

Tiny Python Projects

Learn coding and testing with puzzles and games

Ken Youens-Clark

MEAP



MANNING



MEAP Edition
Manning Early Access Program
Tiny Python Projects
Learn coding and testing with puzzles and games
Version 6

Copyright 2020 Manning Publications

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

welcome

Hey, thanks for buying the MEAP edition of *Tiny Python Projects: Learn coding and testing with puzzles and games!!* This is as close to an “interactive” book as I could make. You see, I want you to be actively involved in learning how to write Python really well by using a process called “test-driven development” (TDD). So, really, you’ll learn two things from this book: how to write Python, and how to use and write tests for your code.

The first chapter describes how to structure a Python program into discrete functions, one of which is dedicated to getting command-line arguments and producing documentation for your program. Because the processing of command-line arguments using the “argparse” module is so central to each exercise, I’ve included an appendix with many examples. The following chapters each describe a coding challenge that you can try your hand at solving and a set of tests that have been provided in a Git repository so you can use the TDD process as you write your code solution. Each chapter is dedicated to some central idea, like how to manipulate a string or list, when and how to use a dictionary, how to use random events, how to write functions and algorithms, etc. As the programs get more complex, I encourage you to write your own functions *and the tests for them* so that you learn how to become a self-sufficient tester.

I really look to getting feedback from readers with suggestions for improvements and more extension exercises. Please be sure to post any questions, comments, or suggestions you have about the book in the [liveBook Discussion Forum](#).

—Ken Youens-Clark, Tucson, AZ

brief contents

Preface

Getting Started: Introduction and Installation Guide

- 1 How to write and test a Python program*
- 2 The Crow's Nest: Working with strings*
- 3 Going on a picnic: Working with lists*
- 4 Jump the Five: Working with dictionaries*
- 5 Howler: Working with files and STDOUT*
- 6 Words Count: Reading files/ STDIN, iterating lists, formatting strings*
- 7 Gashlycrumb: Looking items up in a dictionary*
- 8 Apples and Bananas: Find and replace*
- 9 Dial-A-Curse: Generating random insults from lists of words*
- 10 Telephone: Randomly mutating strings*
- 11 Bottles of Beer Song: Writing and testing functions*
- 12 Ransom: Randomly capitalizing text*
- 13 Twelve Days of Christmas: Algorithm design*
- 14 Rhymers: Using regular expressions to create rhyming words*
- 15 The Kentucky Friar: More regular expressions*
- 16 The Scrambler: Randomly reordering the middles of words*
- 17 Mad Libs: Using regular expressions*
- 18 Gematria: Numeric encoding of text using ASCII values*

19 Workout of the Day: Parsing CSV file, creating text table output

20 Password Strength: Generating a secure and memorable password

21 Tic-Tac-Toe: Exploring state

22 Tic-Tac-Toe Redux: An interactive version with type hints

Epilogue

Appendix: Using argparse

****preface****

Why Write Python?

Python is an excellent, general-purpose programming language. You can write a program to send secret messages to your friends or to play chess. There are Python modules to help you wrangle complex scientific data, explore machine learning algorithms, and generate publication-ready graphics. Many college-level computer science programs have moved away from languages like C and Java to Python as their introductory language because Python is a relatively easy language to learn. We can use Python to study fundamental and powerful ideas from computer science. As I show you ideas like regular expressions and higher-order functions, I hope to encourage you to study further.

Who Am I?

My name is Ken Youens-Clark. I work as a Senior Scientific Programmer at the University of Arizona. Most of my career has been spent working in bioinformatics using computer science ideas to study biological data. I began my undergraduate degree as a Jazz Studies major on the drumset at the University of North Texas in 1990. I changed my major a few times and eventually ended up with a BA in English literature in 1995. I didn't really have a plan for my career, but I did like computers. Around 1995, I started tinkering with databases and HTML at my first job out of college, building the company's mailing list and first website. I was definitely hooked! After that, I managed to learned VisualBasic on Windows 3.1 and, through the next few years, I programmed in several languages and companies before landing in a bioinformatics group at Cold Spring Harbor Laboratory in 2001 led by Lincoln Stein, an early advocate for open software, data, and science. In 2015 I moved to Tucson, AZ, to work at the University of Arizona where I finished my MS in Biosystems Engineering in 2019. When I'm not coding, I like playing music, riding bikes, cooking, reading, and being with my wife and children.

Who Are You?

I think my ideal reader is someone who's been trying to learn to code well but isn't quite sure how to level up. Perhaps you are someone who's been playing with Python or some other language that has a similar syntax like Java(Script) or Perl? Maybe you've cut your teeth on something really different like Haskell or Scheme and you're wondering how to translate your ideas to

Python? You may already been writing Python and are looking for interesting challenges with enough structure to help you know when you're moving in the right direction?

This is a book that will try to teach you well-structured, documented, testable code in Python. The material introduces best-practices from industry such as "test-driven development"—that's when the *tests* for a program exist even before the program itself is written! I will show you how to read documentation and Python Enhancement Proposals (PEPs) and how to write idiomatic code that other Python programmers would immediately recognize and understand.

This is probably not an ideal book for the absolute beginning programmer. I assume no prior knowledge of the Python language specifically because I'm thinking of someone who is coming from another language. If you've never a program an *any* language at all, you might do well to come back to this material when you are comfortable with ideas like variables, loops, and functions.

Why Did I Write This Book?

Over the years, I've had many opportunities to help people learn programming, and I always find it rewarding. The structure of this book comes from my own experience in the classroom where I think formal specifications and tests can be useful aids in learning how to break a program into smaller problems that need to be solved to create the whole program.

The biggest barrier to entry I've found when I'm learning a new language is that small concepts of the language are usually presented outside of any useful context. Most programming language tutorials will start with printing "HELLO, WORLD!" (and this is book is no exception). Usually that's pretty simple. After that, I usually struggle to write a complete program that will accept some arguments and do something *useful*. Here I try to show you many, many examples of programs that do useful things in the hopes that you can modify these programs to make more programs for your own use.

More than anything, I think you need to practice. It's like the old joke: "What's the way to Carnegie Hall? Practice, practice, practice." These coding challenges are short enough that you could probably finish each in a few hours to days. This is more material than I could work through in a semester-long university-level class, so I imagine the whole book would take you several months. I hope you will solve the problems, then think about them and come back later to see if you can solve them differently, maybe using a more advanced technique or making them run faster.



Getting Started: Introduction and Installation Guide

This is a book that will help you learn how to write Python programs that run on the command line. If you have never used the command line before, don't worry! You can use programs like PyCharm or Microsoft's VSCode to help you write and run these programs. If you are completely new to programming or to the Python language, I will try to cover everything I think you'll need to know, although you might find it useful to read another book first if you've never heard of things like variables and functions.

In this section, we'll discuss:

- Why we should learn to write command-line programs
- Tools and environments for writing code
- How and why we test software

Figure 0.1. This is the PyCharm tool being used to edit and run the hello.py program from chapter 1. "Hello, World!"

The screenshot shows the PyCharm interface with the following details:

- Title Bar:** tiny_python_projects [~/work/tiny_python_projects] - .../01_hello/hello08_formatted.py
- Project View (Left):** Shows the project structure under 'tiny_python_projects'. It includes subfolders like '01_hello' containing various Python files (hello01_print.py, hello02_comment.py, etc.) and other files like Makefile, README.md, and test.py.
- Code Editor (Center):** Displays the contents of 'hello08_formatted.py'.

```

1  #!/usr/bin/env python3
2  """
3      Author: Ken Youens-Clark <kyclark@gmail.com>
4      Purpose: Say hello
5  """
6
7  import argparse
8
9  #
10 def get_args():
11     """Get the command-line arguments"""
12
13     parser = argparse.ArgumentParser(description='Say hello')
14     parser.add_argument('-n', '--name', default='World', help='Name to greet')
15     return parser.parse_args()
16
17 #
18 def main():
19     """Make a jazz noise here"""
20
21     args = get_args()
22     print('Hello, ' + args.name + '!')
23
24 #
25 if __name__ == '__main__':
26     main()
27
28 
```
- Terminal (Bottom):** Shows the output of running the script with the command: '(venv) [cholla@~/work/tiny_python_projects/01_hello]\$./hello08_formatted.py -n World'. The output is 'Hello, World!'
- Status Bar (Bottom):** Shows information about the PyCharm version (2020.1), update status, terminal encoding (24:39 LF UTF-8), Git status (master), and Python version (Python 3.8).

Why do I want you to write command-line programs? For one, I think they strip a program down to the most bare essentials. We're not going to try to complicated programs like an interactive 3D game that requires lots of other software to work. These programs all work with the barest of inputs and create only text output. We're going to focus on learning the core Python language and how to write and test our

programs.

Another reason is that I want to show you how to write programs that can run on any computer that has Python installed. I'm writing this book on my Mac laptop, but I can run all the programs on any of the Linux machines I use in my work or on a friend's Windows machine. Any computer with the same version of Python can run any of these programs, and that is pretty cool.

The biggest reason I want to show you how to write command-line programs, though, is because I want to show you how to *test* programs to make sure they work. Not long ago, NASA shared the source code from the Apollo 11 mission that first landed humans on the moon. (You can find it on GitHub!) I can't understand any of the code, but I do understand that lives depended on that code working perfectly. While I don't think anyone will die if I make a mistake in one of my programs, I still really, really want to be sure that my code is as perfect as possible.

What does it mean to test a program? Well, if my program is supposed to add two numbers together, I run it with many pairs of numbers and check that it prints the correct sum. I might also give it a number and a word to make sure that it doesn't try to add "3" plus "seahorse" but instead complains that I didn't give it two numbers. Testing gives me some measure of confidence in my code, and I hope you will come to see how testing can help you understand programming more deeply.

The exercises are meant to be silly enough to pique your interest, but they each contain lessons that can be applied to all sorts of real-world problems. Almost every program I've ever written needs to accept some input data — whether from the user or from a file or maybe a database — and produce some output, sometimes text on the screen or maybe a new file. These are the kinds of skills you'll learn by writing these programs.

In each chapter, I describe some puzzle or song or game that I want you to write. Then I show you a solution and discuss how it works. Most importantly, each chapter includes tests so that you check if your program is working correctly.

When you're done with this book, you should be able to:

- Write and run command-line Python programs.
- Handle arguments to your programs.
- Write and run tests for your programs and functions.
- Use Python data structures like strings, lists, and dictionaries.
- Read and write text files in your programs.
- Use regular expressions to find patterns in text.
- Use and control randomness to make your programs behave unpredictably.

"Codes are a puzzle. A game, just like any other game." - Alan Turing

Alan Turing is perhaps most famous for cracking the Enigma code that the Nazis used to encrypt messages during World War II. The fact that the Allies could read enemy messages is credited with shortening the war by years and saving millions of lives. *The Imitation Game* is a fun movie that shows how Turing published puzzles in newspapers

to find people who could help him break what was supposed to be an unbreakable code.

I think we can learn tons from writing a program that generates random insults or produces verses to "The Twelve Days of Christmas" or plays Tic-Tac-Toe. Some of the programs even dabble a bit in cryptography, like "Jump The Five" where we encode all the numbers in a piece of text or "Gematria" where we create signatures for words by summing the numeric representations of their letters. I hope you'll find the programs both amusing and challenging!

The programming techniques in each exercise are not specific to Python. Most every language has variables, loops, functions, strings, lists, and dictionaries. After you write your solutions in Python, I would encourage you to write solutions in another language you know and compare what parts of a different language make it easier or harder to write your programs. If your programs support the same command-line options, you can even use the included tests to verify those programs!

0.1 **Using test-driven development**

"Test-driven development" is described by Kent Beck in his 2002 book by that title as a method to create more reliable programs. The basic idea is that we write tests even before we write code. The tests define what it means to say that our program works "correctly." We run the tests and verify that our code fails. Then we write the code to make each test pass. We always run *all of the tests* so that, as we fix new tests, we ensure we don't break tests that were passing before. When all the tests pass, we have at least some assurances that the code we've written conforms to some manner of specification.

Each program you are asked to write comes with tests that will tell you when the code is working acceptably. The first test in every exercise is whether the expected program exists. The second test checks if the program will print a help message if we ask for it. After that, your program will be run with various inputs and options.

Since I've written around 250 of tests for the programs and you have not yet written one of the programs, you're going to encounter many failed tests. That's OK! In fact, it's a really good thing, because when you pass them all you'll know that your programs are correct. We'll learn to read the failed tests carefully to figure out what to fix. Then we correct the program and run the tests again. We may get another failed test, in which case we'll repeat the process until finally all the tests pass. Then you are done.

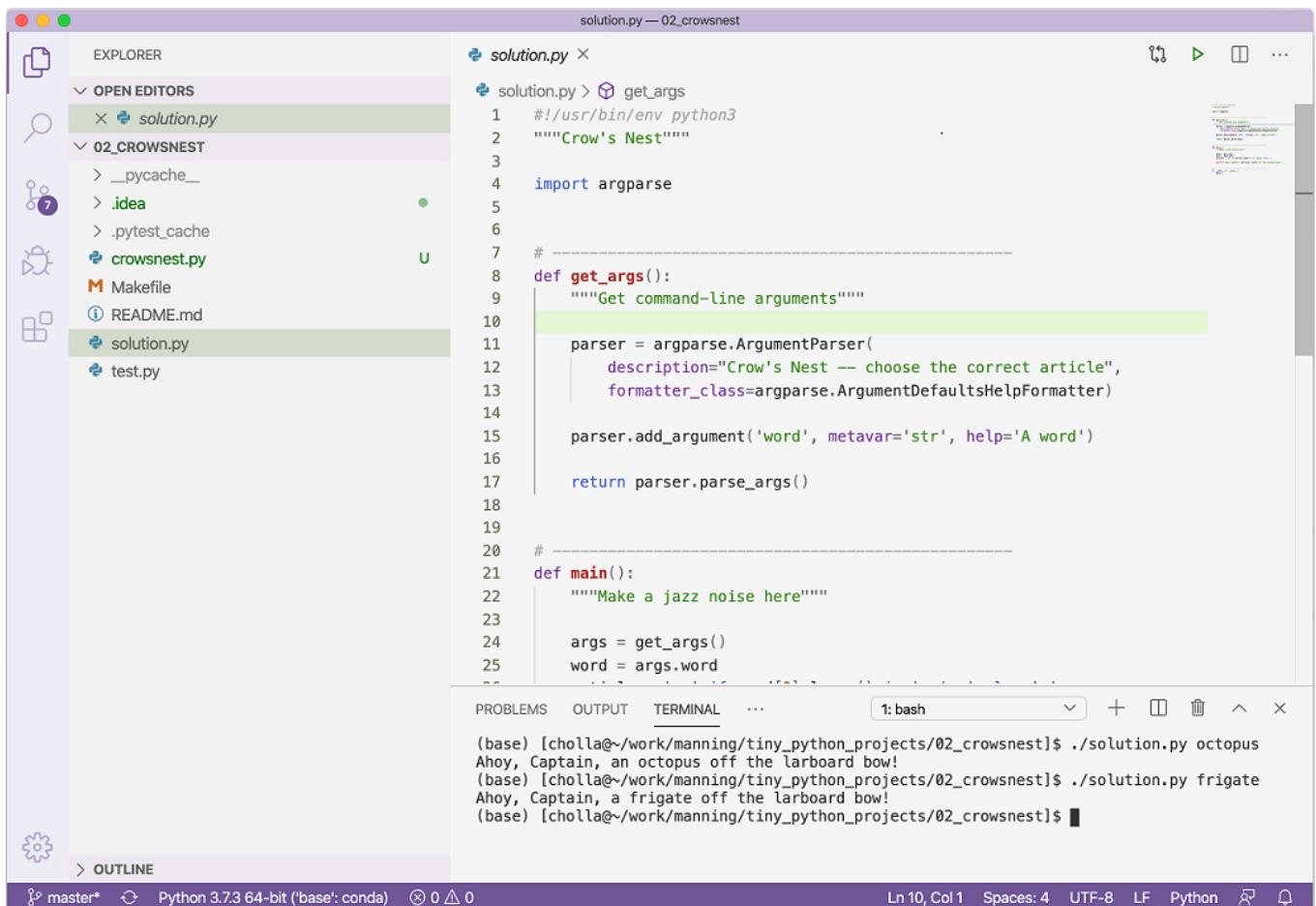


It doesn't matter if you solved the problem the same way as in the solution I provide. All that matters is that you figure out a way to pass the tests.

0.2 Setting up your environment

If you want to write these programs on your computer, you will need Python version 3.6 or later. It's quite possible that it's already installed on your computer. You'll also need some way to execute the `python3` command — something we often call a "command line." If you use a Windows computer, you may want to install Windows Subsystem for Linux. On a Mac, the default Terminal app is sufficient. You can also use a tool like VSCode or PyCharm which have terminals built into them.

Figure 0.2. An IDE like VSCode combines a text editor for writing your code along with a terminal (lower-right window) for running your programs and many other tools.



I wrote and tested the programs with the Python version 3.8, but they should work with any version 3.6 or newer. Python 2 reached end-of-life at the end of 2019 and should no longer be used. To see what version of Python you have installed, open a terminal window and type `python3 --version`. If it says something like command "python3"

not found, then you need to install Python. You can download the latest version from www.python.org/downloads/.

If you are using a computer that doesn't have Python and you don't have any way to install Python, then you can do everything in this book using the website repl.it.

0.3 Code examples

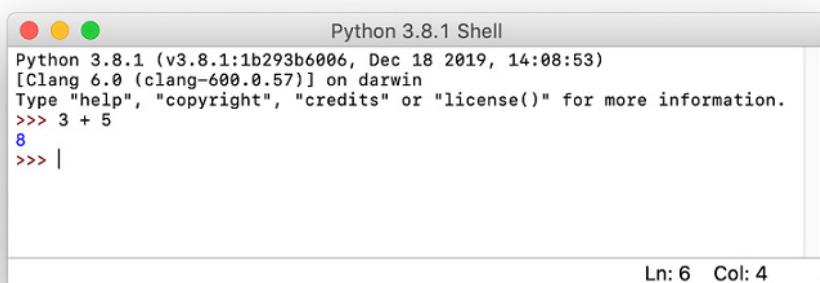
Throughout the book, I will show commands and code using a fixed-width font. When the text is preceded with the \$ (dollar sign), that means it's something you can type on the command line. For instance, there is a program called cat (short for "concatenate") that will print the contents of a file to the screen. Here is how I can run it to print the contents of the file `spiders.txt` that lives in the `inputs` directory:

```
$ cat inputs/spiders.txt
Don't worry, spiders,
I keep house
casually.
```

If you want to run that command, *do not copy* the leading \$, only the text that follows, otherwise you'll probably get an error like \$: command not found.

Python has a really excellent tool called IDLE that allows you to interact directly with the language to try out ideas. You can start it with the command `idle3`. That should open a new window with a prompt that looks like >>>:

Figure 0.3. The IDLE application allows you to interact directly with the Python language. Each statement you type is evaluated when you press Enter and the results are shown in the window.

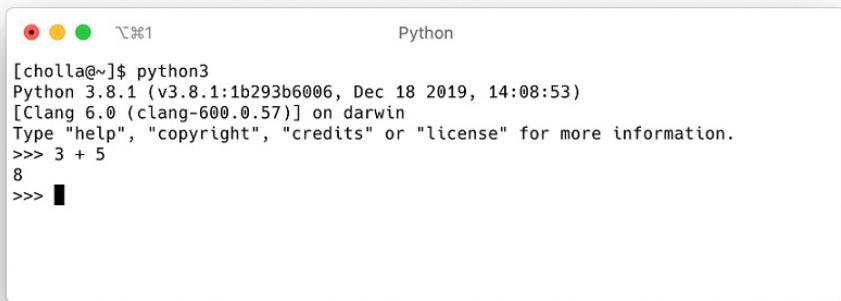


You can type Python statements there, and they will be immediately evaluated and printed. For example, type `3 + 5`<Enter> and you should see 8:

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

This interface is called a REPL because it's a read-evaluate-print-loop. (I pronounce this like "repple" in a way that sort of rhymes with "pebble.") You can get a similar tool by typing `python3` on the command line:

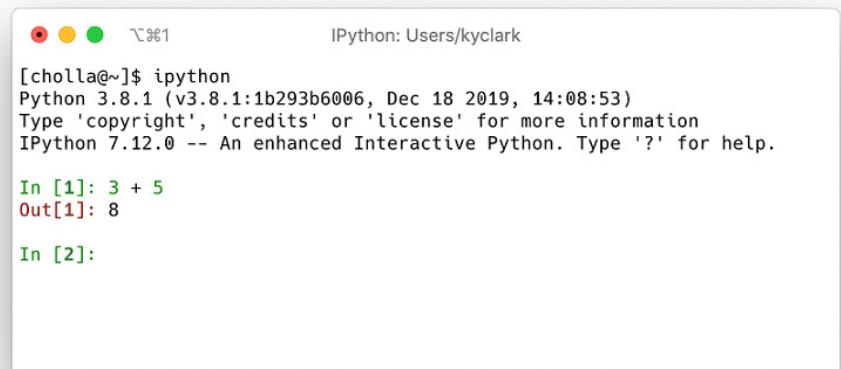
Figure 0.4. The command `python3` in the terminal will give you a REPL similar to the IDLE3 interface.



```
[cholla@~]$ python3
Python 3.8.1 (v3.8.1:1b293b6006, Dec 18 2019, 14:08:53)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 3 + 5
8
>>> █
```

The `ipython` program is yet another "interactive python" REPL that has many enhancements over IDLE and `python3`. This is what it looks like on my system:

Figure 0.5. The `ipython` application is another REPL interface to try out your ideas with Python.



```
[cholla@~]$ ipython
Python 3.8.1 (v3.8.1:1b293b6006, Dec 18 2019, 14:08:53)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.12.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: 3 + 5
Out[1]: 8

In [2]:
```

I would also recommend you look into using Jupyter Notebooks as they also allow you to interactively run code with the added bonus that you can save a Notebook as a file and share all your code with other people!

Whichever REPL interface you use, you can type Python statements like `x = 10<Enter>` to assign the value 10 to the variable `x`:

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

As with the command-line prompt \$, do not copy the leading >>> or Python will complain:

```
>>> >>> x = 10
      File "<stdin>", line 1
        >>> x = 10
          ^
SyntaxError: invalid syntax
```

The ipython REPL has a magical %paste mode that removes the leading >>> prompts so that you can copy and paste all the code examples:

```
In [1]: >>> x = 10
In [2]: x
Out[2]: 10
```

Whichever way you choose to interact with Python, I suggest you *manually type all the code yourself* as this builds muscle memory and forces you to interact with the syntax of the language.

0.4 Getting the code

All the tests and solutions are available at github.com/kyclark/tiny_python_projects. You can use the program git (which you may need to install) to copy that code to your computer with the following command:

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

Now you should have new directory called `tiny_python_projects` on your computer.

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 going to break off from my code and add your own programs to the repository. If you plan to use repl.it to write the exercises, I recommend you do fork my repo into your account so that you can configure repl.it to interact with your own GitHub repositories.

To fork, do the following:

1. Create an account on GitHub.com
2. Go to github.com/kyclark/tiny_python_projects
3. Click the "Fork" button to make a copy of the repository into your account.

Figure 0.6. The "Fork" button on my GitHub repository will make a copy of the code into your account.

The screenshot shows a GitHub repository page for 'kyclark / tiny_python_projects'. At the top, there's a navigation bar with links for 'Pull requests', 'Issues', 'Marketplace', and 'Explore'. Below the navigation bar, the repository name 'kyclark / tiny_python_projects' is displayed, along with a 'Watch' button, a 'Star' button (showing 27 stars), and a 'Fork' button (showing 23 forks). The 'Fork' button is highlighted with a red box. Below this, there's a menu bar with options like 'Code', 'Issues 0', 'Pull requests 0', 'Actions', 'Projects 0', 'Wiki', 'Security', 'Insights', and 'Settings'. The main content area shows a brief description: 'Learning Python through test-driven development of games and puzzles'. There are buttons for 'Edit' and 'Manage topics'. Below this, there are summary statistics: '401 commits', '1 branch', '0 packages', '0 releases', '1 contributor', and 'MIT'. There are also buttons for 'Branch: master', 'New pull request', 'Create new file', 'Upload files', 'Find file', and a prominent green 'Clone or download' button. The main part of the page lists commit history for the 'master' branch, showing commits from 'kyclark' with descriptions like 'cleanup', 'removing type hints, naming with help', and 'renumbering dirs'. The commits are timestamped with dates like '4 days ago' and '16 hours ago'.

Now 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/tiny_python_projects
```

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. After you have cloned your repository to your computer, go into your `tiny_python_projects` directory:

```
$ cd tiny_python_projects
```

And then execute this command:

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

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

```
$ git pull upstream master
```

0.5 *Installing modules*

I recommend using a few tools that may not be installed on your system. We can use the pip module to install them like so:

```
$ python3 -m pip install black flake8 ipython mypy pylint pytest yapf
```

I've also included a `requirements.txt` file in the top level of the repository that you can use to install all the modules and tools with this command:

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

If, for example, you wish 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.

0.6 *Code formatters*

Most IDEs and text editors will have tools to help you format your code so that it's easier to read and find problems. In addition, the Python community has created a standard for writing code so that other Python programmers can readily understand it. The PEP8 (Python Enhancement Proposal) document at www.python.org/dev/peps/pep-0008/ describes best practices for formatting code, and most editors will automatically apply formatting for you. For instance, the repl.it interface has an "auto-format" button, VSCode has a "Format Document" command, and PyCharm has a "Reformat Code" command.

Figure 0.7. The repl.it tool has an "auto-format" button to reformat your code according to community standards. The interface also includes a command line for running and testing your program.

```

@kyclark/tiny_python_projects
Learning Python through test-driven development of games and puzzles

Files .pytest_cache 01_hello hello.py saved
01_hello/hello.py
1 #!/usr/bin/env python3
2 """
3 Purpose: Say hello
4 Author: Ken Youens-Clark
5 """
6
7 import argparse
8
9
10 def get_args() -> argparse.Namespace:
11     """Get command-line arguments"""
12
13     parser = argparse.ArgumentParser(
14         description='Say hello',
15         formatter_class=argparse.ArgumentDefaultsHelpFormatter)
16
17     parser.add_argument('-n',
18                         '--name',
19                         default='World',
20                         metavar='str',
21                         help='The name to greet')
22
23     return parser.parse_args()
24
25
26
27 def main() -> None:
28

```

```

https://tinypythonprojects.kyclark.repl.run

> pwd
/home/runner/tinypythontoprojects
> cd 01_hello/
> make test
pytest -xv test.py
===== test session starts =====
platform linux -- Python 3.8.1, pytest-5.3.5, py-1.8.1, pluggy-0.13.1 -- /home/runner/.local/share/virtualenvs/python3/binary/python3
cachedir: .pytest_cache
rootdir: /home/runner/tinypythontoprojects/01_hello
collected 4 items

test.py::test_exists PASSED [ 25%]
test.py::test_usage PASSED [ 50%]
test.py::test_default PASSED [ 75%]
test.py::test_input PASSED [100%]

===== 4 passed in 1.35s =====
>

```

There are also command-line tools that integrate with your editor. I used yapf (Yet Another Python Formatter, github.com/google/yapf) to format every program in the book, but another popular formatter is black (github.com/psf/black). Whatever you use, I encourage you to use it *often*. For instance, I can tell yapf to format the `hello.py` program that we will write by running the following command. Note that the `-i` tells yapf to format the code "in place" so that the file will be overwritten with the newly formatted code:

```
$ yapf -i hello.py
```

0.7 Code linters

A code linter is a tool that will report problems in your code like declaring a variable but never using it. Two that I like are pylint (www pylint.org/) and flake8 (flake8.pycqa.org/en/latest/), and both can find errors in your code that the Python interpreter itself will not complain about. In the final chapter, I will show you how to incorporate "type hints" into your code which the mypy tool (mypy-lang.org/) can use to find problems like using text when you should be using a number.

0.8 How to start writing new programs

I think it's much easier to start writing code with a standard template, so I wrote a program called `new.py` that will help you create new Python programs with boilerplate code that will be expected of every program. It's located in the `bin` directory, so if you are in the top directory of the repository, you can run it like this:

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

Here we see that `new.py` is asking you to provide the name of the `program` to create. For each chapter, the program you write needs to live in the directory that has the `test.py` for that program. We can use this to start off the "Crow's Nest" program in the `02_crowsnest` directory like so:

```
$ bin/new.py 02_crowsnest/crowsnest.py
Done, see new script "02_crowsnest/crowsnest.py."
```

If you open that file now, you'll see that it has written a lot of code for you that we'll explain later. For now, just realize that the resulting `crowsnest.py` program is one that can run like so:

```
$ 02_crowsnest/crowsnest.py
usage: crowsnest.py [-h] [-a str] [-i int] [-f FILE] [-o] str
crowsnest.py: error: the following arguments are required: str
```

Later we'll learn how to modify the program to do what the tests expect.

An alternative to running `new.py` is to copy the file `template.py` from the `template` directory to the name of the program you need to write. So we could accomplish the same thing like so:

```
$ cp template/template.py 02_crowsnest/crowsnest.py
```

You do not have to use either `new.py` or copy the `template.py` to start your programs. These are provided to save you time and provide your programs with an initial structure, but you are welcome to write your programs however you please.

0.9 Why Not Notebooks?

Many people are familiar with Jupyter Notebooks as they provide a way to integrate Python code and text and images into a document that other people can execute like a program. I really love Notebooks especially for interactively exploring data, but I find them difficult to use in teaching for the following reasons:

- A Notebook is stored in JavaScript Object Notation (JSON), not as line-oriented text. This makes it really difficult to compare Notebooks to each other to find how they differ.
- Code and text and images can live in mixed together in separate cells. These cells can be interactively run in any order which can lead to very subtle problems in the

logic of a program. The programs we write, however, will always be run from top to bottom in entirety every time which I think makes them easier to understand.

- There is no way for Notebooks to accept different values at the time when they are run. That is, if you test a program with one input file and then want to change to a different file, you have to change *the program itself*. We will learn how to pass in the file as an *argument* to the program so that we can change the value without changing the *code*.
- It's difficult to automatically run tests on a Notebook or the functions they contain. We will use the `pytest` module to run our programs over and over with different input values and verify that the programs create the correct output.

0.10 **The scope of topics we cover**

This focus of this book is to show you how amazingly useful all the built-in features of the language are. The exercises really push you to practice how to manipulate strings, lists, dictionaries, and files. We spend several chapters focusing on regular expressions, and every exercise except for the last requires you to accept and validate command-line arguments of varying types and numbers.

Every author is biased towards some subjects, and I can be no different. I've chosen these topics because they reflect the ideas which are fundamental to the work I've done over the last 20 years. For instance, I have spent many more hours than I would care to admit parsing really messy data from countless Excel spreadsheets and XML files. The world of genomics that has consumed most of my career is based primarily on efficiently parsing text files, and much of my web development work is predicated on understanding how text is encoded and transferred to and from the web browser. For that reason, you'll find many exercises that entail processing text and files and that challenge you to think about how to transform inputs into outputs. If you work through every exercise, I believe you'll be a much improved programmer who understands the basic ideas that are common across many languages.

0.11 **Why not object-oriented programming?**

One topic you'll notice is missing from this book is how to write object-oriented code in Python. If you are not familiar with "object-oriented programming" (OOP), then you can skip this part. I think OOP is a somewhat advanced topic which is beyond the scope of this book. I prefer to focus on how to write small functions and their accompanying tests. I think this leads to more transparent code because the functions should be short, should only use the values explicitly passed as arguments, and should have enough tests that you completely understand how they will behave under both favorable and unfavorable circumstances.

The Python language is itself inherently object-oriented. Almost everything from strings to lists and dictionaries that we use are actually *objects*, so you'll get plenty of practice using objects! I don't think it's necessary to create objects to solve any of the problems I present. In fact, even though I spent many years writing object-oriented code, I haven't written in this style for the last few years. I tend to draw my inspiration

from the world of purely functional programming, and I hope I can convince you by the end of this book that you can do anything you want by combining functions!

Although I personally avoid OOP, I would recommend you learn about it. There have been several seismic paradigm shifts in the world of programming from procedural to object-oriented and now functional. You can find dozens of books on OOP in general and programming objects in Python specifically. This is a deep and fascinating topic, and I encourage you to try writing object-oriented solutions to compare to my solutions!

0.12 A Note about the lingo

Often in programming books you will see "foobar" used in examples. The word has no real meaning, but its origin probably comes from the military acronym "FUBAR" (Fouled Up Beyond All Recognition). If I use "foobar" in an example, it's because I don't want to talk about any specific thing in the universe, just the idea of a string of characters. If I need a list of items, usually the first item will be "foo," the next will be "bar." After that, convention uses "baz" and "quux," again because they mean nothing at all. Don't get hung up on "foobar." It's just a shorthand placeholder for something that could be more interesting later.

We also tend to call errors code "bugs." This comes from the days of computing before the invention of transistors. Early machines used vacuum tubes, and the heat from the machines would attract actual bugs like moths that could cause short circuits. The "operators" (the people running the machines) would have to hunt through the machinery to find and remove the bugs, hence the term to "debug."



How to write and test a Python program



Before we start writing the exercises, I want to discuss how to write programs that are documented and tested using the following principles. Specifically, we're going to:

- Write our first Python program to say "Hello, World!"
- Handle command-line arguments using `argparse`
- Run tests for your code with `pytest`
- Learn about `$PATH`
- Use tools like `yapf` or `black` to format your code
- Use tools like `flake8` and `pylint` to find problems in your code
- Use the `new.py` program to create new programs



It's pretty common to write "Hello, World!" as your first program in any language, so let's start there. We're going to work towards making a version that will greet a name that is passed as an argument. It will also print a helpful message when we ask for it, and we're going to use tests to make sure it does everything correctly. In the `01_hello` directory, you'll see there several versions of a "hello" program we'll write. There is also a program called `test.py` that we're going to use to test the program.

Start off by creating a text file called `hello.py` in that directory. If you are working in VSCode or PyCharm, you can use "File → Open" to open the `01_hello` directory as a project. Both tools have something like a "File → New" menu option that will allow you to create a new *file* in that directory. It's very important to create the `hello.py` file

inside the `01_hello` directory so that the `test.py` program can find it!

Once you've started a new file, add this line:

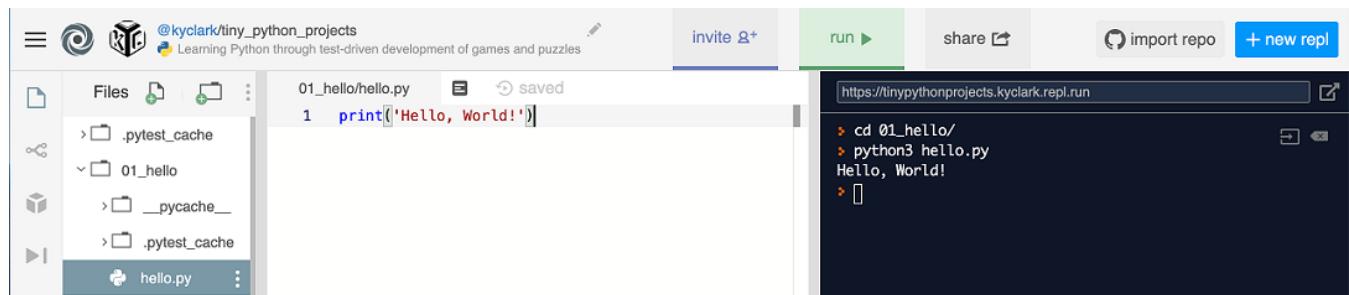
```
print('Hello, World!')
```

It's time to run our new program! You can open a terminal window in VSCode or PyCharm or some other terminal. Be sure to navigate to the directory where your `hello.py` program is located. We can run it with the command `python3 hello.py` to have Python version 3 execute the commands in the file called `hello.py`. You should see this:

```
$ python3 hello.py
Hello, World!
```

Here is how it looks in the repl.it interface:

Figure 1.1. Writing and running our first program using repl.it.



The screenshot shows the repl.it interface. At the top, there's a header with user information (@kyclark/tiny_python_projects) and various buttons like 'invite', 'run', 'share', 'import repo', and '+ new repl'. Below the header is a file browser on the left showing a directory structure: Files, .pytest_cache, 01_hello (containing __pycache__ and .pytest_cache), and a file named hello.py which is selected. The main area is a code editor with the following content:

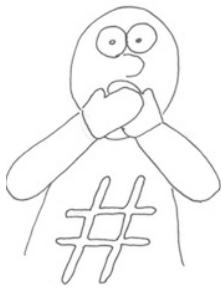
```
01_hello/hello.py  saved
1 print('Hello, World!')
```

To the right is a terminal window with the URL https://tinypythoprojects.kyclark.repl.run. It contains the command history and output:

```
cd 01_hello/
python3 hello.py
Hello, World!
> 
```

If that was your first Python program, congratulations!

1.1 Comment lines



In Python, the `#` character and anything following it is ignored by Python. This is useful to add comments to your code or to temporarily disable lines of code when testing and debugging. It's always a good idea to document your programs with the purpose of the program and/or the author's name and email address. We can use a comment for that:

```
# Purpose: Say hello
print('Hello, World!')
```

If you run it again, you should see the same output as before because the "Purpose" line is ignored. Note that any text to the left of the `#` is executed, so you can add a comment to the end of a line, if you like.

1.2 Testing our program

The most fundamental idea I want to teach you is how to test your programs. I've written a `test.py` program in the `01_hello` directory that we can use to test our new `hello.py` program. We will use the `pytest` program to execute all the commands and tell us how many tests we passed. We'll include the `-v` option that tells `pytest` to create "verbose" output. If you run it like this, you should see the following as the first several lines. After that will be many more lines showing you more information about the tests that didn't pass.

NOTE

If you get the error `pytest: command not found`, then you need to install the `pytest` module. Refer to the "Installing modules" section of the introduction.

```
$ pytest -v test.py
===== test session starts =====
...
collected 5 items

test.py::test_exists PASSED [ 20%] ①
test.py::test_runnable PASSED [ 40%] ②
test.py::test_executable FAILED [ 60%] ③
test.py::test_usage FAILED [ 80%] ④
test.py::test_input FAILED [100%] ⑤

===== FAILURES =====
```

- ① The first test always checks that the expected file exists. Here the test was looking for `hello.py`.
- ② The second test tried to run the program with `python3 hello.py` and then checked if the program printed "Hello, World!" If you miss even one character like forgetting the comma, then the test will point out the error, so read carefully!
- ③ The third test checks that the program is "executable." This test failed, and so next we're going to talk about how to make that pass.
- ④ The fourth test asked the program for help and didn't get anything. We're going to add the ability to print a "usage" statement that describes how to use our program.
- ⑤ The last test checks that the program can greet a name that we'll pass as an argument. Since our program doesn't yet accept arguments, we'll need to add that, too.

I've written all the tests in an order that I hope helps you to write the program in a logical fashion. If you can't pass one of the tests, there's no reason to continue running the tests after it. I recommend you always run the tests with the two flags `-x` to stop on the first failing test and `-v` to print verbose output. You can combine these like `-xv` or `-vx`. Here's what our tests look like with those options:

```
$ pytest -xv test.py
===== test session starts =====
...
collected 5 items

test.py::test_exists PASSED [ 20%]
test.py::test_runnable PASSED [ 40%]
test.py::test_executable FAILED [ 60%] ①
```

```
=====
      FAILURES =====
      test_executable _____
```

```
def test_executable():
    """Says 'Hello, World!' by default"""

    out = getoutput({prg})
>     assert out.strip() == 'Hello, World!'
E     AssertionError: assert '/bin/sh: ./h...ission denied' == 'Hello, World!'
E         - /bin/sh: ./hello.py: Permission denied
E         + Hello, World!
```

```
test.py:30: AssertionError
!!!!!!!!!!!!!! stopping after 1 failures !!!!!!!
===== 1 failed, 2 passed in 0.09s =====
```

- ① Notice that this test fails and so no more tests are run. This is because we ran pytest with the -x option.
- ② The > at the beginning of this line shows the source of the subsequent errors.
- ③ The E at the beginning of this line show that this is an "Error" you should read. The AssertionError is saying that test.py program is trying to execute the command ./hello.py to see if it will produce the text "Hello, World!"
- ④ The - character is showing the actual output from the command is "Permission denied".
- ⑤ The + character is showing the test expected to get "Hello, World!"

Let's talk about how to fix this error!

1.3 ***Adding the shebang line***

One thing we have learned about Python programs is that they live in plain text files that we ask python3 to execute. Many other programming languages such as Ruby and Perl work in the same way — we type Ruby or Perl commands into a text file and run it with the right language. It's common to put a special comment line in programs like these to indicate the language that needs to be used to execute the commands in the file. This comment line starts off with #!, and the nickname for this is "shebang" (pronounced "shuh-bang" — I always think of the # as "shuh" and the ! as the "bang!"). Just as with any other comment, Python will ignore the shebang, but the operating system (like Mac or Windows) will use it to decide which program to use to run the rest of the file.

Here is the shebang you should add:

```
#!/usr/bin/env python3
```

The env program will tell you about your "environment." When I run env on my computer, I see many lines of output like USER=kyclark and HOME=/Users/kyclark. These values are accessible as the variables \$USER and \$HOME:

```
$ echo $USER
kyclark
$ echo $HOME
```

```
/Users/kyclark
```

If you run env on your computer, you should see your login name and your home directory which, of course, will have different values from mine, but we both (probably) have both of these ideas.

We can use the env command to find and run programs. If you run env python3, it will run the python3 program if it can find one. Here's is what I see on my computer:

```
$ env python3
Python 3.8.1 (v3.8.1:1b293b6006, Dec 18 2019, 14:08:53)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The env program is looking for python3 in our environment. If Python has not been installed, then it won't be able to find it, but it also might be the case that Python has been installed more than once! We can use the which command to see which python3 it finds:

```
$ which python3
/Library/Frameworks/Python.framework/Versions/3.8/bin/python3
```

If I run this on repl.it, we see that python3 exists in a different place. Where does it exist on your computer?

```
$ which python3
/home/runner/.local/share/virtualenvs/python3/bin/python3
```

Just as my \$USER name is different from yours, my python3 is probably different from yours. If the env command is able to find a python3, it will execute it. As shown above, if you run python3 by itself, it will open a REPL.

If I were to put my python3 path as the shebang line like so:

```
#!/Library/Frameworks/Python.framework/Versions/3.8/bin/python3
```

Then my program would not work when I run it on another computer that has python3 installed in a different location. I doubt it would work on your computer, either. This why we should always use the env program to find the python3 that is specific to the machine on which it's running!

Now your program should look like this:

```
#!/usr/bin/env python3 ①
# Purpose: Say hello ②
print('Hello, World!') ③
```

- ① The shebang line telling the operating system to use the /usr/bin/env program to find python3 to interpret this program.
- ② A comment line documenting the purpose of the program.
- ③ A Python command to print some text to the screen.

1.4 Making a program executable

So far we've been explicitly telling `python3` to run our program, but, since we added the shebang, we can execute the program directly and let the OS figure out that it should use `python3`. The advantage is that we could copy our program to a place where other programs live and execute it from anywhere on our computer!

To do this, the first step is to make our program "executable" using the command `chmod` (*change mode*). Think of it like turning your program "on." Run this command to make `hello.py` executable:

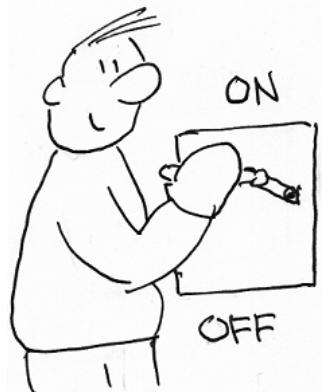
```
$ chmod +x hello.py ①
```

- ① The `+x` will add an "executable" attribute to the file.

Now you can run the program like so:

```
$ ./hello.py ①
Hello, World!
```

- ① The `./` is the current directory and is necessary to run a program when you are in the same directory as the program.



1.5 Understanding \$PATH

One of the biggest reasons to set the shebang line and make your program executable is so that you can install and run your Python programs just like other commands and programs. We used the `which` command earlier to find the location of `python3` on the repl.it instance:

```
$ which python3
/home/runner/.local/share/virtualenvs/python3/bin/python3
```

How was the `env` program able to find it? Windows, Mac, and Linux all have a `$PATH` variable that is a list of directories where the OS will look in to find a program. For instance, here is the `$PATH` my repl.it instance:

```
> echo $PATH
/home/runner/.local/share/virtualenvs/python3/bin:/usr/local/bin:\n/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

The directories are separated by the colon (:). Notice that the directory where `python3` lives is the first one in the `$PATH`. It's a pretty long string, so I broke it with the \ character to make it easier to read. If you copy your `hello.py` program to any of the directories listed in your `$PATH`, and then you can execute the program like `hello.py` — without the leading `./` and without having to be in the same directory as the

program!

Think about \$PATH like this: If you lose your keys in your house, would you start looking in the upperleftmost kitchen cabinet and work your way through each cabinet and then all the drawers where you keep your silverware and kitchen gadgets and then move on to your bathrooms and bedroom closets? Or would you start off by looking in places like the keyhooks beside the front door and then move on to search the pockets of your favorite jacket and your purse or backpack and then maybe under the couch cushions and so forth?

The \$PATH variable is a way of telling your computer to only look in places where executable programs can be found. The only alternative is for the OS to search *every directory*, and that could possibly take several minutes to even hours! You can control both the names of the directories in the \$PATH and their relative order so that the OS will find the programs you need.

It's very common for programs to be installed into /usr/local/bin, so we could try to copy our program there using the cp command. Unfortunately, I do not have permission to do this on repl.it:

```
> cp 01_hello/hello.py /usr/local/bin
cp: cannot create regular file '/usr/local/bin/hello.py': Permission denied
```

But I can on my own laptop:

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

I can verify that the program is found:

```
$ which hello.py
/usr/local/bin/hello.py
```

And now I can execute it from any directory on my computer:

```
$ hello.py
Hello, World!
```

1.6 Altering your \$PATH

Often we may find ourselves working on a computer that won't allow us to install programs into your \$PATH such as on repl.it. An alternative is to alter your \$PATH to include a directory where you can put your programs. For instance, I often create a bin directory in my home directory, which can often be written with the tilde (~).

On most computers, ~/bin would mean "the bin directory in my home directory." It's also common to see \$HOME/bin where \$HOME is the name of your home directory. Here is how I can create this directory on the repl.it machine, copy a program to it, and then add it to my \$PATH:

```
$ mkdir ~/bin
$ cp 01_hello/hello.py ~/bin
$ PATH=~/bin:$PATH
```

(1)
(2)
(3)

```
$ which hello.py
/home/runner/bin/hello.py
```

(4)
(5)

- ① Use the `mkdir` ("make directory") command to create `~/bin`.
- ② Use the `cp` command to copy the `01_hello/hello.py` program to the `~/bin` directory.
- ③ Put the `~/bin` directory first in our `$PATH`.
- ④ Use the `which` command to look for the `hello.py` program. If the steps above worked, the OS should now be able to find the program in one of the directories listed in our `$PATH`.

Now I can be in any directory:

```
$ pwd
/home/runner/tinypythoprojects
```

And I can run it:

```
$ hello.py
Hello, World!
```

While the shebang and the executable stuff all seem like a lot of work, the payoff is that you can create a Python program that can be installed onto your computer or anyone else's and run just like any other program.

1.7 Adding a parameter and help

Throughout the book, I'll use "string diagrams" to visualize the inputs and outputs of the programs we'll write. If we create one for our program as it is, there are no inputs and the output is always "Hello, World!"



It's not terribly interesting for our program to always say "Hello, World!" It would be nice if it could say "Hello" to something else like the entire "Universe." We *could* change the code to:

```
print('Hello, Universe')
```

But that would mean we'd have to change the code everytime we wanted to make it greet a different name. We'd like to have a way to change the *behavior* of the program without always having to change *the program itself*. We can do that by finding the parts of the program that we want to change — like the name to greet — and providing that value as an *argument* to our program. That is, we'd like our program to work like this:

```
$ ./hello.py Terra
Hello, Terra!
```

How would the person using our program know to do this? *It's our program's responsibility to provide a help message!* Most command-line programs will respond to arguments like `-h` and `--help` with helpful messages about how to use the programs. We need our program to print something like this:

```
$ ./hello.py -h
usage: hello.py [-h] name

Say hello

positional arguments:
  name        Name to greet ①

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

- ① Note that the name is called a *positional* argument.

To do this, we're going to use the `argparse` module. Modules are files of code we can bring into our programs. We can also learn to create modules to share our code with other people. There are hundreds to thousands of modules that you can use in Python, which is one of the reasons why it's so exciting to use the language!

The `argparse` module will "parse" the "arguments" to the program. To use it, change your program to look like the below. I really recommend you type everything yourself and don't copy and paste:

```
#!/usr/bin/env python3 ①
# Purpose: Say hello ②

import argparse ③

parser = argparse.ArgumentParser(description='Say hello') ④
parser.add_argument('name', help='Name to greet') ⑤
args = parser.parse_args() ⑥
print('Hello, ' + args.name + '!') ⑦

#!/usr/bin/env python3
# Purpose: Say hello

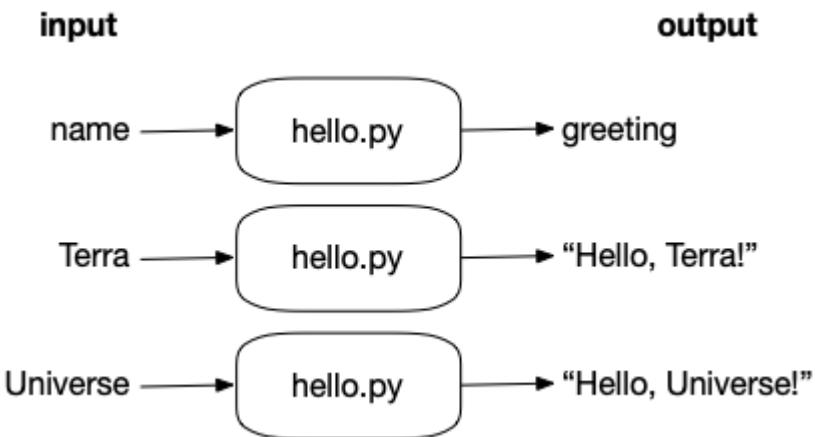
import argparse

parser = argparse.ArgumentParser(description='Say hello')

args = parser.parse_args()
print('Hello, ' + args.name + '!')
```

- ① The shebang line tells the OS which program to use to execute this program.
- ② This comment documents the purpose of the program.
- ③ We must import the `argparse` module in order to use it.
- ④ The parser will figure out all the arguments. The description appears in the help message.
- ⑤ We need to tell the parser to expect a name that will be the object of our salutations.
- ⑥ We ask the parser to parse any arguments to the program.
- ⑦ Print the greeting using the `args.name` value.

Here is a string diagram of our program now:



Now when you try to run the program like before, it triggers an error and a "usage" statement (notice that "usage" is the first word of the output):

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

- ① We run the program with no arguments, but the program now expects a single argument (a "name").
- ② Since the program doesn't get the expected argument, it stops and prints a "usage" to let the user know how to properly invoke the program
- ③ The error message tells the user that they have not supplied a required parameter called name.

We changed our program so that it requires a name or it won't run. That's pretty cool! Let's give it a name to greet:

```
$ ./hello.py Universe
Hello, Universe!
```

Try running your program with both `-h` and `--help` arguments and verify that you see the help messages. Our program works really well now and has nice documentation, all because we added those few lines using `argparse`. That's a big improvement!

1.8 Making the argument optional

What if I'd like run my program like before with no arguments and have it print "Hello, World!" We can make the `name` optional by changing the name of the argument to `--name`:

```
#!/usr/bin/env python3
# Purpose: Say hello

import argparse

parser = argparse.ArgumentParser(description='Say hello')
```

```
parser.add_argument('-n', '--name', metavar='name',      ①
                    default='World', help='Name to greet')
args = parser.parse_args()
print('Hello, ' + args.name + '!')
```

- ① The only change to this program is to add `-n` for the "short" and `--name` for the "long" option names. We also indicate a default value. The `metavar` will show up in the usage to describe the argument.

Now we can run it like before:

```
$ ./hello.py
Hello, World!
```

Or we can use the `--name` option:

```
$ ./hello.py --name Terra
Hello, Terra!
```

And now our help message has changed:

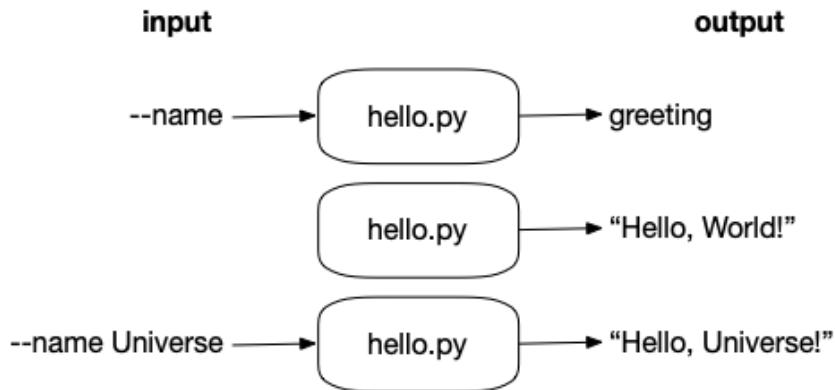
```
$ ./hello.py -h
usage: hello.py [-h] [-n NAME]

Say hello

optional arguments:
  -h, --help            show this help message and exit
  -n name, --name name  Name to greet ①
```

- ① Notice that the argument is now an *option* and no longer a *positional* argument. It's common to provide both short and long names to make it easy to type the options. The `metavar` value of "name" appears here to describe what the value should be.

Here is a string diagram that describes our program:



Our program is really flexible now, greeting a default value when run with no arguments or allowing us to say "hi" to something else. Remember that parameters that start with dashes are "optional," so they can be left out and may have default values. Parameters that *don't* start with dashes are "positional" and are usually required and so

do not have default values.

Table 1.1. Two kinds of command-line parameters.

Type	Example	Required	Default
Positional	name	Yes	No
Optional	-n (short), --name (long)	No	Yes

1.9 *Running our tests*

Let's run our tests again to see how close we are doing:

```
$ make test
pytest -xv test.py
=====
test session starts =====
...
collected 5 items

test.py::test_exists PASSED [ 20%]
test.py::test_runnable PASSED [ 40%]
test.py::test_executable PASSED [ 60%]
test.py::test_usage PASSED [ 80%]
test.py::test_input PASSED [100%]

===== 5 passed in 0.38s =====
```

Wow, we're passing all our tests! I actually get excited whenever I see my programs pass all their tests, even when I'm the one who wrote the tests. Before we were failing on the "usage" and "input" tests. Adding the argparse code fixed both of those because argparse allows us to accept arguments when our program runs and will also create documentation for how to run our program.

1.10 *Adding the main() function*

Our program works really well now, but it's not quite up to community standards and expectations. For instance, it's very common for computer programs — not just ones written in Python — to start at a place called `main()`. Most Python programs define a function called `main()`, and then there is an idiom in Python programs to call the `main()` function at the end of the code like this:

```
#!/usr/bin/env python3
# Purpose: Say hello

import argparse

def main(): ①
    parser = argparse.ArgumentParser(description='Say hello')
    parser.add_argument('-n', '--name', metavar='name',
                        default='World', help='Name to greet')
    args = parser.parse_args()
    print('Hello, ' + args.name + '!')

if __name__ == '__main__': ②
    main() ③
```

- ① The `def` is to "define" a function. The name of the function is `main()`. The empty parentheses show that this function accepts no arguments.
- ② Every program or module in Python has a name which can be accessed through the variable `__name__`. When the program is executing, then `__name__` is set to "`__main__`".¹
- ③ If this is true, then call the `main()` function.

As our programs get longer, we'll start creating more functions. To start off, we'll always put the main part of our program inside the `main()` function. You'll see that Python programmers will approach this in different ways, but I will always create and execute a `main()` function like this in every program in the book so as to be consistent.

1.11 Adding the `get_args()` function

As a matter of personal taste, I like to put all the `argparse` code into its own function that I always call `get_args()`. For some of my programs, this can get quite long, so I like to put it into a separate place function. Getting and validating the arguments is one idea in my mind, and so it belongs by itself.

I always put `get_args()` as the first function so that I can see it immediately when I read the source code. I usually put `main()` right after it. You are, of course, welcome to structure your programs however you like. Here is how the program looks now:

```
#!/usr/bin/env python3
# Purpose: Say hello

import argparse

def get_args():          ①
    parser = argparse.ArgumentParser(description='Say hello')
    parser.add_argument('-n', '--name', metavar='name',
                        default='World', help='Name to greet')
    return parser.parse_args() ②

def main():              ③
    args = get_args()      ④
    print('Hello, ' + args.name + '!')

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

- ① The `get_args()` function dedicated to getting the arguments. All the `argparse` code now lives here.
- ② We need to call `return` to send the results of parsing the arguments back to the `main()` function.
- ③ The `main()` function is much shorter now.
- ④ Call the `get_args()` function to get parsed arguments. If there is a problem with the arguments or if the user asks for `--help`, then the program never gets to this point because `argparse` will cause it to exit. If our program does make it this far, then the input values must have been OK!

Nothing has changed about the way the program works. We're just organizing the code to group ideas together — the code that deals with `argparse` now lives in the

¹ See docs.python.org/3/library/main.html for more information.

`get_args()` function, and everything else lives in `main()`.

1.11.1 Checking style and errors

Figure 1.2.
pylint makes
your code latty
fresh!



Our program works really well now. We can use tools like `flake8` and `pylint` tools to check if our program has problems. These tools are called "linters," and their job is to suggest ways to improve a program. If you haven't installed them yet, you can use the `pip` module to do so now:

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

The `flake8` program wants me to put two blank lines between each of the function def definitions:

```
$ flake8 hello.py
hello.py:6:1: E302 expected 2 blank lines, found 1
hello.py:12:1: E302 expected 2 blank lines, found 1
hello.py:16:1: E305 expected 2 blank lines after class or function
definition, found 1
```

And `pylint` says that the functions are missing documentation ("docstrings"):

```
$ pylint hello.py
*****
Module hello
hello.py:1:0: C0114: Missing module docstring (missing-module-docstring)
hello.py:6:0: C0116: Missing function or method docstring (missing-function-docstring)
hello.py:12:0: C0116: Missing function or method docstring (missing-function-docstring)

-----
Your code has been rated at 7.00/10 (previous run: -10.00/10, +17.00)
```

A "docstring" is a string or comment that occurs just after the `def` of the function. It's common to have several lines of documentation for a function, so programmers often will use Python's triple quotes (single or double) to create a multi-line string. Following is what the program looks like when I add docstrings. I have also used `yapf` to format the program and fix the spacing problems, but you are welcome to use `black` or any other tool you like:

```
#!/usr/bin/env python3
"""
Author: Ken Youens-Clark <kyclark@gmail.com>
Purpose: Say hello
"""

import argparse

# -----
def get_args():
    """Get the command-line arguments"""
    parser = argparse.ArgumentParser(description='Say hello')
    parser.add_argument('-n', '--name', default='World', help='Name to greet')
```

```

    return parser.parse_args()

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

    args = get_args()
    print('Hello, ' + args.name + '!')

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

```

- ① Triple-quoted, multi-line docstring for the entire program. It's common practice to write a long docstring just after the shebang to document the overall purpose of the function. I like to include at least my name, email address, the purpose of the script so that any future person using my program will know who wrote it, how to get in touch with me if they have problems, and what the program is supposed to do.
- ② A big horizontal "line" comment to help me find the functions. You can omit these if you don't like them.
- ③ The docstring for the get_args function. I like to use triple-quotes even for a single-line comment as they help me to see the docstring better.

To learn how to use yapf or black on the command line, run them with the -h or --help flag and read the documentation. If you are using an IDE like VSCode or PyCharm or if you are using the repl.it interface, there are commands to reformat your code.

1.12 Testing hello.py

We've made many changes to our program. Are we sure it still works correctly? Let's run our test again. Because this is something you will do literally hundreds of times, I've created a shortcut you might like to use. In every directory you'll find a file called Makefile that looks like this:

```

$ cat Makefile
.PHONY: test

test:
    pytest -xv test.py

```

If you have the program make installed on your computer, then you can run make test when you are in the 01_hello directory. The make program will look for a Makefile in your current working directory and then look for a recipe called "test." Here it will find that the command to run for the "test" target is pytest -xv test.py, and so it will run that command for us:

```

$ make test
pytest -xv test.py
===== test session starts =====
...

```

```
collected 5 items

test.py::test_exists PASSED [ 20%]
test.py::test_runnable PASSED [ 40%]
test.py::test_executable PASSED [ 60%]
test.py::test_usage PASSED [ 80%]
test.py::test_input PASSED [100%]

===== 5 passed in 0.75s =====
```

If you do not have `make` installed, you might like to install it and learn about how Makefiles can be used to execute complicated sets of commands. If you do not want to install or use `make`, then you can always run `pytest -xv test.py` yourself. They both accomplish the same task.

The bigger point to make, though, is that we were able to use our tests to verify that our program still does exactly what it was supposed to do. As you write these programs, you may want to try different solutions. The tests give you the freedom to rewrite your programs and still know that they are working.

1.13 Starting a new program with new.py

The `argparse` module is a standard module that is always installed with Python. It's widely used because it can save us so much time in parsing and validating the arguments to our program. You'll be using `argparse` in every program for this book, and you'll learn about how we can use it to convert text to numbers and validate and open files and much more. There are so many options that I created a Python program called `new.py` that helps you start writing new Python programs that use `argparse`.



I have put my `new.py` program into the `bin` directory of the GitHub repo. I suggest you start every new program with this program. For instance, you could create a new version of `hello.py` using the `new.py`. Go to the top level of your repository and run this:

```
$ bin/new.py 01_hello/hello.py
"01_hello/hello.py" exists. Overwrite? [yN] n
Will not overwrite. Bye!
```

The `new.py` will not overwrite an existing file unless we tell it to, so you can use this without worrying that you might erase your work. Try using it to create a different program name:

```
$ bin/new.py 01_hello/hello2.py
```

```
Done, see new script "01_hello/hello2.py."
```

Now try executing that program:

```
$ 01_hello/hello2.py
usage: hello2.py [-h] [-a str] [-i int] [-f FILE] [-o] str
hello2.py: error: the following arguments are required: str
```

Let's look at the new program:

```
#!/usr/bin/env python3 ①
"""
Author : Ken Youens-Clark <kyclark@gmail.com>
Date   : 2020-02-28
Purpose: Rock the Casbah
"""

import argparse      ③
import os
import sys

# -----
def get_args():      ④
    """Get command-line arguments"""

    parser = argparse.ArgumentParser(
        description='Rock the Casbah',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('positional', ⑤
                        metavar='str',
                        help='A positional argument')

    parser.add_argument('-a',          ⑥
                        '--arg',
                        help='A named string argument',
                        metavar='str',
                        type=str,
                        default='')

    parser.add_argument('-i',          ⑦
                        '--int',
                        help='A named integer argument',
                        metavar='int',
                        type=int,
                        default=0)

    parser.add_argument('-f',          ⑧
                        '--file',
                        help='A readable file',
                        metavar='FILE',
                        type=argparse.FileType('r'),
                        default=None)

    parser.add_argument('-o',          ⑨
                        '--on',
                        help='A boolean flag',
                        action='store_true')

    return parser.parse_args() ⑩
```

```

# -----
def main():          ⑪
    """Make a jazz noise here"""

    args = get_args()      ⑫
    str_arg = args.arg     ⑬
    int_arg = args.int
    file_arg = args.file
    flag_arg = args.on
    pos_arg = args.positional

    print(f'str_arg = "{str_arg}"')
    print(f'int_arg = "{int_arg}"')
    print('file_arg = "{}".format(file_arg.name if file_arg else "")')
    print(f'flag_arg = "{flag_arg}"')
    print(f'positional = "{pos_arg}"')

# -----
if __name__ == '__main__':  ⑭
    main()               ⑮

```

- ① The shebang line should use the env program to find the python3 program.
- ② This docstring is for the program as a whole.
- ③ These lines import various modules the program needs.
- ④ The `get_args()` function is responsible for parsing and validating arguments.
- ⑤ Define a "positional" argument like our first version of `hello.py` that had a `name` argument.
- ⑥ Define an "optional" argument like when we changed to the `--name` option.
- ⑦ Define an optional argument that must be an integer value.
- ⑧ Define an optional argument that must be a file.
- ⑨ Define a "flag" option that is either "on" when present or "off" when absent. We'll learn more about these later.
- ⑩ Return the parsed arguments to `main()`. If there are any problem like the `--int` value was some text rather than a number like 42, then `argparse` will print an error message and the "usage" for the user.
- ⑪ Define the `main()` function where the program starts.
- ⑫ The first thing our `main()` functions will always do is to call `get_args()` to get the arguments.
- ⑬ Each argument's value is accessible through the "long" name of the argument. It is not a requirement to have both a short and long name, but it is common and tends to make your program more readable.
- ⑭ When the program is being executed, the `__name__` value will be equal to the text "`__main__`".
- ⑮ If the condition is true, then call the `main()` function.

The arguments that this program will accept are:

1. A single positional argument of the type `str`. *Positional* means it is not preceded by a flag to name it but has meaning because of its position.
2. An automatic `-h` or `--help` flag that will cause `argparse` to print the usage.
3. A string option called either `-a` or `--arg`
4. A named option argument called `-i` or `--int`
5. A file option called `-f` or `--file`
6. A boolean (off/on) flag called `-o` or `--on`

Looking at the above, we can see that they new.py program has done the following for you:

1. Created a new Python program called hello2.py
2. Used a template to generate a working program complete with docstrings, a `main()` function to start your program, a `get_args` function to parse and document various kinds of arguments, and the code to start your program running in the `main()` function.
3. Made your program executable so that it can be run like `./hello2.py`.

The result is a program that you can immediately execute and which will produce documentation on how to run it. After you use new.py to start your new program, you should open it with your editor and modify the argument names and types to suit the needs of your program. For instance, in the "Crow's Nest" chapter, you can delete everything but the positional argument which you should rename from '`positional`' to something like '`word`' (because the argument is going to be a word).

Note that you can control the `name` and `email` values that are used by new.py by creating a file called `.new.py` (note the leading dot!) in your home directory. Here is mine:

```
$ cat ~/.new.py
name=Ken Youens-Clark
email=kyclark@gmail.com
```

1.14 Using template.py as an alternative to new.py

If you don't want to use new.py, then I have included a sample of the above program as `template/template.py` that you can copy. For instance, in the "Crow's Nest" chapter you should create the program `02_crowsnest/crowsnest.py`. Either you can do this with new.py from the top level of the repository:

```
$ bin/new.py 02_crowsnest/crowsnest.py
```

Or the use `cp` (copy) command to copy the template to your new program:

```
$ cp template/template.py 02_crowsnest/crowsnest.py
```

The main point is that I don't want you to have to start every program from scratch! I think it's much easier to start from a complete, working program and modify it.

NOTE

You could copy the new.py program to your `~/bin` directory, and then you would be able to use it from any directory to create a new program.

Be sure to skim the Appendix that has many examples of programs that use `argparse`. You can copy many of those examples to help you with the exercises!

1.15 Summary

- A Python program is plain text that lives in a file. We need the `python3` program

to interpret and execute the program file.

- You can make a program executable and copy it to a location in your \$PATH so that you can run it like any other program on your computer. Be sure to set the shebang to use the env program to find the correct python3.
- The argparse module will help you document and parse all the parameters to your program. You can validate the types and numbers of arguments which can be positional, optional, or flags. The usage will be automatically generated.
- We will use the pytest program to run the test.py programs for each exercise. The make test shortcut will execute pytest -xv test.py or you can run this command directly.
- You should run your tests often to ensure that everything works.
- Code formatters like yapf and black will automatically format your code to community standards, making it easier to read and debug.
- Code linters like pylint and flake8 can help you correct both programmatic and stylistic problems.
- You can use the new.py program to generate new Python programs that use argparse.

The Crow's Nest: Working with strings



Avast, you corny-faced gollumpus! Ye are barrelman for this watch. D'ye ken what I mean, ye addle pated blunderbuss?! Ah, land lubber ye be! OK, then, you are the lookout in the crow's nest — the little bucket attached to the top of a mast of a sailing ship. Your job is to keep a lookout for interesting or dangerous things, like a ship to plunder or an iceberg to avoid. When you see something like a "narwhal," you are supposed to cry out, "Ahoy, Captain, **a narwhal!** off the larboard bow!" If you see an octopus, you'll shout "Ahoy, Captain, **an octopus!** off the larboard bow!" (We'll assume everything is "off the larboard bow" for this exercise. It's a great place for things to be.)



From this point on, I will describe a coding challenge that you should write on your own. I will discuss key ideas you'll need to solve the problems as well as how to use the provided tests to help you know when your program is correct. You should have a copy of the Git repository locally (see the setup instructions). You should write your program in the chapter's directory, like this program should be written in the `02_crowsnest` directory where the tests for the program live.

In this chapter, we're going to start off working with strings. By the end, you will be

able to:

- Create a program that accepts a positional argument and produces usage documentation
- Create a new output string depending on the inputs to the program
- Run a test suite

Your program should be called `crowsnest.py`. It will accept a single positional argument and will print the given argument inside the "Ahoy" bit along with the word "a" or "an" depending on whether the argument starts with a consonant or a vowel.

That is, if given "narwhal," it should do this:

```
$ ./crowsnest.py narwhal
Ahoy, Captain, a narwhal off the larboard bow!
```

And if given "octopus":

```
$ ./crowsnest.py octopus
Ahoy, Captain, an octopus off the larboard bow!
```

This means we're going to need to write a program that accepts some input on the command line, decides on the proper article ("a" or "an") for the input, and prints out a new string that puts those two values into the "Ahoy" phrase.

2.1 Getting started

You're probably ready to start writing the program! Well, hold on just a minute longer, ye duke of limbs. We need to discuss how we'll use the tests to know when our program is working and how we might get started programming.

2.1.1 How to use the tests

"The greatest teacher, failure is." — Yoda

In the code repository, I've included tests that will guide you in the writing of your program. Before you even write the first line of code, I'd like you run the tests so you we can look at the first failed test:

```
$ cd 02_crowsnest
$ make test
```

Instead of `make test` you can also run `pytest -xv test.py`. Among all the output, you'll notice this line:

```
$ pytest -xv test.py
===== test session starts ====== ①
...
collected 6 items

test.py::test_exists FAILED [ 16%] ②
```

① This the the test start of the output from pytest.

- ② This test FAILED. There are more tests after this, but testing stops here because of the -x flag to pytest.

If you read more, you'll see lots of other output all trying to convince you that the expected file, `crowsnest.py` does not exist. Learning to read the test output is a skill in itself! It takes quite a bit of practice to learn to read test output, so try not to feel overwhelmed. In my terminal (iTerm on a Mac), the output from pytest shows colors and bold print to highlight key failures. The text in bold, red letters is usually where I start, but your terminal may behave differently.

Let's take a gander at the output. It does look a bit daunting at first, but you'll get used to read the messages and finding what needs to be fixed:

```
===== FAILURES =====
test_exists

def test_exists():          ①
    """exists"""

>     assert os.path.isfile(prg)   ②
E     AssertionError: assert False ③
E         + where False = <function isfile at 0x1086f1310>('./crowsnest.py')
E         + where <function isfile at 0x1086f1310> = <module 'posixpath' from
E             + where <module 'posixpath' from
'E             /Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8 posixpath.py'>.isfile
'E             /Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8 posixpath.py' = os.path

test.py:22: AssertionError
!!!!!!!!!!!!!! stopping after 1 failures !!!!!!! ④
===== 1 failed in 0.05s =====
```

- ① This is the actual code inside `test.py` that is running. It's a function called `test_exists`.
- ② The `>` at the beginning of this line indicates this is the line where the error starts. The test is checking if there is a file called `crowsnest.py`. If you haven't created it, this will fail as expected.
- ③ The `E` at the beginning of this line is the "Error" you should read. It's very difficult to understand what the test is trying to tell you, but essentially the `'./crowsnest.py'` file does not exist.
- ④ A warning that no more tests will run after the one failure.

The first test for every program in the book checks that the expected file exists, so let's create it!

2.1.2 *Creating programs with new.py*

In order to pass this test, we need to create a file called `crowsnest.py` inside the `02_crowsnest` directory where `test.py` is located. While it's perfectly fine to start writing from scratch, I suggest you use the `new.py` program to print some useful boilerplate code that you'll need in every exercise. From the top level of the repository, you can run this command to create the new program:

```
$ bin/new.py 02_crowsnest/crowsnest.py
Done, see new script "02_crowsnest/crowsnest.py."
```

If you don't want to use new.py, you could copy the template/template.py program:

```
$ cp template/template.py 02_crowsnest/crowsnest.py
```

At this point you should have the outline of a working program that accepts command-line arguments. If you run your the with no arguments, it will print a short usage statement like the following (notice how "usage" is the first word of the output):

```
$ ./crowsnest.py
usage: crowsnest.py [-h] [-a str] [-i int] [-f FILE] [-o] str
crowsnest.py: error: the following arguments are required: str
```

Run it like `./crowsnest.py --help` and see that it will print a longer help message, too.

NOTE

Those are not the correct parameters for our program, just the default examples given to you by new.py. We need to modify them to suit this program.

2.1.3 Write, test, repeat

We just created the program, so we ought to be able to pass the first test. If you run your tests again, you will in fact pass the first *two* tests:

4. Does the program exist? Yes, we just created it.
5. Does the program print a help message when we ask for help? Yes, we ran it above with no arguments and the `--help` flag and saw that it will produce help messages.

The cycle I hope you'll develop is to write a very small amount of code — literally one or two lines at most! — and then run the program or the tests to see how you're doing. Let's run our tests again:

```
$ make test
pytest -xv test.py
===== test session starts =====
...
collected 6 items

test.py::test_exists PASSED [ 16%] (1)
test.py::test_usage PASSED [ 33%] (2)
test.py::test_consonant FAILED [ 50%] (3)
```

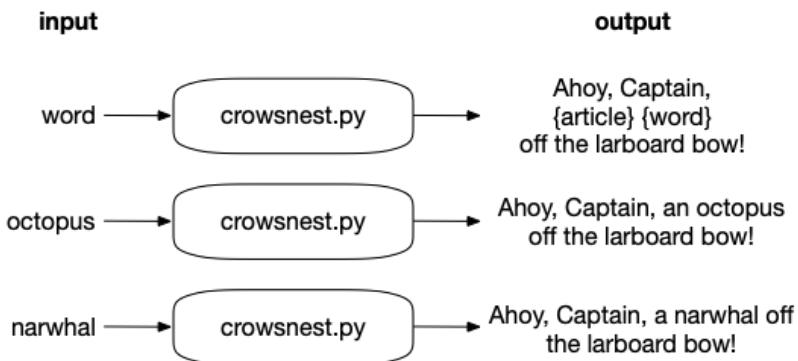
- (1) The expected file exists, so this test passes.
- (2) The program will respond to `-h` and `--help`. The fact that the help is actually *incorrect* is not important at this point. At this point, the test are only checking that you seem to have the outline of a program that will run and process the "help" flags.
- (3) The `test_consonant` test is failing. That's OK! We haven't even started writing the actual program, but at least we have a place to start!

Now we have a working program that accepts some arguments (but not the right ones). Next we need to make our program accept the "narwhal" or "octopus" value that needs to be announced, and we'll use command-line arguments to do that.

2.1.4 Defining your arguments

Here is a diagram sure to shiver your timbers showing the inputs (or *parameters*) and output of the program. We'll use these throughout the book to imagine how code and data work together. In this program, some "word" is the input, and a phrase incorporating that word with the correct article is the output.

Figure 2.1. The input to the program is a word, and the output is that word plus its proper article (and some other stuff).



We need to modify the part of the program that gets the arguments — the aptly named `get_args` function. This function uses the `argparse` module to parse the command-line arguments, and our program needs to take a single, positional argument. If you're unsure what a "positional" argument is, be sure to read the appendix on `argparse`, especially the example "A single, positional argument."

The default `get_args` created by our template names the first argument '`positional`'. Remember that positional arguments are defined by their position and don't have names that start with dashes. You can delete all the arguments except for the positional word. Modify the `get_args` part of your program until it will print this usage:

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

Likewise, it should print a longer usage for the `-h` or `--help` flag:

```
$ ./crowsnest.py -h
usage: crowsnest.py [-h] word

Crow's Nest -- choose the correct article

positional arguments:
  word      A word ①

optional arguments:
  -h, --help  show this help message and exit ②
```

① You need to define a `word` parameter. Notice that it is listed as a "positional" argument.

- ② The `-h` and `--help` flags are created automatically by argparse. You are not allowed to use these as options. They are used to create the documentation for your program.

Do not proceed until your usage matches the above!

When your program prints the correct usage, you can get the word argument inside the `main` function. Modify your program so that it will print the word:

```
def main():
    args = get_args()
    word = args.word
    print(word)
```

And then test that it works:

```
$ ./crowsnest.py narwhal
narwhal
```

And now run your tests. You should still be passing two and failing the third. Let's read the test failure:

```
===== FAILURES =====
test_consonant

def test_consonant():
    """brigatine -> a brigatine"""

    for word in consonant_words:
        out = getoutput(f'{prg} {word}')
>       assert out.strip() == template.format('a', word)
E           AssertionError: assert 'brigatine' == 'Ahoy, Captai...larboard bow!'
E               - brigatine
E               + Ahoy, Captain, a brigatine off the larboard bow!
```

- ① It's not terribly important right now to understand this line, but the `getoutput` function is running the program with a word. We're going to talk about the f-string in this chapter. The output from running the program will go into the `out` variable which will be used to see if the program created the correct output for a given word. None the code in this function is anything you should worry about being able to write yet!
- ② The line starting with `>` shows code that produced an error. The output of the program is compared to an expected string. Since it didn't match, the `assert` produces an exception.
- ③ This line starts with `E` to indicate the "error."
- ④ The line starting with `a` - is what the test got when it ran with the argument '`brigatine`' — it got back just the word "brigatine."
- ⑤ The line starting with the `+` is what the test expected, "Ahoy, Captain, a brigatine off the larboard bow!"

So, we need to get the word into the "Ahoy" phrase. How can we do that?

2.1.5 Concatenating strings

Putting strings together is called "concatenating" or "joining" strings. To demonstrate, I'm going to enter some code directly into the Python interpreter. I want you to type

along. No, really! Type everything you see, and try it for yourself.

Open a terminal and type `python3` or `ipython` to start a REPL, a "Read-Evaluate-Print-Loop" because Python will *read* each line of input, *evaluate* and *print* the results in a *loop*. Here's what it looks like on my system:

```
$ python3
Python 3.8.1 (v3.8.1:1b293b6006, Dec 18 2019, 14:08:53)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

You may also like to use Python's IDLE (integrated development and learning environment) program, `ipython`, or Jupyter Notebooks to interact with the language. I'll stick to the `python3` REPL throughout the book. To exit the REPL, either type `quit()` or `CTRL-d` (the Control key plus the d).

The `>>>` is a prompt where you can type code. Remember *not* to type that part! Let's start off by assigning the variable `word` to the value "narwhal." In the REPL, type `word = 'narwhal'<Enter>`:

```
>>> word = 'narwhal'
```

Note that you can put as many (or no) spaces around the `=` as you like, but convention and readability (and tools like `pylint` or `flake8` that help you find errors in your code) would ask you to use exactly one space on either side. If you type `word<Enter>`, Python will print the current value of `word`:

```
>>> word
'narwhal'
```

Now type `werd<Enter>`:

```
>>> werd
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'werd' is not defined
```

WARNING

There is no `werd` variable because we haven't set `werd` to be anything. Using an undefined variable causes an exception that will crash your program. Python will happily create a `werd` for you when you assign it a value.

