar='FILE',
                        default=sys.stdout)

    parser.add_argument('-i', ❻
                        '--insensitive',
                        help='Case-insensitive search',
                        action='store_true')

    parser.add_argument('-v', ❼
                        '--verbose',
                        help='Be chatty',
                        action='store_true')

    args = parser.parse_args()

    return Args(pattern=args.pattern,
                files=args.file,
                input_format=args.format,
                output_format=args.outfmt,
                outfile=args.outfile,
                verbose=args.verbose,
                insensitive=args.insensitive)
```

❶ The pattern will be a string.

❷ The inputs must be readable text files.

❸ Use `choices` to constrain the input values. The default will be guessed from the input file extension.

❹ Constrain values using `choices`, default to using the input format.

❺ The output file will be a writable text file. The default will be `STDOUT`.

❻ A flag to indicate case-insensitive searching. The default is `False`.

❼ A flag to print the status at the end. The default is `False`.

If you run the following command to search for *LSU* in the *lsu.fq* test file, you should see eight lines of output representing two FASTQ records:

```
$ ./fastx_grep.py LSU tests/inputs/lsu.fq | wc -l
       8
```

If you search for lowercase *lsu*, however, you should see no output:

```
$ ./fastx_grep.py lsu tests/inputs/lsu.fq | wc -l
       0
```

Use the `-i|--insensitive` flag to perform a case-insensitive search:

```
$ ./fastx_grep.py -i lsu tests/inputs/lsu.fq  | wc -l
       8
```

You can use the `-o|--outfile` option to write the results to a file instead of `STDOUT`, and the `-v|--verbose` flag will provide a summary status:

```
$ ./fastx_grep.py -o out.fq -i lsu -v tests/inputs/lsu.fq
  1: tests/inputs/lsu.fq
Done, checked 4, wrote 2 to "out.fq".
```

If you look at the *out.fq* file, you'll see it's in FASTQ format just like the original input. You can use the `-O|--outfmt` option to change this to something like FASTA:

```
$ ./fastx_grep.py -O fasta -o out.fa -i lsu -v tests/inputs/lsu.fq
  1: tests/inputs/lsu.fq
Done, checked 4, wrote 2 to "out.fa".
```

I can verify this works:

```
$ head -3 out.fa
>ITSLSUmock2p.ITS_M01380:138:000000000-C9GKM:1:1101:14440:2042 2:N:0
CAAGTTACTTCCTCTAAATGACCAAGCCTAGTGTAGAACCATGTCGTCAGTGTCAGTCTG
AGTGTAGATCTCGGTGGTCGCCGTATCATTAAAAAAAAAAATGTAATACTACTAGTAATT
```

Note that the program also works on FASTA input:

```
$ ./fastx_grep.py -o out.fa -i lsu -v tests/inputs/lsu.fa
  1: tests/inputs/lsu.fa
Done, checked 2, wrote 2 to "out.fa".
```

Run **`pytest -v`** to see all the tests for the program that include guessing the file format, handling empty files, searching lowercase and uppercase input both with and without case-sensitivity, writing output files, printing verbose output, and writing different output formats. When you think you understand all the options your program must handle, start over:

```
$ new.py -fp 'Grep through FASTX files' fastx_grep.py
Done, see new script "fastx_grep.py".
```

Guessing the File Format

If you look at *out.fa* created in the preceding section, you'll see it's in FASTA format, matching the input format, but I never indicated the input file format. The program intelligently checks the file extension of the input file and guesses at the format using the assumptions in Table 16-1. Similarly, if no output format is specified, then the input file format is assumed to be the desired output format. This is an example of the *DWIM* principle in software development: Do What I Mean.

Table 16-1. Common file extensions for FASTA/Q files

Extension	Format
`fasta`	FASTA
`fa`	FASTA
`fna`	FASTA (nucleotides)
`faa`	FASTA (amino acids)
`fq`	FASTQ
`fastq`	FASTQ

Your program will similarly need to guess the format of the input files. I created a `guess_format()` function that takes the name of a file and returns a string of either *fasta* or *fastq*. Here is a stub for the function:

```
def guess_format(filename: str) -> str:
    """ Guess format from extension """

    return ''
```

Here is the test I wrote. After defining the arguments, I would recommend you start with this function. Do not proceed until your code passes this:

```
def test_guess_format() -> None:
    """ Test guess_format """

    assert guess_format('/foo/bar.fa') == 'fasta'
    assert guess_format('/foo/bar.fna') == 'fasta'
    assert guess_format('/foo/bar.faa') == 'fasta'
    assert guess_format('/foo/bar.fasta') == 'fasta'
    assert guess_format('/foo/bar.fq') == 'fastq'
    assert guess_format('/foo/bar.fastq') == 'fastq'
    assert guess_format('/foo/bar.fx') == ''
```

It might help to sketch out how the program should work. Note that in my solution I print all optional messages to `STDERR`:

```
def main():
    get the program arguments

    initialize counters for how many sequences I process and take
    for each input file:
        maybe print the file number and name
        guess the input format or complain that it can't be guessed
        figure out the output format from the args or use the input
format

        for each record in the input file:
            increment the number of sequences seen
            if the sequence ID or description matches the pattern:
                increment the number of sequences taken
                write the sequence to the output file in the output
format

    maybe print a final status of how many sequences were written
where
```

For instance, I can run the program on three input files. It checks six sequences and writes four of them to the indicated output file:

```
$ ./fastx_grep.py -O fasta -o out.fa -i lsu -v tests/inputs/*.f[aq]
  1: tests/inputs/empty.fa
  2: tests/inputs/lsu.fa
  3: tests/inputs/lsu.fq
Done, checked 6, wrote 4 to "out.fa".
```

This is a complex program that may take you quite a while to finish. There is value in your struggle, so just keep writing and running the tests, which you should also read to understand how to challenge your program.

Solution

In my experience, this is a realistically complicated program that captures many patterns I write often. It starts by validating and processing some number of input files. I'm a truly lazy programmer[2] who always wants to give as little information as possible to my programs, so I'm happy to write a little code to guess the file formats for me. I sometimes want to write the output to a file, and I sometimes want my program to be as quiet as possible. To handle each option, I encapuslate the logic inside one function or line of code, and I use tests to ensure the program works with and without the options. When it all comes together, I end up with a capable and flexible tool.

Guessing the File Format from the File Extension

I'll start with the function for guessing a file's format from the file extension:

```
def guess_format(filename: str) -> str:
    """ Guess format from extension """

    ext = re.sub('^[.]', '', os.path.splitext(filename)[1]) ❶

    return 'fasta' if re.match('f(ast|a|n)?a$', ext) else 'fastq' if
re.match( ❷
        'f(ast)?q$', ext) else ''
```

❶ Use the `os.path.splitext()` function to get the file extension and remove the leading dot.

❷ Return the string "fasta" if the extension matches one of the patterns for FASTA files from Table 16-1, "fastq" if it matches a FASTQ pattern, and the empty string otherwise.

Recall that `os.path.splitext()` will return both the root of the filename and the extension as a 2-tuple:

```
>>> import os
>>> os.path.splitext('/foo/bar.fna')
('/foo/bar', '.fna')
```

Since I'm only interested in the second part, I can use the _ to assign the first member of the tuple to a throwaway:

```
>>> _, ext = os.path.splitext('/foo/bar.fna')
>>> ext
'.fna'
```

Instead, I chose to index the tuple to select only the extension:

```
>>> ext = os.path.splitext('/foo/bar.fna')[1]
>>> ext
'.fna'
```

Since I don't want the leading dot, I could use a string slice to remove this, but this looks really cryptic and unreadable to me:

```
>>> ext = os.path.splitext('/foo/bar.fna')[1][1:]
>>> ext
'fna'
```

Instead, I'd prefer to use the `re.sub()` function I first introduced in Chapter 2. The pattern I'm looking for is a literal dot at the beginning of the string. The caret `^` indicates the start of the string, and the `.` is a metacharacter that means one of anything. To show that I want a literal dot, I must either place a backslash in front of it like `^\.` or place it inside a character class as in `^[.]`:

```
>>> import re
>>> ext = re.sub('^[.]', '', os.path.splitext('/foo/bar.fna')[1]) ❶
>>> ext
'fna'
```

As shown in Table 16-1, there are four common extensions for FASTA files which I can represent using one compact regular

expression. Recall that there are two functions in the `re` module for searching:

1. `re.match()`: find a match from the beginning of a string
2. `re.search()`: find a match anywhere inside a string

In this example, I'm using the `re.match()` function to ensure that the pattern is found at the beginning of the extension:

```
>>> re.match('f(ast|a|n)?a$', ext)
<re.Match object; span=(0, 3), match='fna'>
```

To get the same results from `re.search()`, I would need to use a caret at the beginning to anchor the pattern to the start of the string:

```
>>> re.search('^f(ast|a|n)?a$', ext)
<re.Match object; span=(0, 3), match='fna'>
```

Figure 16-1 describes each part of the regular expression:

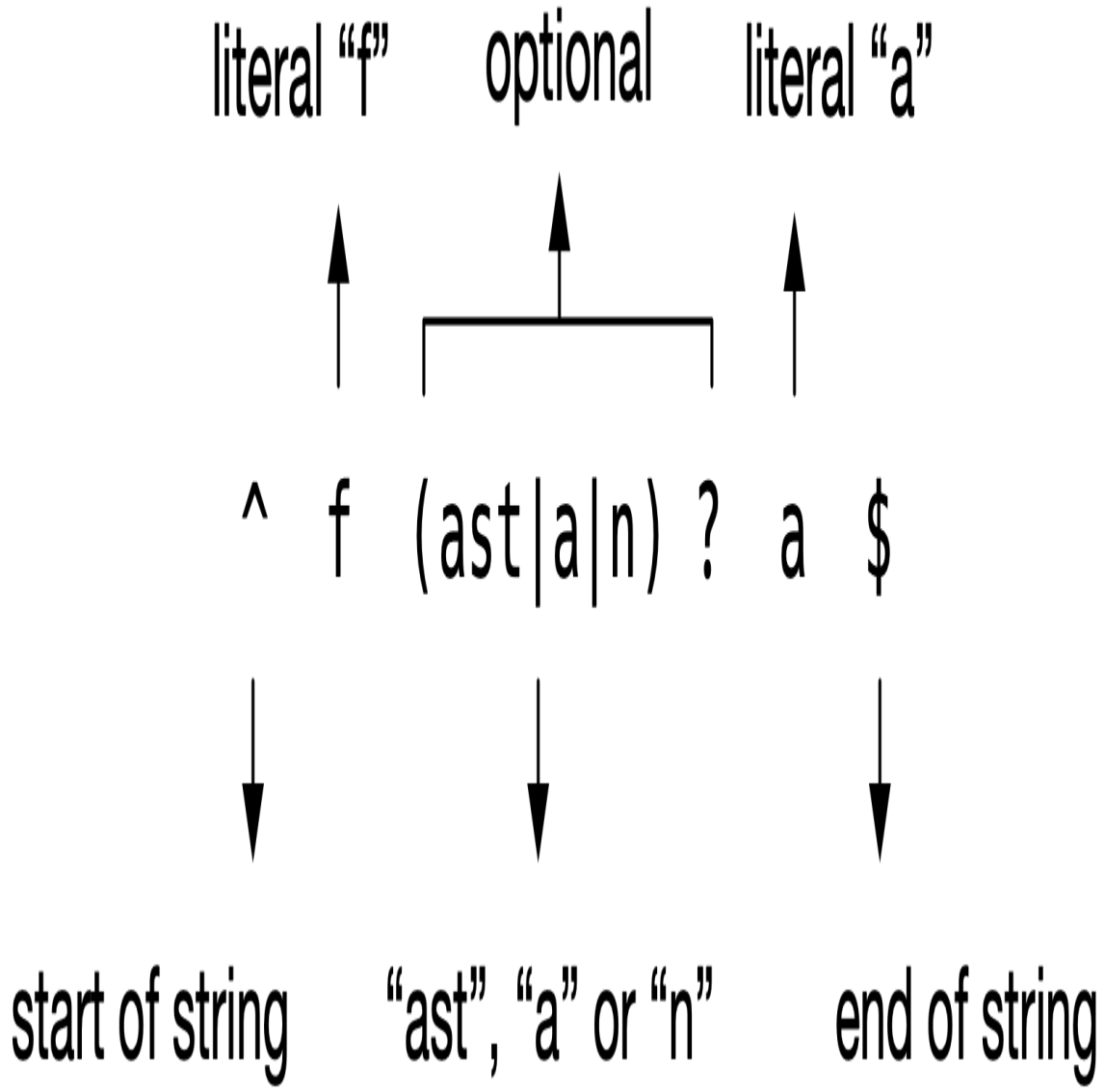


Figure 16-1. A regular expression for matching the four FASTA patterns.

It may help to see this drawn as a finite state machine diagram as shown in Figure 16-2.

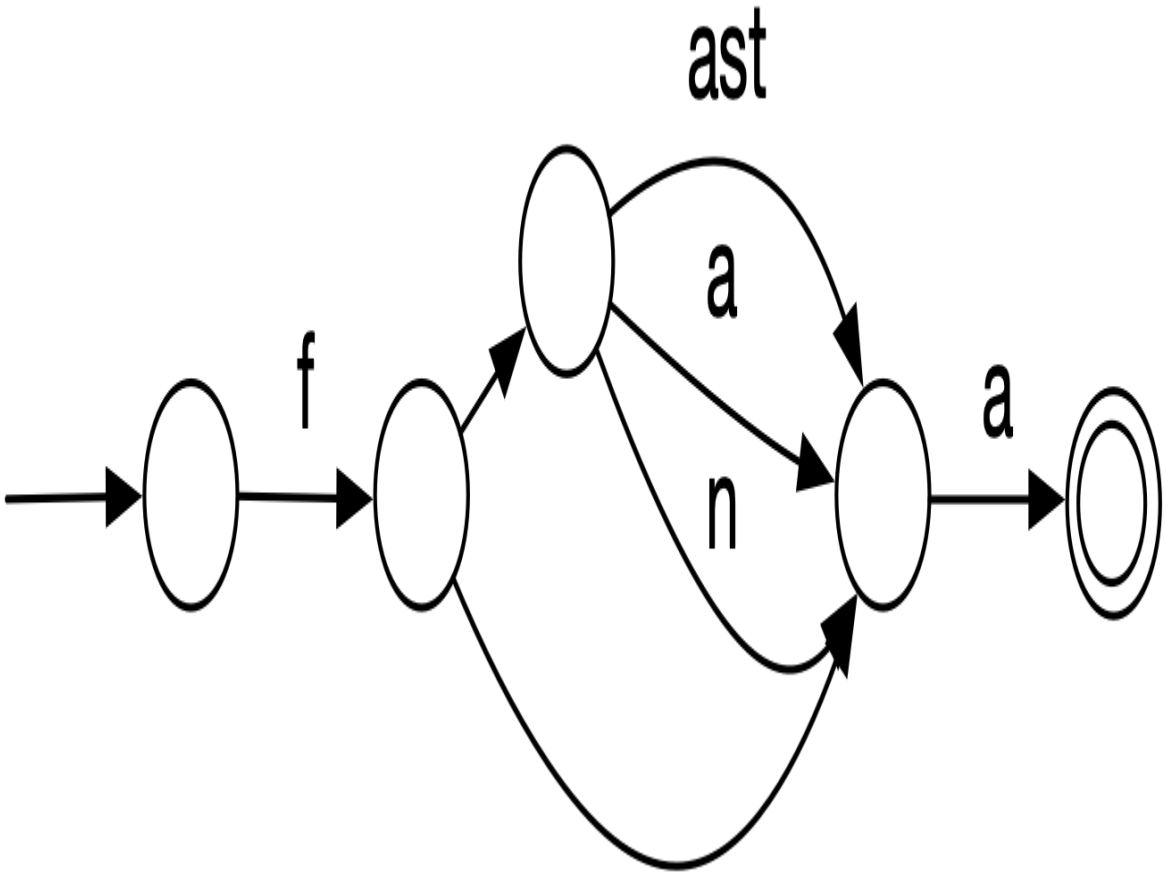


Figure 16-2. A finite state machine diagram for matching the four FASTA patterns.

As there are only two patterns for FASTQ files, the pattern is somewhat simpler:

```
>>> re.search('^f(ast)?q$', 'fq')
<re.Match object; span=(0, 2), match='fq'>
>>> re.search('^f(ast)?q$', 'fastq')
<re.Match object; span=(0, 5), match='fastq'>
```

Figure 16-3 explains this regular expression.

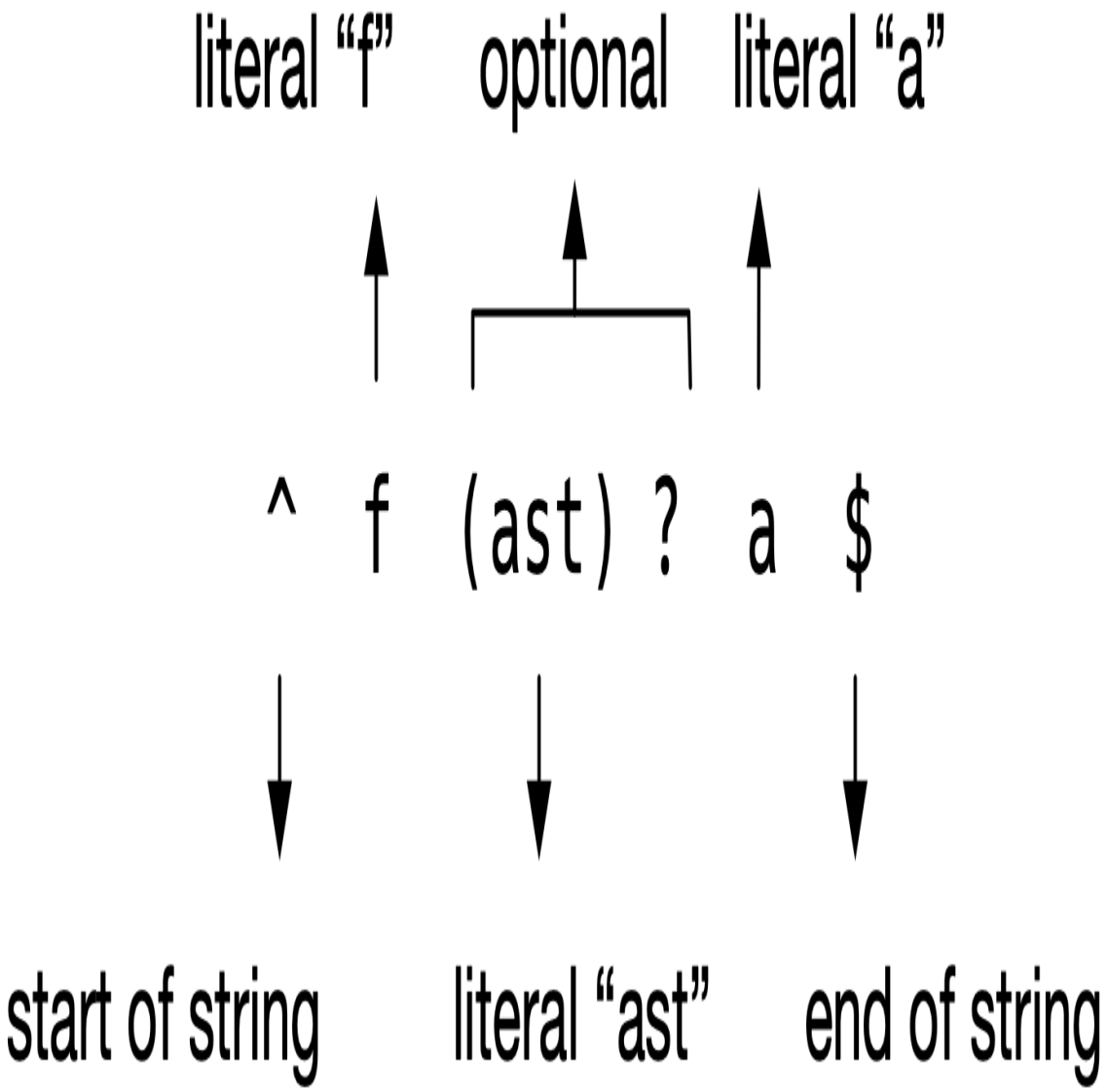


Figure 16-3. A regular expression for matching the two FASTQ patterns.

Figure 16-4 shows the same idea expressed as a finite state machine.

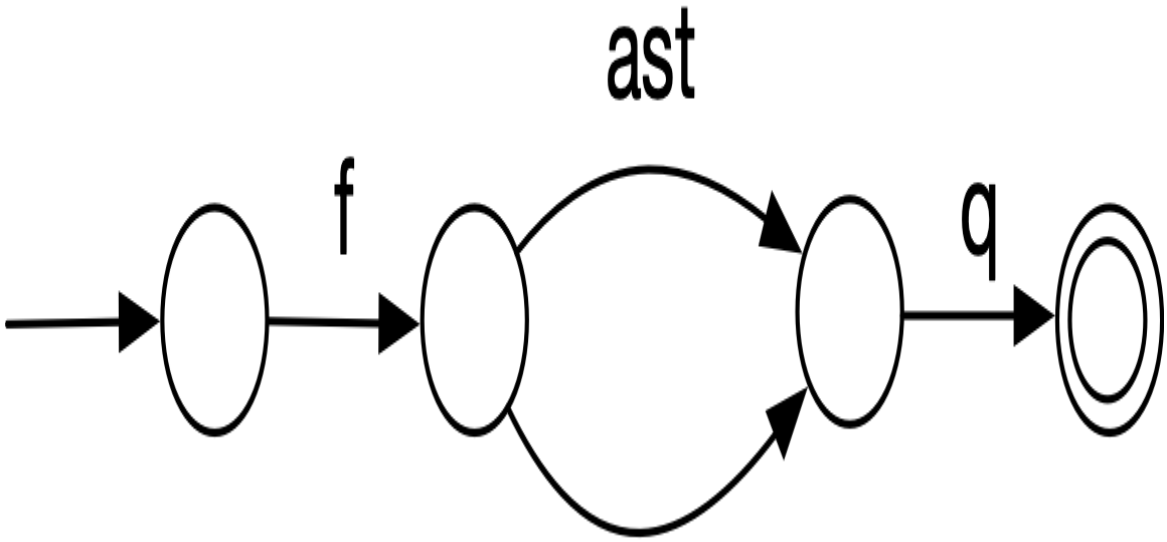


Figure 16-4. A finite state machine diagram for matching the two FASTQ patterns.

I Love It When a Plan Comes Together

Here is how I wrote the `main()` using the structure I introduced in the first part of the chapter:

```
def main() -> None:
    args = get_args()

    def progress(msg: str) -> None: ❶
        if args.verbose:
            print(msg, file=sys.stderr)

    def search(text: str) -> Optional[Match[str]]: ❷
        flag = re.IGNORECASE if args.insensitive else 0
        return re.search(args.pattern, text, flag)

    num_checked, num_took = 0, 0 ❸
    for i, fh in enumerate(args.files, start=1): ❹
        progress(f'{i:3}: {fh.name}') ❺
        input_format = args.input_format or guess_format(fh.name) ❻

        if not input_format: ❼
            sys.exit(f'Please specify file format for "{fh.name}"')

        output_format = args.output_format or input_format ❽
```

```
        for rec in SeqIO.parse(fh, input_format): ❾
            num_checked += 1 ❿
            if any(map(search, [rec.id, rec.description])): ⓫
                num_took += 1 ⓬
                SeqIO.write(rec, args.outfile, output_format)

    outfile = 'STDOUT' if args.outfile == sys.stdout else
args.outfile.name ⓭
    progress(f'Done, checked {num_checked}, wrote {num_took} to "
{outfile}".')
```

❶ Define a function to print optional messages.

❷ Define a function to perform the regex match.

❸ Initialize variables for the number of sequences checked and taken.

❹ Iterate through each input file.

❺ Optionally print the file number and name.

❻ Use the input format or guess it from the filename.

❼ Halt if there is no input format.

❽ Use the output format or use the input format.

❾ Iterate through each sequence in the file.

❿ Increment the number of sequences checked.

⓫ Search the sequence ID or description for the pattern.

⓬ Increment the taken counter and write the sequence to the output file in the correct format.

⓭ Create and optionally print the final status.

The `progress()` and `search()` functions are written as closures inside `main()` so as to reference values from the `args`. In the case of the `progress()`, I only want to print messages if `args.verbose` is `True`:

```
def progress(msg: str) -> None: ❶
    if args.verbose: ❷
        print(msg, file=sys.stderr) ❸
```

❶ The function takes a string and returns `None`.

❷ Check if the user wants verbose output.

❸ Print the given message to `STDERR`.

In the case of the `search()` function, I need to figure out if the search should or should not respect the case:

```
def search(text: str) -> Optional[Match[str]]: ❶
    flag = re.IGNORECASE if args.insensitive else 0 ❷
    return re.search(args.pattern, text, flag) ❸
```

❶ The function takes a string and returns either a `re.Match` object or `None`.

❷ Figure out the optional search flags.

❸ Perform the search on the given text using the correct flag.

I'd also like to take a moment to highlight my use of the `any()` function to search both the record's ID and description fields using this function, but first I need to explain about *and* and *or* bitwise operations.

Combining Regular Expression Search Flags

In the preceding code, I used the flag `re.IGNORECASE` as an optional third argument to the `re.search()` function. Regular expression search flags bear some explaining. I think it's best to start with the documentation from `help(re)`:

```
Each function other than purge and escape can take an optional 'flags'
argument
consisting of one or more of the following module constants, joined by
"|".
A, L, and U are mutually exclusive.
    A  ASCII       For string patterns, make \w, \W, \b, \B, \d, \D
                   match the corresponding ASCII character categories
                   (rather than the whole Unicode categories, which is
the
                   default).
                   For bytes patterns, this flag is the only available
                   behaviour and needn't be specified.
    I  IGNORECASE  Perform case-insensitive matching.
    L  LOCALE      Make \w, \W, \b, \B, dependent on the current
locale.
    M  MULTILINE   "^" matches the beginning of lines (after a
newline)
                   as well as the string.
                   "$" matches the end of lines (before a newline) as
well
                   as the end of the string.
    S  DOTALL      "." matches any character at all, including the
newline.
    X  VERBOSE     Ignore whitespace and comments for nicer looking
RE's.
    U  UNICODE     For compatibility only. Ignored for string patterns
(it
                   is the default), and forbidden for bytes patterns.
```

The program needs to handle both case-sensitive and -insensitive searching. That is, if I search for lowercase *lsu* but the record header has only uppercase *LSU*, I would expect this to fail:

```
>>> type(re.search('lsu', 'This contains LSU'))
<class 'NoneType'>
```

One way to disregard case is to force both the search pattern and string to upper or lowercase:

```
>>> re.search('lsu'.upper(), 'This contains LSU'.upper())
<re.Match object; span=(14, 17), match='LSU'>
```

Another method is to provide an optional flag to the `re.search()` function:

```
>>> re.search('lsu', 'This contains LSU', re.IGNORECASE)
<re.Match object; span=(14, 17), match='LSU'>
```

This can be shortened to `re.I`:

```
>>> re.search('lsu', 'This contains LSU', re.I)
<re.Match object; span=(14, 17), match='LSU'>
```

If I look more closely, I can find that `re.IGNORECASE` is an enum or *enumeration* of possible values:

```
>>> type(re.IGNORECASE)
<enum 'RegexFlag'>
```

According to the documentation, this is “a subclass of enum.IntFlag” which is described thusly:

> *Base class for creating enumerated constants that can be combined using the bitwise operators without losing their IntFlag membership. IntFlag members are also subclasses of int.*

This means that `re.IGNORECASE` is deep down an `int`, just like `False` is actually `0` and `True` is actually `1`. I can use a little detective work to figure out the integer values of the flags by adding 0:

```
>>> for flag in sorted([re.A, re.I, re.L, re.M, re.S, re.X, re.U]):
...     print(f'{flag:15} {flag + 0:5} {0 + flag:#011b}')
...
re.IGNORECASE       2 0b000000010
```

```
re.LOCALE            4 0b000000100
re.MULTILINE         8 0b000001000
re.DOTALL           16 0b000010000
re.UNICODE          32 0b000100000
re.VERBOSE          64 0b001000000
re.ASCII           256 0b100000000
```

Note how each value is a power of 2 so that each flag can be represented by a single, unique bit. This makes it possible to combine flags using the | *bitwise* operator mentioned in the documentation. To demonstrate, I can use the prefix `0b` to represent a string of raw bytes. Here are the binary representations of the numbers one and two. Note that each uses just a single bit set to 1:

```
>>> one = 0b001
>>> two = 0b010
```

If I use | to *or* the bits together, each of the three bits is combined using the truth table shown in Table 16-2.

Table 16-2. Truth table for OR bitwise operations

First	Second	Result
T	T	T
T	F	T
F	T	T
F	F	F

As shown in Figure 16-5, Python will look at each bit and select 1 if either of the bits is 1 and 0 only if both bits are 0 resulting in `0b011` which is the binary representation of the number three because the bits for positions 1 and 2 are both set:

```
>>> one | two
3
```

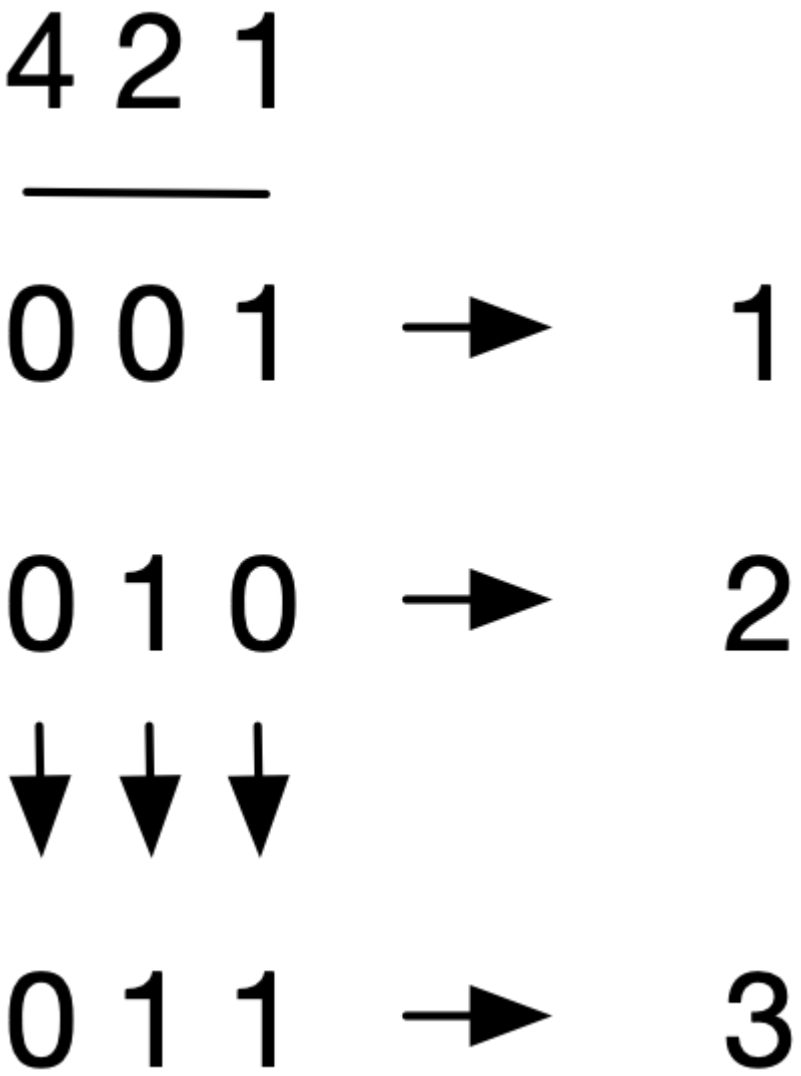


Figure 16-5. When ORing each column of bits, a 1 in any position yields a 1. If all bits are 0, the result 0.

When using the & operator, Python will only yield a 1 when both bits are 1 and 0 otherwise as shown in Table 16-3.

Table 16-3. Truth table for AND bitwise operations

First	Second	Result
T	T	T
T	F	F
F	T	F
F	F	F

This will result in the value `0b000`, which is zero:

```
>>> one & two
0
```

I can use the `|` operator to join multiple flags using bitwise *or*. For example, `re.IGNORECASE` is 2 which is represented as `0b010`, and `re.LOCALE` is 4 which is represented as `0b100`. Bitwise OR combines these as `0b110` which is the number 6:

```
>>> 0b010 | 0b100
6
```

I can verify this is true:

```
>>> (re.IGNORECASE | re.LOCALE) == 6
True
```

Let me bring all this back to the `search()` function I wrote. If the user indicates case-insensitive searching, then I want to execute something like this:

```
>>> re.search('lsu', 'This contains LSU', re.IGNORECASE)
<re.Match object; span=(14, 17), match='LSU'>
```

If they want case-sensitive, I cannot indicate a `None` value. That is, the following code will throw a nasty `TypeError` exception:

```
>>> re.search('lsu', 'This contains LSU', None)
```

One way to avoid this would be to use an `if` statement:

```
def search(text: str) -> Optional[Match[str]]:
    if args.insensitive:
        return re.search(args.pattern, text)

    return re.search(args.pattern, text, re.IGNORECASE)
```

I really dislike this solution as it leads to writing `re.search()` twice, and I usually stick to the *DRY* principle of "Don't Repeat Yourself." I have to put an integer value in that third slot, so the question is what value means *no flag*? It turns out that the number is 0. If I wanted to expand this program to include any of the additional search flags from the documentation, I could use `|` to combine either each flag's value or 0:

```
def search(text: str) -> Optional[Match[str]]:
    flag = re.IGNORECASE if args.insensitive else 0
    return re.search(args.pattern, text, flag)
```

I've talked about the idea of *reducing* multiple values to a single value in Chapters 6 and 12. For instance, I can use addition to reduce a list of numbers to their sum or multiplication to create the product. A list of strings can be concatenated into a single value using the `str.join()` function. Bitwise operations can be applied the same to reduce, for instance, all the regex flags:

```
>>> (re.A | re.I | re.L | re.M | re.S | re.X | re.U) + 0
382
```

Because these flags use unique bits, it's possible to know exactly which flags were used to generate a particular value. That is I can

use the `&` operator to determine if a given bit is on. For instance, earlier I showed how to combine the flags `re.IGNORECASE` and `re.LOCALE` using `|`:

```
>>> flags = re.IGNORECASE | re.LOCALE
```

To see if a given flag is present in the `flags` variable, it will be returned when I *and* it because only the `1` bits present in both values will be returned:

```
>>> flags & re.IGNORECASE
re.IGNORECASE
```

If I *and* a flag that's not present in the combined values, the result is 0:

```
>>> (flags & re.VERBOSE) + 0
0
```

Reducing Booleans Values

Let me bring this back to the `any()` function I used in this program. As with the bitwise combinations of integer values, I can similarly reduce multiple boolean values. That is, here is the same information as Table 16-2 using the `or` operator to combine booleans:

```
>>> True or True
True
>>> True or False
True
>>> False or True
True
>>> False or False
False
```

This is the same as using `any()` with a list of booleans. If *any* of the values is truthy, then the whole expression is `True`:

```
>>> any([True, True])
True
>>> any([True, False])
True
>>> any([False, True])
True
>>> any([False, False])
False
```

Here is the same data as Table 16-3 using `and` to combine booleans:

```
>>> True and True
True
>>> True and False
False
>>> False and True
False
>>> False and False
False
```

This is the same as using `all()`. Only if *all* of the values are truthy will the whole expression be `True`:

```
>>> all([True, True])
True
>>> all([True, False])
False
>>> all([False, True])
False
>>> all([False, False])
False
```

Here is the line of code where I use this idea:

```
if any(map(search, [rec.id, rec.description])):
```

The `map()` function feeds each of the `rec.id` and `rec.description` value to the `search()` function resulting in a list of values that can be interpreted for their truthiness. If any of these is truthy, meaning I

found a match in at least one of the fields, then `any()` will return `True` and so the sequence should be written to the output file.

Going Further

- Sometimes the sequence header contains key/value metadata like "Organism=*Oryza sativa*". Add an option to search these values. Be sure you add an input file example to the *tests/inputs* directory and the appropriate tests to *tests/fastx_grep_test.py*.
- Expand to program handle additional input file formats like GenBank, EMBL, SwissProt, etc. Again, be sure to add example files and tests to ensure your program works.
- Alter the program to select sequences with some minimum length and quality score.

Review

- The FASTQ file format requires each record to be represented by four lines: a header, the sequence, a separator, and the quality scores.
- Regular expression matches can accept flags that control, for example, whether to perform a case-insensitive match. By default, regexes are case-sensitive.
- To indicate multiple regex flags, use the `|` (or) bitwise operator to combine the integer values of the flags.
- Boolean values can be reduced using `and` and `or` operations as well as the `any()` and `all()` functions.
- The DWIM (Do What I Mean) aesthetic means you try to anticipate what your user would want a program to do

naturally and intelligently

- The DRY (Don’t Repeat Yourself) principle means that you never duplicate the same idea in your code but rather isolate it to one locus or function.

---

1 Possibly this is short for *global regular expression print*.

2 According to *Programming Perl* (O’Reilly, 2012), the three great virtues of a programmer are laziness, impatience, and hubris.

Chapter 17. DNA Synthesizer: Creating Synthetic Data with Markov Chains

A Markov chain is a model for representing a sequence of possibilities found in a given data set. It is a machine learning (ML) algorithm because it discovers or learns patterns from input data. In this exercise, I'll show how to use Markov chains trained on a set of DNA sequences to generate novel DNA sequences.

In this exercise, you will:

- Read some number of input sequence files to find all the unique k-mers for a given *k*.
- Create a Markov chain using these k-mers to produce some number of novel sequences of lengths bounded by a minimum and maximum.
- Learn about generators.
- Use a random seed to replicate random selections.

Understanding Markov Chains

In Claude Shannon's "A Mathematical Theory of Communication" (1948), the author describes a "Markoff process" that is surprisingly similar to graphs and the finite state diagrams I've been using to illustrate regular expressions. Shannon describes this process as "a finite number of possible *states* of a system" and "a set of transition probabilities" that one state will lead to another.

For one example of a Markov process, Shannon describes a system for generating strings of text by randomly selecting from the 26 letters of the English alphabet and a space. In a “zero-order approximation,” each character has an equal probability of being chosen. This process generates strings where letter combinations like “BZ” and “QR” might appear as frequently as “ST” and “QU.” An examination of actual English words, however, would show that the latter two or orders of magnitude more common than the first two:

```
$ for LETTERS in bz qr st qu
> do echo -n $LETTERS && grep $LETTERS /usr/share/dict/words | wc -l;
done
bz       4
qr       1
st   21433
qu    3553
```

To more accurately model the possible transition from one letter to another, Shannon introduces a “first-order approximation … obtained by choosing successive letters independently but each letter having the same probability that it has in the natural language.” For this model, I need to train the selection process on representative texts of English. Shannon notes that the letter *E* has a probability of 0.12 reflecting the frequency of its use in English words whereas *W*, being much less frequently used, has a probability of 0.02, as shown in Figure 17-1.

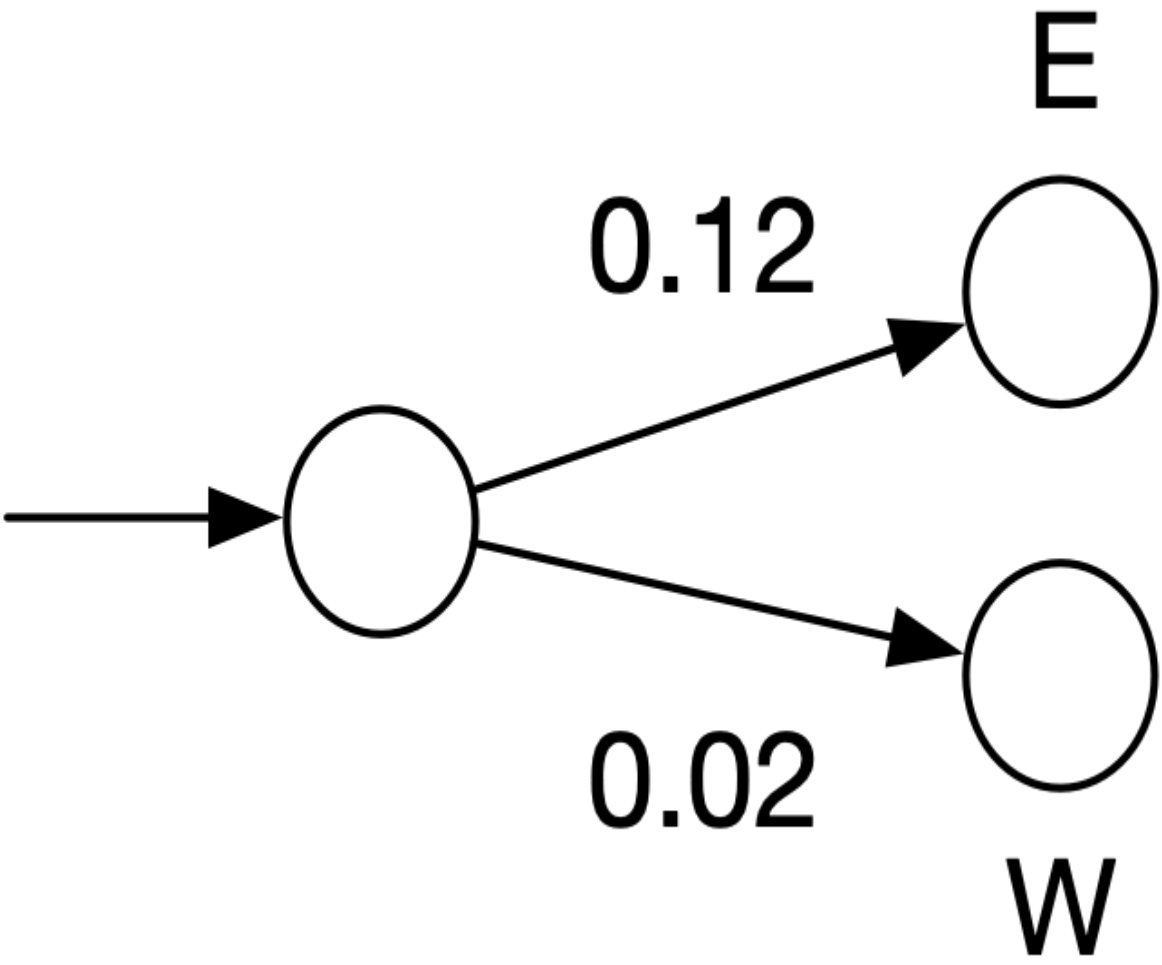


Figure 17-1. A finite state diagram that includes the probability of moving from any character in English to the letters “E” or “W.”

Shannon goes on to describe a “second-order approximation” where subsequent letters are “chosen in accordance with the frequencies with which the various letters follow the first one.” This brings in k-mers that I’ve used several times in Part One. In linguistics, these are called N-grams. For instance, what possible 3-mers could be created given the 2-mer “TH”? The letters “E” or “R” would be rather likely while “Z” would be impossible as no English word contains the sequence “THZ.”

I can perform a rough estimation of how often I can find these patterns. I find approximately 236K English word using **`wc -l`** to count the lines of my system dictionary:

```
$ wc -l /usr/share/dict/words
  235886 /usr/share/dict/words
```

To find the frequency of the substrings, I need to account for the fact that some words may have the pattern twice. For instance, here are a few words that have more than one occurrence of the pattern "THE":

```
$ grep -E '.*the.*the.*' /usr/share/dict/words | head -3
diathermotherapy
enthelminthes
hyperthermesthesia
```

I can use **`grep -io`** to search in a case-insensitive fashion (`-i`) for the strings "THR" and "THE" while the `-o` flag tells `grep` to return *only* the matching strings which will reveal all the matches in each word. I find that "THR" occurs 1270 times, while "THE" occurs 3593 times:

```
$ grep -io thr /usr/share/dict/words | wc -l
    1270
$ grep -io the /usr/share/dict/words | wc -l
    3593
```

Dividing these numbers by the total number of words leads to a frequency of 0.005 for "THR" and 0.015 for "THE," as shown in Figure 17-2.

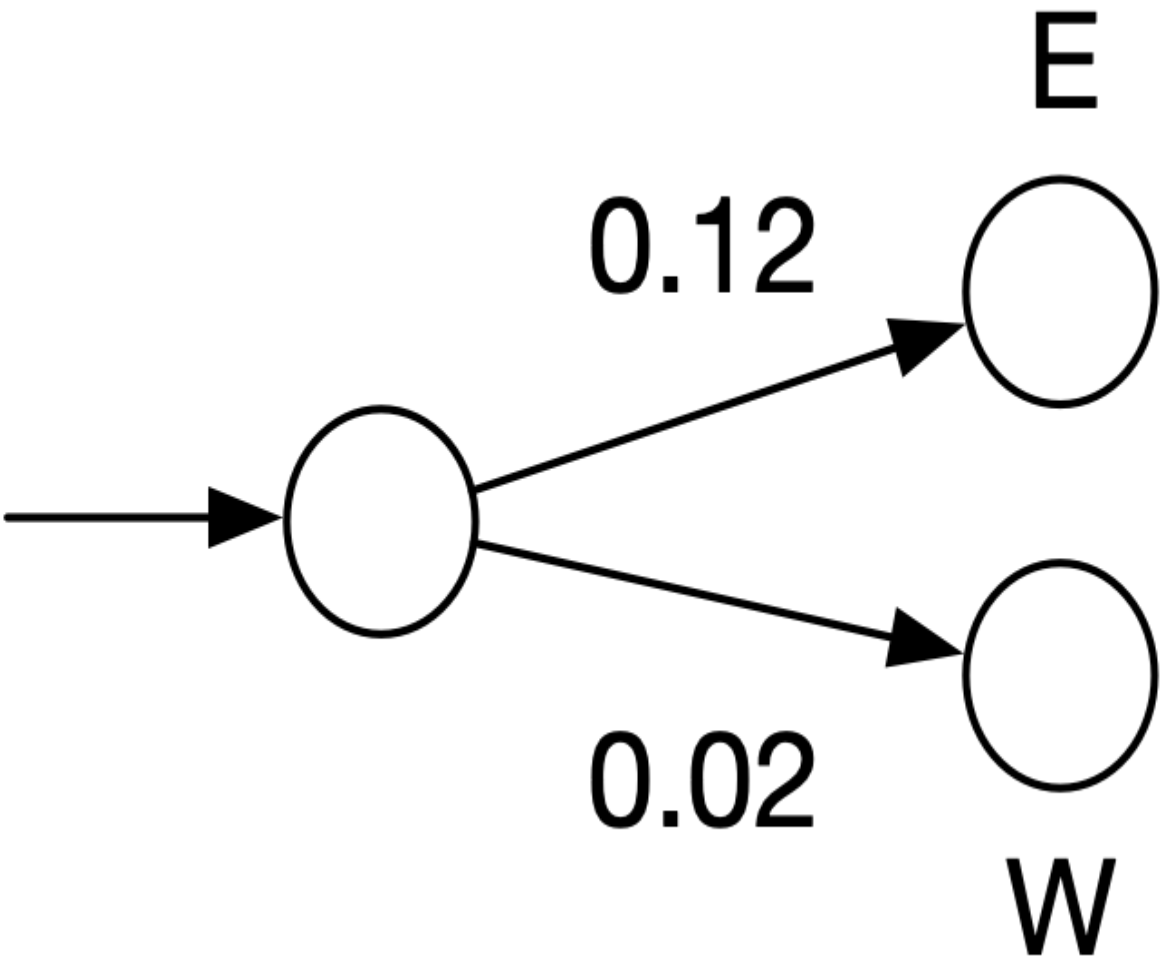


Figure 17-2. A finite state diagram showing the probability of moving from "TH" to either an "R" or an "E."

I can apply these ideas to generate novel DNA sequences by reading some sample sequences and noting the ordering of the bases at the level of some k-mer like 10 base pairs. The number of possible 10-mers using the 4 bases A, C, T, and G is 4^{10} or 1,048,576. That's a very large number, but the number of actual 10-mers found in a real sample is likely to be far less.

TODO: How many unique 10-mers in the human genome?

It's important to note that different training texts will affect the model. For instance, English words and spellings have changed over time,

so training on older English texts like *Beowulf* and *Canterbury Tales* will yield different results than articles from modern newspapers. This is the *learning* part of machine learning. Many ML algorithms are designed to find patterns from some sets of data to apply to another set. In the case of this program, the generated sequences will bear some resemblance in composition to the input sequences. Running the program with the human genome as training data will produce different results from using a viral metagenome from an oceanic hydrothermal flume.

Getting Started

You should work in the *17_synth* directory containing the inputs and tests for this program. Start by copying the solution to the program `synth.py`:

```
$ cd 17_synth
$ cp solution.py synth.py
```

This program has a large number of parameters. Run the help to see them:

```
$ ./synth.py -h
usage: synth.py [-h] [-o FILE] [-f format] [-n number] [-x max] [-m
min]
                [-k kmer] [-s seed]
                FILE [FILE ...]

Create synthetic DNA using Markov chain

positional arguments:
  FILE                  Training file(s) ❶

optional arguments:
  -h, --help            show this help message and exit
  -o FILE, --outfile FILE
                        Output filename (default: out.fa) ❷
  -f format, --format format
```

```
                        Input file format (default: fasta) ❸
  -n number, --num number
                        Number of sequences to create (default: 100) ❹
  -x max, --max_len max
                        Maximum sequence length (default: 75) ❺
  -m min, --min_len min
                        Minimum sequence length (default: 50) ❻
  -k kmer, --kmer kmer  Size of kmers (default: 10) ❼
  -s seed, --seed seed  Random seed value (default: None) ❽
```

❶ The only required parameter is one or more input files.

❷ The output filename will default to *out.fa*.

❸ The input format should be either "fasta" or "fastq" and defaults to the first.

❹ The default number of sequences generated will be 100.

❺ The default maximum sequence length is 75 bp.

❻ The default minimum sequence length is 50 bp.

❼ The default k-mer length is 10 bp.

❽ The default random seed is the `None` value.

As usual, I create an `Args` class to represent these parameters. I use the following `typing` imports. Note the `Dict` is used later in the program:

```
from typing import NamedTuple, List, TextIO, Dict, Optional

class Args(NamedTuple):
    files: List[TextIO] ❶
    outfile: TextIO ❷
    file_format: str ❸
    num: int ❹
    min_len: int ❺
```

```
    max_len: int ❻
    k: int ❼
    seed: Optional[int] ❽
```

❶ The input `files` will be a list of open filehandles.

❷ The `outfile` will be an open filehandle.

❸ The `file_format` of the input files is a string like “fasta” or “fastq.”

❹ The `num` number of sequences to generate is an integer.

❺ The `min_len` is an integer.

❻ The `max_len` is an integer.

❼ The `k` for k-mer length is an integer.

❽ The random seed can either be the `None` value or an integer.

Here is how I define the program’s parameters:

```
def get_args() -> Args:
    """ Get command-line arguments """

    parser = argparse.ArgumentParser(
        description='Create synthetic DNA using Markov chain',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file',
                        help='Training file(s)',
                        metavar='FILE',
                        nargs='+',
                        type=argparse.FileType('rt')) ❶

    parser.add_argument('-o',
                        '--outfile',
                        help='Output filename',
                        metavar='FILE',
                        type=argparse.FileType('wt'), ❷
```

