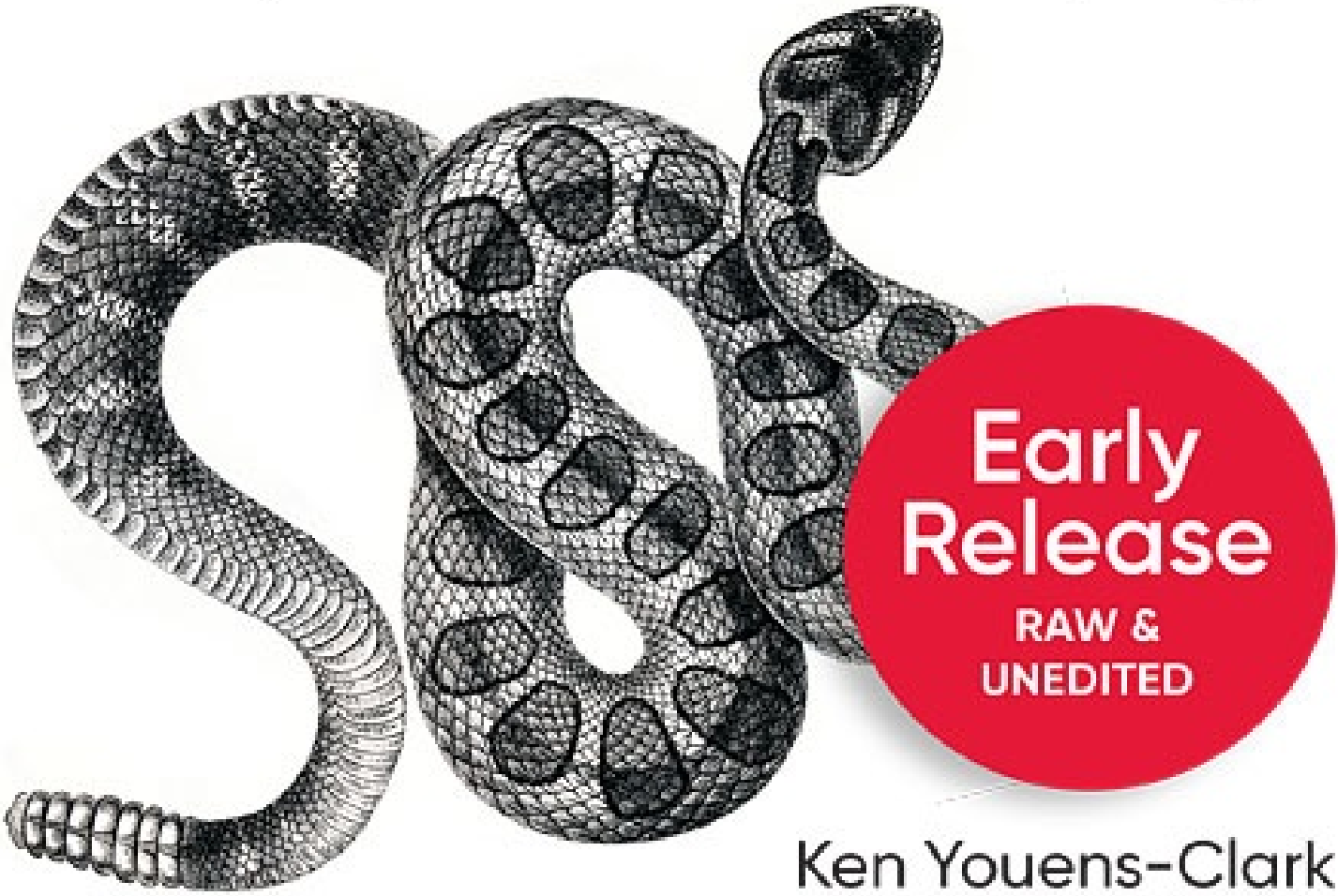
O'REILLY®

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 Documented, Tested Programs for Biology

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

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 .

Acquisitions Editor: Michelle Smith

Editors: Corbin Collins

Production Editor: Caitlin Ghegan

Copyeditor: TK

Proofreader: TK

Indexer: TK

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

February 2022: First Edition

Revision History for the Early Release

- 2020-12-14: First 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, allows one to mix types like characters and numbers in operations that will produce runtime exceptions.

This book shows how to avoid many potential problems in Python by using certain best practices and tools:

- Since Python 3.6, you can add type hints to indicate, for instance, that a variable should be *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 for others 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 in 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 on their own or programmers who've learned biology on their own (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. That book also introduces testing and using 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, they will never rely on some hidden, mutable state like global variables, and they will always return the same values given the same arguments. Every function will have a test that passes to ensure the code 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.

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. The second part covers several programs that show you how to accomplish many basic tasks I've encountered.

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

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, or Notepad++.

I wrote and tested all the examples using Python versions 3.8.6 and 3.9.0. 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.

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:

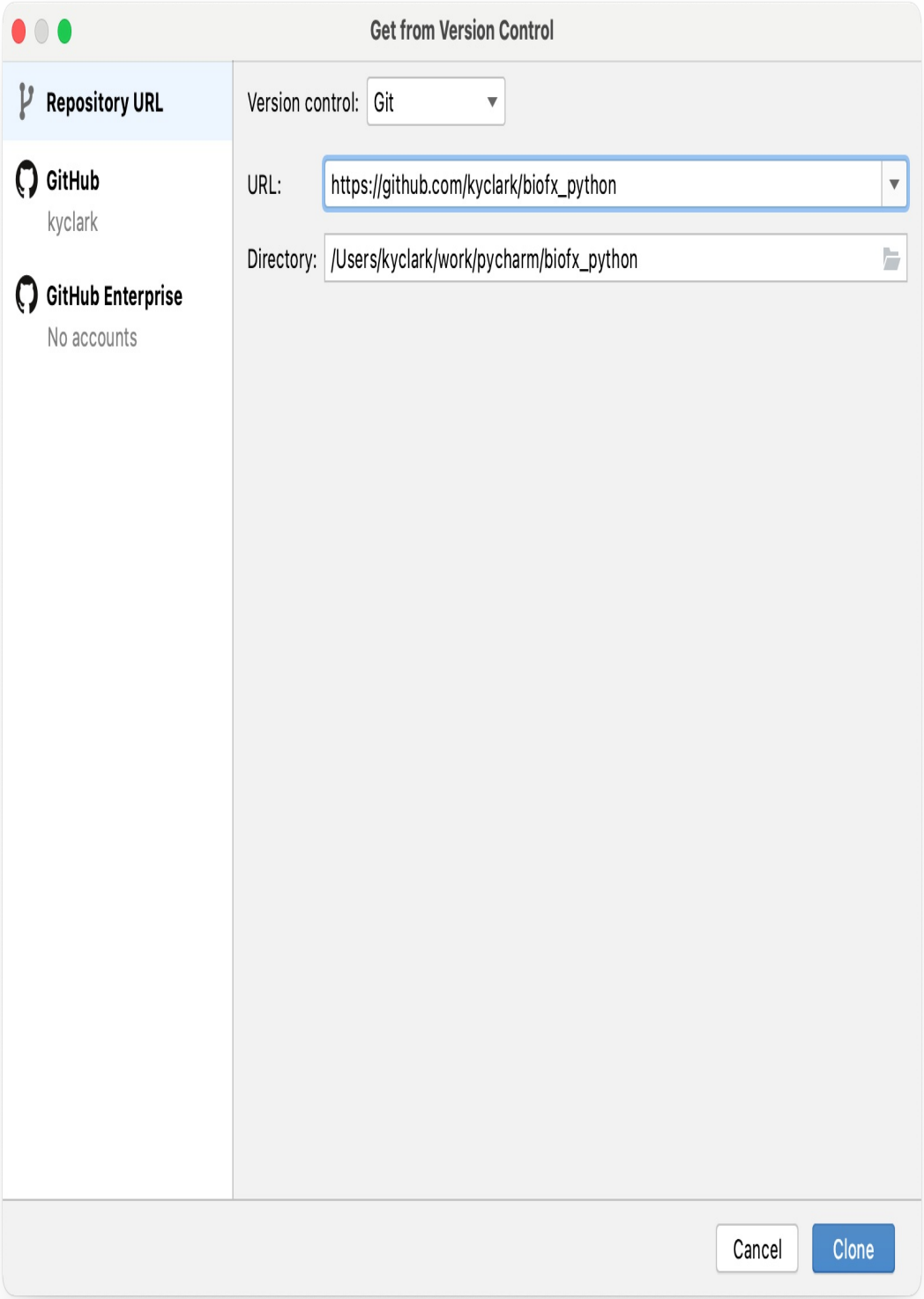

Get from Version Control
Repository URL
GitHub
kyclark
GitHub Enterprise
No accounts
Version control:
Git
URL:
https://github.com/kyclark/biofx_python
Directory:
/Users/kyclark/work/pycharm/biofx_python
Cancel
Clone

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.