In Python, there are many ways we can concatenating strings. The `+` operator can be used to join strings together:

```
>>> 'Ahoy, Captain, a ' + word + ' off the larboard bow!'
'Ahoy, Captain, a narwhal off the larboard bow!'
```

If you change your program to print that instead of just the `word`, you should be able to four tests:

<code>test.py::test_exists PASSED</code>	[16%]
<code>test.py::test_usage PASSED</code>	[33%]
<code>test.py::test_consonant PASSED</code>	[50%]

```
test.py::test_consonant_upper PASSED [ 66%]
test.py::test_vowel FAILED [ 83%]
```

If we look closely at the failure, you'll see this:

```
E           - Ahoy, Captain, a aviso off the larboard bow!
E           + Ahoy, Captain, an aviso off the larboard bow!
E           ?           +
```

So we hard-coded the "a" before the word, but we really need to figure out whether to put "a" or "an" depending on whether the word starts with a vowel. How can we do that?

2.1.6 Variable types

Before we go much further, I need to take a small step back and point out that our word variable is a "string." Every variable in Python has a "type" that describes the kind of data they hold. Because we put the value for word in quotes ('narwhal'), the word holds a "string" which Python represents with a class called `str`. (A "class" is a collection of code and functions that we can use.)

The `type` function will tell us what kind of data Python thinks this is:

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

Whenever you put a value in single (' ') or double quotes (""), Python will interpret it as a `str`:

```
>>> type("submarine")
<class 'str'>
```

WARNING

If you forget the quotes, then Python will look for some variable or function by that name. If there is no variable or function by that name, it will cause an exception.

```
>>> word = narwhal
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'narwhal' is not defined
```

Exceptions are bad, and we will try to write code that avoids them or at least knows how to handle them gracefully.

2.1.7 Getting just part of a string

Back to our problem! We need to put either "a" or "an" in front of the word we're given based on whether the first character of word is a vowel or a consonant. In Python, we use square brackets and an *index* to get an individual character from a string. The index is the numeric position of an element in a sequence, and we must remember that indexing starts at 0.

```
>>> word[0]
'n'
```

Or directly on a string:

```
>>> 'narwhal'[0]
'n'
```

You can use this with a variable:

n a r w h a l
0 1 2 3 4 5 6



n a r w h a l
0 1 2 3 4 5 6



This means that the last index is *one less than the length*, which is often confusing. The length of "narwhal" is 7, but the last character is found at index 6:

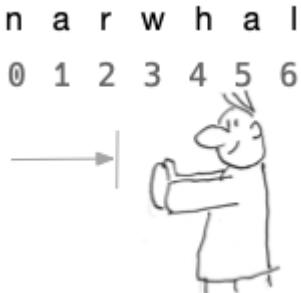
```
>>> word[6]
'l'
```

You can also use negative numbers to count backwards from the end, so the last index is also -1:

```
>>> word[-1]
'l'
```

n a r w h a l
-6 -5 -4 -3 -2 -1





You can use the "slice" notation `[start:stop]` to get a range of characters. Both `start` and `stop` are optional. The default value for `start` is `0` (the beginning of the string), and the `stop` value is *not inclusive*:

```
>>> word[:3]
'nar'
```

And the default value for `stop` is the end of the string:

```
>>> word[3:]
'whal'
```

In the next chapter, we'll see that this is the same syntax for slicing lists. A string is (sort of) a list of characters, so this isn't too strange.

2.1.8 Finding help in the REPL

The class `str` has a ton of functions we can use to handle strings, but what are they? A large part of programming is knowing how to ask questions and where to look for answers. A common refrain you may hear is "RTFM" — Read the Fine Manual. The Python community has created reams of documentation which are all available at docs.python.org/3/. You will need to refer to the documentation constantly to remind yourself how to use certain functions.

The docs for the string class are here:

docs.python.org/3/library/string.html

I prefer to read the docs directly inside the REPL by typing `help(str)`:

```
>>> help(str)
```



Inside the `help`, you move up and down in the text using the up and down cursor arrows on your keyboard. You can also press the `<Space>` bar or the letter `f` (or sometime `CTRL-f`) to jump forward to the next page, and the letter `b` (or sometimes `CTRL-b`) to jump backward. You can search through the documentation by pressing `/` and then the text you want to find. If you press `n` (for "next") after a search, you will jump to the next place that string is found. To leave the help, press `q` (for "quit").

2.1.9 String methods



Now that we know `word` is a string (`str`), we have all these incredibly useful *methods* we can call on the variable. (A "method" is a function that belongs to a variable like `word`.) For instance, if I wanted to shout about the fact that we have a "narwhal," I could print it in UPPERCASE LETTERS. If I search through the help, I see there is a function called `upper`. Here is how to call it:

```
>>> word.upper()
'NARWHAL'
```

You must include the parentheses () or else you're talking about the *function itself*:

```
>>> word.upper
<built-in method upper of str object at 0x10559e500>
```

That will actually come in handy later when we use functions like `map()` and `filter()`, but for now we want Python to *execute* or *call* the `str.upper()` function on the variable `word`, so we add the parens. Note that the function returns an uppercase version of the word but *does not* change the value of `word` itself:

```
>>> word
'narwhal'
```

There is another `str` function with "upper" in the name called `str.isupper()`. The name helps you know that this will return a True/False type answer. Let's try it:

```
>>> word.isupper()
False
```

We can chain methods together like so:

```
>>> word.upper().isupper()
True
```

That makes sense. If I convert the `word` to uppercase, then `word.isupper()` returns `True`.

I find it odd that the `str` class does not include a method to get the length of a string. For that, we use a separate function called `len()`, short for "length":

```
>>> len('narwhal')
7
>>> len(word)
7
```

Are you typing all this into Python yourself? I recommend you do! Find other methods in the `str` help and try them out.



2.1.10 String comparisons

So now you know how to get the first letter of `word` by using `word[0]`. Let's assign it to the variable `char`:

```
>>> word = 'octopus'
>>> char = word[0]
>>> char
'o'
```

If you check the type of our new `char` variable, it is a `str`. Even a single character is still considered by Python to be a "string":

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

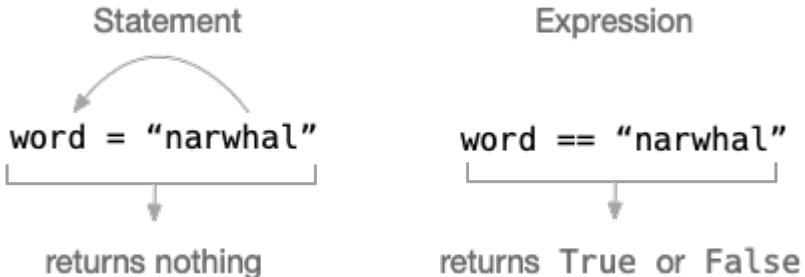
Now we need to figure out if `char` is a vowel or a consonant. We'll say that letters "a," "e," "i," "o," and "u" make up our set of "vowels." You can use `==` to compare strings:

```
>>> char == 'a'
False
>>> char == 'o'
True
```

NOTE

Be careful to always use one equal sign (`=`) when *assigning a value* to a variable, like `word = 'narwhal'` and two equal signs (`==`, which, in my head, I say "equal-equal") when you *compare two values* like `word == 'narwhal'`. The first is a statement that changes the value of `word`, and the second is an *expression* that returns `True` or `False`.

Figure 2.2. An expression returns a value. A statement does not.



We need to compare our `char` to *all* the vowels. You can use `and` and `or` in such comparisons and they will be combined according to standard Boolean algebra:

```
>>> char == 'a' or char == 'e' or char == 'i' or char == 'o' or char == 'u'
True
```

What if the word is "Octopus" or "OCTOPUS"?

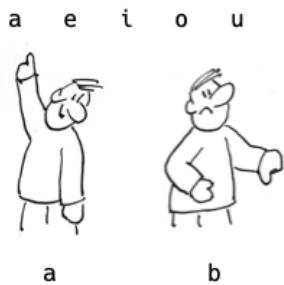
```
>>> word = 'OCTOPUS'
>>> char = word[0]
>>> char == 'a' or char == 'e' or char == 'i' or char == 'o' or char == 'u'
False
```

Do we have to make 10 comparisons in order to check the uppercase versions, too? What if we were to lowercase `word[0]`? Remember, that `word[0]` returns a `str`, and so we can chain other `str` methods onto that:

```
>>> word = 'OCTOPUS'
>>> char = word[0].lower()
>>> char == 'a' or char == 'e' or char == 'i' or char == 'o' or char == 'u'
True
```

An easier way to determine if `char` is a vowel would be to use Python's `x in y` construct where we want to know if the value `x` is in the collection `y`. We can ask if the letter '`a`' is in the longer string '`aeiou`':

```
>>> 'a' in 'aeiou'
True
```



But the letter '`b`' is not:

```
>>> 'b' in 'aeiou'
False
```

Let's use that to test the first character of the lowercased word (which is '`o`'):

```
>>> word = 'OCTOPUS'
>>> word[0].lower() in 'aeiou'
True
```

2.1.11 Conditional branching

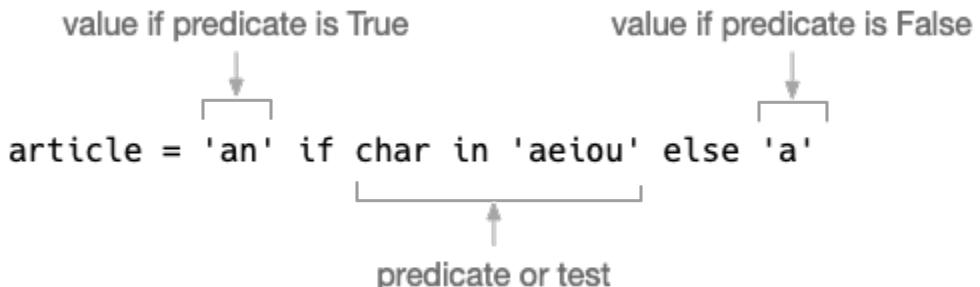
Once you have figured out if the first letter is a vowel, you will need to select an article. We'll use a very simple rule that, if the word starts with a vowel, choose "an," otherwise choose "a." This misses exceptions like when the initial "h" in a word is silent, for instance, we say "a hat" but "an honor". Nor will we consider when an initial vowel has a consonant sound as in "union" where the "u" sounds like a "y."

We can create a new variable called `article` that we will set to the empty string and then use an `if/else` statement to figure out what to put in it:

```
>>> article = ''                                ①
>>> if word[0].lower() in 'aeiou': ②
...     article = 'an'                          ③
... else:                                     ④
...     article = 'a'                           ⑤
... 
```

- ① Initialize `article` to the empty string.
- ② Check if the first, lowercased character of `word` is a vowel.
- ③ If it is, set `article` to 'an'
- ④ Otherwise,
- ⑤ Set `article` to 'a'.

Here is a much shorter way to write that with an `if expression` (expressions return values, statements do not). The `if` expression is written a little backwards. First comes the value if the test (or "predicate") is `True`, then the predicate, then the value if the predicate is `False`.



This way is also safer because the `if` expression is *required* to have the `else`. There's no chance that we could forget to handle both cases:

```
>>> article = 'an' if char in 'aeiou' else 'a'
```

Let's verify that we have the correct `article`:

```
>>> article
```

'an'

2.1.12 String formatting

Now we have two variables, `article` and `word` that need to be incorporated into our "Ahoy!" phrase. We saw earlier that we can use the plus sign (+) to concatenate strings. Another method to create new strings from other strings is to use the `str.format()` method. To do so, you create a string template with curly brackets {} that indicate placeholders for values. The values that will be substituted go as arguments to the `str.format()`, and they are substituted in the same order that the {} appear:

```
'Ahoy, Captain, {} {} off the larboard bow!'.format(article, word)
```

Here it is in code:

```
>>> 'Ahoy, Captain, {} {} off the larboard bow!'.format(article, word)
'Ahoy, Captain, an octopus off the larboard bow!'
```

Another method uses the special "f-string" where you can put the variables directly into the {} brackets. It's a matter of taste which one you choose.

```
>>> f'Ahoy, Captain, {article} {word} off the larboard bow!'
'Ahoy, Captain, an octopus off the larboard bow!'
```

Python variables are very variable

A note that in some programming languages, you have to declare the variable's name and what type of data it will hold. If a variable is declared to be a number, then it can never hold a value of a different type like a string. This is called *static typing* because the type of the variable can never change. Python is a *dynamically typed* language because you do not have to declare a variable or what kind of data the variable will hold. You can change the value and type of data at any time. This could be either great or terrible news. As Hamlet says, "There is nothing either good or bad, but thinking makes it so."



Hints:

- Start your program with `new.py` and fill in the `get_args()` with a single position argument called `word`.
- You can get the first character of the word by indexing it like a list, `word[0]`.
- Unless you want to check both upper- and lowercase letters, you can use either the `str.lower()` or `str.upper()` method to force the input to one case for

checking if the first character is a vowel or consonant.

- There are fewer vowels (five, if you recall) than consonants, so it's probably easier to check if the first character is one of those.
- You can use the `x in y` syntax to see if the element `x` is in the collection `y` where "collection" here is a `list`.
- Use the `str.format()` or f-strings to insert the correct article for the given word into the longer phrase.
- Run `make test` (or `pytest -xv test.py`) *after every change to your program* to ensure your program compiles and is on the right track.

Now go write the program before you turn the page and study a solution! Look alive, you ill-tempered shabaroon!

2.2 Solution

```
#!/usr/bin/env python3
"""Crow's Nest"""

import argparse

# -----
def get_args():          ①
    """Get command-line arguments"""

    parser = argparse.ArgumentParser(          ②
        description="Crow's Nest -- choose the correct article", ③
        formatter_class=argparse.ArgumentDefaultsHelpFormatter) ④

    parser.add_argument('word', metavar='word', help='A word') ⑤

    return parser.parse_args()                                ⑥

# -----
def main():           ⑦
    """Make a jazz noise here"""

    args = get_args()          ⑧
    word = args.word          ⑨
    article = 'an' if word[0].lower() in 'aeiou' else 'a'      ⑩

    print(f'Ahoy, Captain, {article} {word} off the larboard bow!') ⑪

# -----
if __name__ == '__main__': ⑫
    main()                ⑬
```

- ① Defines the function `get_args()` to handle the command-line arguments. I like put this first so I can see it right away when I'm reading the code.
- ② The parser will do the work of parsing the arguments.
- ③ The description shows in the usage to describe what the program does.
- ④ Show the default values for each parameter in the usage.

- ⑤ Define a positional argument called word.
- ⑥ The result of parsing the arguments will be returned to line 24.
- ⑦ Defines the `main()` function where the program will start.
- ⑧ `args` contains the return value from the `get_args()` function.
- ⑨ Put the `args.word` value from the arguments into the variable `word`.
- ⑩ Choose the correct article using an `if` expression to see if the lowercased, first character of `word` is or is not in the set of vowels.
- ⑪ Print the output string using an f-string to interpolate the `article` and `word` variables inside the string.
- ⑫ Check if we are in the "main" namespace, which means the program is *running*.
- ⑬ If so, call the `main()` function to make the program start.

2.3 Discussion

I'd like to stress that the preceding is *a* solution, not *the* solution. There are many ways to express the same idea in Python. As long as your code passes the test suite, it is correct.

That said, I created my program with `new.py` which automatically gives me two functions:

1. `get_args()` where I define the arguments to the program
2. `main()` where the program starts

Let's talk about these two functions.

2.3.1 Defining the arguments with `get_args`

I prefer to put the `get_args()` function first so that I can see right away what the program expects as input. You don't have to define this as a separate function. You could put all this code inside `main()`, if you prefer. Eventually our programs are going to get longer, though, and I think it's nice to keep this as a separate idea. Every program I present will have a `get_args()` function that will handle defining and validating the input.

Our program specifications (the "specs") say that the program should accept one positional argument. I changed the 'positional' argument name to 'word' because I'm expecting a single word:

```
parser.add_argument('word', metavar='word', help='Word')
```

I would really recommend you never leave the "positional" argument named 'positional' because it is an entirely undescriptive term. Naming your variables *what they are* will make your code more readable. Since the program doesn't need any of the other options created by `new.py`, you can delete the rest of the `parser.add_argument()` calls. The `get_args()` function will return the result of parsing the command line arguments which I put into the variable `args`:

```
return parser.parse_args()
```

If argparse is not able to parse the arguments — for example, there are none — it will never return from `get_args()` but will instead print the "usage" for the user and exit with an error code to let the operating system know that the program exited without success. (On the command line, an exit value of 0 means there were 0 errors. Anything other than 0 is considered an error.)

2.3.2 **The main thing**

Many programming languages will automatically start from the `main()` function, so I always define a `main()` function and start my programs there. This is not a requirement, just how I like to write programs. Every program I present will start with the `main()` function which will first call `get_args()` to get the program's inputs:

```
args = get_args()
```

I can now access the word by call `args.word`. Note the lack of parentheses. It's not `args.word()` because is not a function call. Think of `args.word` like a slot where the value of the "word" lives:

```
word = args.word
```

I like to work through my ideas using the REPL, so I'm going to pretend that `word` has been set to "octopus":

```
>>> word = 'octopus'
```

2.3.3 **Classifying the first character of a word**

To figure out whether the article I choose should be a or an, I need to look at the first character of the word which we can get like so. In the introduction, we used this:

```
>>> word[0]
'o'
```

I can check if the first character is in the string of vowels, both lower- and uppercase:

```
>>> word[0] in 'aeiouAEIOU'
True
```

I can make this shorter, however, if I use `word.lower()` function so I'd only have to check the lowercase vowels:

```
>>> word[0].lower() in 'aeiou'
True
```

Remember that the `x in y` form is a way to ask if element `x` is in the collection `y`. You can use it for letters in a longer string (like the vowels):

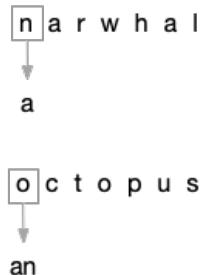
```
>>> 'a' in 'aeiou'
True
```

Or for a string in list of other strings:

```
>>> 'tanker' in ['yatch', 'tanker', 'vessel']
True
```

We can use membership in the "vowels" as a condition to choose "an," otherwise we choose "a": As mentioned in the introduction, the `if` expression is the shortest and safest for a "binary" choice (where there are only two possibilities):

```
>>> article = 'an' if word[0].lower() in 'aeiou' else 'a'
>>> article
'an'
```



The safety comes from the fact that Python will not even run this program if you forget the `else`. We can change the `word` to "galleon" and check that it still works:

```
>>> word = 'galleon'
>>> article = 'an' if word[0].lower() in 'aeiou' else 'a'
>>> article
'a'
```

2.3.4 Printing the results

Finally we need to print out the phrase with our `article` and `word`. As noted in the introduction, you can use `str.format()`:

```
>>> article = 'a'
>>> word = 'ketch'
>>> print('Ahoy, Captain, {} {} off the larboard bow!'.format(article, word))
Ahoy, Captain, a ketch off the larboard bow!
```

Python's f-strings will *interpolate* any code inside the `{}` placeholders, so variables get turned into their contents:

```
>>> print(f'Ahoy, Captain, {article} {word} off the larboard bow!')
Ahoy, Captain, a ketch off the larboard bow!
```

However you chose to print out the `article` and `word` is fine as long as it passes the tests. While it's a matter of personal taste which you choose, I find f-strings a bit easier to read as my eyes don't have to jump back and forth from the `{}` placeholders to the variables that will go inside them.

2.3.5 Running the test suite

"A computer is like a mischievous genie. It will give you exactly what you ask for, but not always what you want. - Joe Sondow"

Computers are a bit like bad genies. They will do exactly what you tell them but not necessarily what you *want*. In an episode of *The X-Files*, the character Mulder wishes for peace on Earth and a genie removes all humans but him. Tests are what we can use to verify that our programs are doing what we *actually* want them to do. Tests they can never prove that our program is truly free from errors, only that the bugs we imagined

or found while writing the program no longer exist. Still, we write and run tests because they are really quite effective and much better than not doing so.

This is the idea behind "test-driven development":

- We can write tests *even before* we write the software.
- We run the tests to verify that our as-yet-unwritten software definitely fails to deliver on some task.
- Then we write the software to fulfill the request.
- Then we run the test to check that it now *does* work.
- We keep running all the tests to ensure that, when we add some new code, we do not break existing code.

We won't be discussing how to *write* our own tests just yet. That will come later. For now, I've written all the tests for you. I hope that by the end of this book, you will see the value of testing and will always start off by writing *tests first and code second!*

2.4 Summary

- All Python's documentation is available on docs.python.org/3/ and with the `help` command in the REPL.
- Variables in Python are dynamically typed according to whatever value you assign them and come into existence when you assign a value to them.
- Strings have methods like `str.upper()` and `str.isupper()` that you can call to alter them or get information.
- You can get parts of a string by using square brackets and indexes like `[0]` for the first letter or `[-1]` for the last.
- You can concatenate strings with the `+` operator.
- The `str.format()` method allows you to create a template with `{}` placeholders that get filled in with the arguments.
- F-strings like `f'{article} {word}'` allow variables and code to go directly inside the brackets.
- The `x in y` expression will report if the value `x` is present in the collection `y`.
- Statements like `if/else` do not return a value while expressions like `x if y else z` do return a value.
- Test-driven development is a way to ensure programs meet some minimum criteria of correctness. Every feature of a program should have tests, and writing and running test suites should be an integral part of writing programs.

2.5 Going Further

- Have your program match the case of the incoming word, e.g., "an octopus" and "An Octopus." Copy an existing `test_` function in the `test.py` to verify that your program works correctly while still passing all the other tests. Try writing the test first, then make your program pass the test. That's *test-driven development!*
- Accept a new parameter that changes "larboard" (the left side of the boat) to "starboard" (the right side.²). You could either make an option called `--side` that defaults to "larboard," or you could make a `--starboard` flag that, if present, changes the side to "starboard."
- The provided tests only give you words that start with an actual alphabetic character. Expand your code to handle words that start with numbers or punctuation. Should your program reject these? Add more tests to ensure that your program does what you intend.



² "Starboard" has nothing to do with stars but with the "steering board" or a rudder which typically would be on the right-side of the boat for right-handed sailors!



Going on a picnic: Working with lists

Writing code makes me hungry! Let's write a program to consider some tasty foods we'd like to eat. So far we've handled *one* of something like a name to say "hello" to or a nautical-themed object to point out. In this program, we want to eat one or more foods which we will store in a *list*, a variable that can hold any number of items. We use lists all the time in life. Maybe it's your top-five favorite songs, your birthday wish-list, or a bucket list of the best types of buckets.

In this exercise, we're going on a picnic, and we want to print a list of items to bring. You will learn to:

- Write a program that accepts multiple positional arguments.
- Use `if`, `elif`, and `else` to handle conditional branching with three or more options.
- Find and alter items in a list.
- Sort and reverse lists.
- Format a list into a new string.

The items will be passed as positional arguments. When there is only one item, you'll print that:

```
$ ./picnic.py salad
You are bringing salad.
```



What? Who just brings salad on a picnic? When there are two items, you'll put "and" in between them:

```
$ ./picnic.py salad chips
You are bringing salad and chips.
```



Hmm, chips. That's an improvement. When there are three or more items, you will separate the items with commas:

```
$ ./picnic.py salad chips cupcakes
You are bringing salad, chips, and cupcakes.
```

There's one other twist. You will also need to accept a --sorted argument that will require you to sort the items before you print them, but we'll deal with that in a bit. So, your Python program must:

- Store one or more positional arguments in a list
- Count the number of arguments
- Possibly modify the list like maybe to sort the items
- Use the list to print a new a string that formats the arguments according to how

many items there are.

How should we begin?

3.1 Starting the program

I will always recommend you start programming either by running new.py or by copying template/template.py to the program name. This time the program should be called picnic.py, and we need to create it in the 03_picnic directory. You can do this using the new.py program from the top level of your repository:

```
$ bin/new.py 03_picnic/picnic.py
Done, see new script "03_picnic/picnic.py."
```

Now go into the 03_picnic directory and run `make test` or `pytest -xv test.py`. You should pass the first two tests (program exists, program creates usage), and fail the third:

<code>test.py::test_exists PASSED</code>	[14%]
<code>test.py::test_usage PASSED</code>	[28%]
<code>test.py::test_one FAILED</code>	[42%]

The rest of the output is complaining about the fact that the test expected "You are bringing chips" but got something else:

```
===== FAILURES =====
test_one -----
def test_one():
    """one item"""

    out = getoutput(f'{prog} chips')
>     assert out.strip() == 'You are bringing chips.' ①
E     assert 'str_arg = "...nal = "chips"' == 'You are bringing chips.' ②
E         + You are bringing chips. ③
E         - str_arg = "" ④
E         - int_arg = "0"
E         - file_arg = ""
E         - flag_arg = "False"
E         - positional = "chips"

test.py:31: AssertionError
===== 1 failed, 2 passed in 0.56 seconds =====
```

- ① The program is being run with the argument "chips."
- ② This is the line that is causing the error. The output is tested to see if it is equal (==) to the string "You are bringing chips."
- ③ The line starting with a + sign is showing what was expected.
- ④ The lines starting with the - sign is showing what was returned by the program.

Let's run the program with the argument "chips" and see what it gets:

```
$ ./picnic.py chips
str_arg = ""
int_arg = "0"
```

```
file_arg = ""
flag_arg = "False"
positional = "chips"
```

Right, that's not correct at all! Remember, the template doesn't have the *correct* arguments, just some examples, so the first thing we need to do is to fix the `get_args()` function. Here is what your program should print a usage statement if given *no arguments*:

```
$ ./picnic.py
usage: picnic.py [-h] [-s] str [str ...]
picnic.py: error: the following arguments are required: str
```

And here is the usage for the `-h` or `--help` flags:

```
$ ./picnic.py -h
usage: picnic.py [-h] [-s] str [str ...]

Picnic game

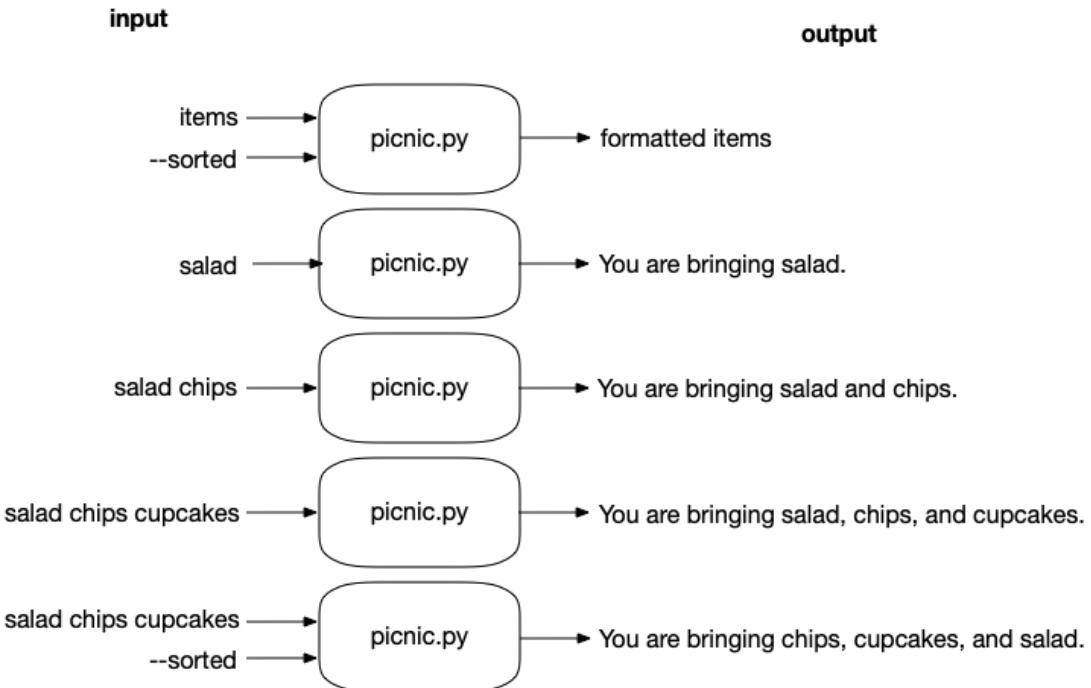
positional arguments:
  str          Item(s) to bring

optional arguments:
  -h, --help    show this help message and exit
  -s, --sorted  Sort the items (default: False)
```

We need a single positional argument and an optional flag called `--sorted`. Modify your `get_args()` until it produces the above output. Note that there should be one or more of the `item` parameter, so you should define it with `nargs='+'`. Refer to the section "One or more of the same positional arguments" in the the `argparse` appendix.

3.2 Writing `picnic.py`

Here is a tasty diagram of the inputs and outputs for the `picnic.py` program we'll write:



The program should accept one or more "positional" arguments as the items to bring on a picnic as well as a `-s` or `--sorted` flag to indicate whether or not to sort the items. The output will be "You are bringing" and then the list of items formatted according the following rules:

1. If one item, state the item:

```
$ ./picnic.py chips
You are bringing chips.
```

2. If two items, put "and" in between the items. Note that "potato chips" is just *one string* that happens to contain *two words*. If we leave out the quotes, then there would be three arguments to the program. Note that it doesn't matter here if we use single or double quotes:

```
$ ./picnic.py "potato chips" salad
You are bringing potato chips and salad.
```

3. If three or more items, place a comma and space between each item and the word "and" before the final element. Don't forget the comma before the "and" (sometimes called the "Oxford comma") because your author was an English lit major and, while I may have finally stopped using two spaces after the end of a sentence, you can pry the Oxford comma from my cold, dead hands:

```
$ ./picnic.py "potato chips" salad soda cupcakes
You are bringing potato chips, salad, soda, and cupcakes.
```

Be sure to sort if given the `-s` or `--sorted` flag:

```
$ ./picnic.py --sorted salad soda cupcakes
You are bringing cupcakes, salad, and soda.
```

In order to figure out how many items we have, how to sort and slice them, and how to format the output string, we need to talk about the `list` type in Python in order to solve this problem.

3.3 *Introduction to lists*

We're going to learn how to define the positional arguments so that they are available to us as a `list`. That is, if we run the program like this:

```
$ ./picnic.py salad chips cupcakes
```

Then the arguments `salad` `chips` `cupcakes` would be available as a `list` of the strings inside our program. If we print a `list` in Python, we'll see something like this:

```
['salad', 'chips', 'cupcakes']
```

The square brackets tell us this is a `list`, and the quotes are the elements tell us they are strings. Note that the items are shown in the same order as they were provided on the command line. Lists always keep their order!

['salad', 'chips', 'cupcakes']
↑
↑
↑
 square brackets
 mean a list

Let's go into the REPL and create a variable called `items` to hold some tasty foods we plan to bring on our picnic. I really want you to type these commands yourself, too, whether in the `python3` REPL or `ipython` or a Jupyter Notebook. It's very important to interact in real time with the language!

To create a new, empty list, we can either use the `list()` function:

```
>>> items = list()
```

Or use empty square brackets:

```
>>> items = []
```

Check what Python says for the type. Yep, it's a `list`:

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

One of the first things we need to know is how many `items` we have for our picnic. Like a `str`, we can use `len()` (length) to get the number of elements in `items`:

```
>>> len(items)
```

0

The length of an empty list is 0.

3.3.1 Adding one element to a list

An empty list is not very useful. Let's see how we can add new items. We used `help(str)` in the last chapter to read the documentation about the string *methods*, the functions that belong to every `str` in Python. Here I want you to use `help(list)` to find the list methods:

```
>>> help(list)
```

Remember that the Space bar or f (or Ctrl-f) will go "forward" while b (or Ctrl-b) will go back, and the / will let you search for a string. You'll see lots of "double-under" methods like `_len_`. Skip over those, and the first method we can find is `list.append()`, which we can use to add items to the end of the list. If we evaluate our `items`, we see that the empty brackets tell us that it's empty:

```
>>> items
[]
```

Let's add "sammiches" to the end:

```
>>> items.append('sammiches')
```

Nothing happened, so how do we know that worked? Let's check the length. It should be 1:

```
>>> len(items)
1
```

Hooray! That worked. In the spirit of testing, use the `assert` statement to verify that the length is 1. The fact that nothing happens is good. When an assertion fails, it triggers an exception that results in a lot of messages. Here, no news is good news:

```
>>> assert len(items) == 1
```

If you type `items<Enter>` in the REPL, Python will show you the contents:

```
>>> items
['sammiches']
```

Cool, we added one element.

3.3.2 Adding many elements to a list

Let's try to add "chips" and "ice cream" to the `items`:

```
>>> items.append('chips', 'ice cream')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: append() takes exactly one argument (2 given)
```

Here is one of those pesky exceptions, and these will cause your programs to *crash*, something we want to avoid at all costs. We see that `append()` takes exactly one argument, and we gave it two. If you look at the items, you'll see that nothing was added:

```
>>> items  
['sammiches']
```

OK, so maybe we were supposed to give it a list of items to add? Let's try that:

```
>>> items.append(['chips', 'ice cream'])
```

Well, that didn't cause an exception, so maybe it worked? We would expect there to be 3 items, so let's use an assertion to check that:

```
>>> assert len(items) == 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

Another exception because `len(items)` is not 3! What is the length?

```
>>> len(items)  
2
```

Only 2? Let's look at items:

```
>>> items  
['sammiches', ['chips', 'ice cream']]
```

Check that out! Lists can hold any type of data like strings and numbers and even other lists. We asked `items.append()` to add `['chips', 'ice cream']`, which is a list, and that's just what it did. Of course, it's not what we *wanted*.

```
['sammiches', ['chips', 'ice cream']]
```



Let's reset items so we can fix this:

```
>>> items = ['sammiches']
```

If you read further into the help, you will find the `list.extend()` method:

```
| extend(self, iterable, /)
|     Extend list by appending elements from the iterable.
```

Let's try that:

```
>>> items.extend('chips', 'ice cream')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: extend() takes exactly one argument (2 given)
```

Well that's frustrating! Now Python is telling us that `extend()` takes exactly one argument which, if you refer to the help, should be an iterable. A list is something you can iterate (travel over from beginning to end), so that will work:



```
>>> items.extend(['chips', 'ice cream'])
```

Nothing happened. No exception, so maybe that worked? Let's check the length. It *should* be 3:

```
>>> assert len(items) == 3
```

Yes! Let's look at the `items` we've added:

```
>>> items
['sammiches', 'chips', 'ice cream']
```

Great! This is sounding like a pretty delicious outing.

If you know everything that will go into the list, you can create it like so:

```
>>> items = ['sammiches', 'chips', 'ice cream']
```

The `list.append()` and `list.extend()` methods add new elements to the *end* of a given list. The `list.insert()` method allows you to place new items at any position by specifying the index. I can use the index 0 to put a new element at the beginning of `items`:

```
>>> items.insert(0, 'soda')
>>> items
['soda', 'sammiches', 'chips', 'ice cream']
```

In addition to `help(list)`, you can also find lots of great documentation here:

docs.python.org/3/tutorial/datastructures.html

I recommend you read over all the `list` functions so you get an idea of just how powerful this data structure is!

3.3.3 Indexing lists

So now we have a list of `items`. We know how to use `len()` to find how many `items` there are, and now we need to know how to get parts of the list to format. Indexing a list in Python looks exactly the same as indexing a `str`. (This actually makes me a bit uncomfortable, so I tend to imagine a `str` as a list of characters and then I feel somewhat better.)

0	1	2	3
['soda', 'sammiches', 'chips', 'ice cream']			
-4	-3	-2	-1

All indexing in Python is zero-offset, so the first element of `items` is at index `items[0]`:

```
>>> items[0]
'soda'
```

If the index is negative, Python starts counting backwards from the end of the list. The index `-1` is the last element of the list:

```
>>> items[-1]
'ice cream'
```

You should be very careful when using indexes to reference elements in a list. This is unsafe code:

```
>>> items[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

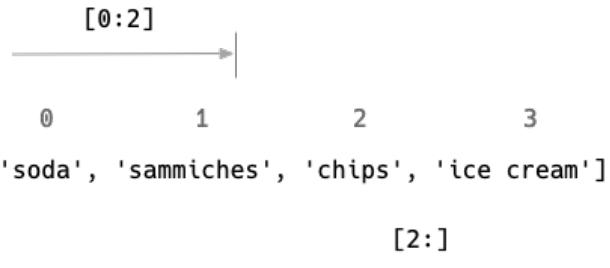
WARNING

Referencing an index that is not present will cause an exception.

We'll soon learn how to safely *iterate* or travel through a *list* so that we don't have to use indexes to get at elements.

3.3.4 Slicing lists

You can extract "slices" (sub-lists) of a list by using `list[start:stop]`. To get the first two items elements, you use `[0:2]`. Remember that the 2 is actually the index of the *third* element but *it's not inclusive*:



```
>>> items[0:2]
['soda', 'sammiches']
```

If you leave out `start`, it will be `0`, so this does the same thing:

```
>>> items[:2]
['soda', 'sammiches']
```

If you leave out `stop`, it will go to the end of the list:

```
>>> items[2:]
['chips', 'ice cream']
```

Oddly, it is completely *safe* for slices to use list indexes that don't exist. Here I can ask for all the elements from index `10` to the end even though there is nothing at index `10`. Instead of an exception, we get an empty list:

```
>>> items[10:]
[]
```

For our exercise, you're going to need to get the word "and" into the list if there are three or more elements. Could you use a list index to do that?

3.3.5 Finding elements in a list

Did we remember to pack the chips?! Often we want to know if some items is in a list. The `index` method will return the location of an element in a list:

```
>>> items.index('chips')
2
```

WARNING

`list.index` is unsafe code because it will cause an exception if the argument is not present in the list!

See what happens if we check for the fog machine:

```
>>> items.index('fog machine')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 'fog machine' is not in list
```

You should never use `index` unless you have first verified that an element is present. The `x in y` that we used in "Crow's Nest" to see if a letter was in the list of vowels can also be used for lists. We get back a `True` value if `x` is in the collection of `y`:

```
>>> 'chips' in items
True
```

I hope they're salt and vinegar chips.

The same returns `False` if it is not present:

```
>>> 'fog machine' in items
False
```

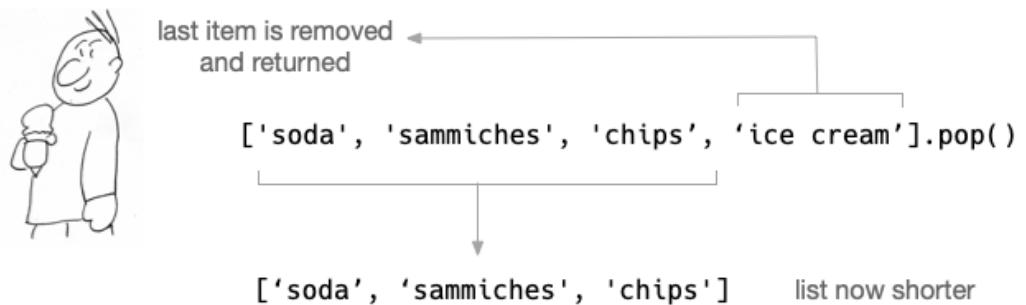


We're going to need to talk to the planning committee. What's a picnic without a fog machine?

3.3.6 **Removing elements from a list**

The `list.pop()` method will remove *and return* the element at the index. By default it will remove the *last* item (-1):

```
>>> items.pop()
'ice cream'
```

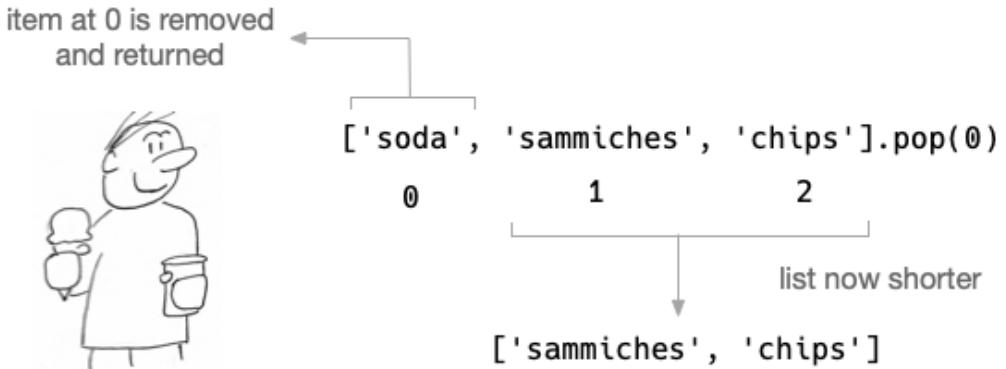


If we look at the list, we will see it's shorter by one:

```
>>> items
['soda',
 'sammiches',
 'chips']
```

```
>>> items.pop(0)
'soda'
```

You can use an item's index to remove an element at a particular location. For instance, we can use `0` to remove the first element:

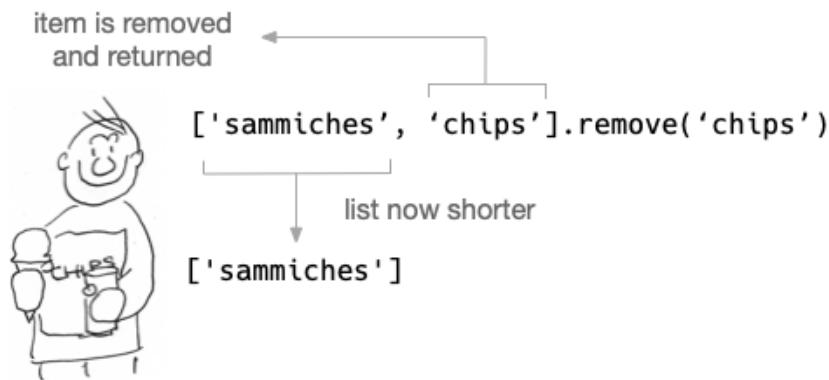


And now our `list` is shorter still:

```
>>> items
['sammiches', 'chips']
```

You can also use the `list.remove()` method to remove the first occurrence of a given item:

```
>>> items.remove('chips')
>>> items
['sammiches']
```



WARNING | The `list.remove()` will cause an exception if the element is not present.

If we try to `items.remove()` the chips again, we get an exception:

```
>>> items.remove('chips')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```



So don't use this code unless you've verified that a given element is in the list:

```
item = 'chips'
if item in items:
    items.remove(item)
```

3.3.7 Sorting and reversing a list

If the `--sorted` flag is present, we're going to need to sort the `items`. You might notice in the help documentation that two methods, `list.reverse()` and `list.sort()` stress that they work *IN PLACE*. That means that the list itself will be either reversed or sorted and *nothing will be returned*. So, given this list:

```
>>> items = ['soda', 'sammiches', 'chips', 'ice cream']
```

The `sort` method will return nothing:

```
>>> items.sort()
```

`None ← items.sort()`

 items are sorted,
 and nothing is returned

But if you inspect `items`, you will see they have been sorted alphabetically:

```
>>> items
['chips', 'ice cream', 'sammiches', 'soda']
```

Note that Python will sort a list of numbers *numerically*, so we've got that going for us, which is nice:

```
>>> sorted([4, 2, 10, 3, 1])
[1, 2, 3, 4, 10]
```

WARNING

Sorting a list that mixes strings and numbers will cause an exception!

```
>>> sorted([1, 'two', 3, 'four'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'str' and 'int'
```

As with `list.sort()`, we see nothing on the `list.reverse()` call:

```
>>> items.reverse()
```

But the `items` are now in the opposite order:

```
>>> items
['soda', 'sammiches', 'ice cream', 'chips']
```

The `list.sort()` and `list.reverse()` methods are easily confused with the the `sorted()` and `reversed()` functions. The `sorted()` function accepts a list as an argument and *returns* a new list:

```
>>> items = ['soda', 'sammiches', 'chips', 'ice cream']
>>> sorted(items)
['chips', 'ice cream', 'sammiches', 'soda']
```

It's crucial to note that the `sorted()` function *does not alter* the given list:

```
>>> items
['soda', 'sammiches', 'chips', 'ice cream']
```

The `list.sort()` method is a function that belongs to the `list`. It can take arguments that affect the way the sorting happens. Let's look at the `help(list.sort)`:

```
sort(self, /, *, key=None, reverse=False)
    Stable sort *IN PLACE*.
```

So we could also `sort()` the `items` in reverse like so:

```
>>> items.sort(reverse=True)
```

And now they look like this:

```
>>> items
['soda', 'sammiches', 'ice cream', 'chips']
```



The `reversed()` function works a bit differently:

```
>>> reversed(items)
<list_reverseiterator object at 0x10e012ef0>
```

I bet you were expecting to see a new `list` with the `items` in reverse? This is an example of a *lazy* function in Python. The process of reversing a `list` might take a while, so Python is showing that it has generated an "iterator object" that will provide the reversed list just as soon as we actually need the elements.

We can do that in the REPL by using the `list` function to evaluate the iterator:

```
>>> list(reversed(items))
['ice cream', 'chips', 'sammiches', 'soda']
```

As with the `sorted()` function, the original `items` itself remain unchanged:

```
>>> items
['soda', 'sammiches', 'chips', 'ice cream']
```

If you use the `list.sort()` method instead of the `sorted()` function, you might end up deleting your data. Imagine you wanted to set your `items` equal to the sorted list of `items` like so:

```
>>> items = items.sort()
```

What is in `items` now? If you print the `items` in the REPL, you won't see anything useful, so inspect the type:

```
>>> type(items)
<class 'NoneType'>
```

It's no longer a `list`. We set it equal to the result of called `items.sort()` method that

works on the `list` *in-place* and returns `None`. I would note that I tend to not use the `sort()`/`reverse()` methods because I don't generally like to mutate my data. I would tend to do something like this:

```
>>> items = ['soda', 'sammiches', 'chips', 'ice cream']
>>> sorted_items = sorted(items)
>>> sorted_items
['chips', 'ice cream', 'sammiches', 'soda']
```

Now I have explicitly named a `sorted_items` list, and the original `items` has not been altered.

If the `--sorted` flag is given to your program, you will need to sort your items in order to pass the test. Will you use `list.sort()` or the `sorted()` function?

3.3.8 Lists are mutable

As we've seen, we can change a `list` quite easily. The `list.sort()` and `list.reverse()` methods change the whole list, but you can also change any single element by referencing it by index. Maybe we make our picnic slightly healthier by changing out the chips for apples:

```
>>> items
['soda', 'sammiches', 'chips', 'ice cream']
>>> if 'chips' in items:          ①
...     idx = items.index('chips') ②
...     items[idx] = 'apples'       ③
... 
```

- ① See if the string 'chips' is in the list of `items`.
- ② Assign the index of 'chips' to the variable `idx`.
- ③ Use the index `idx` to change the element to 'apples'.

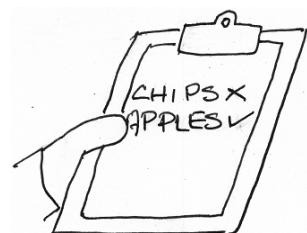
Let's look at `items` to verify:

```
>>> items
['soda', 'sammiches', 'apples', 'ice cream']
```

We can also write a couple of tests:

```
>>> assert 'chips' not in items ①
>>> assert 'apples' in items   ②
```

- ① Make sure "chips" are no longer on the menu.
- ② Check that we now have some "apples."

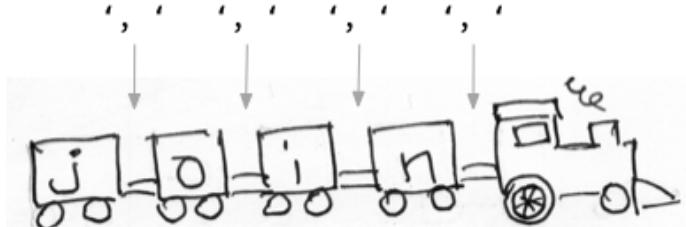


3.3.9 Joining a list

In our exercise, you'll need to print a string based on the number of elements in the given list. The string will intersperse some other string like ', ' in between all the

elements of the list. Oddly, this is the syntax to join a list on the string made of the comma and a space:

```
>>> ', '.join(items)
'soda, sammiches, chips, ice cream'
```



Here we use the `str.join()` method and pass the `list` as an argument. It always feels backwards to me, but that's the way it goes. The result of `str.join()` is a *new string*:

```
>>> type(', '.join(items))
<class 'str'>
```

The original `list` remains unchanged:

```
>>> items
['soda', 'sammiches', 'chips', 'apples']
```

There is quite a bit more that we can do with Python's `list`, but that should be enough for you to solve this problem.

3.4 Conditional branching with `if/elif/else`

You need to use the conditional branching based on the number of items to correctly format the output. In the "Crow's Nest" exercise, there were two conditions (a "binary" choice) — either a vowel or not — so we used `if/else` statements. Here we have three options to consider, so you will have to use `elif` (`else-if`). For instance, we want to classify someone by their age by three options:

1. If their age is greater than 0, it is valid.
2. If their age is less than 18, they are a minor.
3. Otherwise they are 18 years or older, which means they can vote:

Here is how I could write that code:

```
>>> age = 15
>>> if age < 0:
...     print('You are impossible.')
... elif age < 18:
...     print('You are a minor.')
... else:
...     print('You can vote.')
...
You are a minor.
```

See if you can use that to figure out how to write the three options for `picnic.py`. That

is, first write the branch that handles one item. Then write the branch that handles two items. Then write the last branch for three or more items. Run the tests *after every change to your program*.

Now go write the program yourself before you continue to look at my solution.

Hints:

- Go into your `03_picnic` directory and run `new.py picnic.py` to start your program. Then run `make test` (or `pytest -xv test.py`) and you should pass the first two tests.
- Next work on getting your `--help` usage looking like the above. It's very important to define your arguments correctly. For the `items` argument, look at `nargs` in `argparse` as discussed in chapter 1's "One or more of the same positional arguments" section.
- If you use `new.py` to start your program, be sure to leave the "boolean flag" and modify it for your `sorted` flag.
- Solve the tests in order! First handle one item, then handle two items, then handle three. Then handle the sorted items.

You'll get the best benefit from this book if you try writing the program and passing the tests before reading the solution!

3.5 Solution

```
#!/usr/bin/env python3
"""Picnic game"""

import argparse

# -----
def get_args(): ①
    """Get command-line arguments"""

    parser = argparse.ArgumentParser(
        description='Picnic game',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('item', ②
                        metavar='str',
                        nargs='+',
                        help='Item(s) to bring')

    parser.add_argument('-s', ③
                        '--sorted',
                        action='store_true',
                        help='Sort the items')

    return parser.parse_args() ④

# -----
def main(): ⑤
    """Make a jazz noise here"""

```

```

args = get_args() ⑥
items = args.item ⑦
num = len(items) ⑧

if args.sorted: ⑨
    items.sort() ⑩

bringing = '' ⑪
if num == 1: ⑫
    bringing = items[0] ⑬
elif num == 2: ⑭
    bringing = ' and '.join(items) ⑮
else: ⑯
    items[-1] = ' and ' + items[-1] ⑰
    bringing = ', '.join(items) ⑱

print('You are bringing {}'.format(bringing)) ⑲

# -----
if __name__ == '__main__': ⑳
    main() ㉑

```

- ① The `get_args()` function is placed first so that we can easily see what the program accepts when we read it. Note that the function order here is not important to Python, only to us, the reader.
- ② The `item` argument uses the `nargs='+'` so that it will accept *one or more* positional arguments which will be strings.
- ③ The dashes in the short (`-s`) and long (`--sort`) names make this an *option*. There is no value associated with this argument. It's either present (in which case it will be `True`) or absent (or `False`).
- ④ Process the command-line arguments and return them to the caller.
- ⑤ The `main()` function is where the program will start.
- ⑥ Call the `get_args()` function and put the returned value into the variable `args`. If there is a problem parsing the arguments, the program will fail before the values are returned.
- ⑦ Copy the `item` list from the `args` into the new variable `items`.
- ⑧ Use the length function `len()` to get the number of `items` in the list. There can never be zero arguments because we defined the argument using `nargs='+'` which always requires at least one value.
- ⑨ The `args.sorted` value will either be `True` or `False` because it was defined as a "flag."
- ⑩ If we are supposed to sort the items, call the `items.sort()` method to sort them *in-place*.
- ⑪ Initialize the variable `bringing` with an empty string. We'll put the items we're bringing into this.
- ⑫ If the number of items is 1...
- ⑬ Then we will assign the one item to `bringing`.
- ⑭ If the number of items is 2...
- ⑮ Put the string ' `and` ' in between the `items`.
- ⑯ Otherwise...
- ⑰ Alter the last element in `items` to append the string ' `and` ' before whatever is already there.
- ⑱ Join the `items` on the string `', '`.
- ⑲ Print the output string, using the `str.format()` method to interpolate the `bringing` variable.

- ㉐ When Python runs the program, it will read all the lines to this point but will not run anything. Here we look to see if we are in the "main" namespace.
- ㉑ If we are, call the `main()` function to make the program begin.

3.6 Discussion

How did it go? Did it take you long to write your version? How different was it from mine? Let's talk about my solution. It's perfectly fine if yours is really different from mine, just as long as you pass the tests!

3.6.1 Defining the arguments

This program can accept a variable number of arguments which are all the same thing (strings). In my `get_args()`, I define an `item` like so:

```
parser.add_argument('item',
                   metavar='str',
                   nargs='+',
                   help='Item(s) to bring') ①
②
③
④
```

- ① A single, required, positional (because no dashes in name) argument called `item`.
- ② An indicator to the user in the usage that this should be a string.
- ③ The number of arguments where '+' means *one or more*.
- ④ A longer help description that appears for the -h or --help options.

This program also accepts -s and --sorted arguments. Remember that the leading dashes makes them optional. They are "flags," which typically means that they are `True` if they are present and `False` if absent.

```
parser.add_argument('-s',
                   '--sorted',
                   action='store_true',
                   help='Sort the items') ①
②
③
④
```

- ① The short flag name.
- ② The long flag name.
- ③ If the flag is present, store a `True` value. The default value will be `False`.
- ④ The longer help description.

3.6.2 Assigning and sorting the items

To get the arguments, in `main()` I call `get_args()` and assign them to the `args` variable. Then I create the `items` variable to hold the `args.item` value(s):

```
args = get_args()
items = args.item
```

If `args.sorted` is `True`, then I need to sort my `items`. I chose the *in-place* sort method here:

```
if args.sorted:
    items.sort()
```

Now I have the items, sorted if needed, and I need to format them for the output.

3.6.3 **Formatting the items**

I suggested you solve the tests in order. There are 4 conditions we need to solve:

1. Zero items
2. One item
3. Two items
4. Three or more items

The first test is actually handled by argparse — if the user fails to provide any arguments, they get a usage:

```
$ ./picnic.py
usage: picnic.py [-h] [-s] str [str ...]
picnic.py: error: the following arguments are required: str
```

Since argparse handles the case of no arguments, we have to handle the other three conditions. Here's one way to do that:

```
bringing = ''          ①
if num == 1:           ②
    bringing = items[0] ③
elif num == 2:          ④
    bringing = ' and '.join(items) ⑤
else:                  ⑥
    items[-1] = 'and ' + items[-1] ⑦
    bringing = ', '.join(items) ⑧
```

- ① Initialize a variable for what we are bringing.
- ② Check if the number of items is one.
- ③ If there is one item, then bringing is the one item.
- ④ Check if the number of items is two.
- ⑤ If two items, join the items on the string ' and '.
- ⑥ Otherwise...
- ⑦ Insert the string 'and ' before the last item
- ⑧ Join all the items on the string ', '.

Can you come up with any other ways?

3.6.4 **Printing the items**

Finally to print the output, I can use a format string where the {} indicates a placeholder for some value like so:

```
>>> print('You are bringing {}'.format(bringing))
You are bringing salad, soda, and cupcakes.
```

Or, if you prefer, you can use an f''-string:

```
>>> print(f'You are bringing {bringing}.')
You are bringing salad, soda, and cupcakes.
```

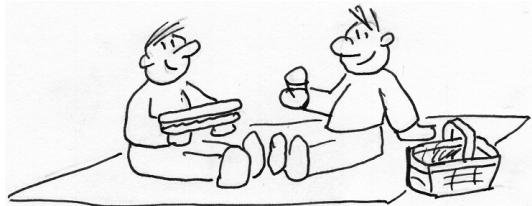
They both get the job done, so whichever you prefer.

3.7 Summary

- Python lists are ordered sequences of other Python data types such as strings and numbers.
- There are methods like `list.append()` and `list.extend()` to add elements to a list. Use `list.pop()` and `list.remove()` to remove them.
- You can use `x in y` to ask if element `x` is in the list `y`. You could also use `list.index` to find the index of an element, but this will cause an exception if the element is not present.
- Lists can be sorted and reversed, and elements within lists can be modified. Lists are useful when the order of the elements is important.
- Strings and lists share many features such as using `len()` to find their lengths, using zero-based indexing where `0` is the first element and `-1` is the last, and using slices to extract smaller pieces from the whole.
- The `str.join()` method can be used to make a new `str` from a `list`.
- `if/elif/else` can be used to branch code depending on conditions.

3.8 Going Further

- Add an option so the user can choose not to print with the Oxford comma (even though that is a morally indefensible option).
- Add an option to separate items with some character passed in by the user (like a semicolon if the list of items needed to contain commas).

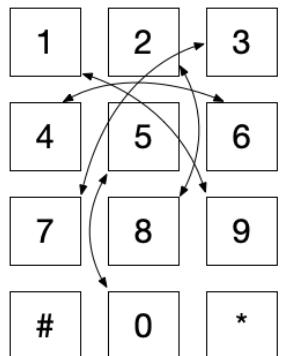


Jump the Five: Working with dictionaries



"When I get up, nothing gets me down." - D. L. Roth

In an episode of the television show *The Wire*, drug dealers encode telephone numbers that they text in order to obscure them from the police who they assume are intercepting their messages. They use an algorithm we'll call "Jump The Five" where a number is changed to the one that is opposite on a US telephone pad if you jump over the 5. In this exercise, we'll first learn how to encrypt messages using this algorithm, and then we'll see how we can use it to decrypt messages, you feel me?



If we start with "1" and jump across the 5, we get to "9," then "6" jumps the 5 to become "4," and so forth. The numbers "5" and "0" will swap with each other. In this exercise, we're going to write a Python program called `jump.py` that will take in some text as a positional argument. Each number in the text will be encoded using this algorithm. All non-number will pass through unchanged, for example:

```
$ ./jump.py 867-5309
243-0751
$ ./jump.py 'Call 1-800-329-8044 today!'
Call 9-255-781-2566 today!
```

We will need some way to inspect each character in the input text and identify the numbers. We will learn how to use a `for` loop for this, and then we'll look at how it

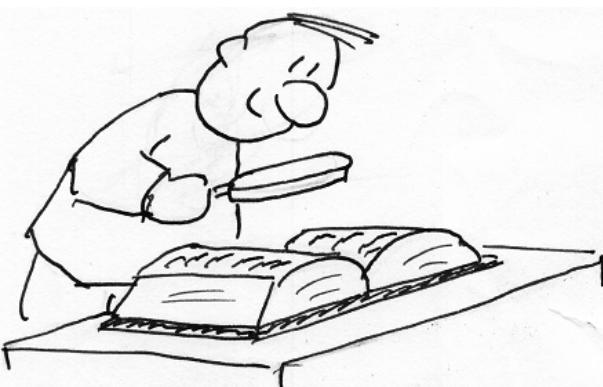
can be rewritten as a "list comprehension." Then we will need some way to associate a number like "1" with the number "9," and so on for all the numbers. We'll learn about a data structure in Python called a "dictionary" type that allows us to do exactly that.

In this chapter, you will learn to:

- Create a dictionary.
- Use a `for` loop to process text character-by-character.
- Check if items exist in a dictionary.
- Retrieve values from a dictionary.
- Print a new string with the numbers substituted for their encoded values.

Before we get started with the coding, we need to learn about Python's dictionaries.

4.1 Dictionaries



A Python `dict` allows us to relate some *thing* (a "key") to some other *thing* (a "value"). An actual dictionary does this. If we look up a word like "quirky" in a dictionary (www.merriam-webster.com/dictionary/quirky), we can find a definition. We can think of the word itself as the "key" and the definition as the "value."

quirky ➔ unusual, esp. in an interesting or appealing way

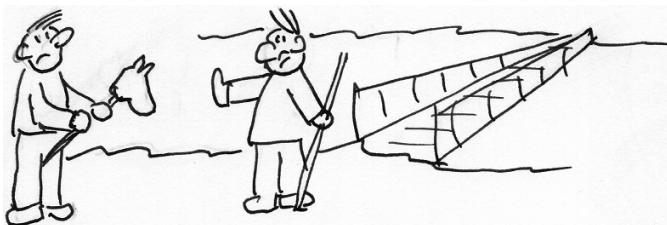
Dictionaries actually provide quite a bit more information such as pronunciation, part of speech, derived words, history, synonyms, alternate spellings, etymology, first known use, etc. (I really love dictionaries.) Each of those attributes has a value, so we could also think of the dictionary entry for a word itself as another "dictionary":

definition	➔ unusual, esp. in an interesting or appealing way
pronunciation	➔ 'kwər-kē
part of speech	➔ adjective

Let's see how we can use Python's dictionaries to go beyond word definitions.

4.1.1 Creating a dictionary

In the film *Monty Python and the Holy Grail*, King Arthur and his knights must cross The Bridge of Death. Anyone who wishes to cross must correctly answer three questions from the Keeper. Those who fail are cast into the Gorge of Eternal Peril.



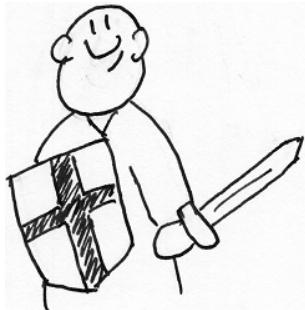
Let us ride to CAMEL...no, sorry, let us create and use a dict to keep track of the questions and answers as key/value pairs. Once again, I want you to fire up your python3/ipython REPL or Jupyter Notebook and type these out for yourself!

Lancelot goes first. We can use the `dict()` function to create an empty dictionary for his answers.

```
>>> answers = dict()
```

Or we can use empty curly brackets (both methods are equivalent):

```
>>> answers = {}
```



The Keeper's first question: "What is your name?" Lancelot answers "My name is Sir Lancelot of Camelot." We can add the key "name" to our `answers` by using square brackets ([]) — not curly braces! — and the literal string 'name':

```
>>> answers['name'] = 'Sir Lancelot'
```

If you type `answers<Enter>` in the REPL, Python will show you a structure in curly braces to indicate this is a `dict`:

```
>>> answers
{'name': 'Sir Lancelot'}
```

You can verify with the `type()` function:

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

{'name': 'Sir Lancelot'}

curly brackets mean "dictionary"

Next the Keeper asks, "What is your quest?" to which Lancelot answers "To seek the Holy Grail." Let's add "quest" to the answers:

```
>>> answers['quest'] = 'To seek the Holy Grail'
```

There's no return value to let us know something happened, so type `answers` to inspect the variable again to ensure our new key/value was added:

```
>>> answers
{'name': 'Sir Lancelot', 'quest': 'To seek the Holy
Grail'}
```



Finally the Keeper asks "What is your favorite color?," and Lancelot answers "blue."³

```
>>> answers['favorite_color'] = 'blue'
>>> answers
{'name': 'Sir Lancelot', 'quest': 'To seek the Holy Grail', 'favorite_color': 'blue'}
```

If you knew all the answers beforehand, you could create `answers` using the `dict()` function with this syntax where you do *not* have to quote the keys and the keys are separate from the values with equal signs:

```
>>> answers = dict(name='Sir Lancelot', quest='To seek the Holy Grail',
favorite_color='blue')
```

Or this syntax using curly braces {} where the keys must be quoted and are followed by a colon (:):

```
>>> answers = {'name': 'Sir Lancelot', 'quest': 'To seek the Holy Grail',
'favorite_color': 'blue'}
```

³ Note that I'm using "favorite_color" (with an underscore) as the key, but I could use "favorite color" (with a space) or "FavoriteColor" or "Favorite color," but each one of those is a separate and distinct string/key. I prefer to use the same naming conventions from PEP8 for dictionary keys as for variable and functions names which suggests lowercase names with words separated by underscores.

It might be helpful to think of the dictionary `answers` as a box that inside holds the key/value pairs that describe Lancelot's answers just the way the "quirky" dictionary holds all the information about that word.

answers

<code>name</code>	⇒ Sir Lancelot
<code>quest</code>	⇒ To seek the Holy Grail
<code>favorite color</code>	⇒ blue

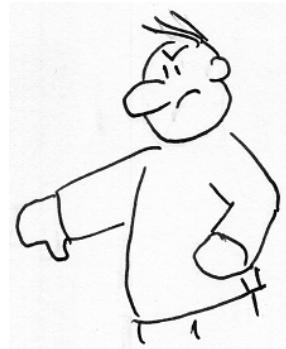
4.1.2 Accessing dictionary values

To retrieve the values, you use the key name inside square brackets ([]). For instance, I can get the name like so:

```
>>> answers['name']
'Sir Lancelot'
```

Let's request his "age":

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



WARNING

You will cause an exception if you ask for a dictionary key that doesn't exist!

Just as with lists, you can use the `x in y` to first see if a key exists in the dict:

```
>>> 'quest' in answers
True
>>> 'age' in answers
False
```



The `dict.get()` method is a *safe* way to ask for a value:

```
>>> answers.get('quest')
'To seek the Holy Grail'
```

When the requested key does not exist in the `dict`, it will return the special value `None`:

```
>>> answers.get('age')
```

That doesn't print anything because the REPL won't print a `None`, but we can check the `type()`:

```
>>> type(answers.get('age'))
<class 'NoneType'>
```



There is an optional second argument you can pass to `dict.get()` which is the value to return *if the key does not exist*:

```
>>> answers.get('age', 'NA')
'NA'
```

4.1.3 Other dictionary methods

If you want to know how "big" a dictionary is, the `len()` (length) function on a `dict` will tell you how many key/value pairs are present:

```
>>> len(answers)
3
```

The `dict.keys()` method will give you just the keys:

```
>>> answers.keys()
dict_keys(['name', 'quest', 'favorite_color'])
```

And `dict.values()` will give you the values:

```
>>> answers.values()
dict_values(['Sir Lancelot', 'To seek the Holy Grail', 'blue'])
```

Often we want both together, so you might see code like this:

```
>>> for key in answers.keys():
...     print(key, answers[key])
...
name Sir Lancelot
quest To seek the Holy Grail
favorite_color blue
```

An easier way to write this would be to use the `dict.items()` method which will return the contents of the dictionary as a new list containing each key/value pair:

```
>>> answers.items()
dict_items([('name', 'Sir Lancelot'), ('quest', 'To seek the Holy Grail'),
('favorite_color', 'blue')])
```

The above for loop could also be written using the `dict.items()` method:

```
>>> for key, value in answers.items(): ①
...     print(f'{key:15} {value}')      ②
...
name           Sir Lancelot
quest          To seek the Holy Grail
favorite_color blue
```

- ① For each key/value pair, unpack them into the variables `key` and `value`. Note that you don't have to call them `key` and `value`. You could use `k` and `v` or `question` and `answer`.
- ② Print the key in a left-justified field 15 characters wide. The value is printed normally.

for key, value in [('name', 'Sir Lancelot'), ...]:

In the REPL you can execute `help(dict)` to see all the methods available to you like `pop` to remove a key/value or `update` to merge with another dict.

Each key in the dict is unique. That means if you set a value for a given key twice:

```
>>> answers = {}
>>> answers['favorite_color'] = 'blue'
>>> answers
{'favorite_color': 'blue'}
```

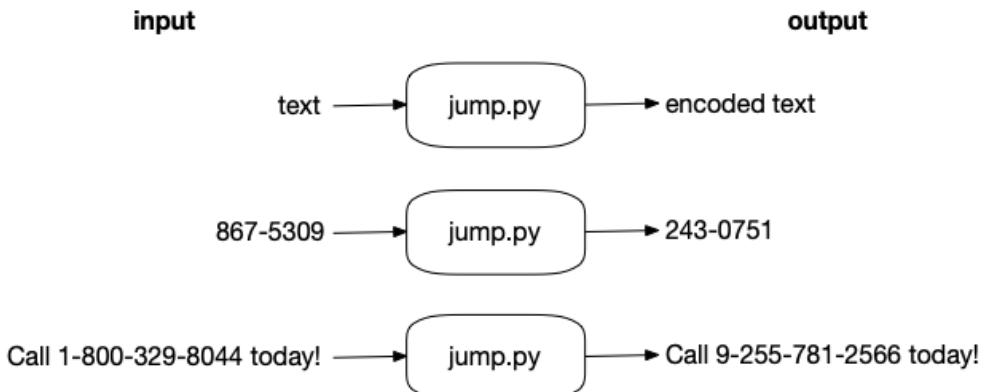
You will not have two entries but one entry with the *second* value:

```
>>> answers['favorite_color'] = 'red'
>>> answers
{'favorite_color': 'red'}
```

Keys don't have to be strings — you can also use numbers like `int` and `float`. Whatever value you use must be immutable. For instance, a list could not be a key because it is mutable.

4.2 Writing jump.py

Now let's get started with writing our program. Here is a diagram of the inputs and outputs. Note that your program will only affect the numbers in the text. Anything that is *not* a number is unchanged:



When run with no arguments, -h, or --help, your program should print a usage:

```
$ ./jump.py -h
usage: jump.py [-h] str

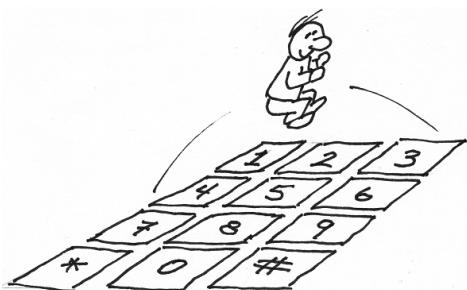
Jump the Five

positional arguments:
  str      Input text

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

Note that we will be processing *text* representations of the "numbers," so the string '1' will be converted to the string '9', not the actual integer value 1 to the integer value 9. Keep that in mind as you figure out a way to represent the following substitution table:

```
1 => 9
2 => 8
3 => 7
4 => 6
5 => 0
6 => 4
7 => 3
8 => 2
9 => 1
0 => 5
```



How would you represent this using a dict? Try creating a dict called `jumper` in the REPL and then using a test. Remember that `assert` will return nothing if the statement

is True:

```
>>> assert jumper['1'] == '9'
>>> assert jumper['5'] == '0'
```

Next, you will need a way to visit each character in the given text. I suggest you use a `for` loop like so:

```
>>> for char in 'ABC123':
...     print(char)
...
A
B
C
1
2
3
```

Now, rather printing the `char`, print the value of `char` in the `jumper` table or print the `char` itself. Look at the `dict.get()` method! Also, if you read `help(print)`, you'll see there is an `end` option to change the newline that gets stuck onto the end to something else.

Hints:

- The numbers can occur anywhere in the text, so I recommend you process the input character-by-character with a `for` loop.
- Given any one character, how can you look it up in your table?
- If the character is in your table, how can you get the value (the translation)?
- If how can you `print()` the translation or the value without printing a newline? Look at `help(print)` in the REPL to read about the options to `print()`.
- If you read `help(str)` on Python's `str` class, you'll see that there is a `str.replace()` method. Could you use that?

Now spend the time to write the program on your own before you look at the solutions! Use the tests to guide you.

4.3 Solution

```
#!/usr/bin/env python3
"""
Jump the Five"""

import argparse

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

    parser = argparse.ArgumentParser(
        description='Jump the Five',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('text', metavar='str', help='Input text') ②
```

①

```

    return parser.parse_args()

# -----
def main():                                     ③
    """Make a jazz noise here"""

    args = get_args()                         ④
    jumper = {'1': '9', '2': '8', '3': '7', '4': '6', '5': '0',
              '6': '4', '7': '3', '8': '2', '9': '1', '0': '5'}      ⑤

    for char in args.text:                    ⑥
        print(jumper.get(char, char), end='')   ⑦
    print()                                    ⑧

# -----
if __name__ == '__main__':                      ⑨
    main()

```

- ① Define the `get_args()` function first so it's easy to see when we read the program.
- ② We define one positional argument called `text`.
- ③ Define a `main()` function where the program starts.
- ④ Get the command-line args.
- ⑤ Create a dict for the lookup table.
- ⑥ Process each character in the text.
- ⑦ Print either the value of the char in the jumper table or the char if it's not present, making sure *not* to print a newline by adding `end=' '`.
- ⑧ Print a newline.
- ⑨ Call the `main()` function if the program is in the "main" namespace.

4.4 Discussion

4.4.1 Defining the arguments

If you look at the solution, you'll see that the `get_args()` function is defined first. Our program needs to define one positional argument. Since it's some "text" I expect, I call the argument '`text`' and then assign that to a variable called `text`.

```
parser.add_argument('text', metavar='str', help='Input text')
```

While all that seems rather obvious, I think it's very important to name things *what they are*. That is, please don't leave the name of the argument as '`positional`' — that does not describe what it *is*. It may seem like overkill to use `argparse` for such a simple program, but it handles the validation of the correct *number* and *type* of arguments as well as the generation of help documentation, so it's well worth the effort.

4.4.2 Using a dict for encoding

I suggested you could represent the substitution table as a dict where each

number key has its substitute as the value in the dict. For instance, I know that if I jump from 1 over the 5 I should land on 9:

```
>>> jumper = {'1': '9', '2': '8', '3': '7', '4': '6', '5': '0',
...           '6': '4', '7': '3', '8': '2', '9': '1', '0': '5'}
>>> jumper['1']
'9'
```

Since there are only 10 numbers to encode, this is probably the easiest way to write this. Note that the numbers are written with quotes around them, so they are actually of the type `str` and not `int` (integers). I do this because I will be reading characters from a `str`. If we stored them as actual numbers, I would have to coerce the `str` types using the `int` function:

```
>>> type('4')
<class 'str'>
>>> type(4)
<class 'int'>
>>> type(int('4'))
<class 'int'>
```

4.4.3 Method 1: Using a for loop to print() each character

As suggested in the introduction, I can process each character of the text using a `for` loop. To start, I might first see if each character of the text is in the `jumper` table using the `x in y` construct.

```
>>> text = 'ABC123'
>>> for char in text:
...     print(char, char in jumper)
...
A False
B False
C False
1 True
2 True
3 True
```

NOTE

When `print()` is given more than one argument, it will put a space in between each of bit of text. You can change that with the `sep` argument. Read `help(print)` to learn more.

Now let's try to translate the numbers. I could use an `if` expression where I print the value from the `jumper` table if `char` is present, otherwise print the `char`:

```
>>> for char in text:
...     print(char, jumper[char] if char in jumper else char)
...
A A
B B
C C
1 9
2 8
3 7
```

It's a bit laborious to check for every character. The `dict.get()` method allows us to

safely ask for a value if it is present. For instance, the letter "A" is not in `jumper`. If we try to retrieve that value, we'll get an exception:

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

But if we use `jumper.get()`, there is no exception:

```
>>> jumper.get('A')
```

When a key doesn't exist in the dictionary, the special `None` value is returned:

```
>>> for char in text:
...     print(char, jumper.get(char))
...
A None
B None
C None
1 9
2 8
3 7
```

We can provide a second, optional argument to `get` that is the default value to return when the key does not exist. In our case, if a character does not exist in the `jumper`, we want to print the character itself. If we had "A," then we'd want to print "A":

```
>>> jumper.get('A', 'A')
'A'
```

But if we have, "5" then we want to print "0":

```
>>> jumper.get('5', '5')
'0'
```

So we can use that to process all the characters:

```
>>> for char in text:
...     print(jumper.get(char, char))
...
A
B
C
9
8
7
```

I don't want that newline printing after every character, so I can use `end=''` to tell Python to put the empty string at the end instead of a newline. When I run this in the REPL, the output is going to look funny because I have to hit `<Enter>` after the `for` loop for it to run, then I'll be left with ABC987 with no newline and then the `>>>` prompt:

```
>>> for char in text:
...     print(jumper.get(char, char), end='')
...
```

ABC987>>>

And so in your code you have to add another `print()`. Mostly I wanted to point out a couple things you maybe didn't know about `print()`. It's useful that you can change what is added at the end, and that you can `print()` with no arguments to print a newline. There are several other really cool things `print()` can do, so I'd encourage you to read `help(print)` and try them out!

4.4.4 Method 2: Using a for loop to build a new string

There are several other ways we could solve this. I don't like all the `print()` statements in the first solution, so here's another take:

```
def main():
    args = get_args()
    jumper = {'1': '9', '2': '8', '3': '7', '4': '6', '5': '0',
              '6': '4', '7': '3', '8': '2', '9': '1', '0': '5'}
    new_text = ''  
①
    for char in args.text:  
②
        new_text += jumper.get(char, char)  
③
    print(new_text)  
④
```

- ① In this alternate solution, you create an empty `new_text` variable.
- ② Same for loop...
- ③ Append either the encoded number or the original char to the `new_text`
- ④ Print the `new_text`.

While it's fun to explore all the things we can do with `print()`, that code is a bit ugly. I think it's cleaner to create a `new_text` variable and call `print()` once with that. To do this, we start off by setting a `new_text` equal to the empty string:

```
>>> new_text = ''
```

And we use our same for loop to process each character in the text. Each time through the loop, we use `+=` to append the right-hand side of the equation to the left-hand side. The `+=` adds the value on the right to the variable on the left:

```
>>> new_text += 'a'  
>>> assert new_text == 'a'  
>>> new_text += 'b'  
>>> assert new_text == 'ab'
```

On the right, we're using the `jumper.get()` method. Each character will be appended to the `new_text`:

```
>>> new_text = ''  
>>> for char in text:  
...     new_text += jumper.get(char, char)  
...
```

`new_text += jumper.get(char, char)`

the result of `jumper.get`
is appended to `new_text`

Now we can call `print()` one time with our new value:

```
>>> print(new_text)
ABC987
```

4.4.5 Method 3: Using a for loop to build a new list

This method is the same as above, but, rather than `new_text` being a `str`, it's a `list`:

```
def main():
    args = get_args()
    jumper = {'1': '9', '2': '8', '3': '7', '4': '6', '5': '0',
              '6': '4', '7': '3', '8': '2', '9': '1', '0': '5'}
    new_text = []                                ①
    for char in args.text:                      ②
        new_text.append(jumper.get(char, char))  ③
    print(''.join(new_text))                    ④
```

- ① Initialize `new_text` as an empty list.
- ② Iterate through each character of the text.
- ③ Append the results of the `jumper.get` call to the `new_text` variable.
- ④ Join the `new_text` on the empty string to create a new `str` to print.

As we go through the book, I'll keep reminding you how Python treats strings and lists similarly. Here I'm using `new_text` exactly the same as above, starting off with it empty and then making it longer for each character. We could actually use the exact same `+=` syntax instead of the `list.append()` method:

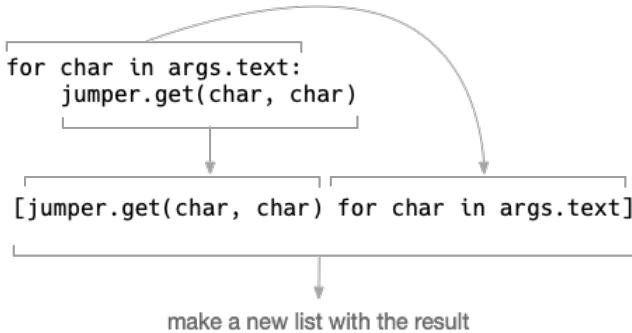
```
for char in args.text:
    new_text += jumper.get(char, char)
```

After the `for` loop is done, we have all the new characters that need to be put back together into a new string to print.