```
                    default='out.fa')

    parser.add_argument('-f',
                        '--format',
                        help='Input file format',
                        metavar='format',
                        type=str,
                        choices=['fasta', 'fastq'], ❸
                        default='fasta')

    parser.add_argument('-n',
                        '--num',
                        help='Number of sequences to create',
                        metavar='number',
                        type=int,
                        default=100) ❹

    parser.add_argument('-x',
                        '--max_len',
                        help='Maximum sequence length',
                        metavar='max',
                        type=int,
                        default=75) ❺

    parser.add_argument('-m',
                        '--min_len',
                        help='Minimum sequence length',
                        metavar='min',
                        type=int,
                        default=50) ❻

    parser.add_argument('-k',
                        '--kmer',
                        help='Size of kmers',
                        metavar='kmer',
                        type=int,
                        default=10) ❼

    parser.add_argument('-s',
                        '--seed',
                        help='Random seed value',
                        metavar='seed',
                        type=int,
                        default=None) ❽
```

```
    args = parser.parse_args()

    return Args(file=args.file,
                outfile=args.outfile,
                file_format=args.format,
                num=args.num,
                min_len=args.min_len,
                max_len=args.max_len,
                k=args.kmer,
                seed=args.seed)
```

❶ The `type` restricts the values to readable text files, and the `nargs` requires one or more values.

❷ The `type` restricts the value to a writable text file, and the default will be *out.fa*.

❸ The `choices` restricts the values to either “fasta” or “fastq,” and the default will be “fasta.”

❹ The `type` restricts this to a valid integer value, and the default is 100.

❺ The `type` restricts this to a valid integer value, and the default is 75.

❻ The `type` restricts this to a valid integer value, and the default is 50.

❼ The `type` restricts this to a valid integer value, and the default is 10.

❽ The `type` restricts this to a valid integer value, and the default is `None`.

It might seem a little odd that the `seed` has `type=int` but has a default of `None` because `None` is not an integer. What I’m saying is

that, if the user provides any value for the seed, it must be a valid integer; otherwise, the value will be `None`. This is also reflected in the `Args.seed` definition as an `Optional[int]` which means the value can either be `int` or `None`. Note that this is equivalent to `typing.Union[int, None]`, the union of the `int` type and `None` value.

Understanding Random Seeds

There is an element of randomness to this program as you generate the sequences. I can start with Shannon's zero-order implementation where I select each base independently at random. I can use the `random.choice()` function to select one base:

```
>>> bases = list('ACGT')
>>> import random
>>> random.choice(bases)
'G'
```

If I wanted to generate a 10-bp sequence, I can use a list comprehension with the `range()` function:

```
>>> [random.choice(bases) for _ in range(10)]
['G', 'T', 'A', 'A', 'C', 'T', 'C', 'T', 'C', 'T']
```

I could further select a random sequence length between some minimum and maximum length using the `random.randint()` function:

```
>>> [random.choice(bases) for _ in range(random.randint(10, 20))]
['G', 'T', 'C', 'A', 'C', 'C', 'A', 'G', 'C', 'A', 'G']
```

If you execute the preceding code on your computer, it's highly unlikely you will see the same output as shown. Fortunately, these selections are only pseudorandom as they are produced deterministically by a random number generator (RNG). Truly random, unreproducible choices would make testing this program impossible.

I can use a *seed* or initial value to force the following pseudorandom selections to be predictable. If you read `help(random.seed)`, you'll see that the "supported seed types are None, int, float, str, bytes, and bytearray." For instance, I can seed using an integer:

```
>>> random.seed(1)
>>> [random.choice(bases) for _ in range(random.randint(10, 20))]
['A', 'G', 'A', 'T', 'T', 'T', 'T', 'C', 'A', 'T', 'A', 'T']
```

I can also use a string:

```
>>> random.seed('markov')
>>> [random.choice(bases) for _ in range(random.randint(10, 20))]
['G', 'A', 'G', 'C', 'T', 'A', 'A', 'C', 'G', 'T', 'C', 'C', 'C', 'G',
'G']
```

If you execute the preceding code, you should get the exact output shown. By default, the random seed is `None`, which you'll notice is the default for the program. This is the same as not setting the seed at all, so when the program runs with the default it will act in a pseudorandom manner. When testing I can provide a value that will produce a known result to verify that the program works correctly.

Note that I have forced the user to provide an integer value. Although I find using integers to be convenient, you can seed using strings, numbers, or bytes when writing your own programs. Just remember that the number `4` and the string `'4'` are two different values and so will produce different results:

```
>>> random.seed(4) ❶
>>> [random.choice(bases) for _ in range(random.randint(10, 20))]
['G', 'A', 'T', 'T', 'C', 'A', 'A', 'A', 'T', 'G', 'A', 'C', 'G']
>>> random.seed('4') ❷
>>> [random.choice(bases) for _ in range(random.randint(10, 20))]
['G', 'A', 'T', 'C', 'G', 'G', 'A', 'G', 'A', 'C', 'C', 'A']
```

❶ Seed using the integer value `4`

❷ Seed using the strin value `'4'`

The random seed affects every call to `random` functions from that point forward. This creates a *global* change to your program, and so should be viewed with extreme caution. Typically I will set the random seed in my program immediately after validating the arguments:

```
def main() -> None:
    args = get_args()
    random.seed(args.seed)
```

If the seed is the default value of `None`, this will have no effect on the `random` functions. If the user has provided a seed, then all subsequent `random` calls will be affected.

Reading the Training Files

The first step in my program is to read the training files. Due to how I defined this argument with `argparse`, the process of validating the input files has been handled, and I know I will have a `List[TextIO]` which is a list of open filehandles. I will use `Bio.SeqIO.parse()` as in previous chapters to read the sequences.

From the training files, I want to produce a dictionary that describes the weighted possible bases that can follow each k-mer. I think it's helpful to use a type alias to define a couple of new types to describe this. First I want a dictionary that maps a base like "A" to a floating-point value between 0 and 1 to describe the probability of choosing this base. I'll call it a `WeightedChoice`:

```
WeightedChoice = Dict[str, float]
```

For instance, in the sequence "ACGTACGC," the 3-mer "ACG" can be followed by either "T" or "C" with equal likelihood. I represent this

like so:

```
>>> choices = {'T': 0.5, 'C': 0.5}
```

Next, I want a type that maps the k-mer "ACG" to the choices. I'll call this a `Chain` as it represents the Markov chain:

```
Chain = Dict[str, WeightedChoice]
```

It would look like this:

```
>>> weighted = {'ACG': {'T': 0.5, 'C': 0.5}}
```

Each k-mer from the sequences in the input file will have a dictionary of weighted options for use in selecting the next base. Here is how I use it to define a function to read the training files:

```
def read_training(fhs: List[TextIO], file_format: str, k: int) ->
Chain: ❶
    """ Read training files, return dict of chains """

    pass ❷
```

❶ The function accepts a list of filehandles, the file format of the files, and the size of the k-mers to read. It returns the type `Chain`.

❷ Use `pass` to do nothing and return `None` for now.

Since k-mers figure prominently into this solution, you may want to use the `find_kmers()` function from Part 1. As a reminder, for a function with this signature:

```
def find_kmers(seq: str, k: int) -> List[str]:
    """ Find k-mers in string """
```

Here is the test I would use:

```
def test_find_kmers() -> None:
    """ Test find_kmers """

    assert find_kmers('ACTG', 2) == ['AC', 'CT', 'TG']
    assert find_kmers('ACTG', 3) == ['ACT', 'CTG']
    assert find_kmers('ACTG', 4) == ['ACTG']
```

I think it's helpful to see exactly what goes into this function and what I expect it to return. In the *tests/unit_test.py* file, you'll find all the unit tests for this program. Here is the test for this function:

```
def test_read_training() -> None: ❶
    """ Test read_training """

    f1 = io.StringIO('>1\nACGTACGC\n') ❷
    assert read_training([f1], 'fasta', 4) == { ❸
        'ACG': { 'T': 0.5, 'C': 0.5 },
        'CGT': { 'A': 1.0 },
        'GTA': { 'C': 1.0 },
        'TAC': { 'G': 1.0 }
    }

    f2 = io.StringIO('@1\nACGTACGC\n+\n!!!!!!!!') ❹
    assert read_training([f2], 'fastq', 5) == {
        'ACGT': { 'A': 1.0 },
        'CGTA': { 'C': 1.0 },
        'GTAC': { 'G': 1.0 },
        'TACG': { 'C': 1.0 }
    }
```

❶ The function takes no arguments and returns `None`.

❷ Define a mock filehandle containing a single sequence in FASTA format.

❸ Read the data in FASTA format and return the Markov chains for 4-mers.

❹ Define a mock filehandle containing a single sequence in FASTQ format.

❺ Read the data in FASTQ format and return the Markov chains for 5-mers.

To understand this, I've included a program called `kmer_tiler.py` that will show you the overlapping k-mers in a given sequence. The first test is that the 3-mer "ACG" is followed by two equal possible choices in "T" or "C" to create the 4-mers "ACGT" and "ACGC." Looking at the output from the `kmer_tiler.py`, I can see these two possibilities:

```
$ ./kmer_tiler.py ACGTACGC -k 4
There are 5 4-mers in "ACGTACGC."
ACGTACGC
ACGT ❶
 CGTA
  GTAC
   TACG
    ACGC ❷
```

❶ "ACG" followed by "T"

❷ "ACG" followed by "C"

Using this information, I can create Shannon's second-order approximation. For instance, if I randomly select the 3-mer "ACG" to start generating a new sequence, I can add either "T" or "C" with equal probability. Given this training data, I could never append either "A" or "G" as these patterns never occur.

This is a tricky function to write, so let me give you some pointers. First, you need to find all the k-mers in all the sequences in all the files. For each k-mer, you need to find all the possible endings for a sequence of length *k* - 1. That is, if *k* is 4, you first find all the 4-mers and then note how the leading 3-mer can be completed with the last base.

I used the `collections.Counter` and end up with an interim data structure that looks like this:

```
{
    'ACG': Counter({'T': 1, 'C': 1}),
    'CGT': Counter({'A': 1}),
    'GTA': Counter({'C': 1}),
    'TAC': Counter({'G': 1})
}
```

Since the input files are all DNA sequences, each k-mer can have at most 4 possible choices. The key to the Markov chain is in giving these values weights, so next, I need to divide each option by the total number of options. In the case of "ACG," there are two possible values each occurring once, so they each get a weight of 1/2 or 0.5. The data structure I return from this function looks like the following:

```
{
    'ACG': {'T': 0.5, 'C': 0.5},
    'CGT': {'A': 1.0},
    'GTA': {'C': 1.0},
    'TAC': {'G': 1.0}
}
```

I recommend you first focus on writing a function that passes this test.

Generating the Sequences

Next, I recommend you concentrate on using the `Chain` to generate new sequences. Here is a stub for your function:

```
def gen_seq(chain: Chain, k: int, min_len: int, max_len: int) ->
Optional[str]: ❶
    """ Generate a sequence """

    return '' ❷
```

❶ The function accepts the `Chain`, the size of the k-mers, and the minimum and maximum sequence length. It might return a new sequence as a string for reasons I'll explain shortly.

❷ For now, return the empty string.

> **NOTE**
>
> When stubbing a function, I interchange `pass` with returning some dummy value. Here I use the empty string since the function returns a `str`. The point is only to create a function that Python parses and which I can use for testing. At this point, I *expect* the function to fail.

Here is the test I wrote for this:

```
def test_gen_seq() -> None: ❶
    """ Test gen_seq """

    chain = { ❷
        'ACG': { 'T': 0.5, 'C': 0.5 },
        'CGT': { 'A': 1.0 },
        'GTA': { 'C': 1.0 },
        'TAC': { 'G': 1.0 }
    }

    state = random.getstate() ❸
    random.seed(1) ❹
    assert gen_seq(chain, k=4, min_len=6, max_len=12) == 'CGTACGTACG'
❺
    random.seed(2) ❻
    assert gen_seq(chain, k=4, min_len=5, max_len=10) == 'ACGTA' ❼
    random.setstate(state) ❽
```

❶ The function accepts no arguments and returns `None`.

❷ This is the data structure returned by the `read_training()` function.

❸ Save the current global state of the `random` module.

❹ Set the state to a known value of `1`.

❺ Verify that the proper sequence is generated.

❻ Set the state to a different known value of `2`.

❼ Verify that the proper sequence is generated.

❽ Restore the `random` module to any previous state.

> **NOTE**
>
> As noted before, calling `random.seed()` globally modifies the state of the `random` module. I use `random.getstate()` to save the current state before modifying and then restore that state when the testing is finished.

This is a tricky function to write, so I'll give you some direction. You will first randomly select the length of the sequence to generate. I'd recommend using the `random.randint()` function. Note that the upper and lower bounds are inclusive:

```
>>> min_len, max_len = 5, 10
>>> import random
>>> seq_len = random.randint(min_len, max_len)
>>> seq_len
9
```

Next, you should initialize the sequence using one of the keys from the Markov `Chain` structure. Note the need to coerce the `list(chain.keys())` in order to avoid the Error "*dict_keys* object is not subscriptable":

```
>>> chain = {
...     'ACG': { 'T': 0.5, 'C': 0.5 },
```

```
...     'CGT': { 'A': 1.0 },
...     'GTA': { 'C': 1.0 },
...     'TAC': { 'G': 1.0 }
... }
>>> seq = random.choice(list(chain.keys()))
>>> seq
'ACG'
```

I decided to set up a loop with the condition that the length of the sequence is less than the chosen sequence length:

```
>>> while len(seq) < seq_len:
...     break
...
```

While inside the loop, I will keep appending bases. To select each new base, I need to get the last *k - 1* bases of the ever-growing sequence which I can get using a list slice and negative indexing:

```
>>> while len(seq) < seq_len:
...     prev = seq[-1 * (k - 1):]
...     print(prev)
...     break
...
ACG
```

If this previous value occurs in the given chain, then I can select the next base using the `random.choices()` function. If you read `help(random.choices)`, you will see that this function accepts a `population` from which to select, `weights` to consider when making the selection, and a `k` for the number of choices to return. The keys of the chain for a given k-mer are the population:

```
>>> opts = chain['ACG']
>>> pop = opts.keys()
>>> pop
dict_keys(['T', 'C'])
```

The values of the chain are the weights:

```
>>> weights = opts.values()
>>> weights
dict_values([0.5, 0.5])
```

Note the need to coerce the keys and values using `list()` and that `random.choices()` always returns a `list` even when you ask for just one, so you'll need to select the first value:

```
>>> from random import choices
>>> next = choices(population=list(pop), weights=list(weights), k=1)
>>> next
['T']
```

I can append this to the sequence:

```
>>> seq += next[0]
>>> seq
'ACGT'
```

The loop repeats until either the sequence is the correct length or I select a previous value that does not exist in the chain. The next time through the loop, the `prev` 3-mer will be "CGT" as these are the last 3 bases in `seq`. It happens that "CGT" is a key in the `chain`, but you may sometimes find that there is no way to continue the sequence because the next k-mer doesn't exist in the `chain`. In this case, you can exit your loop and return `None` from the function, hence the reason that the `gen_seq()` function signature returns an `Optional[str]`. I don't want my function to return sequences that are too short. I recommend that you not move on until this function passes the unit test.

Structuring the Program

Once you can read the training files and then generate a new sequence using the Markov chain algorithm, you are ready to print

the new sequences to the output file. Here is a general outline of how my program works:

```
def main() -> None:
    args = get_args()
    random.seed(args.seed)
    chains = read_training(...)
    seqs = calls to gen_seq(...)
    print each sequence to the output file
    print the final status
```

Note that the program will only generate FASTA output, and each sequence should be numbered from 1 as the ID. That is, your output file should look something like this:

```
>1
GGATTAGATA
>2
AGTCAACG
```

The test suite is pretty large as there are so many options to check. I recommend you run `make test` or read the *Makefile* to see the longer command to ensure you are properly running all the unit and integration tests.

Solution

I have just one solution for this program as it's complicated enough. I'll start with my function to read the training files:

```
def read_training(fhs: List[TextIO], file_format: str, k: int) ->
Chain:
    """ Read training files, return dict of chains """

    counts: Dict[str, Dict[str, int]] = defaultdict(Counter) ❶
    for fh in fhs: ❷
        for rec in SeqIO.parse(fh, file_format): ❸
            for kmer in find_kmers(str(rec.seq), k): ❹
                counts[kmer[:k - 1]][kmer[-1]] += 1 ❺
```

```
    def weight(freqs: Dict[str, int]) -> Dict[str, float]: ❻
        total = sum(freqs.values()) ❼
        return {base: freq / total for base, freq in freqs.items()} ❽

    return {kmer: weight(freqs) for kmer, freqs in counts.items()} ❾
```

❶ Initialize a dictionary to hold the Markov chains.

❷ Iterate through each filehandle.

❸ Iterate through each sequence in the filehandle.

❹ Iterate through each k-mer in the sequence.

❺ Use the prefix of the k-mer as the key into the Markov chain, add to the count of the final base.

❻ Define a function that will turn the counts into weighted values.

❼ Find the total number of bases.

❽ Divide the frequencies of each base by the total.

❾ Use a dictionary comprehension to convert the raw counts into weights.

This uses the `find_kmers()` function from Part One which is

```
def find_kmers(seq: str, k: int) -> List[str]:
    """ Find k-mers in string """

    n = len(seq) - k + 1 ❶
    return [] if n < 1 else [seq[i:i + k] for i in range(n)] ❷
```

❶ The number of k-mers is the length of the sequence minus *k* plus 1.

❷ Use a list comprehension to select all the k-mers from the sequence.

Here is how I wrote the `gen_seq()` function to generate a single sequence:

```
def gen_seq(chains: Chain, k: int, min_len: int, max_len: int) ->
Optional[str]:
    """ Generate a sequence """

    seq = random.choice(list(chains.keys())) ❶
    seq_len = random.randint(min_len, max_len) ❷

    while len(seq) < seq_len: ❸
        prev = seq[-1 * (k - 1):] ❹
        if choices := chains.get(prev): ❺
            seq += random.choices(population=list(choices.keys()), ❻
                                  weights=list(choices.values()),
                                  k=1)[0]
        else:
            break ❼

    return seq if len(seq) >= min_len else None ❽
```

❶ Initialize the sequence to a random choice from the keys of the chain.

❷ Select a length for the sequence.

❸ Execute a loop while the length of the sequence is less than the desired length.

❹ Select the last *k* - 1 bases.

❺ Attempt to get a list of choices for this k-mer.

❻ Randomly choose the next base using the weighted choices.

❼ If we cannot find this k-mer in the chain, exit the loop.

❽ Return the new sequence if it is long enough, otherwise return `None`.

To integrate all these, here is my `main()` function:

```
def main() -> None:
    args = get_args()
    random.seed(args.seed) ❶
    if chain := read_training(args.files, args.file_format, args.k): ❷
        seqs = (gen_seq(chain, args.k, args.min_len, args.max_len) ❸
                for _ in count())

        for i, seq in enumerate(filter(None, seqs), start=1): ❹
            print(f'>{i}\n{seq}', file=args.outfile) ❺
            if i == args.num: ❻
                break

        print(f'Done, see output in "{args.outfile.name}".') ❼
    else:
        sys.exit(f'No {args.k}-mers in input sequences.') ❽
```

❶ Set the random seed.

❷ Read the training files in the given format using the given size *k*. This may fail if the sequences are shorter than *k*.

❸ Create a generator to create the sequences.

❹ Use `filter()` with a predicate of `None` to remove falsey elements from the `seqs` generator. Use `enumerate()` to iterate through the index positions and sequences starting at 1 instead of 0.

❺ Print the sequence in FASTA format using the index position as the ID.

❻ Break out of the loop if enough sequences have been generated.

❼ Print the final status.

❽ Let the user know why no sequences could be generated.

I'd like to take a moment to explain the generator in the preceding code. I use the `range()` function to generate the desired number of sequences. I could have used a list comprehension like so:

```
>>> from solution import gen_seq, read_training
>>> import io
>>> f1 = io.StringIO('>1\nACGTACGC\n')
>>> chain = read_training([f1], 'fasta', k=4)
>>> [gen_seq(chain, k=4, min_len=3, max_len=5) for _ in range(3)]
['CGTACG', 'CGTACG', 'TACGTA']
```

A list comprehension will force the creation of all the sequences before moving to the next line. If I were creating millions of sequences, the program will block here and will likely use a large amount of memory to store all the sequences.

If I replace the square brackets `[]` of the list comprehension with parentheses `()`, then it becomes a lazy generator:

```
>>> seqs = (gen_seq(chain, k=4, min_len=3, max_len=5) for _ in
range(num))
>>> type(seqs)
<class 'generator'>
```

I can still treat this like a list by iterating over the values, but these values are only produced as needed. That means the line to create the generator executes almost immediately and moves on to the `for` loop. Additionally, the program only uses the memory needed to produce the next sequence.

One small problem with using `range()` and the number of sequences is that I know the `gen_seq()` function may sometimes return `None` to indicate that random choices lead down a chain that didn't produce a

long enough sequence. I need to the generator to run with no upper limit, and I'll write code to stop requesting sequences when enough have been generated. I can use `itertools.count()` to create an infinite sequence, and I use `filter()` with a predicate of `None` to remove falsey elements:

```
>>> seqs = ['ACGT', None, 'CCCGT']
>>> list(filter(None, seqs))
['ACGT', 'CCCGT']
```

I can run the final program to create an output file using the defaults:

```
$ ./synth.py tests/inputs/*
Done, see output in "out.fa".
```

and then use `seqmagique.py` from Chapter 15 to verify that it generated the correct number of sequences in the expected ranges:

```
$ ../15_seqmagique/seqmagique.py out.fa
name      min_len    max_len    avg_len    num_seqs
out.fa         50         75      63.56         100
```

Flippin' sweet.

Going Further

- Add a sequence `--type` option to produce either DNA or RNA.

Now that you understand Markov chains, you might be interested to see how they are used elsewhere in bioinformatics. For instance, the HMMER tool uses hidden Markov models to find homologs in sequence databases and to create sequence alignments.

Review

- Random seeds are used to replicate pseudorandom selections.
- Markov chains can be used to encode the probabilities that a node in a graph can move to another node or state.
- List comprehensions can be made into a lazy generator by replacing the square brackets with parentheses.

1. Preface
    a. Who Should Read This?
    b. Programming Style: Why I Avoid OOP and Exceptions
    c. Structure
    d. Test-Driven Development
    e. Using the Command Line and Installing Python
    f. Getting the Code and Tests
    g. Installing Modules
    h. Installing the new.py Program
    i. Why Did I Write This Book?
    j. Acknowledgments
    k. Conventions Used in This Book
    l. Using Code Examples
    m. O'Reilly Online Learning
    n. How to Contact Us
2. I. The Rosalind.info Challenges
3. 1. Tetranucleotide Frequency: Counting Things
    a. Getting Started
        i. Creating the Program Using new.py
        ii. Using argparse
        iii. Tools for Finding Errors in Your Code

iv. Introducing Named Tuples

v. Adding Types to Named Tuples

vi. Representing the Arguments with a NamedTuple

vii. Reading Input from the Command Line or a File

viii. Testing Your Program

ix. Running the Program to Test the Output

b. Solution 1: Iterating and Counting the Characters in a String

i. Counting the Nucleotides

ii. Writing and Verifying a Solution

c. Solutions

i. Solution 2: Creating a count Function and Adding a Unit Test

ii. Solution 3: Using str.count

iii. Solution 4: Using a Dictionary to Count all the Characters

iv. Solution 5: Count Only the Desired Bases

v. Solution 6: Using collections.defaultdict

vi. Solution 7: Using collections.Counter

d. Going Further

e. Review

4. 2. Transcribing DNA into mRNA: Mutating Strings, Reading and Writing Files
    a. Getting Started
        i. Defining the Program's Parameters
        ii. Defining an Optional Parameter
        iii. Defining One or More Required Positional Parameters
        iv. Using nargs to Define the Number of Arguments
        v. Using argparse.FileType to Validate File Arguments
        vi. Defining the Args Class
        vii. Outlining the Program Using Pseudocode
        viii. Iterating the Input Files
        ix. Creating the Output Filenames
        x. Opening the Output Files
        xi. Writing the Output Sequences
        xii. Printing the Status Report
        xiii. Using the Test Suite
    b. Solutions
        i. Solution 1: Using str.replace
        ii. Solution 2: Using re.sub
    c. Benchmarking
    d. Going Further

    e. Review

5. 3. Complementing a Strand of DNA: String Manipulation
    a. Getting Started
        i. Iterating over a Reversed String
        ii. Creating a Decision Tree
        iii. Refactoring
    b. Solutions
        i. Solution 1: Using a for Loop and Decision Tree
        ii. Solution 2: Using a Dictionary Lookup
        iii. Solution 3: Using a List Comprehension
        iv. Solution 4: Using str.translate
        v. Solution 5: Using Bio.Seq
    c. Review
6. 4. Creating the Fibonacci Sequence: Writing, Testing, and Benchmarking Algorithms
    a. Getting Started
        i. An Imperative Approach
    b. Solutions
        i. Solution 1: An Imperative Solution Using a List as a Stack
        ii. Solution 2: Creating a Generator Function

iii. Solution 3: Using Recursion and Memoization

c. Benchmarking the Solutions

d. Testing the Good, the Bad, and the Ugly

e. Running the Test Suite on All the Solutions

f. Going Further

g. Review

7. 5. Computing GC Content: Parsing FASTA and Analyzing Sequences

a. Getting Started

b. Parsing FASTA using Biopython

c. Iterating the Sequences Using a for Loop

d. Solutions

i. Solution 1: Using a List

ii. Solution 2: Type Annotations and Unit Tests

iii. Solution 3: Keeping a Running Max Variable

iv. Solution 4: Using a List Comprehension with a Guard

v. Solution 5: Using the filter Function

vi. Solution 6: Using the map Function and Summing Booleans

vii. Solution 7: Using Regular Expressions to Find Patterns

viii. Solution 8: A More Complex find_gc Function

e. Benchmarking

f. Going Further

g. Review

8. 6. Finding the Hamming Distance: Counting Point Mutations

a. Getting Started

i. Reading the Input

ii. Iterating the Characters of Two Strings

b. Solutions

i. Solution 1: Iterate and Count

ii. Solution 2: Creating a Unit Test

iii. Solution 3: Using the zip Function

iv. Solution 4: Using the zip_longest Function

v. Solution 5: Using a List Comprehension

vi. Solution 6: Using the filter Function

vii. Solution 7: Using the map Function with zip_longest

viii. Solution 8: Using the starmap and operator.ne Functions

c. Going Further

d. Review

9. 7. Translating RNA into Protein: More Functional Programming

    a. Getting Started
        i. K-mers and Codons
        ii. Translating Codons
    b. Solutions
        i. Solution 1: Using a for Loop
        ii. Solution 2: Adding Unit Tests
        iii. Solution 3: Another Function and a List Comprehension
        iv. Solution 4: Functional Programming with the map, partial, and takewhile Functions
        v. Solution 5: Using Bio.Seq
    c. Going Further
    d. Review
10. 8. Find a Motif in DNA: Exploring Sequence Similarity
    a. Getting Started
        i. Finding Subsequences
    b. Solutions
        i. Solution 1: Using the str.find Method
        ii. Solution 2: Using the str.index Method
        iii. Solution 3: A Purely Functional Approach
        iv. Solution 4: Using kmers
        v. Solution 5: Finding Overlapping Patterns Using Regular Expressions

    c. Going Further
    d. Review
11. 9. Overlap Graphs: Sequence Assembly Using Shared K-mers
    a. Getting Started
        i. Managing Runtime Messages with STDOUT, STDERR, and Logging
        ii. Finding Overlaps
        iii. Grouping Sequences by the Overlap
    b. Solutions
        i. Solution 1: Using Set Intersections to Find Overlaps
        ii. Solution 2: Using a Graph to Find All Paths
    c. Going Further
    d. Review
12. 10. Finding the Longest Shared Subsequence: Finding K-mers, Writing and Functions, and Using Binary Search
    a. Getting Started
        i. Finding the Shortest Sequence in a FASTA File
        ii. Extracting k-mers From a Sequence
    b. Solutions
        i. Solution 1: Counting Frequencies of k-mers

        ii. Solution 2: Speeding Things Up with a Binary Search

    c. Going Further

    d. Review

13. 11. Finding a Protein Motif: Fetching Data and Using Regular Expressions

    a. Getting Started

        i. Downloading Sequences Files on the Command Line

        ii. Downloading Sequences Files with Python

        iii. Writing a Regular Expression to Find the Motif

    b. Solutions

        i. Solution 1: Using a Regular Expression

        ii. Solution 2: Writing a Manual Solution

    c. Going Further

    d. Review

14. 12. Inferring mRNA from Protein: Products and Reductions of Lists

    a. Getting Started

        i. Creating the Product of Lists

    b. Solutions

        i. Solution 1: Using a Dictionary for the RNA Codon Table

        ii. Solution 2: Turn the Beat Around
        iii. Solution 3: Encode the Minimal Information
    c. Going Further
    d. Review
15. 13. Location Restriction Sites: Using, Testing, and Sharing Code
    a. Getting Started
        i. Finding All Subsequences Using k-mers
        ii. Finding All Reverse Complements
        iii. Putting It All Together
    b. Solutions
        i. Solution 1: Using the zip and enumerate Functions
        ii. Solution 2: Using the operator.eq Function
        iii. Solution 3: Writing a revp Function
        iv. Testing the Program
    c. Going Further
    d. Review
16. 14. Finding Open Reading Frames
    a. Getting Started
        i. Translating Proteins Inside Each Frame Shift
        ii. Finding the ORFs in a Protein Sequence
    b. Solutions

        i. Solution 1: Using the str.index Function
        ii. Solution 2: Using the str.partition Function
        iii. Solution 3: Using a Regular Expression
    c. Going Further
    d. Review
17. II. Other Programs
18. 15. Seqmagique: Creating and Formatting Reports
    a. Using Seqmagick to Analyze Sequence Files
    b. Check Files Using MD5 Hashes
    c. Getting Started
        i. Formatting Text Tables Using tabulate
    d. Solutions
        i. Solution 1: Formatting with tabulate
        ii. Solution 2: Formatting with rich
    e. Going Further
    f. Review
19. 16. FASTX Grep: Creating a Utility Program to Select Sequences
    a. Finding Lines in a File Using grep
    b. The Structure of a FASTQ Record
    c. Getting Started
    d. Guessing the File Format

    e. Solution
        i. Guessing the File Format from the File Extension
        ii. I Love It When a Plan Comes Together
        iii. Combining Regular Expression Search Flags
        iv. Reducing Booleans Values
    f. Going Further
    g. Review
20. 17. DNA Synthesizer: Creating Synthetic Data with Markov Chains
    a. Understanding Markov Chains
    b. Getting Started
    c. Understanding Random Seeds
    d. Reading the Training Files
    e. Generating the Sequences
    f. Structuring the Program
    g. Solution
    h. Going Further
    i. Review