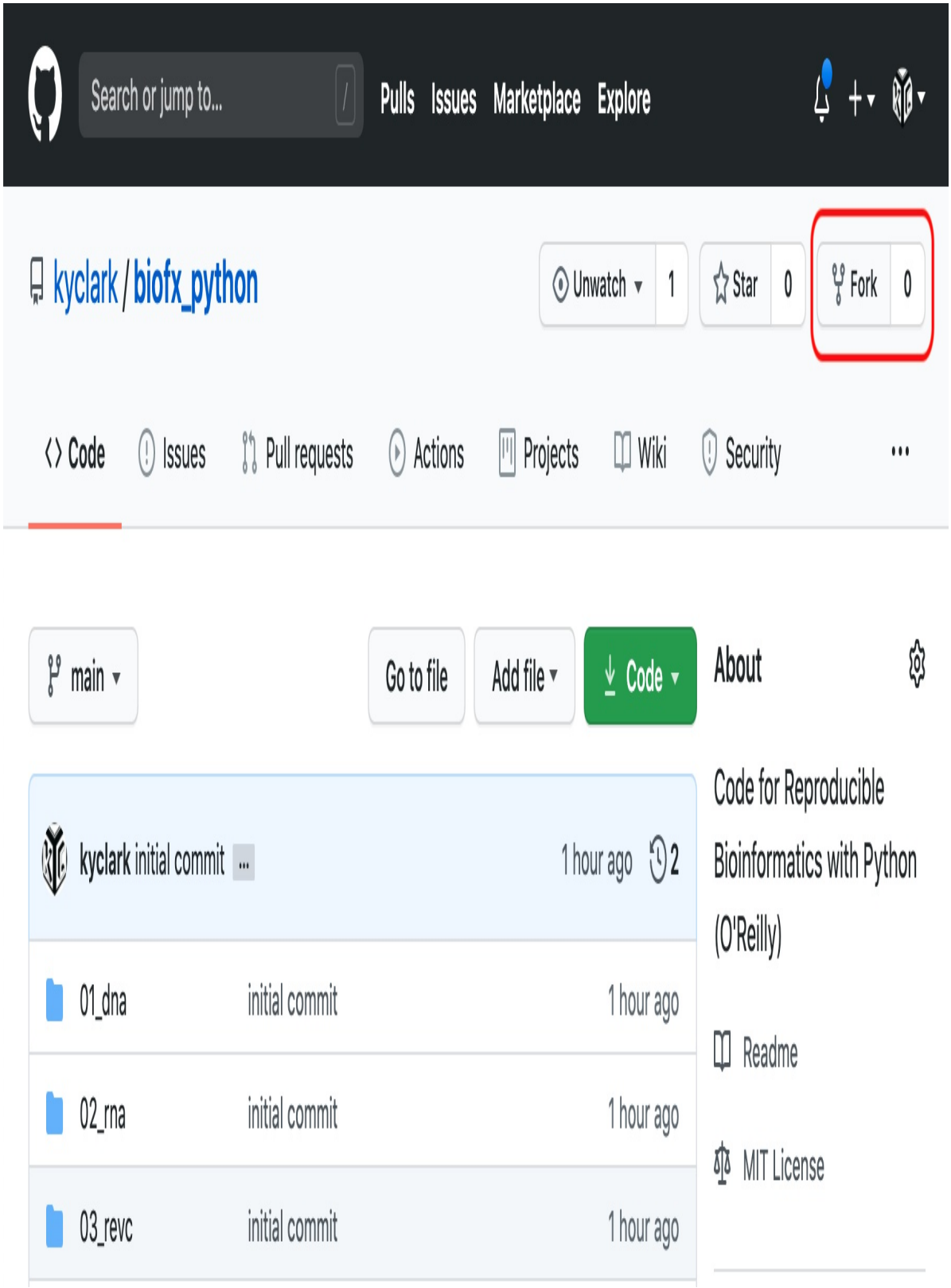


Figure P-2. The Fork button on my GitHub repository will make a copy of the code into your account.

Now that you have a copy of my all 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.

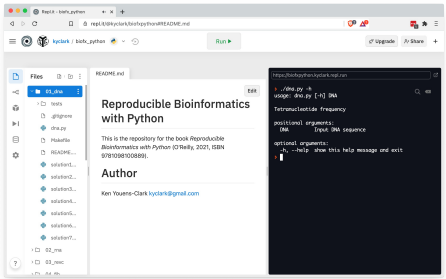


Figure P-3. The Repl.it website can import your repository and run it on a virtual machine all from a browser.

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
```

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. Every program in this book will have the same predictable structure which is generated by this program. To install it, I suggest you clone my GitHub repository:

```
$ git clone https://github.com/kyclark/new.py
```

Inside that directory is the file `new.py` which you should copy to a location in your `PATH`. If you are not familiar with `PATH`, then today is your lucky day to learn about it! The `PATH` is a variable that defines the directories that will be searched for a given program. That is if I type the command **`foo`** and there's no shell built-in command, shell function, command alias, or program the shell can execute as **`foo`**, then I'll be told this command cannot be found:

```
$ foo
-bash: foo: command not found
```

In Windows PowerShell, I can inspect the `PATH` with **`echo $env:Path`**, while on Unix platforms I use the command **`echo $PATH`**. Both paths are printed as a long string with no spaces listing all the directory names separated by semicolons on Windows or by colons on Unix. If the operating system didn't have some concept of a *path*, it would have to search *every directory* on a machine for a given command. This could take minutes to hours, so it makes sense to restrict the searching to just a few directories.

Following is my path on my Mac. Note that I have to put a dollar sign (`$`) in front to tell my shell (`bash`) that this is a variable and not the literal string *PATH*. To make this more readable, I'll use Perl to replace the colons with newlines. Note that this command will only work on a Unix command line where Perl is installed:

```
$ echo $PATH | perl -pe 's/:/\n/g' ❶
/Users/kyclark/.local/bin ❷
/Library/Frameworks/Python.framework/Versions/3.8/bin ❸
/usr/local/bin ❹
/usr/bin ❺
/bin
/usr/sbin
/sbin
```

❶ The Perl substitute `s//` command replaces the first pattern (`:`) with the second `\n` globally `g`.

❷ This is a custom directory I usually create for installing my own programs.

❸ This is where Python installed itself.

❹ A standard directory for user-installated software.

❺ The rest are more standard directories for finding programs.

> **NOTE**
>
> The directories will be searched in the order they are defined, so the order can be quite important. For instance, the Python path is listed before system paths so that when I type **`python3`** it will use the one found in my local Python directory *before* one that might have been pre-installed on my system.

Notice that all the directory names in my `PATH` end in *bin*. This is short for *binaries* and comes from the fact that many programs exist in a binary form. That is, the source code for a C program is written in a pseudo-English language that is compiled into a machine-readable executable file. The contents of this file are binary-encoded instructions that the operating system can execute.

Python programs, by contrast, are usually installed as their source code files which are executed by Python at runtime. I suggest you copy the `new.py` program to one of the directories that are already listed in your `PATH`. For instance, `/usr/local/bin` is a good place, especially if you are working on your own personal machine like a laptop where you have administrator privileges:

```
$ cp new.py/new.py /usr/local/bin
```

You may not have sufficient permissions to do this, however. Unix systems were designed from the beginning to be *multi-tenant* operating systems, meaning that they support many different people using the system concurrently. It's important to keep users from writing and deleting files they shouldn't, and so the OS may prevent you from writing `new.py` to a directory that you don't own. If, for instance, you are working on a shared HPC system at a university, you certainly won't have such privileges.

When you cannot install into system directories, it's easiest to create a location in your `HOME` directory for such files. On my laptop, this is my `HOME` directory:

```
$ echo $HOME
/Users/kyclark
```

On almost all my systems, I create a *$HOME/.local* directory for installing programs. Most shells interpret the tilde (~) as `HOME`:

```
$ mkdir ~/.local
```

> **NOTE**
>
> By convention, files and directories that have names starting with a dot are normally hidden by the `ls` command. You can use **`ls -a`** to list *all* the contents of a directory. You may notice many other *dotfiles* that are used by various programs to persist options and program state. I like to call this *.local* so I won't normally see it in my directory listing.

Creating a directory in your `$HOME` for software installations is especially useful when compiling programs from source, a very common operation in bioinformatics. Most installations of this sort begin by using a `configure` program to gather information about your system such as the location of your C compiler and such. This program almost always has a `--prefix` option that I'll set to this directory:

```
$ ./configure --prefix=$HOME/.local
```

The resulting installation will put the binary compiled files into *$HOME/.local/bin*. It might also install header files and manual pages and other supporting data into other directories in *$HOME/.local*.

Wherever you decide to install local programs, you'll need to ensure your `PATH` is updated to search in that directory in addition to the others. I tend to use the `bash` shell, and one of the dotfiles in my `HOME` is a file called `.bashrc` (or sometimes `.bash_profile` or even `.profile`). I can add this line to put my custom directory first in the `PATH`:

```
export PATH=$HOME/.local/bin:$PATH
```

You may need something slightly different if you are using a different shell. Recently macOS started using `zsh` (Z shell) as the default shell, or your HPC might use another shell. They all have the idea of `PATH` and all allow you to customize this in some manner. On Windows, you can use this command to

append the directory to your path:

```
> $env:Path += ";~/.local/bin"
```

Here is how I can make the directory and copy the program:

```
$ mkdir ~/.local/bin
$ cp new.py ~/.local/bin
```

You should now be able to execute **`new.py`** from any location on your machine on a Unix machine:

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

Windows shells like `cmd.exe` and PowerShell don't read and execute the shebang like Unix shells, so you are required that you to prefix the command `python.exe` or `python3.exe` before a program name:

```
PS C:\Users\kyclark> python.exe C:\Users\kyclark\.local\bin\new.py
usage: new.py [-h] [-n NAME] [-e EMAIL] [-p PURPOSE] [-t] [-f] program
new.py: error: the following arguments are required: program
```

WARNING

Be sure `python.exe --version` shows that you are using version 3 and not version 2. You may need to install the latest version of Python. I will only show Windows commands using `python.exe` assuming this means Python 3, but you may need to use `python3.exe` depending on your system.

I'll show how to use this program to start every program so that you don't have to begin coding from a blank screen.

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 for 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 lit 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 helping her with classroom teaching proved 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

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)

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.

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

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active after final publication.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at ccollins@oreilly.com.

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

In 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
```

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 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
```

Run the command `pytest` and you should see something like the following to indicate that the program passes all the tests:

```
$ 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 programs. *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 the `metavar` value. Delete all the other arguments. 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 figure this out 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 can also 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)   ⓫


# --------------------------------------------------
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.

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

⓬ 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 there's no possibility to know positively *at compile time* what attributes will be available in the parsed arguments or what their types would be.

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 `python3` compiler itself 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.

Tools like linters and type checkers can spot problems with 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. This program will find when I've used *types* incorrectly 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 A PYTHON REPL

In the next section, I want to show you how to use the `python3` REPL to

interact with the Python compiler. Here REPL 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 REPL inside an IDE (integrated development environment) like VS Code as in Figure 1-1.

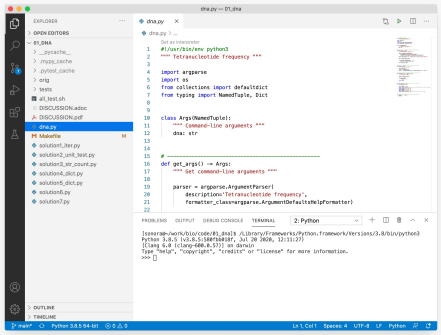


Figure 1-1. You can run a Python REPL in VS Code.

Regardless of your choice, I *highly* encourage you to type all the examples yourself. You will learn so much from interacting with the Python compiler. The standard `python3` REPL uses `>>>` as a prompt. Be sure you don't type the `>>>` prompt, only the text that follows:

```
Python 3.9.0 (v3.9.0:9cf6752276, Oct  5 2020, 11:29:23)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

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'])
```

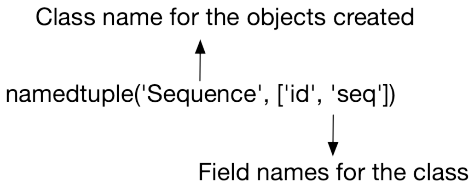


Figure 1-2. The `namedtuple()` function generates a way to make objects of the class `Sequence` that have the fields `id` and `seq`.

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:

```
>>> 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 `mypy` will 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` 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 than only 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 absolute 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 and the `expected` value from the program.

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 Bases

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):

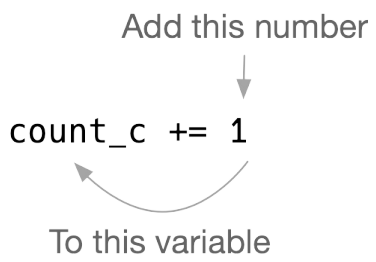


Figure 1-3. The += operator will add the value on the right-hand side to the variable on the left-hand side.

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 to me 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: 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*.

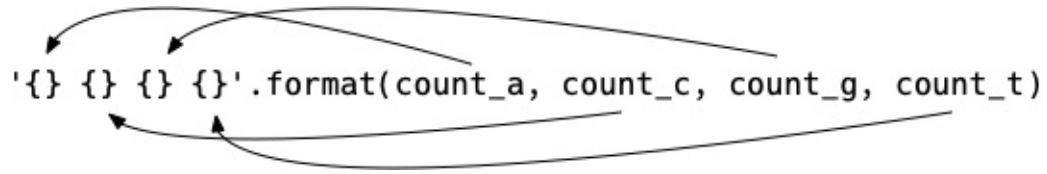


Figure 1-4. The str.format() function uses a template containing curly brackets to define placeholders that are filled in with the values of the arguments.

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['T'], counts['G']
(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')
3
>>> 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:

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

The `NoneType` is the type for the `None` value.

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." This 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}
```

```
    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.

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 the 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. 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.

The following chapters will be a bit shorter as I'll build upon many of the foundational ideas I've covered here.

---

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

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active after final publication.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at ccollins@oreilly.com.

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, this program should 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 last chapter 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 to RNA

positional arguments: ❷
  FILE                  Input RNA 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 Ts changed to Us:

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

You should run the program with multiple inputs and verify 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*.

While you are free to write and structure your programs however you like (so long as they pass the tests), I will always start a program using `new.py` and 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

You need to 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 are not files. 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(file=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.file: 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.file`. 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 pseudo-code from the previous section. Remember that `args.file` 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.file:
```