4.4.6 Method 4: Turning a for loop into a list comprehension

A shorter solution uses a "list comprehension" which is basically a one-line `for` loop inside square brackets (`[]`) which results in a new list:

```
def main():
    args = get_args()
    jumper = {'1': '9', '2': '8', '3': '7', '4': '6', '5': '0',
              '6': '4', '7': '3', '8': '2', '9': '1', '0': '5'}
    print(''.join([jumper.get(char, char) for char in args.text]))
```



A list comprehension is read backwards from a `for` loop, but it's all there. It's one line of code instead four!

```
>>> text = '867-5309'
>>> [jumper.get(char, char) for char in text]
['2', '4', '3', '-', '0', '7', '5', '1']
```

You can use the `str.join()` on the empty string to turn that list into a new string you can `print()`:

```
>>> print(''.join([jumper.get(char, char) for char in text]))
243-0751
```

4.4.7 Method 5: Using the `str.translate()` function

This last method uses a really powerful method from the `str` class to change all the characters in one step:

```
def main():
    args = get_args()
    jumper = {'1': '9', '2': '8', '3': '7', '4': '6', '5': '0',
              '6': '4', '7': '3', '8': '2', '9': '1', '0': '5'}
    print(args.text.translate(str.maketrans(jumper)))
```

The argument to `str.translate()` is a translation table that defines how each character should be translated. That's exactly what our `jumper` does!

```
>>> text = 'Jenny = 867-5309'
>>> text.translate(str.maketrans(jumper))
'Jenny = 243-0751'
```

I'll explain this in much greater detail in the "Apples and Bananas" exercise.

4.4.8 (Not) using `str.replace()`

Note that you could *not* use `str.replace()` to change each number 0-9. Watch how we start off with this string:

```
>>> text = '1234567890'
```

When you change "1" to "9," now you have two 9s:

```
>>> text = text.replace('1', '9')
>>> text
'9234567890'
```

Which means when you try to change all the 9s to 1s, you end up with two 1s:

```
>>> text = text.replace('9', '1')
>>> text
'1234567810'
```

You might try to write it like so:

```
>>> text = '1234567890'
>>> for n in jumper.keys():
...     text = text.replace(n, jumper[n])
...
>>> text
'1234543215'
```

But the correctly encoded string is "9876043215", which is exactly why `str.translate()` exists!

4.5 Summary

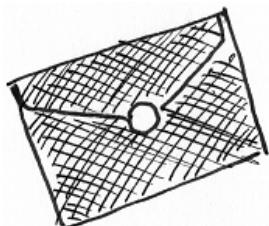
- You create a new dictionary using the `dict()` function or with empty curly brackets `({})`.
- Dictionary values are retrieved using their keys inside square brackets or by using the `dict.get()` method.
- For a `dict` called `x`, you can use `'key' in x` to determine if a key exists.
- You can use a `for` loop to iterate the characters of a `str` just like you can iterate through the elements of a `list`. You can think of strings as lists of characters.
- The `print()` function takes optional keyword arguments like `end=' '` which we can use to print a value to the screen without a newline.

4.6 Going Further

- Try creating a similar program that encodes the numbers with strings, e.g., "5" becomes "five", "7" becomes "seven."
- What happens if you feed the output of the program back into itself. For example, if you run `./jump.py 12345`, you should get `98760`. If you run `./jump.py 98760`, do you recover the original numbers? This is called "round-tripping," and it's a common operation with algorithms that encode and decode text.

5

Howler: Working with files and *STDOUT*



In Harry Potter, a "Howler" is a nasty-gram that arrives by owl at Hogwarts. It will tear itself open, shout a blistering message at the recipient, and then combust. In this exercise, we're going to write a program that will transform text into a rather mild-mannered version of a Howler by MAKING ALL THE LETTERS UPPERCASE. The text that we'll process will be given as a single, positional argument.

For instance, if our program is given the input, "How dare you steal that car!", it should scream back "HOW DARE YOU STEAL THAT CAR!" Remember spaces on the command line delimit arguments, so multiple words need to be enclosed in quotes to be considered one argument:

```
$ ./howler.py 'How dare you steal that car!'
HOW DARE YOU STEAL THAT CAR!
```

The argument to the program may also name a file, in which case we need to read the file for the input:

```
$ ./howler.py ../inputs/fox.txt
THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.
```

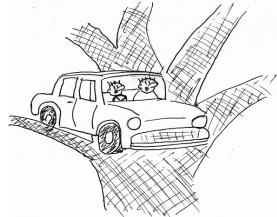


Our program will also accept an `-o` or `--outfile` option that names an output file into which the output text should be written. In that case, *nothing* will be printed on the command line:

```
$ ./howler.py -o out.txt 'How dare you steal  
that car!'
```

And there should now be a file called `out.txt` that has the output:

```
$ cat out.txt  
HOW DARE YOU STEAL THAT CAR!
```



In this exercise, you will learn to:

- Accept text input from the command line or from a file.
- Change strings to uppercase.
- Print output either to the command line or to a file that needs to be created.
- Make plain text behave like a file handle.

5.1 **Reading files**

This will be our first exercise that will involve reading files. The argument to the program will be some text. That text might name an input file in which case you will open and read the file. If it's not the name of a file, then you'll use the text itself.

The built-in `os` (operating system) module has a method for detecting if a string is the name of a file. To use it, you must import the `os` module. For instance, there's probably not a file called "blargh" on your system:

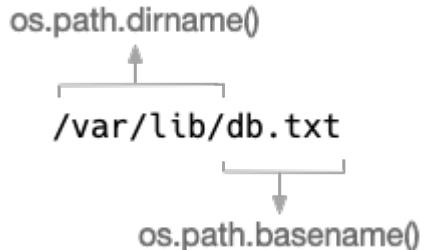
```
>>> import os  
>>> os.path.isfile('blargh')  
False
```



The `os` module contains loads of useful submodules and functions. Consult the documentation at docs.python.org/3/library/os.html or use `help(os)` in the REPL. The `os.path` module has handy functions like `basename()` and `dirname()` for getting a file's name or directory from a path, for example:

```
>>> file = '/var/lib/db.txt'  
>>> os.path.dirname(file)  
'/var/lib'  
>>> os.path.basename(file)  
'db.txt'
```

In the top level of the repository is a directory called `inputs` that contains several files we'll use for many of the exercises. Here I'll use a file called `inputs/fox.txt`. Note you will need to be in the main directory of the repo for this to work:



```
>>> file = 'inputs/fox.txt'
>>> os.path.isfile(file)
True
```

Once you've determined that the argument is the name of a file, you must `open()` it to `read()` it. The return from `open()` is a *file handle* and is what we use to `read()` the file. I usually call this variable `fh` to remind me that it's a file handle. If I have more than one open file handle like both input and output handles, I may call them `in_fh` and `out_fh`⁴.

```
>>> fh = open(file)
```

WARNING

If you try to `open()` a file that does not exist, you'll get an exception.

This is unsafe code:

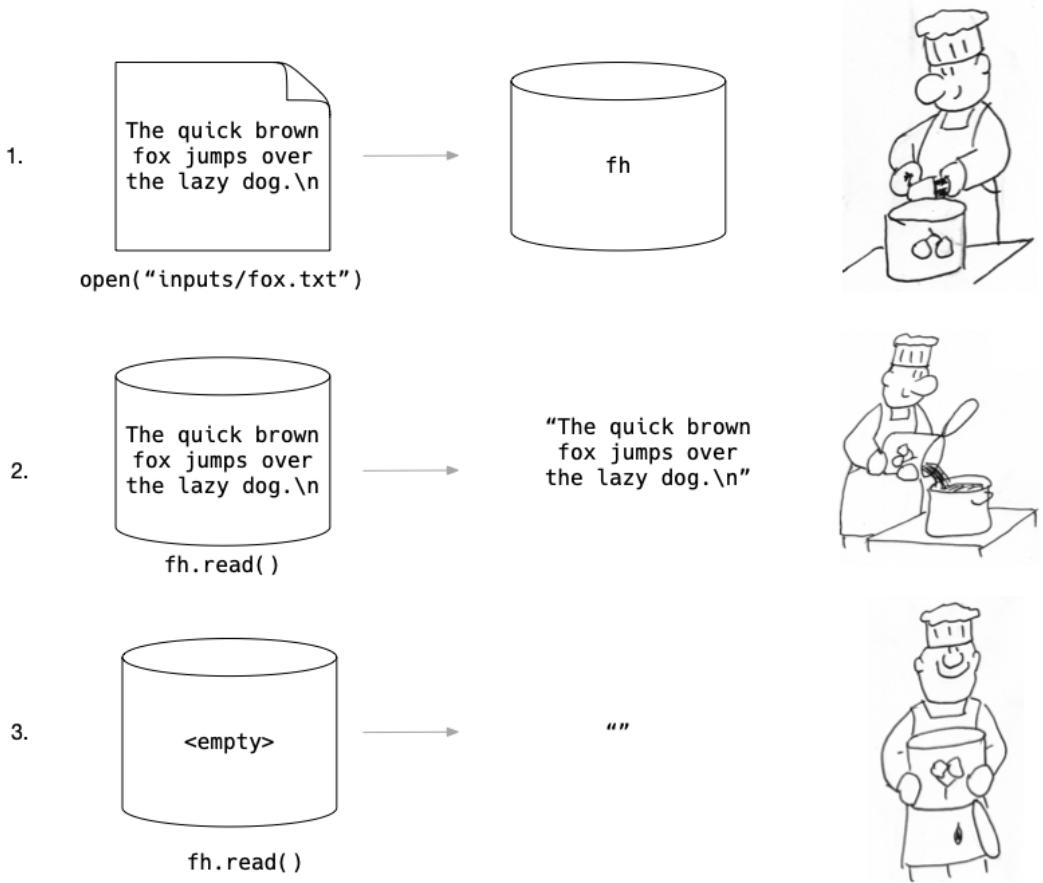
```
>>> file = 'blargh'
>>> open(file)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundException: [Errno 2] No such file or directory: 'blargh'
```

Always check that the file exists!

```
>>> file = 'inputs/fox.txt'
>>> if os.path.isfile(file):
...     fh = open(file)
```

Maybe think of a file like a can of tomatoes. The file's name like "inputs/fox.txt" is the label on the can which is not the same as the *contents*. To get at the text inside (or the "tomatoes"), we need to *open* the can.

⁴ Per www.python.org/dev/peps/pep-0008/#function-and-variable-names, function and variable "names should be lowercase, with words separated by underscores as necessary to improve readability."



1. The file handle (fh) is a mechanism I can use to get at the contents of the file. To get at the tomatoes, we need to `open()` the can.
2. The `fh.read()` method returns what is inside the file. With the can opened, we can get at the contents.
3. Once the file handle has been read, there's nothing left.⁵

Let's see what `type()` the fh is:

```
>>> type(fh)
<class '_io.TextIOWrapper'>
```

In computer lingo, "io" means "input/output." The fh object is something that handles I/O operations. You can use `help(fh)` (using the name of the variable itself) to read the docs on the class `TextIOWrapper`. The two methods you'll use quite often are `read()` and `write()`. Right now, we care about `read`. Let's see what that gives us:

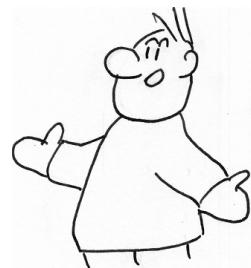
```
>>> fh.read()
```

⁵ You can use `fh.seek(0)` to reset the file handle to the beginning if you want to read it again.

```
'The quick brown fox jumps over the lazy dog.\n'
```

Do me a favor and execute that line one more time. What do you see?

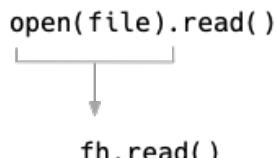
```
>>> fh.read()
''
```



A file handle is different from something like a `str`. Once you read a file handle, it's empty! It's like pouring the tomatoes out of the can. Now the can is empty, and you can't empty it again.

We can actually compress the `open()` and `fh.read()` into one line of code by *chaining* those methods together. The `open()` returns a file handle that can be used for the call to `fh.read()`. Run this:

```
>>> open(file).read()
'The quick brown fox jumps over the lazy dog.\n'
```



And now run it again:

```
>>> open(file).read()
'The quick brown fox jumps over the lazy dog.\n'
```

Each time you `open()` the file, you get a fresh file handle to `read()`.

If you want to preserve the contents, you need to copy them into a variable.

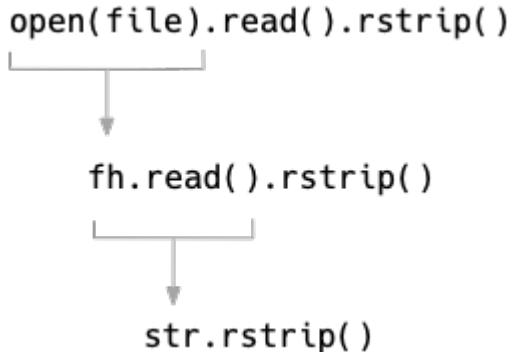
```
>>> text = open(file).read()
>>> text
'The quick brown fox jumps over the lazy dog.\n'
```

The `type()` of the result is a `str`:

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

If I want, I can chain a `str` method onto the end of that. For instance, maybe I want to remove the trailing newline. The `str.rstrip` method will remove any whitespace (which includes newlines) from the *right* end of a string.

```
>>> text = open(file).read().rstrip()
>>> text
'The quick brown fox jumps over the
lazy dog.'
```



Once you have your input text — whether it is from the command line or from a file — you need to UPPERCASE it. The `str.upper()` is probably what you want.

5.2 Writing files

The output of the program should either appear on the command line or be written to a file. Command-line output is also called "standard out" or `STDOUT`. (It's the *standard* or normal place for *output* to occur.) There's also an option to the program to write the output to a file, so let's look at how to do that. You still need to `open()` a file handle, but we have to use an optional second argument, the string '`w`', that instructs Python to open it for *writing*. Other modes include '`r`' for *reading* (the default) and '`a`' for *appending*.

Table 5.1. File writing modes

Mode	Meaning
w	write
r	read
a	append

You can additionally describe the kind of content, whether '`t`' for *text* (the default) or '`b`' for *binary*:

Table 5.2. File content modes

Mode	Meaning
t	text
b	bytes

You can combine these two tables like '`rb`' to *read a binary file* or '`at`' to *append to*

a *text* file. Here we will use '*wt*' to *write* a *text* file. I'll call my variable `out_fh` to remind me that this is the "output file handle":

```
>>> out_fh = open('out.txt', 'wt')
```

If the file does not exist, it will be created. If the file does exist, then it will be *overwritten* which means that all the previous data will be lost! If you don't want an existing file to be lost, you can use the `os.path.isfile()` function we saw earlier to first check if a file exists and perhaps `open()` in the "append" mode instead. For this exercise, we'll use the '*wt*' mode to write text.



You can use the `write()` method of the file handle to put text into the file. Whereas the `print()` function will append a newline (`\n`) unless you instruct it not to, the `write()` method will *not* add a newline, so you have to explicitly add one.

If you use the `fh.write()` method in the REPL, you will see that it returns the number of bytes written. Here each character — including the newline (`\n`) — is a byte:

```
>>> out_fh.write('this is some text\n')
18
```

You can check that this is correct:

```
>>> len('this is some text\n')
18
```

Most code tends to ignore this return value; that is, we don't bother to capture the results into a variable or check that we got a non-zero return. If `write()` fails, there's usually some much bigger problem with your system.

It's also possible to use the `print()` function with the optional `file` argument. Notice that I don't include a newline with `print()` because it will add one:

```
>>> print('this is some more text', file=out_fh)
```

When you are done writing to a file handle, you should `fh.close()` it so that Python can clean up the file and release the memory associated with it. This function returns no value:

```
>>> out_fh.close()
```

We can verify that our text made it. Note that the newline appears here as `\n`. You need to `print()` the string for it to create an actual newline:

```
>>> open('out.txt').read()
'this is some text\nthis is some more text\n'
```

When you `print()` on an open file handle, the text will be appended to any previously

written data. Look at this code, though:

```
>>> print("I am what I am an' I'm not ashamed.", file=open('hagrid.txt', 'wt'))
```

If you run that line twice, will the file called "hagrid.txt" have the line once or twice? Let's find out:

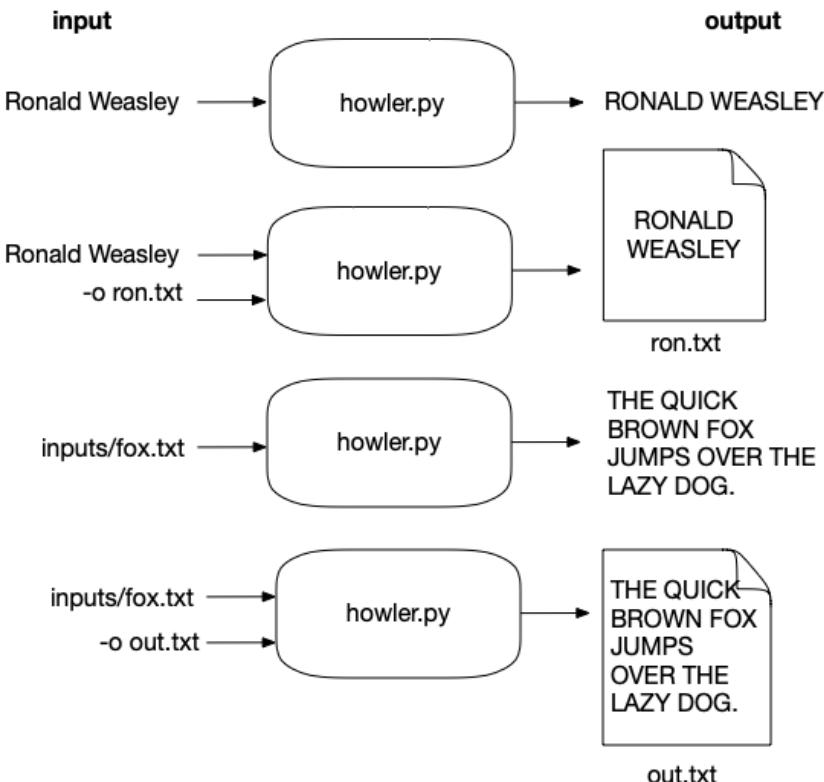
```
>>> open('hagrid.txt').read()
"I am what I am an' I'm not ashamed\n"
```

Just once! Why is that? Remember, each call to `open()` gives us a new file handle, so calling `open()` twice results in new file handles. Each time you run that code, the file is opened anew in *write* mode and the existing data is *overwritten*. So as to avoid confusion, I would recommend instead you write code more along these lines:

```
fh = open('hagrid.txt', 'wt')
fh.write("I am what I am an' I'm not ashamed.\n")
fh.close()
```

5.3 Writing `howler.py`

Here is a string diagram showing the overview of the program and some example inputs and outputs:



When run with no arguments, it should print a short usage:

```
$ ./howler.py
usage: howler.py [-h] [-o str] STR
howler.py: error: the following arguments are required: STR
```

When run with -h or --help, the program should print a longer usage statement:

```
$ ./howler.py -h
usage: howler.py [-h] [-o str] str

Howler (upper-cases input)

positional arguments:
  str                  Input string or file

optional arguments:
  -h, --help            show this help message and exit
  -o str, --outfile str
                        Output filename (default: )
```

If the argument is a regular string, it should uppercase that:

```
$ ./howler.py 'How dare you steal that car!'
HOW DARE YOU STEAL THAT CAR!
```

If the argument is the name of a file, it should uppercase the *contents of the file*:

```
$ ./howler.py ../inputs/fox.txt
THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.
```

If given an --outfile filename, the uppercased text should be written to the indicated file and nothing should be printed to STDOUT:

```
$ ./howler.py -o out.txt ../inputs/fox.txt
$ cat out.txt
THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.
```

Hints:

- Start with new.py and alter the `get_args()` section until your usage statements match the ones above.
- Run the test suite and try to pass just the first test that handles text on the command line and prints the uppercased output to STDOUT.
- The next test is to see if you can write the output to a given file. Figure out how to do that.
- The next test is for reading input from a file. Don't try to pass all the tests at once!
- There is a special file handle that always exists called "standard out" (often `STDOUT`). If you `print()` without a `file` argument, then it defaults to `sys.stdout`. You will need to `import sys` in order to use it.

Be sure you really try to write the program and pass all the tests before moving on to read the solution! If you get stuck, maybe whip up a batch of Polyjuice Potion and freak out your friends.

5.4 Solution

```

#!/usr/bin/env python3
"""Howler"""

import argparse
import os
import sys

# -----
def get_args():
    """get command-line arguments"""

    parser = argparse.ArgumentParser(
        description='Howler (upper-case input)',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('text', metavar='str', help='Input string or file') ①

    parser.add_argument('-o', ②
                        '--outfile',
                        help='Output filename',
                        metavar='str',
                        type=str,
                        default='')

    args = parser.parse_args() ③

    if os.path.isfile(args.text): ④
        args.text = open(args.text).read().rstrip() ⑤

    return args ⑥

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

    args = get_args() ⑦
    out_fh = open(args.outfile, 'wt') if args.outfile else sys.stdout ⑧
    out_fh.write(args.text.upper() + '\n') ⑨
    out_fh.close() ⑩

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

```

- ① The `text` argument is a string that may be the name of a file.
- ② The `--outfile` option is also a string that names a file.
- ③ Parse the command-line arguments into the variable `args` so that we can manually check the `text` argument.
- ④ Check if `args.text` names an existing file.
- ⑤ If so, overwrite the value of `args.text` with the results of reading the file.
- ⑥ Now that we've fixed up the `args`, we can return them to the caller.
- ⑦ Call `get_args()` to get the arguments to the program.

- ⑧ Use an if expression to choose either sys.stdout or a newly opened file handle to write the output.
- ⑨ Use the opened file handle to write the output converted to upper.
- ⑩ Close the file handle.

5.5 Discussion

How did it go for you this time? I hope you didn't sneak into Professor Snape's office again. You really don't want more Saturday detentions.

5.5.1 Defining the arguments

The get_args() function, as always, is the first. Here I define two arguments. The first is a positional text argument. Since it may or may not name a file, all I can know is that it will be a string.

NOTE

The order of positional parameters *relative to each other* is important. The first positional parameter you define will handle the first positional argument provided. It's not important, however, to define positional parameters before or after options and flags. You can declare those in any order you like.

```
parser.add_argument('text', metavar='str', help='Input string or file')
```

The other argument is an option, so I give it a short name of -o and a long name of --outfile. Even though the default type() for all arguments is str, I like to state this explicitly. The default value is the empty string. I could just as easily use the special None type which is also the default value, but I prefer to use a defined argument like the empty string.

```
parser.add_argument('-o',
                   '--outfile',
                   help='Output filename',
                   metavar='str',
                   type=str,
                   default='')
```

5.5.2 Reading input from a file or the command line

This is a deceptively simple program that demonstrates a couple of very important elements of file input and output. The text input might be a plain string or it might be the name of a file. This pattern will come up repeatedly in this book:

```
if os.path.isfile(args.text):
    args.text = open(args.text).read().rstrip()
```

The function os.path.isfile() will tell us if there is a file with the name in text. If that returns True, then we can safely open(file) to get a *file handle* which has a method called read which will return *all* the contents of the file.

WARNING

You should be aware that `fh.read()` will return the *entire file* as a single string. Your computer must have more memory available than the size of the file. For all the programs in this book, that will safe as the files are small. In my day job, I regularly deal with gigabyte-sized files, so calling `fh.read()` would likely crash my program if not my whole system because I would exceed my available memory!

The result of `open(file).read()` is a `str` which has a method called `rstrip` that will return a copy of the string *stripped* of the whitespace off the *right* side. I call this so that the input text will look the same whether it comes from a file or directly from the command line. That is, when you provide the input text directly on the command line, you have to press `<Enter>` to terminate the command. That `<Enter>` is a newline, and the operating system automatically removes it before passing it to the program. I want the text to look the same whether it comes directly from the command line or from a file.

The longer way to write the above would be:

```
if os.path.isfile(text):
    fh = open(text)
    text = fh.read()
    text = text.rstrip()
    fh.close()
```

In my version, I choose to handle this inside the `get_args()` function. This is the first time I'm showing you that you can intercept and alter the arguments before passing them on to `main()`. We'll use this idea quite a bit in more exercises. I like to do all the work to validate the user's arguments inside `get_args()`. I could just as easily do this in `main()` after the call to `get_args()`, so this is entirely a style issue.

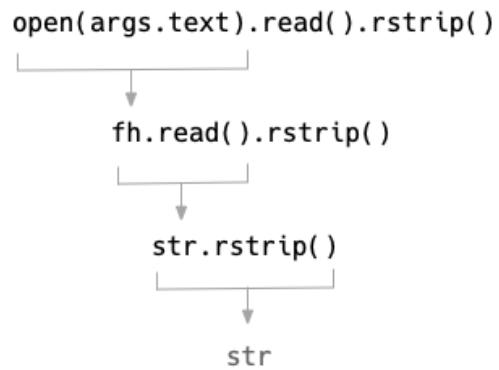
5.5.3 Choosing the output file handle

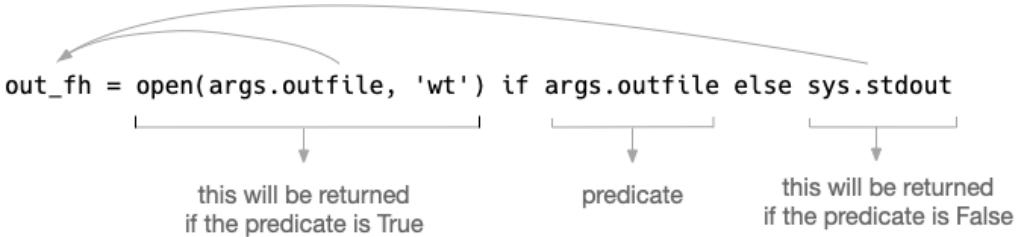
This line decides where to put the output of our program:

```
out_fh = open(args.outfile, 'wt') if args.outfile else sys.stdout
```

The `if` expression will open `args.outfile` for writing text (`wt`) if the user provided that argument; otherwise, we will use `sys.stdout` which is a file handle to STDOUT. Note that we don't have to call `open()` on `sys.stdout` because it is always available and open for business.

Figure 5.1. The `open()` function returns a file handle (`fh`). The `fh.read()` function returns a `str`. The `str.rstrip()` function returns a new `str` with the whitespace removed from the right side. All these functions can be chained together.





5.5.4 Printing the output

To get uppercase, we can use the `text.upper()` method, then we need to find a way to print it to the output file handle. I chose to do:

```
out_fh.write(text.upper())
```

But you could also do:

```
print(text.upper(), file=out_fh)
```

Finally I need to close the file handle with `out_fh.close()`.

5.5.5 A low-memory version

There is a potentially serious problem waiting to bite us in this program. In the `get_args()`, we're reading the entire file into memory with this line:

```
if os.path.isfile(args.text):
    args.text = open(args.text).read().rstrip()
```

We could, instead, only `open()` the file:

```
if os.path.isfile(args.text):
    args.text = open(args.text)
```

And later read it line-by-line:

```
for line in args.text:
    out_fh.write(line.upper())
```

The problem, though, is how to handle the times when the `text` argument is actually `text` and not the name of a file. The `io` (input-output) module in Python has a way to represent text as *stream*:

```
>>> import io
>>> text = io.StringIO('foo\nbar\nbaz\n')
>>> for line in text:
...     print(line, end='')
...
foo
bar
baz
```

① Import the `io` module.

- ② Use the `io.StringIO()` function to turn the given `str` value into something we can treat like an open file handle.
- ③ Use a `for` loop to iterate the "lines" of text by separated by newlines.
- ④ Print the line using the `end=''` option to avoid having 2 newlines.

This is the first time we're seeing that we can treat a regular string value as if it were a generator of values similar to a file handle. This is a particularly useful technique for testing any code that needs to read an input file. You can use the return from `io.StreamIO()` as a "mock" file handle so that your code doesn't have to read an *actual* file, just a given value that can produce "lines" of text!

To make this work, we can change how we handle the `args.text` like so:

```
#!/usr/bin/env python3
"""Low-memory Howler"""

import argparse
import os
import io
import sys

# -----
def get_args():
    """get command-line arguments"""

    parser = argparse.ArgumentParser(
        description='Howler (upper-cases input)',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('text', metavar='str', help='Input string or file')

    parser.add_argument('-o',
                        '--outfile',
                        help='Output filename',
                        metavar='str',
                        type=str,
                        default='')

    args = parser.parse_args()

    if os.path.isfile(args.text):
        args.text = open(args.text)          ①
    else:
        args.text = io.StringIO(args.text + '\n') ③

    return args

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

    args = get_args()
    out_fh = open(args.outfile, 'wt') if args.outfile else sys.stdout
    for line in args.text:                 ④
        out_fh.write(line.upper()) ⑤
    out_fh.close()
```

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

- ① Check if args.text is a file.
- ② If so, replace args.text with the file handle created by opening the file named by it.
- ③ Otherwise, replace args.text with an io.StringIO() value that will act like an open file handle.
Note that we need to add a newline to the text so that it will look like the lines of input coming from an actual file.
- ④ Read the input (whether io.StringIO() or a file handle) line-by-line.
- ⑤ Handle the line.

5.6 Review

- To `read()` or `write()` to files, you must `open()` them.
- The default mode for `open()` is for reading a file.
- To write a text file, you must use '`wt`' as the second argument to `open()`.
- Text is the default type of data that you `write()` to a file handle. You must use the '`b`' flag to indicate that you want to write binary data.
- The `os.path` module contains many useful functions such as `os.path.isfile()` that will tell you if a file exists by the given name.
- `STDOUT` (standard output) is always available via the special `sys.stdout` file handle which is always open.
- The `print()` function takes an optional `file` argument of where to put the output. That argument must be an open file handle such as `sys.stdout` (the default) or the result of `open()`.



5.7 Going Further

- Add a flag that will lowercase the input instead. Maybe call it `--ee` for the poet e e cummings who liked to write poetry devoid of uppercase letters.
- Alter the program handle multiple input files
- Change `--outfile` to `--outdir` and write each input file to the same file name in the output directory.

Words Count: Reading files/STDIN, iterating lists, formatting strings

"I love to count!" — Count von Count

Counting things is a surprisingly important programming skill. Maybe you're trying to find how many pizzas were sold each quarter or how many times you see certain words in a set of documents. Usually the data we deal with in computing comes to us in files, so we're going to push a little further into reading files and manipulating strings by writing a Python version of the venerable Unix `wc` ("word count") program.



We're going to write a program called `wc.py` that will count the lines, words, and bytes found in each input. The counts will appear in columns 8 characters wide and will be followed by the name of the file. The inputs for the program which may be given as one or more positional arguments. For instance, here is what it should print for one file:

```
$ ./wc.py ../inputs/scarlet.txt
 7035 68061 396320 ../inputs/scarlet.txt
```

When counting multiple files, there will be an additional "total" line summing each column:

```
$ ./wc.py ../inputs/const.txt ../inputs/sonnet-29.txt
 865    7620   44841 ../inputs/const.txt
    17     118    661 ../inputs/sonnet-29.txt
```

```
882    7738   45502 total
```

There may also be *no* arguments, in which case we'll read from "standard in" which is often written as STDIN. We started talking about STDOUT in "Howler" when we used `sys.stdout` as a file handle. STDIN is the complement to STDOUT — it's the "standard" place to read input on the command line. When our program is given *no* positional arguments, we'll read from `sys.stdin`.

For instance, the `cat` program will print the contents of a file to STDOUT. We can use the pipe operator (`|`) to funnel that output into our program:

```
$ cat ../inputs/fox.txt | ./wc.py
      1      9      45 <stdin>
```

Another option is to use the `<` operator to redirect input from a file:

```
$ ./wc.py < ../inputs/fox.txt
      1      9      45 <stdin>
```

Tools that print to STDOUT and read from STDIN can be chained together to create novel, *ad hoc* programs! One of the handiest command-line tools is `grep` which can find patterns of text in files. If, for instance, we wanted to find all the lines of text that contain the word "scarlet" in all the files in the `inputs` directory using this command:

```
$ grep scarlet ../inputs/*.txt
```

On the command line, the `*` is a wildcard that will match anything so `*.txt` will match any file ending with `.txt`. If you run that command, you'll see quite a bit of output. To count the lines found by `grep`, we can "pipe" that output into our `wc.py` program like so:

```
$ grep scarlet ../inputs/*.txt | ./wc.py
    104    1188    9182 <stdin>
```

In this exercise, you will:

- Learn how to process zero or more positional arguments
- Validate input files
- Read from files or from "standard in"
- Use multiple levels of `for` loops
- Break files into lines, words, and bytes
- Use counter variables
- Format string output

6.1 Writing `wc.py`

Let's get started! Create your program and modify the arguments until it will print the following usage if run with the `-h` or `--help` flags:

```
$ ./wc.py -h
```

```
usage: wc.py [-h] [FILE [FILE ...]]

Emulate wc (word count)

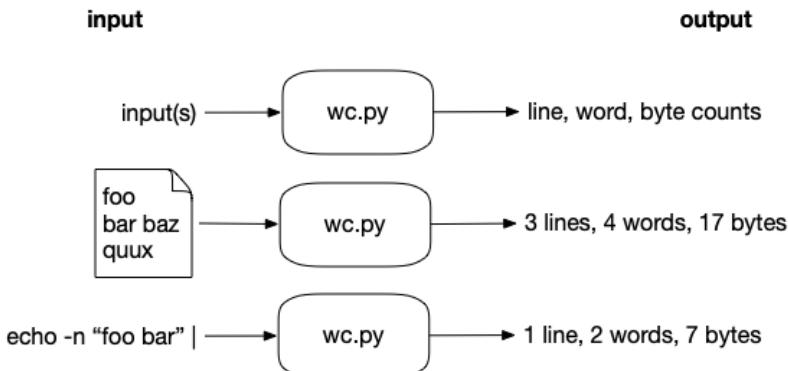
positional arguments:
  FILE      Input file(s) (default: [<_io.TextIOWrapper name='<stdin>' mode='r' encoding='UTF-8']])

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

Given a non-existent file, your program should print an error message and exit with a non-zero exit value:

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

Here is a string diagram to help you think about how the program should work:



6.1.1 Defining file inputs

The first step will be to define your arguments to `argparse`. This program takes *zero or more* positional arguments and nothing else. Remember that you never have to define the `-h` or `--help` arguments as `argparse` handles those automatically.

In "Picnic," we used `nargs='+'` to indicate one or more items for our picnic. Here we want to use `nargs='*'` to indicate *zero or more*. For what it's worth, there's one other value that `nargs` can take and that is `?` for *zero or one*. In all cases, the argument(s) will be returned as a list. Even if there are no arguments, you will still get an empty list (`[]`). For this program, if there are no arguments, we'll read STDIN.

Table 6.1. Possible values for `nargs`

Symbol	Meaning
<code>?</code>	zero or one
<code>*</code>	zero or more
<code>+</code>	one or more

Any arguments that are provided to our program *must be readable files*. In "Howler" we learned how to test if the input argument is a file by using `os.path.isfile`. The input was allowed be either plain text or a file name, so we had to check this ourselves.

In this program, the input arguments are required to be files, so we can define our arguments using `type=argparse.FileType('r')`. This means that `argparse` takes on all the work to validate the inputs from the user and produce useful error messages. If the user provides valid input, then `argparse` will provide you with a *list of open file handles*. All in all, this saves you quite a bit of time. (Be sure to review the "File arguments" section in the `argparse` appendix.)

In "Howler," we used `sys.stdout` to write to `STDOUT`. To read from `STDIN`, we'll use Python's `sys.stdin` file handle. Like `sys.stdout`, the `sys.stdin` file handle does not need an `open()` — it's always present and available for printing.

Because we are using `nargs` to define our argument, the result will always be a *list*. In order to set `sys.stdin` as the `default` value, we should place it in a *list* like so:

```
parser.add_argument('file',
                    metavar='FILE',
                    nargs='*',
                    type=argparse.FileType('r'), ①
                    default=[sys.stdin], ②
                    help='Input file(s)') ③
```

- ① Zero or more of this argument.
- ② If arguments are provided, they must be readable files. The files will be opened by `argparse` and will be provided as file handles.
- ③ The default will be a *list* containing `sys.stdin` which is like an open file handle to `STDIN`. We do not need to `open()` it.

6.1.2 Iterating lists

Your program will end up with a *list* of file handles. In "Jump The Five," we used a `for` loop to iterate through the characters in the input text. Here we can use a `for` loop over the `file` inputs.

```
for fh in args.file:
    # read each file
```

The `fh` is a "file handle." We saw in "Howler" how to manually `open()` and `read()` a file. Here the `fh` is already open, so we can read the contents from it. There are many ways to read a file, however. The `fh.read()` method will give you the *entire contents* of the file in one go. If the file is large — say, if the size of the file exceeds your available memory on your machine — then your program will crash. I would recommend, instead, that you use a `for` loop on the `fh`. Python will understand this to mean that you wish to read each line of input, one-at-a-time.

```
for fh in args.file: # ONE LOOP!
    for line in fh: # TWO LOOPS!
        # process the line
```

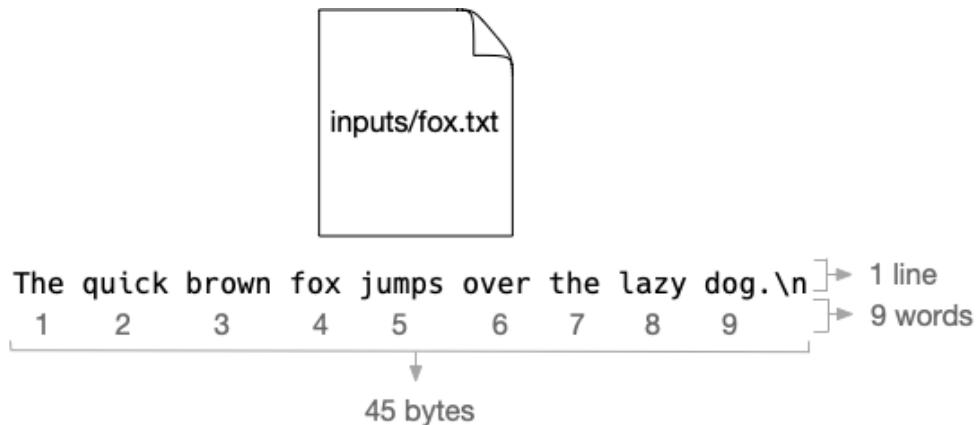
So that's two levels of `for` loops, one for each file handle and then another for each line in each file handle. TWO LOOPS! I LOVE TO COUNT!

6.1.3 ***What you're counting***

The output for each file will be the number of lines, words, and bytes (like characters and whitespace), each printed in a field 8 characters wide followed by a space and then the name of the file which will be available to you via `fh.name`. Let's take a look at the output from the standard `wc` program on my system. Notice that when run with just one argument, it produces counts only for that file:

```
$ wc fox.txt
    1      9     45 fox.txt
```

The `fox.txt` file is short enough that you could manually verify that it does in fact contain one line, nine words, and 45 bytes which includes all the characters, spaces, and the trailing newline:



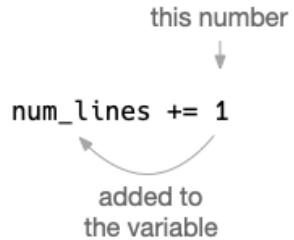
When run with multiple files, the standard `wc` program also shows a "total" line:

```
$ wc fox.txt sonnet-29.txt
    1      9     45 fox.txt
    17    118    669 sonnet-29.txt
    18    127    714 total
```

We are going to emulate the behavior of this program. For each file, you will need to create variables to hold the numbers for lines, words, and bytes. For instance, if you use the `for line in fh` loop that I suggest, then you need to have a variable like `num_lines` to increment on each iteration.

That is, somewhere in your code you will need to set a variable to 0 and then, inside the `for` loop, make it go up by one. The idiom in Python is to use the `+=` operator to add some value on the right-hand side to the variable on the left-hand-side like so:

```
num_lines = 0
for line in fh:
    num_lines += 1
```



You will also need to count the number of words and bytes, so you'll need similar `num_words` and `num_bytes` variables. To get the "words," we'll use the `str.split()` method to break each line on spaces.⁶ You can then use the length of the resulting list as the number of words. For the number of bytes, you can use the `len()` (length) function on the `line` and add that to a `num_bytes` variable.

6.1.4 Formatting your results

This is the first exercise where the output needs to be formatted in a particular way. Don't try handle this part manually. That way lies madness. Instead, you need to learn the magic of the `str.format()` method. The help doesn't have much in the way of documentation, so I'd recommend you read PEP3101 (www.python.org/dev/peps/pep-3101/).

We've seen that the curly braces `{}` inside the `str` part create placeholders that will be replaced by the values passed to the method:

```
>>> import math
>>> 'Pi is {}'.format(math.pi)
'Pi is 3.141592653589793'
```

You can put formatting information inside the curly braces to specify how you want the value displayed. If you are familiar with `printf()` from C-type languages, this is the same idea. For instance, I can print just two numbers of pi after the decimal. The `:` introduces the formatting options, and the `0.02f` describes two decimal points of precision:

```
>>> 'Pi is {:.02f}'.format(math.pi)
'Pi is 3.14'
```

The formatting information comes after the colon `:` inside the curly braces. You can also use the f-string method where the variable comes *before* the colon:

```
>>> f'Pi is {math.pi:0.02f}'
'Pi is 3.14'
```

Here you need to use `{:8}` for each of lines, words, and characters so that they all line up in neat columns. The `8` describes the width of the field which is assumed to be a

⁶ Splitting the text on spaces doesn't actually produce "words" because it won't separate the punctuation like commas and periods from the letters, but it's close enough for this program.

string. The text will be right-justified. Place a single space between the last column and the name of the file which you can find in `fh.name`.

Hints:

- Start with `new.py` and delete all the non-positional arguments.
- Use `nargs='*' to indicate zero or more positional arguments for your file argument.`
- How could you use `sys.stdin` for the `default`? Remember that both `narg='*' and nargs='+' mean that the arguments will be supplied as a list. How can you create a list that contains just sys.stdin for the default value?`
- Remember that you are just trying to pass one test at a time. Create the program, get the help right, then worry about the first test.
- Compare the results of your version to the `wc` installed on your system. Note that not every Unix-like system has the same `wc`, so results may vary.

Time to write this yourself before you read the solution. Fear is the mind-killer. You can do this.

6.2 Solution

```
#!/usr/bin/env python3
"""Emulate wc (word count)"""

import argparse
import sys

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

    parser = argparse.ArgumentParser(
        description='Emulate wc (word count)',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file',
                        metavar='FILE',
                        nargs='*',
                        default=[sys.stdin], ①
                        type=argparse.FileType('r'), ②
                        help='Input file(s)')

    return parser.parse_args()

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

    args = get_args()

    total_lines, total_bytes, total_words = 0, 0, 0 ③
    for fh in args.file: ④
        num_lines, num_words, num_bytes = 0, 0, 0 ⑤
```

```

for line in fh:                      ⑥
    num_lines += 1                   ⑦
    num_bytes += len(line)          ⑧
    num_words += len(line.split())   ⑨

    total_lines += num_lines        ⑩
    total_bytes += num_bytes
    total_words += num_words

    print(f'{num_lines:8}{num_words:8}{num_bytes:8} {fh.name}') ⑪

if len(args.file) > 1:                ⑫
    print(f'{total_lines:8}{total_words:8}{total_bytes:8} total') ⑬

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

```

- ① If you set the default to a list with `sys.stdin`, then you have handled the STDIN option.
- ② If the user supplies any arguments, `argparse` will check if they are valid file inputs. If there is a problem, `argparse` will halt execution of the program and show the user an error message.
- ③ These are the variables for the "total" line, if we need them.
- ④ Iterate through the list of `arg.file` inputs. I use the variable `fh` to remind me that these are open file handles, even STDIN.
- ⑤ Initialize variables to count *just this file*.
- ⑥ Iterate through each line of `fh`.
- ⑦ For each line, we increment `lines` by 1.
- ⑧ The number of bytes is incremented by the length of the line.
- ⑨ To get the number of words, we can call `line.split()` the line on spaces (the default). The length of that list is added to the words.
- ⑩ Add all the counts for lines, words, and bytes for this file to the `total_` variables.
- ⑪ Print the counts for this file using the `{ :8 }` option to print in a field 8 characters wide followed by a single space and then the name of the file.
- ⑫ Check if we had more than 1 input.
- ⑬ Print the "total" line.

6.3 Discussion

6.3.1 Defining the arguments

This program is rather short and seems rather simple, but it's definitely not exactly easy. One part of the exercise is to really get familiar with `argparse` and the trouble it can save you. The key is in defining the `file` positional arguments. If you use `nargs='*'` to indicate zero or more arguments, then you know `argparse` is going to give you back a list with zero or more elements. If you use `type=argparse.FileType('r')`, then any arguments provided must be readable files. The list that `argparse` returns will be a list of *open file handles*. Lastly, if you use `default=[sys.stdin]`, then you understand that `sys.stdin` is essentially an open file handle to read from "standard in" (AKA STDIN), and you are letting `argparse` know

that you want the default to be a list containing `sys.stdin`.

6.3.2 **Reading a file using a for loop**

I can create a list of open file handles in the REPL to mimic what I'd get from `args.file`:

```
>>> files = [open('../inputs/fox.txt')]
```

Before I use a for loop to iterate through them, I need to set up three variables to track the *total* number of lines, words, and characters. I could define them on three separate lines:

```
>>> total_lines = 0
>>> total_words = 0
>>> total_bytes = 0
```

Or I can declare them on a single line. Technically I'm creating a tuple on the right-hand side by placing commas in between the three zeros, and I'm "unpacking" those three values into three variables on the left-hand side. We'll have more to say about tuples much later:

```
>>> total_lines, total_words, total_bytes = 0, 0, 0
```

Inside the for loop for each file handle, I initialize three more variables to hold the count of lines, characters, and words *for this particular file*. I then use another for loop to iterate over each line in the file handle (`fh`). For the lines, I can add 1 on each pass through the for loop. For the bytes, I can add length of the line (`len(line)`) to track the number of "characters" (which may be printable characters or whitespace so it's easiest to call them "bytes"). Lastly for the words, I can use `line.split()` to break the line on whitespace to create a list of "words." It's not actually a perfect way to count actual words, but it's close enough. I can use the `len()` function on the list to add to the words variable. The for loop ends when the end of the file is reached, and that is when I can `print()` out the counts and the file name using `{:8}` placeholders in the print template to indicate a text field 8 characters wide.

```
>>> for fh in files:
...     lines, words, bytes = 0, 0, 0
...     for line in fh:
...         lines += 1
...         bytes += len(line)
...         words += len(line.split())
...     print(f'{lines:8}{words:8}{bytes:8} {fh.name}')
...     total_lines += lines
...     total_bytes += bytes
...     total_words += words
...
1      9      45 ../inputs/fox.txt
```

Notice that the call to `print()` lines up with the *second* for loop so that it will run after we're done iterating over the lines in `fh`. I chose to use the f-string method to print each of lines, words, and bytes in a space 8 characters wide followed by one space and then the `fh.name` of the file. After printing, I can add the counts to my "total"

variables to keep a running total.



Lastly, if the number of file arguments is greater than 1, I need to print my totals:

```
if len(args.file) > 1:  
    print(f'{total_lines:8}{total_words:8}{total_bytes:8} total')
```

6.4 Review

- The `nargs` (number of arguments) option to `argparse` allows you to validate the number of arguments from the user. The star ('*') means zero or more while '+' means one or more.
- If you define an argument using `type=argparse.FileType('r')`, then `argparse` will validate that the user has provided a readable file and will make the value available in your code as an open file handle.
- You can read and write from the Unix standard in/out file handles by using `sys.stdin` and `sys.stdout`.
- You can nest `for` loops to handle multiple levels of processing.
- The `str.split()` method will split a string on spaces into words.
- The `len()` function can be used on both strings and lists. For the latter, it will tell you the number of elements contained.
- The `str.format()` and Python's f-strings both recognize the same printf-style formatting options to allow you to control how a value is displayed.

6.5 Going Further

- By default, `wc` will print all the columns like our program, but it will also accept flags to print `-c` for number of characters, `-l` for number of lines, and `-w` for number of words. When any of these flags are present, only those columns for the given flags are shown, so `wc.py -wc` would show just the columns for words and characters. Add both short and long flags for these options to your program so that it behaves exactly like `wc`.
- Write your own implementation of other system tools like `cat` (to print the contents of a file to `STDOUT`), `head` (to print just the first `n` lines of a file), `tail` (to print the last `n` lines of a file), and `tac` (to print the lines of a file in reverse order).

7

Gashlycrumb: Looking items up in a dictionary

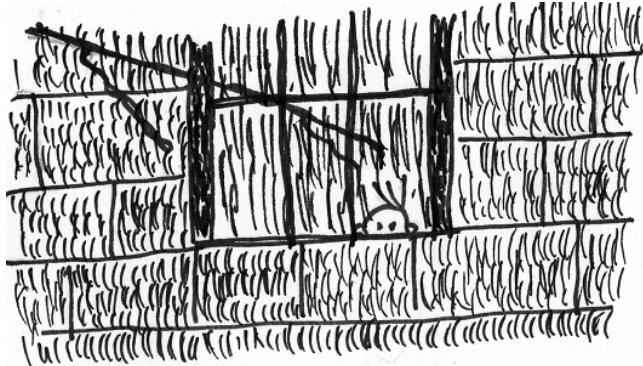
Every time you log into a website, the code behind it has to look up your username and password to compare to the values you put into the login form. Whenever you give your phone number at the hardware store or scan your library card to checkout a book, a computer program uses one piece of information to find other things like how often you buy compost or if you have any overdue books. Probably all these examples would be using a database to find that information. We're going to use a dictionary that we will fill with information from an input file.

In this exercise, we're going to look up a lines of text from an input file that start with the letters provided by the user. The text will come from an input file which will default to Edward Gorey's "The Gashlycrumb Tinies," an abecedarian book that describes various and ghastly ways in which children die. Instead of "A is for artichoke, B is for blackberry," we get:

A is for Amy who fell down the stairs. B is for Basil assaulted by bears.

Our `gashlycrumb.py` program will take one or more letters of the alphabet as positional

Figure 7.1. N is for Neville who died of ennui.



arguments and will look up the lines of text from an *optional* input file that start with that letter. The input file will have each letter on a separate line:

```
$ head -2 gashlycrumb.txt
A is for Amy who fell down the stairs.
B is for Basil assaulted by bears.
```

When our unfortunate user runs our program, here is what they would see. Note that we will consider the letter in a *case-insensitive* fashion:

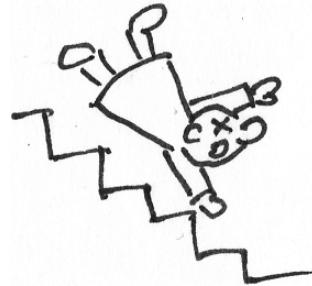
```
$ ./gashlycrumb.py e f
E is for Ernest who choked on a peach.
F is for Fanny sucked dry by a leech.
```

In this exercise, you will:

- Accept one or more positional arguments we'll call `letter`
- Accept an optional `--file` argument which must be a file. The default value will be '`gashlycrumb.txt`' (provided).
- Read the file, find the first letter of each line, and build a data structure that associates that letter to the line of text. (We'll only be using files where each line starts with a single, unique letter. This program would fail with any other format of text.)
- For each letter provided by the user, either print the line of text for the letter if present or a message if it isn't.
- Learn how to "pretty print" a data structure.

You can draw from several previous programs:

- From the "Word Count" program, you know how to take a file input and read it line-by-line.
- From the "Crow's Nest" program, you know how to get the first letter of a bit of text.
- From the "Jump The Five" program, you know how to build a dictionary and lookup a value.



Now you'll put all those skills together to recite morbid poetry!

7.1 Writing `gashlycrumb.py`

Before you begin writing, I would encourage you to run the tests with `make test` or `pytest -xv test.py` in the `gashlycrumb` directory. The first test should fail:

```
test.py::test_exists FAILED
```

This is just a reminder that the first thing to do is to create the file called `gashlycrumb.py`. You can do this however you like, maybe by running `new.py gashlycrumb.py` in the `gashlycrumb` directory or by copying the `template/template.py` file, or by just starting a new file from scratch.

Run your tests again and you should pass the first and possibly the second tests if your program produces a usage statement. Next let's get the arguments straight. Modify your program's parameters in the `get_args()` function so that it will produce the following usage when the program is run with *no arguments* or with the `-h` or `--help` flags:

```
$ ./gashlycrumb.py -h
usage: gashlycrumb.py [-h] [-f FILE] letter [letter ...]

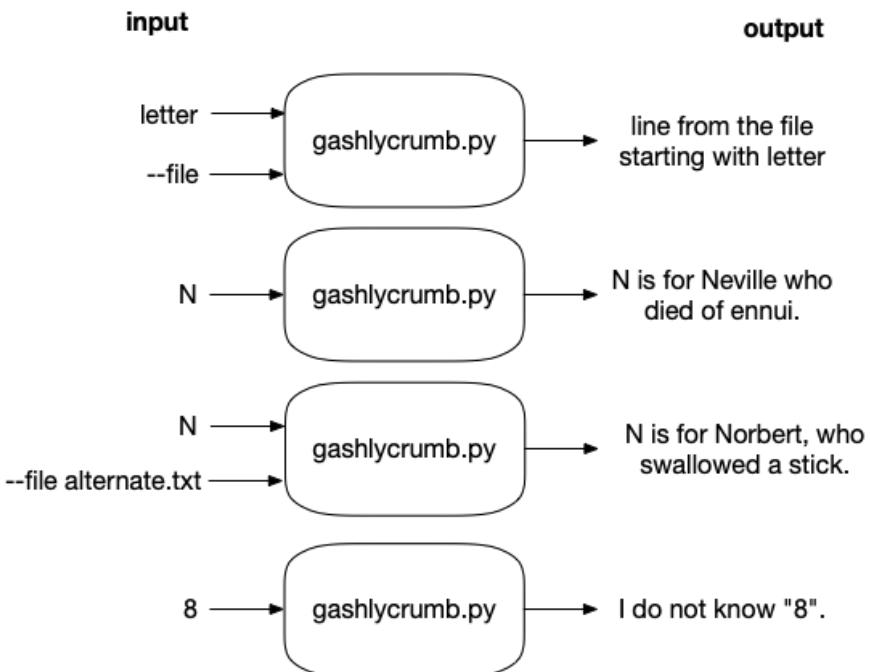
Gashlycrumb

positional arguments:
  letter            Letter(s)          ①

optional arguments:
  -h, --help         show this help message and exit      ②
  -f FILE, --file FILE  Input file (default: gashlycrumb.txt) ③
```

- ① The letter is a required positional argument that accepts one or more values (notice `[str ...]` to indicate this).
- ② The `-h` and `--help` arguments are created automatically by `argparse`.
- ③ The `-f` or `--file` argument is an option with a default value of `gashlycrumb.txt`.

Here is a string diagram showing how the program will work:



Once you have the correct usage, start off by echoing each of the `letter` arguments:

```
def main():
    args = get_args()
    for letter in args.letter:
        print(letter)
```

Try running it to make sure it works:

```
$ ./gashlycrumb.py a b
a
b
```

Next read the file line-by-line using a for loop. Note that I'm using `end=''` with the `print` so that it won't print the newline that's already attached to each line of the file:

```
def main():
    args = get_args()
    for letter in args.letter:
        print(letter)

    for line in args.file:
        print(line, end='')
```

Try running it to ensure you can read the input file:

```
$ ./gashlycrumb.py a b | head -4
a
b
A is for Amy who fell down the stairs.
B is for Basil assaulted by bears.
```

Use the "alternate.txt" file, too:

```
$ ./gashlycrumb.py a b --file alternate.txt | head -4
a
b
A is for Alfred, poisoned to death.
B is for Bertrand, consumed by meth.
```

If provided a `--file` argument that does not exist, your program should exit with an error and message. Note that, if you use `type=argparse.FileType('r')`, this error should be produced automatically by `argparse`:

```
$ ./gashlycrumb.py -f blaragh b
usage: gashlycrumb.py [-h] [-f FILE] letter [letter ...]
gashlycrumb.py: error: argument -f/--file: can't open 'blaragh': \
[Errno 2] No such file or directory: 'blaragh'
```

Now think about how you can use the first letter of each line to create an entry into a `dict`. Use the `print` command to look at your dictionary. Figure out how to check if the given letter is in (wink, wink, nudge, nudge) your dictionary. If given a value that does not exist in the list of first characters on the lines from the input file (when searched without regard to case), you should print a message:

```
$ ./gashlycrumb.py 3
I do not know "3".
$ ./gashlycrumb.py CH
I do not know "CH".
```



If the given letter is in the dictionary, print the value for it:

```
$ ./gashlycrumb.py a
A is for Amy who fell down the stairs.
$ ./gashlycrumb.py z
Z is for Zillah who drank too much gin.
```



Run the test suite to ensure your program meets all the requirements. Read the errors closely and fix your program.

Hints:

- Start with `new.py` and remove everything but the positional `letter` and optional `--file` parameters.
- Use `type=argparse.FileType('r')` to validate the `--file` argument.
- Use `nargs='+'` to define the positional argument `letter` so it will require one or more values.
- A dictionary is a natural data structure that you can use to associate some value like the letter "A" to some phrase like "A is for Amy who fell down the stairs." Create a new, empty dict.
- Once you have an open file handle, you can read the file line-by-line with a `for` loop.
- Each line of text is a string. How can you get the first character of a string?
- Using that first character, how can you set the value of a `dict` to be the key and the line itself to be the value?
- Iterate through each `letter` argument. How can you check that a given value is in the dictionary?

No skipping ahead to the solution until you have written your own version! If you peek, you will die a horrible death stampeded by kittens.

7.2 Solution

```
#!/usr/bin/env python3
"""
Lookup tables"""

import argparse
```

```

# -----
def get_args():
    """get command-line arguments"""

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

    parser.add_argument('letter',
                        help='Letter(s)',
                        metavar='letter',
                        nargs='+',          ①
                        type=str)

    parser.add_argument('-f',
                        '--file',
                        help='Input file',
                        metavar='FILE',
                        type=argparse.FileType('r'), ②
                        default='gashlycrumb.txt')

    return parser.parse_args()

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

    args = get_args()

    lookup = {}           ③
    for line in args.file: ④
        lookup[line[0].upper()] = line.rstrip()      ⑤

    for letter in args.letter: ⑥
        if letter.upper() in lookup:                ⑦
            print(lookup[letter.upper()])           ⑧
        else:
            print(f'I do not know "{letter}".')     ⑨

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

```

- ① A positional argument we'll call letter that uses nargs='+' to indicate one or more values are required.
- ② The optional --file argument must be a readable file because of type=argparse.FileType('r'). The default value is the gashlycrumb.txt file which we know to exist.
- ③ Create an empty dictionary to hold our lookup table.
- ④ Iterate through each line of the args.file which will be an open file handle.
- ⑤ Uppercase the first character of the line to use as the key into the lookup and set the value to be the line stripped of whitespace on the right side.
- ⑥ Use a for loop to iterate over each letter in args.letter.
- ⑦ See if the letter is in the lookup dictionary, checking the letter.upper() value to disregard case.

- ⑧ If so, print the line of text from the lookup for the letter.
- ⑨ Otherwise, print a message that the letter is unknown.

7.3 Discussion

Did the frightful paws of the kittens hurt much? Let's talk about how I solved this problem. Remember, mine is just one of many possible solutions.

7.3.1 Handling the arguments

I prefer to have all the logic for parsing and validating the command-line arguments in the `get_args()` function. In particular, `argparse` can do a fine job verifying tedious things such as an argument being an existing, readable file which is why I use `type=argparse.FileType('r')` for that argument. If the user doesn't supply a valid arguments, then `argparse` will throw an error, printing a helpful message along with the short usage and exiting with an error code.

By the time I get to the line `args = get_args()`, I know that I have one or more "letter" arguments and a valid, open file handle in the `args.file` slot. In the REPL, I can use `open` to get a file handle which I like to usually like to call `fh`. For copyright purposes, I'll use my alternate text:

```
>>> fh = open('alternate.txt')
```

7.3.2 Reading the input file

We know that we want to use a dictionary where the keys are the first letters of each line and the values are the lines themselves. That means we need to start by creating a new, empty dictionary either by using the `dict()` function or by setting a variable equal to an empty set of curly braces (`{}`). I'll call my variable `lookup`:

```
>>> lookup = {}
```

We can use a `for` loop to read each line of text. From the "Crow's Nest" program, we know we can use `line[0].upper()` to get the first letter of `line` and uppercase it. We can use that as the key into `lookup`. Each line of text ends with a newline that I'd like to remove. The `str.rstrip()` method will strip whitespace from the right side of the line (`rstrip = right strip`). The result of that will be the value for my `lookup`:

```
for line in fh:
    lookup[line[0].upper()] = line.rstrip()
```

We'd like to look at the resulting `lookup` dictionary. We can print it from the program or type `lookup` in the REPL, but it's going to be hard to read. I encourage you to try it. Luckily there is a lovely module called `pprint` to "pretty print" data structures. Here is how we can import the `pprint()` function from the `pprint` module with the alias `pp`:

```
from pprint import pprint as pp
```



```
>>> from pprint import pprint as pp
```

Now let's take a peek at lookup:

```
>>> pp(lookup)
{'A': 'A is for Alfred, poisoned to death.', 'B': 'B is for Bertrand, consumed by meth.', 'C': 'C is for Cornell, who ate some glass.', 'D': 'D is for Donald, who died from gas.', 'E': 'E is for Edward, hanged by the neck.', 'F': 'F is for Freddy, crushed in a wreck.', 'G': 'G is for Geoffrey, who slit his wrist.', 'H': "H is for Henry, who's neck got a twist.", 'I': 'I is for Ingrid, who tripped down a stair.', 'J': 'J is for Jared, who fell off a chair.', 'K': 'K is for Kevin, bit by a snake.', 'L': 'L is for Lauryl, impaled on a stake.', 'M': 'M is for Moira, hit by a brick.', 'N': 'N is for Norbert, who swallowed a stick.', 'O': 'O is for Orville, who fell in a canyon.', 'P': 'P is for Paul, strangled by his banyan.', 'Q': 'Q is for Quintanna, flayed in the night.', 'R': 'R is for Robert, who died of spite.', 'S': 'S is for Susan, stung by a jelly.', 'T': 'T is for Terrange, kicked in the belly.', 'U': "U is for Uma, who's life was vanquished.", 'V': 'V is for Victor, consumed by anguish.', 'W': "W is for Walter, who's socks were too long.", 'X': 'X is for Xavier, stuck through with a prong.', 'Y': 'Y is for Yoeman, too fat by a piece.', 'Z': 'Z is for Zora, smothered by a fleece.'}
```

Hey, that looks like a handy data structure. Hooray for us!

7.3.3 Using a *dictionary comprehension*

In "Jump the Five," we saw that a list comprehension is a way to build a list by putting a for loop inside []. If we change the brackets to curly braces ({}), then we create a dictionary comprehension. This code is equivalent to the for loop above:

```
>>> fh = open('gashlycrumb.txt')
>>> lookup = { line[0].upper(): line.rstrip() for line in fh }
```

```
lookup = {}
for line in fh:
    lookup[line[0].upper()] = line.rstrip()
>>> lookup = { line[0].upper(): line.rstrip() for line in fh }
```

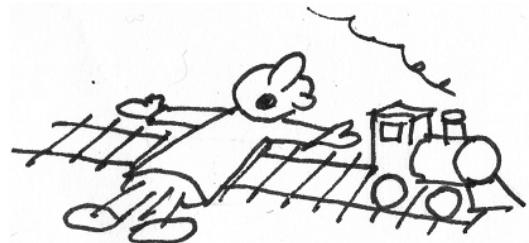
The diagram illustrates the equivalence between a standard for loop and a dictionary comprehension. It shows a bracket spanning the entire for loop, another bracket spanning the assignment part of the loop, and a third bracket spanning the entire dictionary comprehension. Arrows point from the start of the for loop to the start of the comprehension, from the assignment part to the value part of the comprehension, and from the end of the for loop to the end of the comprehension.

If you print the lookup again, you should see the same output as above. It may seem

like showing off to write one line of code instead of three, but it really does make a good deal of sense to write compact, idiomatic code. More code always means more chances for bugs, so I usually try to write code that is as simple as possible (but no simpler).

7.3.4 Dictionary lookups

Now that I have a lookup, I can ask if some value is in the keys. Note that I know the letters are in uppercase and I assume the user could give me lower, so I use `letter.upper()` to only compare that case:



```
>>> letter = 'a'
>>> letter.upper() in lookup
True
>>> lookup[letter.upper()]
'A is for Amy who fell down the stairs.'
```

If the letter is found, I can print the line of text for that letter; otherwise, I can print the message that I don't know that letter:

```
>>> letter = '4'
>>> if letter.upper() in lookup:
...     print(lookup[letter.upper()])
... else:
...     print('I do not know "{}".format(letter)')
...
I do not know "4".
```

An even shorter way to write that would use the `dict.get()` method:

```
def main():
    args = get_args()
    lookup = {line[0].upper(): line.rstrip() for line in args.file}

    for letter in args.letter:
        print(lookup.get(letter.upper(), f'I do not know "{letter}".')) ①
```

① If `lookup.get()` will return the value for `letter.upper()` or the phrase "I don't know..."

7.4 Review

- A dictionary comprehension is a way to build a dictionary in a one-line `for` loop.
- Defining file input arguments using `argparse.FileType` saves you time and code.
- Python's `pprint` module is used for "pretty printing" complex data structures.

7.5 Going Further

- Write a phonebook that reads a file and creates a dictionary from the names of

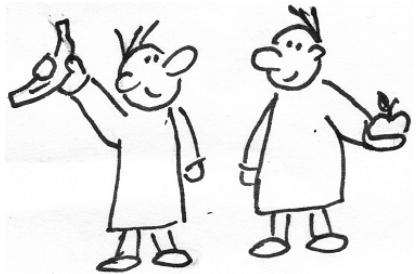
your friends to their email or phone numbers.

- Create a program that uses a dictionary to count the number of times you see each word in a document.
- Write an interactive version that takes input directly from the user. Use `while True` to set up an infinite loop and keep using `input` to get the user's next letter:

```
$ ./gashlycrumb_interactive.py
Please provide a letter [! to quit]: t
T is for Titus who flew into bits.
Please provide a letter [! to quit]: 7
I do not know "7".
Please provide a letter [! to quit]: !
Bye
```

Apples and Bananas: Find and replace

Have you ever misspelled a word? I haven't, but I've heard that many other people often do. We can use computers to find and replace all instances of a misspelled word with the correction. Or maybe you'd like to replace all mentions of your ex's name in your poetry with your new love's name? Find and replace is your friend.



To get us started, let us consider the children's song "Apples and Bananas" wherein we intone about our favorite fruits to consume:

```
I like to eat, eat, eat apples and bananas
```

Subsequent verses substitute the main vowel sound in the fruits for various other vowel sounds, such as the long "a" (as in "hay") sound:

```
I like to ate, ate, ate ay-ples and ba-nay-nays
```

Or the ever-popular long "e" (as in "knee"):

```
I like to eat, eat, eat ee-ples and bee-nee-nees
```

And so forth. In this exercise, we'll write a Python program called `apples.py` takes some text given as a single positional argument and replaces all the vowels in the text with a given `-v` or `--vowel` options (default a).

The program should handle text on the command line:

```
$ ./apples.py foo
faa
```

And accept the `-v` or `--vowel` option:

```
$ ./apples.py foo -v i
fii
```

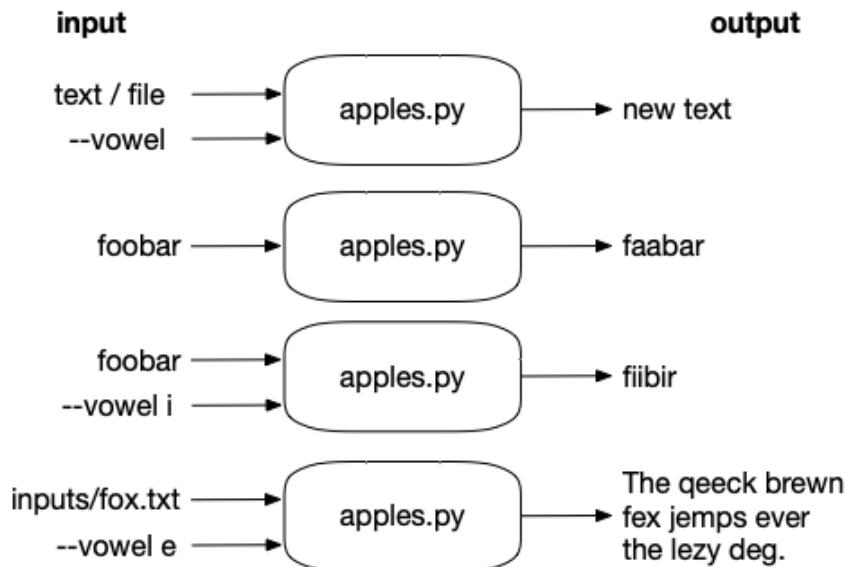
Your program should *preserve the case* of the input vowels:

```
$ ./apples.py -v i "APPLES AND BANANAS"
IPPLIS IND BININIS
```

As with the "Howler" program, the text argument may name a file in which case your program should read the contents of the file.

```
$ ./apples.py ../inputs/fox.txt
Tha qaack brawn fax jamps avar tha lazy dag.
$ ./apples.py --vowel e ../inputs/fox.txt
The qeeck brewn fex jemps ever the lezy deg.
```

It might help to look at a diagram of the program's inputs and output:



Here is the usage that should print when there are *no arguments*:

```
$ ./apples.py
usage: apples.py [-h] [-v vowel] text
apples.py: error: the following arguments are required: text
```

And the program should always print usage for the `-h` and `--help` flags:

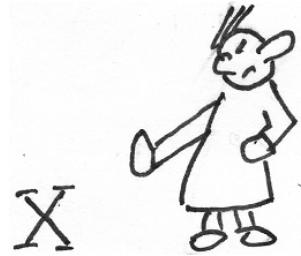
```
$ ./apples.py -h
usage: apples.py [-h] [-v vowel] text
```

```
Apples and bananas

positional arguments:
  text                  Input text or file

optional arguments:
  -h, --help            show this help message and exit
  -v vowel, --vowel vowel
                        The vowel to substitute (default: a)
```

The program should complain if the `--vowel` argument is not a single, lowercase vowel:



```
$ ./apples.py -v x foo
usage: apples.py [-h] [-v str] str
apples.py: error: argument -v/--vowel: \
invalid choice: 'x' (choose from 'a', 'e', 'i', 'o', 'u')
```

So our program is going to need to:

- Take a positional argument that might be some plain text or may name a file.
- If the argument is a file, use the contents as the input text.
- Take an optional `-v` or `--vowel` argument that should default to the letter "a".
- Verify that the `--vowel` option is in the set of vowels "a," "e," "i," "o," and "u."
- Replace all instances of vowels in the input text with the specified (or default) `--vowel` argument.
- Print the new text to STDOUT.

8.1 Altering strings

So far in our discussions of Python strings, numbers, lists, and dictionaries, we've seen how easily we can change or *mutate* variables. There is a problem, however, in that *strings are immutable*. Suppose we have a `text` variable that holds our input text:

```
>>> text = 'The quick brown fox jumps over the lazy dog.'
```

If we wanted to turn the first "e" (at index 2) into an "i," we cannot do this:

```
>>> text[2] = 'i'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

To change `text`, we need to set it equal to an entirely new value. In "Jump the Five" we saw that you can use a `for` loop to iterate over the characters in a string. For instance, I could laboriously uppercase the `text` like so:

```

new = ''          ①
for char in text: ②
    new += char.upper() ③

```

- ① Set a new variable equal to the empty string.
- ② Iterate through each character in the text.
- ③ Append the uppercase version of the character to the new variable.

We can inspect the `new` value to verify that it is all uppercase:

```

>>> new
'THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.'

```

Using this idea, you could iterate through the characters of `text` and build up a new string. Whenever the character is a vowel, you can change it for the given vowel, otherwise you use the character itself. We had to identify vowels in "Crow's Nest," so you can refer back to how you did that.

8.1.1 Using the `str.replace()` method

In "Jump The Five," we used the `str.replace()` method that might work. Let's look at the help:

```

>>> help(str.replace)
replace(self, old, new, count=-1, /)
    Return a copy with all occurrences of substring old replaced by new.

    count
        Maximum number of occurrences to replace.
        -1 (the default value) means replace all occurrences.

    If the optional argument count is given, only the first count occurrences are
    replaced.

```

Let's play with that in the REPL. I could replace "T" for "X". Can you see a way to replace all the vowels using this idea?

```

>>> text.replace('T', 'X')
'Xhe quick brown fox jumps over the lazy dog.'

```

Remember that this method never mutates the given string but instead returns a *new string* that you will need to assign to a variable.

8.1.2 Using `str.translate()`

We also looked at the `str.translate()` method. The documentation is a bit more cryptic.

```

>>> help(str.translate)
translate(self, table, /)
    Replace each character in the string using the given translation table.

    table
        Translation table, which must be a mapping of Unicode ordinals to
        Unicode ordinals, strings, or None.

```

```
The table must implement lookup/indexing via __getitem__, for instance a
dictionary or list. If this operation raises LookupError, the character is
left untouched. Characters mapped to None are deleted.
```

In "Jump The Five," we created a dict that associated the string '1' to the string '9' and so forth:

```
jumper = {'1': '9', '2': '8', '3': '7', '4': '6', '5': '0',
           '6': '4', '7': '3', '8': '2', '9': '1', '0': '5'}
```

That was the argument to the `str.maketrans()` function that makes a translation table suitable to use with `str.translate()` to change all the characters present in as keys in the dictionary to their corresponding values:

```
>>> '876-5309'.translate(str.maketrans(jumper))
'234-0751'
```

What should be the keys and values of the dict if you want to change all the vowels, both lower- and uppercase, to some other value?

8.1.3 Other ways to mutate strings

If you know about regular expressions, that's a strong solution. If you haven't heard of those, don't worry as I'll introduce them in the discussion. The point is for you to go *play* with this and come up with a solution. I found 8 ways to change all the vowels to a new character, so there are many ways you could approach this.

How many *different* methods can you find on your own before you look at my solution?

Hints:

- Consider using the `choices` option in the `argparse` documentation for how to constrain the `--vowel` options.
- Be sure to change both lower- and uppercase versions of the vowel, preserving the case of the input characters.

Now is the time to dig in and see what you can do before you look at my solutions.

8.2 Solution

```
#!/usr/bin/env python3
"""Apples and Bananas"""

import argparse
import os

# -----
def get_args():
    """get command-line arguments"""

    parser = argparse.ArgumentParser(
        description='Apples and bananas',
```

```

formatter_class=argparse.ArgumentDefaultsHelpFormatter)

parser.add_argument('text', metavar='text', help='Input text or file') ①

parser.add_argument('-v',
                    '--vowel',
                    help='The vowel(s) allowed',
                    metavar='vowel',
                    type=str,
                    default='a',
                    choices=list('aeiou')) ②

args = parser.parse_args()

if os.path.isfile(args.text): ③
    args.text = open(args.text).read().rstrip() ④

return args

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

    args = get_args()
    text = args.text
    vowel = args.vowel
    new_text = [] ⑤

    for char in text:
        if char in 'aeiou': ⑥
            new_text.append(vowel) ⑦
        elif char in 'AEIOU': ⑧
            new_text.append(vowel.upper()) ⑨
        else: ⑩
            new_text.append(char) ⑪

    print(''.join(new_text)) ⑫

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

```

- ① The input might be text or a file name, so define as a string.
- ② Use the choices to restrict the user to one of the listed vowels.
- ③ Check if the text argument is a file.
- ④ If it is, read the file, using the str.rstrip() to remove any trailing whitespace.
- ⑤ Create a new list to hold the characters we'll select.
- ⑥ Iterate through each character of the text.
- ⑦ See if the current character is in the list of lowercase vowels.
- ⑧ If it is, use the vowel instead of the character.
- ⑨ See if the current character is in the list of uppercase vowels.
- ⑩ If it is, use the value of vowel.upper() instead of the character.
- ⑪ Otherwise, take the character itself.
- ⑫ Print a new string made by joining the new_text list on the empty string.

8.3 Discussion

I came up with eight ways to write my solution. All of them have the same `get_args()`, so let's look at that first.

8.3.1 Defining the parameters

This is one of those problems that has many valid and interesting solutions. The first problem to solve is, of course, getting and validating the user's input. As always, I will use `argparse`. I usually define all my required parameters first. The `text` parameter is a positional string that *might* be a file name:

```
parser.add_argument('text', metavar='str', help='Input text or file')
```

The `--vowel` option is also a string, and I decided to use the `choices` option to have `argparse` validate that the user's input is in the `list('aeiou')`:

```
parser.add_argument('-v',
                   '--vowel',
                   help='The vowel to substitute',
                   metavar='str',
                   type=str,
                   default='a',
                   choices=list('aeiou'))
```

That is, `choices` wants a list of options. I could pass in `['a', 'e', 'i', 'o', 'u']`, but that's a lot of typing on my part. It's much easier to type `list('aeiou')` to have Python turn the str "aeiou" into a list of the characters. Both of these are the same because `list(str)` creates a list of the individual characters in a given string. And remember, the use of single or double quotes doesn't matter. Any value enclosed in either type of quote is a `str`, even if it's just one character:

```
>>> ['a', 'e', 'i', 'o', 'u']
['a', 'e', 'i', 'o', 'u']
>>> list('aeiou')
['a', 'e', 'i', 'o', 'u']
```

We can even write a test for this. The absence of any error means that it's OK:

```
>>> assert ['a', 'e', 'i', 'o', 'u'] == list('aeiou')
```

The next task is detecting if `text` is the name of a file that should be read for the text or is the text itself. This is the same code I used in "Howler," and again I choose to handle the `text` argument inside the `get_args()` function so that, by the time I get `text` inside my `main()`, it's all be handled:

```
if os.path.isfile(args.text):
    args.text = open(args.text).read().rstrip()
```



At this point, the user's arguments to the program have been fully vetted. We've got `text` either from the command line or from a file, and we've verified that the `--vowel` is actually one of the allowed characters. To me, this is a single "unit" where I've handled the arguments, and processing can now go forward by returning the arguments:

```
return args
```

8.4 Eight ways to replace the vowels

How many ways did you find to replace the vowels? You only needed one, of course, to pass the test, but I hope you probed the edges of the language to see how many different techniques there are. I know that the Zen of Python says:

There should be one — and preferably only one — obvious way to do it. - www.python.org/dev/peps/pep-0020/

But I really come from the Perl mentality that "There Is More Than One Way To Do It" (TIMTOWTDI or "Tim Toady").

8.4.1 Method 1: Iterate every character

The first method is similar to "Jump The Five" where we can use a `for` loop on a string to access each character. Here is code you can copy and paste into the ipython REPL

```

>>> text = 'Apples and Bananas!'          ①
>>> vowel = 'o'                          ②
>>> new_text = []                        ③
>>> for char in text:                  ④
...     if char in 'aeiou':              ⑤
...         new_text.append(vowel)        ⑥
...     elif char in 'AEIOU':            ⑦
...         new_text.append(vowel.upper()) ⑧
...     else:                            ⑨
...         new_text.append(char)
...
>>> text = ''.join(new_text)           ⑩
>>> text
'Opplos ond Bononos!'

```

- ① Set a `text` variable to the string "Apples and Bananas!"

- ② Set the vowel variable to the string "o."
- ③ Set the variable new_text to an empty list.
- ④ Use a for to iterate text, putting each character into the char variable.
- ⑤ If the character is in the set of lowercase vowels,
- ⑥ Substitute in the vowel to the new_text.
- ⑦ If the character is in the set of uppercase vowels,
- ⑧ Substitute the vowel.upper() version into new_text.
- ⑨ Otherwise, use the char as-is.
- ⑩ Turn the list called new_text into new str by joining it on the empty string ('').

Note that it would be just fine to start off making new_text an empty string and then concatenating the new characters. Then you wouldn't have to str.join() them at the end. Whatever you prefer:

```
new_text += vowel
```

8.4.2 Method 2: Using the str.replace() method

I'm going to show you several alternate solutions. They're all functionally equivalent because they all pass the tests, but the point here is to explore the Python language and understand it!

For the alternate solutions, I'll just show the main() function. Here is a way to solve the problem using the str.replace() method:

```
def main():
    args = get_args()
    text = args.text
    vowel = args.vowel

    for v in 'aeiou': ①
        text = text.replace(v, vowel).replace(v.upper(), vowel.upper()) ②

    print(text)
```

- ① Iterate through the list of vowels. We don't have to say list('aeiou') here because Python will automatically treat the string 'aeiou' like a list because we are using it in a *list context* with the for.
- ② Use the str.replace() method twice to replace both the lower- and upper-case versions of the vowel in the text.

I mentioned in the introduction the str.replace() method that will return a new string with all instances of one string replaced by another.

```
>>> s = 'foo'
>>> s.replace('o', 'a')
'faa'
>>> s.replace('oo', 'x')
'fx'
```

Note that the original string remains unchanged:

```
>>> s
'foo'
```

You don't have to chain the two `str.replace()` methods. It could be written as two separate statements:

```
text = text.replace(v, vowel).replace(v.upper(), vowel.upper())
      ↓
text = text.replace(v, vowel)
text = text.replace(v.upper(), vowel.upper())
```

8.4.3 Method 3: Using the `str.translate()` method

```
def main():
    args = get_args()
    vowel = args.vowel
    trans = str.maketrans('aeiouAEIOU', vowel * 5 + vowel.upper() * 5) ①
    text = args.text.translate(trans) ②

    print(text)
```

- ① Create the translation table.
- ② Call the `str.translate()` method on the `text` variable passing the `trans` table as the argument.

How can we use `str.translate()` method to solve this? I showed in "Jump The Five" how the jumper dictionary could be used to create a translation table using the `str.maketrans()` method to convert each number to another number. In this problem I need to change all the lower- and upper-case vowels (10 total) to some given vowel. For instance, to make all the vowels into the letter "o," I could create translation table `t` like so:

```
t = {'a': 'o',
      'e': 'o',
      'i': 'o',
      'o': 'o',
      'u': 'o',
      'A': 'O',
      'E': 'O',
      'I': 'O',
      'O': 'O',
      'U': 'O'}
```

I can use `t` with the `str.translate()` method:

```
>>> 'Apples and Bananas'.translate(str.maketrans(t))
'Opplos ond Bononos'
```

If you read the documentation for `str.maketrans()`, you will find that another way to make the translation is to supply two strings of equal lengths:

```
maketrans(x, y=None, z=None, /)
```

```
Return a translation table usable for str.translate().
```

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`. If there is a third argument, it must be a string, whose characters will be mapped to `None` in the result.

The first string should contain the letters I want to replace which are the lower- and uppercase vowels 'aeiouAEIOU'. The second string is composed of the letters to use for substitution. I would like to use 'ooooo' for the 'aeiou' and '00000' for 'AEIOU'. I can repeat my vowel five times using the * operator that you normally associate with numeric multiplication. This is (sort of) "multiplying" a string, so, OK, I guess:

```
>>> vowel * 5
'ooooo'
```

Now to handle the uppercase version, too:

```
>>> vowel * 5 + vowel.upper() * 5
'ooooo00000'
```

Now to make the translation table:

```
>>> trans = str.maketrans('aeiouAEIOU', vowel * 5 + vowel.upper() * 5)
```

Let's inspect the `trans` table. I want to "pretty print" the data structure so I can see it, so I will use the `pprint.pprint` (pretty print) function:

```
>>> from pprint import pprint as pp
>>> pp(trans)
{65: 79,
 69: 79,
 73: 79,
 79: 79,
 85: 79,
 97: 111,
 101: 111,
 105: 111,
 111: 111,
 117: 111}
```

The enclosing curly braces {} tell us that `trans` is a dict. Each character is represented by its *ordinal* value, which is the character's position in the ASCII table (www.asciitable.com/). (You will use this later in the "Gematria" program.) You can go back and forth from characters and their ordinal values by using the `chr()` and `ord()` functions. Here are the `ord()` values for the vowels:

```
>>> for char in 'aeiou':
...     print(char, ord(char))
...
a 97
e 101
i 105
o 111
```

```
u 117
```

And here you can create the same output but starting with the `ord()` values to get the `chr()` values:

```
>>> for num in [97, 101, 105, 111, 117]:
...     print(chr(num), num)
...
a 97
e 101
i 105
o 111
u 117
>>>
```

If you'd like to inspect all the ordinal values for all the printable characters, you can run this:

```
>>> import string
>>> for char in string.printable:
...     print(char, ord(char))
```

I don't include the output because there are 100 printable characters:

```
>>> print(len(string.printable))
100
```

So the `trans` table is a mapping. The lowercase vowels ("aeiou") all map to the ordinal value 111 which is "o." The uppercase vowels ("AEIOU") map to 79 which is "O." I can use the `dict.items()` method to iterate over the key/value pairs of `trans` to verify this is the case:

```
>>> for x, y in trans.items():
...     print(f'{chr(x)} => {chr(y)}')
...
a => o
e => o
i => o
o => o
u => o
A => O
E => O
I => O
O => O
U => O
```

The original text will be unchanged by the `str.translate()` method, so we overwrite `text` with the new version:

```
>>> text = 'Apples and Bananas!' ①
>>> trans = str.maketrans('aeiouAEIOU', vowel * 5 + vowel.upper() * 5) ②
>>> text = text.translate(trans) ③
>>> text
'Opplos ond Bononos!'
```

- ① Initialize `text`.
- ② Make a translation table.

- ③ Use the translation table as the argument to `text.translate()`. Overwrite the original value of `text` with the result.

That was a lot of explanation about `ord()` and `chr()` and dictionaries and such, but look how simple and elegant that solution is! This is much shorter than Method 1. Fewer lines of code (LOC) means fewer opportunities for bugs!

8.4.4 Method 4: List comprehension

```
def main():
    args = get_args()
    vowel = args.vowel
    text = [ ①
        vowel if c in 'aeiou' else vowel.upper() if c in 'AEIOU' else c ②
        for c in args.text
    ]
    print(''.join(text)) ③
```

- ① Use a list comprehension to process all the characters in `args.text` to create a new list called `text`.
- ② Use a compound `if` expression to handle three cases (lowercase vowel, uppercase vowel, the default).
- ③ Print a new string by joining `text` on the empty string.

Following up on Method 1, we can use a "list comprehension" to significantly shorten the `for` loop. In Gashlycrumb we looked at a "dictionary comprehension" as a one-line method to create a new dictionary using a `for` loop. Here we can do the same, creating a new list.

For example, let's generate a list of the squared values of the numbers 1 through 4. We can use the `range()` function to get the numbers from a starting number to an ending number (not inclusive!). `range()` is a *lazy* function, which means it won't actually produce values until your program actually needs them. That is, a lazy function is a promise to do something. If your program branches in such a way that you never need to produce the values, then the work is never done, meaning your code is more efficient!

In the REPL, I must use the `list()` function to force the production of the values, but most of the time your code doesn't need to do this:

```
>>> list(range(1, 5))
[1, 2, 3, 4]
```

I can write a `for` loop to `print()` the squares:

```
>>> for num in range(1, 5):
...     print(num ** 2)
...
1
4
9
16
```

But what I really want is a `list` that contains those values. A simple way to do this is to create an empty `list` to which we will `list.append()` those values:

```
>>> squares = []
>>> for num in range(1, 5):
...     squares.append(num ** 2)
```

And now I can verify that I have my squares:

```
>>> assert len(squares) == 4
>>> assert squares == [1, 4, 9, 16]
```



A list comprehension is novel from a `for` loop in that it *returns a new list*:

```
>>> [num ** 2 for num in range(1, 5)]
[1, 4, 9, 16]
```

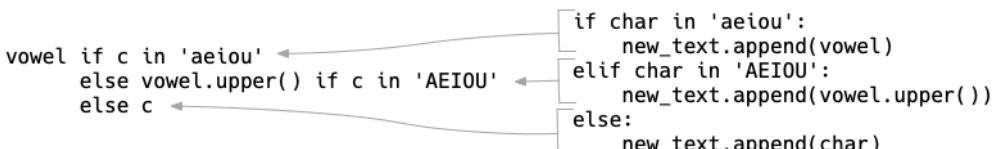
I can assign this to the `squares` variable and verify that I still have what I expected. Ask yourself which version of the code you'd rather maintain, the longer one with the `for` loop or the shorter one with the list comprehension?

```
>>> squares = [num ** 2 for num in range(1, 5)]
>>> assert len(squares) == 4
>>> assert squares == [1, 4, 9, 16]
```

For our program, we're going to condense the `if/elif/else` logic from Method 1 into a compound `if` expression. First let's see how we could shorten the `for` loop version:

```
>>> text = 'Apples and Bananas!'
>>> new_text = []
>>> for c in text:
...     new_text.append(vowel if c in 'aeiou' else vowel.upper() if c in 'AEIOU' else c)
...
>>> ''.join(new_text)
'Opplos ond Bononos!'
```

Here's a diagram that shows how the parts of the expression match up to the original `if/elif/else`:



And now to turn that into a list comprehension:

```
>>> text = 'Apples and Bananas!'
>>> new_text = [
...     vowel if c in 'aeiou' else vowel.upper() if c in 'AEIOU' else c ①
```

```
...     for c in text ] ②
...
>>> ''.join(new_text)
'Opplos ond Bononos!'
```

- ① Do this
- ② For these things.

8.4.5 Method 5: List comprehension with function

The compound if expression inside the list comprehension borders is complicated enough that it probably should be a function. We can *define* a new function with the def statement and call it new_char(). It accepts a character we'll call c. After that, it's the same compound if expression as before:

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

    def new_char(c): ①
        return vowel if c in 'aeiou' else vowel.upper() if c in 'AEIOU' else c ②

    text = ''.join([new_char(c) for c in args.text]) ③

    print(text)
```

- ① Define a function to choose a new character. Note that it uses the vowel variable because the function has been declared in the same scope. This is called a "closure" because new_char() *closes over* the variable.
- ② Use the compound if expression to select the correct character.
- ③ Use a list comprehension to process all the characters in text.

You can play with the new_char() function by putting this into your REPL:

```
vowel = 'o'
def new_char(c):
    return vowel if c in 'aeiou' else vowel.upper() if c in 'AEIOU' else c
```

It should always return the letter "o" if the argument is a lowercase vowel:

```
>>> new_char('a')
'o'
```

And "O" if the argument is an uppercase vowel:

```
>>> new_char('A')
'O'
```

Otherwise it should return the given character:

```
>>> new_char('b')
'b'
```

We can use the new_char() function to process all the characters in text using a list

comprehension:

```
>>> text = 'Apples and Bananas!'
>>> text = ''.join([new_char(c) for c in text])
>>> text
'Opplos ond Bononos!'
```

A note about the fact that the `new_char()` function is declared *inside* the `main()` function. Yes, you can do that! The function is then only "visible" inside the `main()` function. The reason I do this is because I want to reference the `vowel` variable inside the function without actually passing it as an argument.

As an example, I define a `foo()` function that has a `bar()` function inside it. I can call `foo()` and it will call `bar()`, but from outside of `foo()` the `bar()` function does not exist ("is not visible" or "is not in scope"):

```
>>> def foo():
...     def bar():
...         print('This is bar')
...     bar()
...
>>> foo()
This is bar
>>> bar()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'bar' is not defined
```

I did this because I actually created a special type of function with `new_char()` called a "closure" because it is "closing" around the `vowel`. That is, I wanted to use reference the `vowel` variable inside the `new_char()` function without explicitly passing it as an argument. If I declare the function in the same scope as `vowel`, I can do this.

If I had defined `new_char()` outside of `main()`, the `vowel` would not be visible to `new_char()` because it only exists inside the `main()` function. If we draw a box around the `main()` function, that is where the `new_char()` function can be seen. Outside of that, the function is consider undefined.

Figure 8.1. Visibility of the new_char() function is limited just to the main() function. Code outside of main() cannot see or call new_char(). This also makes it difficult to unit test the function!

```
def main():
    args = get_args()
    text = args.text
    vowel = args.vowel
    new_char( ) and vowel only visible inside box
    def new_char(c):
        → return vowel if c in 'aeiou' else vowel.upper() if c in 'AEIOU' else c
    text = ''.join([new_char(c) for c in text])
    print(text)
```

new_char()
closes over
vowel



new_char and vowel not visible here

There are many reasons why to write a function like this, even if it has just one line. I think of functions as *units* of code that describe some concept — ideally just *one* per function! We can formalize our understand of our functions with assertions:

```
>>> assert all([new_char(v) == 'o' for v in 'AEIOU'])
>>> assert all([new_char(v) == 'o' for v in 'aeiou'])
```

In two lines of code, I've just tested all 10 vowels by using list comprehensions plus the `all()` function. Let's take a moment to understand `all()` and `any()`. If you read `help(all)`, you'll see "Return True if `bool(x)` is True for all values `x` in the iterable." So all the values need to the True for the entire expression to be True:

```
>>> all([True, True, True])
True
```

It's the same as putting `and` in between all the values:

```
>>> True and True and True
True
```

If any value is `False`, then the whole expression is `False`:

```
>>> all([True, False, True])
False
```

The `any()` function returns `True` if *any* of the values are `True`:

```
>>> any([True, False, True])
True
```

It's the same as putting or in between all the values⁷:

```
>>> True or False or True
True
```

And False if there are no True values:

```
>>> any([False, False, False])
False
```

We can check the values manually:

```
>>> [new_char(v) == 'O' for v in 'AEIOU']
[True, True, True, True, True]
```

So then all() should be True for the entire expression:

```
>>> all([new_char(v) == 'O' for v in 'AEIOU'])
True
```

We can move these into formal `test_` functions that can be run with `pytest`. This is the idea of *unit testing* where I personally think of functions as units. Some people have other ideas about what is the best *unit* to test. I recommend you read further about testing to understand the range of opinions.

8.4.6 Method 6: The `map()` function

For this next method, I want to introduce the `map()` function as it's quite similar to a list comprehension. The `map()` function accepts two arguments:

1. A function
2. An iterable like a list, lazy function, or generator.



I like to think of `map()` like a paint booth — you load up the booth with, say, blue paint. Unpainted cars go in, blue paint is applied, and blue cars come out. I can create a function to "paint" my cars by adding the string "blue" to the beginning:

```
>>> list(map(lambda car: 'blue ' + car, ['BMW', 'Alfa Romeo', 'Chrysler']))
['blue BMW', 'blue Alfa Romeo', 'blue Chrysler']
```

⁷ Here is a lesson in logic brought to you by my youngest who used to try to avoid flossing his teeth before bed. When he would come out of the bathroom, I would ask "Did you brush and floss your teeth?" He would invariably reply "Yes." I would then ask "Did you brush your teeth?" "Yes," he would say. "Did you floss your teeth?" I would ask. "No," he would usually say. So, he was using `or` while I was using `and`.

The first argument you see here starts with the keyword `lambda` which is used to create an *anonymous* function. That is, with the regular `def` keyword, the function name follows. With `lambda`, there is no name, only the list of parameters and the function body.



Think about regular named functions like `add1()` that adds 1 to a value:

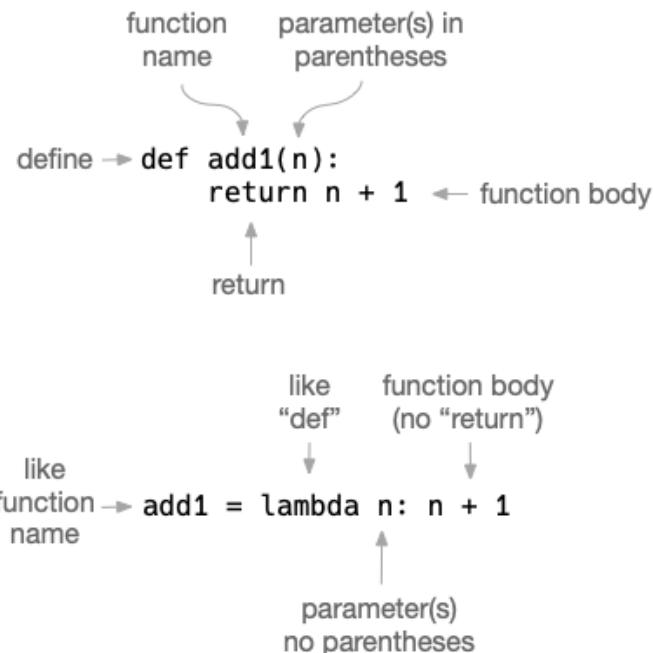
```
def add1(n):
    return n + 1
```

It works as expected:

```
>>> assert add1(10) == 11
>>> assert add1(add1(10)) == 12
```

Now compare to the `lambda` version. We can assign it to the variable `add1` which then kinda sorta acts like the function's name:

```
>>> add1 = lambda n: n + 1
```



It works exactly the same as using `def` to define a function the normal way:

```
>>> assert add1(10) == 11
>>> assert add1(add1(10)) == 12
```

The function body for a `lambda` pretty much needs to fit on one line, and they don't have `return` at the end. In both versions, the argument to the function is `n`. In the usual `def add(n)`, the argument is defined in the parentheses just after the function name. In the `lambda n` version, there is no function name and no parentheses around the function's parameter, `n`.

There is no difference in how you can use them. They are both functions:

```
>>> type(lambda x: x)
<class 'function'>
```

If you are comfortable with using `add1()` in a list comprehension:

```
>>> [add1(n) for n in [1, 2, 3]]
[2, 3, 4]
```

Then it's a short step to using the `map()` function. Note that `map()` is a lazy function like the `range()` function we looked earlier. It won't actually create the values until you actually need them as opposed to a list comprehension which will produce the resulting list immediately. I don't personally tend to worry about the performance of the code as much as I do the readability. When I write code for myself, I prefer to use `map()`, but you should write code that makes the most sense for you and your teammates.

To force the results from `map()` in the REPL, I need to use the `list()` function:

```
>>> list(map(add1, [1, 2, 3]))
[2, 3, 4]
```

We can write the list comprehension with the `add1()` code in-line:

```
>>> [n + 1 for n in [1, 2, 3]]
[2, 3, 4]
```

Which looks very similar to the `lambda` code:

```
>>> list(map(lambda n: n + 1, [1, 2, 3]))      map(lambda n: n + 1, [1, 2, 3])
[2, 3, 4]                                     [2, 3, 4]
```

So here is how I could use a `map()`:

```
def main():
    args = get_args()
    vowel = args.vowel
    text = map(
        lambda c: vowel if c in 'aeiou' else vowel.upper() ①
        if c in 'AEIOU' else c, args.text) ②
    ③
    print(''.join(text)) ④
```

① The `map()` function wants a function for the first argument and a list for the second.

- ② Use lambda to create an *anonymous* function that accepts character, c.
- ③ args.text is the second argument to map(). Because map() expects this argument to be a list, it will automatically coerce it to a list (which is what we want).
- ④ The map() returned a new list into the text variable, so we join it on the empty string to print it.

Higher-order functions

The map() function is called a "higher-order function" (HOF) because it takes *another function* as an argument, which is wicked cool. Later we'll use another HOF called filter().

8.4.7 Method 7: Using map() with a defined function

We are not required to use a lambda expression with map(). Any function at all will work, so let's go back to using our new_char() function:

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

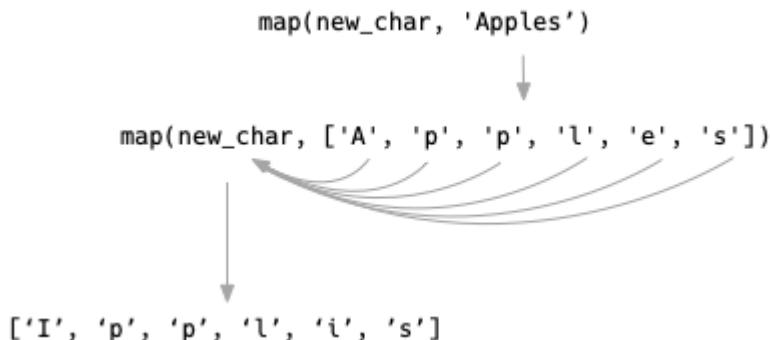
    def new_char(c): ①
        return vowel if c in 'aeiou' else vowel.upper() if c in 'AEIOU' else c

    print(''.join(map(new_char, args.text))) ②
```

- ① Define a function that will return the proper character.
- ② Use map() to apply new_char() to all the characters in text. The result is a list of characters which we can use str.join() to create a new string for print().

Notice that map() uses new_char *without parentheses* as the first argument. If you added the parens, you'd be *calling* the function and would see this error:

```
>>> text = ''.join(map(new_char(), text))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: new_char() missing 1 required positional argument: 'c'
```



map() takes each character from text and passes it as the argument to the new_char() function which decides whether to return the vowel or the original

character. The result of mapping these characters is a new list of characters that we `str.join()` on the empty string to create a new version of `text`.

8.4.8 Method 8: Using regular expressions

A "regular expression" is a term from linguistics and computer science. It is a way of describing a "regular language"⁸. Essentially regular expressions give us a way to *describe patterns of text*, which is truly, outstandingly useful.

To use regular expressions, you must `import re` in your code. Here I will show how we can use the "substitute" function `re.sub()` to find any of the lower- and uppercase vowels and replace them with the given vowel:

```
def main():
    args = get_args()
    text = args.text
    vowel = args.vowel
    text = re.sub('[aeiou]', vowel, text) ①
    text = re.sub('[AEIOU]', vowel.upper(), text) ②
    print(text)
```

- ① Substitute any of the lowercase vowels for the given vowel (which is lowercase because of the restrictions in `get_args()`).
- ② Substitute any of the uppercase vowels for the uppercased vowel.

Regular expressions (also called "regexes") are a separate domain-specific language (DSL). That is, they really have nothing whatsoever to do with Python. They have their own syntax and rules, and they are used in many places from command-line tools to databases. Regexes are incredibly powerful and well worth the effort to learn them.

The `re.sub()` function will *substitute* all instances of text matching a given pattern for a new string. The square brackets around the vowels `'[aeiou]'` create a "character class," meaning anything matching one of the characters listed inside the brackets. The second argument is the string that will replace the found strings — here our `vowel` provided by the user. The third argument is the string we want to change, which is the `text` from the user.

```
>>> import re
>>> text = 'Apples and Bananas!'
>>> vowel = 'o'
>>> re.sub('[aeiou]', vowel, text)
'Applos ond Bononos!'
```

Note that `re.sub()` returns a new string, and the original `text` remains unchanged by the operation:

```
>>> text
'Apples and Bananas!'
```

That almost worked, but it missed the uppercase vowel "A". I could overwrite

⁸ en.wikipedia.org/wiki/Regular_language

the text in two steps to get both lower- and uppercase:

```
>>> text = re.sub('[aeiou]', vowel, text)
>>> text = re.sub('[AEIOU]', vowel.upper(), text)
>>> text
'Opplos ond Bononos!'
```

Or do it in one step just like the `str.replace()` method shown earlier:

```
>>> text = 'Apples and Bananas!'
>>> text = re.sub('[AEIOU]', vowel.upper(), re.sub('[aeiou]', vowel, text))
>>> text
'Opplos ond Bononos!'
```

One of the biggest differences with this solution to all the others is how we use regular expressions to describe what we were looking for and didn't have to write the code to actually find the characters! This is more along the lines of *declarative* programming. We declare what we want, and the computer does the grunt work!

8.5 Refactoring with tests

There are many ways to solve this problem. The most important step is to get your program to work properly. Tests let you know when you've reached that point. From there, you can explore other ways to solve the problem and keep using the tests to ensure you still have a correct program. Tests actually provide great freedom to be creative. Always be thinking about tests you can write for your own programs so that, when you change them later, they will always keep working!

8.6 Review

I showed many ways to solve this seemingly trivial problem. Some of the techniques using higher-order functions and regular expression are quite advanced techniques. It might seem like driving a finishing nail with a sledgehammer, but I want to start introducing you to programming ideas that I'll visit again and again in later chapters.

If you only really understood the first few solutions, that's fine! Just stick with me. The more times you see these ideas applied in different contexts, the more they will begin to make sense.

- You can use `argparse` to limit an argument's values to a *list of choices* that you define.
- Strings cannot be directly modified, but the `str.replace()` and `str.translate()` methods can create a *new, modified string* from an existing string.
- A `for` loop on a string will iterate the characters of the string.
- A list comprehension is a short-hand way to write a `for` loop inside `[]` to create a new *list*.
- Functions can be defined inside other functions. Their visibility is then limited to the enclosing function. * Function can reference variables declared within the same scope creating a closure.

- The `map()` function is similar to a list comprehension. It will create a new, modified list by applying some function to every member of a given list. The original list will not be changed.
- Regular expressions provide a syntax for describing patterns of text with the `re` module. The `re.sub()` method will substitute found patterns with new text. The original text will be unchanged.

8.7 Going Further

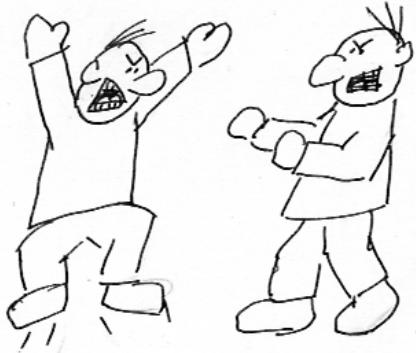
- Write a version that collapses multiple adjacent vowels into a single substituted value. For example, "quick" should become "qack" and not "qaack."

Dial-A-Curse: Generating random insults from lists of words

"He or she is a slimy-sided, frog-mouthed, silt-eating slug with the brains of a turtle."

— Dial-A-Curse

Random events are at the heart of interesting games and puzzles. Humans quickly grow bored of things that are always the same, so let's learn how to make our programs more interesting by having them behave differently each time they are run. This exercise will introduce how to randomly select one or more elements from lists of options. To explore randomness, we'll create a program called `abuse.py` that will insult the user by randomly selecting adjectives and nouns to create slanderous epithets.



In order to test randomness, though, we need to control it. It turns out that "random" events on computers are rarely actually random but only "pseudo-random," which means we can control them using a "seed."⁹ Each time you use the same seed, you get the same "random" choices!

Shakespeare had some of the best insults, so we'll draw from the vocabulary of his

⁹ "The generation of random numbers is too important to be left to chance." — Robert R. Coveyou

works. Here is the list of adjectives you should use:

bankrupt base caterwauling corrupt cullionly detestable dishonest false filthsome filthy foolish foul gross heedless indistinguishable infected insatiate irksome lascivious lecherous loathsome lubbery old peevish rascaly rotten ruinous scurilous scurvy slanderous sodden-witted thin-faced toad-spotted unmannered vile wall-eyed

And these are the nouns:

Judas Satan ape ass barbemonger beggar block boy braggart butt carbuncle coward coxcomb cur dandy degenerate fiend fishmonger fool gull harpy jack jolthead knave liar lunatic maw milksop minion ratcatcher recreant rogue scold slave swine traitor varlet villain worm

For instance, it might produce the following:

```
$ ./abuse.py
You slanderous, rotten block!
You lubbery, scurilous ratcatcher!
You rotten, foul liar!
```

In this exercise, you will learn to:

- Use `parser.error()` from `argparse` to throw errors
- Learn about random seeds to control randomness
- Take random choices and samples from Python lists
- Iterate an algorithm a specified number of times with a `for` loop
- Format output strings

9.1 Writing abuse.py

The arguments to this program are options that have default values, meaning it can run with no arguments at all. The `-n` or `--number` option will default to 3 and will determine how many insults are created:

```
$ ./abuse.py --number 2
You filthsome, cullionly fiend!
You false, thin-faced minion!
```

And the `-a` or `--adjectives` option should default to 2 and will determine how many adjectives are used in each insult:

```
$ ./abuse.py --adjectives 3
You caterwauling, heedless, gross coxcomb!
You sodden-witted, rascaly, lascivious varlet!
You dishonest, lecherous, foolish varlet!
```

Lastly, your program should accept a `-s` or `--seed` argument (default `None`) that will control the randomness of the program. The following should be exactly reproducible, no matter who runs the program on any machine at any time:

```
$ ./abuse.py --seed 1
You filthsome, cullionly fiend!
```

```
You false, thin-faced minion!
You sodden-witted, rascaly cur!
```

When run with no arguments, the program should generate insults using the defaults:

```
$ ./abuse.py
You foul, false varlet!
You filthy, insatiate fool!
You lascivious, corrupt recreant!
```



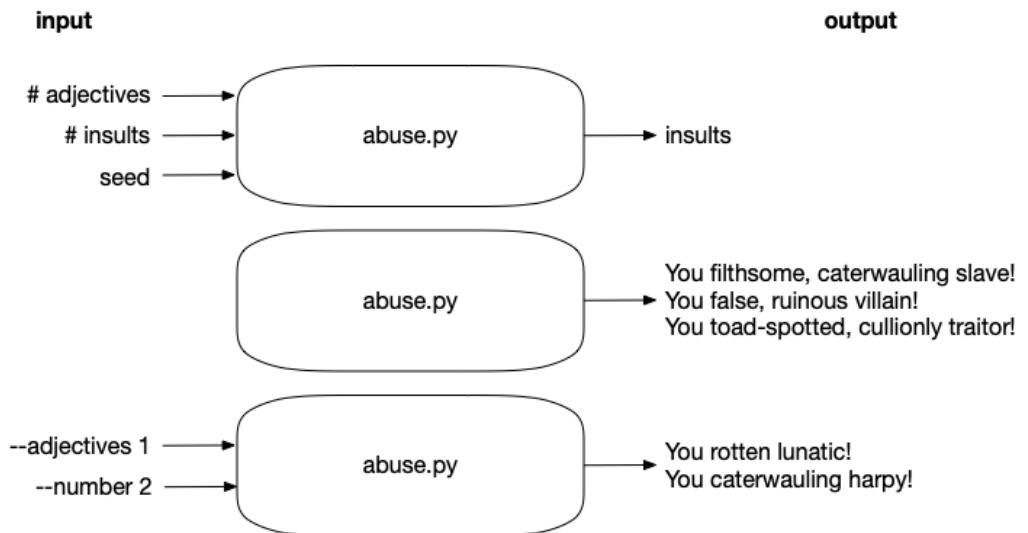
I recommend you start by copying the template/template.py to abuse/abuse.py or by using new.py to create the abuse.py program in the abuse directory of your repository. The next step is to make your program produce the following usage for -h or --help:

```
$ ./abuse.py -h
usage: abuse.py [-h] [-a adjectives] [-n insults] [-s seed]

Heap abuse

optional arguments:
  -h, --help            show this help message and exit
  -a adjectives, --adjectives adjectives
                        Number of adjectives (default: 2)
  -n insults, --number insults
                        Number of insults (default: 3)
  -s seed, --seed seed  Random seed (default: None)
```

Here is a string diagram to help you see the program:



9.1.1 Validating arguments

All of the options for number of insults and adjectives as well as the random seed should all be `int` values. If you define each using `type=int` (remember there are no quotes around the `int`), then `argparse` will handle the validation and conversion of the arguments to an `int` value for you. That is, just by defining `type=int`, the following error will be generated for you:

```
$ ./abuse.py -n foo
usage: abuse.py [-h] [-a adjectives] [-n insults] [-s seed]
abuse.py: error: argument -n/--number: invalid int value: 'foo'
```

Additionally, if either `--number` or `--adjectives` less than 1, your program should exit with an error code and message:

```
$ ./abuse.py -a -4
usage: abuse.py [-h] [-a adjectives] [-n insults] [-s seed]
abuse.py: error: --adjectives "-4" must be > 0
$ ./abuse.py -n -4
usage: abuse.py [-h] [-a adjectives] [-n insults] [-s seed]
abuse.py: error: --number "-4" must be > 0
```

As you start to write your own programs and tests, I recommend you steal from the tests I've written.¹⁰ Let's take a look at one of the tests in `test.py` to see how the program is tested:

```
def test_bad_adjective_num():
    """bad_adjectives"""

    n = random.choice(range(-10, 0))          ①
    rv, out = getstatusoutput(f'{prog} -a {n}')  ②
    assert rv != 0                            ③
    assert re.search(f"--adjectives '{n}' must be > 0", out) ④
```

- ① The name of the function must start with `test_` in order for `pytest` to find and run it.
- ② Use the `random.choice()` function to randomly select a value from the `range()` of numbers from -10 to 0. We will use this same function in our program, so note here how it is called!
- ③ Run the program using the `getstatusoutput()` from the `subprocess` module using a bad `-a` value. This function returns the exit value (which I put into `rv` for "return value") and standard out (`out`).
- ④ Assert that the return value (`rv`) is not 0 where "0" would indicate success (or "zero errors").
- ⑤ Assert that the output somewhere contains the statement that the `--adjectives` argument must be greater than 0.

Next we'll copy the ideas from "Manually checking arguments" in the `argparse` appendix. There I introduce the `parser.error()` function that you can call inside the `get_args()` function to do the following:

1. Print the short usage statement.
2. Print an error message to the user.
3. Stop execution of the program.

¹⁰ "Good composers borrow. Great ones steal." — Igor Stravinsky

4. Exit with a non-zero exit value to indicate an error.

That is, your `get_args()` normally finishes with:

```
return args.parse_args()
```

Instead, put the `args` into a variable and check the `args.adjectives` value to see if it's less than 1. If it is, call `parser.error()` with an error message to report to the user:

```
args = parser.parse_args()

if args.adjectives < 1:
    parser.error(f"--adjectives '{args.adjectives}' must be > 0')
```

Also do this for the `args.number`. If they are both fine, then you can return the arguments to the calling function:

```
return args
```

9.1.2 Importing and seeding the random module

Once you have defined and validated all the program's arguments, you are ready to heap scorn upon the user. We need to add `import random` to our program so we can use functions from that module to select adjectives and nouns. It's best practice to list all your `import` statements, one module at a time, at the top of your program.

As usual first thing we need to do in the `main()` is to call `get_args()` to get our arguments, and the very next step is to pass the `args.seed` value to the `random.seed()` function:

```
def main()
    args = get_args()
    random.seed(args.seed) ①
```

① We call the `random.seed()` function to set the initial value of the `random` module's state.

The `args.seed` defaults to the `None` value, so this is the same as not setting the state at all.

If `args.seed` has a value, then it will be used. There is no return value from `random.seed()` as the only change is internal to the `random` module.

We can read about the `random.seed()` function in the REPL:

```
>>> import random
>>> help(random.seed)
```

There we learn that the function will "initialize internal state [of the `random` module] from hashable object." That is, we set an initial value from some *hashable* Python type. Both `int` and `str` types are hashable, but the tests are written with the expectation that you will define the `seed` argument as an `int`. (Remember that the character '`1`' is different from the *integer value 1!*) The default value for `args.seed` is `None`. If the user has not indicated any seed, this is the same as not setting it at all.

If you look at the `test.py` program, you will notice that all the tests that expect a

particular output will pass a `-s/--seed` argument. Here is the first test for output:

```
def test_01():
    out = getoutput(f'{prog} -s 1 -n 1') ①
    assert out.strip() == 'You filthsome, cullionly fiend!' ②
```

- ① Run the program using the `getoutput()` from the `subprocess` module. Use a seed value of 1, and request 1 insult. This function returns only the output from the program.
- ② Verify that the entire output is the one expected insult.

This means the `test.py` will run your program and capture the output into the `out` variable:

```
$ ./abuse.py -s 1 -n 1
You filthsome, cullionly fiend!
```

It will then verify that the program did in fact produce the expected number of insults with the expected selection of words.

9.1.3 Defining the adjectives and nouns

Above I've given you a long list of adjectives and nouns that you should use in your program. In order to pass the tests, they must be in the same order as I have provided. (You may notice that they are alphabetically sorted.) You could create a list by individually quoting each word:

```
>>> adjectives = ['bankrupt', 'base', 'caterwauling']
```

Or you could save yourself a good bit of typing if you use the `str.split()` method to create a new list from a `str` by splitting on spaces:

```
>>> adjectives = 'bankrupt base caterwauling'.split()
>>> adjectives
['bankrupt', 'base', 'caterwauling']
```

If you try to make one giant string of all the adjectives, it will wrap around and look ugly. I'd recommend you use triple quotes (either single or double quotes) that allow you to include newlines:

```
>>> """
... bankrupt base
... caterwauling
... """.split()
['bankrupt', 'base', 'caterwauling']
```

Once you have variables for adjectives and nouns, you might check that you have the right number:

```
>>> assert len(adjectives) == 36
>>> assert len(nouns) == 39
```

9.1.4 Taking random samples and choices

In addition to the `random` module's `seed()` function, we will also use the `choice()` and `sample()` functions. In the `test_bad_adjective_num` function above, we saw one example of using `random.choice()`. We can use it similarly to select a noun from the list of nouns. Notice that this function returns a single item, so, given a list of `str` values, it will return a single `str`:



```
>>> random.choice(nouns)
'braggart'
>>> random.choice(nouns)
'milksop'
```

For the adjectives, you should use `random.sample()`. If you read the `help(random.sample)`, you will see this function takes the list of adjectives and a `k` parameter for how many items to sample:

```
sample(population, k) method of random.Random instance
    Chooses k unique random elements from a population sequence or set.
```

Note that this function returns a new list:

```
>>> random.sample(adjectives, 2)
['detestable', 'peevish']
>>> random.sample(adjectives, 3)
['slanderous', 'detestable', 'base']
```

There is also a `random.choices()` that works similarly to `sample` but which might select the same items twice because it samples "with replacement," so we will not use that.

9.1.5 Formatting the output

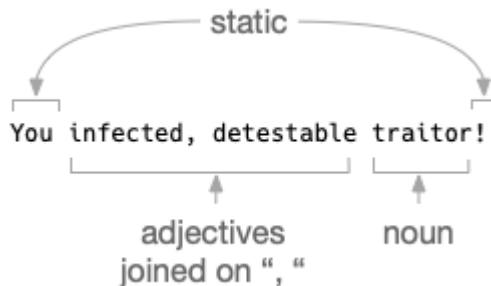
The output of the program is some --number of insults, which you could generate using a `for` loop and the `range()` function. It doesn't matter here that `range()` starts at zero. What's important is that it generates three values:

```
>>> for n in range(3):
...     print(n)
...
0
1
2
```

You can loop the --number of times needed, select your sample of adjectives and your noun, and then format the output. Each insult should start with the string "You ", then have the adjectives joined on a comma and a space, then the noun, and finish with an

exclamation point. You could use either an f-string or the `str.format()` function to print() the output to STDOUT.

Figure 9.1. Each insult will combine the chosen adjectives joined on commas with the selected noun and some static bits of text.



Hints:

- Perform the check for positive values for --adjectives -- number *inside* the `get_args()` function and use `parser.error()` to throw the error while printing a message and the usage.
- If you set the default of `args.seed` to `None` and set `type=int`, you can directly pass the value to `random.seed()`. When the value is `None`, it will be like not setting the value at all.
- Use a `for` loop with the `range()` function to create a loop that will execute - number of times to generate each insult.
- Look at the `random.sample()` and `random.choice()` functions for help in selecting some adjectives and a noun.
- You can use three single ('''') or double quotes ("""") to create a multi-line string and then `str.split()` that to get a list of strings. This is easier than individually quoting a long list of shorter strings (e.g., the list of adjectives and nouns).
- To construct an insult string to print, you can use the + operator to concatenate strings, use the `str.join()` method, or use format strings.

Now give this your best shot before reading ahead to the solution!

9.2 Solution

```
#!/usr/bin/env python3
"""Heap abuse"""

import argparse
import random

# -----
def get_args():
    """Get command-line arguments"""
    (1)
```

```

parser = argparse.ArgumentParser(
    description='Heap abuse',
    formatter_class=argparse.ArgumentDefaultsHelpFormatter)

parser.add_argument('-a',                                     ②
                    '--adjectives',
                    help='Number of adjectives',
                    metavar='adjectives',
                    type=int,
                    default=2)

parser.add_argument('-n',                                     ③
                    '--number',
                    help='Number of adjectives',
                    metavar='adjectives',
                    type=int,
                    default=3)

parser.add_argument('-s',                                     ④
                    '--seed',
                    help='Random seed',
                    metavar='seed',
                    type=int,
                    default=None)

args = parser.parse_args()                                    ⑤

if args.adjectives < 1:                                     ⑥
    parser.error('--adjectives "{}" must be > 0'.format(args.adjectives))

if args.number < 1:                                         ⑦
    parser.error('--number "{}" must be > 0'.format(args.number))

return args                                                 ⑧

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

    args = get_args()                                         ⑨
    random.seed(args.seed)                                   ⑩

    adjectives = """
bankrupt base caterwauling corrupt cullionly detestable dishonest false
filthsome filthy foolish foul gross heedless indistinguishable infected
insatiate irksome lascivious lecherous loathsome lubberly old peevish
rascally rotten ruinous scurilous scurvy slanderous sodden-witted
thin-faced toad-spotted unmannered vile wall-eyed
""".strip().split()                                       ⑪

    nouns = """
Judas Satan ape ass barbemonger beggar block boy braggart butt
carbuncle coward coxcomb cur dandy degenerate fiend fishmonger fool
gull harpy jack jolthead knave liar lunatic maw milksop minion
ratcatcher recreant rogue scold slave swine traitor varlet villain worm
""".strip().split()                                       ⑫

    for _ in range(args.number):                            ⑬
        adjs = ', '.join(random.sample(adjectives, k=args.adjectives)) ⑭

```

```
print(f'You {adjs} {random.choice(nouns)}!') ⑯

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

- ① Bring in the `random` module so we can use functions.
- ② Define the parameter for the number of adjectives, setting the `type=int` and the default value.
- ③ Similarly define the parameter for the number insults.
- ④ The random seed default should be `None`.
- ⑤ Get the args from parsing the argument. Error such as non-integer values will be handled at this point by `argparse`.
- ⑥ If we make it to this point, we can perform additional checking such as verifying that the `args.adjectives` is greater than 0. If there is a problem, call `parser.error()` with the error message to print.
- ⑦ Similarly check for the `args.number`.
- ⑧ If we make it to this point, then all the user's arguments have been validated.
- ⑨ This is where the program actually begins as it is the first action inside our `main()`. We always start off by getting the arguments.
- ⑩ Set the `random.seed()` using whatever value was passed by the user. Any `int` value is valid, and we know that `argparse` has handled the validation and conversion of the argument to an `int`.
- ⑪ Create a list of adjectives by splitting the very long string contained in the triple quotes.
- ⑫ Do the same for the list of nouns.
- ⑬ Use a for loop over the `range()` of the `args.number`. Since we don't actually need the value from the `range()`, we can use the `_` to disregard it.
- ⑭ Use the `random.sample()` function to select the correct number of adjectives and join them on the comma-space str.
- ⑮ Use an f-string to format the output to `print()`.

9.3 Discussion

9.3.1 Defining the arguments

More than half of my solution is just in defining the program's arguments to `argparse`. The effort is well worth the result, because `argparse` will ensure that each argument is a valid integer value because I set `type=int`. Notice there are no quotes around the `int` — it's not the string '`int`' but a reference to the class in Python.

```
parser.add_argument('-a',
                    '--adjectives',
                    help='Number of adjectives',
                    metavar='adjectives',
                    type=int,
                    default=2)
```

- ① The short flag.
- ② The long flag.
- ③ The help message.

- ④ A description of the parameter.
- ⑤ The actual Python type for converting the input, note that this is the bareword `int` for the integer class.
- ⑥ The default value for the number of adjectives per insult.

For `--adjectives` and `--number`, I can set reasonable defaults so that no input is required from the user. This makes your program dynamic, interesting, and testable. How do you know if your values are being used correctly unless you change them and test that the proper change was made in your program? Maybe you started off hardcoding the number of insults and forgot to change the `range()` to use a variable. Without changing the input value and testing that the number of insults changed accordingly, it might be a user who discovers your bug, and that's somewhat embarrassing.

9.3.2 Using `parser.error()`

I really love the `argparse` module for all the work it saves me. For instance, the `type=int` saves me all the trouble of verifying that the input is an integer, creating an error message, and then converting the user's input to an actual `int` value.

Something else I enjoy about `argparse` is that, if I find there is a problem with an argument, I can use `parser.error()` to do four things:

1. Print the short usage of the program to the user
2. Print a specific message about the problem
3. Halt execution of the program
4. Return an error code to the operating system

I can't very easily tell `argparse` that the `--number` should be a positive integer, only that it must be of type `int`. I can, however, inspect the value myself and call `parser.error('message')` if there is a problem. I do all this inside `get_args()` so that, by the time I get the args in my `main()` function, I know they have been validated.

I highly recommend you tuck this tip into your back pocket. It can prove quite handy, saving you loads of time validating user input and generating useful error messages. (And it's really quite likely that the future user of your program will be *you*, and you will really appreciate your efforts!)

9.3.3 Program exit values and STDErr

I would like to highlight the exit value of your program. Under normal circumstances, your program should exit with a value of `0`. In computer science, we often think of `0` as a `False` value, but here it's quite positive. In this instance we should think of it like "zero errors." If you use `sys.exit()` in your code to exit a program prematurely, the default exit value is `0`. If you want to indicate to the operating system or some calling program that your program exited with an error, you should return *any value other than 0*. The `parser.error()` function does this for you automatically.

Additionally, it's common for all error messages to be printed not to STDOUT (standard out) but to STDERR (standard error). Many command shells (like bash) can segregate these two output channels using 1 for STDOUT and 2 for STDERR. Notice how I can use 2> to redirect STDERR to the file called err so that nothing appears on STDOUT:

```
$ ./abuse.py -a -1 2>err
```

And now we can verify that the expected error messages are in the err file:

```
$ cat err
usage: abuse.py [-h] [-a adjectives] [-n insults] [-s seed]
abuse.py: error: --adjectives "-1" must be > 0
```

If you were to handle all of this yourself, you would need to write something like:

```
if args.adjectives < 1:
    parser.print_usage() ①
    print(f"--adjectives \"{args.adjectives}\" must be > 0", file=sys.stderr) ②
    sys.exit(1) ③
```

- ① Print the short "usage". You can also `parser.print_help()` to print the more verbose output for `-h`.
- ② Print the error message to the `sys.stderr` file handle. This is similar to the `sys.stdout` file handle we used in the "Howler" exercise.
- ③ Exit the program with a value that is not 0 to indicate an error.

Writing Pipelines

As you write more programs in your career, you may eventually start chaining them together. We often call these "pipelines" as the output of one program is "piped" to become the input for the next program. If there is an error in any part of the pipeline, we want the entire operation to stop so that the problems can be fixed. A non-zero return value from any program is a warning flag to halt operations.



9.3.4 Controlling randomness with `random.seed()`

Once I'm in `main()` and have my arguments, I can control the randomness of the program by calling `random.seed(args.seed)` because the default value of the `seed` is `None`, and setting `random.seed()` to `None` is the same as not setting it at all. The type of `args.seed` is `int` which is the expected type for our tests. I do not have to validate the argument further. Note that negative integer values are valid!

9.3.5 Iterating for loops with range()

To generate some --number of insults, I use the `range()` function. Because I don't need the number of the insult, I can use the underscore (`_`) as a throwaway value:

```
>>> num_insults = 2
>>> for _ in range(num_insults):
...     print('An insult!')
...
An insult!
An insult!
```

The underscore is a way to unpack a value and indicate that you do not intend to use it. That is, it's not possible to write this:

```
>>> for in range(num_insults):
    File "<stdin>", line 1
        for in range(num_insults):
```

You have to put *something* after the `for` that looks like a variable. If you put a named variable like `n` and then don't use it in the loop, some tools like `pylint` will detect this as a possible error (and well it could be). The `_` shows that you won't use it, which is good information for your future self, some other user, or external tools to know.

You can use multiple `_` variables in the same statement. For instance, I can unpack a 3-tuple so as to get the middle value:

```
>>> x = 'Jesus', 'Mary', 'Joseph'
>>> _, name, _ = x
>>> name
'Mary'
```

9.3.6 Constructing the insults

To create my list of adjectives, I used the `str.split()` method on a long, multi-line string I created using three quotes:

```
>>> adjectives = """
... bankrupt base caterwauling corrupt cullionly detestable dishonest
... false filthsome filthy foolish foul gross heedless indistinguishable
... infected insatiate irksome lascivious lecherous loathsome lubberly old
... peevish rascally rotten ruinous scurilous scurvy slanderous
... sodden-witted thin-faced toad-spotted unmannered vile wall-eyed
... """".strip().split()
>>> nouns = """
... Judas Satan ape ass barbemonger beggar block boy braggart butt
... carbuncle coward coxcomb cur dandy degenerate fiend fishmonger fool
... gull harpy jack jolthead knave liar lunatic maw milksop minion
... ratcatcher recreant rogue scold slave swine traitor varlet villain worm
... """".strip().split()
>>> len(adjectives)
36
>>> len(nouns)
39
```

To select some number of adjectives, I chose to use `random.sample()` function since I needed more than one:

```
>>> import random
>>> random.sample(adjectives, k=3)
['filthsome', 'cullionly', 'insatiate']
```

For just one randomly selected value, I use `random.choice()`:

```
>>> random.choice(nouns)
'boy'
```



To concatenate the adjective using ', ' (a comma and a space) similar to what we did in "Picnic." We can use `str.join()` for that:

```
>>> adjs = random.sample(adjectives, k=3)
>>> adjs
['thin-faced', 'scurvy', 'sodden-witted']
>>> ', '.join(adjs)
'thin-faced, scurvy, sodden-witted'
```

And feed all this to a format string:

```
>>> adjs = ', '.join(random.sample(adjectives, k=3))
>>> print(f'You {adjs} {random.choice(nouns)}!')
You heedless, thin-faced, gross recreant!
```

And now you have a handy way to make enemies and influence people.

9.4 Review

- To indicate a problem to `argparse`, use the `parser.error()` function to print a short usage, report the problem, and exit the program with an error value to indicate a problem.
- The `str.split()` method is a useful way to create a list of string values from a long string.
- The `random.seed()` function can be used to make reproducible "random" selections each time a program is run.
- The `random.choice()` and `random.sample()` functions are useful for randomly selecting one or several items from a list of choices, respectively.

9.5 Going Further

- Read your adjective and nouns from files that are passed as arguments.
- Add tests to verify that the files are processed correctly and new insults are still stinging.

10

Telephone: Randomly mutating strings

"What we have here is a failure to communicate."

— Captain

Now that we've played with randomness, let's apply the idea to randomly mutating a string. This is interesting because strings are actually *immutable* in Python, so we'll have to figure out a way around that. To explore these ideas, we'll write a version of the game of "Telephone." In that game, a secret message is whispered through a line or circle of people. Each time the message is transmitted, it's usually changed in some way. When the last person receives the message, they say it out loud to compare it to the original message.



We will write a program called `telephone.py` that will mimic this game. It will mutate some text by some varying amount and then print "You said: " plus the original text followed by "I heard: " with the modified message. The text may come from the command line:

```
$ ./telephone.py 'The quick brown fox jumps over the lazy dog.'
You said: "The quick brown fox jumps over the lazy dog."
I heard : "TheMquick brown fox jumps ovMr t:e lamy dog."
```

Or from a file:

```
$ ./telephone.py ../inputs/fox.txt
You said: "The quick brown fox jumps over the lazy dog."
I heard : "The quick]b'own fox jumps ovek the la[y dog."
```

In this exercise, you will learn to:

- Round numbers
- Use the `string` module
- Modify strings and lists to introduce random point mutations

10.1 More briefing

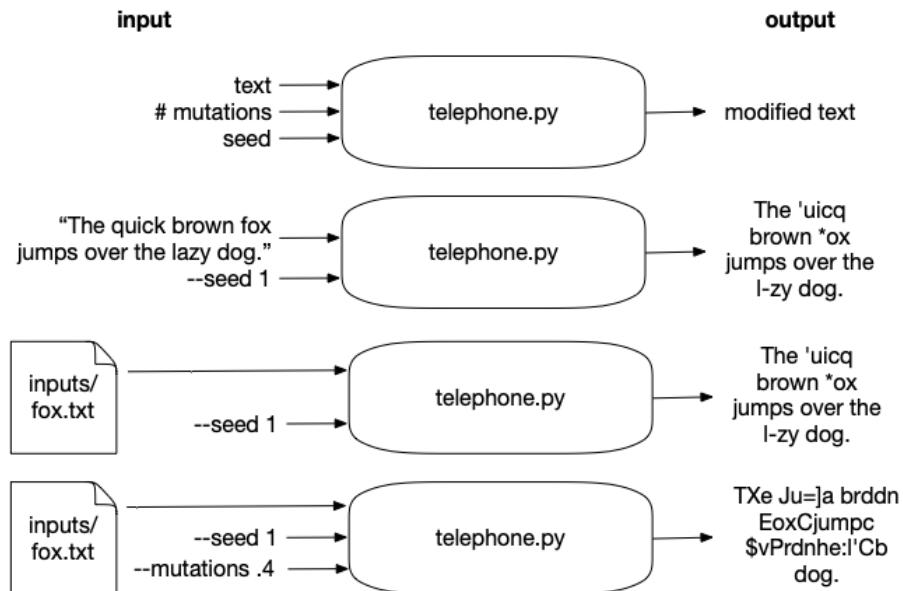
The program should accept a `-m` or `--mutations` option which should be a number between 0 and 1 with a default value of `0.1` (10%). This will be a percentage of the number of letters that should be altered. For instance, `.5` means that 50% of the letters should be changed:

```
$ ./telephone.py ../inputs/fox.txt -m .5
You said: "The quick brown fox jumps over the lazy dog."
I heard : "F#eYquJsY ZrHnna"o. Muz/$ Nver t/Relazy dA!."
```

Because we are using the `random` module, we'll accept an `int` value for the `-s` or `--seed` option so that we can reproduce our "random" selections:

```
$ ./telephone.py ../inputs/fox.txt -s 1
You said: "The quick brown fox jumps over the lazy dog."
I heard : "The 'uicq brown *ox jumps over the l-zy dog."
```

Here is a string diagram of the program:



10.2 Writing telephone.py

I recommend you start by with new.py telephone.py to create the program in the telephone directory. You could also copy the template/template.py to telephone/telephone.py. Modify the get_args() function until your -h output matches the following:

```
$ ./telephone.py -h
usage: telephone.py [-h] [-s seed] [-m mutations] text

Telephone

positional arguments:
  text                  Input text or file

optional arguments:
  -h, --help            show this help message and exit
  -s seed, --seed seed  Random seed (default: None)
  -m mutations, --mutations mutations
                        Percent mutations (default: 0.1)
```

Now run the test suite. You should pass at least the first two tests (the telephone.py program exists and prints something like a "usage" when run with -h or --help).

The next two tests check that your --seed and --mutations options both reject non-numeric values. This should happen automatically if you define these parameter using type of int and float, respectively. Your program should behave like this:

```
$ ./telephone.py -s blargh foo
usage: telephone.py [-h] [-s seed] [-m mutations] text
telephone.py: error: argument -s/--seed: invalid int value: 'blargh'
$ ./telephone.py -m blargh foo
usage: telephone.py [-h] [-s seed] [-m mutations] text
telephone.py: error: argument -m/--mutations: invalid float value: 'blargh'
```

The next test checks if the program rejects --mutations outside of the range 0-1 (where both bounds are inclusive). This is not a check that you can easily describe to argparse, so I suggest that you look at how we handled the validation of the arguments in the abuse.py program. In the get_args() function of that program, we manually checked the value of the arguments and used the parser.error function to throw an error. Your program should do this:

```
$ ./telephone.py -m -1 foobar
usage: telephone.py [-h] [-s seed] [-m mutations] text
telephone.py: error: --mutations "-1.0" must be between 0 and 1
```

Note that a --mutations value of 0 is acceptable, in which case we will print the input text back without modifications. This is another program that accepts the input text either from the command line directly or from a file. I suggest you look at the "Howler" solution where, inside the get_args() function, I suggest that you use os.path.isfile() to detect if the text argument is a file. If it is a file, read the contents of the file for the text value.

Once you have taken care of all the program parameters, start off your `main()` function with setting the `random.seed()` and echoing back the given text:

```
def main():
    args = get_args()
    random.seed(args.seed)
    print(f'You said: "{args.text}"')
    print(f'I heard : "{args.text}"')
```

Your program should handle command-line text:

```
$ ./telephone.py 'The quick brown fox jumps over the lazy dog.'
You said: "The quick brown fox jumps over the lazy dog."
I heard : "The quick brown fox jumps over the lazy dog."
```

Or an input file:

```
$ ./telephone.py ../inputs/fox.txt
You said: "The quick brown fox jumps over the lazy dog."
I heard : "The quick brown fox jumps over the lazy dog."
```

At this point, your code should pass up to `test_for_echo`. The next tests start asking you to mutate the input, so let's discuss how to do that.

10.3 *Calculating the number of mutations*

The number of letters that need to be changed can be calculated by multiplying the length of the input text by the `args.mutations` value. If we want to change 20% of the characters in "[t]he quick brown fox..." string, we'll find that is not a whole number:

```
>>> text = 'The quick brown fox jumps over the lazy dog.'
>>> mutations = .20
>>> len(text) * mutations
8.8
```

We can use the `round` function to give us the nearest integer value. Read `help(round)` to understand how to round floating point numbers to a specific number of digits:

```
>>> round(len(text) * mutations)
9
```

Note that you could also convert a `float` to an `int` by using the `int` function, but this truncates the fractional part of the number rather than rounding it:

```
>>> int(len(text) * mutations)
8
```

You will need this value for later:

```
>>> num_mutations = round(len(text) * mutations)
>>> assert num_mutations == 9
```

10.4 The mutation space

When we change a character, what will we change it to? For this, I'd like to use the `string` module. I'd encourage you to take a look at the documentation by importing the module and reading the `help(string)`:

```
>>> import string
>>> help(string)
```



We can, for instance, get all the lowercase ASCII letters. Note that this is not a method call as there are no parentheses () at the end:

```
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
```

Note that this returned a `str`:

```
>>> type(string.ascii_lowercase)
<class 'str'>
```

For our program, I would like to use `string.ascii_letters` and `string.punctuation`. How do we concatenate two strings together? We can use the + operator. These are the characters we will draw from when replacing:

```
>>> alpha = string.ascii_letters + string.punctuation
>>> alpha
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!#$%&\()*+,./:;<=>?@[\\]^_`{|}~'
```

We're going to use this string to randomly select a character. I need to point out that, even if we both use the same random seed, you and I will get different results if our letters are in a different order. Maybe you create `alpha` like so:

```
>>> alpha = string.punctuation + string.ascii_letters
>>> alpha
'!#$%&\()*+,./:;<=>?@[\\]^_`{|}~abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

So, let's agree we'll sort the `alpha` characters so they are in a consistent order.

10.5 Selecting the characters to mutate

There are at least two approaches we could take to choosing which characters to change.

10.5.1 Non-deterministic selection

One way would be to mimic solution 1 in "Apples". We could iterate for `char` in `text` and flip a coin to decide whether to take the original character or randomly select a new character:

```

new_text = '' ①
for char in args.text: ②
    new_text += random.choice(alpha) if random.random() <= args.mutations else char ③
print(new_text) ④

```

- ① Initialize new_text as an empty string.
- ② Iterate through each character in the text.
- ③ Use random.random() to generate a floating-point value from a uniform distribution of values between 0 and 1. If that value is less than or equal to the args.mutation value, then randomly choose from the alpha; otherwise, use the character.
- ④ Print the resulting new_text.

We used the `random.choice()` function in `abuse.py` to randomly select *one* value from a list of choices. Here we select a character from the alpha if the `random.random()` value falls within the range of the `args.mutation` value (which we know is also a float).

The problem with this approach is that it is *non-deterministic*. That is, by the end of the `for` loop, we are not guaranteed of having made the correct number of changes, which we calculated above as `num_mutations`. If the mutation value was 40%, you could expect that you'd end up changing about 40% of the characters with this code because a random value from a uniform distribution of values between 0 and 1 should be less than or equal to 0.4 about 40% of the time. Since the decision whether to change each character is random, you might not end up with exactly the percentage of mutations requested, especially with a short text. Because of this uncertainty, this approach is considered non-deterministic.

Still, this is a really useful technique that you should note. Imagine you have an input file with millions to potentially billions of lines of text and you want to randomly sample approximately 10% of the lines. The above approach would be a reasonably fast and accurate way to select which lines to take and which to discard. A larger sample size helps to get closer to the desired number of mutations.

10.5.2 Randomly sampling characters

A deterministic approach to the million-line file would require first reading the entire input to count the number of lines, choose which lines to take, and then go back through the file a second time to take those lines. This approach would take *much* longer than the method described above. Depending on how large the input file is, how the program is written, and how much memory your computer has, the program could possibly even crash your computer!

Because our input is rather small, we will use this algorithm because it has the advantage of being exact and testable. Rather than lines of text, though, we'll consider indexes of characters. We've seen the `str.replace()` method (in "Apples") that allows us to change, for instance, all instances of one string to another:

```
>>> 'foo'.replace('o', 'a')
'faa'
```



We can't use that method here, however, because we only want to change some characters. Instead we need to use the indexes of the characters in the text. We can use the `random.sample()` function to select the indexes for us.

The first argument to `random.sample()` needs to be something like a `list`. We should give it a `range()` of numbers up to the length of our text. So, if our text is 44 characters long:

```
>>> text
'The quick brown fox jumps over the lazy dog.'
>>> len(text)
44
```

We can use the `range()` function to make a `list` of numbers up to 44. Note that, in the REPL, we see that `range()` is a lazy function:

```
>>> range(len(text))
range(0, 44)
```

It won't actually produce the 44 values until we force it, which we can do in the REPL using the `list` function. You rarely have to do this in your actual programs, however:

```
>>> list(range(len(text)))
```

We calculated above that the `num_mutations` for altering 20% of `text` is 9. Here are the indexes that need to be changed:

```
>>> indexes = random.sample(range(len(text)), num_mutations)
>>> indexes
[13, 6, 31, 1, 24, 27, 0, 28, 17]
```

I suggest you use a `for` loop to iterate through each of these index values:

```
>>> for i in indexes:
...     print(f'{i:2} {text[i]}')
...
13 w
 6 i
31 t
 1 h
24 s
27 v
 0 T
```

```
28 e
17 o
```

You should replace the character at that position with a randomly selected character from alpha.

```
>>> for i in indexes:
...     print(f'{i:2} {text[i]} changes to {random.choice(alpha)}')
...
13 w changes to b
 6 i changes to W
31 t changes to B
 1 h changes to #
24 s changes to d
27 v changes to :
 0 T changes to C
28 e changes to %
17 o changes to ,
```

I will introduce one other twist — we don't want the replacement value to ever be the same as the character it is replacing. Can you figure out how to get a subset of alpha that *does not* include the character at the position?

10.6 Mutating a string

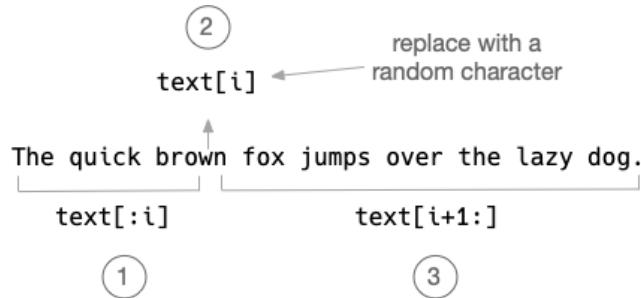
Python `str` variables are *immutable*, meaning we cannot directly modify them. For instance, we want to change the character 'w' at position 13 to a 'b'. It would be handy to directly modify `text[13]`, but that will create an exception:

```
>>> text[13] = 'b'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

The only way to modify the `str` value `text` is to overwrite with a new `str`. So, you need to create a new `str` with

1. the part of `text` before a given index
2. the randomly selected value from `alpha`
3. the part of `text` after a given index

For 1 and 3, you can use string *slices*.



For example, if the index `i` is 13, the slice before it is:

```
>>> text[:13]
'The quick bro'
```

And the part after:

```
>>> text[14:]
'n fox jumps over the lazy dog.'
```

Using the 1, 2, and 3 above, your for loop should be:

```
for i in index:
    text = 1 + 2 + 3
```

Can you figure that out?

10.7 Time to write

OK, the lesson is over. You have to go write this now. Use the tests. Solve them one at a time. You can do this.

10.8 Solution

```
#!/usr/bin/env python3
"""Telephone"""

import argparse
import os
import random
import string ①

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

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

    parser.add_argument('text', metavar='text', help='Input text or file') ②

    parser.add_argument('-s',
                        '--seed',
                        help='Random seed',
                        metavar='seed',
                        type=int,
                        default=None) ③

    parser.add_argument('-m',
                        '--mutations',
                        help='Percent mutations',
                        metavar='mutations',
                        type=float,
                        default=0.1) ④

    args = parser.parse_args() ⑤

    if not 0 <= args.mutations <= 1: ⑥
```

```

        parser.error(f"--mutations '{args.mutations}' must be between 0 and 1')

    if os.path.isfile(args.text):                                ⑦
        args.text = open(args.text).read().rstrip()

    return args                                                 ⑧

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

    args = get_args()
    text = args.text
    random.seed(args.seed)                                     ⑨
    alpha = ''.join(sorted(string.ascii_letters + string.punctuation)) ⑩
    len_text = len(text)                                       ⑪
    num_mutations = round(args.mutations * len_text)          ⑫
    new_text = text                                         ⑬

    for i in random.sample(range(len_text), num_mutations):   ⑭
        new_char = random.choice(alpha.replace(new_text[i], '')) ⑮
        new_text = new_text[:i] + new_char + new_text[i + 1:] ⑯

    print(f'You said: "{text}"\nI heard : "{new_text}"')      ⑰

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

```

- ① We need to import the `string` module here.
- ② The program takes one positional argument for the text. This could be either a string of text or a file which needs to be read.
- ③ The `--seed` parameter is an `int` value with a default of `None`.
- ④ The `--mutations` parameter must be a `float` with a default of `0.1`.
- ⑤ Here we gather the `args` from the command line. If `argparse` detects failures such as non-numeric values for `seed` or `mutations`, then the program dies here and the user sees an error message. If this call succeeds, then `argparse` has validated the arguments and converted the values, e.g., `args.mutations` is actually a `float` value.
- ⑥ If `args.mutations` is *not* in the acceptable range of 0-1, then use `parser.error` to halt the program and print the given message. Note the use of feedback to echo the bad `args.mutation` value to the user.
- ⑦ If `args.text` names an existing file, then read that file for the contents, overwriting the original value of `args.text`.
- ⑧ Return `args` to the caller (line 49).
- ⑨ Be sure to set the `random.seed()` to the value provided by the user. Remember that the default value for `args.seed` is `None`, which is the same as not setting the seed.
- ⑩ Set `alpha` to be the characters we'll use for replacements. The `sorted()` function will return a new list of the characters in the right order, and then we can use the `str.join()` function to turn that back into a `str` value.
- ⑪ Since I use the `len(text)` more than once, I put it into a variable.
- ⑫ Figure the `num_mutations`.

- (13) Make copy of text.
- (14) Use `random.sample()` to get `num_mutations` indexes to change. That function returns a list that we can iterate using the `for` loop.
- (15) Select a new_char using `random.choice()` to select from a string we create by replacing in `alpha` the current character (`text[i]`) with nothing. This ensures that our new character cannot be the same as the one we are replacing.
- (16) Overwrite the text by concatenating the slice before the current index, the `new_char`, and the slice after the current index.
- (17) Print the text.

10.9 Discussion

10.9.1 Defining the arguments

There's nothing in `get_args()` that we haven't seen before. The `--seed` argument is an `int` that we will pass to the `random.seed()` function so as to control the randomness for testing. The default should be `None` so that we can call `random.seed(args.seed)` where `None` is the same as not setting it. (Note that we could also define the seed as a `str` value as both are acceptable seeds.) The `--mutations` is a `float` with a reasonable default, and I test that `args.mutations` is in the proper range and use `parser.error` if it is not. As in other programs, I test if the `text` argument is a file and read the contents if it is.

10.9.2 Mutating a string

Now we saw earlier that we can't just change the `text`:

```
>>> text = 'The quick brown fox jumps over the lazy dog.'
>>> text[13] = 'b'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

So we're going to have to create a *new* string using the `text` before and after `i` which we can get with string slices using `text[start:stop]`. If you leave out `start`, Python starts at `0` (the beginning of the string), and if you leave out `stop` then it goes to the end, so `text[:]` is a copy of the entire string.

If `i` is `13`, then bit before `i` is:

```
>>> i = 13
>>> text[:i]
'The quick bro'
```

And the bit after `i + 1`:

```
>>> text[i+1:]
'n fox jumps over the lazy dog.'
```

Now for what to put in the middle. I noted that we should use `random.choice()` to select a character from `alpha`, which is the combination of all the ASCII letters and

punctuation *without* the current character. I can use the `str.replace()` method to get rid of the current letter:

```
>>> alpha = ''.join(sorted(string.ascii_letters + string.punctuation))
>>> alpha.replace(text[i], '')
'!"#$%&\')*+,--.:;<=>?@ABCDEFGHIJKLMNPQRSTUVWXYZ[\\\]^_`abcdefghijklmnopqrstuvwxyz{|}~'
```

And use that to get a new letter that won't include what it's replacing:

```
>>> new_char = random.choice(alpha.replace(text[i], ''))
>>> new_char
'Q'
```

There are many ways to join strings together into new strings, and the `+` operator is perhaps the simplest:

```
>>> text = text[:i] + new_char + text[i+1:]
>>> text
'The quick broQn fox jumps over the lazy dog.'
```

We do this for each index in the `random.sample()` of indexes, each time overwriting text. After the for loop is done, we have mutated all the positions of the input string and can print it.

10.9.3 Using a list instead of a str

Strings are immutable, but lists are not. We see above that a move like `text[13] = 'b'` creates an exception, but we can change `text` into a list and directly modify it with the same syntax:

```
>>> text = list(text)
>>> text[13] = 'b'
```

And then we can turn that `list` back into a `str` by joining it on the empty string:

```
>>> ''.join(text)
'The quick brobn fox jumps over the lazy dog.'
```

Here is a version of `main()` that uses this approach:

```
def main():
    args = get_args()
    text = args.text
    random.seed(args.seed)
    alpha = ''.join(sorted(string.ascii_letters + string.punctuation))
    len_text = len(text)
    num_mutations = round(args.mutations * len_text)
    new_text = list(text) ①

    for i in random.sample(range(len_text), num_mutations):
        new_text[i] = random.choice(alpha.replace(new_text[i], '')) ②

    print('You said: "{}"\nI heard : "{}".format(text, ''.join(new_text))) ③
```

① Set the `new_text` to be a list of characters from the `text` value.

② Now you can directly modify a value in `new_text`.

- ③ Join `new_list` on the empty string to make a new `str`.

There's no particular advantage of one way over the other, but I would personally choose the second method because I don't like messing around with slicing strings. To me, modifying a `list` in-place makes much more sense than repeatedly chopping up and piecing together a `str`.

Mutations in DNA

For what it's worth, this is (kind of, sort of) how DNA changes over time. The machinery to copy DNA makes mistakes, and mutations randomly occur. Many times the change has no deleterious affect on the organism. Our example only changes characters to other characters, what are called "point mutations" or "single nucleotide variations" (SNV) or "single nucleotide polymorphisms" (SNP) in biology, but we could write a version that would also randomly delete or insert new characters which we call "in-dels" (insertion-deletions). Mutations (that don't result in the demise of the organism) occur at a fairly standard rate, so counting the number of mutations between a conserved region of any two organisms can allow an estimate of how long ago they diverged from a common ancestor!

10.10 Review

- A string cannot be directly modified, but the variable containing the string can be repeatedly overwritten with new values.
- Lists can be directly modified, so it can sometimes help to use `list` on a string to turn it into a `list`, modify that, then use `str.join()` to change it back to a `str`.
- The `string` module has handy functions for dealing with strings.



10.11 Going Further

- Apply the mutations to randomly selected words instead of the whole string.
- Add insertions and deletions in addition to mutations; maybe create arguments for the percentage of each and choose to add or delete characters at the indicated frequency.
- Add an option for `-o` or `--output` that names a file to write the output. The default should be to print to `STDOUT`.
- Add an `argparse` flag to limit the replacements to character values only (no punctuation).
- Add tests to `test.py` for every new feature and ensure your program works properly.

Bottles of Beer Song: Writing and testing functions

Few songs are as annoying as "100 Bottles of Beer on the Wall." Hopefully you've never had to ride for hours in a van with middle school boys who like to sing this. The author has. It's a fairly simple song that we can write an algorithm to generate. This gives us an opportunity to play with counting up and down, formatting strings, and — new to this exercise — writing functions and tests for those functions!



Our program will be called `bottles.py` and will take one option `-n` or `--num` which must be a *positive int* (default 10). The program should print all the verses from `--num` down to 1. There should be two newlines between each verse to visually separate them, but there must be only one newline after the last verse (for one bottle) which should print "No more bottles of beer on the wall" rather than "0 bottles":

```
$ ./bottles.py -n 3
3 bottles of beer on the wall,
3 bottles of beer,
Take one down, pass it around,
2 bottles of beer on the wall!

2 bottles of beer on the wall,
2 bottles of beer,
Take one down, pass it around,
1 bottle of beer on the wall!

1 bottle of beer on the wall,
1 bottle of beer,
```

```
Take one down, pass it around,
No more bottles of beer on the wall!
```

In this exercise, you will:

- Learn how to produce a list of numbers decreasing in value.
- Write a function to create a verse of the song using a test to verify when the verse is correct.
- Explore how for loops can be written as list comprehensions which in turn can be written with the map() function.

11.1 Writing bottles.py

As always, the program should respond to -h or --help with a usage statement:

```
$ ./bottles.py -h
usage: bottles.py [-h] [-n number]

Bottles of beer song

optional arguments:
  -h, --help            show this help message and exit
  -n number, --num number
                        How many bottles (default: 10)
```

Start off by copying the template.py or using new.py to create your bottles.py program. Then modify the get_args() function until your usage matches the above. Since the -h and --help flags are automatically handled by argparse, you need define only the -num option with type=int and default=10.

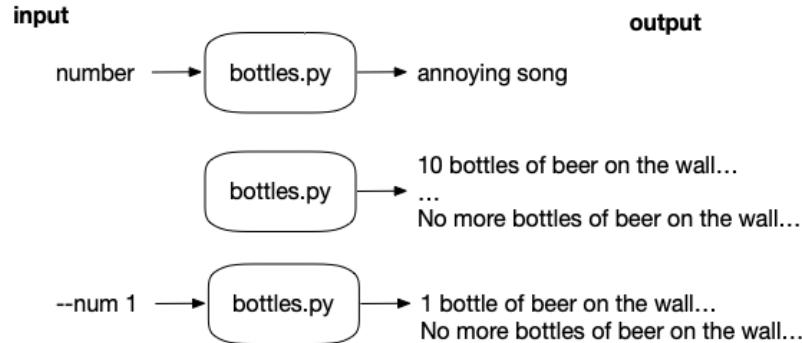
If the -num argument is not an int value, your program should print an error message and exit with an error value. This should happen automatically if you define your parameter to argparse properly:

```
$ ./bottles.py -n foo
usage: bottles.py [-h] [-n number]
bottles.py: error: argument -n/--num: invalid int value: 'foo'
$ ./bottles.py -n 2.4
usage: bottles.py [-h] [-n number]
bottles.py: error: argument -n/--num: invalid int value: '2.4'
```

It should do likewise when the --num is less than 1. To handle this, I suggest you manually check this and call parser.error() inside the get_args() functions as in previous exercises:

```
$ ./bottles.py -n 0
usage: bottles.py [-h] [-n number]
bottles.py: error: --num "0" must be greater than 0
```

Here is a string diagram of the inputs and outputs:



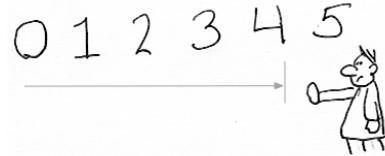
11.2 Counting down

The song starts at the given `--num` value like 10 and needs to count down to 9, 8, 7, and so forth. So how do we do that in Python? We've seen how to use `range(start, stop)` to get a list of integers that go *up* in value. If you give it just one number, it will be considered the `stop` and will assume 0 as the `start`. Because this is a lazy function, I must use the `list()` function in the REPL to force it to produce the numbers:

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

Remember that the `stop` value is never included in the output, so the above stopped at 4 and not 5. If you give `range()` two numbers, they are considered to be `start` and `stop`:

```
>>> list(range(1, 5))
[1, 2, 3, 4]
```



To reverse this sequence, you might be tempted to swap the `start` and `stop` values. Unfortunately, if the `start` is greater than the `stop`, you get an empty list:

```
>>> list(range(5, 1))
[]
```

We saw in "Picnic" that we can use the `reversed` function to reverse a list. This is another lazy function, so, again, I use the `list` function to force the values in the REPL:

```
>>> list(reversed(range(1, 5)))
[4, 3, 2, 1]
```

The `range()` function can also take an optional third argument for a `step` value. For instance, you could use this to count by 5s:

```
>>> list(range(0, 50, 5))
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
```

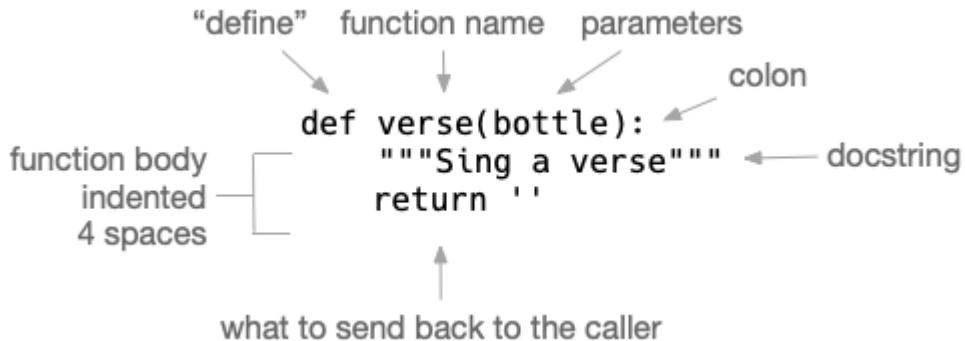


So another way to count down is to swap the `start` and `stop` and use `-1` for the step:

```
>>> list(range(5, 0, -1))
[5, 4, 3, 2, 1]
```

11.3 Writing a function

Up to this point, I've suggested that all your code go into the `main()` function. This is the first exercise where I suggest you write a function. I would like you to consider how to write the code to sing one verse. The function could take the number of the verse and return the text for that verse.