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 File Names

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'
```

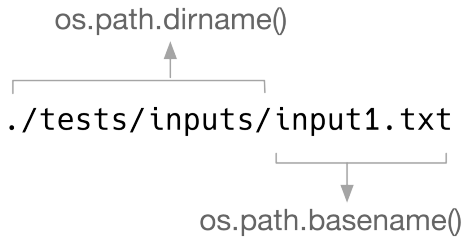


Figure 2-1. The os.path module contains useful functions like `dirname()` and `basename()` to extract parts from a file's path.

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:

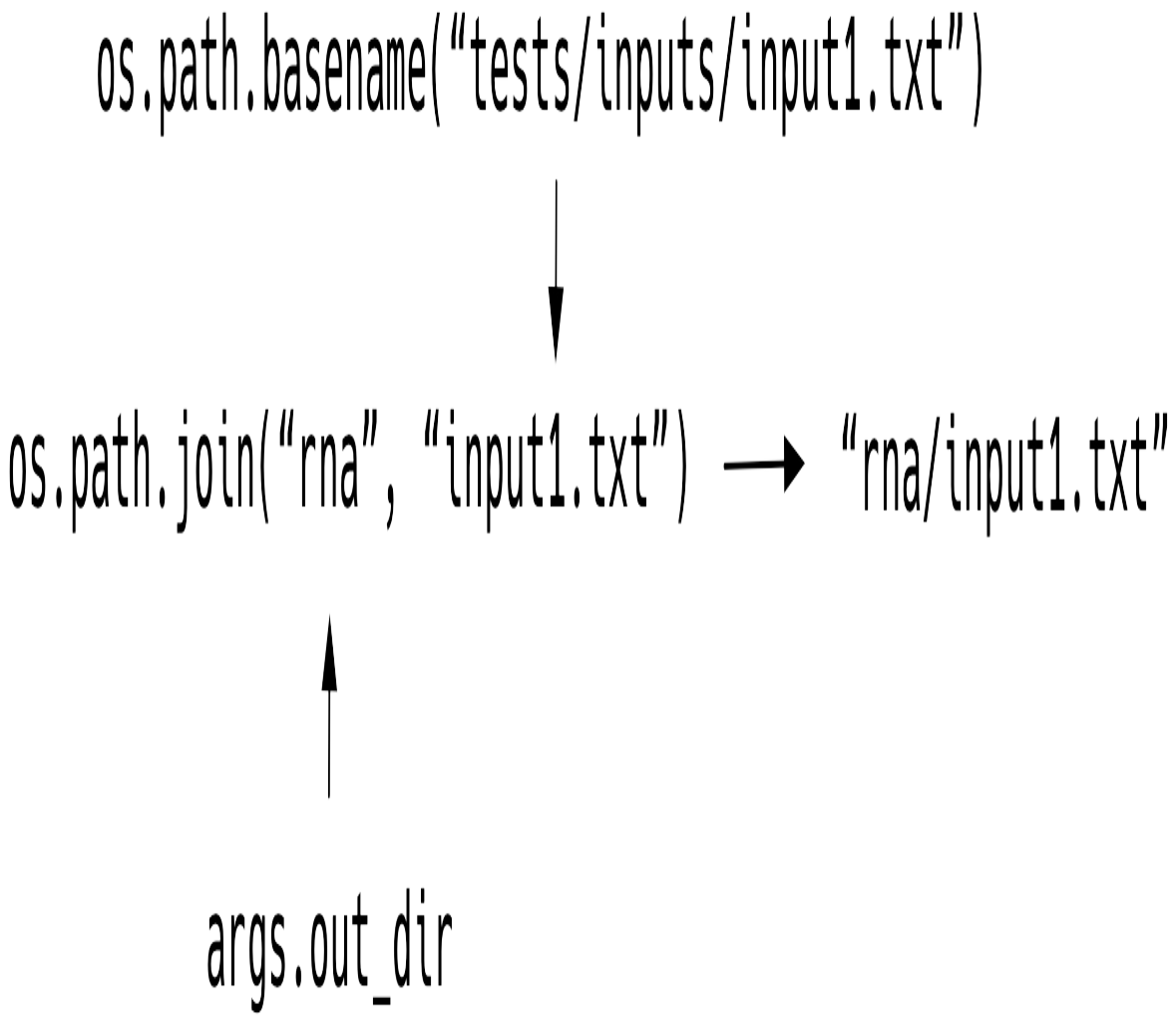


Figure 2-2. The os.path.join() will create the output path by combining the output directory with the basename of the input file.

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

```
for fh in args.file:
    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. This 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 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.file:
    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.file:
    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. I mentioned in Chapter 1 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:

```
>>> 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, 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 Ts to Us 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 Make sure the output looks like a usage statement.

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 urge you to run also use `pylint`, `flake8`, and `mypy` to check your program. Remember 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` not being snakecase or too short. 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

Solution 1: Using str.replace

Here is the entirety of one solution that uses the `str.replace()` method I

discussed in the introduction:

```
#!/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.file: 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. While they may seem somewhat inscrutable and 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 exact location of the found *T* as the start and stop indexes which I can use to extract the substring with a slice:

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

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

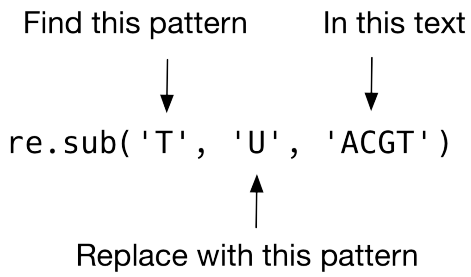


Figure 2-3. The `re.sub()` function will return a new string where all instances of a pattern have been replaced with a new string.