You can start off with something like the diagram. The `def` keyword "defines" a function, and the name of the function follows. Function names should have only letters, numbers and the underscore and cannot start with a number. After the name comes parentheses that hold any parameters that the function accepts. Here our function should be called `verse()` and has the parameter `bottle` (or number or whatever you want to call it). After the parameters comes a colon to show the end of the `def` line. The function body comes next, with all lines being indented at least 4 spaces.

The "docstring" is a comment just after the function definition and will show up in the help for your function. If you enter this into the REPL:

```
>>> def verse(bottle):
...     """Sing a verse"""
...     return ''
...
>>> help(verse)
```

Then you will see:

```
Help on function verse in module __main__:
verse(bottle)
    Sing a verse
```

The return statement tells Python what to send back from the function. It's not very interesting because it will only ever send back the empty string right now:

```
>>> verse(10)
''
```

11.4 Writing a test for verse()

In the spirit of *test-driven development*, let us write a test for `verse()` before we go any further. Here is a test you can use. Add this code into your `bottles.py` program just after your `main()` function:

```
def verse(bottle):
    """Sing a verse"""

    return ''


def test_verse():
    """Test verse"""

    one = verse(1)
    assert one == '\n'.join([
        '1 bottle of beer on the wall,', '1 bottle of beer,', 'Take one down, pass it around,', 'No more bottles of beer on the wall!'
    ])

    two = verse(2)
    assert two == '\n'.join([
        '2 bottles of beer on the wall,', '2 bottles of beer,', 'Take one down, pass it around,', '1 bottle of beer on the wall!'
    ])
```

There are many, many ways you could write this program. I have in mind that my `verse()` function will produce a single verse of the song, returning a new `str` value which is the lines of the verse joined on newlines. You don't have to write it this way, but I would like you to consider what it means to write a function and a *unit test*. If you read about software testing, you'll find that there are different definitions of what a "unit" of code it. In this book, I consider a *function* to be a *unit*, and so my unit tests are tests of individual functions.



verse(1)
verse(2)

By the way, even though the song has potentially hundreds of verses, these two tests should cover everything you need to check. The first test shows that we are looking for "1 bottle" (singular) and not "1 bottles" (plural). We also check that the last line says "No more bottles" instead of "0 bottles." The test for "2 bottles of beer" is making sure that the numbers are "2 bottles" and then "1 bottle." Presumably if you've managed to pass those two tests, your program ought to be able to handle any value.

Here I've written `test_verse()` to test just the `verse()` function. The name of the function matters because I am using the `pytest` module to find all the functions in my code that start with `test_` and run them. If your `bottles.py` program has the above functions for `verse()` and `test_verse()`, you can run `pytest bottles.py`. Try it, and you should see something like this:

```
$ pytest bottles.py
=====
platform darwin -- Python 3.7.3, pytest-4.6.5, py-1.8.0, pluggy-0.12.0
rootdir: /Users/kyclark/work/manning/tiny_python_projects/bottles_of_beer
plugins: openfiles-0.3.2, arraydiff-0.3, doctestplus-0.3.0, remotedata-0.3.1, cov-2.7.1
collected 1 item

bottles.py F [100%]

=====
 FAILURES =====
 test_verse

def test_verse():
    """Test verse"""

    one = verse(1) ①
>     assert one == '\n'.join([
        '1 bottle of beer on the wall,', '1 bottle of beer,', 'Take one down, pass it around,', 'No more bottles of beer on the wall!'
    ])
E     AssertionError: assert '' == '1 bottle of beer on the wal...ottles of beer on the
wall!' ③
E         + 1 bottle of beer on the wall,
E         + 1 bottle of beer,
E         + Take one down, pass it around,
E         + No more bottles of beer on the wall!

bottles.py:49: AssertionError
===== 1 failed in 0.10 seconds =====
```

- ① Call the `verse()` function with the argument 1 to get the one verse of the song.
- ② The > at the beginning of this line indicates this is the source of the error. The test checks if the value of `one` is equal to an expected `str` value. Since it's not, this line throws an exception causing the assertion to fail.
- ③ The E lines show the difference between what was received and what was expected. The value of `one` is the empty string ('') which does not match the expected string "1 bottle of beer..." and so on.

To pass the first test, you could copy the code for the expected value of `one` directly from the test. Change your `verse()` function to be this:

```
def verse(bottle):
    """Sing a verse"""

    return '\n'.join([
        '1 bottle of beer on the wall,', '1 bottle of beer,', 
        'Take one down, pass it around,', 
        'No more bottles of beer on the wall!'
    ])
```

And run your test again. Now the first test should pass and the second one should fail. I'll only show the relevant error lines:

```
===== FAILURES =====
test_verse _____
```

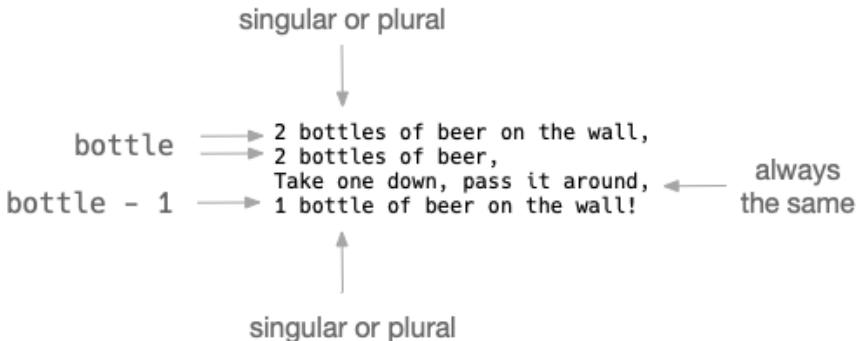
```
def test_verse() -> None:
    """Test verse"""

    one = verse(1)
    assert one == '\n'.join([ ①
        '1 bottle of beer on the wall,', '1 bottle of beer,', 
        'Take one down, pass it around,', 
        'No more bottles of beer on the wall!'
    ])

    two = verse(2) ②
>     assert two == '\n'.join([ ③
        '2 bottles of beer on the wall,', '2 bottles of beer,', 
        'Take one down, pass it around,', '1 bottle of beer on the wall!'
    ])
E     AssertionError: assert '1 bottle of ... on the wall!' == '2 bottles of ... on the
wall!' ④
E         - 1 bottle of beer on the wall,
E         ? ^
E         + 2 bottles of beer on the wall,
E         ? ^
E         + 
E         - 1 bottle of beer,
E         ? ^
E         + 2 bottles of beer, ...
E
E     ...Full output truncated (7 lines hidden), use '-vv' to show
```

- ① This assertion passes this time.
- ② Call the `verse()` with the value of 2.
- ③ Assert that `two` is equal to an expected string.
- ④ These E lines are showing you the problem. It got '1 bottle' but expected '2 bottles', etc.

Figure 11.1. Each verse has four lines where the first two and last are very similar. The third line is always the same. Find the parts that vary.



Go back and look at your `verse()` definition. Think about which parts need to change — the first, second, and fourth lines. The third line is always the same. You're given a value for `bottle` which needs to be used in the first two lines along with either "bottle" or "bottles," depending on the value of `bottle` (Hint: it's only singular for the value 1; otherwise, it's plural). The fourth line needs the value of `bottle - 1` and, again, the proper singular or plural depending on that value. Can you figure how to write this?

Focus on passing those two tests before you move to the next stage of printing the whole song. That is, do not attempt anything until you see this:

```
$ pytest bottles.py
=====
platform darwin -- Python 3.7.3, pytest-4.6.5, py-1.8.0, pluggy-0.12.0
rootdir: /Users/kyclark/work/manning/tiny_python_projects/bottles_of_beer
plugins: openfiles-0.3.2, arraydiff-0.3, doctestplus-0.3.0, remotedata-0.3.1, cov-2.7.1
collected 1 item

bottles.py . [100%]

=====
1 passed in 0.05 seconds =====
```

11.5 Using the `verse()` function

At this point, you know:

1. That the `--num` value is a valid integer value greater than 0.
2. How to count from that `--num` value backwards down to 0.
3. That the `verse()` function will print any one verse properly.

Now you need to put them together. I suggest you start by using a `for` loop with the `range()` function to count down. Use each value from that to produce a `verse()`. There should be 2 newlines after every verse except for the last.

You will use the regular `pytest -xv test.py` (or `make test`) to test the program at this point. In the parlance of testing, the `test.py` is an *integration test* because it checks that the program *as a whole* is working. From this point on, we'll focus more on

how to write *unit* tests to check individual functions as well as *integration* tests to ensure that all the functions work together.

Once you can pass the test suite using a for loop, try to rewrite it using either a list comprehension or a map(). I would suggest commenting out your working code by adding # to the beginnings of the lines and then try other ways to write the algorithm. Use the tests to verify that your code still passes. If is at all motivating, my solution is one line long. Can you write a single line of code that combines the range() and verse() functions to produce the expected output?

Hints:

- Define the --num argument as an int with a default value of 10.
- Use parser.error() to get argparse to print an error message for a negative --num value.
- Write the verse() function. Use the test_verse() function and pytest to make that work properly.
- Combine the verse() function with the range() to create all the verses.

Do try your best to write the program before reading the solution. Also feel free to solve the problem in a completely different way, even writing your own unit tests!

11.6 Solution

```
#!/usr/bin/env python3
"""Bottles of beer song"""

import argparse

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

    parser = argparse.ArgumentParser(
        description='Bottles of beer song',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('-n', ①
                        '--num',
                        metavar='number',
                        type=int,
                        default=10,
                        help='How many bottles')

    args = parser.parse_args() ②

    if args.num < 1: ③
        parser.error(f"--num '{args.num}' must be greater than 0")

    return args

# -----
def main():

    # -----
```

```

"""Make a jazz noise here"""

args = get_args()
print('\n\n'.join(map(verse, range(args.num, 0, -1)))) ④

# -----
def verse(bottle): ⑤
    """Sing a verse"""

    next_bottle = bottle - 1
    s1 = '' if bottle == 1 else 's' ⑥
    s2 = '' if next_bottle == 1 else 's' ⑦
    num_next = 'No more' if next_bottle == 0 else next_bottle ⑧
    return '\n'.join([
        f'{bottle} bottle{s1} of beer on the wall,', ⑨
        f'{bottle} bottle{s1} of beer,', ⑩
        f'Take one down, pass it around,', ⑪
        f'{num_next} bottle{s2} of beer on the wall!', ⑫
    ])

# -----
def test_verse(): ⑬
    """Test verse"""

    one = verse(1) ⑭
    assert one == '\n'.join([
        '1 bottle of beer on the wall,', '1 bottle of beer,', ⑮
        'Take one down, pass it around,', ⑯
        'No more bottles of beer on the wall!' ⑰
    ])

    two = verse(2) ⑯
    assert two == '\n'.join([
        '2 bottles of beer on the wall,', '2 bottles of beer,', ⑰
        'Take one down, pass it around,', '1 bottle of beer on the wall!' ⑱
    ])

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

```

- ① Define the --num argument as an int with a default value of 10.
- ② Parse the command-line argument into the variable args.
- ③ If the args.num is less than 1, use parser.error() to display an error message and exit the program with an error value.
- ④ The map() function expects a function as the first argument and some *iterable* as the second argument. Here we will feed the descending numbers from the range() function to our verse() function. The result from map() is a new list of verses that can be joined on two newlines.
- ⑤ Define a function that can create a single verse().
- ⑥ Define a next_bottle that is one less than the current bottle.
- ⑦ Define a s1 (the first "s") that is either the character 's' or the empty string, depending on the value of bottle.
- ⑧ Do the same for s2 (the second "s"), depending on the value of next_bottle.

- ⑨ Define a value for `next_num` depending on whether the next value is 0 or not.
- ⑩ Create a return string by joining the four lines of text on the newline. Substitute in the variables to create the correct verse.
- ⑪ Define a unit test called `test_verse()` for the `verse()` function. The prefix `test_` means that the `pytest` module will find this function and execute it.
- ⑫ Test the last `verse()` with the value 1.
- ⑬ Test a `verse()` with the value 2.

11.7 Discussion

11.7.1 Defining the arguments

There isn't anything new with the `get_args()` function in this program. By this point, you have had several opportunities to define an optional integer parameter with a default argument as well as using `parser.error()` to halt your program if the user provides a bad argument. By relying on `argparse` to handle so much busy work, you are saving yourself loads of time as well as ensuring that you have good data to work with.

11.7.2 Counting down

We know how to count down from the given `--num`, and we know we can use a `for` loop to iterate:

```
>>> for n in range(3, 0, -1):
...     print(f'{n} bottles of beer')
...
3 bottles of beer
2 bottles of beer
1 bottles of beer
```

Instead of directly making up each verse in the `for` loop, I suggested in the introduction that you create a function called `verse()` to create any given verse and use that with the `range()` of numbers. Up to this point, we've been doing all our work right in the `main()` function. As you grow as a programmer, your programs will become longer, hundreds to even thousands of lines of code (LOC). Long programs and functions can get very difficult to test and maintain, so you should try to break ideas into small, functional units that you can understand and test. Ideally, functions should do *one* thing. If you understand and trust your smaller functions, then you know you can put them together to achieve larger tasks.

11.7.3 Test-Driven Development

I wanted you to add the `test_verse()` function to your program to use with `pytest` to create a working `verse()` function. This idea follows the principles of *Test-Driven Development* described in that book by Kent Beck (2002):

1. Add a new test for an unimplemented unit of functionality.
2. Run all previously written tests and see the newly added test fails.
3. Write code that implements the new functionality.

4. Run all tests and see them succeed.
5. Refactor (rewrite to improve readability or structure).
6. Start at the beginning (repeat).

For instance, assume we want a function that adds 1 to any given number. We'll call it `add1()` and define the function body as `pass` to tell Python "nothing to see here":

```
def add1(n):
    pass
```

Now write a `test_add1()` function where you pass some arguments to the function and use `assert` to verify that you get back the value that you expect.

```
def test_add1():
    assert add1(0) = 1
    assert add1(1) = 2
    assert add1(-1) = 0
```

Run `pytest` (or whatever testing framework you like) and verify that the function *does not work* (of course it won't because it just executes `pass`). Then go fill in the function code that *does work* (`return n + 1` instead of `pass`). Pass all manner of arguments you can imagine, including nothing, one thing, and many things.¹¹

11.7.4 The `verse()` function

I provided you with a `test_verse()` function that shows you exactly what is expected for the arguments of 1 and 2. What I like about writing my tests first is that it gives me an opportunity to think about how I'd like to use the code, what I'd like to give as arguments, and what I expect to get back in return. For instance, what *should* the function `add1()` return if given:

- no arguments?
- more than one argument?
- the value `None`?
- anything other than a numeric type (`int`, `float`, or `complex`) like a `str` value or a `dict`?



You can write test to pass both good and bad values and decide how you want your code to behave under both favorable and adverse conditions.

Here's the `verse()` function I wrote which passes the `test_verse()` function:

```
def verse(bottle):
    """Sing a verse"""

    next_bottle = bottle - 1
    s1 = '' if bottle == 1 else 's'
    s2 = '' if next_bottle == 1 else 's'
```

¹¹ A CS professor once told me in office hours to handle the cases of 0, 1, and n (infinity), and that has always stuck with me.

```

num_next = 'No more' if next_bottle == 0 else next_bottle
return '\n'.join([
    f'{bottle} bottle{s1} of beer on the wall,',
    f'{bottle} bottle{s1} of beer,',
    f'Take one down, pass it around,',
    f'{num_next} bottle{s2} of beer on the wall!',
])

```

It's annotated above, but essentially I isolated all the parts of the return string that vary and created variables to substitute into those places. I need both the bottle and the next_bottle which I can then use to decide if there should be an "s" or not after the the "bottle" strings. I also need to figure out whether to print the next bottle as a number or the string "No more" when the next_bottle is 0. Choosing the values for s1, s2, and num_next all involve *binary* decisions meaning they are a choice between *two* values, so I find it best to use an if expression.

This function passes test_verse(), and so we can move on to using it to generate the song.

11.7.5 Iterating through the verses

I can use a for loop to count down and print each verse():

```

>>> for n in range(3, 0, -1):
...     print(verse(n))
...
3 bottles of beer on the wall,
3 bottles of beer,
Take one down, pass it around,
2 bottles of beer on the wall!
2 bottles of beer on the wall,
2 bottles of beer,
Take one down, pass it around,
1 bottle of beer on the wall!
1 bottle of beer on the wall,
1 bottle of beer,
Take one down, pass it around,
No more bottles of beer on the wall!

```

That's *almost* correct. I want an two newlines in between all the verses. I could use the end option to print to use 2 newlines for all values greater than 1:

```

>>> for n in range(3, 0, -1):
...     print(verse(n), end='\n' * (2 if n > 1 else 1))
...
3 bottles of beer on the wall,
3 bottles of beer,
Take one down, pass it around,
2 bottles of beer on the wall!

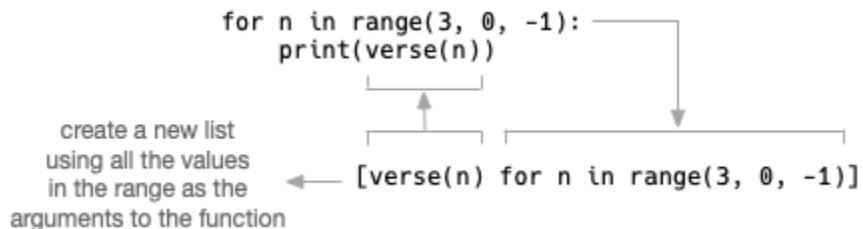
2 bottles of beer on the wall,
2 bottles of beer,
Take one down, pass it around,
1 bottle of beer on the wall!

1 bottle of beer on the wall,
1 bottle of beer,
Take one down, pass it around,

```

```
No more bottles of beer on the wall!
```

Figure 11.2. A for loop compared to a list comprehension.



I know that I could also use the `str.join()` method to put 2 newlines in between items in a list. My items are the verses, and I know I can turn a for loop into a list comprehension:

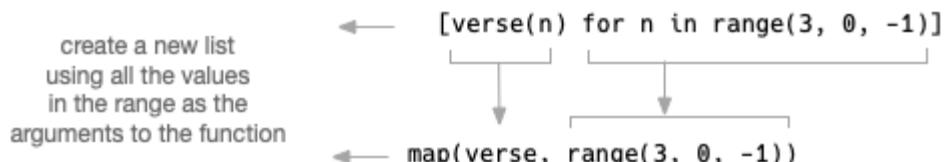
```
>>> verses = [verse(n) for n in range(3, 0, -1)]
>>> print('\n\n'.join(verses))
3 bottles of beer on the wall,
3 bottles of beer,
Take one down, pass it around,
2 bottles of beer on the wall!

2 bottles of beer on the wall,
2 bottles of beer,
Take one down, pass it around,
1 bottle of beer on the wall!

1 bottle of beer on the wall,
1 bottle of beer,
Take one down, pass it around,
No more bottles of beer on the wall!
```

That is a fine solution; however, I would like you to start noticing a pattern we will see repeatedly. That is applying some function to every element of a sequence.

Figure 11.3. A list comprehension can be replaced with `map()`. They both return a new list.



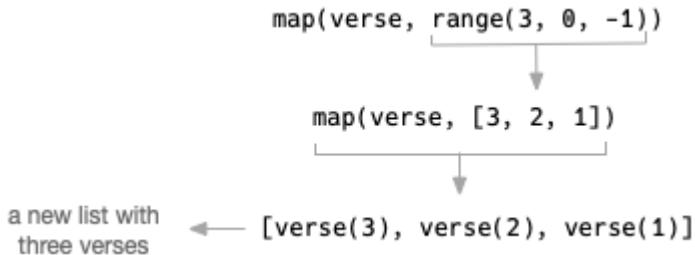
Here we have a descending `range()` of numbers, and we want to send each number through the `verse()` function to collect the results verses on the other end. It's like the paint booth idea in "Apples and Bananas" where the function "painted" the cars "blue" by adding the word "blue" to the front. When we want to apply a function to every element in a sequence, we might consider refactoring the code using `map()`:

```
>>> verses = map(verse, range(3, 0, -1))
>>> print('\n\n'.join(verses))
3 bottles of beer on the wall,
3 bottles of beer,
Take one down, pass it around,
2 bottles of beer on the wall!

2 bottles of beer on the wall,
2 bottles of beer,
Take one down, pass it around,
1 bottle of beer on the wall!

1 bottle of beer on the wall,
1 bottle of beer,
Take one down, pass it around,
No more bottles of beer on the wall!
```

Figure 11.4. The `map()` function will call the `verse()` function with each element produced by the `range()` function. It's functions all the way down.



Whenever I need to transform some sequence of items by some function, I like to start off by thinking about I'll handle just *one* of the items. I find it's much easier to write and test one function with one input rather than some possibly huge list of operations. List comprehensions are often considered more "Pythonic," but I tend to favor `map()` just because it's usually shorter code. If you search for "python list comprehension map," you'll find some people think list comprehensions are easier to read than `map()` but `map()` might possibly be somewhat faster. I wouldn't say either approach is better than the other, and so it really comes down to taste or perhaps a discussion with your teammates.

If you want to use `map()`, remember that it wants a *function* as the first argument and then a sequence of elements that will become arguments to the function. Our `verse()` function (which we've tested!) is the first argument, and the `range()` provides the list. The `map()` functions will make each element of the `range()` an argument to the `verse()` function and will return a new list with the results of all those function calls. Many are the for multi-line loops that can be better written as mapping a function over a list of arguments!

11.7.6 1500 other solutions

There are literally hundreds of ways to solve this problem. The website www.99-bottles-of-beer.net/ claims to have 1500 variations in various languages. Compare your solution to others there. Trivial as the actual program may be, it has allowed us to explore some really interesting ideas in Python, testing, and algorithms!



11.8 Review

- Test-Driven Development (TDD) is central to developing dependable, reproducible code. Tests also give you the freedom to refactor (reorganize and improve for speed or clarity) your code knowing that you can always verify your new version still works the same. As you write your code, always write tests!
- The `range()` function will count backwards if you swap `start` and `stop` and supply the optional third `step` value of `-1`.
- A `for` loop can often be replaced with a list comprehension or a `map()` for shorter, more concise code.

11.9 Going Further

- Replace the Arabic numbers (1, 2, 3) with text (one, two, three).
- Add a `--step` option (positive `int`, default 1) that allows the user to skip, like by 2s or 5s.
- Add a `--reverse` flag to reverse the order of the verses, counting up instead of down.

12

Ransom: Randomly capitalizing text

All this hard work writing code is getting on my nerves. I'm ready to turn to a life of crime! I've kidnapped (cat-napped?) the neighbor's cat. I want to send a ransom note to tell them my demands. In the good old days, I'd cut letters from magazines and paste them onto a piece of paper to spell out my demands. That sounds like too much work. Instead, I'm going to write a Python program called `ransom.py` that will encode text into randomly capitalized letters:

```
$ ./ransom.py 'give us 2 million dollars or the cat gets it!'
gIVE uS 2 MILLioN dollArs OR tHe cAt GETS It!
```

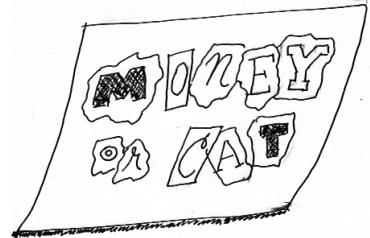


As you can see, my diabolical program accepts the heinous input text as a positional argument. Since this program uses the `random` module, I want to accept a `-s` or `--seed` option so I can replicate the vile output.

```
$ ./ransom.py --seed 3 'give us 2 million dollars or the cat gets it!'
gIVE uS 2 MILLioN dollArs OR tHe cAt GETS It!
```

The dastardly positional argument might name a vicious file, in which case that should be read for the demoniac input text:

```
$ ./ransom.py --seed 2 ../inputs/fox.txt
the qUICk BROWN FOX JUmPs ovEr ThE LAZY DOg.
```



If the unlawful program is run with no arguments, it should print a short, infernal usage:

```
$ ./ransom.py
usage: ransom.py [-h] [-s int] text
ransom.py: error: the following arguments are required: text
```

If the nefarious program is run with -h or --help flags, it should print a longer, fiendish usage:

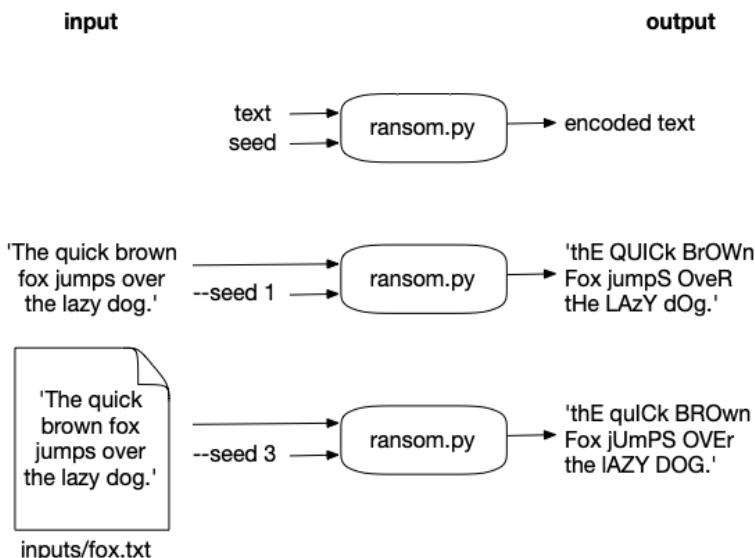
```
$ ./ransom.py -h
usage: ransom.py [-h] [-s int] text

Ransom Note

positional arguments:
  text           Input text or file

optional arguments:
  -h, --help      show this help message and exit
  -s int, --seed int Random seed (default: None)
```

Here is a noxious string diagram to visualize the inputs and outputs:



In this chapter, you will:

- Learn how to use the `random` module to figuratively "flip a coin" to decide between two choices.
- Explore ways to generate new strings from existing one, incorporating random decisions.
- Study the similarities of `for` loops, list comprehensions, and the `map()` function

12.1 Writing `ransom.py`

I would suggest starting with `new.py` or copying the `template/template.py` file to create `ransom.py` in the `ransom` directory. This program, like several before it, accepts a required, positional string for the `text` and an optional integer (default `None`) for the `--seed`. Also as in previous exercises, the `text` argument may name a file that should be read for the `text` value.

To start out, use this for your `main()` code:

```
def main():
    args = get_args()      ①
    random.seed(args.seed) ②
    print(args.text)        ③
```

- ① Get the processed command-line arguments.
- ② Set the `random.seed()` with the value from the user. The default is `None` which is the same as not setting it.
- ③ Start off by echoing back the input.

If you run this program, it should echo the input from the command line:

```
$ ./ransom.py 'your money or your life!'
your money or your life!
```

Or from an input file:

```
$ ./ransom.py ../inputs/fox.txt
The quick brown fox jumps over the lazy dog.
```

The important thing when writing a program is to take baby steps. You should run your program *after every change*, checking manually and with the tests to see if you are progressing. Once you have this working, we can think about how to randomly capitalize this awful message.

12.1.1 Mutating the text

We've seen before that we can't directly modify a `str` value:

```
>>> text = 'your money or your life!'
>>> text[0] = 'Y'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

So how can we randomly change the case of some of the letters? I'd like to suggest that, instead of thinking about how to change many letters, you should think about how to change *one* letter. That is, given a single letter, randomly return that the upper- or lowercase version of the letter. Maybe let's call this function `choose()`. Here's a test for it:

```
def test_choose():
    state = random.getstate() ①
    random.seed(1) ②
    assert choose('a') == 'a' ③
    assert choose('b') == 'b'
    assert choose('c') == 'C'
    assert choose('d') == 'd'
    random.setstate(state) ④
```

- ① The state of the `random` module is *global* to the program. Any change we make here could affect unknown parts of the program, so we save our current state.
- ② Set the random seed to a known value. This is a *global change* to our program. Any other calls to `random` after this line are affected, even if they are in another function or module!
- ③ The `choose()` function is given a series of letters and uses the `assert` statement to test if the value returned by the function is the expected letter.
- ④ Reset the global state to the original value.

Random seeds

Have you wondered how I knew what would be the result of `choose()` for a given random seed? Well, I confess that I wrote the function, then set the seed and ran it with given inputs. I recorded the results as the assertions you see. In the future, these results should still be the same. If they are not, I've changed something and probably broken my program.

12.1.2 Flipping a coin

You need to `choose()` between return the upper- or lowercase version of the character you are given. It's a *binary* choice, meaning we have two options, so we can use the analogy of flipping a coin. Heads or tails? Or, for our purposes, 0 or 1.

```
>>> import random
>>> random.choice([0, 1])
1
```

Or `True` or `False` if you prefer:

```
>>> random.choice([False, True])
True
```



Think about using an `if` expression where you return the uppercase answer when the `0` or `False` option is selected and the lowercase version otherwise. My entire `choose()` function is this one line.

12.1.3 Creating a new string

I'd encourage you to start by mimicking the first approach from "Apples and Bananas." You can iterate through the characters of text using a for loop, using them as the argument to your choose() function, and building a new list or str from the results. Once you can pass the test with a for loop, try to rewrite it as a list comprehension and then a map().

Now off you go! Write the program, pass the tests.

12.2 Solution

```
#!/usr/bin/env python3
"""Ransom note"""

import argparse
import os
import random

# -----
def get_args():
    """get command-line arguments"""

    parser = argparse.ArgumentParser(
        description='Ransom Note',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('text', metavar='text', help='Input text or file') ①

    parser.add_argument('-s',      ②
                        '--seed',
                        help='Random seed',
                        metavar='int',
                        type=int,
                        default=None)

    args = parser.parse_args() ③

    if os.path.isfile(args.text): ④
        args.text = open(args.text).read().rstrip()

    return args ⑤

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

    args = get_args()
    text = args.text
    random.seed(args.seed) ⑥
    ransom = [] ⑦
    for char in args.text:
        ransom.append(choose(char)) ⑧

    print('.join(ransom)) ⑩
```

```

# -----
def choose(char): ⑪
    """Randomly choose an upper or lowercase letter to return"""

    return char.upper() if random.choice([0, 1]) else char.lower() ⑫

# -----
def test_choose(): ⑬
    """Test choose"""

    state = random.getstate() ⑭
    random.seed(1) ⑮
    assert choose('a') == 'a' ⑯
    assert choose('b') == 'b'
    assert choose('c') == 'c'
    assert choose('d') == 'd'
    random.setstate(state) ⑰

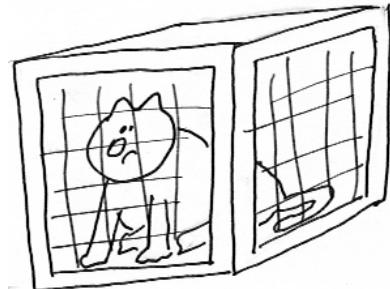
# -----
if __name__ == '__main__':
    main()

```

- ① The `text` argument is a positional string value.
- ② The `--seed` option is an integer that defaults to `None`
- ③ Process the command-line arguments into the `args` variable.
- ④ If the `args.text` is a file, use the contents of that as the new `'args.text` value.
- ⑤ Return the `args` to the caller.
- ⑥ Set the `random.seed()` to the given `args.seed` value. The default is `None`, which is the same as not setting it. That means the program will appear random when no seed is given but will be testable when we do provide one.
- ⑦ Create an empty list to hold the new ransom message.
- ⑧ Use a `for` loop to iterate through each character of `args.text`.
- ⑨ Append the chosen letter to the `ransom` list.
- ⑩ Join the `ransom` list on the empty string to create a new `str` to print.
- ⑪ Define a function to randomly return the upper- or lowercase version of a character.
- ⑫ Use the `random.choice()` to select either 0 or 1 which, in the Boolean context of the `if` expression, evaluate to `False` and `True`, respectively.
- ⑬ Define a `test_choose()` function that will be run by `pytest`. It takes no arguments.
- ⑭ Save the current state of the `random` module.
- ⑮ Set the `random.seed()` to a known value for the purposes of the test.
- ⑯ Use the `assert` statement to verify that we get the expected result from the `choose()` for a known argument.
- ⑰ Reset the `random` module's state so that our changes won't affect any other part of the program.

12.3 Discussion

I like this problem because there are so many interesting ways to solve it. I know, I know, Python likes there to be "one obvious way" to solve it, but let's explore, shall we? There's nothing in the `get_args()` that we haven't seen several times by now, so let's skip that.



12.3.1 Iterating through elements in a sequence

Assume that we have the following cruel message:

```
>>> text = '2 million dollars or the cat sleeps with the fishes!'
```

We want to randomly upper- and lowercase the letters. As suggested in the description of the problem, we can use a `for` loop to iterate over each character. One way to print an uppercase version of the text is to print an uppercase version of *each letter*:

```
for char in text:  
    print(char.upper(), end='')
```

That would give us "2 MILLION DOLLARS OR THE CAT SLEEPS WITH THE FISHES!" Now, instead of always printing `char.upper()`, we could randomly choose between `char.upper()` and `char.lower()`. For that, let's use `random.choice()` to choose between two values like `True` and `False` or `0` and `1`:

```
>>> import random  
>>> random.choice([True, False])  
False  
>>> random.choice([0, 1])  
0  
>>> random.choice(['blue', 'green'])  
'blue'
```



Following the first solution from "Apples and Bananas," we could create a new `list` to hold our ransom message and add these random choices:

```
ransom = []  
for char in text:  
    if random.choice([False, True]):  
        ransom.append(char.upper())  
    else:  
        ransom.append(char.lower())
```

Then we can join the new characters on the empty string to print a new string:

```
print(''.join(ransom))
```

It's much less code to write this with an `if` expression to select whether to take the upper- or lowercase character.

```
ransom = []
for char in text:
    ransom.append(char.upper() if random.choice([False, True]) else char.lower())

ransom.append(char.upper())
    if random.choice([False, True])
        ransom.append(char.upper())
    else:
        ransom.append(char.lower())
```

The diagram illustrates the flow of control for the if expression. A bracket above the first line of the if block points to the condition `random.choice([False, True])`. Another bracket below the condition points to the two branches: `ransom.append(char.upper())` for the `True` branch and `ransom.append(char.lower())` for the `else` branch.

We don't have to use actual Boolean values (`False` and `True`). We could use `0` and `1` instead:

```
ransom = []
for char in text:
    ransom.append(char.upper() if random.choice([0, 1]) else char.lower())
```

When numbers are evaluated *in a Boolean context* (that is, in a place where Python expects to see a Boolean value), `0` is considered `False` and every other number is `True`.

12.3.2 Writing a function to choose the letter

The `if` expression is a bit of code that we could put into a function. I find it hard to read shoved inside the `ransom.append()`. By putting it into a function, we can give it a descriptive name and write a test for it:

```
def choose(char):
    """Randomly choose an upper or lowercase letter to return"""

    return char.upper() if random.choice([0, 1]) else char.lower()
```

Now we can run the `test_choose()` function to test that our function does what we think. This code is much easier to read:

```
ransom = []
for char in text:
    ransom.append(choose(char))
```

12.3.3 Another way to write `list.append()`

The above annotated solution creates an empty list to which we `list.append()` the return from `choose()`. Another way to write `list.append()` is using the `+=` operator to add the right-hand value (the element to add) to the left-hand side (the list):

```
ransom.append(choose(char))
```



```
ransom += choose(char)
```

add the result
of the function
to ransom

```
def main():
    args = get_args()
    random.seed(args.seed)

    ransom = []
    for char in args.text:
        ransom += choose(char)

    print(''.join(ransom))
```

12.3.4 Using a str instead of a list

The list solution requires that `ransom` must be joined on the empty string to make a new string to print. We could, instead, start off with an empty string and build that up, one character at a time using the `+=` operator:

```
def main():
    args = get_args()
    random.seed(args.seed)

    ransom = ''
    for char in args.text:
        ransom += choose(char)

    print(ransom)
```

12.3.5 Using a list comprehension

We've seen this pattern before when we initialize an empty `str` or `list` and then build it up with a `for` loop. It's almost always better written as a list comprehension. Once you get used to the list comprehension, it actually makes more sense because the expression *returns a new list* in one line of code:

```
def main():
    args = get_args()
    random.seed(args.seed)
    ransom = [choose(char) for char in args.text]
    print(''.join(ransom))
```

Or skip creating the `ransom` variable altogether. As a general rule, I only assign a value to a variable if I use it more than once or if I feel it makes my code more readable:

```
def main():
    args = get_args()
    random.seed(args.seed)
```

```
print(''.join([choose(char) for char in args.text]))
```

A `for` loop is really for iterating through some sequence and producing *side effects* like printing values or handling lines in a file. If your goal is to create a new list, a list comprehension is probably the best tool. Any code that would go into the body of the `for` loop to process an element is better placed into a function with a test.

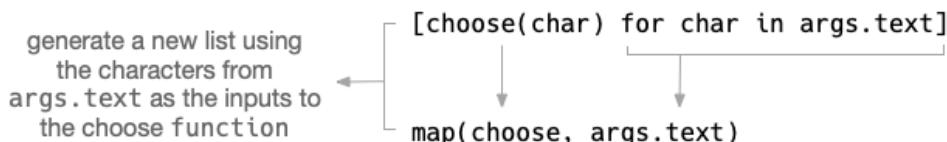
12.3.6 Using a `map()` function

The `map()` solution is fairly elegant and is a good bit less typing than the list comprehension. Remember that `map()` returns a new list built by supplying each element of `args.text` to the `choose()` function:

```
def main():
    args = get_args()
    random.seed(args.seed)
    ransom = map(choose, args.text)
    print(''.join(ransom))
```

Or, again, leave out the `ransom` assignment and use the list that comes back from `map()` directly:

```
def main():
    args = get_args()
    random.seed(args.seed)
    print(''.join(map(choose, args.text)))
```



12.4 Comparing methods

It may seem silly to spend so much time working through so many ways to solve what is an essentially trivial problem, but one of the goals in this book is to explore the various ideas available in Python. The first method is a very imperative solution that a C or Java programmer would probably write. The version using a list comprehension is very idiomatic to Python — it is "Pythonic," as Pythonistas would say. The `map()` solution would look very familiar to someone from a purely functional language like Haskell.

They all accomplish the same goal but embody different aesthetics and programming paradigms. My preferred solution would be the last one using `map()`, but you should choose an approach that makes the most sense to you.

MapReduce

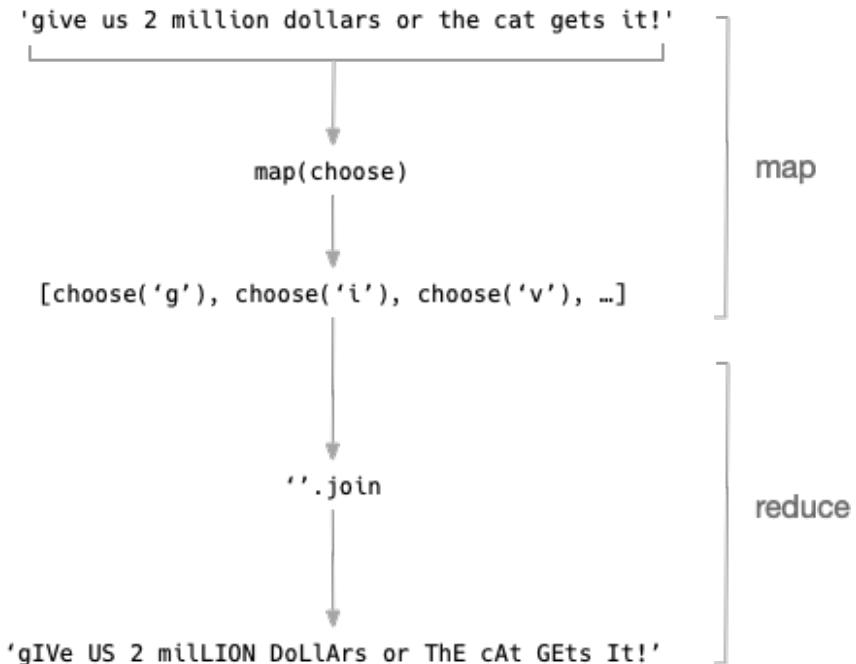
In 2004, Google release a paper on their "MapReduce" algorithm. The "map" phase applies some transformation to all the elements in a collection such as all the pages of the Internet that need to be indexed for searching. These operations can happen in *parallel*, meaning you can use many machines to

process the pages separately from each other and in any order. The "reduce" phase then brings all the processed elements back together, maybe to put the results into a unified database.

In our `ransom.py` program, the "map" part selected a randomized case for the given letter, and the "reduce" part was putting all those bits back together into a new string. Conceivably, `map()` could make use of multiple processors to run the functions *in parallel* as opposed to *sequentially* (like with a `for` loop), possibly cutting the time to produce the results.

Learning about map/reduce was, to me, a bit like learning the name of a new bird. I never even noticed that bird before, but, once I was told its name, I saw it everywhere. Once you understand this pattern, you'll begin see recognize it in many places!

Figure 12.1. MapReduce



12.5 Review

- Whenever you have lots of things to process, try to think about how you'd process just one of them.
- Write a test that helps you imagine how you'd like to use the a function to process one item. What will you pass in, and what do you expect back?
- Write your function to pass your test. Be sure to think about what you'll do with both good and bad input.
- To apply your function to each element in your input, use a `for` loop, a list comprehension, or a `map()`.



12.6 Going Further

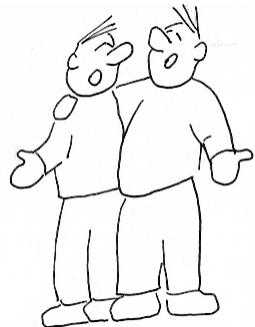
- Write a version that uses other ways of representing letters by combining ASCII characters such as the following. Feel free to make up your own substitutions. Be sure to update your tests.

A	4	K	<
B	3	L	_
C	(M	\\
D)	N	\\
E	3	P	`
F	=	S	5
G	(-	T	+
H	-	V	\ \ /
I	1	W	\ \ \ /
J	_		

13

Twelve Days of Christmas: Algorithm design

Perhaps one of the worst songs of all time and the one that is sure to ruin my Christmas spirit is "The Twelve Days of Christmas." WILL IT EVER STOP!? AND WHAT IS WITH ALL THE BIRDS?! Still, it's pretty interesting to write an algorithm to generate the song starting from any given day because you have to count *up* as you add each verse (day) and then count *down* inside the verses (recapitulating the previous days' gifts). We'll be able to build off what you learned writing the "99 Bottles of Beer" song.



Our program will be called `twelve_days.py` and will generate the "Twelve Days of Christmas" song up to a given `-n` or `--num` argument (default 12). Note that there should be two newlines in between each verse but only one at the end:

```
$ ./twelve_days.py -n 3
On the first day of Christmas,
My true love gave to me,
A partridge in a pear tree.

On the second day of Christmas,
My true love gave to me,
Two turtle doves,
And a partridge in a pear tree.

On the third day of Christmas,
My true love gave to me,
Three French hens,
Two turtle doves,
And a partridge in a pear tree.
```

The text will be printed to `STDOUT` unless there is a `-o` or `--outfile` argument, in which case the text should be placed inside a file by that name. Note there should be 113 lines of text for the entire song:

```
$ ./twelve_days.py -o song.txt
$ wc -l song.txt
 113 song.txt
```

In this exercise, you will:

- Create an algorithm to generate "The Twelve Days of Christmas" from any given day in the range 1-12.
- Reverse a list
- Use the `range()` function
- Write text to a file or to `STDOUT`

13.1 Writing `twelve_days.py`

As always, I suggest you start by running `new.py twelve_days.py` in the `twelve` directory or by copying the `template/template.py` file. Your program should will take two options:

1. `-n` or `--num`: an `int` with a default of 12
2. `-o` or `--outfile`: a `str` with a default of `nothing/STDOUT`

When run with the `-h` or `--help` flag, the program should print a usage:

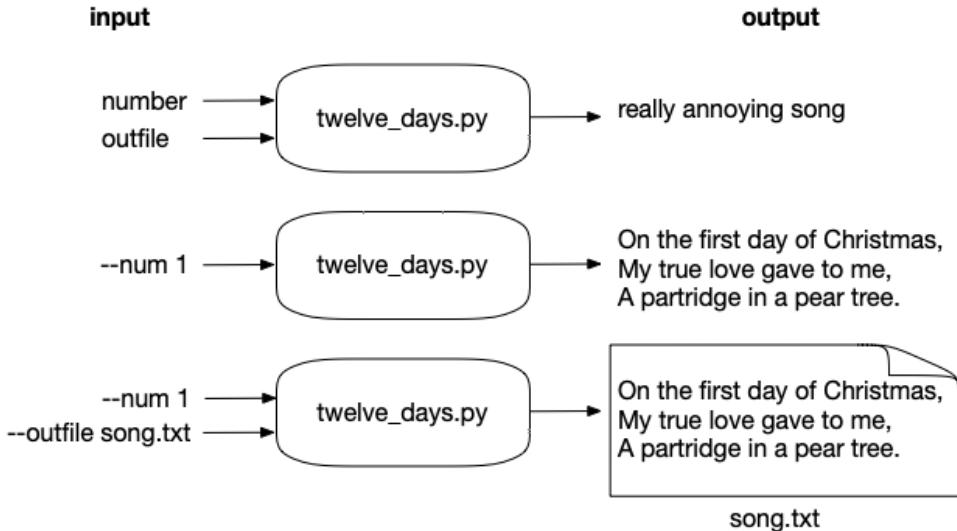
```
$ ./twelve_days.py -h
usage: twelve_days.py [-h] [-n days] [-o FILE]

Twelve Days of Christmas

optional arguments:
  -h, --help            show this help message and exit
  -n days, --num days  Number of days to sing (default: 12)
  -o FILE, --outfile FILE
                        Outfile (STDOUT) (default: )
```

Before trying to write the song, make your usage match the above. At this point, you should pass the first two tests when you run the test suite.

Here is a holly, jolly string diagram to get you in the mood for writing:



The program should complain if the `--num` value is not in the range 1-12. I suggest you check this inside the `get_args()` function and use `parser.error()` to halt with an error and usage:

```
$ ./twelve_days.py -n 21
usage: twelve_days.py [-h] [-n days] [-o FILE]
twelve_days.py: error: --num "21" must be between 1 and 12
```

Once you've handled the bad `--num`, you should pass the first three tests.

13.1.1 Counting

In the "99 Bottles of Beer" song, we need to count down from a given number. Here we need to count up to the `--num`. The `range()` function will give us what we need, but we must remember to start at 1 (the default is 0) and that *the upper bound is not included*:

```
>>> num = 3
>>> list(range(1, num))
[1, 2]
```

You need to add 1 to whatever you're given for `--num`:

```
>>> list(range(1, num + 1))
[1, 2, 3]
```

Let's start by printing something like the first line of each verse:

```
>>> for day in range(1, num + 1):
...     print(f'On the {day} day of Christmas,')
...
On the 1 day of Christmas,
On the 2 day of Christmas,
On the 3 day of Christmas,
```

At this point, I'm starting to think about how we wrote "99 Bottles of Beer." There we ended up creating a `verse()` function that would generate any *one* verse. Then we used `str.join()` to put them all together with two newlines. I suggest we try the same approach here.

```
def verse(day):
    """Create a verse"""
    return f'On the {day} day of Christmas,'
```

Let's see how we can use it:

```
>>> for day in range(1, num + 1):
...     print(verse(day))
...
On the 1 day of Christmas,
On the 2 day of Christmas,
On the 3 day of Christmas,
```

Notice that the function will not `print()` the string but will `return` the verse so that we can test it:

```
>>> assert verse(1) == 'On the 1 day of Christmas,'
```

Here's a simple `test_verse()` function we could start off with:

```
def test_verse():
    """ Test verse """
    assert verse(1) == 'On the 1 day of Christmas,'
    assert verse(2) == 'On the 2 day of Christmas,'
```

This is incorrect because it should say "On the *first* day" or the "*second* day," not "1 day" or "2 day." Still, it's a place to start. Add the `verse()` and `test_verse()` functions to your `twelve_days.py` program and then run `pytest twelve_days.py` to verify your code.

13.1.2 Creating the ordinal value

Maybe the first thing to do is to change the numeric value to its ordinal position, that is "1" to "first," "2" to "second." You could use a dictionary like we used in "Jump The Five" to associate each `int` value 1-12 with its `str` value. That is, you might create a new `dict` called `ordinal`:

```
>>> ordinal = {} # what goes here?
```

So that you can do:

```
>>> ordinal[1]
'first'
>>> ordinal[2]
'second'
```

You could also use a `list` if you think about how you could use the each day in the `range()` to index into a `list` of the ordinal strings.

```
>>> ordinal = [] # what goes here?
```

How would you use the `int` value of a given number to access one of the values in the list called `ordinal`?

Now your `verse()` might look something like:

```
def verse(day):
    """Create a verse"""
    ordinal = [] # something here!
    return f'On the {ordinal[day]} of Christmas,'
```

You can update your test with your expectations:

```
def test_verse():
    """ Test verse """
    assert verse(1) == 'On the first day of Christmas,'
    assert verse(2) == 'On the second day of Christmas,'
```

Once you have this working, you should be able to replicate something like this:

```
>>> for day in range(1, num + 1):
...     print(verse(day))
...
On the first day of Christmas
On the second day of Christmas
On the third day of Christmas
```

If you put the `test_verse()` function inside your `twelve_days.py` program, you can verify that your `verse()` function works by running `pytest twelve_days.py`. The `pytest` module will run any function that has a name starting with `test_`.

Shadowing

You might be tempted to use the variable name `ord`, and you would be allowed by Python to do this. The problem is that Python has function called `ord` that returns "the Unicode code point for a one-character string":

```
>>> ord('a')
97
```

Python will not complain if you define a variable or another function with the name `ord`:

```
>>> ord = {}
```

Such that you could do this:

```
>>> ord[1]
'first'
```

But then it overwrites the actual `ord` function, and so breaks a function call:

```
>>> ord('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'dict' object is not callable
```

This is called "shadowing," and it's quite dangerous. Any code in the scope of the shadowing would be affected by the change.

Tools like `pylint` can help you find problems like this in your programs. Assume the following code:

```
$ cat shadow.py
#!/usr/bin/env python3

ord = {}
print(ord('a'))
```

Here is what `pylint` has to say:

```
$ pylint shadow.py
***** Module shadow
shadow.py:3:0: W0622: Redefining built-in 'ord' (redefined-builtin)
shadow.py:1:0: C0111: Missing module docstring (missing-docstring)
shadow.py:4:6: E1102: ord is not callable (not-callable)

-----
Your code has been rated at -25.00/10
```

It's good to double-check your code with tools like `pylint`, `flake8`, and `mypy`!

13.1.3 Making the verses

Now that we have the basic structure of the program, let's focus on creating the *correct* output. We'll update our `test_verse()` with the actual values for the first two verses. You can, of course, add more tests, but presumably if we can manage the first two, then we can handle all the other days:

```
def test_verse():
    """Test verse"""

    assert verse(1) == '\n'.join([
        'On the first day of Christmas,', 'My true love gave to me,',
        'A partridge in a pear tree.')
    ])

    assert verse(2) == '\n'.join([
        'On the second day of Christmas,', 'My true love gave to me,',
        'Two turtle doves,', 'And a partridge in a pear tree.')
    ])
```

If you add this to your `twelve_days.py` program, you can run `pytest twelve_days.py` to see how your `verse()` function is failing:

```
===== FAILURES =====
test_verse

def test_verse():
    """Test verse"""

>     assert verse(1) == '\n'.join([ ①
        'On the first day of Christmas,', 'My true love gave to me,',
        'A partridge in a pear tree.'
```

```

        ])
E      AssertionError: assert 'On the first...of Christmas,' == 'On the first ... a pear
tree.'
E          - On the first day of Christmas, ②
E          + On the first day of Christmas, ③
E          ?
E          + My true love gave to me,
E          + A partridge in a pear tree.

twelve_days.py:88: AssertionError
=====
1 failed in 0.11 seconds =====

```

- ① The leading > shows this is the code that is creating an exception. We are running `verse(1)` and asking if it's equal to the expected verse.
- ② This is the text that `verse(1)` actually produced, which is only the first line of the verse.
- ③ The lines following are what was expected.

Now we need to supply the rest of the lines for each verse. They all start off the same:

```
On the {ordinal[day]} day of Christmas,
My true love gave to me,
```

And then we need to add these gifts for each day:

1. A partridge in a pear tree
2. Two turtle doves
3. Three French hens
4. Four calling birds
5. Five gold rings
6. Six geese a laying
7. Seven swans a swimming
8. Eight maids a milking
9. Nine ladies dancing
10. Ten lords a leaping
11. Eleven pipers piping
12. Twelve drummers drumming



Note that for every day greater than 1, the last line changes "A partridge..." to "And a partridge in a pear tree."

Each verse needs to count backwards from the given day. For example, if the day is 3, then:

1. Three French hens
2. Two turtle doves

3. And a partridge in a pear tree

We've talked before in the "Picnic" chapter about how to reverse a list either with the `list.reverse` method or the `reversed()` function, and we also used these ideas in "99 Bottles," so this code should not be unfamiliar:

```
>>> day = 3
>>> for n in reversed(range(1, day + 1)):
...     print(n)
...
3
2
1
```

Try to make the function return the first two lines and then the count down of the days:

```
>>> print(verse(3))
On the third day of Christmas,
My true love gave to me,
3
2
1
```

And then, instead of 3 2 1, add the actual gifts:

```
>>> print(verse(3))
On the third day of Christmas,
My true love gave to me,
Three French hens,
Two turtle doves,
And a partridge in a pear tree.
```

If you can get that to work, then you ought to be able to pass the `test_verse()` test.

13.1.4 Using the `verse()` function

Once you have that working, think about a final structure that calls your `verse()`. It could be a `for` loop:

```
verses = []
for day in range(1, args.num + 1):
    verses.append(verse(day))
```

A list comprehension:

```
verses = [verse(day) for day in range(1, args.num + 1)]
```

Or a `map()`:

```
verses = map(verse, range(1, args.num + 1))
```



13.1.5 Printing

Once you have all the verses, you can use the `str.join()` method to print the output.

The default is to print this to "standard out" (STDOUT), but the program will also take an optional --outfile that names a file to write the output. You can go back to the "Howler" solution and copy the code for how we handled this before, but I suggest you investigate using type=argparse.FileType('wt') with a default of sys.stdout. If the user supplies an --outfile argument that is the name of a file that can be written, it will be available to you as a *writable file handle*. If the user does not supply an argument, then the default will be sys.stdout.

13.1.6 Time to write

It's not at all mandatory that you solve the problem the way that I describe. The "correct" solution is the one that you write and understand which passes the test suite. It's fine if you like the idea of creating a functions for verse() and using the provided test. It's fine if you want to go another way, but do try to think of writing small functions *and tests* to solve small parts of your problem and combining them to solve the larger problem.

If you need more than one sitting or even several days, take your time. Sometimes a good walk or a nap can do wonders for solving problems. Don't neglect your hammock or a nice cup of tea.

13.2 Solution

```
#!/usr/bin/env python3
"""Twelve Days of Christmas"""

import argparse
import sys

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

    parser = argparse.ArgumentParser(
        description='Twelve Days of Christmas',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('-n', ①
                        '--num',
                        help='Number of days to sing',
                        metavar='days',
                        type=int,
                        default=12)

    parser.add_argument('-o', ②
                        '--outfile',
                        help='Outfile',
                        metavar='FILE',
                        type=argparse.FileType('wt'),
                        default=sys.stdout)

    args = parser.parse_args() ③

    if args.num not in range(1, 13): ④
```

```

parser.error(f"--num '{args.num}' must be between 1 and 12') ⑤

return args

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

    args = get_args() ⑥
    verses = map(verse, range(1, args.num + 1)) ⑦
    print('\n\n'.join(verses), file=args.outfile) ⑧

# -----
def verse(day): ⑨
    """Create a verse"""

    ordinal = [ ⑩
        'first', 'second', 'third', 'fourth', 'fifth', 'sixth', 'seventh',
        'eighth', 'ninth', 'tenth', 'eleventh', 'twelfth'
    ]

    gifts = [ ⑪
        'A partridge in a pear tree.',
        'Two turtle doves,',
        'Three French hens,',
        'Four calling birds,',
        'Five gold rings,',
        'Six geese a laying,',
        'Seven swans a swimming,',
        'Eight maids a milking,',
        'Nine ladies dancing,',
        'Ten lords a leaping,',
        'Eleven pipers piping,',
        'Twelve drummers drumming,',
    ]

    lines = [ ⑫
        f'On the {ordinal[day - 1]} day of Christmas,',
        'My true love gave to me,'
    ]

    lines.extend(reversed(gifts[:day])) ⑬

    if day > 1:
        lines[-1] = 'And ' + lines[-1].lower() ⑭ ⑮

    return '\n'.join(lines) ⑯

# -----
def test_verse(): ⑰
    """Test verse"""

    assert verse(1) == '\n'.join([
        'On the first day of Christmas,',
        'My true love gave to me,',
        'A partridge in a pear tree.'
    ])

    assert verse(2) == '\n'.join([

```

```

'On the second day of Christmas,', 'My true love gave to me,',  

'Two turtle doves,', 'And a partridge in a pear tree.'  

])  
  

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

```

- ① The --num option is an int with a default of 12.
- ② The --outfile option is a type=argparse.FileType('wt') with a default of sys.stdout. If the user supplies a value, then it must be the name of a writable file in which case argparse will open() the file for writing.
- ③ Capture the results of parsing the command-line arguments (parser.parse_args) into the args variable.
- ④ Check that the given args.num is in the allowed range (1-12, inclusive).
- ⑤ If args.num is invalid, use parser.error() to print a short usage and the error message to STDERR and exit the program with an error value. Note that the error message includes the bad value for the user and explicitly states that a good value should be in the range 1-12.
- ⑥ Get the command-line arguments. Remember that all argument validation happens inside get_args(). If this call succeeds, then we have good arguments from the user.
- ⑦ Generate the verses for the given args.num of days.
- ⑧ Join the verses on two newlines and print() to args.outfile which is an open file handle or sys.stdout.
- ⑨ Define a function to create any one verse from a given number 1-12.
- ⑩ The ordinal values for the numbers are a list of str values.
- ⑪ The gifts for the days is a list of str values.
- ⑫ The lines of each verse start off the same, substituting in the ordinal value of the given day.
- ⑬ I use the list.extend() method to add the gifts, which are a slice from the given day and then reversed().
- ⑭ Check if this is for a day greater than 1.
- ⑮ Change the last of the lines to add "And " to the beginning appended to the lowercased version of the line.
- ⑯ Return the lines joined on the newline.
- ⑰ The unit test for the verse() function.

13.3 Discussion

13.3.1 Defining the arguments

Nothing in the get_args() is new, so we'll throw a sidelong, cursory glace. The -num option is an int value with a default value of 12. The --outfile option is a str with a default of either None or the empty string because both will evaluate to False when we check later. If the --num value is not in range(1, 13) (remembering that 13 is not included, so just the numbers 1-12), I use parser.error() to show an error message and the short usage and exit with a non-zero exit value to indicate a failure. Notice that the error message provides feedback in showing the bad value and explaining exactly

what is allowed.

13.3.2 Making one verse

I chose to make a function called `verse()` to create any one verse given an `int` value of the day:

```
def verse(day):
    """Create a verse"""


```

ordinal = [index	day
'first'	0	1
'second'	1	2
'third'	2	3
'fourth'	3	4
'fifth'	4	5
'sixth'	5	6
'seventh'	6	7
'eighth'	7	8
'ninth'	8	9
'tenth'	9	10
'eleventh'	10	11
'twelfth'	11	12

I decided to use a list to represent the ordinal value of the day:

```
ordinal = [
    'first', 'second', 'third', 'fourth', 'fifth', 'sixth', 'seventh',
    'eighth', 'ninth', 'tenth', 'eleventh', 'twelfth'
]
```

Since the day is based on counting from 1 but Python lists start from 0, I have to subtract 1:

```
>>> day = 3
>>> ordinal[day - 1]
'third'
```

I could have just as easily used a dict:

```
ordinal = {
    1: 'first', 2: 'second', 3: 'third', 4: 'fourth',
    5: 'fifth', 6: 'sixth', 7: 'seventh', 8: 'eighth',
    9: 'ninth', 10: 'tenth', 11: 'eleventh', 12: 'twelfth',
}
```

Then you don't have to subtract 1:

```
>>> ordinal[3]
'third'
```

Whatever works for you. I also used a list for the gifts:

```
gifts = [
    'A partridge in a pear tree.',
    'Two turtle doves,',
    'Three French hens,',
    'Four calling birds,',
    'Five gold rings,',
    'Six geese a laying,',
    'Seven swans a swimming,',
    'Eight maids a milking,',
    'Nine ladies dancing,',
    'Ten lords a leaping,',
    'Eleven pipers piping,',
    'Twelve drummers drumming,',
]
```

```
gifts[:3]
['A partridge in a pear tree.', 0
 'Two turtle doves,', 1
 'Three French hens,', 2
 'Four calling birds,', 3
 'Five gold rings,', 4
 'Six geese a laying,', 5
 'Seven swans a swimming,', 6
 'Eight maids a milking,', 7
 'Nine ladies dancing,', 8
 'Ten lords a leaping,', 9
 'Eleven pipers piping,', 10
 'Twelve drummers drumming,', 11]
```



This makes a bit more sense as I can use a list slice to get the gifts for a given day:

```
>>> gifts[:3]
['A partridge in a pear tree.',
 'Two turtle doves,',
 'Three French hens,']
```

But I want them in reverse order. The `reversed()` function is lazy, so I need to use the `list` function to coerce the values in the REPL:

```
>>> list(reversed(gifts[:3]))
['Three French hens,',
 'Two turtle doves,',
 'A partridge in a pear tree.']
```

The first two lines of any verse are the same, substituting in the ordinal value for the day.

```
lines = [
    f'On the {ordinal[day - 1]} day of Christmas,',
    'My true love gave to me,'
```

]

I need to put these two lines together with the `gifts`. Since each verse is made of some number of lines, I think it will make sense to use a list to represent the entire verse. So I need to add the `gifts` to the `lines`, and I can use the `list.extend()` method to do that:

```
>>> lines.extend(reversed(gifts[:day]))
```

And now there are 5 lines:

```
>>> lines
['On the third day of Christmas',
 'My true love gave to me',
 'Three French hens',
 'Two turtle doves',
 'A partridge in a pear tree.']
>>> assert len(lines) == 5
```

Note that I cannot use the `list.append()` method. It's easy to confuse these two methods. The `list.extend()` method takes another list as the argument, expands it, and adds all of the individual elements to the original list. The `list.append()` method is meant to add one element to the list, so, if you give it a list, it will tack that entire list on to the end of the original list.

Here the `reversed()` iterator will be added the end of `lines` such that it would have three elements rather than the desired five:

```
>>> lines.append(reversed(gifts[:day]))
>>> lines
['On the third day of Christmas',
 'My true love gave to me',
 <list_reverseiterator object at 0x105bc8588>]
```

Maybe you're thinking you could coerce the `reversed()` with a `list`? Thinking you are, young Jedi, but, alas, that will still add a new list to the end:

```
>>> lines.append(list(reversed(gifts[:day])))
>>> lines
['On the third day of Christmas',
 'My true love gave to me',
 ['Three French hens', 'Two turtle doves', 'A partridge in a pear tree.']]
```

And we still have 3 lines rather than 5:

```
>>> len(lines)
3
```

If the `day` is greater than 1, I need to change the last line to say "And a" instead of "A":

```
if day > 1:
    lines[-1] = 'And ' + lines[-1].lower()
```

Note that this is another good reason to represent the `lines` as a list because the elements of a list are *mutable*. I could have represented the `lines` as a `str`, but strings

are *immutable*, so it would be much harder to change the last line.

I want to return a single `str` value from the function, so I join the `lines` on a newline:

```
>>> print('\n'.join(lines))
On the third day of Christmas,
My true love gave to me,
Three French hens,
Two turtle doves,
A partridge in a pear tree.
```

My function returns the joined lines and will pass the `test_verse()` function I provided.

13.3.3 Generating the verses

Given the `verse()` function, I can create all the needed verses by iterating from 1 to the given `--num`. I could collect them in `list` of `verses`:

```
day = 3
verses = []
for n in range(1, day + 1):
    verses.append(verse(n))
```

I can check that I have the right number of verses:

```
>>> assert len(verses) == day
```

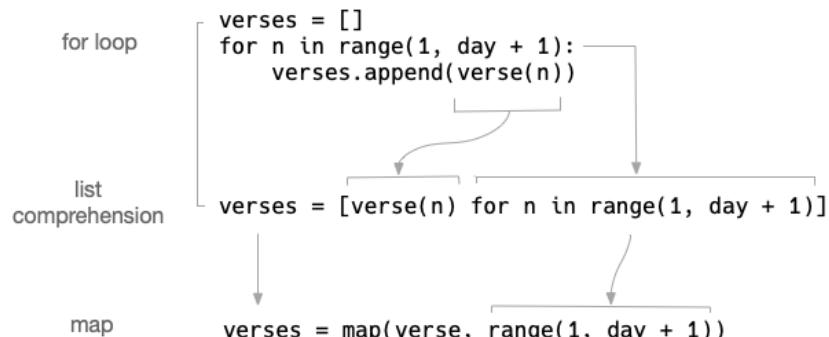
Whenever you see this pattern of creating an empty `str` or `list` and then using a `for` loop to add to it, consider instead using a list comprehension:

```
>>> verses = [verse(n) for n in range(1, day + 1)]
>>> assert len(verses) == day
```

I personally prefer using `map()` over list comprehensions. I will use the `list` function to coerce the lazy `map()` function here, but it's not necessary in the program code:

```
>>> verses = list(map(verse, range(1, day + 1)))
>>> assert len(verses) == day
```

Figure 13.1. Building a list using an for loop, a list comprehension, and map().



All of these methods will produce the correct number of verses. Choose whichever one makes the most sense to you.

13.3.4 Printing the verses

Just like with the "99 Bottles of Beer," I want to `print()` them with two newlines in between. The `str.join()` method is a good choice:

```
>>> print('\n\n'.join(verses))
On the first day of Christmas,
My true love gave to me,
A partridge in a pear tree.

On the second day of Christmas,
My true love gave to me,
Two turtle doves,
And a partridge in a pear tree.

On the third day of Christmas,
My true love gave to me,
Three French hens,
Two turtle doves,
And a partridge in a pear tree.
```

You can use the `print()` function with the optional `file` argument to put the text into an open file handle. The `args.outfile` value will be either the file indicated by the user or `sys.stdout`:

```
print('\n\n'.join(verses), file=args.outfile)
```

Or you can use the `fh.write` method, but you need to remember to add the trailing newline that `print()` adds for you:

```
args.outfile.write('\n\n'.join(verses) + '\n')
```

There are dozens to hundreds of ways to write this algorithm just as there are for "99 Bottles of Beer." If you came up with an entirely different approach which passed the test, that's terrific! Please share it with me. I wanted to stress the idea of how to write, test, and use a single `verse()` function, but I'd love to see other approaches!



13.4 Review

- There are many ways to encode algorithms to perform repetitive tasks. In my version, I wrote and tested a function to handle one task and then mapped a range of input values over that.
- The `range()` function will return `int` values between a given start and stop value, the latter of which is not included.
- You can use the `reversed()` function to reverse the values returned by `range()`.
- If we use `type=argparse.FileType('wt')` to define an argument with `argparse`, we get a file handle that is open for writing text.
- The `sys.stdout` file handle is always open and available for writing.
- Modeling the `gifts` as a `list` allowed me to use a list slice to get all the gifts for a given day. I used the `reversed()` function to put them into the right order for the song.
- I modeled the `lines` as a `list` because a `list` is mutable which I needed in order to change the last line when the day is greater than 1.
- Shadowing a variable or function is reusing an existing variable or function name. If, for instance, you create a variable with the name of an existing function, then the function is effectively hidden because of the shadow. Avoid shadowing by using tools like `pylint` to find these and many other common coding problems.

13.5 Going Further

- Install the `emoji` module (pypi.org/project/emoji/) and print various emojis for the gifts rather than text. For instance, you could use '`:bird:`' to print  for every "bird" like a hen or dove. I also used '`:man:`', '`:woman:`', and '`:drum:`', but you can use whatever you like:

```
On the twelfth day of Christmas,
My true love gave to me,
Twelve ♂s drumming,
Eleven ♀s piping,
Ten ♀s a leaping,
Nine ♀s dancing,
Eight ♀s a milking,
Seven ♀s a swimming,
Six ♀s a laying,
Five gold ♂s,
Four calling ♀s,
Three French ♀s,
Two turtle ♀s,
And a ♀ in a pear tree.
```

Rhymer: Using regular expressions to create rhyming words

In the movie *The Princess Bride*, the characters Inigo and Fezzik have a rhyming game they like to play, especially when their cruel boss, Vizzini, yells at them:

Inigo: That Vizzini, he can fuss.

Fezzik: I think he likes to scream at us.

Inigo: Probably he means no harm.

Fezzik: He's really very short on charm.



When I was writing the `alternate.txt` for the Gashlycrumb exercise, I would come up with a word like "cyanide" and wonder what I could rhyme with that. Mentally I start with the first consonant sound of the alphabet and start substituting "b" for "byanide," skip "c" because that's already the first character, then "d" for "dynanide," and so forth. This is effective but tedious, so I decided to write a program to do this for me, as one does.

This is basically another find-and-replace type of program that we've seen before like swapping all the numbers in a string in "Jump The Five" or all the vowels in a string in "Apples and Bananas." We wrote those programs using very manual, *imperative* methods like iterating through all the characters of a string, comparing them to some wanted value, and possibly returning a new value.

In the final solution for "Apples and Bananas," we briefly touched on "regular

expressions" (also called "regexes"¹²) which gives us a *declarative* way to describe a pattern of text. The material here may seem a bit of a reach, but I really want to help you dig into regexes to see what they can do!

In this exercise, we're going to take a given word and create "words" that rhyme. For instance, the word "bake" rhymes with words like "cake," "make," and "thrake," the last of which isn't actually a dictionary word but just a new string we create by replacing the "b" in "bake" with "thr." The algorithm we'll use is to split a word into any initial consonants and the rest of the word, so "bake" is split into "b" and "ake." We replace the "b" with all the other consonants from the alphabet plus these consonant clusters:



```
b1 br ch cl cr dr fl fr gl gr pl pr sc sh sk sl sm sn sp st
sw th tr tw thw wh wr sch scr shr sph spl spr squ str thr
```

Be sure the output is sorted. For instance, these are the first three words our program will produce for "cake":

```
$ ./rhymers.py cake | head -3
bake
blake
brake
```

And the last three:

```
$ ./rhymers.py cake | tail -3
xake
yake
zake
```

We'll replace any leading consonants with a list of other consonant sounds to create a total of 56 words:

```
$ ./rhymers.py cake | wc -l
56
```

Note that we'll replace *all* the leading consonants, not just the first one. For instance, with the word "chair" we need to replace "ch":

```
$ ./rhymers.py chair | tail -3
xair
yair
zair
```

¹² Pronounced with a hard "g" like in "George"

If a word like "apple" does not start with a consonant, then we'll append all the consonant sounds to the beginning to create words like "bapple" and "shrapple."

```
$ ./rhymer.py apple | head -3
bapple
blapple
brapple
```

Because there is no consonant to *replace*, words that start with a vowel will produce 57 rhyming words:

```
$ ./rhymer.py apple | wc -l
57
```

letters:

```
$ ./rhymer.py GUITAR | tail -3
xuitar
yuitar
zuitar
```

If a word contains *nothing but consonants*, then print a message that the word cannot be rhymed:

```
$ ./rhymer.py RDNZL
Cannot rhyme "RDNZL"
```

To make this a bit easier, the output should always be all lowercase even if the input has uppercase. The task of finding these initial consonants is made significantly easier with regexes.

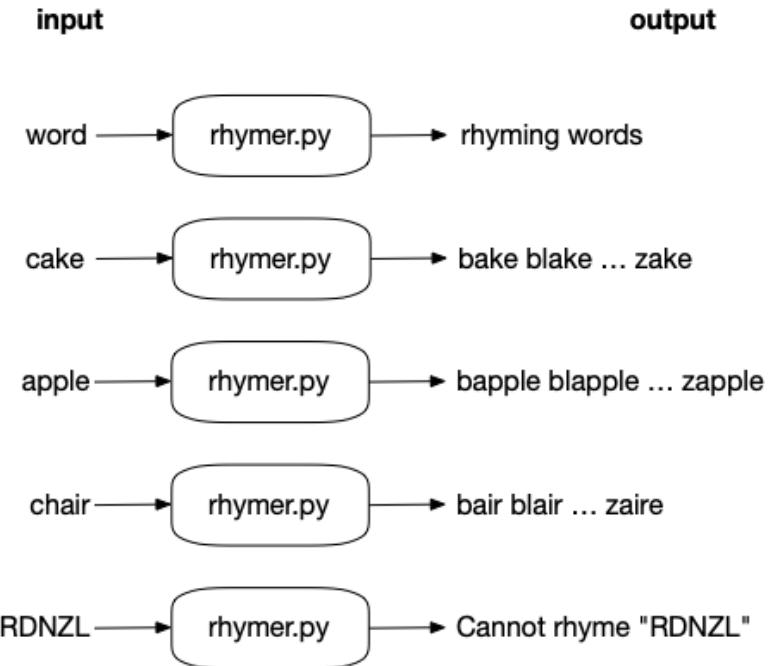


In this program, you will:

- Learn to write and use regular expressions
- Use a guard with a list comprehension
- Explore the similarities of list comprehension and guard to the `filter()` function
- Entertain ideas of "truthiness" when evaluating Python types in a boolean context

14.1 Writing `rhymer.py`

The program takes a single, positional argument which is the string to rhyme. Here is a snazzy, jazzy, frazzy, thwazzy string diagram:



If given no arguments or the `-h` or `--help` flags, it should print a usage statement:

```
$ ./rhymer.py -h
usage: rhymer.py [-h] word

Make rhyming "words"

positional arguments:
  word      A word to rhyme

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

14.1.1 Breaking a word

To me, the main problem of the program is breaking the given word into the leading consonant sounds and the rest (something like the "stem") of the word. I wrote a function called `stemmer()`, and this is my test for it:

```
def test_stemmer():
    """ Test stemmer """
    assert stemmer('') == ('', '')          ①
    assert stemmer('cake') == ('c', 'ake')   ②
    assert stemmer('chair') == ('ch', 'air') ③
    assert stemmer('APPLE') == ('', 'apple') ④
    assert stemmer('RDNZL') == ('rdnzl', '') ⑤
```

The tests cover the following good and bad inputs:

- ① The empty string.

- ② A word with a single leading consonant.
- ③ A word with a leading consonant cluster.
- ④ A word with no initial consonants. Also an uppercase word, so checking that lowercase is returned.
- ⑤ A word with no vowels.

My `stemmer()` function always returns a 2-tuple of the `(start, rest)` of the word, so the argument "cake" produces a tuple with two values of ('c', 'ake'). The argument "chair" is split into the leading "ch" and the "air" strings. The argument "APPLE" has no `start` and only the `rest` of the word, which is lowercase. A string with no vowels like "RDNZL" returns the (lowercased) string as the `start` and nothing for the `rest`.

The first and last tests use "bad" values that can't be rhymed. It's up to me as the programmer to decide what to do with those. Should my `stemmer()` function throw an exception or return `None` to indicate a failure? In this case, I decided to always return the same tuple and let the calling code deal with the problem. If there is nothing in the second position of the tuple, then there is nothing to rhyme. You don't have to make the same decisions I made. If you prefer to throw an exception for unrhymable words, then change your test to handle an exception.

14.1.2 Using regular expressions

It's certainly *possible* to write this program without regular expressions, but I hope you'll see how radically different using regexes can be from manually writing your own search-and-replace code. To start off, we need to bring in the `re` module:

```
>>> import re
```

I would encourage you to then read `help(re)` to get a feel for all that you can do with regexes. They are a deep subject with books¹³ and whole branches of academia devoted to them. There are many helpful websites that can help further explain regexes and even ones like regexr.com/ that can help you write them! We will only scratch the surface of what they can do.

We need to write a regex that will find consonants at the beginning of a string. We can define consonants as the characters of the English which are not the vowels, "a," "e," "i," "o," and "u." Our `stemmer()` will only return lowercase letters, and so there are only 21 consonants we need to define. You could write them out, but I'd rather write a bit of code!

I can start with `string.ascii_lowercase`:

```
>>> import string
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
```

And use a list comprehension with a "guard" clause to filter out the vowels. As I want a `str` of consonants and not a `list`, I use `str.join()` to make a new `str` value:

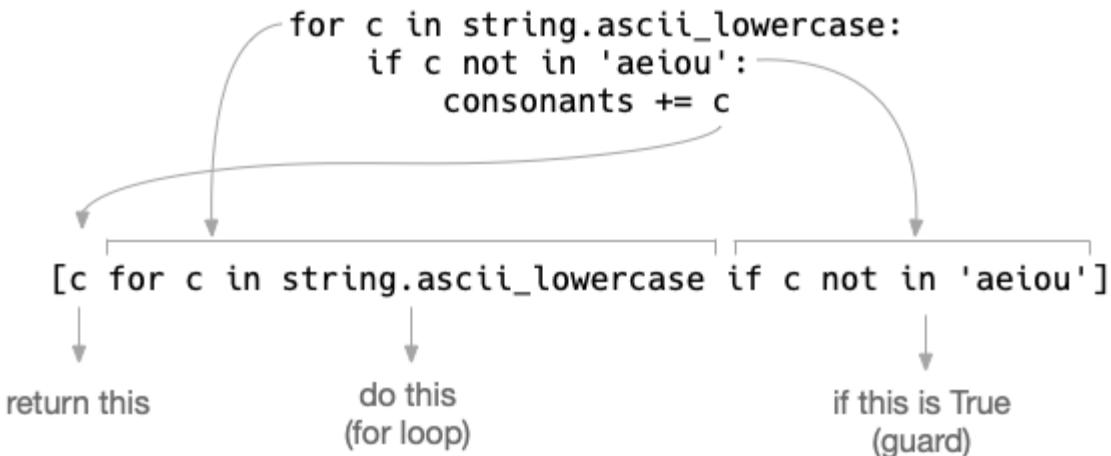
¹³ *Mastering Regular Expressions* by Jeffrey Friedl is one I would recommend

```
>>> consonants = ''.join([c for c in string.ascii_lowercase if c not in 'aeiou'])
>>> consonants
'bcdfghjklmnpqrstvwxyz'
```

The longer way to write this with a `for` loop and an `if` statement is this:

```
consonants = ''
for c in string.ascii_lowercase:
    if c not in 'aeiou':
        consonants += c
```

Figure 14.1. A `for` loop on top can be written as a list comprehension on bottom. This list comprehension includes a guard such that only the consonants are selected which is like the `if` statement above.



To use the consonants in a regex, I need to create a "character class" that includes all these values, which I can do by putting the characters inside square brackets:

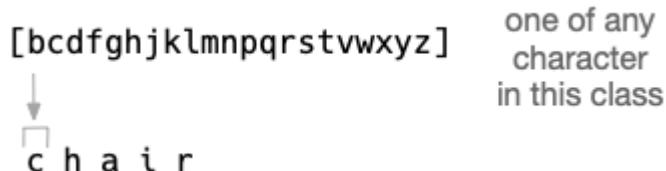
```
>>> pattern = '[' + consonants + ']'
>>> pattern
'[bcdfghjklmnpqrstvwxyz]'
```

The `re` module has two search-like functions called `re.match()` and `re.search()`, and I always get them confused. They both look for a pattern (the first argument) in some text, but the `re.match()` function starts *from the beginning* of the text while the `re.search()` function will match starting *anywhere* in the text.

As it happens, `re.match()` is just fine because we are looking for consonants at the beginning of a string.

```
>>> text = 'chair'
>>> re.match(pattern, text)
①
<re.Match object; span=(0, 1), match='c'> ②
```

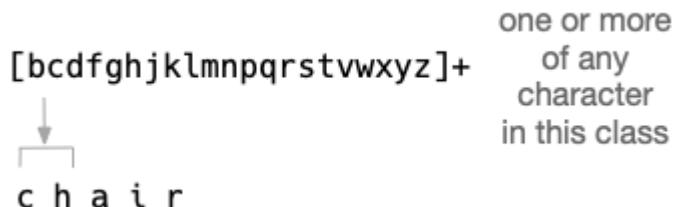
- ① Try to match the given pattern in the given text. If this succeeds, we get an `re.Match` object; otherwise, the value `None` is returned.
- ② The match was successful, so we see a "stringified" version of the `re.Match` object.



The `match='c'` shows us that the regular expression found the string 'c' at the beginning. Both the `re.match()` and `re.search()` functions will return an `re.Match` object on success. You can read `help(re.Match)` to learn more about all the cool things you can do with them:

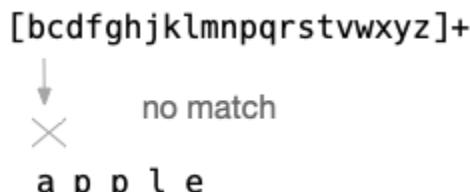
```
>>> match = re.match(pattern, text)
>>> type(match)
<class 're.Match'>
```

How do we get our regex to match the letters 'ch'? We can put a '+' sign after the character class to say we want *one or more*. (Does this sound a bit like `nargs='+'` to say one or more arguments?) I will use an f-string here to create the pattern:



```
>>> re.match(f'[{consonants}]+', 'chair')
<re.Match object; span=(0, 2), match='ch'>
```

What does it give us for a string with no leading consonants like "apple"?



```
>>> re.match(f'[{consonants}]+', 'apple')
```

It appears like we got nothing back from that. What is the `type()` of that return value?

```
>>> type(re.match(f'[{consonants}]+', 'apple'))
```

```
<class 'NoneType'>
```

Both the `re.match()` and `re.search()` functions return `None` to indicate a failure to match any text. We know that only some words will have a leading consonant sound, so this is not surprising. We'll see in a moment how to make this an optional match.

14.1.3 Using capture groups

It's all well and good to have found (or not) the leading consonants, but the goal here is to split the text into two parts: the consonants (if any) and the rest of the word. We can wrap parts of the regex in parentheses to create "capture groups." If the regex matches successfully, we can recover the parts using the `re.Match.groups()` method:

```
>>> match = re.match(f'([{consonants}]+)', 'chair')
>>> match.groups()
('ch',)
```

```
([bcdfghjklmnpqrstvwxyz]+)
↓
c h a i r
↓
('ch',)
```

To capture everything that comes after the consonants, we can use the `.` to match anything and add `+` to mean one or more. We put that into parens to capture it:

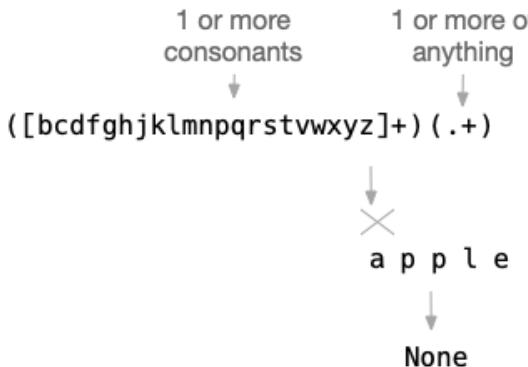
```
>>> match = re.match(f'([{consonants}]+)(.+)', 'chair')
>>> match.groups()
('ch', 'air')
```

```
1 or more           1 or more of
consonants          anything
↓                   ↓
([bcdfghjklmnpqrstvwxyz]+)(.+)
↓                   ↓
c h a i r
↓                   ↓
('ch', 'air')
```

What happens when we try to use this on "apple"? It fails to make the first match on the consonants, and so *the whole match fails* and returns `None`:

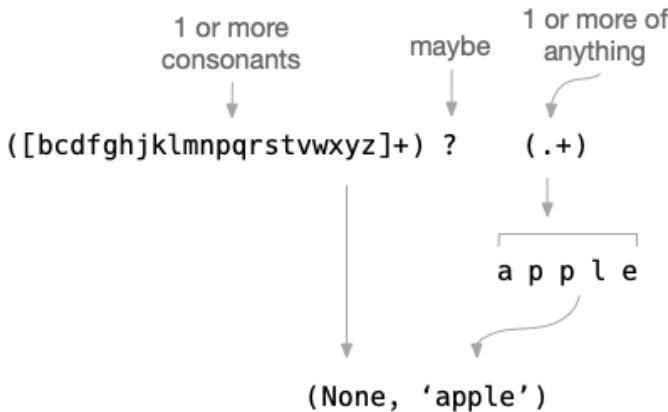
```
>>> match = re.match(f'([{consonants}]+)(.+)', 'apple')
>>> match.groups()
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'groups'
```



Remember that `re.match()` returns `None` when it fails to find the pattern. We can add `?` at the end of the consonants pattern to make it optional:

```
>>> match = re.match(f'([{consonants}]*)?(.+)', 'apple')
>>> match.groups()
(None, 'apple')
```



The `match.groups()` function returns a tuple containing the matches for each grouping created by the parentheses. You can also use the `match.group()` (singular) with a group number to get a specific group. Note that these start numbering from 1:

```
>>> match.group(1) ①
>>> match.group(2) ②
'apple'
```

- ① There was no match for the first group on "apple," so this is a `None`.
- ② The second group captured the entire word.

If you match on the "chair," there are values for both groups:

```
>>> match = re.match(f'([{consonants}]+)?(.+)', 'chair')
>>> match.group(1)
'ch'
>>> match.group(2)
'air'
```

So far we've only dealt with lowercase text because our program will always emit lowercase values. Still, let's explore what happens when we try to match uppercase text:

```
>>> match = re.match(f'([{consonants}]+)?(.+)', 'CHAIR')
>>> match.groups()
(None, 'CHAIR')
```

Not surprisingly, that fails. Our pattern only defines lowercase characters. We could add all the uppercase consonants, but it's a bit easier to use a third optional argument to `re.match()` to tell indicate case-insensitive searching:

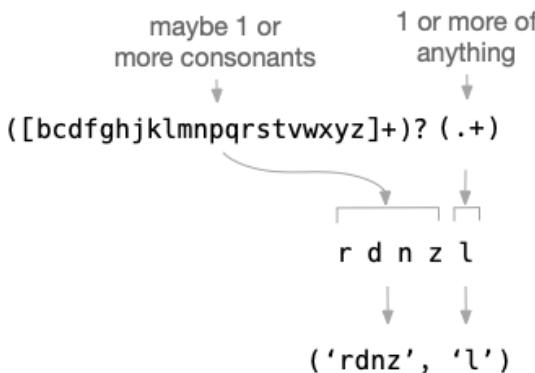
```
>>> match = re.match(f'([{consonants}]+)?(.+)', 'CHAIR', re.IGNORECASE)
>>> match.groups()
('CH', 'AIR')
```

Or you can force the text you are searching to lowercase:

```
>>> match = re.match(f'([{consonants}]+)?(.+)', 'CHAIR'.lower())
>>> match.groups()
('ch', 'air')
```

What do you get when you search on text that has nothing but consonants?

```
>>> match = re.match(f'([{consonants}]+)?(.+)', 'rdnzl')
>>> match.groups()
('rdnz', 'l')
```



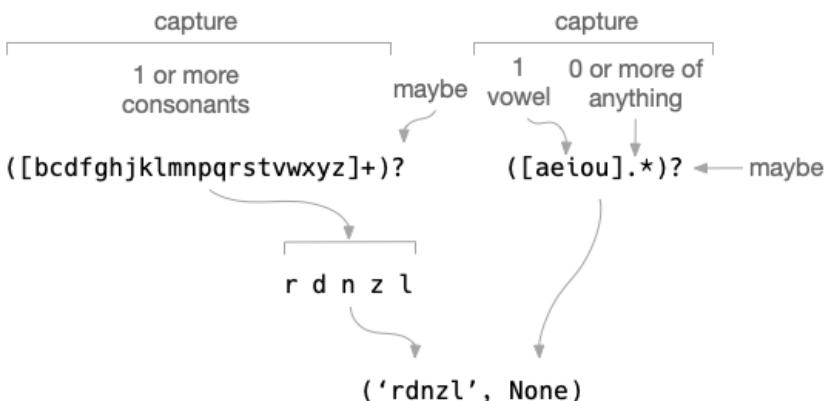
Were you expecting the first group to include *all* the consonants and the second group to have nothing? It might seem a bit odd that it decided to split off the "l" into the last group, but we have to think *extremely literally* about how the regex engine is working. We described an optional group of one or more consonants that *must be followed* by one or more of anything else. The "l" counts as one or more of anything else, so the regex matched exactly what we requested. If we change the `(.+)` to `(.*)` to make

it *zero or more*, then it works as expected:

```
>>> match = re.match(f'([{consonants}]+)?(.*)', 'rdnzl')
>>> match.groups()
('rdnzl', '')
```

Our regex is not quite complete. We want to indicate that there should be a vowel after the consonants, and then anything else. We can use another character class to describe the vowels, followed by *zero or more* (*) of anything (.). The whole second capture group is optional, so we add ?:

```
>>> match = re.match(f'([{consonants}]+)?([aeiou].*)?', 'rdnzl')
>>> match.groups()
('rdnzl', None)
```



14.1.4 Truthiness

The last thing is to return the correct values from the `stemmer()` function. I decided that I would always return a 2-tuple of (`start`, `rest`), and that I would always use the empty string to denote a missing value rather than a `None`:

```
return (match.group(1) or '', match.group(2) or '') if match else ('', '')
```

That may be a bit hard to read. Here is what it looks like split in two:

```
if match:          ①
    return (match.group(1) or '', match.group(2) or '') ②
else:            ③
    return ('', '') ④
```

- ① The `match` will be `None` if the regex failed which is "falsey." If it succeeds, then it will be "truthy."
- ② Return a tuple that has the first group match or the empty string and the second group match or the empty string.
- ③ Otherwise the regex failed.
- ④ Return a tuple of two empty strings. Theoretically this should never happen, but it makes sense to handle the possibility that our regex was unable to match anything at all.

In the first return, I'm using the `or` operator to decide between something on the left *or* something on the right. The `or` will return the first "truthy" value, the one that sort of-kind of evaluates to `True` in a boolean context:

```
>>> True or False      ①
True
>>> False or True     ②
True
>>> 1 or 0             ③
1
>>> 0 or 1             ④
1
>>> 0.0 or 1.0         ⑤
1.0
>>> '0' or ''          ⑥
'0'
>>> 0 or False         ⑦
False
>>> [] or ['foo']       ⑧
['foo']
>>> {} or dict(foo=1)  ⑨
{'foo': 1}
```

- ① It's easiest to see with literal `True` and `False` values.
- ② No matter the order, the `True` value will be taken.
- ③ In a boolean context, the `int` value `0` is "falsey" and any other value is "truthy."
- ④ The number values behave exactly like actual boolean values.
- ⑤ `float` values also behave like `int` values.
- ⑥ With `str` values, the empty string is the "falsey" and so anything else is "truthy." It may look odd because it returns `'0'` but that's not the *numeric* value zero but the *string* we use to represent the value of zero. Wow, such philosophical.
- ⑦ If no value is "truthy," then the last value is returned.
- ⑧ The empty `list` is "falsey," and so any non-empty `list` is "truthy."
- ⑨ The same is true of dictionaries. The empty `dict` is "falsey," and any non-empty `dict` is "truthy."

If you use the above regex and return in your `stemmer()`, it should pass the `test_stemmer()` test.

14.1.5 Creating the output

Let's review what the program should do:

- Take a positional string argument.
- Try to split it into two parts: any leading consonants and the rest of the word.
- If the split is successful, combine the "rest" of the word (which might actually be the entire word if there are no leading consonants) with all the other consonant sounds listed above. Be sure to *not* include the original consonant sound and to sort the rhyming strings.
- If you are unable to split the word, then print the message `Cannot rhyme "<word>"`.

Now it's time to write the program. Have fun storming the castle!

14.2 Solution

```
#!/usr/bin/env python3
"""Make rhyming words"""

import argparse
import re      ①
import string

# -----
def get_args():
    """get command-line arguments"""

    parser = argparse.ArgumentParser(
        description='Make rhyming "words"',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('word', metavar='word', help='A word to rhyme')

    return parser.parse_args()

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

    args = get_args()
    prefixes = list('bcdfghjklmnpqrstvwxyz') + ( ②
        'bl br ch cl cr dr fl fr gl gr pl pr sc ' ③
        'sh sk sl sm sn sp st sw th tr tw thw wh wr '
        'sch scr shr sph spl spr squ str thr').split()

    start, rest = stemmer(args.word) ④
    if rest: ⑤
        print('\n'.join(sorted([p + rest for p in prefixes if p != start]))) ⑥
    else:
        print(f'Cannot rhyme "{args.word}"') ⑦

# -----
def stemmer(word):
    """Return leading consonants (if any), and 'stem' of word"""

    vowels = 'aeiou'
    consonants = ''.join( ⑧
        [c for c in string.ascii_lowercase if c not in vowels]) ⑨
    pattern = ( ⑩
        '([' + consonants + ']+)?' # capture one or more consonants, optional
        '('                      # start capture
        '[' + vowels + ']'       # at least one vowel
        '.*'                     # zero or more of anything else
        ')?')                   # end capture, optional group
    match = re.match(pattern, word.lower()) ⑪
    return (match.group(1) or '', match.group(2) or '') if match else ('', '') ⑫

# -----
```

```

def test_stemmer(): ⑬
    """test the stemmer"""

    assert stemmer('') == ('', '')
    assert stemmer('cake') == ('c', 'ake')
    assert stemmer('chair') == ('ch', 'air')
    assert stemmer('APPLE') == ('', 'apple')
    assert stemmer('RDNZL') == ('rdnzl', '')

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

```

- ① The `re` module is for regular expressions.
- ② Get the command-line arguments.
- ③ Define all the prefixes that will be added to create rhyming words.
- ④ Split the word argument into two possible parts. Because the `stemmer()` function always returns a 2-tuple, we can unpack the values into two separate values.
- ⑤ Check if there is a part of the word that we can use to create rhyming strings.
- ⑥ If there is, use a list comprehension to iterate through all the prefixes and add them to the stem of the word. Use a guard to ensure that any given prefix is not the same as the beginning of the word. Sort all the values and print them, joined on newlines.
- ⑦ If there is nothing we can use to create rhymes, let the user know.
- ⑧ Since I will use the vowels more than once, I assign them to a variable.
- ⑨ The consonants are the letters that are not vowels. I will only match to lowercase letters.
- ⑩ The pattern is defined using consecutive literal strings that Python will join together into one string. By breaking up the pieces onto separate lines, I can comment on each part of the regular expression.
- ⑪ Use the `re.match()` function to start matching *at the beginning* of the the given word which I convert to lowercase.
- ⑫ Even though I would expect this pattern to never fail, I conservatively use an `if` expression to ensure that I have a value for the `match`. If I do, I return a 2-tuple consisting of the two capture groups, but I ensure that neither group is a `None` but rather an empty string to denote a missing value. If there is no `match`, I return a tuple of two empty strings.
- ⑬ The afore-described test function for the `stemmer()` function.

14.3 Discussion

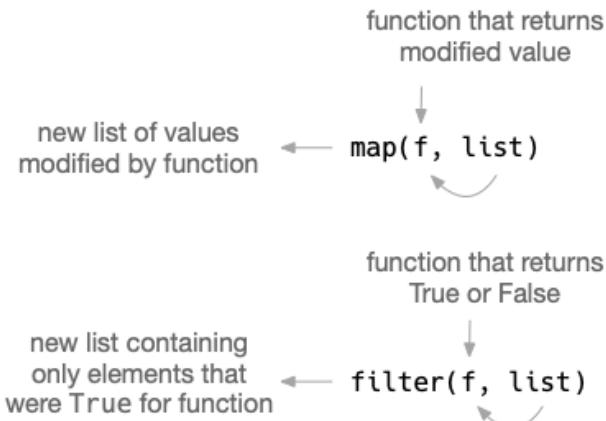
There are many ways you could have written this, but, as always, I wanted to break the problem down into some units I could write and test. For me, this came down to splitting the word into a possible leading consonant sound and the rest of the word. If I can manage that, I can create rhyming strings; if I cannot, then I need to alert the user.

14.3.1 Stemming a word

For the purposes of this program, the "stem" of a word is the part after any initial consonants which I define using a list comprehension with a guard to take only the letters that are not vowels:

```
>>> vowels = 'aeiou'
```

```
>>> consonants = ''.join([c for c in string.ascii_lowercase if c not in vowels])
```



I showed how this is a more concise way to write a `for` loop with an `if` statement. We've looked at `map()` several times now and talked about how it is a *higher-order function* (HOF) because it takes *another function* as the first argument, applying it to all the members of a sequence to produce a new, transformed sequence (like painting cars blue). Here I'd like to introduce another HOF called `filter()` which takes a function and some *iterable* (something that can be *iterated* like a list). As with `map()`, the function is applied to all the elements of the sequence. The return value of function will be evaluated by its "truthiness." Only those elements that evaluate as "truthy" will be returned by `filter()`.

Here is another way to write the idea of the list comprehension using a `filter()`:

```
>>> consonants = ''.join(filter(lambda c: c not in vowels, string.ascii_lowercase))
```

Just as with `map()`, I use the `lambda` keyword to create an *anonymous function*. The `c` is the name of the variable that will hold the argument which, in this case, will be each character from `string.ascii_lowercase`. The entire body of the function is the evaluation `c not in vowels`. Each of the vowels will return `False` for this:

```
>>> 'a' not in vowels
False
```

And each of the consonants will return `True`:

```
>>> 'b' not in vowels
True
```

Therefore only the consonants will be allowed to pass through the `filter()`. To think back to our "blue" cars, let's write a `filter()` that only accepts cars that start with the string "blue":

```
>>> cars = ['blue Honda', 'red Chevy', 'blue Ford']
>>> list(filter(lambda car: car.startswith('blue '), cars))
['blue Honda', 'blue Ford']
```

When the `car` variable has the value "red Chevy," the `lambda` returns `False`, and so that value is rejected:

```
>>> car = 'red Chevy'
>>> car.startswith('blue ')
False
```

Note that if none of the elements from the original iterable are accepted, then `filter()` will produce an empty list (`[]`). For example, I could `filter()` for numbers greater than 10. Note that `filter()` is another *lazy* function that I must coerce using the `list` function in the REPL:

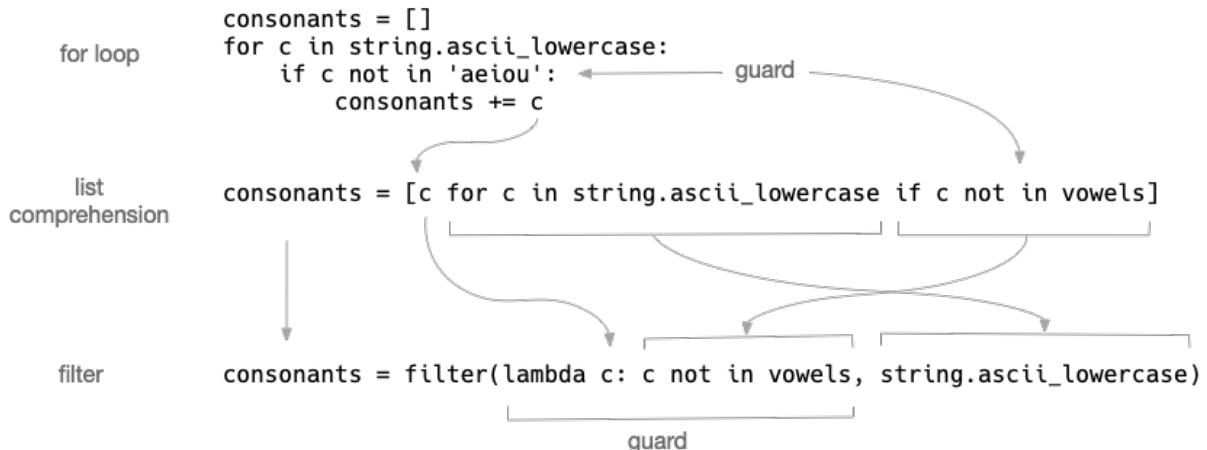
```
>>> list(filter(lambda n: n > 10, range(0, 5)))
[]
```

A list comprehension would also return an empty list:

```
>>> [n for n in range(0, 5) if n > 10]
[]
```

Here is a diagram showing the relationship of creating a new list called `consonants` using an imperative, `for`-loop approach, an idiomatic list comprehension with a guard, and a purely functional approach using `filter()`. All of these are perfectly acceptable, though the most Pythonic way is probably the list comprehension. The `for` loop would be very familiar to a C or Java programmer, while the `filter()` would be immediately recognizable to the Haskeller or even someone from a Lisp-like language. The `filter()` might be slower than the list comprehension, especially if the iterable were large. Choose whichever way makes more sense for your style and application.

Figure 14.2. The `filter()` function is similar to a list comprehension with a guard.



14.3.2 Writing a regular expression

We talked in the introduction about the individual parts of the regular expression I

ended up using. I'd like to take a moment to mention the way I formatted the regex in the code using an interesting trick of the Python interpreter that will stitch together string literals into one long string if they are enclosed in a grouping like parentheses:

```
>>> this_is_just_to_say = ('I have eaten '
... 'the plums '
... 'that were in '
... 'the icebox')
>>> this_is_just_to_say
'I have eaten the plums that were in the icebox'
```

Note that there are no commas after each string as that would create a tuple with 4 individual strings:

```
>>> this_is_just_to_say = ('I have eaten ',
... 'the plums ',
... 'that were in ',
... 'the icebox')
>>> this_is_just_to_say
('I have eaten ', 'the plums ', 'that were in ', 'the icebox')
```

The advantage of writing out the regular expression like this is to add comments to each important part:

```
pattern = (
    '([' + consonants + ']+)?' # capture one or more consonants, optional
    '('                      # start capture
    '[' + vowels + ']'       # at least one vowel
    '.*?'                   # zero or more of anything else
    ')?')'                  # end capture, optional group
```

I could have written the entire regex on one line. Ask yourself which version would you rather read and maintain:¹⁴

```
pattern = '([' + consonants + ']+)?([' + vowels + '].*)?'
```

I really worked hard on creating the regular expression, and I have a good bit of confidence that this regex will never fail to match something, even the empty string, because both elements (the leading consonant cluster and the rest of the word) are optional. Still, I will very conservatively call the `re.match()` so I can check the return value which I called `match`. Note that I lowercase the word:

```
match = re.match(pattern, word.lower())
```

It's possible my function might receive some value which might cause the `re.match()` to return a `None`. I use the `match` value in an `if` expression to be absolutely sure that I can safely call `match.group()`. As stated in the introduction, I would prefer to use the empty string rather than `None` to indicate a missing group, so I use the `or` operator to choose the first "truthy" value for both groups. If there is no `match`, I return a 2-tuple of empty strings:

¹⁴ Looking at code you wrote more than two weeks ago is like looking at code you are seeing for the first time." - Dan Hurvitz

```
return (match.group(1) or '', match.group(2) or '') if match else ('', '')
```

14.3.3 Testing and using the stemmer() function

I first introduced the `test_stemmer()` so we could think about what values we might pass in, both good and bad, and what we might expect to receive:

```
def test_stemmer():
    """test the stemmer"""
    assert stemmer('') == ('', '')
    assert stemmer('cake') == ('c', 'ake')
    assert stemmer('chair') == ('ch', 'air')
    assert stemmer('APPLE') == ('', 'apple')
    assert stemmer('RDNZL') == ('rdnzl', '')
```

I really recommend that you always write a `test_x` function *before* you write the `x` function as it can help you think of the function as this black box. Something goes in, something comes out, and what happens inside is a separate concern.

One of the very interesting things about Python code is that your `rhymer.py` program is also — kind of, sort of — a *module* of code. That is, you haven't explicitly written it to be this container of reusable functions, but it is. You can even run the functions from inside the REPL. For this to work, be sure you run `python3` inside the same directory as the `rhymer.py` code:

```
>>> from rhymer import stemmer
```

And now you can run and test your `stemmer()` function manually:

```
>>> stemmer('apple')
('', 'apple')
>>> stemmer('banana')
('b', 'anana')
```

The deeper meaning of `if __name__ == '__main__':`

Note that if you were to change the last two lines of `rhymer.py` from this:

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

To this:

```
main()
```

Then the `main()` function would be run when you try to import the module!:

```
>>> from rhymer import stemmer
usage: [-h] str
: error: the following arguments are required: str
```

So don't do that.

If you don't explicitly import a function, then you can use the fully qualified function

name by adding the module name to the front:

```
>>> import rhymer
>>> rhymer.stemmer('apple')
('', 'apple')
>>> rhymer.stemmer('banana')
('b', 'anana')
```

There are many advantages to writing many small functions rather than long, sprawling programs. One is that small functions are much easier to write, understand, and test. Another is that you can put your tidy, tested functions into modules and share them across different programs you write. As you write more and more programs, you will find yourself solving some of the same problems repeatedly. It's far better to create modules with reusable code rather than copying pieces from one program to another. If you ever find a bug in a shared function, you can fix it once and all the programs sharing the function get the fix. The other way is to find the duplicated code in every program and change it (and hoping that this doesn't introduce even more problems because the code is entangled with other code!).



14.3.4 Creating rhyming strings

The `stemmer()` function should always return a 2-tuple of the `(start, rest)` of a given word. As such, I can unpack the two values into separate variables:

```
>>> start, rest = stemmer('cat')
>>> start
'c'
>>> rest
'at'
```

If there is a value for `rest`, I can add all my prefixes to the beginning:

```
>>> prefixes = list('bcdfghjklmnpqrstvwxyz') +
...     'bl br ch cl cr dr fl fr gl gr pl pr sc '
...     'sh sk sl sm sn sp st sw th tr tw wh wr'
...     'sch scr shr sph spl spr squ str thr').split()
```

I decided to use another list comprehension with a guard to skip any prefix that is the same as the `start` of the word. The result will be a new list which I pass to the `sorted` function to get the correctly ordered strings:

```
>>> sorted([p + rest for p in prefixes if p != start])
['bat', 'blat', 'brat', 'chat', 'clat', 'crat', 'dat', 'drat', 'fat',
 'flat', 'frat', 'gat', 'glat', 'grat', 'hat', 'jat', 'kat', 'lat',
```

```
'mat', 'nat', 'pat', 'plat', 'prat', 'qat', 'rat', 'sat', 'scat',
'schat', 'scrat', 'shat', 'shrat', 'skat', 'slat', 'smat', 'snat',
'spat', 'sphat', 'splat', 'sprat', 'squat', 'stat', 'strat', 'swat',
'tat', 'that', 'thrat', 'thwat', 'trat', 'twat', 'vat', 'wat',
'what', 'wrat', 'xat', 'yat', 'zat']
```

I then `print()` that list, joined on newlines. If there is no rest of the given word, I `print()` a message that the word cannot be rhymed:

```
if rest:
    print('\n'.join(sorted([p + rest for p in prefixes if p != start])))
else:
    print(f'Cannot rhyme "{args.word}"')
```

14.3.5 Writing `stemmer()` without regular expressions

It is certainly possible to write a solution that does not use regular expressions. My idea was to find the first position of a vowel in the given string. If one is present, use a list slice to return the portion of the string up to that position and the portion starting at that position:

```
def stemmer(word):
    """Return leading consonants (if any), and 'stem' of word"""
    word = word.lower()          ①
    vowel_map = map(lambda c: word.index(c) if c in word else -1, 'aeiou') ②
    pos = list(filter(lambda v: v >= 0, vowel_map)) ③

    if pos:
        first = min(pos)      ④
        return (word[:first], word[first:]) ⑤
    else:
        return (word, '')       ⑥
```

- ① Lowercase the given word to avoid dealing with uppercase letters.
- ② `map()` over the vowels 'aeiou' to get their index if they are present in the given word. Use -1 to indicate the vowel is not present.
- ③ Use `filter()` to allow only values greater than 0. The value -1 indicates "not found" as 0 is a valid index for a string. We use `list()` to coerce the values because `filter()` is lazy.
- ④ The `pos` variable is a `list`. In a boolean context, the empty list evaluates to "falsey" and anything else is "truthy."
- ⑤ The first position of a vowel will be the *minimum* from the positions (`min(pos)`).
- ⑥ Use the `first` to slice the portion of `word` up to (but not including) `first` and then the portion from `first` to the end.
- ⑦ If `pos` is the empty list (meaning no vowels were present), then return a 2-tuple of the word and the empty string to indicate there is no rest of the word to use for rhyming.

This function will also pass the `test_stemmer()` function. By writing a test just for the idea of this one function and exercising it with all the different values I would expect, I'm free to *refactor* my code. As stated before, the `stemmer()` is a black box. What goes on inside the function is of no concern to the code that calls it. As long as the function passes the tests, then it is "correct" (for certain values of "correct").

Small functions and their *tests* will set you free to improve your programs! First make something work, and make it beautiful. Then try to make it better, using your tests to ensure it keeps working as expected.



14.4 Review

- Regular expressions allow us to declare a pattern that we wish to find. The `regex engine` will sort out whether the pattern is found or not. This is a *declarative* approach to program rather than the *imperative* method of manually seeking out patterns by writing code ourselves.
- We can wrap parts of the pattern in parentheses to "capture" them into groups that we can fetch from the result of `re.match()` or `re.search()`.
- You can add a guard to a list comprehension to avoid taking some elements from an iterable.
- The `filter()` function is another way to write a list comprehension with a guard. Like `map()`, it is a lazy, higher-order function that takes a function that will be applied to every element of an iterable. Only those elements which are "truthy" are returned.
- Python can evaluate many types including strings, numbers, lists, and dictionaries in a *boolean context* to arrive at a sense of "truthiness." That is, you are not restricted to just `True` and `False` in `if` expressions. The empty string `''`, the `int 0`, the `float 0.0`, the empty `list []`, and the empty `dict {}` are all considered "falsey," and so any value from those types like the non-empty `str`, `list` or `dict`, or any numeric value not zero-ish is considered "truthy."
- You can break long string literals into shorter strings in your code and then group them with parentheses to have Python join them into one long string. It's advisable to break long regexes into shorter strings and add comments on each line to document the function of each pattern.

- Write small functions *and tests* and share them in modules. Every .py file can be a module from which you can import functions. Sharing small, tested functions is better than writing long programs and copying/pasting code as needed.

14.5 Going Further

- Add an --output option to write the words to a given file. The default should be to write to STDOUT.
- Read an input file and create rhyming words for all the words in the file. You can borrow from the "Words Count" program to read a file and break it into words, then iterate each word and create an output file for each word with the rhyming words.
- Write a new program that find all unique consonant sounds in a dictionary of English words like the file /usr/share/dict/words that usually exists on most Unix systems. Print them out in alphabetical order and use those to expand this program's consonants.
- Write a program to create Pig Latin where you move the initial consonant sound from the beginning of the word to the end and add "-ay" so that "cat" becomes "atcay." If a word starts with a vowel, then add "-yay" to the end so that "apple" becomes "apple-yay."
- Write a program to create spoonerisms where the initial consonant sounds of adjacent words are switched so you get "blushing crow" instead of "crushing blow."

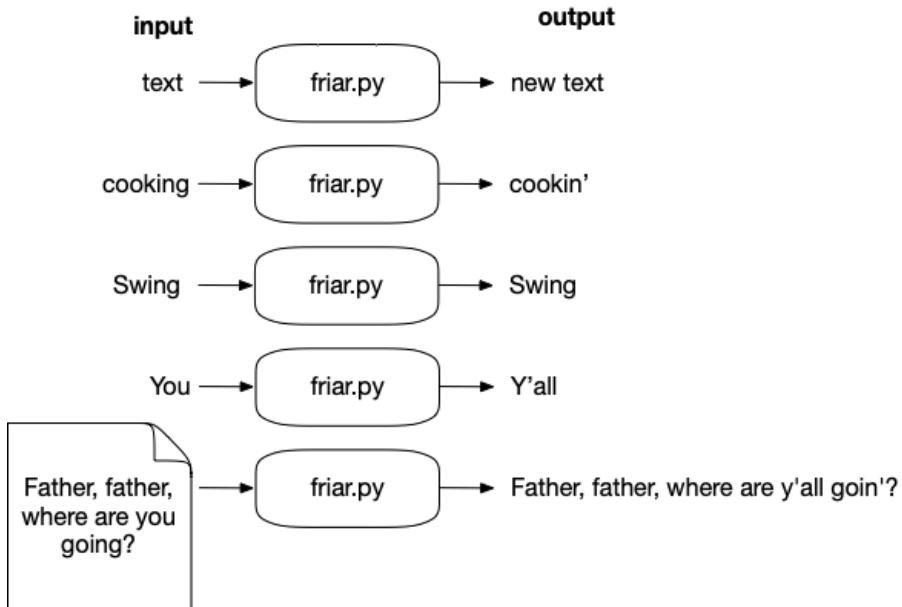
15

The Kentucky Friar: More regular expressions

Your author grew up in the American Deep South where we tend to drop the final "g" of words ending in "ing," like "cookin'" instead of "cooking." We also tend to say "y'all" for the second-person, plural pronoun, which makes sense because Standard English is missing a distinctive word for this. In this exercise, we'll write a program called `friar.py` that will accept some input as a single positional argument and transforms the text by replacing the "g" with an apostrophe ('') words ending in "-ing" and also changes "you" to "y'all". Granted, we have no way to know if we're changing the first- or second-person "you," but it makes for a fun challenge, nonetheless.



Here is a string diagram to help you see the inputs and outputs:



When run with no arguments or the `-h` or `--help` flags, your program should present usage:

```
$ ./friar.py -h
usage: friar.py [-h] text

Southern fry text

positional arguments:
  text      Input text or file

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

We will only change "-ing" words with *two syllables*, so "cooking" becomes "cookin'" but "swing" will stay the same. Our heuristic for identifying two-syllable "-ing" words is to inspect the part of the word before the "-ing" ending to see if it contains a vowel, which in this case will include "y." We can split "cooking" into "cook" and "ing"; since there is an "o" in "cook," we should drop the final "g":

```
$ ./friar.py Cooking
Cookin'
```

When we remove "ing" from "swing," though, we're left with "sw" which contains no vowel, so it will remain the same:

```
$ ./friar.py swing
swing
```

Be mindful to keep the case the same on the first letter, e.g., "You" should become "Y'all":

```
$ ./friar.py you
y'all
$ ./friar.py You
Y'all
```

As in several previous exercises, the input may name a file, in which case you should read the file for the input text. To pass the tests, you will need to preserve the line structure of the input, so I recommend you read the file line-by-line. So given this input:

```
$ head -2 inputs/banner.txt
O! Say, can you see, by the dawn's early light,
What so proudly we hailed at the twilight's last gleaming -
```

The output should have the same line breaks:

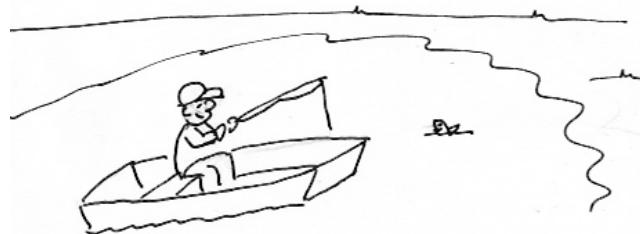
```
$ ./friar.py inputs/banner.txt | head -2
O! Say, can y'all see, by the dawn's early light,
What so proudly we hailed at the twilight's last gleamin' -
```

To me, it's quite amusing to transform texts this way, but maybe I'm just weird:

```
$ ./friar.py inputs/raven.txt
Presently my soul grew stronger; hesitatin' then no longer,
"Sir," said I, "or Madam, truly your forgiveness I implore;
But the fact is I was nappin', and so gently y'all came rappin',
And so faintly y'all came tappin', tappin' at my chamber door,
That I scarce was sure I heard y'all" - here I opened wide the door: -
Darkness there and nothin' more.
```

In this exercise, you will:

- Learn more about using regular expressions
- Use both `re.match()` and `re.search()` to find patterns anchored to the beginning of a string or anywhere in the string, respectively.
- Learn how the `$` symbol in a regex anchors a pattern to the *end* of a string.
- Learn how to use `re.split()` to split a string.
- Explore how to write a manual solution for finding two-syllable "ing" words or the word "you."



15.1 Writing friar.py

As usual, I would recommend you start with `new.py` `friar.py` or copy the

template/template.py to 15_friar/friar.py. I recommend you start with a simple version that echos back the input from the command line:

```
$ ./friar.py cooking
cooking
```

Or a file (use your best Scarlett O'Hara voice):

```
$ ./friar.py inputs/blake.txt
Father, father, where are y'all goin'?
Oh do not walk so fast!
Speak, father, speak to your little boy,
Or else I shall be lost.
```

We need to process the input line-by-line, then word-by-word. You can use the `str.splitlines()` method to get each line of the input, and then the `str.split()` method to break the line on spaces into word-like units. This code:

```
for line in args.text.splitlines():
    print(line.split())
```

Will create this output:

```
$ ./friar.py tests/blake.txt
['Father,', 'father,', 'where', 'are', 'you', 'going?']
['Oh', 'do', 'not', 'walk', 'so', 'fast!']
['Speak,', 'father,', 'speak', 'to', 'your', 'little', 'boy,']
['Or', 'else', 'I', 'shall', 'be', 'lost.']
```

If you look closely, it's going to be difficult to handle some of these word-like units as the adjacent punctuation is still attached to the words as in 'Father,' and 'going?'. We could write a regular expression to find just the parts of each "word" composed of ASCII letters, but I'd like to show you another way to split the text *using a regular expression!*

15.1.1 Splitting text using regular expressions

As in "Rhymer," we need to `import re` to use regexes:

```
>>> import re
```

For demonstration purposes, I'm going to set `text` to the first line:

```
>>> text = 'Father, father, where are you going?'
```

By default, `str.split()` breaks text on spaces. Note that whatever text is used for splitting is missing from the result, so here there are no spaces:

```
>>> text.split()
['Father,', 'father,', 'where', 'are', 'you', 'going?']
```

You can pass an optional value to `str.split()` to indicate the string you want to use for splitting. If we choose the comma, then we end up with 3 strings instead of 6. Note that there are no commas in the resulting list as that is the argument to `str.split()`:

```
>>> text.split(',')
['Father', ' father', ' where are you going?']
```

The `re` module has a function called `re.split()` that works similarly. I recommend you read `help(re.split)` as this is a very powerful and flexible function. Like `re.match()` that we used in "Rhymer," this function wants at least a pattern and a string. We can use `re.split()` with a comma to get the same output as `str.split()`, and, as before, the commas are missing from the result:

```
>>> re.split(',', text)
['Father', ' father', ' where are you going?']
```

15.1.2 Shorthand classes

We are after the things that look like "words" in that they are composed of the characters that normally occur in words. The characters that *don't* normally occur in words (things like punctuation) are what we want to use for splitting. We've seen before that we can create a *character class* by putting literal values inside square brackets, like `'[aeiou]'` for the vowels. What if we create a character class where we enumerate all the non-letter characters? We could do something like this:

```
>>> import string
>>> ''.join([c for c in string.printable if c not in string.ascii_letters])
'0123456789!#$%&\()'*)+,-./:;<=>?@[\\]^_`{|}~ \t\n\r\x0b\x0c'
```

That won't be necessary because almost every implementation of regular expression engines define shorthand character classes.¹⁵ Here is a table of some of the most common shorthand classes and how they can be written longhand:

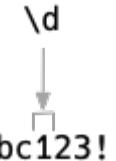
Table 15.1. Regex shorthand classes

Character class	Shorthand	Other ways to write
Digits	\d	[0123456789], [0-9]
Whitespace	\s	[\t\n\r\x0b\x0c], same as <code>string.whitespace</code>
Word characters	\w	[a-zA-Z0-9_-]

The shorthand `\d` means any *digit* and is equivalent to `'[0123456789]'`. I can use the `re.search()` method to look anywhere in a string for any digit. Here it will find the character '1' in the string 'abc123!' because this is the first digit in the string:

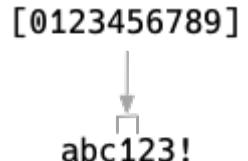
¹⁵ There is a basic flavor of regular expression syntax that is recognized by everything from Unix command-line tools like `awk` to regex support inside of languages like Perl, Python, and Java. Some tools add extensions to their regexes that may not be understood by other tools. For example, there was a time when Perl's regex engine added many new ideas that eventually became a dialect known as "PCRE" (Perl-Compatible Regular Expressions). Not every tool that understands regexes will understand every flavor of regex, but, in all my years of writing and using regexes, I've rarely found this to be a problem.

```
>>> re.search('\d', 'abc123!')
<re.Match object; span=(3, 4), match='1'>
```



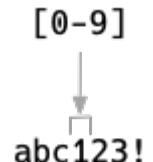
Which is the same as using the longhand version:

```
>>> re.search('[0123456789]', 'abc123!')
<re.Match object; span=(3, 4), match='1'>
```



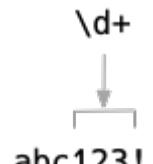
Or the version that uses the range of characters '[0-9]':

```
>>> re.search('[0-9]', 'abc123!')
<re.Match object; span=(3, 4), match='1'>
```



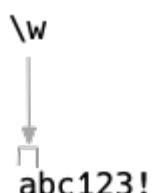
To have it find *one or more digits in a row*, add the +:

```
>>> re.search('\d+', 'abc123!')
<re.Match object; span=(3, 6), match='123'>
```



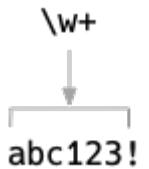
The \w shorthand means "any word-like character." It includes all the Arabic numbers, the letters of the English alphabet, the dash ('-'), and underscore ('_'). The first match in the string is 'a':

```
>>> re.search('\w', 'abc123!')
<re.Match object; span=(0, 1), match='a'>
```



If you add the +, then it matches one or more word characters in a row which includes abc123 but not the !:

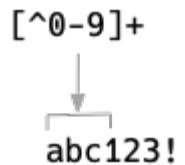
```
>>> re.search('\w+', 'abc123!')
<re.Match object; span=(0, 6), match='abc123'>
```



15.1.3 Negated shorthand classes

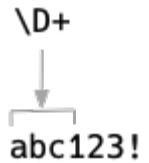
You can complement or "negate" a character class by putting the caret \wedge *immediately inside* the character class. One or more of any character *not* a digit is $[^\wedge0-9]^+$, and so 'abc' is found:

```
>>> re.search('[^\w-9]^+', 'abc123!')
<re.Match object; span=(0, 3), match='abc'>
```



The class $[^\wedge0-9]^+$ can also be written as \D+:

```
>>> re.search('\D+', 'abc123!')
<re.Match object; span=(0, 3), match='abc'>
```



Therefore, anything *not* a word character is \W:

```
>>> re.search('\W', 'abc123!')
<re.Match object; span=(6, 7), match='!'>
```

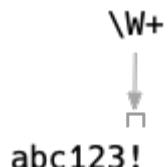


Table 15.2. Negated regex shorthand classes

Character class	Shorthand	Other ways to write
Not a digit	\D	[^\wedge0123456789], [^\w-9]
Not whitespace	\S	[^\t\n\r\x0b\x0c]
Word characters	\W	[^\w-9_]

15.1.4 Using `re.split()` with a captured regex

We can use `\W` as the argument to `re.split()`:

```
>>> re.split('\W', 'abc123!')
['abc123', '']
```

There is a problem, though, because the result of `re.split()` *omits those strings matching the pattern*. Here we've lost the non-word character '!'. If we read `help(re.split)` closely, we can find the solution:

If *capturing parentheses are used in pattern*, then the text of all groups in the pattern are also returned as part of the resulting list.

We used capturing parentheses in the "Rhymer" to tell the regex engine to "remember" certain patterns that we then use `re.Match.groups()` to retrieve the groups after a match. Here it means that we need to put the split pattern in parens:

```
>>> re.split('(\W)', 'abc123!')
['abc123', '!', '']
```

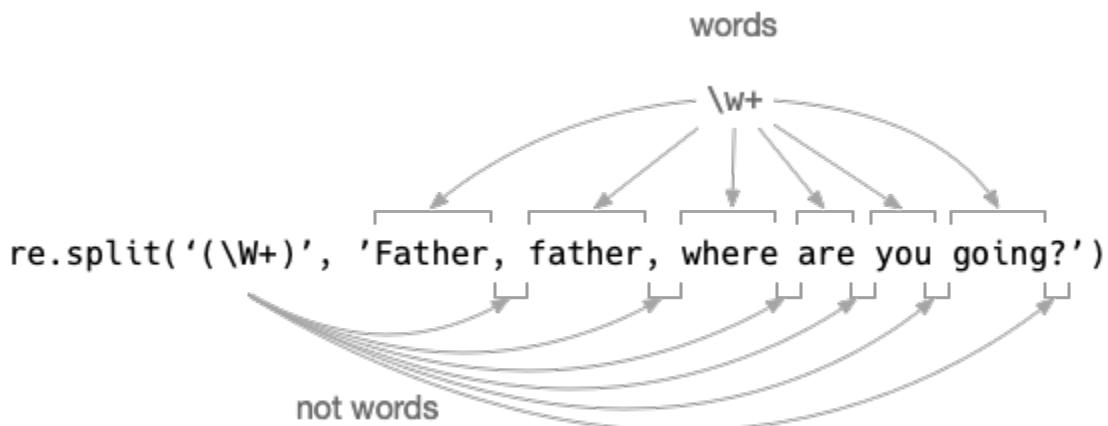
If we try that on our text, we get both the pieces of text matching and not matching the regular expression:

```
>>> re.split('(\W)', text)
['Father', ',', '.', ' ', 'father', ',', '.', ' ', 'where', ' ', 'are', ' ', 'you', ' ',
'going', '?', '']
```

I'd like to group all the non-word characters together by adding `+` to the regex:

```
>>> re.split('(\W+)', text)
['Father', ' ', ' ', 'father', ' ', ' ', 'where', ' ', 'are', ' ', 'you', ' ', 'going', '?', '']
```

Figure 15.1. The `re.split()` function can use a captured regex to return both the parts that match the regex and those that did not.



That is so cool! Now we have a way to process each *actual* word and the bits in between them.

15.1.5 Writing the `fry()` function

For me, the next step is to write function called `fry()` to decide whether to modify *just one word*. That is, rather than thinking about how to handle all the text at once, I just want to think about how I will handle one word at a time. To help me think about how this function should work, I'll start off by writing the `test_fry()` function and a stub for the actual `fry()` function that has just the single command `pass` which tells Python to do nothing. If you like this idea, feel free to paste this into your program:

```
def fry(word):
    pass      ①

def test_fry(): ②
    assert fry('you') == "y'all"      ③
    assert fry('You') == "Y'all"      ④
    assert fry('fishing') == "fishin'" ⑤
    assert fry('Aching') == "Achin'"   ⑥
    assert fry('swing') == "swing"     ⑦
```

- ① `pass` is way to do nothing. You might call it a "no-operation" or "NO-OP" which kind of looks like "NOPE" which is another way to remember it does nothing. We're just defining a `fry()` function as a placeholder so we can write the test.
- ② The `test_fry()` function will pass in words we expect to be changed or not. We can't check every word, so we'll rely on spot-checking the major cases.
- ③ The word "you" should become "y'all."
- ④ Ensure the word's capitalization is preserved.
- ⑤ This is a two-syllable -ing word that should be changed by dropping the final "g" for an apostrophe.
- ⑥ This is a two-syllable -ing word that starts with a vowel. It should likewise be changed.
- ⑦ This is a one-syllable -ing that should not be changed.

Then run `pytest friar.py` to see that, as expected, the test will fail:

```
===== FAILURES =====
test_fry

def test_fry():
>     assert fry('you') == "y'all" ①
E     assert None == "y'all"       ②
E     + where None = fry('you')

friar.py:47: AssertionError
===== 1 failed in 0.08 seconds =====
```

- ① The first test is failing.
- ② The result of `fry('you')` was `None` which does not equal "y'all."

Let's change our `fry()` function to handle that string:

```
def fry(word):
```

```
if word == 'you':
    return "y'all"
```

And run our tests:

```
===== FAILURES =====
test_fry

def test_fry():
    assert fry('you') == "y'all" ①
>     assert fry('You') == "Y'all" ②
E     assert None == "Y'all"      ③
E     + where None = fry('You')

friar.py:49: AssertionError
===== 1 failed in 0.16 seconds =====
```

- ① Now the first test passes.
- ② The second test fails because the "You" is capitalized.
- ③ The function returned None but should have returned "Y'all."

Let's handle that:

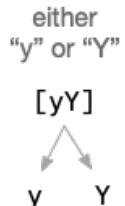
```
def fry(word):
    if word == 'you':
        return "y'all"
    elif word == 'You':
        return "Y'all"
```

If you run the test, you'll see the first two tests pass now; however, I'm definitely not happy with that solution. There is already a good bit of duplicated code. Can we write a more elegant way to match both "you" and "You" and which returns the correctly capitalized answer? Yes, we can!

```
def fry(word):
    if word.lower() == 'you':
        return word[0] + "'all"
```

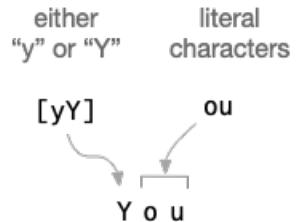
Better still, we can write a regular expression! There is one difference between "you" and "You" — the "y" or "Y" — that we can represent using the character class '[yY]'. This will match the lowercase version:

```
>>> re.match('[yY]ou', 'you')
<re.Match object; span=(0, 3), match='you'>
```



And the capitalized:

```
>>> re.match('[yY]ou', 'You')
<re.Match object; span=(0, 3), match='You'>
```



Now I'd like to reuse the initial character (either "y" or "Y") in the return value. I could *capture* it by placing it into parentheses. Try to rewrite your fry() function using this idea and getting it to pass the first two tests again before moving on:

```
>>> match = re.match('([yY])ou', 'You')
>>> match.group(1) + "all"
"Y'all"
```

The next step is to handle a word like "fishing":

```
===== FAILURES =====
test_fry

def test_fry():
    assert fry('you') == "y'all"
    assert fry('You') == "Y'all"
>   assert fry('fishing') == "fishin'" ①
E   assert None == "fishin'"           ②
E       + where None = fry('fishing')

friar.py:52: AssertionError
===== 1 failed in 0.10 seconds =====
```

① The third test fails.

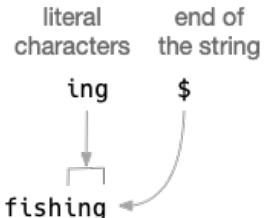
② The return from fry('fishing') was None but the value "fishin'" was expected.

How can we identify a word that ends with "ing." There is actually a str.endswith() function:

```
>>> 'fishing'.endswith('ing')
True
```

A regular expression to find "ing" at the end of a string would use \$ (pronounced "dollar") to *anchor* an expression:

```
>>> re.search('ing$', 'fishing')
<re.Match object; span=(4, 7), match='ing'>
```



We can use a string slice to get all the characters up to the last at index -1 and then append an apostrophe. Add this to your fry() function and see how many tests you pass:

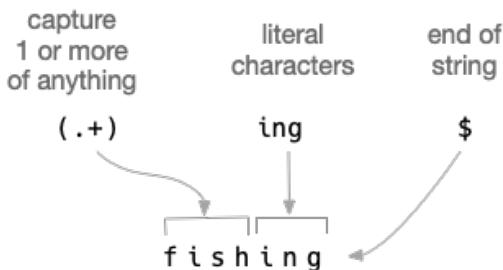
```
if word.endswith('ing'):
    return word[:-1] + ""
```

f i s h i n g

`word[:-1] + ""` → fishin'

Or you could use a groups with a regex to capture the first part of the word:

```
>>> match = re.search('(.+)ing$', 'fishing')
>>> match.group(1) + "in"
"fishin"
```



You should be able to get results like this:

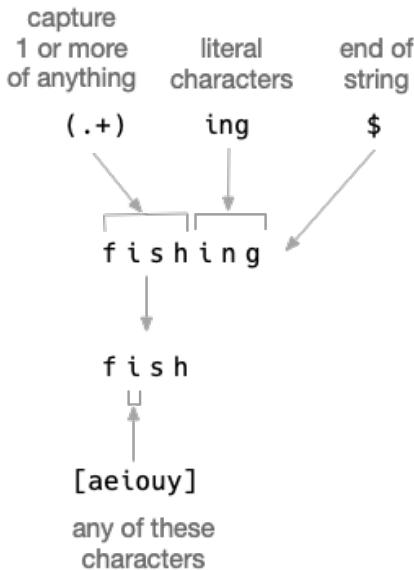
```
===== FAILURES =====
test_fry
=====
def test_fry():
    assert fry('you') == "y'all"
    assert fry('You') == "Y'all"
    assert fry('fishing') == "fishin'"
    assert fry('Aching') == "Achin'"
>     assert fry('swing') == "swing" ①
E     assert "swin'" == 'swing'
E         - swin' ③
E             ^
E         + swing
E             ^
friar.py:59: AssertionError
===== 1 failed in 0.10 seconds =====
```

- ① This test failed.
- ② The result of `fry('swing')` was "swin'" but should have been "swing."
- ③ Sometimes the test results will be able to highlight the exact point of failure. Here you are being shown that there is a ' where there should be a g.

We need a way to identify words that have two syllables. I mentioned before that we'll use a heuristic that looks for a vowel '[aeiouy]' in the part of the word *before* the "ing" ending. Another regex could do the trick:

```
>>> match = re.search('(.+)ing$', 'fishing') ①
>>> first = match.group(1) ②
>>> re.search('[aeiouy]', first) ③
<re.Match object; span=(1, 2), match='i'> ④
```

- ① The `(.+)` will match and capture one or more of anything followed by the characters `ing`. The return from `re.search()` will either be an `re.Match` object if the pattern was found or `None` to indicate it was not.
- ② Here we know there will be a `match` value, so we can use `match.group(1)` to get the first capture group which will be anything immediately before `ing`. In actual code, we should check that `match` is not `None` or we'd trigger an exception by trying to execute the `group` method on a `None`.
- ③ We can use `re.search()` on the `first` part of the string to look for a vowel.
- ④ As the return from `re.search()` is an `re.Match` object, we know there is a vowel in the first part, so the word looks to have two syllables.



If the word matches this test, then return the word with the final "g" replaced with an apostrophe; otherwise, return the word unchanged. I would suggest you not proceed until you are passing all of `test_fry()`.

15.1.6 Using the `fry()` function

Now your program should be able to:

1. Read input from the command line or a file
2. Read the input line-by-line
3. Split each line into words and non-words
4. `fry()` any individual word

The next step is to apply the `fry()` function to all the word-like units. I hope you can

see a familiar pattern emerging — applying a function to all elements of a list! You can use a `for` loop:

```
for line in args.text.splitlines():
    words = []
    for word in re.split(r'(\W+)', line.rstrip()):
        words.append(fry(word))
    print('.join(words))
```

- ① Preserve the structure of the newlines in `args.text` by using `str.splitlines()`.
- ② Create a `words` variable to hold the transformed words.
- ③ Split each line into words and non-words.
- ④ Add the "fried" word to the `words` list.
- ⑤ Print a new string of the joined words.

That (or something like it) should work well enough to pass the tests. Once you have a version that works, see if you can rewrite the `for` loop as a list comprehension and a `map()`.

Alrighty! Time to bear down and write this.

15.2 Solution

```
#!/usr/bin/env python3
"""Kentucky Friar"""

import argparse
import os
import re

# -----
def get_args():
    """get command-line arguments"""
    parser = argparse.ArgumentParser(
        description='Southern fry text',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('text', metavar='text', help='Input text or file')

    args = parser.parse_args()

    if os.path.isfile(args.text): ①
        args.text = open(args.text).read()

    return args

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

    args = get_args() ②

    for line in args.text.splitlines(): ③
        print('.join(map(fry, re.split(r'(\W+)', line.rstrip())))) ④
```

```

# -----
def fry(word): ⑤
    """Drop the `g` from `-ing` words, change `you` to `y'all`"""

    ing_word = re.search('(.+)ing$', word) ⑥
    you = re.match('([Yy])ou$', word) ⑦

    if ing_word: ⑧
        prefix = ing_word.group(1) ⑨
        if re.search('[aeiouy]', prefix, re.IGNORECASE): ⑩
            return prefix + "in"
        elif you: ⑪
            return you.group(1) + "all" ⑫
    return word ⑬

# -----
def test_fry(): ⑮
    """Test fry"""

    assert fry('you') == "y'all"
    assert fry('You') == "Y'all"
    assert fry('fishing') == "fishin"
    assert fry('Aching') == "Achin"
    assert fry('swing') == "swing"

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

```

- ① If the argument is a file, replace the `text` value with the contents from the file.
- ② Get the command-line arguments. The `text` value will either be the command-line text or the contents of a file by this point.
- ③ Use the `str.splitlines()` method to preserve the line breaks in the input text.
- ④ `map()` the pieces of text split by the regular expression through the `fry()` function which will return the words modified as needed. Use the `str.join` method to turn that resulting `list` back into a `str` to print.
- ⑤ Define a `fry()` function that will handle one word.
- ⑥ Search for "ing" anchored to the end of word. Use a capture to remember the part of the string *before* the "ing".
- ⑦ Search for "you" or "You" starting from the beginning of word. Capture the [yY] alternation in a group.
- ⑧ If the search for "ing" returned a match...
- ⑨ Get the prefix (the bit before the "ing") which is group number 1.
- ⑩ Perform a case-insensitive search for a vowel (plus "y") in the prefix. If nothing is found, then `None` will be returned which evaluates to `False` in this boolean context. If a match is returned, then the not-`None` value will evaluate to `True`.
- ⑪ Append "in" to the prefix and return it to the caller.
- ⑫ If the match for "you" succeeded...
- ⑬ Then return the captured first character plus "all."
- ⑭ Otherwise, return the word unaltered.

⑯ The tests for `fry()`.

15.3 Discussion

15.3.1 Breaking text into lines

In several previous exercises, I used a technique of reading an input file into the `args.text` value. If the input is coming from a file, then there will be newlines separating each line of text. I had suggested a `for` loop to handle each line of input text returned by `str.splitlines()` to preserve the newlines in the output. I suggested you also start with a second `for` loop to handle each word-like unit returned by the `re.split()`:

```
for line in args.text.splitlines():
    words = []
    for word in re.split('(\W+)', line.rstrip()):
        words.append(fry(word))
    print(''.join(words))
```

That's a total of 5 LOC that could be written in 2 if we replace the second `for` with a list comprehension:

```
for line in args.text.splitlines():
    print(''.join([fry(w) for w in re.split(r'(\W+)', line.rstrip())]))
```

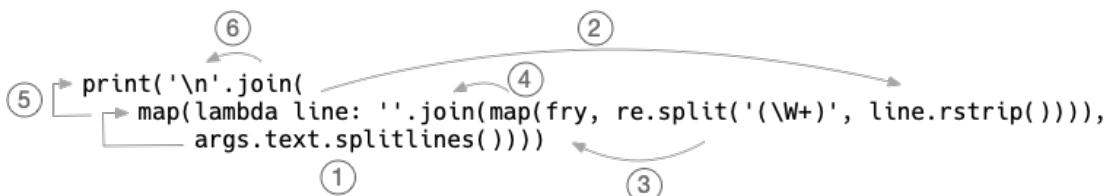
Or slightly shorter using a `map()`:

```
for line in args.text.splitlines():
    print(''.join(map(fry, re.split(r'(\W+)', line.rstrip()))))
```

Or, if you're just a little bit weird, you can replace both `for` loops with `map()`:

```
print('\n'.join(
    map(lambda line: ''.join(map(fry, re.split(r'(\W+)', line.rstrip()))),
        args.text.splitlines())))
```

Here's annotation to help you follow that:



1. `args.text` is split on newlines, each of which is passed to as `line` to the `lambda`.
2. The `line` is stripped of whitespace on the right and fed as the argument to `re.split()`.
3. Each word-like unit from `re.split()` is passed by `map()` through the `fry()` function.
4. The new list returned by `map()` is joined on the empty string to create a new `str`.
5. The `map()` returns a list of new `str` values created from the processed lines of

text.

6. These new strings are joined back together on newlines to reconstitute the original line endings.

I find that last one a little hard to write and even harder to read, so I'd tend to use the version before it. Remember that code is *read* far more often than it is *written*. The last version is probably too clever by half (but it can be fun to challenge yourself by writing weird code)

15.3.2 Writing the `fry()` function manually

You were not required, of course, to write a `fry()` function. However you wrote your solution, I hope you wrote tests for it. Here is a version that is fairly close to some of the suggestions I made in the introduction. This version uses no regular expressions:

```
def fry(word):
    """Drop the `g` from `-ing` words, change `you` to `y'all`"""

    if word.lower() == 'you':          ①
        return word[0] + "all"         ②

    if word.endswith('ing'):           ③
        if any(map(lambda c: c.lower() in 'aeiouy', word[:-3])): ④
            return word[:-1] + "'" ⑤
        else:
            return word             ⑥

    return word                      ⑦
```

- ① Force the word to lowercase and see if it matches "you."
- ② If so, return the first character (to preserve the case) plus "all".
- ③ If the word ends with "ing" ...
- ④ Check if it's True that any of the characters in our vowel set are in the word up to the "ing" suffix.
- ⑤ If so, return the word up to the last index plus the apostrophe.
- ⑥ Else return the word unchanged.
- ⑦ If the word is neither an "ing" or "you" word, then return it unchanged.

Let's take a moment to appreciate the `any` function as it's one of my favorites. Here I'm using a `map()` to check if each of the vowels exists in the portion of the word before the "ing" ending:

```
>>> word = "cooking"
>>> list(map(lambda c: (c, c.lower() in 'aeiouy'), word[:-3]))
[('c', False), ('o', True), ('o', True), ('k', False)]
```

The first character of "cooking" is "c," and it does not appear in the string of vowels. The next two characters ("o") do appear in the vowels, but "k" does not.

Let's reduce this to just the `True/False` values:

```
>>> list(map(lambda c: c.lower() in 'aeiouy', word[:-3]))
[False, True, True, False]
```

And now we can use `any` to tell us if *any* of the values are `True`:

```
>>> any([False, True, True, False])
True
```

It's exactly the same as joining the values with `or`:

```
>>> False or True or True or False
True
```

The `all()` function returns `True` only if *all* the values are true:

```
>>> all([False, True, True, False])
False
```

Which is the same as joining those values on `and`:

```
>>> False and True and True and False
False
```

If it's `True` that one of the vowels appears in the first part of the word, we have determined this is (probably) a two-syllable word and can return the word with the final "g" replaced with an apostrophe. Otherwise, we return the unaltered word:

```
if any(map(lambda c: c.lower() in 'aeiou', word[:-3])):
    return word[:-1] + "'"
else:
    return word
```

This approach works fine, but it's all quite manual as we have to write quite a bit of code to find our patterns.

15.3.3 Writing the `fry()` function with regular expressions

Let's revisit the version of the `fry()` function that uses regular expressions:

```
def fry(word):
    """Drop the `g` from `-ing` words, change `you` to `y'all`"""

    ing_word = re.search('(.+)ing$', word) ①
    you = re.match('([Yy])ou$', word) ②

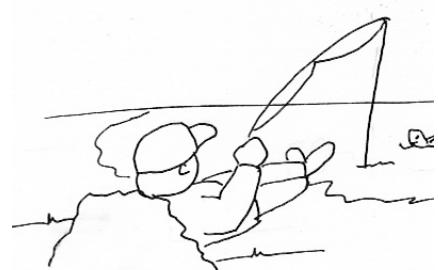
    if ing_word: ③
        prefix = ing_word.group(1) ④
        if re.search('[aeiou]', prefix, re.IGNORECASE): ⑤
            return prefix + "in'" ⑥
        elif you: ⑦
            return you.group(1) + "'all" ⑧

    return word ⑨
```

- ① The pattern '`(.+)``ing$`' matches one or more of anything followed by "ing." The \$ anchors the pattern to the end of the string, so this is looking for a string that ends in "ing," but the string cannot just be "ing" as it has to have at least one of something before it. The parentheses capture the part before the "ing."

- ② The `re.match()` starts matching at the beginning of the given word and is looking for either an upper- or lowercase "y" followed by "ou" and then the end of the string (\$).
- ③ If `ing_word` is `None`, that means it failed to match. If it is not `None` (so it is "truthy"), that means it is an `re.Match` object we can use.
- ④ The prefix is the bit before the "ing" that we wrapped in parentheses. Because it is the first set of parens, we can fetch it with `ing_word.group(1)`.
- ⑤ Use `re.search()` to look anywhere in the prefix for any of the vowels (plus "y") in a case-insensitive fashion. Remember that `re.match()` would look at the beginning of word, which is not what we want.
- ⑥ Return the prefix plus the string "in" so as to drop the final "g."
- ⑦ If the `re.match()` for the "you" pattern fails, then you will be `None`. If it is not `None`, then it matched and you is an `re.Match` object.
- ⑧ We used parens to capture the first character so as to maintain the case. That is, if the word was "You," then we want to return "Y'all." Here we return that first group plus the string "all."
- ⑨ If the word matched neither a two-syllable "ing" patter or the word "you," then return the word unchanged.

I've been using regexes for maybe 20 years, so this version seems much simpler to me than the manual version. You may feel differently. If you are completely new to regexes, trust me that they are so very worth the effort to learn. I absolutely would not be able to do much of my work without them!



15.4 Review

- Regular expressions can be used to find patterns in text. The patterns can be quite complicated, like a grouping of non-word characters in between groupings of word characters.
- The `re` module has the functions `re.match()` to find pattern at the beginning of some text, `re.search()` to find a pattern anywhere inside some text, and `re.split()` to break text on a pattern.
- If you use capturing parens on the pattern for `re.split()`, then the captured split pattern will be included in the returned values. This allows you to reconstruct the original string with the separators.

15.5 Going Further

- You could also replace "your" with "y'all's." For instance, "Where are your britches?" could become "Where are y'all's britches?"
- Change "getting ready" or "preparing" to "fixin'," as in "I'm getting ready to eat" to "I'm fixin' to eat." Also change the string "think" to "reckon," as in "I think this

is funny" to "I reckon this is funny." You should also change "thinking" to "reckoning," which then should become "reckonin'." That means you either need to make two passes for the changes or find both "think" and "thinking" in the one pass.

- Make a version of the program for another regional dialect. I lived in Boston for a while and really enjoyed saying "wicked" all the time instead of "very" as in "IT'S WICKED COLD OUT!"

16

The Scrambler: Randomly reordering the middles of words

Yuor brian is an azinamg cmiobiaontn of hdarware and sftraowe. You're rdineag tihs rghit now eevn thgouh the wrdos are a mses, but yuor biran can mkae snese of it bceause the frsit and lsat ltrtees of ecah wrod hvae saeytd the smae. Yuor biran de'onst atlauclcy raed ecah lteetr of ecah wrod but rades wlohe wdors. The scamrbeld wrdos difteienly solw you dwon, but y'roue not rlleay eevn tyinrg to ulsrmbance the lrttees, are you? It jsut hnaepps!



In this exercise, we will write a program called `scrambler.py` that will scramble each word of the text given as an argument. The scrambling should only work on words with 4 characters or more and should only scramble the letters in the middle of the word, leaving the first and last characters unchanged. The program should take a `-s` or `--seed` option (an int with default None) to pass to `random.seed()`.

It should handle text on the command line:

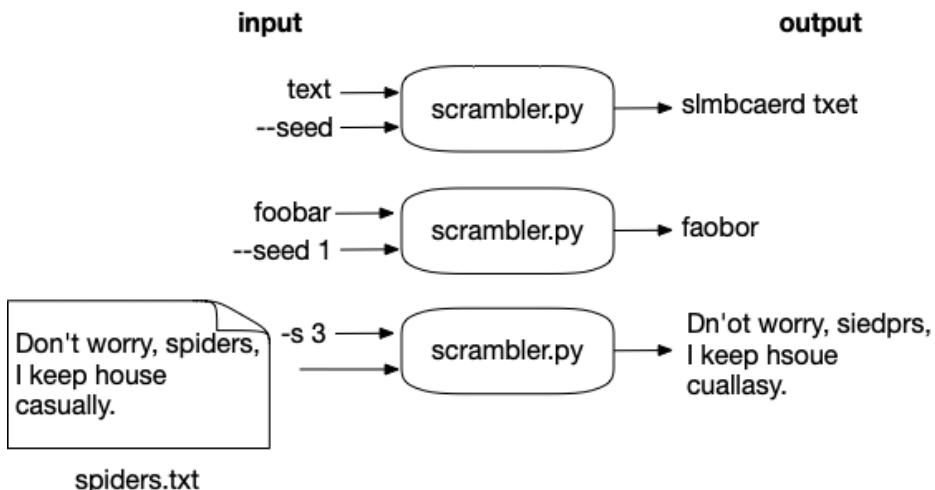
```
$ ./scrambler.py --seed 1 "foobar bazquux"
faobor buuzaqx
```

Or from a file:

```
$ cat ../inputs/spiders.txt
Don't worry, spiders,
I keep house
casually.
```

```
$ ./scrambler.py ../inputs/spiders.txt
D'not wrory, sdireps,
I keep hsoue
csalluay.
```

Here's a string diagram to help you think about it:



In this exercise, you will:

- Use a regular expression to split text into words.
- Use the `random.shuffle()` function to shuffle a list.
- Create scrambled versions of words by shuffling the middle letters while leaving the first and last unchanged.

16.1 Writing `scrambler.py`

I recommend you start by using `new.py scrambler.py` to create the program in the `scrambler` directory. You can also copy `template/template.py` to `16_scrambler/scrambler.py`. You can refer to previous exercises like "Howler" to remember how to handle our positional argument that might be text or might a file of text to read. When run with no arguments or the flags `-h` or `--help`, it should present usage:

```
$ ./scrambler.py -h
usage: scrambler.py [-h] [-s seed] text

Scramble the letters of words

positional arguments:
  text                  Input text or file

optional arguments:
  -h, --help            show this help message and exit
  -s seed, --seed seed  Random seed (default: None)
```

Once your program's usage statement matches this, then change your `main()` definition to this:

```
def main():
    args = get_args()
    print(args.text)
```

And verify that your program can echo text from the command line:

```
$ ./scrambler.py hello
hello
```

Or from an input file:

```
$ ./scrambler.py ../inputs/spiders.txt
Don't worry, spiders,
I keep house
casually.
```

16.1.1 *Breaking the text into lines and words*

As in the "Kentucky Friar," we want to preserve the line breaks of the input text by using `str.splitlines()`:

```
for line in args.text.splitlines():
    print(line)
```

If we are reading the `spiders.txt` haiku, the first line is:

```
>>> line = "Don't worry, spiders,"
```

We need to break the line into words. In the "Words Count" exercise, it was enough to use `str.split()`. This leaves punctuation stuck to our words — both `worry`, and `spiders`, have commas:

```
>>> line.split()
['Don''t', 'worry,', 'spiders,']
```

In the "Friar," we used the `re.split()` function with the regular expression `(\W+)` to split text on one or more non-word characters. Let's try that:

```
>>> re.split('(\W+)', line)
['Don', "'", 't', ' ', 'worry', ',', ' ', 'spiders', ',', '']
```

That won't work because it splits `Don't` into three parts, `Don`, `'`, and `t`. When I was working through this, my next idea was to use `\b` to break on *word boundaries*. Note I have to put an `r''` in front of the first quote `r'\b'` to denote that it is a "raw" string.

Raw strings

The `r''` before the opening quote in the expression `r'\b'` creates a "raw" string which prevents Python from trying to interpret escaped sequences in the string. For instance, we usually type `\n` to represent a newline and `\t` for a tab character, but `\b` isn't supposed to be turned in any other character. We want Python to pass the literal string `\b` (that is, a backslash and then the character b) to the regex engine where it can be interpreted as "word boundary." It's generally safer to create regexes as raw strings

because of the number of backslash-escaped sequences that are used like \w for "word character" or \s for "whitespace."

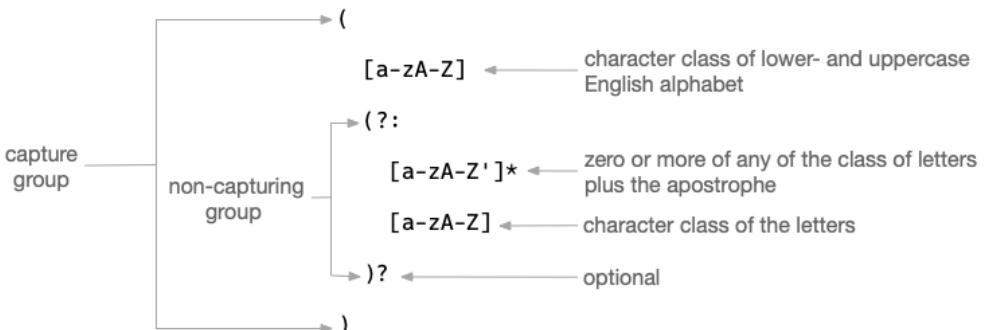
This still won't work because \b thinks the apostrophe is a word boundary and so splits the contracted word:

```
>>> re.split(r'\b', "Don't worry, spiders,")
['', 'Don', "'", 't', ' ', 'worry', ' ', 'spiders', '']
```

While searching the Internet for a regex to split the text properly, I found this pattern on a Java discussion board which works just perfectly to separate *words* from *non-words*¹⁶:

```
>>> re.split("(?<=[a-zA-Z](?:[a-zA-Z']*[a-zA-Z])?)", "Don't worry, spiders,")
['', "Don't", ' ', 'worry', ' ', 'spiders', '']
```

The beautiful thing about regular expressions is that they are their own language, one that is used inside many other languages from Perl to Haskell. Let's dig into this pattern:



16.1.2 Capturing, non-capturing, and optional groups

Above you see that groups can contain other groups. For instance, I can capture the entire string "foobarbaz" as well as the contained string "bar" like so:

```
>>> match = re.match('(foo(bar)baz)', 'foobarbaz')
```

Capture groups are numbered by the position of their left parenthesis. Since the first left paren starts the capture starting at "f" and going to "z", that is group 1:

```
>>> match.group(1)
'foobarbaz'
```

The second left paren starts just before the "b" and goes to the "r":

```
>>> match.group(2)
```

¹⁶ I would like to stress that a significant part of my job is spent looking for answers both in the books I own but also on the Internet!

```
'bar'
```

I can also make a group *non-capturing* by using the starting sequence `(?:`. If I use this on the second group, then I no longer capture the substring "bar":

```
>>> match = re.match('(foo(?:bar)baz)', 'foobarbaz')
>>> match.groups()
('foobarbaz',)
```

This is often used when you are using a grouping primarily for the purpose of making it optional by placing a ? after the closing paren. For instance, I can make the "bar" optional and then match both "foobarbaz":

```
>>> re.match('(foo(?:bar)?baz)', 'foobarbaz')
<re.Match object; span=(0, 9), match='foobarbaz'>
```

As well as "foobaz":

```
>>> re.match('(foo(?:bar)?baz)', 'foobaz')
<re.Match object; span=(0, 6), match='foobaz'>
```

16.1.3 Compiling a regex

You will often see the `re.compile()` is used to store a regular expression into a variable that has some meaningful name. I often do this, especially if I use the same regex at different points in my code:

```
>>> splitter = re.compile("([a-zA-Z](?:[a-zA-Z]*[a-zA-Z]))")
>>> splitter.split("Don't worry, spiders,")
['', "Don't", ' ', 'worry', ' ', ' ', 'spiders', ',']
```

Whenever you use something like `re.search()` or `re.split()`, the regex engine must parse the str value you provide for the regex into something it understands and can use. This parsing step must happen *each time* you call the function, so we can *compile* the regex just once and reuse it over and over. When you compile the regex and assign it to a variable, the parsing step is done before you call the function which improves performance.

16.1.4 Scrambling a word

Now that we have a way to process the *lines* and then *words* of our text, let's think about how we'll scramble our words by starting with just *one word*. You and I will need to use the same algorithm for scrambling the words, so here are the rules:

- If the word is 3 characters or shorter, return the word unchanged.
- Use a string slice to copy the characters not including the first and last.
- Use the `random.shuffle()` method to mix up



the letters in the middle.

- Return the new "word" by combining the first, middle, and last parts.

I recommend you create a function called `scramble()` that will do all this and a test for it. Feel free to add this to your program:

```
def scramble(word):          ①
    """Scramble a word"""
    pass

def test_scramble():
    """Test scramble"""
    state = random.getstate() ②
    random.seed(1)            ③
    assert scramble("a") == "a" ④
    assert scramble("ab") == "ab"
    assert scramble("abc") == "abc"
    assert scramble("abcd") == "acbd" ⑤
    assert scramble("abcde") == "acbde" ⑥
    assert scramble("abcdef") == "aecbdf"
    assert scramble("abcde'f") == "abcd'ef" ⑦
    random.setstate(state)
```

- ① The `pass` is a no-op (no operation), so this function literally does nothing. This is just a placeholder so that we can write a test and verify that the function fails.
- ② The change we'll make by setting the `random.seed()` in the next line will be a global change. We'll want to restore the state after the testing, so here we use `random.getstate()` to get the current state of the `random` module.
- ③ Set the `random.seed()` to a known value for testing.
- ④ Words with 3 characters or fewer should be returned unchanged.
- ⑤ This word looks unchanged, but that's just because with the seed of 1 the shuffling didn't end up changing the middle characters.
- ⑥ Now it's more evident that the word is being scrambled.
- ⑦ Restore the state to the previous value.

Inside your `scramble()` function, you will have a word like "worry." We can use list slices to extract part of a string. Since Python starts numbering at 0, we use 1 to indicate the *second* character:

```
>>> word = 'worry'
>>> word[1]
'o'
```

The last index of any string is -1:

```
>>> word[-1]
'y'
```

If we want a slice, we use the `list[start:stop]` syntax. Since the `stop` position is not included, we can get the `middle` like so:

```
>>> middle = word[1:-1]
>>> middle
'orr'
```

We can import random to get access to the random.shuffle() function. Like the list.sort() and list.reverse() methods, the argument will be shuffled **in-place**, and the function will return None. That is, you might be tempted to write code like this:

```
>>> import random
>>> x = [1, 2, 3]
>>> shuffled = random.shuffle(x)
```

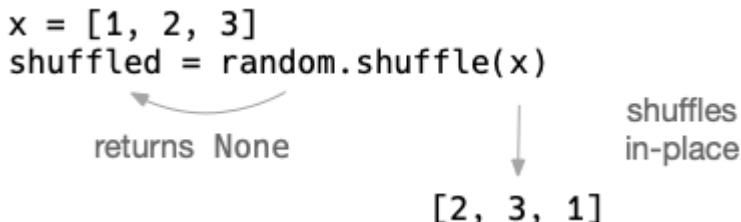
What is the value of shuffled? Is it something like [3, 1, 2] or is it None?