The result is a new string where the Ts have all be replaced with Us:

```
>>> 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.

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 Ts for Us, 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 *seq.txt* that is generated using the defaults is about 95M. 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

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active after final publication.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at ccollins@oreilly.com.

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.

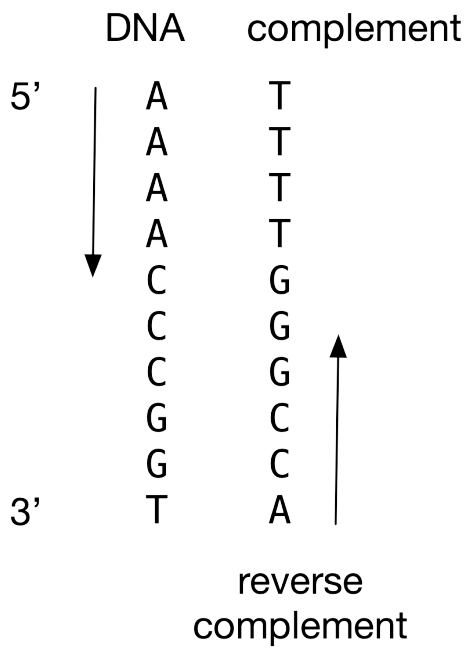


Figure 3-1. The reverse complement of DNA is the complement read from the opposite direction.

While 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, I will show:

- 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

I'll start with how you can reverse a string. It happens 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*? 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))
'TGGCCCAAAA'
```

In the REPL, I can coerce the 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()` says it 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):
...     rev += base
```

```
...
>>> rev
'TGGCCCAAAA'
```

❶ Put the empty string in between each of the elements in the reversed DNA to create a new string.

Since I still need to complement the bases, I'm not quite done.

Creating a Decision Tree

There are a total of 8 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 file name.

❸ Make sure the exit value is 0.

❹ Open and read the expected file and compare that to the output.

It's equally 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 preceding algorithm 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 my first drum set teacher 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 shown in 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 use the base as-is just the `else` case in solution 1 did:

```
>>> 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` to a be 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.

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 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.

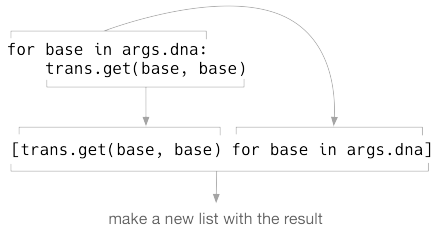


Figure 3-2. A list comprehension uses a for loop to generate a new list.

This shortens the 3 lines to initialize a `complement` and loop through the DNA down to 1 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. If it were longer, I would definitely write this on two separate lines. 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 Ts to Us when transcribing DNA to RNA. Could I use that here? Let's try. I'll start by initializing the DNA string and replacing the As with Ts. 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 Ts to As now, and see if you can spot the problem:

```
>>> dna = dna.replace('T', 'A')
>>> dna
'AAAACCCGGA'
```

As shown in Figure 3-3, all the As that turned into Ts in the first move were just changed back to As. Clearly, that way lies madness.

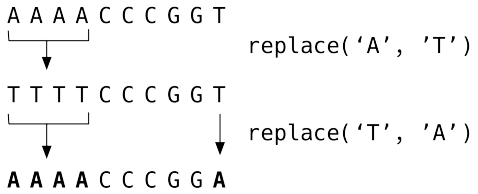


Figure 3-3. Iteratively using `str.replace()` leads to double replacements of values and the wrong answer.

Fortunately, Python has the `str.translate()` function for exactly this purpose. The function requires a description of all the complementing strings as the key/values pairs of a dictionary which is the `trans` table. I first have to use 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

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active after final publication.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at ccollins@oreilly.com.

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 sometimes 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.

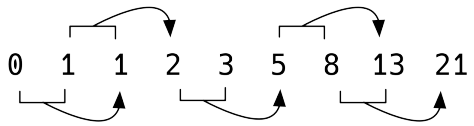


Figure 4-1. The first 8 numbers of the Fibonacci sequence. After the initial 0 and 1, subsequent numbers are created by adding the two previous numbers.

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. This is a well-behaved program. It will also repel a floating point value:

```
$ ./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 (*n*) argument of *-3* value is not in the stated range of values.

❷ The litter (*k*) 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('n', ❸
                        metavar='generations',
                        type=int, ❹
                        help='Number of generations')

    parser.add_argument('k', ❺
                        metavar='litter',
```

```
                        type=int,
                        help='Size of litter per generation')

    args = parser.parse_args() ❻

    if not 1 <= args.n <= 40: ❼
        parser.error(f'n "{args.n}" must be between 1 and 40') ❽

    if not 1 <= args.k <= 5: ❾
        parser.error(f'k "{args.k}" must be between 1 and 5') ❿

    return Args(generations=args.n, litter=args.k) ⓫
```

❶ The `generations` field must be an `int`.

❷ The `litter` field must also be an `int`.

❸ The `generations` positional parameter is defined first, so it will receive the first positional value. Internally this will be called `n`.

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. Internally this will be called `k`.

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.n` 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.k` argument.

❿ Generate an error that includes information the user needs to fix the problem.

⓫ If the program makes it to this point, then the arguments are valid integer values in the accepted range, so return the `Args`.