```
>>> type(shuffled)
<class 'NoneType'>
```

The shuffled values now holds None while the x list has been shuffled *in-place*:

```
>>> x
[2, 3, 1]
```

The return from random.shuffle() was None, so shuffled was assigned None:



If you've been following along, it turns out that we cannot shuffle the `middle` like this:

```
>>> random.shuffle(middle)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/kyclark/anaconda3/lib/python3.7/random.py", line 278, in shuffle
    x[i], x[j] = x[j], x[i]
TypeError: 'str' object does not support item assignment
```

The `middle` variable is a str:

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

The `random.shuffle()` function is trying to directly modify a str value in-place, but str values in Python are *immutable*. One workaround is to make `middle` into a new list of the characters from `word`:

```
>>> middle = list(word[1:-1])
>>> middle
['o', 'r', 'r']
```

And that is something we can shuffle:

```
>>> random.shuffle(middle)
>>> middle
['r', 'o', 'r']
```

Then it's a matter of creating a new string with the original first letter, the shuffled middle, and the last letter. I'll leave that for you to work out.

Use `pytest scrambler.py` to have `pytest` execute the `test_scramble()` function to see if it works correctly. Run this command *after every change to your program*. Ensure that your program always compiles and runs properly. Only make *one change at a time*, then save your program and run the tests!

16.1.5 Scrambling all the words

As in several previous exercises, we're now down to applying the `scramble()` function to all the words. Can you see a familiar pattern?

```
splitter = re.compile("[a-zA-Z](?:[a-zA-Z]*[a-zA-Z])?")
for line in args.text.splitlines():
    for word in splitter.split(line):
        # what goes here?
```

We've talked about how to apply a function to each element in a sequence. You might try a `for` loop, a list comprehension, or maybe a `map()`. Think about how you can split the text into words and feed them to the `scramble()` function, then join them back together to reconstruct the text.

Note that this approach will pass both the words and the non-words (the bits in between each word) to the `scramble()` function. You don't want to modify the non-words, so you'll need a way to check that the argument looks like a word. Maybe a regular expression?

That should be enough to go on. Write your solution and use the included tests to check your program.

16.2 Solution

```
#!/usr/bin/env python3
"""Scramble the letters of words"""

import argparse
import os
import re
import random

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

    parser = argparse.ArgumentParser(
        description='Scramble the letters of words',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('text', metavar='text', help='Input text or file') ①
```

```

parser.add_argument('-s',          ②
                    '--seed',
                    help='Random seed',
                    metavar='seed',
                    type=int,
                    default=None)

args = parser.parse_args()        ③

if os.path.isfile(args.text):    ④
    args.text = open(args.text).read().rstrip()

return args                      ⑤

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

    args = get_args()            ⑥
    random.seed(args.seed)       ⑦
    splitter = re.compile("([a-zA-Z](?:[a-zA-Z]*[a-zA-Z]))") ⑧

    for line in args.text.splitlines(): ⑨
        print('.join(map(scramble, splitter.split(line))))') ⑩

# -----
def scramble(word):              ⑪
    """For words over 3 characters, shuffle the letters in the middle"""

    if len(word) > 3 and re.match(r'\w+', word): ⑫
        middle = list(word[1:-1])                ⑬
        random.shuffle(middle)                   ⑭
        word = word[0] + ''.join(middle) + word[-1] ⑮

    return word                         ⑯

# -----
def test_scramble(): ⑰
    """Test scramble"""

    random.seed(1)
    assert scramble("a") == "a"
    assert scramble("ab") == "ab"
    assert scramble("abc") == "abc"
    assert scramble("abcd") == "acbd"
    assert scramble("abcde") == "acbde"
    assert scramble("abcdef") == "aecbdf"
    assert scramble("abcde'f") == "abcd'ef"
    random.seed(None)

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

```

① The text argument may be plain text on the command line or the name of a file to read.

- ② The seed option is an int that defaults to None.
- ③ Get the args so we can check the text value.
- ④ If args.text names an existing file, replace the value of args.text with the result of opening and reading the file's contents.
- ⑤ Return the arguments to the caller.
- ⑥ Get the command-line arguments.
- ⑦ Use the args.seed to set the random.seed() value. If args.seed is the default None, then this is the same as not setting the seed.
- ⑧ Save the compiled regex into a variable.
- ⑨ Use str.splitlines() to preserve the line breaks in args.text.
- ⑩ Use the splitter to break the line into a new a list that map() will feed into the scramble() function. Join the resulting list on the empty string to create a new str to print.
- ⑪ Define a function to scramble() a single word.
- ⑫ Only scramble words with 4 or more characters if they contain word characters.
- ⑬ Copy the second to second-to-last characters of the word into a new list called middle.
- ⑭ Shuffle the middle.
- ⑮ Set the word equal to the first character plus the middle plus the last character.
- ⑯ Return the word which may have been altered if it met the criteria.
- ⑰ The test for the scramble() function.

16.3 Discussion

There is nothing new in the get_args(), so I trust you understand that code. Refer to "Howler" if you want to revisit how to handle the args.text coming from the command line or from a file.

16.3.1 Processing the text

As mentioned in the introduction, I often will assign a *compiled* regex to a variable. Here I did it with the splitter:

```
splitter = re.compile("[a-zA-Z](?:[a-zA-Z']*[a-zA-Z])?")
```

The other reason I like to use re.compile() is because I feel it can make my code more readable. Without this, I would have to write:

```
for line in args.text.splitlines():
    print(''.join(map(scramble, re.split("[a-zA-Z](?:[a-zA-Z']*[a-zA-Z])?", line))))
```

I find this version much easier to read:

```
splitter = re.compile("[a-zA-Z](?:[a-zA-Z']*[a-zA-Z])?")
for line in args.text.splitlines():
    print(''.join(map(scramble, splitter.split(line))))
```

Maybe you still find that code somewhat confusing. Let's diagram the flow of the data:

Figure 16.1. A visualization of how the data moves through the map() function.

```

print('.join(map(scramble, splitter.split("Don't worry, spiders,")))) ①
    ↓
print('.join(map(scramble, ['', "Don't", ' ', 'worry', ' ', 'spiders', ','])) ②
    ↓
    ↙
print('.join([scramble('),
            scramble("Don't"),
            scramble(' '),
            scramble('worry'),
            scramble(','),
            scramble('spiders'),
            scramble(',')]))
    ↓
    ↙
print('.join([', "Dn'ot", ' ', 'wrroy', ' ', 'seirpds', ','])) ④
    ↓
    ↙
print("D'ont wrroy, srpdeis,") ⑤

```

1. First Python will split the string "Don't worry, spiders,".
2. The split creates a new list words and non-words fed to map as the inputs to the `scramble()` function.
3. The `map()` essentially creates a list in which the `scramble()` function is called for each element.
4. The result of `map()` is a new list with the results of each application of the `scramble()` function. This list is the argument to the `str.join()` function.
5. The result of the join is a new string which is the argument to `print()`.

A longer way to write this with a `for` loop might look like this:

```

for line in args.text.splitlines():
    words = []
    for word in splitter.split(line):
        words.append(scramble(word))
    print('.join(words)) ⑤

```

- ① Use `str.splitlines()` to preserve the original line breaks.
- ② For each line of input, create a `words` variable to hold the scrambled words.
- ③ Use the `splitter` to split the line.
- ④ Add the result of `scramble(word)` to the `words` list.
- ⑤ Join the words on the empty string and pass to `print()`.

Because the goal is to create a new list, this is better written as a list comprehension:

```

for line in args.text.splitlines():
    words = [scramble(word) for word in splitter.split(line)]

```

```
print(''.join(words))
```

Or you could go quite the opposite direction and replace all `for` loops with `map()`:

```
print('\n'.join(
    map(lambda line: ''.join(map(scramble, splitter.split(line))),
        args.text.splitlines())))
```

The last one reminds me of a programmer I used to work who would jokingly say "If it was hard to write, it should be hard to read!" It becomes somewhat clearer if you rearrange the code. Note that `pylint` will complain about assigning a `lambda`, but I really don't agree with that criticism:

```
scrambler = lambda line: ''.join(map(scramble, splitter.split(line)))
print('\n'.join(map(scrambler, args.text.splitlines())))
```

Writing code that is correct, tested, and understandable is as much an art as it is a craft. Choose the version that you (and your teammates!) believe is the most readable.

16.3.2 Scrambling a word

Let's take a closer look at my `scramble()` function because I wrote it in a way that would make it easy to incorporate into a `map()`:

```
def scramble(word):
    """For words over 3 characters, shuffle the letters in the middle"""
    if len(word) > 3 and re.match(r'\w+', word): ①
        middle = list(word[1:-1]) ②
        random.shuffle(middle) ③
        word = word[0] + ''.join(middle) + word[-1] ④

    return word ⑤
```

- ① Check if the given word is one we ought to scramble. First, it must be longer than 3 characters. Second, it must contain one or more word characters because the function will be passed both "word" and "non-word" strings. If either check returns `False`, we will return the word unchanged. The `r'\w+'` is used to create a "raw" string. Note that regex works fine with or without being a raw string, but `pylint` complains about an "invalid escape character" unless it is a raw string.
- ② Copy the middle of the word to a new `list` called `middle`.
- ③ Shuffle the `middle` *in-place*. Remember that this function returns `None`.
- ④ Reconstruct the word by joining together the first character, the shuffled `middle`, and the last character.
- ⑤ Return the word which may or may not have been shuffled.

One key mistake you could make would be writing this code:

```
middle = random.shuffle(middle)
```

Honestly, I would prefer that `random.shuffle()` accept a `list` (or `str`) and return a new, shuffled `list` (or `str`) rather than mutating the given argument, but c'est la vie. The function was written to mutate the data, and that's that.

16.4 Review

- The regex we used to split the text into words was quite complex but also gave us exactly what we needed. Writing the program without this piece would have made it significantly more difficult. Regexes, while complex and deep, are wildly powerful black magic that can make your programs incredibly flexible and useful.
- The `random.shuffle()` function accepts a list which is mutated in-place.
- List comprehensions and `map()` can often lead to more compact code, but going too far can reduce readability. Choose wisely.



16.5 Going Further

- Write a version where the `scramble()` function sorts the middle letters into alphabetical order rather than shuffling them.
- Write a version that reverses each word rather than scrambles.
- Write a program to *unscramble* the text. For this, you'd probably need to have a dictionary of English words — most Unix systems have a file at `/usr/share/dict/words` for this. You would have to split the scrambled text into words/non-words, then compare each "word" to the words in your dictionary. I would recommend you start by comparing the words as anagrams and then using the first and last letters to positively identify the unscrambled word.

17

Mad Libs: Using regular expressions

When I was a wee lad, we used to play at Mad Libs for hours and hours. This was before computers, mind you, before televisions or radio or even paper! No, scratch that, we had paper. Anyway, point is we only had Mad Libs to play, and we loved it! And now you must play!

We'll write a program called `mad.py` that will read a file given as a positional argument and find all the placeholders noted in angle brackets like `<verb>` or `<adjective>`. For each placeholder, we'll prompt the user for the part of speech being requested like "Give me a verb" and "Give me an adjective." (Notice that you'll need to use the correct article just as in "Crow's Nest.") Each value from the user will then replace the placeholder in the text, so if the user says "drive" for "verb," then `<verb>` in the text will be replaced with `drive`. When all the placeholders have been replaced with inputs from the user, print out the new text.



There is a `17_mad_libs/inputs` directory with some sample files you can use, but I encourage you to create your own. For instance, here is a version of the "fox" text:

```
$ cat inputs/fox.txt
The quick <adjective> <noun> jumps <preposition> the lazy <noun>.
```

When the program is run with this file as the input, it will ask for each of the placeholders and then print the silliness:

```
$ ./mad.py inputs/fox.txt
Give me an adjective: surly
Give me a noun: car
Give me a preposition: under
Give me a noun: bicycle
The quick surly car jumps under the lazy bicycle.
```

By default, this is an interactive program that will use the `input()` prompt to ask the user for their answers, but, for testing purposes, you will have an option for `-i` or `--inputs` so the test suite can pass in all the answers and bypass the interactive `input()` calls:

```
$ ./mad.py inputs/fox.txt -i surly car under bicycle
The quick surly car jumps under the lazy bicycle.
```

In this exercise, you will:

- Print learn to use `sys.exit()` to halt your program and indicate an error status.
- Learn about greedy matching with regular expressions.
- Use `re.findall()` to find all matches for a regex.
- Use `re.sub()` to substitute found patterns with new text.
- Explore ways to write without using regular expressions.

17.1 Writing mad.py

To start off, use `new.py mad.py` to create the program or copy `template/template.py` to `17_mad_libs/mad.py`. You would do well to define the positional `file` argument as `type=argparse.FileType('r')`. The `-i` or `--inputs` option should use `nargs='*'` to define a list of zero or more `str` values.

First modify your `mad.py` until it produces the following usage when given no arguments or the `-h` or `--help` flag:

```
$ ./mad.py -h
usage: mad.py [-h] [-i [input [input ...]]] FILE

Mad Libs

positional arguments:
  FILE           Input file

optional arguments:
  -h, --help      show this help message and exit
  -i [input [input ...]], --inputs [input [input ...]]
                  Inputs (for testing) (default: None)
```

If the given `file` argument does not exist, the program should error out:

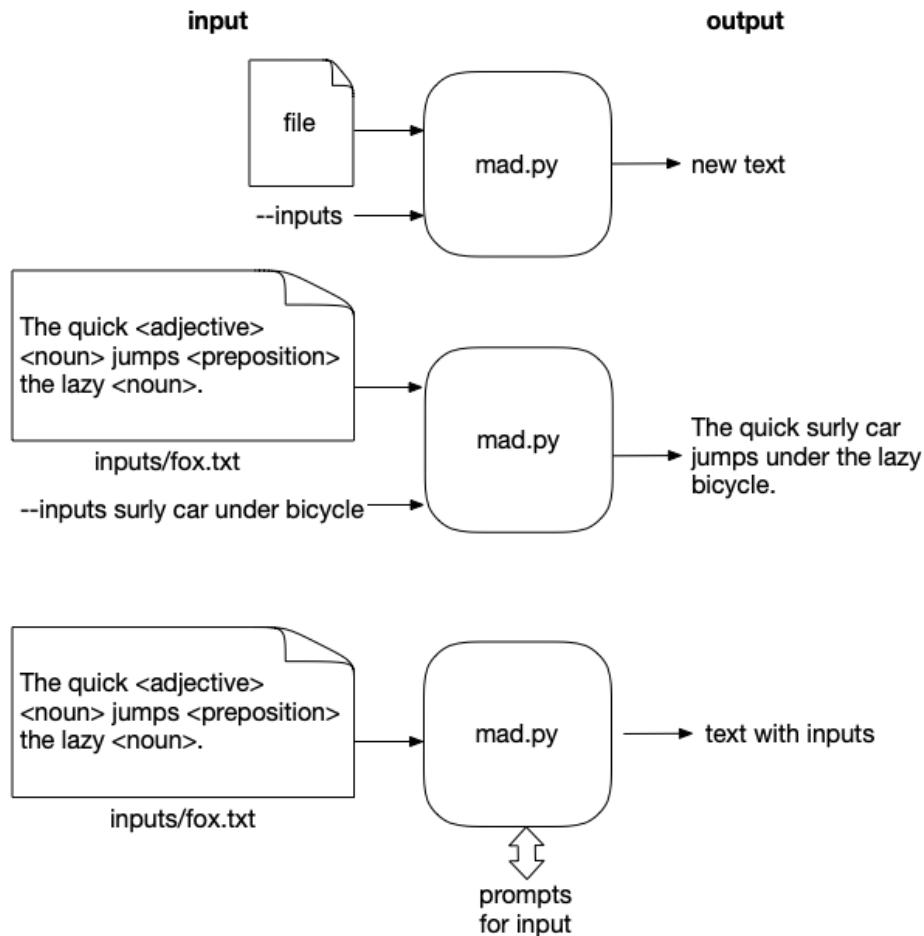
```
$ ./mad.py blarhg
usage: mad.py [-h] [-i [str [str ...]]] FILE
mad.py: error: argument FILE: can't open 'blarhg': \
```

```
[Errno 2] No such file or directory: 'blargh'
```

If the text of the file contains no <> placeholders, it should print a message and *exit with an error value* (something other than 0). Note this does not need to print a usage, so you don't have to use `parser.error()` as in previous exercises:

```
$ cat no_blanks.txt
This text has no placeholders.
$ ./mad.py no_blanks.txt
"no_blanks.txt" has no placeholders.
```

Here is a string diagram to help you visualize the program:



17.1.1 Using regular expressions to find the pointy bits

The first thing we need to do is `read()` the input file:

```
>>> text = open('inputs/fox.txt').read().rstrip()
>>> text
```

```
'The quick <adjective> <noun> jumps <preposition> the lazy <noun>.'
```

We need to find all the <...> bits, so let's use a regular expression. We can find a literal < character like so:

```
>>> import re
>>> re.search('<', text)
<re.Match object; span=(10, 11), match='<'>
```



The quick <adjective> <noun> jumps <preposition> the lazy <noun>.

Now let's find that bracket's mate. The . means "anything," and we can add a + after it to mean "one or more". I'll capture the match so it's easier to see:

```
>>> match = re.search('<.+>', text)
>>> match.group(1)
'<adjective> <noun> jumps <preposition> the lazy <noun>'
```



The quick <adjective> <noun> jumps <preposition> the lazy <noun>.

Hmm, that matched all the way to the end of the string instead of stopping at the first available >. It's common when we use * or + for zero/one or more that the regex engine is "greedy" on the *or more* part. The pattern matches beyond where we want them to, but they are technically matching exactly what we describe. Remember that . means *anything*, and a right angle bracket is anything. It matches as many characters as possible until it finds the last right angle to stop which is why this pattern is called "greedy."



We can make the regex "non-greedy" by changing + to +? so that it matches the shortest possible string:

```
>>> re.search('<.+?>', text)
<re.Match object; span=(10, 21), match='<adjective>'>
```



Rather than using `.` for "anything," it would be more accurate to say that we want to match one or more of anything *that is not either of the angle brackets*. The character class `[<>]` would match either bracket. We can negate (or complement) the class by putting a caret (^) as the first character so we have `[^<>]`. That will match anything that is not a left or right angle bracket:

```
>>> re.search('<[^<>]+>', text)
<re.Match object; span=(10, 21), match='<adjective>'>
```

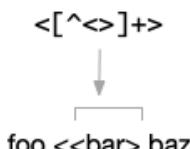


Why do we have both brackets inside the negated class? Wouldn't the right bracket be enough? Well, I'm guarding against *unbalanced* brackets. With only the right bracket, it would match this text:

```
>>> re.search('<[^>]+>', 'foo <<bar> baz')
<re.Match object; span=(4, 10), match='<<bar>'>
```



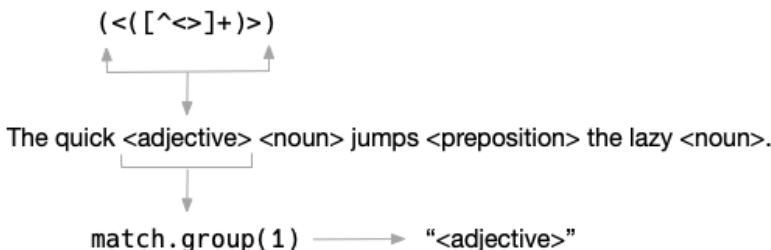
But with *both* brackets in the negated class, it finds the correct, balanced pair:



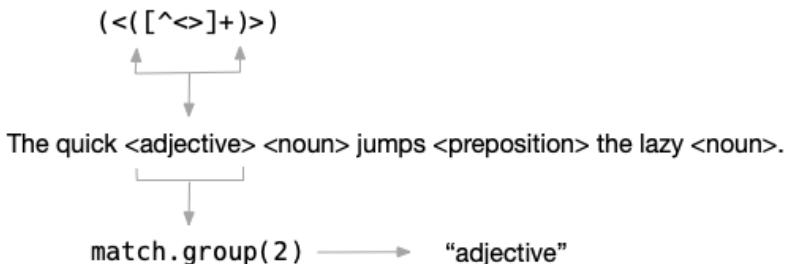
```
>>> re.search('<[^<>]+>', 'foo <<bar> baz')
<re.Match object; span=(5, 10), match='<bar>'>
```

We'll add two sets of parens (), one to capture the *entire* placeholder pattern:

```
>>> match = re.search('<(<[^<>]+>)>', text)
>>> match.groups()
('<adjective>', 'adjective')
```



And another for the string *inside* the <>:



There is a very handy function called `re.findall()` that will return all matching text groups as a list of tuple values:

```
>>> from pprint import pprint
>>> matches = re.findall('<([^\wedge\wedge]+)>', text)
>>> pprint(matches)
[('<adjective>', 'adjective'),
 ('<noun>', 'noun'),
 ('<preposition>', 'preposition'),
 ('<noun>', 'noun')]
```

Note that the capture groups are returned in the order of their opening parentheses, so the entire placeholder is the first member of each tuple and the contained text is the second. We can iterate over this list, *unpacking* each tuple into variables:



```
>>> for placeholder, name in matches:
...     print(f'Give me {name}')
...
Give me adjective
Give me noun
Give me preposition
Give me noun
```

Figure 17.1. Since the list contains 2-tuples, we can unpack them into two variables in the `for` loop.

```
for placeholder, name in [('<adjective>', 'adjective')]:  
    print(f'Give me {name}')
```

You should insert the correct article ("a" or "an", see the "Crow's Nest" exercise) to use as the prompt for `input()`.

17.1.2 Halting and printing errors

If you find there are no placeholders in the text, you need to print an error message. It's common to print *error* message to `STDERR` (standard error), and the `print()` function allows us to specify a `file` argument. We'll use `sys.stderr` just as we did in the "Abuse" chapter. To do that, be sure you import that module:

```
import sys
```

You may recall that `sys.stderr` is like an already open file handle, so there's no need to `open()` it:

```
print('This is an error!', file=sys.stderr)
```

If there really are no placeholders, then we should exit the program *with an error value* to indicate to the operating system that our program failed to run properly. The normal exit value for a program is `0` as in "zero errors," so we need to exit with some `int` value that is *not* `0`. I always use `1`:

```
sys.exit(1)
```

One of the tests checks if your program can detect missing placeholders and if your program exits correctly.¹⁷

17.1.3 Getting the values

For each one of those parts of speech, you need a value that will come either from the `--inputs` argument or directly from the user. If we have nothing for `--inputs`, then you can use the `input()` function to get some answer from the user. The function takes a `str` value to use as a prompt:

```
>>> value = input('Give me an adjective: ')  
Give me an adjective: blue
```

And returns a `str` value of whatever the user typed before hitting the Return key:

```
>>> value  
'blue'
```

¹⁷ In Perl, there is a function called `die` which prints a message to `STDERR` and then halts the program with a non-zero value. If you like that idea, I released a module called `dire` that has a `die` function which you can use!

If, however, you have values for the inputs, use those and do not bother with the `input()` function. I'm only making you handle this option for testing purposes, so you can assume you will always have the correct number of inputs for the number of placeholders in the text provided in the same order as the placeholders they should replace.

```

surly      car      under      bicycle
↓         ↓        ↓          ↓

```

The quick <adjective> <noun> jumps <preposition> the lazy <noun>.

So assume this:

```
>>> inputs = ['surly', 'car', 'under', 'bicycle']
```

You need to remove and return the first string, "surly," from `inputs`. The `list.pop()` method is what you need, but it wants to remove the *last* element by default:

```
>>> inputs.pop()
'bicycle'
```

The `list.pop()` method takes an optional argument to indicate the index of the element you want to remove. Can you figure out how to make that work? Be sure to read `help(list.pop)` if you're stuck.

17.1.4 Substituting the text

When you have values for each of the placeholders, you will need to substitute them into the text. I suggest you look into the `re.sub()` function that will *substitute* text matching a given regular expression for some given text. I would definitely recommend you read `help(re.sub)`:

```
sub(pattern, repl, string, count=0, flags=0)
    Return the string obtained by replacing the leftmost
    non-overlapping occurrences of the pattern in string by the
    replacement repl.
```

I don't want to give away the ending, but you will need to use a pattern similar to the one above to replace each <placeholder> with each value.

Note that it's not a requirement that you use the `re` functions to solve this. I would challenge you, in fact, to try writing a manual solution that does not use the `re` module at all! Now go write the program and use the tests to guide you!

17.2 Solution

```
#!/usr/bin/env python3
"""
Mad Libs"""

import argparse
import re
```

```

import sys

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

    parser = argparse.ArgumentParser(
        description='Mad Libs',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file',                                     ①
                        metavar='FILE',
                        type=argparse.FileType('r'),
                        help='Input file')

    parser.add_argument('-i',                                     ②
                        '--inputs',
                        help='Inputs (for testing)',
                        metavar='input',
                        type=str,
                        nargs='*')

    return parser.parse_args()

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

    args = get_args()
    inputs = args.inputs
    text = args.file.read().rstrip()                                ③
    blanks = re.findall('(<[^>]+>) ', text)                  ④

    if not blanks:                                                 ⑤
        print(f'{args.file.name} has no placeholders.', file=sys.stderr) ⑥
        sys.exit(1)                                              ⑦

    tmpl = 'Give me {} {}: '                                     ⑧
    for placeholder, pos in blanks:
        article = 'an' if pos.lower()[0] in 'aeiou' else 'a' ⑩
        answer = inputs.pop(0) if inputs else input(tmpl.format(article, pos)) ⑪
        text = re.sub(placeholder, answer, text, count=1)      ⑫

    print(text) ⑬

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

```

- ① The file argument should be a readable file.
- ② The --inputs option may have zero or more strings.
- ③ Read the input file, stripping off the trailing newline.
- ④ Use a regex to find all the matches for a left angle bracket followed by one or more of anything that is not a left or right angle bracket followed by a right angle bracket. Use two capture groups to capture the entire expression and the text inside the brackets.
- ⑤ Check if there are no placeholders.

- ⑥ Print a message to STDERR that the given file name contains no placeholders.
- ⑦ Exit the program with a non-zero status to indicate an error to the operating system.
- ⑧ Create a string template for the prompt to ask for input() from the user.
- ⑨ Iterate through the blanks, unpacking each tuple into variables.
- ⑩ Choose the correct article based on the first letter of the name of the part of speech (pos), "an" for those starting with a vowel and "a" otherwise.
- ⑪ If there are inputs, remove the first one for the answer, otherwise use the input() to prompt the user for a value.
- ⑫ Replace the current placeholder text with the answer from the user. Use count=1 to ensure that only the first value is replaced. Overwrite the existing value of text so that all the placeholders will be replaced by the end of the loop.
- ⑬ Print the resulting text to STDOUT.

17.3 Discussion

17.3.1 Defining the arguments

If you define the file with type=argparse.FileType('r'), then argparse will verify that the value is a file, creating an error and usage if it is not, and then will open() it for you. Quite the time saver. I also define --inputs with nargs='*' so that I can get any number of strings as a list. If nothing is provided, the default value will be None, so be sure you don't assume it's a list and try doing list operations on a None.

17.3.2 Substituting with regular expressions

There is a subtle bug waiting for you in using re.sub(). Suppose we have replaced the first <adjective> with "blue" so that we have this:

```
>>> text = 'The quick blue <noun> jumps <preposition> the lazy <noun>.'
```

Now we want to replace <noun> with "dog," and so we try this:

```
>>> text = re.sub('<noun>', 'dog', text)
```

Let's check on the value of text now:

```
>>> text
'The quick blue dog jumps <preposition> the lazy dog.'
```

Since there were two instances of the string <noun>, both got replaced with "dog."

```
re.sub('<noun>', 'dog', 'The quick blue <noun> jumps <preposition> the lazy <noun>.'
```

We must use count=1 to ensure that only the first occurrence is changed:

```
>>> text = 'The quick blue <noun> jumps <preposition> the lazy <noun>.'
>>> text = re.sub('<noun>', 'dog', text, count=1)
>>> text
'The quick blue dog jumps <preposition> the lazy <noun>.'
```

```
re.sub('<noun>', 'dog', 'The quick blue <noun> jumps <preposition> the lazy <noun>.', count=1)
```

And now we can keep moving to replace the other placeholders.

17.3.3 Finding the placeholders without regular expressions

I trust the explanation of the regex solution in the introduction was sufficient. I find that solution fairly elegant, but it is certainly possible to solve this without using regexes. Here is how I might solve it manually.

First I need a way to search the text for <...>. I *start off* by writing a test that helps me imagine what I might give to my function and what I might expect in return for both good and bad values. I decided to return None when the pattern is missing and to return a tuple of (start, stop) indices when the pattern is present:

```
def test_find_brackets():
    """Test for finding angle brackets"""
    assert find_brackets('') == None
    assert find_brackets('<>') == None
    assert find_brackets('<x>') == (0, 2)
    assert find_brackets('foo <bar> baz') == (4, 8)
```

- ① There is no text, so it should return None.
- ② There are angle brackets but they lack any text inside, so this should return None.
- ③ The pattern should be found at the beginning of a string.
- ④ The pattern should be found further into the string.

Now to write the code that will satisfy that test. Here is what I wrote:

```
def find_brackets(text):
    """Find angle brackets"""
    start = text.index('<') if '<' in text else -1
    stop = text.index('>') if start >= 0 and '>' in text[start + 2:] else -1
    return (start, stop) if start >= 0 and stop >= 0 else None
```

- ① Find the index of the left bracket if one is found in the text.
- ② Find the index of the right bracket if one is found starting two positions after the left.
- ③ If both brackets were found, return a tuple of their start and stop positions, otherwise return None.

This function works well enough to pass the given tests, but is not quite correct because it will return a region that contains unbalanced brackets:

```
>>> text = 'foo <<bar> baz'
>>> find_brackets(text)
[4, 9]
>>> text[4:10]
'<<bar>'
```

That may seem unlikely, but I chose angle brackets to make you think of HTML tags like <head> and . HTML is notorious for being incorrect, maybe because it was hand-generated by a human who messed up a tag or because some tool that generated

the HTML had a bug. The point is that most web browsers have to be fairly relaxed in parsing HTML, and it would not be unexpected to see a malformed tag like <> instead of the correct <head>.

The regex version, on the other hand, specifically guards against matching internal brackets by using the class [^<>] to define text that cannot contain any angle brackets. I could write a version of `find_brackets()` that finds only balanced brackets, but, honestly, it's just not worth it. This function points out that one of the strengths of the regex engine is that it can find a partial match (the first left bracket), see that it's unable to make a complete match, and start over (at the next left bracket). Writing this myself would be tedious and, frankly, not that interesting.

Still, this function works for all the given test inputs. Note that it only returns one set of brackets at a time. This is because I will alter the text after I find each set of brackets which will likely change the start and stop positions of any following brackets, so it's best to handle one set at a time.

Here is how I would incorporate it into the `main` function:

```
def main():
    args = get_args()
    inputs = args.inputs
    text = args.file.read().rstrip()
    had_placeholders = False
    tmpl = 'Give me {} {}: ' ①

    while True:
        brackets = find_brackets(text) ②
        if not brackets:
            break ③

        start, stop = brackets ④
        placeholder = text[start:stop + 1] ⑤
        pos = placeholder[1:-1] ⑥
        article = 'an' if pos.lower()[0] in 'aeiou' else 'a' ⑦
        answer = inputs.pop(0) if inputs else input(tmpl.format(article, pos)) ⑧
        text = text[0:start] + answer + text[stop + 1:] ⑨
        had_placeholders = True ⑩

    if had_placeholders:
        print(text) ⑪
    else:
        print(f'{args.file.name} has no placeholders.', file=sys.stderr) ⑫
        sys.exit(1) ⑬
```

- ① Create a variable to track whether we find placeholders. Assume the worst.
- ② Create a template for the `input()` prompt.
- ③ Start an infinite loop. The `while` will continue as long as it has a "truthy" value as `True` will always be.
- ④ Call the `find_brackets()` function with the current value of `text`.
- ⑤ If the return is `None`, then this will be "falsely."
- ⑥ If there are no brackets found, use `break` to exit the infinite `while` loop.
- ⑦ Now that we know `brackets` is not `None`, unpack the `start` and `stop` values.

- ⑧ Find the entire <placeholder> value by using a string slice with the start and stop values, adding 1 to the stop to include that index.
- ⑨ The "part of speech" is the bit inside, so this will extract adjective from <adjective>.
- ⑩ Choose the correct article for the part of speech.
- ⑪ Get the answer from the inputs or from an input() call.
- ⑫ Overwrite the text using a string slice up to the start, the answer, and then the rest of the text from the stop.
- ⑬ Note that we saw a placeholder.
- ⑭ We exit the loop when we no longer find placeholders. Check if we ever saw one.
- ⑮ If we did see a placeholder, print the new value of the text.
- ⑯ If we never saw a placeholder, print an error message to STDERR.
- ⑰ Exit with a non-zero value to indicate an error.

17.4 Review

- Regular expressions are almost like functions where we *describe* the patterns we want to find. The regex engine will do the work of trying to find the patterns, handling mismatches and starting over to find the pattern in the text.
- Regex patterns with * or + are "greedy" in that they match as many characters as possible. Adding a ? after them makes them "not greedy" so that they match as *few* characters as possible.
- The re.findall() function will return a list of all the matching strings or capture groups for a given pattern.
- The re.sub() function will substitute a pattern in some text with new text.
- You can halt your program at any time using the sys.exit() function. If given no arguments, the default exit value will be 0 to indicate no errors. If you wish to indicate there was an error, use any non-zero value such as 1.

17.5 Going Further

- Extend your code to find all the HTML tags enclosed in <...> and </...> in a web page you download from the Internet.
- Write a program that will look for unbalanced open/close pairs for parentheses (), square brackets [], and curly brackets {}. Create input files that have balanced and unbalanced text, and write tests that verify your program identifies both.

18

Gematria: Numeric encoding of text using ASCII values

Gematria is a system for assigning a number to a word by summing the numeric values of each of the characters¹⁸. In the standard encoding (*Mispar hechrechi*), each character of the Hebrew alphabet is assigned a numeric value ranging from 1 to 400, but there are more than a dozen other methods for calculating the numeric value for the letters. To encode a word, these values are added together. Revelation 13:18 from the Christian Bible says "Let the one who has insight calculate the number of the wild beast, for it is a man's number, and its number is 666." Some scholars believe that number is derived from the encoding of the characters representing Nero Caesar's name and title and was used as a way of writing about the Roman emperor without naming him.



We will write a program called `gematria.py` that will numerically encode each word in a given text by similarly adding numeric values for the characters in each word. There are many ways we could assign these values; for instance, we could start by giving "a" the value 1, "b" the value 2, and so forth. Instead, we will use the ASCII table¹⁹ to derive a numeric for English alphabet characters. For non-English characters, we could consider using a Unicode value, but this exercise will stick to ASCII letters.

The input text may be given on the command line:

```
$ ./gematria.py 'foo bar baz'
```

¹⁸ en.wikipedia.org/wiki/Gematria

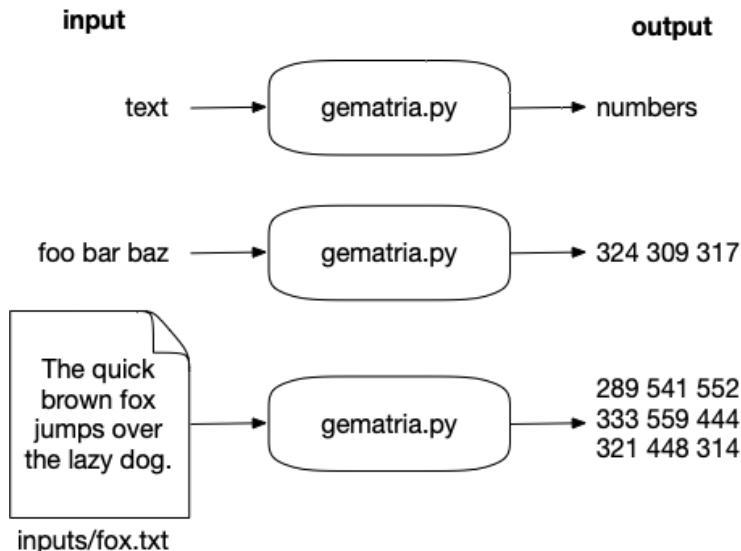
¹⁹ en.wikipedia.org/wiki/ASCII

```
324 309 317
```

Or in a file:

```
$ ./gematria.py ../inputs/fox.txt
289 541 552 333 559 444 321 448 314
```

Here is a string diagram showing how the program should work:



In this exercise, you will:

- Learn about the `ord()` and `chr()` functions
- Explore how characters are organized in the ASCII table
- Understand character ranges used in regular expression
- Use the `re.sub()` function
- Learn how `map()` can be written without `lambda`
- Use the `sum()` function and see how that relates to using `reduce()`
- Learn how to perform case-insensitive string sorting

18.1 Writing `gematria.py`

I will always recommend you start your programs in some way that avoids having to type all the boilerplate text. Either copy `template/template.py` to `18_gematria/gematria.py` or use `new.py gematria.py` in the `18_gematria` directory to create a starting point. Modify the program until it prints the following usage if given no arguments or the `-h` or `--help` flag:

```
$ ./gematria.py -h
usage: gematria.py [-h] text
```

```
Gematria

positional arguments:
  text      Input text or file

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

As in previous exercises, the input may come from the command line or from a file. I suggest you copy the code you used in "Howler" to handle this, then modify your `main()` to the following:

```
def main():
    args = get_args()
    print(args.text)
```

Verify that your program will print text from the command line:

```
$ ./gematria.py 'Death smiles at us all, but all a man can do is smile back.'
Death smiles at us all, but all a man can do is smile back.
```

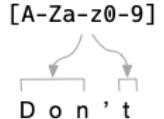
Or from a file:

```
$ ./gematria.py ../inputs/spiders.txt
Don't worry, spiders,
I keep house
casually.
```

18.1.1 Cleaning a word

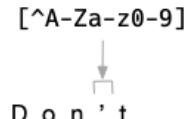
Let's discuss how a single word will be encoded as it will affect how we will break the text in the next section. In order to be absolutely sure we are only dealing with ASCII values, let's remove anything that is not an upper- or lowercase English alphabet character or any of the Arabic numerals 0-9. We can define that class of characters using the regular expression `[A-Za-z0-9]`. We can use the `re.findall()` function we used in "Mad Libs" to find all the characters in word that match this class. In the word "Don't," we should expect to find everything except the apostrophe:

```
>>> re.findall('[A-Za-z0-9]', "Don't")
['D', 'o', 'n', 't']
```

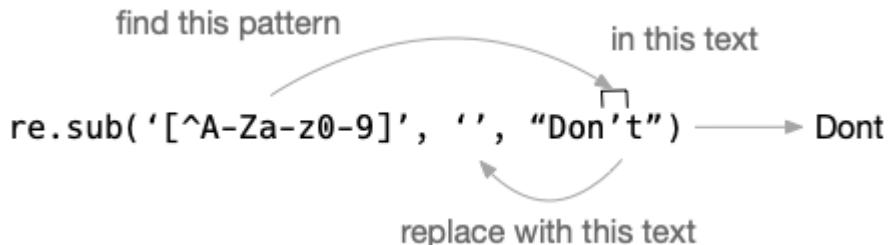


Anything *not* in that class can be defined by placing a caret (^ or "hat") as the first character inside the class like `[^A-Za-z0-9]`. Now we would expect to match *only* the apostrophe:

```
>>> import re
>>> re.findall('[^A-Za-z0-9]', "Don't")
["'"]
```



We can use the `re.sub()` function to replace any characters in that second class with the empty string. As you learned in "Mad Libs," this will replace *all* occurrences of the pattern unless we use the `count=n` option:



```
>>> word = re.sub('[^A-Za-z0-9]', '', "Don't")
>>> word
'Dont'
```

We will want use this operation to clean each word that we'll encode.

18.1.2 *Ordinal character values and ranges*

We will encode a string like "Dont" by converting *each character* to a numeric value and then adding them together, so let's first figure out how to encode a single character. Python has a function called `ord()` that will convert a character to its "ordinal" value (its order in the ASCII table):

```
>>> ord('D')
68
>>> ord('o')
111
```

The `chr()` function works in reverse to convert a number to a character:

```
>>> chr(68)
'D'
>>> chr(111)
'o'
```

Here is a table showing the lower-order ASCII values 0-127 ²⁰. For simplicity's sake, I show "NA" for "not available" for the ones up to index 31 as they are not printable:

\$./asciitbl.py	0 NA	16 NA	32 SPACE	48 0	64 @	80 P	96 `	112 p
	1 NA	17 NA	33 !	49 1	65 A	81 Q	97 a	113 q
	2 NA	18 NA	34 "	50 2	66 B	82 R	98 b	114 r
	3 NA	19 NA	35 #	51 3	67 C	83 S	99 c	115 s
	4 NA	20 NA	36 \$	52 4	68 D	84 T	100 d	116 t
	5 NA	21 NA	37 %	53 5	69 E	85 U	101 e	117 u
	6 NA	22 NA	38 &	54 6	70 F	86 V	102 f	118 v
	7 NA	23 NA	39 '	55 7	71 G	87 W	103 g	119 w
	8 NA	24 NA	40 (56 8	72 H	88 X	104 h	120 x
	9 NA	25 NA	41)	57 9	73 I	89 Y	105 i	121 y

²⁰ I included the `asciitbl.py` program I used to create this.

10 NA	26 NA	42 *	58 :	74 J	90 Z	106 j	122 z
11 NA	27 NA	43 +	59 ;	75 K	91 [107 k	123 {
12 NA	28 NA	44 ,	60 <	76 L	92 \	108 l	124
13 NA	29 NA	45 -	61 =	77 M	93]	109 m	125 }
14 NA	30 NA	46 .	62 >	78 N	94 ^	110 n	126 ~
15 NA	31 NA	47 /	63 ?	79 O	95 _	111 o	127 DEL

We can use a for loop to cycle through all the characters in a string:

```
>>> word = "Dont"
>>> for char in word:
...     print(char, ord(char))
...
D 68
o 111
n 110
t 116
```

Note that upper- and lowercase letters have different `ord()` values. It makes sense because they are two different letters:

```
>>> ord('D')
68
>>> ord('d')
100
```

We can iterate over the values from "a" to "z" by finding their `ord()` values:

```
>>> [chr(n) for n in range(ord('a'), ord('z') + 1)]
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']
```

The letters "a" through "z" lies contiguously in the ASCII table. You can do the same for "A" to "Z" and "0" to "9" which is why we can use `[A-Za-zA-Z0-9]` as a regex.

Note that the uppercase letters have lower ordinal values than their lowercase versions, which is why you cannot use the range `[a-z]`. Try this in the REPL and note the error you get:

```
>>> re.findall('[a-Z]', word)
```

The last line I see is this:

```
re.error: bad character range a-Z at position 1
```

You *can* use the range `[A-z]`:

```
>>> re.findall('[A-z]', word)
['D', 'o', 'n', 't']
```

But see that that "Z" and "a" are not contiguous:

```
>>> ord('Z'), ord('a')
(90, 97)
```

There are other characters in between them:

```
>>> [chr(n) for n in range(ord('Z') + 1, ord('a'))]
['[', '\\', '^', '_', '`']
```

If we try to use that range on all the printable characters, you'll see that it matches characters that are not letters:

```
>>> import string
>>> re.findall('[A-z]', string.printable)
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z',
 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z',
 '[', '\\', '^', '_', '`']
```

That is why it is safest to specify the characters we want as the three ranges, `[A-Za-z0-9]`, which you may sometimes hear pronounced as "A to Z, a to z, zero to nine" as it assumes you understand that there are two "a to z" ranges which are distinct according to their case.



18.1.3 Summing and reducing

Let's keep reminding ourselves what the goal is here: convert all the characters in a word and then sum those values. There is a handy Python function called `sum()` that will add a list of numbers:

```
>>> sum([1, 2, 3])
6
```

We can manually encode the string "Dont" by calling `ord()` on each letter and passing the results as a list to `sum()`:

```
>>> sum([ord('D'), ord('o'), ord('n'), ord('t'))])
405
```

So the question is how to apply the function `ord()` to all the elements of a str and give the result to `sum()`. You've seen this pattern many times now. What's the first tool you'll reach for? We can always start with our handy `for` loop:

```
>>> word = 'Dont'
>>> vals = []
>>> for char in word:
...     vals.append(ord(char))
...
>>> vals
[68, 111, 110, 116]
```

Can you see how to make that into a single line using a list comprehension?

```
>>> vals = [ord(char) for char in word]
>>> vals
[68, 111, 110, 116]
```

And from there, we can move to a `map()`:

```
>>> vals = map(lambda char: ord(char), word)
>>> list(vals)
[68, 111, 110, 116]
```

Here I'd like to show that the `map()` version here doesn't even need the `lambda` declaration because the `ord()` function expects a single value which is exactly what it will get from `map()`. Here is a nicer way to write it:

```
>>> vals = map(ord, word)
>>> list(vals)
[68, 111, 110, 116]
```

To my eye, that is a really beautiful piece of code! Now we can `sum()` that to get a final value for our `word`:

```
>>> sum(map(ord, word))
405
```

Which is correct:

```
>>> sum([68, 111, 110, 116])
405
```

Using `functools.reduce()`

If Python has a `sum()` function, you might suspect it also has a `product()` function to multiply a list of numbers together. Alas, this is not a built-in function, but it does represent a common idea of *reducing* a list of values into a single value. The `reduce()` function from the `functools` module provides a generic way to reduce a list. Let's consult the documentation for how to use it:

```
>>> from functools import reduce
>>> help(reduce)
reduce(...)
    reduce(function, sequence[, initial]) -> value
```

Apply a function of two arguments cumulatively to the items of a sequence, from left to right, so as to reduce the sequence to a single value.
For example, `reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])` calculates $((1+2)+3)+4)+5$. If `initial` is present, it is placed before the items of the sequence in the calculation, and serves as a default when the sequence is empty.

This is another higher-order function which wants *another function* as the first argument, just like `map()` and `filter()`. The documentation shows us how to write our own `sum()` function:

```
>>> reduce(lambda x, y: x + y, [1, 2, 3, 4, 5])
15
```

If we change the `+` operator to `*`, then we have a product:

```
>>> reduce(lambda x, y: x * y, [1, 2, 3, 4, 5])
120
```

Here is how you might write a function for this:

```
def product(vals):
    return reduce(lambda x, y: x * y, vals)
```

And now you can call it:

```
>>> product(range(1,6))
120
```

Instead of writing our own `lambda`, we can use *any function that expects two arguments*. The operator.`mul` function fits this bill:

```
>>> import operator
>>> help(operator.mul)
mul(a, b, /)
    Same as a * b.
```

So it would be easier to write this:

```
def product(vals):
    return reduce(operator.mul, vals)
```

Fortunately, the `math` module also contains a `prod()` function you can use:

```
>>> import math
>>> math.prod(range(1,6))
120
```

If you think about it, the `str.join()` method also reduces a `list` of strings to a single `str` value. Here's how we can write our own:

```
def join(sep, vals):
    return reduce(lambda x, y: x + sep + y, vals)
```

I much prefer the syntax of calling this `join` over the `str.join()` function:

```
>>> join(', ', ['Hey', 'Nonny', 'Nonny'])
'Hey, Nonny, Nonny'
```

Whenever you have a `list` of any values that you want to combine to produce a single value, consider using the `reduce()` function!

18.1.4 Encoding the words

That was a lot just to get to summing the ordinal values of the characters, but wasn't it fascinating to explore? Let's get back on track, though! We can create a function to

encapsulate the idea of converting a word into a numeric value derived from summing the ordinal values of the characters. I called mine `word2num()`, and here is my test:

```
def test_word2num():
    """Test word2num"""
    assert word2num("a") == "97"
    assert word2num("abc") == "294"
    assert word2num("ab'c") == "294"
    assert word2num("4a-b'c,") == "346"
```

Notice that my function returns a `str` value, not an `int`. This is because I want to use the result with the `str.join()` function that only accepts `str` values. So '`405`' instead of `405`:

```
>>> from gematria import word2num
>>> word2num("Don't")
'405'
```

To summarize, the `word2num()` function accepts a word, removes unwanted characters, converts the remaining characters to `ord()` values, and returns a `str` representation of the `sum()` of those values.

18.1.5 Breaking the text

The tests expect you to maintain the same line breaks as the original text, so I recommend you use `str.splitlines()` as in other exercises. In "Friar" and "Scrambler," we used different regexes to split each line into "words," a process sometimes called "tokenization" in programs that deal with Natural Language Processing (NLP). If you write a `word2num()` that passes the tests I provide, then you can use `str.split()` to break a line on spaces because the function will ignore anything that is not a character or number. You are, of course, welcome to break the line into words using whatever means you like.

This code will maintain the line breaks and reconstruct the text. Can you modify it to add the `word2num()` function so that it instead prints out encoded words?

```
def main():
    args = get_args()
    for line in args.text.splitlines():
        for word in line.split():
            # what goes here?
            print(''.join(line.split()))
```

The quick brown fox jumps over the lazy dog.



289 541 552 333 559 444 321 448 314

The output will be one number for each word:

```
$ ./gematria.py ../inputs/fox.txt
289 541 552 333 559 444 321 448 314
```

Time to finish writing the solution. Be sure to use the tests! See you on the flip side.

18.2 Solution

```
#!/usr/bin/env python3
"""Gematria"""

import argparse
import os
import re

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

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

    parser.add_argument('text', metavar='text', help='Input text or file') ①

    args = parser.parse_args() ②

    if os.path.isfile(args.text): ③
        args.text = open(args.text).read().rstrip() ④

    return args ⑤

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

    args = get_args() ⑥

    for line in args.text.splitlines():
        print(' '.join(map(word2num, line.split()))) ⑦ ⑧

# -----
def word2num(word): ⑨
    """Sum the ordinal values of all the characters"""

    return str(sum(map(ord, re.sub('[^A-Za-z0-9]', '', word)))) ⑩

# -----
def test_word2num(): ⑪
    """Test word2num"""

    assert word2num("a") == "97"
    assert word2num("abc") == "294"
    assert word2num("ab'c") == "294"
    assert word2num("4a-b'c,)") == "346"
```

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

- ① The text argument is a string which might be a file name.
- ② Get the parsed command-line arguments.
- ③ Check if the text argument names an existing file.
- ④ If it does, overwrite the args.text with the contents of the file.
- ⑤ Return the fixed up arguments.
- ⑥ Get the parsed arguments.
- ⑦ Split args.text on newlines to retain line breaks.
- ⑧ Split the line on spaces, map() the result through word2num(), then join that result on spaces.
- ⑨ Define a function to convert a word to a number.
- ⑩ Use re.sub() to remove anything not an alpha-numeric character, map() the resulting string through the ord() function, sum() the ordinal values of the characters, and return a str representation of the number.
- ⑪ Define a function to test the word2num() function.

18.3 Discussion

I trust you understand the get_args() as we've used this exact code several times now. Let's jump to the word2num() function.

18.3.1 Writing word2num()

I could have written the function like this:

```
def word2num(word):
    vals = []
    for char in re.sub('[^A-Za-z0-9]', '', word):
        vals.append(ord(char))

    return str(sum(vals))
```

- ① Initialize an empty list to hold the values.
- ② Iterate all the characters returned from re.sub().
- ③ Convert the character to an ordinal value and append that to the values.
- ④ Sum the values and return a string representation.

That's four lines of code instead of one I wrote. I would at least rather use a list comprehension which collapses three lines of code into one:

```
def word2num(word):
    vals = [ord(char) for char in re.sub('[^A-Za-z0-9]', '', word)]
    return str(sum(vals))
```

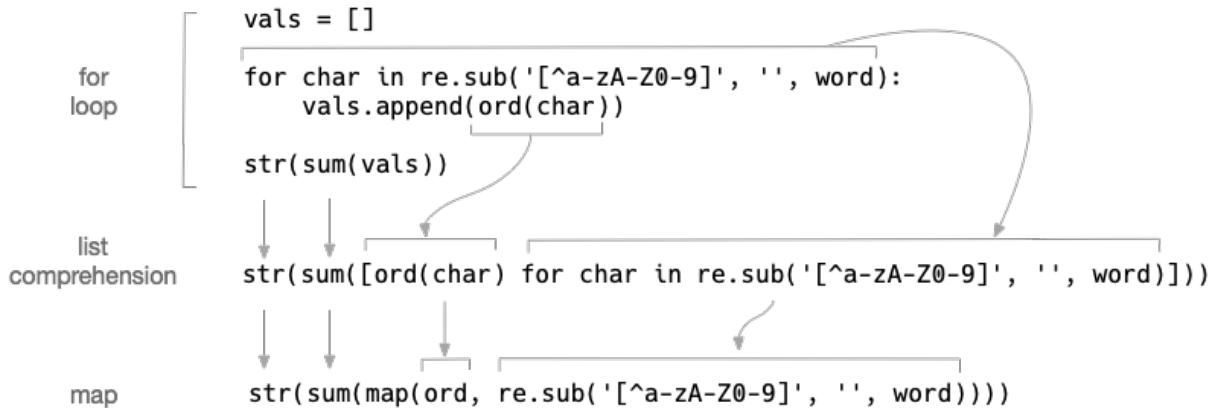
Which can be written in one line though it could be argued that readability suffers:

```
def word2num(word):
    return str(sum([ord(char) for char in re.sub('[^A-Za-z0-9]', '', word)]))
```

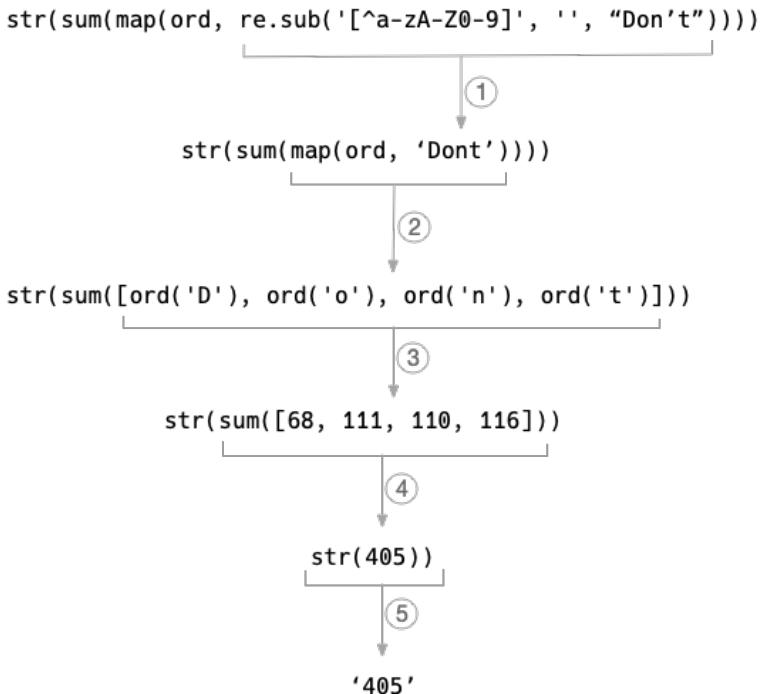
I still think the `map()` version is the most readable and concise:

```
def word2num(word):
    return str(sum(map(ord, re.sub('[^A-Za-z0-9]', '', word))))
```

Figure 18.1. How the for loop, a list comprehension, and a map() relate to each other.



Here's a diagram to help you see how the data moves through the `map()` version with the string "Don't":



1. The `re.sub()` function will replace any character not in the character class with

- the empty string. This will turn a word like "Don't" into "Dont" (without the apostrophe).
2. The `map()` will apply the given function `ord()` to each element of a sequence. Here that "sequence" is a word, so it will use each character of the word.
 3. The result of `map()` is a new `list` where each character from "Dont" is given to the `ord()` function.
 4. The results of the calls to `ord()` will be a `list` of `int` values, one for each letter.
 5. The `sum()` function will reduce a list of numbers to a single value by adding them together.
 6. We will be using the `str.join()` function to create the output, and that function will throw an exception if all the values are not strings, so we use the `str()` function to turn the return from `sum()` into a string representation of the number.

18.3.2 Sorting

The point of this exercise was less about the `ord()` and `chr()` functions and more about exploring regular expressions, function application, and how characters are represented inside programming languages like Python. For instance, sorting of strings is case-sensitive. Note that all the uppercase letters get sorted as a group and then the lowercase:

```
>>> words = 'banana Apple Cherry anchovies cabbage Beets'
>>> sorted(words)
['Apple', 'Beets', 'Cherry', 'anchovies', 'banana', 'cabbage']
```

This is because all the uppercase ordinal values are lower than those of the lowercase letters. In order to perform a case-insensitive sorting of strings, you can use `key=str.casefold`. The `str.casefold()` function will return "a version of the string suitable for caseless comparisons". We are using here *without parentheses* because we are passing *the function itself* as the argument for key:

```
>>> sorted(words, key=str.casefold)
['anchovies', 'Apple', 'banana', 'Beets', 'cabbage', 'Cherry']
```

If you add the parens, it will cause an exception. This is exactly the same as how we pass functions as arguments to `map()` and `filter()`:

```
>>> sorted(words, key=str.casefold())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: descriptor 'casefold' of 'str' object needs an argument
```

The option is the same with `list.sort()` if you prefer to sort the list in-place:

```
>>> words.sort(key=str.casefold)
>>> words
['anchovies', 'Apple', 'banana', 'Beets', 'cabbage', 'Cherry']
```

Unix command-line tools like the `sort` program behave in the same way due to the same representation of characters. Given a file of these same words:

```
$ cat words.txt
banana
Apple
Cherry
anchovies
cabbage
Beets
```

The `sort` program on my Mac²¹ will first sort the uppercase words and then the lowercase:

```
$ sort words
Apple
Beets
Cherry
anchovies
banana
cabbage
```

I have to read the `sort` manual page (via `man sort`) to find the `-f` flag to perform a case-insensitive sort:

```
$ sort -f words
anchovies
Apple
banana
Beets
cabbage
Cherry
```

18.3.3 Testing

I would like to take a moment to point out how often I use my own tests. Every time I write an alternate version of a function or program, I run my own tests to verify that I'm not accidentally showing you buggy code. Having a test suite gives me the freedom and confidence to extensively refactor my programs because I know I can check my work. If I ever find a bug in my code, I add a test to verify that the bug exists. Then I fix the bug and verify that it's handled. I know if I accidentally reintroduce that bug, my tests will catch it!

For the purposes of this book, I try to never write a program over 100 lines. It's common for programs to grow to thousands of lines of code (LOC) spread over dozens of modules. I recommend you start writing and using tests no matter how small you start. It's a good habit to establish early on and will only help you as you write more LOC!

18.4 Review

- The `ord()` function will return the ordinal value of a character which is its position in the ASCII table.
- The `chr()` function will return the character at a given position in the ASCII table.

²¹ The GNU coreutils 8.30 version on one of my Linux machines will perform a case-insensitive sort by default. How does your `sort` work?

- We can use character ranges like `a-z` in regular expressions because the characters lie contiguously in the ASCII table.
- The `re.sub()` function will replace matching patterns of text in a string with new values such as replacing all non-characters with the empty string to remove punctuation and whitespace.
- A `map()` can be written with a direct function reference instead of a `lambda` if the function expects a single positional argument.
- The `sum()` function reduces a list of numbers using addition. We can manually write a version of this using the `functools.reduce()` function. Consider of how that relates to the "map-reduce" concept we discussed before.
- To perform a case-insensitive sort of string values, use `key=str.casefold` option with both the `sorted()` and `list.sort()` functions.

18.5 Going Further

- Create a version that will handle non-English characters by using Unicode values for characters. You'll find that the idea of a "character" is quite different in Unicode. Something that appears to be one entity may actually be composed of more than one Unicode character, hence ideas like "grapheme clusters"²² and "code points"²³.
- Analyze text files to find other words that sum to the value 666. Are these particularly scary words?

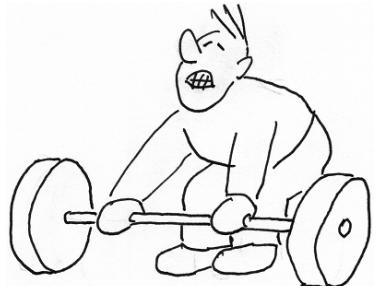
²² unicode.org/reports/tr29/

²³ en.wikipedia.org/wiki/Code_point

19

Workout Of the Day: Parsing CSV file, creating text table output

Several years ago, I joined a workout group. We meet several times a week in our coach's unpaved driveway. We pick up and drop heavy things and run around trying to keep Death at bay for another day. I'm no paragon of strength and fitness, but it's been a nice way to exercise and visit with friends. One of my favorite parts of going is that our coach will write a "Workout Of (the) Day" or "WOD" on the board. Whatever it says is what I do. It doesn't matter if I actually want to do 200 pushups that day, I just get them done no matter how long it takes.²⁴



In that spirit, we'll write a program called `wod.py` to help us create a random daily workout that we have to do, no questions asked:

Exercise	Reps
Pushups	40
Plank	38
Situps	99
Hand-stand pushups	5

²⁴ See "The Paradox of Choice" where increasing the number of choices given to people actually creates more distress and feelings of dissatisfaction is whatever choice is made. Imagine an ice cream shop with three flavors: chocolate, vanilla, and strawberry. If you choose chocolate, you'll likely be happy with that choice. Now imagine the shop has 60 flavors of ice cream including 20 different fruit creams and sorbets and 12 different chocolate varieties from Rocky Road to Fudgetastic Caramel Tiramisu Ripple. Now when you choose a "chocolate" variety, you may leave with remorse about the 11 other kinds you could have chosen. Sometimes having no choice at all provides a sense of calm. Call it fatalism or whatnot.

NOTE

Each time you run the program, you are required to perform all the exercises *immediately*. Heck, even just *reading* them means you have to do them. Like **NOW**. Sorry, I don't make the rules. Better get going on those situps!

We'll choose from a list of exercises stored in a *delimited text file*. In this case, the "delimiter" is the comma, and it will separate each field value. Data files that use commas as delimiters are often described as "comma-separated values" or CSV files. Usually the first line of the file names the columns, and each subsequent line represents a row in the table:

```
$ head -3 inputs/exercises.csv
exercise,reps
Burpees,20-50
Situps,40-100
```

In this exercise, you will:

- Parse delimited text files using the `csv` module
- Coerce text values to numbers
- Print tabular data using the `tabulate` module
- Handle missing and malformed data

This chapter and the next are meant to be a step up in how challenging they are. You will be applying many of the skills you've learned in previous chapters, so get ready!

19.1 Writing wod.py

Let's start by taking a look at the usage that should print when run with `-h` or `--help`, Modify your program's parameters until it produces this:

```
$ ./wod.py -h
usage: wod.py [-h] [-f FILE] [-s seed] [-n exercises] [-e]

Create Workout Of (the) Day (WOD)

optional arguments:
  -h, --help            show this help message and exit
  -f FILE, --file FILE CSV input file of exercises (default:
                        inputs/exercises.csv)
  -s seed, --seed seed  Random seed (default: None)
  -n exercises, --num exercises
                        Number of exercises (default: 4)
  -e, --easy             Halve the reps (default: False)
```

Our program will read an input `-f` or `--file` (default `inputs/exercises.csv`) and will produce a `-n` or `--num` of exercises (default 4). There might be an `-e` or `--easy` flag to indicate that the reps should be cut in half. We'll be using the `random` module to choose the exercises, so we'll need to accept a `-s` or `--seed` option to pass to `random.seed()` for testing purposes.

19.1.1 Reading delimited text files

We're going to use three modules that might not be installed on your system already.

1. The `csv` module to parse the input file
2. Tools from the `csvkit` module to look at the input file on the command line
3. The `tabulate` module to format our output table

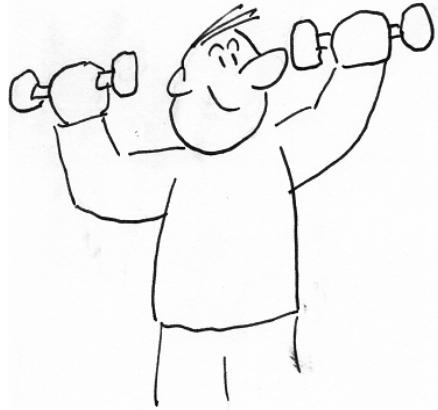
Run this command to install these modules:

```
$ python3 -m pip install csv csvkit tabulate
```

There is also a `requirements.txt` file which is a common way to document the dependencies for a program. You can install all the modules with this command:

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

Despite having "csv" in the name, the `csvkit` can handle just about any delimited text file. For instance, it's typical to use the tab (`\t`) character as a delimiter, too. The module includes many tools that you can read about on their documentation page at csvkit.readthedocs.io/en/1.0.3/. I've included several delimited files in the `19_wod/inputs` directory you can use to test your program. After installing `csvkit`, you should be able to use the `csvlook` program to parse the `inputs/exercises.csv` file into a table structure showing the columns:



```
$ csvlook --max-rows 3 inputs/exercises.csv
exercise | reps
----- | -----
Burpees | 20-50
Situps | 40-100
Pushups | 25-75
... | ...
```

The "reps" column of the input file will have two numbers separated by a dash like `10-20` meaning "from 10 to 20 reps." To select the final value for the "Reps," you will use the `random.randint()` function to select an integer value between the low and high values. When run with a seed, your output should exactly match this:

```
$ ./wod.py --seed 1 --num 3
Exercise      Reps
-----      -----
Pushups        32
Situps         71
Crunches       27
```

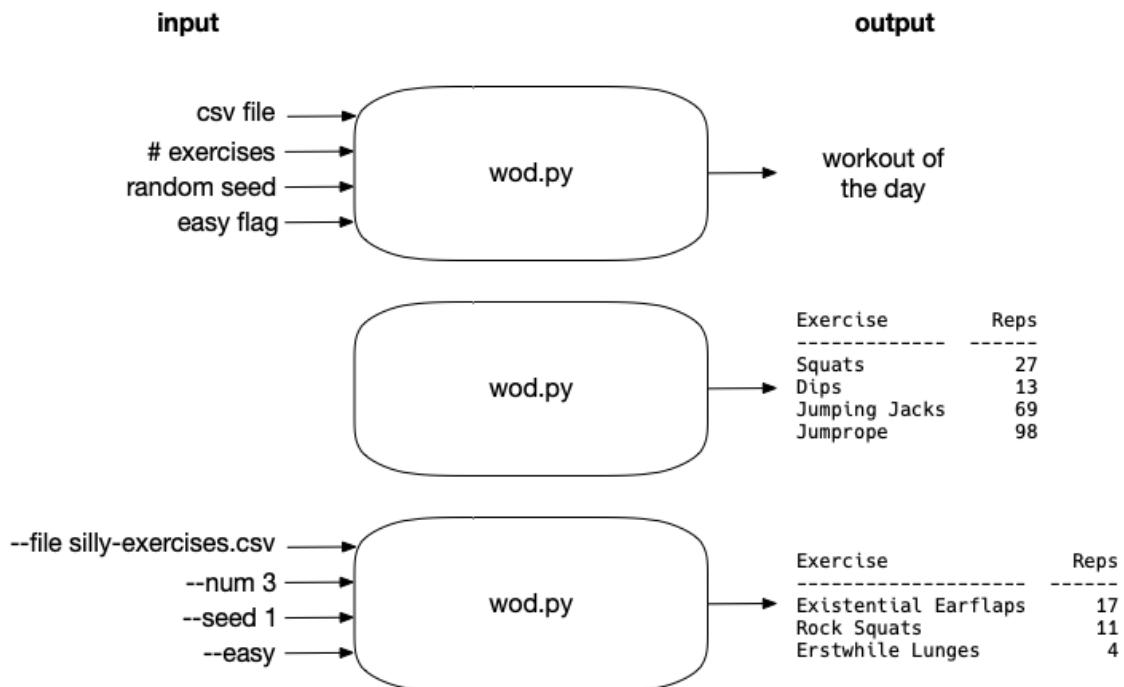
When run with the `--easy` flag, the "reps" should halved:

```
$ ./wod.py --seed 1 --num 3 --easy
Exercise      Reps
-----
Pushups        16
Situps         35
Crunches       13
```

The `--file` option should default to the `inputs/exercises.csv` file, or we can indicate different input file:

```
$ ./wod.py --file inputs/silly-exercises.csv
Exercise      Reps
-----
Hanging Chads   46
Squatting Chinups 46
Rock Squats     38
Red Barchettas  32
```

Here is our trusty string diagram to help you think about it:



19.1.2 Manually reading a CSV file

First I'm going to show you how to manually parse each record from a CSV file into a list of dictionaries, then I'm going to show you how to use the `csv` module to do this more quickly. The reason we want to make a dictionary from each record is so that we can get at the values for each "exercise" and the number of "reps" (repetitions or how

many times to repeat a given exercise). We're going to need to split the reps into low and high values so that we can get a range of numbers to randomly select the number of reps. Finally, we're going to randomly select some of the exercises along with their reps to make a workout. Whew, just describing that was a workout!

Notice that the "reps" is given as a range from a low number to a high number, separated by a dash:

```
$ head -3 inputs/exercises.csv
exercise,reps
Burpees,20-50
Situps,40-100
```

It would be convenient to read this as a list of dictionaries where each column name in the first line is combined with each line of data like this:

```
$ ./manual1.py
[{'exercise': 'Burpees', 'reps': '20-50'},
 {'exercise': 'Situps', 'reps': '40-100'},
 {'exercise': 'Pushups', 'reps': '25-75'},
 {'exercise': 'Squats', 'reps': '20-50'},
 {'exercise': 'Pullups', 'reps': '10-30'},
 {'exercise': 'Hand-stand pushups', 'reps': '5-20'},
 {'exercise': 'Lunges', 'reps': '20-40'},
 {'exercise': 'Plank', 'reps': '30-60'},
 {'exercise': 'Crunches', 'reps': '20-30'}]
```

It may seem like overkill to use a dictionary for a record that contains just two columns, but I regularly deal with records that contain dozens to *hundreds* of columns and then field names are essential! A dictionary is really the only sane way to handle most delimited text files, so it's good to learn using a small example like this.

Let's look at the `manual1.py` code that will do this:

```
#!/usr/bin/env python3

from pprint import pprint          ①

with open('inputs/exercises.csv') as fh:    ②
    headers = fh.readline().rstrip().split(',') ③
    records = []                            ④
    for line in fh:
        rec = dict(zip(headers, line.rstrip().split(','))) ⑤
        records.append(rec)                  ⑥

pprint(records)                      ⑦
```

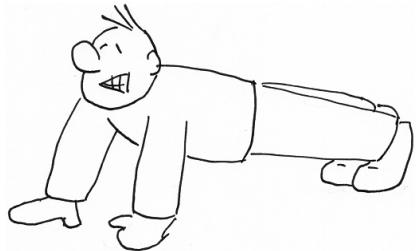
- ① We will use the "pretty print" module to print the data structure.
- ② Use the `with` construct to `open()` the exercises as the `fh` variable. One advantage to use `with` is that the file handle will be closed automatically when the code moves beyond the `with` block.
- ③ Use `fh.readline()` to read only the first line of the file. Remove the whitespace from the right side (`str.rstrip()`), and then `str.split()` the resulting string on commas to create a list of strings in `headers`.
- ④ Initialize `records` as an empty list.

- ⑤ Use a for loop to read the rest of the lines of the fh.
- ⑥ Strip and split the line of text into a list of field values. Use the zip() function to create a new list of tuples containing each of the headers with each of the values. Use the dict() function to turn this list of tuples into a dictionary.
- ⑦ Append the rec dictionary onto the records.
- ⑧ Pretty print the records.

Let's break this down a bit more. First we'll open() the file and read the first line:

```
>>> fh = open('exercises.csv')
>>> fh.readline()
'exercise,reps\n'
```

The line still has a newline stuck to it, so we can use the str.rstrip() function to remove that. Note that I need to keep re-opening this file for this demonstration or each subsequent call to fh.readline() would read the next line of text:



```
>>> fh = open('exercises.csv')
>>> fh.readline().rstrip()
'exercise,reps'
```

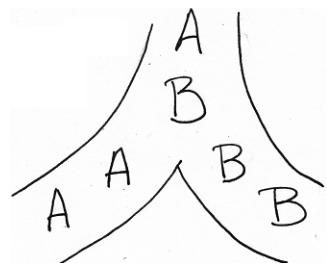
Now let's str.split() that line on the comma to get a list of strings:

```
>>> fh = open('exercises.csv')
>>> headers = fh.readline().rstrip().split(',')
>>> headers
['exercise', 'reps']
```

We can read the next line of the file likewise to get a list of the field values:

```
>>> line = fh.readline().rstrip().split(',')
>>> line
['Burpees', '20-50']
```

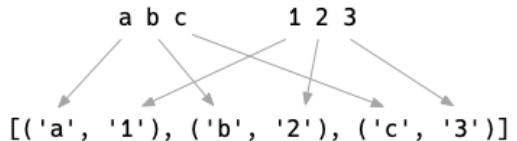
Next I use the zip() function to merge the two lists into one list. Think of two lines of cars merging to exit a parking lot. It's customary for one car from one lane (say "A") to merge into traffic, then a car from the other lane (say "B"). The cars are combining like the teeth of a zipper, and the result is "A," "B," "A," "B," and so forth.



The zip() function will group the elements of lists into tuples, grouping all the

elements in the first position together, then the second position, etc. Note that this is another *lazy* function, so I will use `list` to coerce this in the REPL:

```
>>> list(zip('abc', '123'))
[('a', '1'), ('b', '2'), ('c', '3')]
```



It can work with more than two lists. Note that it will only create groupings for the shortest list. Here, the first two lists have 4 elements ("abcd" and "1234"), but the last has only 3 ("xyz") and so only 3 tuples are created:

```
>>> list(zip('abcd', '1234', 'xyz'))  
[('a', '1', 'x'), ('b', '2', 'y'), ('c', '3', 'z')]
```

In our data, `zip()` will combine the headers and line values "exercise" with "Burpees," "reps" with "20-50":

```
>>> list(zip(headers, line))
[('exercise', 'Burpees'), ('reps', '20-50')]
```

```
zip(['exercise', 'reps'], ['Burpees', '20-50'])  
[('exercise', 'Burpees'), ('reps', '20-50')]
```

That created a list of tuple values. Instead of `list()`, use `dict()` to create a dictionary:

```
>>> rec = dict(zip(headers, line))
>>> rec
{'exercise': 'Burpees', 'reps': '20-50'}
```

Recall that the `dict.items()` function will turn a dict into a list of tuple (key/value) pairs, so you can think of these two data structures as being fairly interchangeable:

```
>>> rec.items()
dict_items([('exercise', 'Burpees'), ('reps', '20-50')])
```

We can drastically shorten our code by replacing the `for` loop with a list comprehension:

```
with open('inputs/exercises.csv') as fh:  
    headers = fh.readline().rstrip().split(',') ①  
    records = [dict(zip(headers, line.rstrip().split(','))) for line in fh] ②  
    pprint(records)
```

① We still need to break out the headers separately by reading the first line.

② This combines the 3 lines of a for loop into a single list comprehension.

We can use `map()` to write the equivalent code:

```

with open('inputs/exercises.csv') as fh:
    headers = fh.readline().rstrip().split(',')
    mk_rec = lambda line: dict(zip(headers, line.rstrip().split(','))) ①
    records = map(mk_rec, fh)
    pprint(list(records))

```

- ① flake8 will complain about assigning this lambda expression. I generally write my code so as to produce no warnings, but I do tend to disagree with this suggestion. I quite like writing one-line functions using a lambda assignment.

In the next section, I'm going to show you how to use the `csv` module to handle much of this code, which may lead you to wonder why I bother showing how to handle this yourself. I'm sorry to tell you that I have to handle data that is terribly formatted such that the first line is almost never the header and there may be other rows of information between the "header" row and the actual data. When you've seen as many badly formatted Excel files as I have, you'll come to appreciate that you sometimes have no choice but to parse the file yourself!

19.1.3 Parsing with the `csv` module

Parsing delimited text files in this way is extremely common. It would not make sense to write or copy this code every time you needed to parse a file. Luckily, some very kind souls have already written some really nice code to do this and have released it as the `csv` module. Let's look at how our code can change if we use `csv.DictReader()` (see `using_csv1.py` in the repo):

```

#!/usr/bin/env python3

import csv
from pprint import pprint

with open('inputs/exercises.csv') as fh:
    reader = csv.DictReader(fh, delimiter=',') ②
    records = [] ③
    for rec in reader: ④
        records.append(rec) ⑤

    pprint(records)

```

- ① Import the `csv` module.
- ② Create a `csv.DictReader()` which will create a dict for each record in the file that zips the headers in the first line with the data values in the subsequent lines. Use the `delimiter` to indicate the `str` value to split each line.
- ③ Initialize an empty list to hold the records.
- ④ Use a `for` loop to iterate through each `rec` (record) in the reader.
- ⑤ The `rec` will be a dict. Append that to the list of records.

This code creates the same list of dict values as before but with much less code:

```

$ ./using_csv1.py
[OrderedDict([('exercise', 'Burpees'), ('reps', '20-50')]),
 OrderedDict([('exercise', 'Situps'), ('reps', '40-100')]),

```

```
OrderedDict([('exercise', 'Pushups'), ('reps', '25-75'))],  
OrderedDict([('exercise', 'Squats'), ('reps', '20-50'))],  
OrderedDict([('exercise', 'Pullups'), ('reps', '10-30'))],  
OrderedDict([('exercise', 'Hand-stand pushups'), ('reps', '5-20'))],  
OrderedDict([('exercise', 'Lunges'), ('reps', '20-40'))],  
OrderedDict([('exercise', 'Plank'), ('reps', '30-60'))],  
OrderedDict([('exercise', 'Crunches'), ('reps', '20-30'))])
```

We can remove the entire for loop and use the `list()` function to coerce the reader to give us that same list. This code (in `using_csv2.py`) will print the same output:

```
with open('inputs/exercises.csv') as fh:  
    reader = csv.DictReader(fh, delimiter=',')  
    records = list(reader)  
    pprint(records)
```

- ① Open the file.
- ② Create a `csv.DictReader()` to read the `fh` using the comma for the delimiter.
- ③ Use the `list()` function to coerce all the values from the reader.
- ④ Pretty-print the records.

19.1.4 Creating a function to read a CSV file

Let's try to imagine how we could write and test a function we might call `read_csv()` to read in our data. I'll start with the `test_read_csv()` definition:

```
def test_read_csv():  
    text = io.StringIO('exercise,reps\nBurpees,20-50\nSitups,40-100')  
    assert read_csv(text) == [('Burpees', 20, 50), ('Situps', 40, 100)]
```

- ① Use `io.StringIO()` to create a mock file handle to wrap around a valid text that we might read from a file. The `\n` represents the newlines that break each line in the input data, and each line uses commas to separate the fields. We previously used `io.StringIO()` in the low-memory version of "Howler."
- ② Affirm that our imaginary `read_csv()` file would turn this text into a list of tuple values with the name of the exercise and the "reps" which have been split into a low and high values. Note that these values have been converted to integers.

Hey, we just did all that work to make a list of dict values, so why am I suggesting that we now create a list of tuple values? I'm looking ahead here to how I'll use the `tabulate` module to print out the result, so just trust me here that this is a good way to go!

Let's go back to using the `csv.DictReader()` to parse our file and think about how we can break the "reps" value into int values for the low and high:

```
reader = csv.DictReader(fh, delimiter=',')  
exercises = []  
for rec in reader:  
    name, reps = rec['name'], rec['reps']  
    low, high = 0, 0 # what goes here?  
    exercises.append((name, low, high))
```

You have a couple of tools at your disposal. Imagine your `reps` is this:

```
>>> reps = '20-50'
```

The `str.split()` function could break that into two strings, "20" and "50":

```
>>> reps.split('-')
['20', '50']
```

How could you turn each of the `str` values into integers?