I could check that the *n* and *k* 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'n = {args.generations}')
    print(f'k = {args.litter}')
```

Run your program with invalid inputs and verify 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
n = 1
k = 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_n PASSED                                 [
42%]
tests/fib_test.py::test_bad_k 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 'n = 5\nk = 3' == '19' ❹
E         - 19    ❺
E         + n = 5 ❻
E         + k = 3

tests/fib_test.py:56: AssertionError
======================== short test summary info
========================
FAILED tests/fib_test.py::test_1 - AssertionError: assert 'n = 5\nk =
...
!!!!!!!!!!!!!!!!!!!!!! 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 the values of n = 5 and k = 3.

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 5th 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:

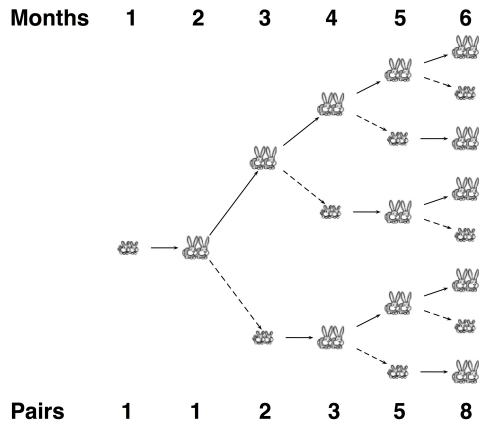


Figure 4-2. A visualization of the growth of the Fibonacci sequence as mating pairs of rabbits.

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 second number 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]
```

Can you imagine how to take this idea and repeat it *generations* times? (Remember that Python uses 0-based indexing, so it will actually be *generations - 1* times.) 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 it. This would normally indicate a possible error, so I'm indicating that I know I'm not using the value. Here 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.

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 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.

The type signature for `Generator` looks a little complicated. I don't need to dive into it further here, but I recommend you read the docs on the `typing` module. Otherwise, this code does exactly what the first version of the program did only inside a fancy-pants generator function.

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.

While 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.

Or, since I only use the `seq` variable one time, make it 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

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

I defined the `fib()` function *inside* the `main()` function so as to use the `args.litter` value inside the `fib()` function. This kind of function is called a *closure* because it closes around a value (the litter size). If I had defined it outside the `main()` function, I would have had to deal with passing 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. To illustrate the problem, let's 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.

I can accomplish the same thing by using the *decorator syntax* like so:

```
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
```

Add this a decorator on the `fib()` 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'll use 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`. 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 this 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 `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 file name.

6. Make the program executable.
7. Run the `make test` command, capture the return value and output.
8. See if the return value is not zero.
9. Exit this program while printing the output from the testing and returning a non-zero value.
10. Unless the program is supposed to be quiet, print the testing output.
11. 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 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 4-3). 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.

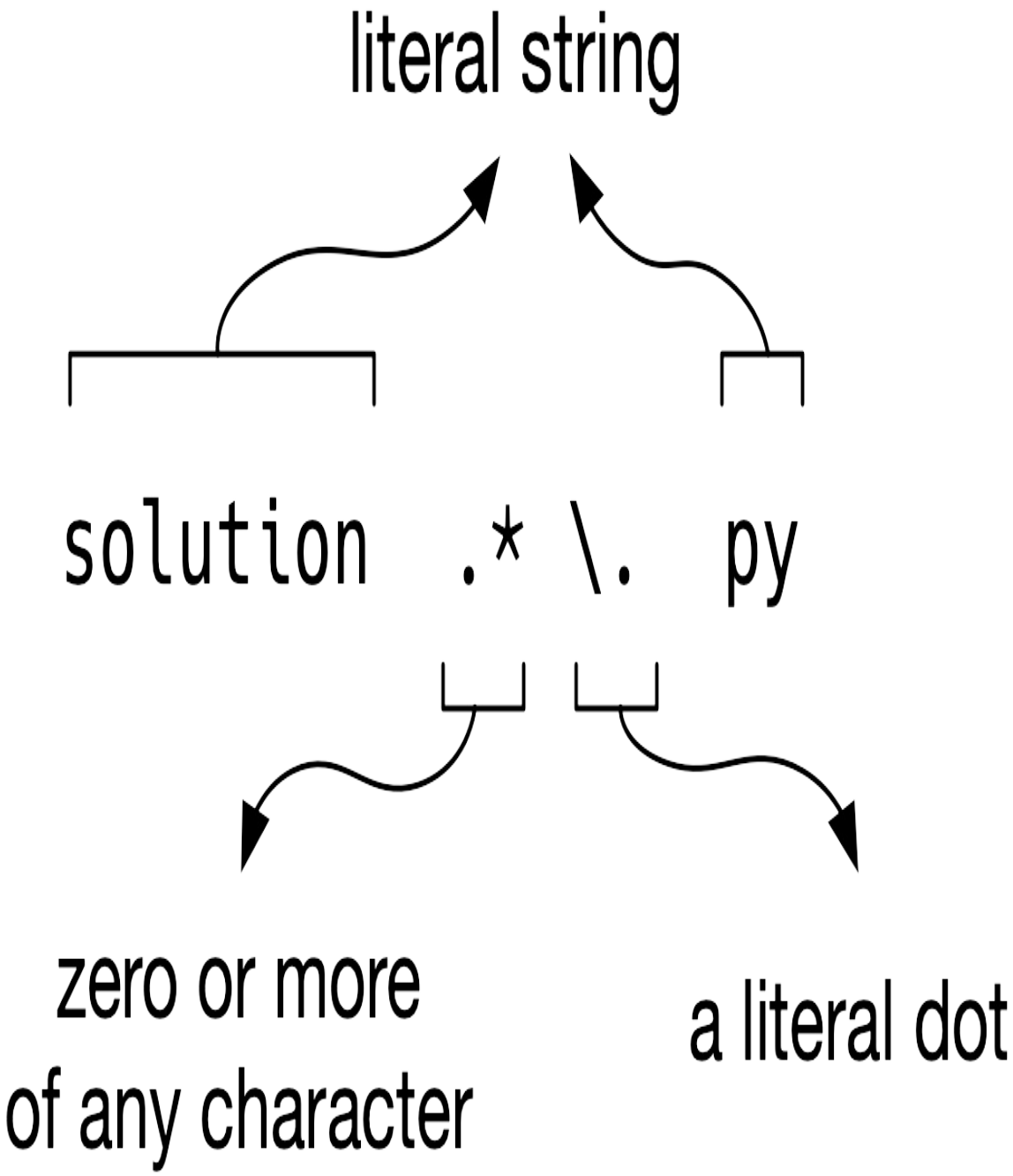


Figure 4-3. A regular expression to find files matching the file glob "solution.py":*

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:

```
>>> 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:

```
>>> from functools import partial
>>> 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'>
```

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 emoization 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

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book. Please note that the GitHub repo will be made active after final publication.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at ccollins@oreilly.com.

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.

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 it looks like a dictionary:

```
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:

```
>>> 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
```

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 = sorted(seqs)[-1] 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.

5. Check if the base is a *C* or *G*.
6. Increment the GC counter.
7. Calculate the GC content.
8. Append a new tuple of the GC content and the sequence ID.
9. Sort the sequences and take the last one.
10. 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 the `sorted()` function. First off, `sorted()` 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]
```

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 TDD:

- Define a function to test
- Write the test
- Ensure the function fails the test
- Make the function work
- Ensure the function passes the test

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 like this 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*:

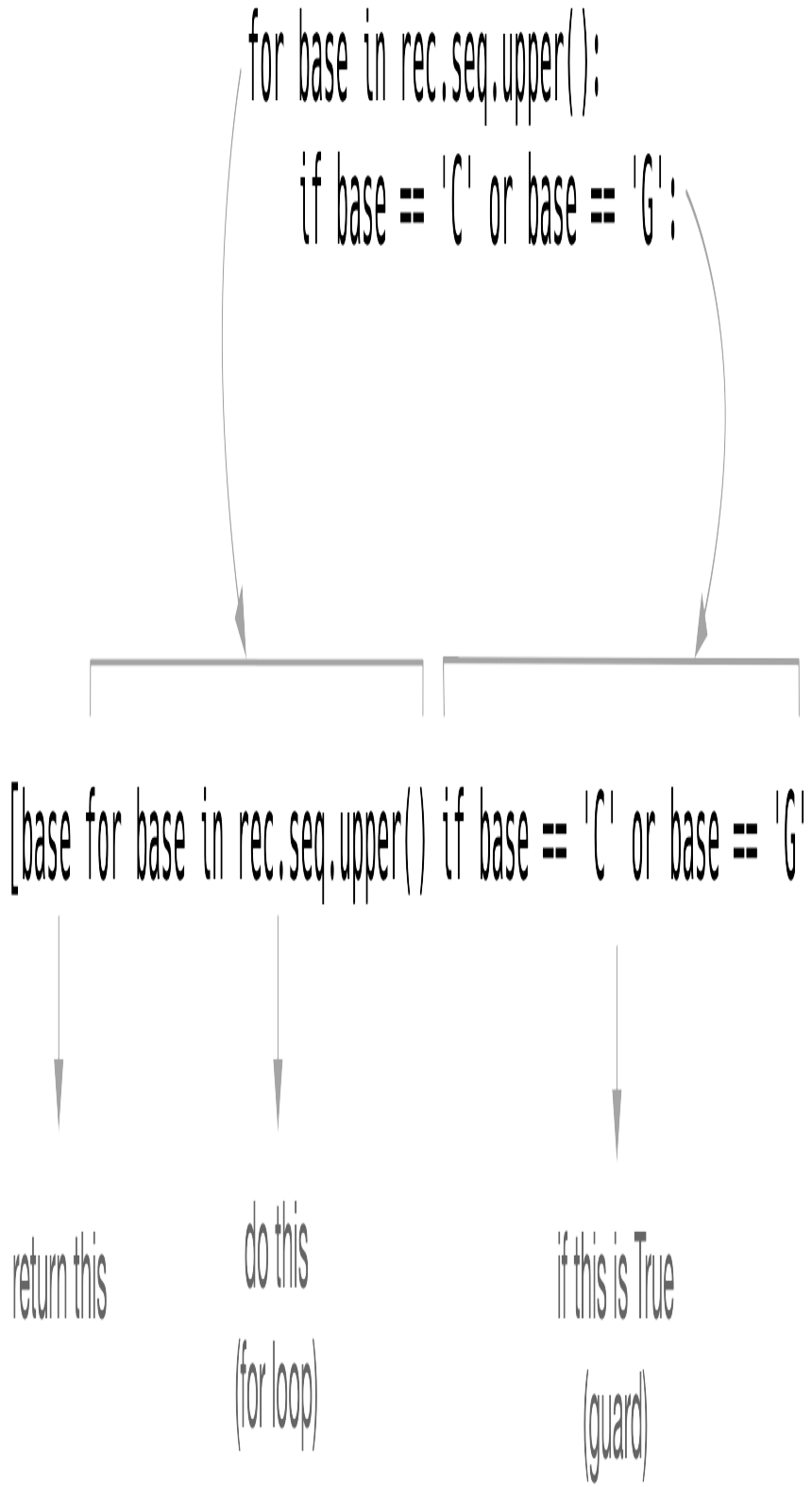


Figure 5-1. A list comprehension with a guard will select only those elements returning a truthy value for

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 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']
```

A shorter way to write that `if` comparison is using `in`:

```
>>> list(filter(lambda base: base in 'GC', '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. Here is how I would incorporate that into my function:

```
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 real data as 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.

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(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

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. 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 Bases

ii. Writing and Verifying a Solution

c. Solutions

i. Solution 2: 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 File Names
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. Index