Another way you could go is to use a regular expression. Remember that `\d` will match a digit, and so `\d+` will match one or more digits. (Refer back to chapter 15 to refresh your memory on `\d` as a shortcut to the character class of digits.) You can wrap that expression in parentheses to capture the "low" and "high" values:

```
>>> match = re.match('(\d+)-(\d+)', reps)
>>> match.groups()
('20', '50')
```

Can you write a `read_csv()` function that passes the above test?

19.1.5 Selecting the exercises

Let's start off by making our `main()` print out the data structure from reading an input file, then we'll modify it to actually print our exercise regimen:

```
def main():
    args = get_args()           ①
    random.seed(args.seed)      ②
    pprint(read_csv(args.file)) ③
```

- ① Get the command-line arguments.
- ② Set the `random.seed()` with the `args.seed` value.
- ③ Read the `args.file` (which will be an open file handle) using the `read_csv()` function and print the resulting data structure. Note that I've imported the `pprint()` function for demonstration purposes.

If you run the above code, you should see this:

```
$ ./wod.py
[('Burpees', 20, 50),
 ('Situps', 40, 100),
 ('Pushups', 25, 75),
 ('Squats', 20, 50),
 ('Pullups', 10, 30),
 ('Hand-stand pushups', 5, 20),
 ('Lunges', 20, 40),
 ('Plank', 30, 60),
 ('Crunches', 20, 30)]
```

We will use the `random.sample()` function to select the `--num` of exercises indicated by the user. Add `import random` to your program and modify your `main` to this:

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

```
random.seed(args.seed)          ①
    exercises = read_csv(args.file) ②
    pprint(random.sample(exercises, k=args.num)) ③
```

- ① Always set your random seed before calling `random` functions.
- ② Read the input file.
- ③ Randomly select the given number of exercises.

Now instead of printing all the exercises, it should print a random sample of the correct number of exercises. In addition, your sampling should exactly match this output because you have set the `random.seed()` value:

```
$ ./wod.py -s 1
[('Pushups', 25, 75),
 ('Situps', 40, 100),
 ('Crunches', 20, 30),
 ('Burpees', 20, 50)]
```

We need to iterate through the sample and select a single "reps" value using the `random.randint()` function. The first exercise is "Pushups," and the range is between 25 and 75 reps:

```
>>> import random
>>> random.seed(1)
>>> random.randint(25, 75)
33
```

If the `args.easy` is `True`, you will need to halve that value. Unfortunately, we cannot have a fraction of a "rep":

```
>>> 33/2
16.5
```

You can use the `int()` function to truncate the number to the integer component:

```
>>> int(33/2)
16
```

19.1.6 *Formatting the output*

Modify your program until it can reproduce this output:

```
$ ./wod.py -s 1
[('Pushups', 56), ('Situps', 88), ('Crunches', 27), ('Burpees', 35)]
```

We will use the `tabulate()` function from the `tabulate` module to format this list of tuple values into a text table:

```
>>> from tabulate import tabulate
>>> wod = [('Pushups', 56), ('Situps', 88), ('Crunches', 27), ('Burpees', 35)]
>>> print(tabulate(wod))
-----
Pushups  56
Situps   88
Crunches 27
```

```
Burpees    35
-----
```

If you read `help(tabulate)`, you will see that there is a `headers` option where you can give a list of strings to use for the headers:

```
>>> print(tabulate(wod, headers=['Exercise', 'Reps']))
Exercise      Reps
-----
Pushups        56
Situps         88
Crunches       27
Burpees        35
```

If you synthesize all these ideas, you should be able to pass the provided tests.

19.1.7 Handling bad data

None of the tests will give your program bad data, but I have provided several "bad" CSV files in the `inputs` directory that you might be interested in figuring out how to handle:

- The `bad-headers-only.csv` is well-formed but has no data. It only has headers.
- The `bad-empty.csv` file is empty. That is, it is a zero-length file that I created with `touch bad-empty.csv` and has no data at all.
- The `bad-headers.csv` has headers that are capitalized, so "Exercise" instead of "exercise," "Reps" instead of "reps."
- The `bad-delimiter.tab` uses the tab character (`\t`) instead of the comma (,) as the field delimiter.
- The `bad-reps.csv` file contains reps that are not in format x-y or which are not numeric or integer values.

Once your program passes the given tests, trying running it on all the "bad" files to see how your program breaks. What should your program do when there is no usable data? Should your program print error messages when it encounters bad or missing values, or should it quietly ignore errors and only print the usable data? These are all real-world concerns that you will encounter, and it's up to you to decide what your program will do. After the solution, I will show you ways I might deal with these files.

19.1.8 Time to write!

OK, enough lollygagging. Time to write this program. You must do 10 pushups every time you find a bug!

Hints:

- Use the `csv.DictReader()` to parse the input CSV files
- Break the `reps` field on the `-` character, coerce the low/high values to `int` values, and then use the `random.randint()` module to choose a random integer in that range.
- Use `random.sample()` to select the correct number of exercises.

- Use the `tabulate` module to format the output into a text table.

19.2 Solution

```

#!/usr/bin/env python3
"""Create Workout Of (the) Day (WOD)"""

import argparse
import csv
import io
import random
from tabulate import tabulate ①

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

    parser = argparse.ArgumentParser(
        description='Create Workout Of (the) Day (WOD)',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('-f',
                        '--file',
                        help='CSV input file of exercises',
                        metavar='FILE',
                        type=argparse.FileType('r'), ②
                        default='exercises.csv')

    parser.add_argument('-s',
                        '--seed',
                        help='Random seed',
                        metavar='seed',
                        type=int,
                        default=None)

    parser.add_argument('-n',
                        '--num',
                        help='Number of exercises',
                        metavar='exercises',
                        type=int,
                        default=4)

    parser.add_argument('-e',
                        '--easy',
                        help='Halve the reps',
                        action='store_true')

    args = parser.parse_args()

    if args.num < 1: ③
        parser.error(f'--num "{args.num}" must be greater than 0')

    return args

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

    args = get_args()

```

```

random.seed(args.seed)
wod = []                                     ④
    exercises = read_csv(args.file)           ⑤

for name, low, high in random.sample(exercises, k=args.num): ⑥
    reps = random.randint(low, high)          ⑦
    if args.easy:                           ⑧
        reps = int(reps / 2)
    wod.append((name, reps))                 ⑨

print(tabulate(wod, headers=('Exercise', 'Reps'))) ⑩

# -----
def read_csv(fh):                            ⑪
    """Read the CSV input"""

    exercises = []                         ⑫
    for row in csv.DictReader(fh, delimiter=','):
        low, high = map(int, row['reps'].split('-')) ⑬
        exercises.append((row['exercise'], low, high)) ⑭

    return exercises                      ⑯

# -----
def test_read_csv():                        ⑯
    """Test read_csv"""

    text = io.StringIO('exercise,reps\nBurpees,20-50\nSitups,40-100') ⑯
    assert read_csv(text) == [('Burpees', 20, 50), ('Situps', 40, 100)] ⑯

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

```

- ① Import the tabulate function to format the output table.
- ② The --file option, if provided, must be a file.
- ③ Ensure that the args.num is a positive value.
- ④ Initialize wod as an empty list.
- ⑤ Read the input file into a list of exercises.
- ⑥ Randomly sample the given number of exercises. The result will be a list of tuples that each contain three values which can be unpacked directly into the variables name, low, and high.
- ⑦ Randomly select a value for the reps that is in the provided range.
- ⑧ If args.easy is "truthy," then cut the reps in half.
- ⑨ Append a tuple of the name of the exercise and the reps to the wod.
- ⑩ Use the tabulate() function to format the wod into a text table using the appropriate headers.
- ⑪ Define a function to read an open CSV file handle.
- ⑫ Initialize exercises to an empty list.
- ⑬ Iterate through the file handle using the csv.DictReader() to create a dictionary combining the column names from the first row with the field values from the rest of the file. Use the comma as the field delimiter.
- ⑭ Split the "reps" column on the dash, turn those values into integers, and assign to low and high.

- ⑯ Append a tuple containing the name of the exercise with the low and high values.
- ⑰ Return the list of exercises to the caller.
- ⑱ Define a function that pytest will use to test the `read_csv()` function.
- ⑲ Create a mock file handle containing valid sample data.
- ⑳ Verify that `read_csv()` can handle valid input data.

19.3 Discussion

How did that go for you? Did you manage to modify your program to gracefully handle all the bad input files? Let's dig into the program, starting with the `read_csv()` function.

19.3.1 Reading a CSV file

In the introduction, I left you with one line where you needed to split the "reps" column and convert the values to integers. Here is one way:

```
def read_csv(fh):
    exercises = []
    for row in csv.DictReader(fh, delimiter=','):
        low, high = map(int, row['reps'].split('-')) ①
        exercises.append((row['exercise'], low, high))

    return exercises
```

- ① Split the "reps" field on the dash, `map()` the values through the `int()` function, assign to `low` and `high`.

The annotated line works as follows. Assume a "reps" value like so:

```
>>> '20-50'.split('-')
['20', '50']
```

We need to turn each of those into an `int` value, which is what the `int()` function will do. We could use a list comprehension:

```
>>> [int(x) for x in '20-50'.split('-')]
[20, 50]
```

But the `map()` is much shorter and easier to read, in my opinion:

```
>>> list(map(int, '20-50'.split('-')))
[20, 50]
```

Since that produces exactly two values, we can assign them to two variables:

```
>>> low, high = map(int, '20-50'.split('-'))
>>> low, high
(20, 50)
```

19.3.2 Potentials run-time errors

This code makes many, many assumptions, however, that will cause it to fail miserably

when the data doesn't match the expectations. For instance, what happens if the "reps" field contains no dash? It will produce one value:

```
>>> list(map(int, '20'.split('-')))
[20]
```

And that will cause a *run-time* exception when we try to assign one value to two variables:

```
>>> low, high = map(int, '20'.split('-'))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not enough values to unpack (expected 2, got 1)
```

What if one or more of the values cannot be coerced to an `int`? It will cause exception, and, again, you won't discover this until you *run* the program with bad data!

```
>>> list(map(int, 'twenty-thirty'.split('-')))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'twenty'
```

What happens if there is no `reps` field in the record as in the case when the field names are capitalized?

```
>>> rec = {'Exercise': 'Pushups', 'Reps': '20-50'}
```

Then the dictionary access `rec['reps']` will cause an exception:

```
>>> list(map(int, rec['reps'].split('-')))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'reps'
```

While the `read_csv()` function seems to work just fine as long as we pass it well-formed data, the real world usually does not always give us clean data sets. An unfortunately large part of my job, in fact, is finding and correcting errors like this.

In the introduction, I suggested you might use a regular expression to extract the low and high values from the "reps" field. A regex has the advantage of inspecting the entire field, ensuring that it looks correct. Here is a more robust way to implement `read_csv()`:

```
def read_csv(fh):
    exercises = []
    for row in csv.DictReader(fh, delimiter=','):
        name, reps = row.get('exercise'), row.get('reps') ①
        if name and reps:
            match = re.match('(\d+)-(\d+)', reps) ②
            if match:
                low, high = map(int, match.groups()) ③
                exercises.append((name, low, high)) ④
    return exercises ⑤
```

- ① Initialize exercises as an empty list.
- ② Iterate through the rows of the data.
- ③ Use the dict.get() function to try to retrieve the values for "exercise" and "reps."
- ④ Check if we have "truthy" values for the exercise name and reps.
- ⑤ Use a regex to look for one or more digits followed by a dash followed by one or more digits. Use capturing parentheses for the digits so they can be extracted.
- ⑥ Check if there was a match. Remember that re.match() will return None to indicate a failure to match.
- ⑦ Unpack the low and high values from the two capture groups, map() them through the int function to coerce the str values. This is safe because we use a regex to verify that they look like digits.
- ⑧ Append the name, low, and high values as a tuple to the exercises.
- ⑨ Return the exercises to the caller. If no valid data was found, then we will return an empty list.

19.3.3 Using pandas.read_csv() to parse the file

I specifically chose the function name `read_csv()` because this is similar to a built-in function in the R programming language called `read.csv` and which was copied in the `pandas.read_csv()` function. Both of these other functions will return a "data frame," a two-dimensional object that allows you to deal with columns and rows of data. To run the `using_pandas.py` version, you'll need to install `pandas` like so:

```
$ python3 -m pip install pandas
```

And now you can try running this program:

```
import pandas as pd

df = pd.read_csv('inputs/exercises.csv')
print(df)
```

To see this output:

```
$ ./using_pandas.py
      exercise      reps
0       Burpees    20-50
1       Situps    40-100
2      Pushups    25-75
3      Squats     20-50
4      Pullups    10-30
5 Hand-stand  pushups    5-20
6      Lunges     20-40
7      Plank      30-60
8     Crunches    20-30
```

Learning how to use `pandas` is far beyond the scope of this book. Mostly I just want you to be aware that this is a very popular way to parse delimited text files, especially if you intend to run statistical analyses over various columns of the data.

19.3.4 Formatting the table

Let's look at the `main()` I included in the solution and notice a run-time exception waiting to happen:

```

def main():
    args = get_args()
    random.seed(args.seed)
    wod = []
    exercises = read_csv(args.file)

    for name, low, high in random.sample(exercises, k=args.num): ①
        reps = random.randint(low, high)
        if args.easy:
            reps = int(reps / 2)
        wod.append((name, reps))

    print(tabulate(wod, headers=('Exercise', 'Reps')))

```

- ① This line will fail if `exercises` is None or an empty list or if `args.num` is greater than the number of elements in `exercises`.

If you test the given solution with the `bad-headers-only.csv` file, then you would see this error:

```

$ ./wod.py -f inputs/bad-headers-only.csv
Traceback (most recent call last):
  File "./wod.py", line 93, in <module>
    main()
  File "./wod.py", line 62, in main
    for name, low, high in random.sample(exercises, k=args.num):
  File "/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/random.py", line
363, in sample
    raise ValueError("Sample larger than population or is negative")
ValueError: Sample larger than population or is negative

```

A safer way to handle this is to check that `read_csv()` returns enough data to pass to `random.sample()`. We have a couple of possible errors:

1. No usable data was found in the input file.
2. We are trying to sample too many records from the file.

To handle these possible errors, I find it useful to create a `die()` function that will print a message to `sys.stderr` and exit the program with a non-zero value. I then use this function to exit when I find problems:

```

def main():
    """Make a jazz noise here"""

    args = get_args()
    random.seed(args.seed)
    exercises = read_csv(args.file) ①

    if not exercises: ②
        die(f'No usable data in --file "{args.file.name}"')

    num_exercises = len(exercises)
    if args.num > num_exercises: ③
        die(f'--num "{args.num}" greater than exercises "{num_exercises}"')

    wod = [] ④
    for name, low, high in random.sample(exercises, k=args.num):
        reps = random.randint(low, high)

```

```

if args.easy:
    reps = int(reps / 2)
    wod.append((name, reps))

print(tabulate(wod, headers=('Exercise', 'Reps')))

def die(msg):          ⑤
    """Print message to STDERR and exit with an error"""

    print(msg, file=sys.stderr)
    sys.exit(1)

```

- ① Read the input file into exercises. The function should only return a list, possibly empty.
- ② See if exercises is "falsey" such as an empty list.
- ③ Check if we are trying to sample too many records.
- ④ Continue after we verify that we have enough valid data.
- ⑤ Define a function to die() where we print a message to sys.stderr and then call sys.exit() with a non-zero value. I stole this function name from Perl, and I released a module called dire that exports this and a warn() function because I sometimes miss those.

The version in `solution2.py` has these updated functions and gracefully handles all the bad input files. You may note that the `test_` functions have been moved to the `unit.py` file such that I can run `pytest -xv unit.py`. The `test_csv()` function became much longer as I tested with various bad inputs, so it seemed more readable to me to move that code to a separate file:

```

import io
from wod import read_csv ①

def test_read_csv():
    """Test read_csv"""

    good = io.StringIO('exercise,reps\nBurpees,20-50\nSitups,40-100') ②
    assert read_csv(good) == [('Burpees', 20, 50), ('Situps', 40, 100)]

    no_data = io.StringIO('')                                     ③
    assert read_csv(no_data) == []

    headers_only = io.StringIO('exercise,reps\n')                 ④
    assert read_csv(headers_only) == []

    bad_headers = io.StringIO('Exercise,Reps\nBurpees,20-50\nSitups,40-100') ⑤
    assert read_csv(bad_headers) == []

    bad_numbers = io.StringIO('exercise,reps\nBurpees,20-50\nSitups,fourty-100') ⑥
    assert read_csv(bad_numbers) == [('Burpees', 20, 50)]

    no_dash = io.StringIO('exercise,reps\nBurpees,20\nSitups,40-100') ⑦
    assert read_csv(no_dash) == [('Situps', 40, 100)]

    tabs = io.StringIO('exercise\treps\nBurpees\t20-40\nSitups\t40-100') ⑧
    assert read_csv(tabs) == []

```

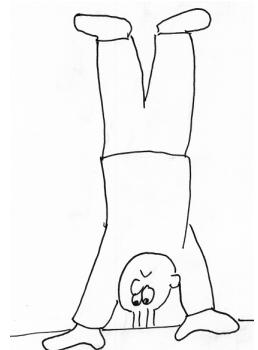
- ① Remember that you can import your own functions from your own modules into other programs. Here we are bringing in our `read_csv()` function. If we had instead used `import wod` then we could call `wod.read_csv()`.
- ② The original, valid input.
- ③ Testing with no data at all.
- ④ Well-formed file (correct headers and delimiter), but no data.
- ⑤ The headers are capitalized and only lowercase are expected.
- ⑥ A string ("forty") that cannot be coerced by `int` to a numeric value.
- ⑦ A "reps" value ("20") missing a dash.
- ⑧ Well-formed data with correct headers but using a tab for the delimiter.

19.4 Review

- The `csv` module is useful for parsing text files delimited by commas and tabs.
- Text values representing numbers must be coerced to numeric values using `int` or `float` in order to be used as numbers.
- The `tabulate` module can be used to create text tables to format tabular output.
- Great care must be taken to anticipate and handle bad and missing data values. Tests can help you imagine all the ways in which your code might fail.

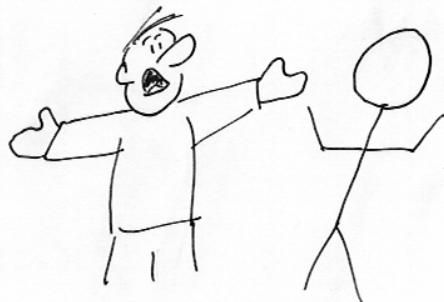
19.5 Going Further

- Add an option to use a different delimiter or guess that the delimiter is a tab if the input file extension is ".tab" as in the `bad-delimiter.tab` file.
- The `tabulate` module supports many table formats including plain, simple, grid, pipe, orgtbl, rst, mediawiki, latex, latex_raw and latex_booktabs. Add an option to choose a different `tabulate` format using these as the valid choices. Choose a reasonable default value.



Password Strength: Generating a secure and memorable password

It's not easy to create passwords that are both difficult to guess and easy to remember. An XKCD comic (xkcd.com/936/) describes an algorithm that provides both security and recall by suggesting that a password be composed of "four random common words." For instance, the comic suggests that the password composed of the words "correct," "horse," "battery," and "staple" would provide "~44 bits of entropy" which would require around 550 years for a computer to guess given 1,000 guess per second.



We're going to write a program called `password.py` that will create passwords by randomly combining the words from some input files. Many computers have a file that lists thousands of English words each on a separate line. On most of my systems, I can find this at `/usr/share/dict/words`, and it contains over 235,000 words! As the file can vary by system, I've added a version the repo so that we can use the same file. This file is a little large, so I've compressed to `inputs/words.txt.zip`. You should unzip it before using it:

```
$ unzip inputs/words.txt.zip
```

Now we should both have the same `inputs/words.txt` file so that this is reproducible

for you:

```
$ ./password.py ../inputs/words.txt --seed 14
CrotalLeavesMeeredLogy
NatalBurreltizzyOddman
UnbornSignerShodDehort
```

Hmm, maybe those aren't going to be the easiest to remember! Perhaps instead we should be a bit more judicious about the source of our words? We're drawing from a pool of over 200K words, but the average speaker tends to use somewhere between 20,000 and 40,000 words.

We can generate more memorable passwords by drawing from some actual piece of English text such as the US Constitution. Note that to use a piece of input text in this way, we will need to remove any punctuation as we have done in previous exercises:

```
$ ./password.py --seed 8 ../inputs/const.txt
DulyHasHeadsCases
DebtSevenAnswerBest
ChosenEmitTitleMost
```

Another strategy for generating memorable words could be to limit the pool of words to more interesting parts of speech like nouns, verbs, and adjectives taken from texts like novels or poetry. I've included a program I wrote called `harvest.py` that uses a Natural Language Processing library in Python called "spaCy" (spacy.io) that will extract those parts of speech into files that we can use as input to our program. If you want to use this program on your own input files, you'll need to be sure to first install the module:

```
$ python3 -m pip install spacy
```

I ran the `harvest.py` program on some texts and placed the outputs into directories in the source repo. For instance, here is the output drawing from nouns found in the US Constitution:

```
$ ./password.py --seed 5 const/nouns.txt
TaxFourthYearList
TrialYearThingPerson
AidOrdainFifthThing
```

And here we have passwords generated using only verbs found in *The Scarlet Letter* by Nathaniel Hawthorne:

```
$ ./password.py --seed 1 scarlet/verbs.txt
CrySpeakBringHold
CouldSeeReplyRun
WearMeanGazeCast
```

And here are some generated from adjectives extracted from William Shakespeare's sonnets:

```
$ ./password.py --seed 2 sonnets/adjs.txt
BoldCostlyColdPale
FineMaskedKeenGreen
```

```
BarrenWiltFemaleSeldom
```

Just in case that is not a strong enough password, we will also provide a `--133t` flag to further obfuscate the text by:

1. Passing the generated password through the `ransom.py` algorithm from Chapter 12
2. Substituting various characters with given table as we did in `jump_the_five.py` from Chapter 4
3. Adding a randomly selected punctuation character to the end

Here is what the Shakespearean passwords look like with this encoding:

```
$ ./password.py --seed 2 sonnets/adjs.txt --133t
B0LDco5TLYColdp@l3,
f1n3M45K3dK3eNGR33N[
B4rReNW1LTFeM4l3seldoM/
```

In this exercise, you will:

- Take an optional list of input files as positional arguments.
- Use a regular expression to remove non-word characters.
- Filter words by some minimum length requirement.
- Use sets to create unique lists.
- Generate some given number of passwords by combining some given number of randomly selected words.
- Optionally encode text using a combination of algorithms we've previously written.

20.1 Writing `password.py`

Our program will be called `password.py` and will create some `--num` number of passwords (default 3) each created by randomly choosing some `--num_words` (default 4) from a unique set of words from one or more input files. As it will use the `random` module, the program will also accept a random `--seed` argument. The words from the input files will need to be a minimum length of some `--min_word_len` (default 3) up to a `--max_word_len` (default 6) after removing any non-characters.

As always, your first priority is to sort out the inputs to your program. Do not move ahead until your program can produce this usage with the `-h` or `--help` flags and can pass the first 8 tests:

```
$ ./password.py -h
usage: password.py [-h] [-n num_passwords] [-w num_words] [-m mininum]
                   [-x maximumm] [-s seed] [-l]
                   FILE [FILE ...]

Password maker

positional arguments:
  FILE                  Input file(s)

optional arguments:
```

```

-h, --help      show this help message and exit
-n num_passwords, --num num_passwords
               Number of passwords to generate (default: 3)
-w num_words, --num_words num_words
               Number of words to use for password (default: 4)
-m mininum, --min_word_len minimum
               Minimum word length (default: 3)
-x maximummm, --max_word_len maximumm
               Maximum word length (default: 6)
-s seed, --seed seed Random seed (default: None)
-l, --133t    Obfuscate letters (default: False)

```

The words from the input files will be title-cased (first letter uppercase, the rest lowercased) which we can achieve using the `str.title()` method. This makes it easier to see and remember the individual words in the output. Note that we can vary the number of words included in each password as well as the number of passwords generated:

```
$ ./password.py --num 2 --num_words 3 --seed 9 sonnets/*
QueenThenceMasked
GullDeemdEven
```

The `--min_word_len` argument helps to filter out shorter, less interesting words like "a", "I," "an," "of," etc., while the `--max_word_len` prevents the passwords from becoming unbearably long. If you increase these values, then the passwords change quite drastically:

```
$ ./password.py -n 2 -w 3 -s 9 -m 10 -x 20 sonnets/*
PerspectiveSuccessionIntelligence
DistillationConscienceCountenance
```

The `--133t` flag is a nod to "leet"-speak where 31337 H4X0R means "ELITE HACKER"²⁵. When this flag is present, we'll encode each of the passwords, first by passing the word through the `ransom()` algorithm we wrote:

```
$ ./ransom.py MessengerRevolutionImportune
MesSENGeRReVolUtionImpoRtune
```

Then we'll use the following substitution table to substitute characters in the same way we did in "Jump the Five":

```

a => @
A => 4
O => 0
t => +
E => 3
I => 1
S => 5

```

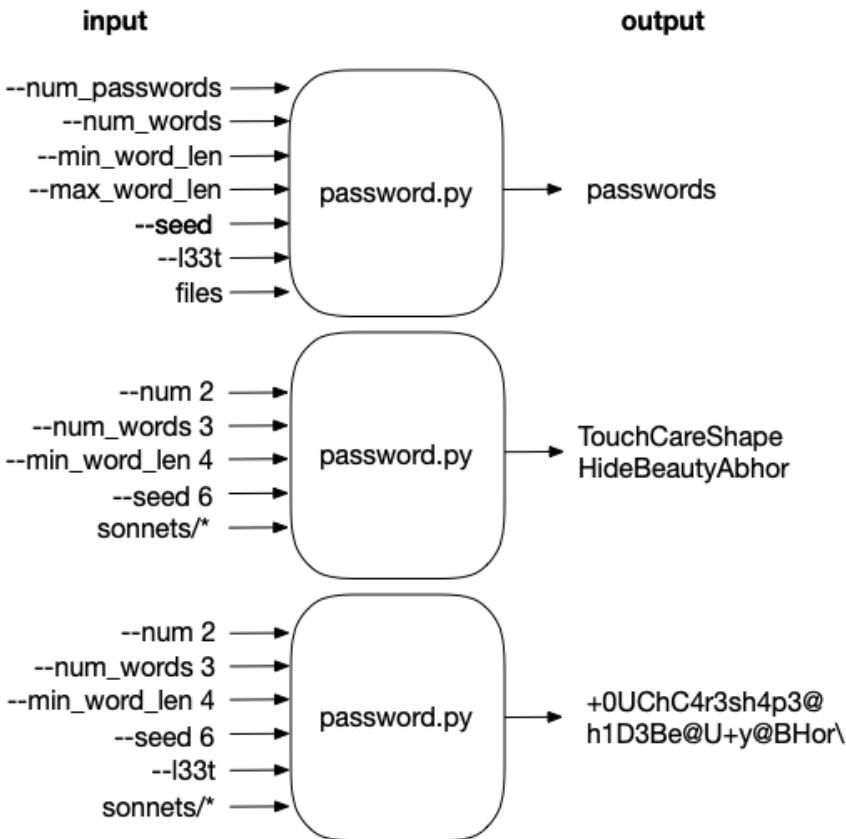
To cap it off, we'll use `random.choice()` to select one character from `string.punctuation` to add to the end:

```
$ ./password.py --num 2 --num_words 3 --seed 9 --min_word_len 10 --max_word_len 20
```

²⁵ See the Wiki page en.wikipedia.org/wiki/Leet or the Cryptii translator cryptii.com/

```
sonnets/* --l33t
p3RsPeC+1Vesucces5i0niN+3lL1Genc3$
D1s+iLL@+ioNconsc1eNc3coun+eN@Nce^
```

Here is the string diagram to summarize the inputs:



20.1.1 Creating a unique list of words

Let's start off by making our program print the name of each input file:

```
def main():
    args = get_args()
    random.seed(args.seed) ①

    for fh in args.file: ②
        print(fh.name) ③
```

- ① Always set `random.seed()` right away as it will globally affect all actions by the `random` module.
- ② Iterate through the file arguments.
- ③ Print the name of the file.

Let's test it with the `words.txt` file:

```
$ ./password.py ../inputs/words.txt
../inputs/words.txt
```

Or with some of the other inputs:

```
$ ./password.py scarlet/*
scarlet/adjs.txt
scarlet/nouns.txt
scarlet/verbs.txt
```

Our first goal is to create a unique list of words we can use for sampling. So far we've used lists to keep ordered collections and dictionaries to create key/value structures. The elements in a list do not have to be unique, so we can't use that. The keys of a dictionary *are* unique, however, so that's a possibility:

```
def main():
    args = get_args()
    random.seed(args.seed)
    words = {}           ①

    for fh in args.file: ②
        for line in fh: ③
            for word in line.lower().split(): ④
                words[word] = 1 ⑤

    print(words)
```

- ① Create an empty dict to hold the words.
- ② Iterate through the files.
- ③ Iterate through the lines of the file.
- ④ Lowercase the line and split it on spaces into words.
- ⑤ Set the key `words[word]` equal to 1 to indicate we saw it. We're only using a dict to get the unique keys. We don't care about the values, so you could use whatever value you like.

If you run this on the US Constitution, you should see a fairly large list of words (some output elided here):

```
$ ./password.py ../inputs/const.txt
{'we': 1, 'the': 1, 'people': 1, 'of': 1, 'united': 1, 'states,' : 1, ...}
```

I can spot one problem in that the word 'states,' has a comma attached to it. If we try in the REPL with the first bit of text from the Constitution, we can see the problem:

```
>>> 'We the People of the United States,'.lower().split()
['we', 'the', 'people', 'of', 'the', 'united', 'states,']
```

How can we get rid of punctuation?

20.1.2 Cleaning the text

We've seen several times that splitting on spaces leaves punctuation, but splitting on non-word characters can break contracted words like "Don't" in two. I'd like to create a function that will `clean()` a word. First I'll imagine the test for it. Note that in this exercise, I'll put all my unit tests into a file called `unit.py` which I can run with `pytest`

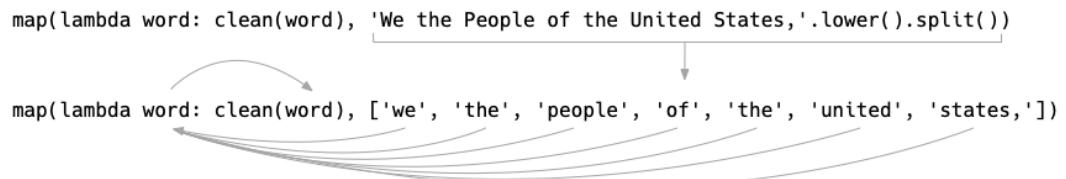
-xv unit.py.

Here is the test for our `clean()` function:

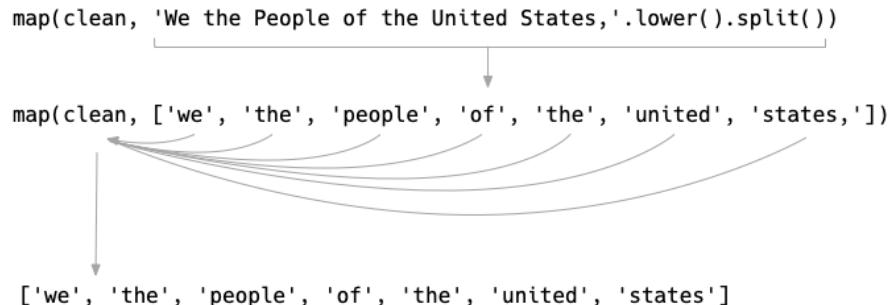
```
def test_clean():
    assert clean('') == ''                      ①
    assert clean("states,") == 'states'          ②
    assert clean("Don't") == 'Dont'              ③
```

- ① It's always good to test your functions on nothing just to make sure it does something sane.
- ② The function should remove punctuation at the end of a string.
- ③ The function should not split a contracted word in two.

I would like to apply this to all the elements returned by splitting each line into words, and `map()` is a fine way to do that. We often use a `lambda` when writing `map()`:



Notice that I do not need to write a `lambda` for the `map()` because the `clean()` function expects a single argument:



See how it integrates with the code:

```
def main():
    args = get_args()
    random.seed(args.seed)
    words = {}

    for fh in args.file:
        for line in fh:
            for word in map(clean, line.lower().split()): ①
                words[word] = 1

    print(words)
```

- ① Use `map()` to apply the `clean()` function to the results of splitting the line on spaces. No `lambda` is required because `clean()` expects a single argument.

If I run that on the US Constitution again, I see that 'states' has been fixed:

```
$ ./password.py ../inputs/const.txt
{'we': 1, 'the': 1, 'people': 1, 'of': 1, 'united': 1, 'states': 1, ...}
```

I'll leave it to you to write the `clean()` function that will satisfy that test.

20.1.3 Using a set

There is a better data structure than a dict to use for our purposes here. It's called a **set**, and you can think of it like a unique list or just the keys of a dict. Here is how we could change our code to use a set to keep track of *unique* words:

```
def main():
    args = get_args()
    random.seed(args.seed)
    words = set() ①

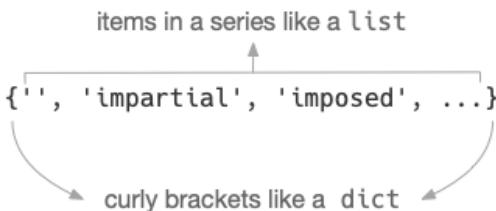
    for fh in args.file:
        for line in fh:
            for word in map(clean, line.lower().split()):
                words.add(word) ②

    print(words)
```

- ① Use the `set()` function to create an empty set.
- ② Use `set.add()` to add a value to a set.

If you run this code now, you will see a slightly different output where Python shows you a data structure in curly brackets (`{}`) that makes you think of a dict but you'll notice that the contents look more like a list:

```
$ ./password.py ../inputs/const.txt
{'', 'impartial', 'imposed', 'jared', 'levying', ...}
```



We're using sets here only for the fact that they so easily allow us to keep a unique list of words, but sets are much more powerful than this. For instance, you can find the shared values between two lists by using `set.intersection()`:

```
>>> nums1 = set(range(1, 10))
>>> nums2 = set(range(5, 15))
>>> nums1.intersection(nums2)
{5, 6, 7, 8, 9}
```

You can read `help(set)` in the REPL or the documentation online to learn about all the amazing things you can do with sets.

20.1.4 Filtering the words

If we look again at the output we have, we'll see that the empty string is the first element:

```
$ ./password.py ../inputs/const.txt
{'', 'impartial', 'imposed', 'jared', 'levying', ...}
```

We need a way to filter out unwanted values like strings that are too short. In the "Rhymers" exercise, we looked at the `filter()` function which is a higher-order function that takes two arguments:

1. A function that accepts one element and returns `True` if the element should be kept or `False` if the element should be excluded.
2. Some "iterable" (like a `list` or `map()`) that produces a sequence of elements to be filtered.

In our case, we want to accept only words that have a length greater or equal to the `--min_word_len` argument and less than or equal to the `--max_word_len`. In the REPL, I can use a `lambda` to create an anonymous function that accepts a word and makes these comparisons. The result of that comparison is either `True` or `False`. Only words with a length between 3 and 6 are allowed, so this has the effect of removing short, uninteresting words. Remember that `filter()` is lazy, so I have to coerce it using the `list` function in the REPL to see the output:

```
>>> shorter = ['', 'a', 'an', 'the', 'this']
>>> min_word_len = 3
>>> max_word_len = 6
>>> list(filter(lambda word: min_word_len <= len(word) <= max_word_len, shorter))
['the', 'this']
```

It will also remove longer words:

```
>>> longer = ['that', 'other', 'egalitarian', 'disequilibrium']
>>> list(filter(lambda word: min_word_len <= len(word) <= max_word_len, longer))
['that', 'other']
```

One way we could incorporate the `filter()` is to create a `word_len()` function that encapsulates the above `lambda`. Note that I defined it inside the `main()` in order to create a *closure* because I want to include the values of `args.min_word_len` and `args.max_word_len`:

```
def main():
    args = get_args()
    random.seed(args.seed)
    words = set()

    def word_len(word): ①
        return args.min_word_len <= len(word) <= args.max_word_len

    for fh in args.file:
        for line in fh:
            for word in filter(word_len, map(clean, line.lower().split())): ②
                words.add(word)
```

```
print(words)
```

- ① This function will return True if the length of the given word is in the allowed range.
- ② We can use `word_len` (without the parentheses!) as the function argument to `filter()`.

We can again try our program to see what it produces:

```
$ ./password.py ../inputs/const.txt
{'measures', 'richard', 'deprived', 'equal', ...}
```

Try it on multiple inputs such as all the nouns, adjectives, and verbs from *The Scarlet Letter*:

```
$ ./password.py scarlet/*
{'walk', 'lose', 'could', 'law', ...}
```



20.1.5 Titlecasing the words

We used the `line.lower()` function to lowercase all the input, but the passwords we generate will need each word to be in "Title Case" where the first letter is uppercase and the rest of the word is lower. Can you figure out how to change the program to produce this output?

```
$ ./password.py scarlet/*
{'Dark', 'Sinful', 'Life', 'Native', ...}
```

Now we have a way to process any number of files to produce a unique list of title-cased words that have non-word characters removed and have been filtered to remove the ones that are too short or long. That's quite a lot of power packed into a few lines of code!

20.1.6 Sampling and making a password

We're going to use the `random.sample()` function to randomly choose some `--num` number of words from our set to create an unbreakable yet memorable password. We've talked before about the importance of using a random seed to test that our "random" selections are reproducible. It's also quite important that the items from which we sample always be ordered in the same way so that the same selections are made. If we use the `sorted()` function on a set, we get back a sorted list which is perfect for using with `random.sample()`. I can add this line to the code from before:

```
words = sorted(words)
print(random.sample(words, args.num_words))
```

Now when I run it with *The Scarlet Letter* input, I will get a list of words that might make an interesting password:

```
$ ./password.py scarlet/*
['Lose', 'Figure', 'Heart', 'Bad']
```

The result of `random.sample()` is a list that you can join on the empty string in order to make a new password:

```
>>> ''.join(random.sample(words, num_words))
'TokenBeholdMarketBegin'
```

You will need to create `args.num` of passwords. How will you do that?

20.2 I33t-ify

The last piece of our program is to create a `l33t()` function that will obfuscate the password. The first step is to convert it with the same algorithm we wrote for `ransom.py`. I'm going to create a `ransom()` function for this, and here is the test that is in `unit.py`. I'll leave it to you to create the function that satisfies this test²⁶:

```
def test_ransom():
    state = random.getstate() ①
    random.seed(1) ②
    assert (ransom('Money') == 'moNeY')
    assert (ransom('Dollars') == 'D0ll1aRs')
    random.setstate(state) ③
```

- ① Save the current global state.
- ② Set the `random.seed()` to a known value for the test.
- ③ Restore the state.

Next I will substitute some of the characters according to the following table. I would recommend you revisit "Jump The Five" to see how you did that:

a => @
A => 4
O => 0
t => +
E => 3
I => 1
S => 5

I wrote a `l33t()` function that combines the `ransom()` with the substitution above and finally adds a punctuation character by appending `random.choice(string.punctuation)`. Here is the `test_l33t()` function you can use to write your function. It works almost identically to the above test, so I shall eschew commentary:

```
def test_l33t():
    state = random.getstate()
    random.seed(1)
    assert l33t('Money') == 'moNeY{'
    assert l33t('Dollars') == 'D0ll14r$`'
```

²⁶ You can run `pytest -xv unit.py` to run the unit tests. The program will import the various functions from your `password.py` file to test. Open `unit.py` and inspect it to understand how this happens!

```
random.setstate(state)
```

20.2.1 Putting it all together

Without giving away the ending, I'd like to say that you need to be *really careful* about the order of operations that include the `random` module. My first implementation would print different passwords given the same seed when I used the `--l33t` flag. Here was the output for plain passwords:

```
$ ./password.py -s 1 -w 2 sonnets/*
EagerCarcanet
LilyDial
WantTempest
```

I would have expected the *exact same passwords* only encoded. Here is what my program produced instead:

```
$ ./password.py -s 1 -w 2 sonnets/* --l33t
3@G3RC@rC@N3+{
m4dness5iNcoN5+4n+
MouTh45s15T4nCe^
```

The first password looks OK, but what are those other two? I modified my code to print both the original password and the l33ted one:

```
$ ./password.py -s 1 -w 2 sonnets/* --l33t
3@G3RC@rC@N3+{ (EagerCarcanet)
m4dness5iNcoN5+4n+| (MadnessInconstant)
MouTh45s15T4nCe^ (MouthAssistance)
```

The `random` module uses a global state to make each of its "random" choices. In my first implementation, I was modifying this state after choosing the first password by immediately modifying the new password with the `l33t()` function. Because the `l33t()` function also uses `random` functions, the state was altered for the next password. The solution was to first generate *all* the passwords and then to alter them using the `l33t()` function, if necessary.

Those are all the pieces you should need to write your program. You have the unit tests to help you verify the functions, and you have the integration tests to ensure your program works as a whole.

20.3 Solution

```
#!/usr/bin/env python3
"""Password maker, https://xkcd.com/936/"""

import argparse
import random
import re
import string

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


```

```

parser = argparse.ArgumentParser(
    description='Password maker',
    formatter_class=argparse.ArgumentDefaultsHelpFormatter)

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

parser.add_argument('-n',
                    '--num',
                    metavar='num_passwords',
                    type=int,
                    default=3,
                    help='Number of passwords to generate')

parser.add_argument('-w',
                    '--num_words',
                    metavar='num_words',
                    type=int,
                    default=4,
                    help='Number of words to use for password')

parser.add_argument('-m',
                    '--min_word_len',
                    metavar='mininum',
                    type=int,
                    default=3,
                    help='Minimum word length')

parser.add_argument('-x',
                    '--max_word_len',
                    metavar='maximumm',
                    type=int,
                    default=6,
                    help='Maximum word length')

parser.add_argument('-s',
                    '--seed',
                    metavar='seed',
                    type=int,
                    help='Random seed')

parser.add_argument('-l',
                    '--l33t',
                    action='store_true',
                    help='Obfuscate letters')

return parser.parse_args()

# -----
def main():
    args = get_args()
    random.seed(args.seed)
    words = set()                                ①

    def word_len(word):
        return args.min_word_len <= len(word) <= args.max_word_len ③
    ②

```

```

for fh in args.file:
    for line in fh:
        for word in filter(word_len, map(clean, line.lower().split())): ⑥
            words.add(word.title()) ⑦

words = sorted(words) ⑧
passwords = [
    ''.join(random.sample(words, args.num_words)) for _ in range(args.num)
] ⑨

if args.l33t:
    passwords = map(l33t, passwords) ⑩
    ⑪

print('\n'.join(passwords)) ⑫

# -----
def clean(word): ⑬
    """Remove non-word characters from word"""

    return re.sub('[^a-zA-Z]', '', word) ⑭

# -----
def l33t(text): ⑮
    """l33t"""

    text = ransom(text)
    xform = str.maketrans({ ⑯
        'a': '@', 'A': '4', '0': '0', 't': '+', 'E': '3', 'I': '1', 'S': '5'
    })
    return text.translate(xform) + random.choice(string.punctuation) ⑰

# -----
def ransom(text): ⑲
    """Randomly choose an upper or lowercase letter to return"""

    return ''.join( ⑳
        map(lambda c: c.upper() if random.choice([0, 1]) else c.lower(), text))

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

```

- ① Set the `random.seed()` to the given value or the default `None` which is the same as not setting the seed.
- ② Create an empty set to hold all the unique of words we'll extract from the texts.
- ③ Create a `word_len()` function for `filter()` that returns `True` if the word's length is in the allowed range and `False` otherwise.
- ④ Iterate through each open file handle.
- ⑤ Iterate through each line of text in the file handle.
- ⑥ Iterate through each word generated by splitting the lowercased line on spaces, removing non-word characters with the `clean()` function, and filtering for words of an acceptable length.
- ⑦ Titlecase the word before adding it to the set.
- ⑧ Use the `sorted()` function to order words into a new list.

- ⑨ Use a list comprehension with a range to create the correct number of passwords. Since I don't need the actual value from range, I can use the _ to ignore the value.
- ⑩ See if the args.133t flag is True.
- ⑪ Use map() to run all the passwords through the 133t() function to produce a new list of passwords. It's safe to call the 133t() function here. If we had used the function in the list comprehension, it would have altered the global state of the random module thereby altering the following passwords.
- ⑫ Print the passwords joined on newlines.
- ⑬ Define a function to clean() a word.
- ⑭ Use a regular expression to substitute the empty string for anything that is not an English alphabet character.
- ⑮ Define a function to 133t() a word.
- ⑯ First use the ransom() function to randomly capitalize letters.
- ⑰ Make a translation table/dict for character substitutions.
- ⑱ Use the str.translate() function to perform the substitutions, append a random piece of punctuation.
- ⑲ Define a function for the ransom() algorithm we wrote in chapter 12.
- ⑳ Return a new string created by randomly upper- or lowercasing each letter in a word.

20.4 Discussion

I hope you found this program challenging and interesting. Let's break it down a bit. There wasn't anything new in get_args(), so let's start with the auxiliary functions:

20.4.1 Cleaning the text

I chose to use a regular expression to remove any characters that are outside the set of lowercase and uppercase English characters:

```
def clean(word):
    """Remove non-word characters from word"""
    return re.sub('[^a-zA-Z]', '', word) ①
```

- ① The re.sub() function will substitute any text matching the pattern (the first argument) found in the given text (the third argument) with the value given by the second argument.

Recall from the "Gematria" exercise that we can write the character class [a-zA-Z] to define the characters in the ASCII table bounded by those two ranges. We can then *negate* or complement that class by placing a caret ^ as the *first character* inside that class, so [^a-zA-Z] can be read as "any character not matching a to z or A to Z."

It's perhaps easier to see it in action in the REPL. In this example, only the letter "AbCd" will be left from the text "A1b*C!d4":

```
>>> import re
>>> re.sub('[^a-zA-Z]', '', 'A1b*C!d4')
'AbCd'
```

If the only goal were to match ASCII letters, it's possible to solve it by looking for membership in `string.ascii_letters`:

```
>>> import string
>>> text = 'A1b*C!d4'
>>> [c for c in text if c in string.ascii_letters]
['A', 'b', 'C', 'd']
```

It honestly seems like more effort to me. Besides, if the function needed to be changed to allow, say, numbers and a few specific pieces of punctuation, then the regular expression version becomes significantly easier to write and maintain.

20.4.2 A king's ransom

The `ransom()` function was taken straight from the `ransom.py` program, so there isn't too much to say about it except, hey, look how far we've come! What was an entire idea for a chapter is now a single line in a much longer and more complicated program:

```
def ransom(text):
    """Randomly choose an upper or lowercase letter to return"""
    return ''.join(①
        map(lambda c: c.upper() if random.choice([0, 1]) else c.lower(), text)) ②
```

- ② Use `map()` iterate through each character in the text and select either the upper- or lowercase version of the character based on a "coin" toss using `random.choice()` to select between a "truthy" value (1) or a "falsey" value (0).
- ① Join the resulting list from the `map()` on the empty string to create a new str.

20.4.3 How to 133t()

The `133t()` function builds on the `ransom()` and then adds a text substitution that is straight out of "Jump The Five." I like the `str.translate()` version of that program, so I used it again here:

```
def l33t(text):
    """l33t"""
    text = ransom(text) ①
    xform = str.maketrans({②
        'a': '@', 'A': '4', '0': '0', 't': '+', 'E': '3', 'I': '1', 'S': '5'
    })
    return text.translate(xform) + random.choice(string.punctuation) ③
```

- ① First randomly capitalize the given text.
- ② Make a translation table from the given dict which describes how to modify one character to another. Any characters not listed in the keys of this dict will be ignored.
- ③ Use the `str.translate()` method to make all the character substitutions. Use `random.choice()` to select one additional character from `string.punctuation` to append to the end.

20.4.4 Processing the files

Now to apply these to the processing of the text. To use these, we need to create a unique set of all the words in our input files. I wrote this bit of code both with an eye on performance and for style:

```
words = set()
```

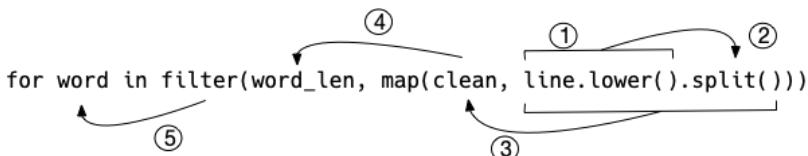
```

for fh in args.file: ①
    for line in fh: ②
        for word in filter(word_len, map(clean, line.lower().split())): ③
            words.add(word.title()) ④

```

- ① Iterate through each open file handle.
- ② Read the file handle line-by-line with a for loop, *not* with a method like `fh.read()` which will read the entire contents of the file at once.
- ③ Reading this code actually requires starting at the end where we split the `line.lower()` on spaces. Each word from `str.split()` goes into `clean()` which then must pass through the `filter()` function.
- ④ Titlecase the word before adding it to the set.

Here's a diagram of that for line:



1. `line.lower()` will return a lowercase version of `line`.
2. The `str.split()` method will break the text on whitespace to return words.
3. Each word is fed into the `clean()` function to remove any character that is not in the English alphabet.
4. The cleaned words are filtered by the `word_len()` function.
5. The resulting word has been transformed, cleaned, and filtered.

If you don't like the `map()` and `filter()` functions, then you might rewrite the code like so:

```

words = set()
for fh in args.file: ①
    for line in fh: ②
        for word in line.lower().split(): ③
            word = map(clean) ④
            if args.min_word_len <= len(word) <= args.max_word_len: ⑤
                words.add(word.title()) ⑥

```

- ① Iterate through each open file handle.
- ② Iterate through each line of the file handle.
- ③ Iterate through each "word" from splitting the lowercased line on spaces.
- ④ Clean the word up.
- ⑤ Check if the word is an acceptable length.
- ⑥ Add the titlecased word to the set.

However you choose to process the files, at this point you should have a complete set of all the unique, titlecased words from the input files.

20.4.5 Sampling and creating the passwords

As noted above, it's vital to sort the words for our tests so that we can verify that we are making consistent choices. If you only wanted random choices and didn't care about testing, you would not need to worry about sorting — but then you'd also be a morally deficient person for not testing, so perish the thought! I chose to use the `sorted()` function as there is no other way to sort a set:

```
words = sorted(words) ①
```

- ① There is no `set.sort()` function. Sets are ordered internally by Python. Calling `sorted()` on a set will create a new, sorted list.

We need to create some given number of passwords, and I thought it might be easiest to use a `for` loop with a `range()`. In my code, I used `for _ in range(...)` because I don't need to know the value each time through the loop. The underscore (`_`) is a way to indicate that you are ignoring the value. It's fine to say `for i in range(...)` if you want, but some linters might complain if they see that your code declares the variable `i` but never uses it. That could legitimately be a bug, so it's best to use the `_` to show that you mean to ignore this value.

Here is the first way I wrote the code that led to the bug I mentioned in the discussion where different passwords would be chosen even when I used the same random seed. *Can you spot the bug?*

```
for _ in range(args.num): ①
    password = ''.join(random.sample(words, args.num_words)) ②
    print(l33t(password) if args.l33t else password) ③
```

- ① Iterate through the `args.num` of passwords to create.
- ② Each password will be based on a random sampling from our `words`, and we will choose the value given in `args.num_words`. The `random.sample()` function returns a list of words that we `str.join()` on the empty string to create a new string.
- ③ If the `args.l33t` flag is True, then we'll print the `l33t` version of the password; otherwise, we'll print the password as-is. **This is the bug!** Calling `l33t()` here modifies the global state used by the `random` module, so the next time we call `random.sample()` we get a *different sample*.

The solution is to separate the concerns of *generating* the passwords and possibly modifying them:

```
passwords = [
    ''.join(random.sample(words, args.num_words)) for _ in range(args.num)
]

if args.l33t:
    passwords = map(l33t, passwords)

print('\n'.join(passwords)) ③
```

- ① Use a list comprehension iterate through `range(args.num)` to generate the correct number of passwords.
- ② If the `args.leet` flag is True, then use the `l33t()` function to modify the passwords.
- ③ Print the passwords joined on newlines.

20.5 Review

This exercise kind of has it all. Validating user input, reading files, using a new data structure in the `set`, higher-order functions with `map()` and `filter()`, random values, and lots of functions and tests! I hope you enjoyed programming it, and maybe you'll even use the program to generate your new passwords. Be sure to share those passwords with your author, especially the ones to your bank account and favorite shopping sites!



20.6 Going Further

- The substitution part of the `l33t()` function changes every available character which perhaps makes the password too difficult to remember. It would be better to modify only maybe 10% of the password similar to how we changed the input strings in the "Telephone" exercise.
- Create programs that combine other skills you've learned. Like maybe a lyrics generator that randomly selects lines from a files of songs by your favorite bands, then encodes the text with the "Kentucky Friar," then changes all the vowels to one vowel with "Apples and Bananas," and then SHOUTS IT OUT with "The Howler"?

Congratulations, you are now 733+ HAX0R!

21

Tic-Tac-Toe: Exploring state

One of my favorite movies is the 1983 release "War Games" starring Matthew Broderick whose character "David" plays a young hacker who enjoys cracking into computers systems ranging from his school's grade book to a Pentagon server that has the potential to launch intercontinental ballistic missiles. Central to the plot is the game of "Tic-Tac-Toe," a game so simple that it usually ends in a draw between the two players. In the movie, David engages Joshua, an artificial intelligence (AI) agent, who is capable of playing lots of nice games like chess. David would rather play the game "Global Thermonuclear War" with Joshua.



Eventually David realizes that Joshua is using the simulation of a war game to trick the US military into initiating a nuclear first strike against the Soviet Union. Understanding the Mutually Assured Destruction (MAD) doctrine, David asks Joshua to play himself at Tic-Tac-Toe so that he can explore the futility of games that can never result in victory. After hundreds or thousands of rounds all ending in draws, Joshua concludes that "the only winning strategy is not play," at which point Joshua stops trying to destroy the Earth and suggests instead that they could play "a nice game of chess."

I assume you already know the game of Tic-Tac-Toe, but we'll review briefly in case your childhood missed countless games of this with your friends. The game starts out with a 3-by-3 square grid. There are two players who take turns marking first X and then O into the cells. A player wins by placing the same mark in any three squares in a straight line horizontally, vertically, or diagonally. This is usually impossible as each player will generally use their moves to block a potential win by their opponent.

We will program a simulation of the game that will explore the idea of program "state" which is a way to think of how the pieces of a program change over time. For instance, we start off with a blank board, and the first player to go is X. After each round, some cell on the board is taken by a player, and play alternates between the X and O. We will need to keep track of these ideas and more so that, at any moment, we always know the "state" of the game.

If you recall, the hidden state of the `random` module proved to be a problem in the "Password" chapter where an early solution we explored produced inconsistent results depending on the order of operations that used the module. In this exercise, we're going to think about ways to make the "state" of our game and — any changes to it — explicit.

In our version of Tic-Tac-Toe, we'll write a program that plays just one round of the game. Your program will be given a string that represents a Tic-Tac-Toe board at any time during a game. The default is the empty board at the beginning of the game, before either player has a move. The program may also be given one move to add to that board. It will print a picture of the board and report if there is a winner.

For our program, need to track at least two ideas in our state:

1. The board, or which player has marked which squares of the grid.
2. The winner, if there is one.

In the next chapter, we'll write an interactive version of the game where we will need to track and update several more items in our state through as many rounds as needed to finish a game.

In this exercise, you will:

- Consider how to use elements like strings and lists to represent aspects of a program's state.
- Enforce the rules of a game as code such as preventing a player from taking a cell that has already been taken.
- Use a regular expression to validate the initial board.
- Use `and` and `or` to reduce combinations of boolean values to a single value.
- Use lists of lists to find a winning board.
- Use the `enumerate()` function to iterate a `list` with the index and value.

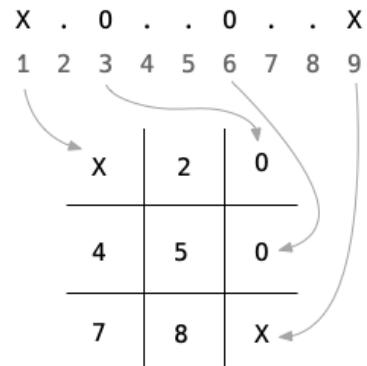
21.1 Writing `tictactoe.py`

The initial state of the board will come from a `-b` or `--board` option that describes

which cells are occupied by which players. Since there are nine cells, we'll use a string that is nine characters long composed only of the characters X and 0 or the dot (.) to indicate the cell is open. The default board will be 9 dots, and the grid should number the cells from 1 to 9 as they are all empty. As there is no winner, the result will be "No winner":

```
$ ./tictactoe.py
-----
| 1 | 2 | 3 |
-----
| 4 | 5 | 6 |
-----
| 7 | 8 | 9 |
-----
No winner.
```

The --board will describe which cells to mark for which player where each position of the string describes each cell ascending from 1 to 9. In the string X.0..0..X, the positions 1 and 9 are occupied by "X" and positions 3 and 6 by "0".



Here is how the grid would be rendered by the program:

```
$ ./tictactoe.py -b X.0..0..X
-----
| X | 2 | 0 |
-----
| 4 | 5 | 0 |
-----
| 7 | 8 | X |
-----
No winner.
```

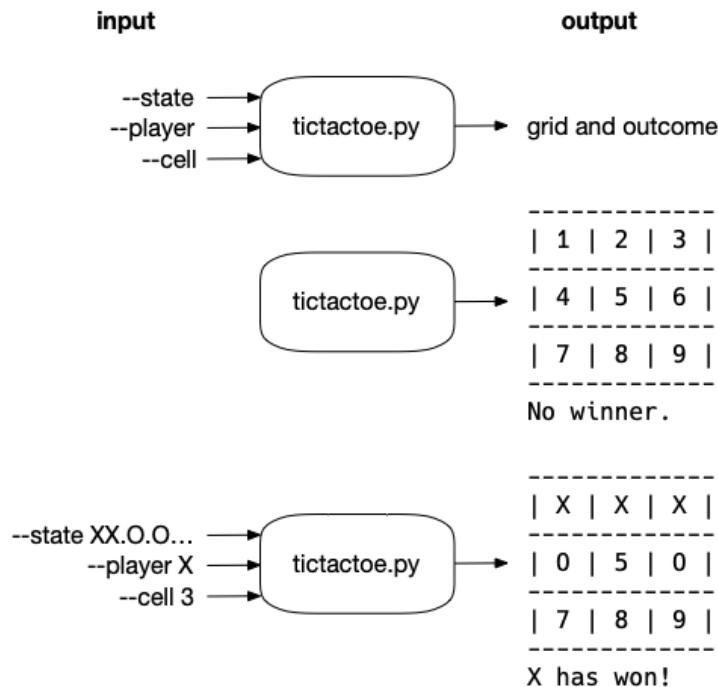
We can additionally modify the given --board by passing a -c or --cell option of 1-9 and a -p or --player of "X" or "O". For instance, we can mark the first cell as "X" like so:

```
$ ./tictactoe.py --cell 1 --player X
-----
| X | 2 | 3 |
-----
| 4 | 5 | 6 |
-----
| 7 | 8 | 9 |
-----
No winner.
```

The winner, if any, should be declared with gusto:

```
$ ./tictactoe.py -b XXX.....  
-----  
| X | X | X |  
-----  
| 4 | 5 | 6 |  
-----  
| 7 | 8 | 9 |  
-----  
X has won!
```

As usual, we'll use a test suite to ensure that our program works properly. Here is our string diagram:



21.1.1 Validating user input

There's a fair bit of input validation that needs to happen. The `--board` needs to ensure that any argument is exactly 9 characters and composed only of X, 0, and .:

```
$ ./tictactoe.py --board XXX000..  
usage: tictactoe.py [-h] [-b board] [-p player] [-c cell]  
tictactoe.py: error: --board "XXX000.." must be 9 characters of ., X, 0
```

Likewise, the `--player` can only be X or 0:

```
$ ./tictactoe.py --player A --cell 1  
usage: tictactoe.py [-h] [-b board] [-p player] [-c cell]  
tictactoe.py: error: argument -p/--player: \  
invalid choice: 'A' (choose from 'X', '0')
```

And the --cell can only be an integer value from 1 to 9:

```
$ ./tictactoe.py --player X --cell 10
usage: tictactoe.py [-h] [-b board] [-p player] [-c cell]
tictactoe.py: error: argument -c/--cell: \
invalid choice: 10 (choose from 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Both --player and --cell must be present together or neither can be present:

```
$ ./tictactoe.py --player X
usage: tictactoe.py [-h] [-b board] [-p player] [-c cell]
tictactoe.py: error: Must provide both --player and --cell
```

Lastly, if the --cell specified is already occupied by an X or an O, the program should error out:

```
$ ./tictactoe.py --player X --cell 1 --board X..O.....
usage: tictactoe.py [-h] [-b board] [-p player] [-c cell]
tictactoe.py: error: --cell "1" already taken
```

I would recommend you put all this error checking into `get_args()` so that you can use `parser.error()` to throw the errors and halt the program.

21.1.2 Altering the board

The initial board, once validated, describes which cells are occupied by which player. This board can be altered by the addition of the --player and --cell arguments. It may seem silly to not just pass in the already altered --board, but this is necessary practice for writing the interactive version.

If you represent board as a `str` value like 'XX.O.O..X' and you need to change, for instance, cell 3 to an X, how will you do that? For one thing, the "cell" 3 is not found at *index* 3 in the given board — the index is *one less* than the cell number. The other issue is that a `str` is immutable. Just as in "Telephone," you'll need to figure out a way to modify one character in the board value.

21.1.3 Printing the board

Once you have a board, you need to format it with ASCII characters to create a grid. I recommend you make a function called `format_board()` that takes the board as an argument and returns a `str` that uses dashes - and vertical pipes | to create a table. I have provided a `unit.py` file that contains, for instance, the following test for the default, unoccupied grid:

```
def test_board_no_board():
    """makes default board"""

    board = """ ①
-----
| 1 | 2 | 3 |
-----
| 4 | 5 | 6 |
-----
| 7 | 8 | 9 |
-----
```

```
""" .strip()

    assert format_board('.' * 9) == board ②
```

- ① Use the triple quotes as the string has embedded newlines. The final `str.strip()` call will remove the trailing newline which I used to format the code.
- ② If you multiply a string by an integer value, Python will create a new `str` by repeating the given string that number of times. Here we create a string of nine dots as the input to `format_board()`. We expect the return should be an empty board as formatted above.

Then try formatting a board with some other combination. Here's another test I wrote that you may like to use, but feel free to write your own:

```
def test_board_with_board():
    """makes board"""

    board = """
-----
| 1 | 2 | 3 |
-----
| 0 | X | X |
-----
| 7 | 8 | 9 |
-----
"""

    assert format_board('...OXX...') == board ①
```

- ① The given board should have the first and third rows open and the second row with "OXX".

It would be impractical to test every possible combination for the board. When you're writing tests, you often have to rely on spot-checking your code. Here I am checking the empty board and a non-empty board. Presumably if the function can handle these two arguments, it can handle any others.

21.1.4 Determining a winner

Once you have validated the input and printed the board, the last task is to declare a winner if there is one. I chose to write a function called `find_winner()` that returns either `X` or `O` if one of those is the winner or returns `None` if there is no winner. To test this, I wrote out every possible winning board to test my function with values for both players. You are welcome to use this test:

```
def test_winning():
    """test winning boards"""

    wins = [('PPP.....'), ('...PPP...'), ('.....PPP'), ('P..P..P..'), ①
            ('.P..P..P.'), ('..P..P..P'), ('P...P...P'), ('..P..P..P.')]
    for player in 'XO':
        other_player = 'O' if player == 'X' else 'X' ②
        for board in wins:
            board = board.replace('P', player) ③ ④ ⑤
```

```

        dots = [i for i in range(len(board)) if board[i] == '.'] ⑥
        mut = random.sample(dots, k=2) ⑦
        test_board = ''.join([
            other_player if i in mut else board[i] ⑧
            for i in range(len(board))
        ])
        assert find_winner(test_board) == player ⑨
    
```

- ① This is a list of the board indexes that, if occupied by the same player, would win.
- ② Check for both players X and O.
- ③ The test will randomly select a couple of the empty cells to fill with the other player, so figure out which one that will be.
- ④ Iterate through each of the lists of winning combinations.
- ⑤ Change all the P (for "player") values in the given board to the given player that we're checking.
- ⑥ Find the indexes of the open cells indicated by a dot.
- ⑦ Randomly sample two open cells. We will mutate these, so I call them `mut`.
- ⑧ Alter the board to change the two selected `mut` cells to the `other_player`.
- ⑨ Assert that `find_winner()` will determine that this board wins for the given player.

I also wanted to be sure I would not falsely claim that a losing board is winning, so I also wrote the following test to ensure that `None` is returned when there is no winner:

```

def test_losing():
    """test losing boards"""

    losing_board = list('XXOO.....') ①

    for _ in range(10):
        random.shuffle(losing_board) ②
        assert find_winner(''.join(losing_board)) is None ③
    
```

- ① No matter how this board is arranged, it cannot win as there are only two marks for each player.
- ② Run 10 tests.
- ③ Shuffle the losing board into another configuration.
- ④ Assert that, no matter how the board is arranged, will still find no winner.

If you choose the same function names as I did, then you can run `pytest -xv unit.py` to run the unit tests I wrote. If you wish to write different functions, you can create your own unit tests either inside your `tictactoe.py` or in another unit file.

After printing the board, be sure to print "{winner} has won!" or "No winner" depending on the outcome. All righty, you have your orders, so get marching!

21.2 Solution

```

#!/usr/bin/env python3
"""Tic-Tac-Toe"""

import argparse
import re

```

```

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

    parser = argparse.ArgumentParser(
        description='Tic-Tac-Toe',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('-b',                                ①
                        '--board',
                        help='The state of the board',
                        metavar='board',
                        type=str,
                        default='.' * 9)

    parser.add_argument('-p',                                ②
                        '--player',
                        help='Player',
                        choices='XO',
                        metavar='player',
                        type=str,
                        default=None)

    parser.add_argument('-c',                                ③
                        '--cell',
                        help='Cell 1-9',
                        metavar='cell',
                        type=int,
                        choices=range(1, 10),
                        default=None)

    args = parser.parse_args()

    if any([args.player, args.cell]) and not all([args.player, args.cell]): ④
        parser.error('Must provide both --player and --cell')

    if not re.search('^[.XO]{9}$', args.board): ⑤
        parser.error(f"--board '{args.board}' must be 9 characters of ., X, O")

    if args.player and args.cell and args.board[args.cell - 1] in 'XO': ⑥
        parser.error(f"--cell '{args.cell}' already taken")

    return args


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

    args = get_args()
    board = list(args.board)                                ⑦

    if args.player and args.cell:                         ⑧
        board[args.cell - 1] = args.player ⑨

    print(format_board(board))                            ⑩
    winner = find_winner(board)                          ⑪
    print(f'{winner} has won!' if winner else 'No winner.') ⑫

# -----

```

```

def format_board(board):          ⑯
    """Format the board"""

    cells = [str(i) if c == '.' else c for i, c in enumerate(board, 1)] ⑰
    bar = '-----'
    cells_tmpl = '| {} | {} | {} |'
    return '\n'.join([
        cells_tmpl.format(*cells[:3]), bar,
        cells_tmpl.format(*cells[3:6]), bar,
        cells_tmpl.format(*cells[6:]), bar
    ])

# -----
def find_winner(board):          ⑯
    """Return the winner"""

    winning = [[0, 1, 2], [3, 4, 5], [6, 7, 8], [0, 3, 6], [1, 4, 7],
               [2, 5, 8], [0, 4, 8], [2, 4, 6]] ⑰

    for player in ['X', 'O']:  ⑱
        for i, j, k in winning: ⑲
            combo = [board[i], board[j], board[k]] ⑳
            if combo == [player, player, player]: ⑲
                return player ⑲
    return None ⑲

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

```

- ⑯ The --board will default to nine dots. If you use the multiplication operator * with a str value and an int (in any order), the result is the str value repeated int times. So '.' * 9 will produce '.....'.
- ⑰ The --player must be either X or O which can be validated using choices.
- ⑱ The --cell must be an integer between 1 and 9 which can be validated with type=int and choices=range(1, 10) remembering that the upper bound (10) is not included.
- ⑲ The combination of any() and all() is a way to test that both arguments are present or neither is.
- ⑳ Use a regular expression to check that the --board is comprised of exactly nine valid characters.
- ⑲ If both --player and --cell are present and valid, then verify that the cell in the board is not currently occupied by an X or an O.
- ⑲ Since we may need to alter the board, it's easiest to convert it to a list.
- ⑲ We modify board if cell and player are "truthy." We validated the arguments in get_args(), so it's safe to use them here. That is, we won't accidentally assign an index value that is out of range because we have taken the time to check that the cell value is acceptable.
- ⑲ Since we use 1-based counting for the cells, we need to subtract 1 from the cell to change the correct index in board.
- ⑲ Now that we have possibly modified board, we can print the board.
- ⑲ Look for a winner in the board.
- ⑲ Print the outcome of the game. The find_winner() function returns either X or O if one of the players has won or None to no indicate no winner.
- ⑯ Define a function to format the board. The function does not print() the board because that would make it hard to test. The function returns a new str value that can be printed or tested.

- (14) Iterate through the cells in the board and decide whether to print the cell number if the cell is unoccupied or the player occupying the cell.
- (15) The return from the function is a new str created by joining all the lines of the grid on newlines.
- (16) Define a function that returns a winner or the value None if there is no winner. Again, the function does not print() the winner but only returns an answer that can be printed or tested.
- (17) There are 8 winning boards which are defined as 8 lists of the cells that need to be occupied by the same player. Note that I chose here to represent the actual zero-offset index values and not the 1-based values we expect from the user.
- (18) Iterate through both players, X and O.
- (19) Iterate through each winning combination of cells, unpacking them into the variables i, j, and k.
- (20) Create a combo that is the value of the board for each of i, j, and k.
- (21) Check if the combo is the same player in every position.
- (22) If that is True, return the player. If this is never True, we fall off the end of the function, and the value None is returned by default.

21.2.1 Validating the arguments and mutating the board

Most of the validation can be handled by using argparse effectively. Both the --player and --cell options can be handled by the choices option. It's worth taking time to appreciate the use of any() and all() in this code:

```
if any([args.player, args.cell]) and not all([args.player, args.cell]):
    parser.error('Must provide both --player and --cell')
```

We can play with these functions in the REPL. The any() function is the same as using or in between boolean values:

```
>>> True or False or True
True
```

If *any* of the items in a given list is "truthy," then the whole expression will evaluate to True:

```
>>> any([True, False, True])
True
```

If *cell* is a non-zero value and *player* is not the empty string, then they are both "truthy":

```
>>> cell = 1
>>> player = 'X'
>>> any([cell, player])
True
```

The all() function is the same as using and in between all the elements in a list, so *all* of the elements need to be "truthy" in order for the whole expression to be True:

```
>>> cell and player
'X'
```

Why does that return `X`? It returns the last "truthy" value which is the `player` value, so if we reverse the arguments, we'll get the `cell` value:

```
>>> player and cell
1
```

If we use `all()`, it evaluates the truthiness of anding the values, which will be `True`:

```
>>> all([cell, player])
True
```

We are trying to figure out if the user has provided only *one* of the arguments for `-player` and `--cell`, because we need both or we want neither. So pretend `cell` is `None` (the default) but `player` is `X`. It's true that `any()` of those values is "truthy":

```
>>> cell = None
>>> player = 'X'
>>> any([cell, player])
True
```

But it's not true that they *both* are:

```
>>> all([cell, player])
False
```

So when we and those two expressions, they return `False`:

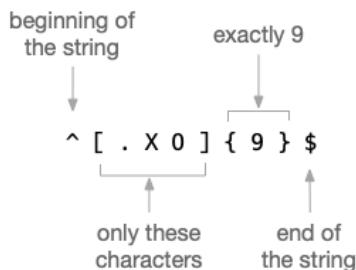
```
>>> any([cell, player]) and all([cell, player])
False
```

Because that is the same as saying:

```
>>> True and False
False
```

The default for `--board` is provided, and we can use a regular expression to verify that it's correct. Our regular expression creates a character class composed of the the dot, "X," and "O" by using `[.XO]`. The `{9}` indicates that there must be exactly 9 characters, and the `^` and `$` characters anchor the expression to the beginning and end of the string, respectively.

Figure 21.1. We can use a regular expression to exactly describe a valid --board.



You could manually validate this using the magic of `all()` again to check:

1. Is the length of `board` exactly 9 characters?
2. Is it true that each of the characters is one of those allowed?

Here is one way to write it:

```
>>> board = '...XXXOOO'
>>> len(board) == 9 and all([c in '.XO' for c in board])
True
```

The `all()` part is checking this:

```
>>> [c in '.XO' for c in board]
[True, True, True, True, True, True, True, True]
```

Since each character `c` ("cell") in `board` is in the allowed set of characters, all the comparisons are `True`. If we change one of the characters, a `False` will show up:

```
>>> board = '...XXXOOA'
>>> [c in '.XO' for c in board]
[True, True, True, True, True, True, True, True, False]
```

And any `False` value in an `all()` expression will return `False`:

```
>>> all([c in '.XO' for c in board])
False
```

The last piece of validation checks if the --cell being set to --player is already occupied:

```
if args.player and args.cell and args.board[args.cell - 1] in 'XO':
    parser.error(f"--cell \"{args.cell}\" already taken")
```

Because the --cell starts counting from 1 instead of 0, we must subtract 1 when we use it as an index into the --board argument. Given the following inputs where the first cell has been set to X and now 0 wants the same cell:

```
>>> board = 'X.....'
>>> cell = 1
>>> player = 'O'
```

We can ask if the value in `board` at `cell - 1` has already been set:

```
>>> board[cell - 1] in 'XO'
True
```

Or you could instead check if that position is *not* a dot:

```
>>> board[cell - 1] != '.'
True
```

It's rather exhausting to validate all the inputs, but this is the only way to ensure that the game is played properly. In the `main()` function, we can use the arguments to possibly mutating the board of the game. At this point, we've completely validated that

we have good values for `player` and `cell` and that we are allowed to alter board at a given position. I decided to make board into a list precisely because I might need to alter it in this way:

```
if player and cell:
    board[cell - 1] = player
```

21.2.2 Formatting the board

Now it's time to create the grid. I chose to create a function that returns a `str` that I could test rather than directly printing the grid. Here is my version:

```
def format_board(board):
    """Format the board"""

    cells = [str(i) if c == '.' else c for i, c in enumerate(board, start=1)] ①
    bar = '-----'
    cells_tmpl = '| {} | {} | {} |'
    return '\n'.join([
        bar,
        cells_tmpl.format(*cells[:3]), bar, ②
        cells_tmpl.format(*cells[3:6]), bar,
        cells_tmpl.format(*cells[6:]), bar
    ])
```

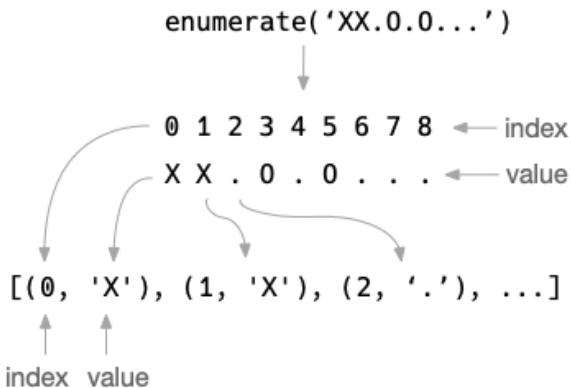
- ① I used a list comprehension to iterate through each position and character of board using the `enumerate()` function. Because I would rather start counting from index position 1 instead of 0, I use the `start=1` option. If the character is a dot, I want to print the position as the cell number, otherwise print the character which will be X or O.
- ② The "splat" (*) is a shorthand to expand the list returned by the list slice operation into values that the `str.format()` function can use. The "splat" syntax of `*cell[:3]` is a shorter way of writing the code like so:

```
return '\n'.join([
    bar,
    cells_tmpl.format(cells[0], cells[1], cells[2]), bar,
    cells_tmpl.format(cells[3], cells[4], cells[5]), bar,
    cells_tmpl.format(cells[6], cells[7], cells[8]), bar
])
```

The `enumerate()` function returns a list of tuples that include the index and value of each element in a list. Since it's a lazy function, I must use the `list()` function in the REPL to view the values:

```
>>> board = 'XX.O.O...'
>>> list(enumerate(board))
[(0, 'X'), (1, 'X'), (2, '.'), (3, 'O'), (4, '.'), (5, 'O'), (6, '.'), (7, '.'), (8, '.')]
```

Figure 21.2. The `enumerate()` function will return the index and value of items in a series. By default, the initial index is 0.



In this instance, we would rather start counting at 1, so we can use the `start=1` option:

```
>>> list(enumerate(board, start=1))
[(1, 'X'), (2, 'X'), (3, '.'), (4, '0'), (5, '.'), (6, '0'), (7, '.'), (8, '.'), (9, '.')]
```

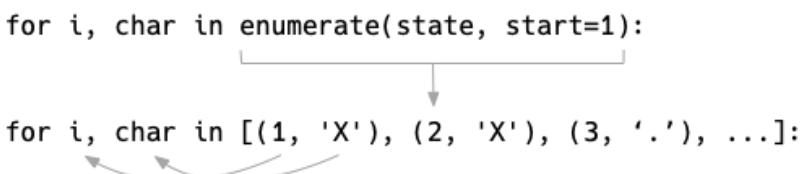
Our list comprehension could be written instead as a `for` loop, but since the intention is to create a `list` I would definitely recommend the above version:

```
cells = []
for i, char in enumerate(board, start=1):
    cells.append(str(i) if char == '.' else char)
```

- ① Initialize an empty list to hold the `cells`.
- ② Unpack each tuple of the index (starting at 1) and value of each character in `board` into the variable variables `i` (for "integer") and `char`.
- ③ If the `char` is a dot, then we want to use the `str` version of the `i` value; otherwise we use the `char` value.

Here is a visualization of how the `enumerate()` is unpacked into `i` and `char`:

Figure 21.3. The tuples containing the index and values returned by `enumerate()` can be assigned to two variables in the `for` loop.



This version of `format_board()` passes all the tests found in `unit.py`.

21.2.3 Finding the winner

The last major piece to this program is determine if either player has won by placing three of their marks in a row horizontally, vertically, or diagonally.

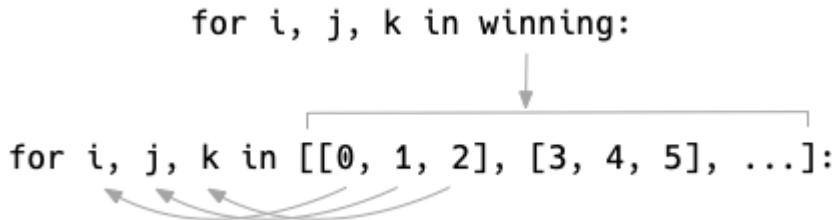
```
def find_winner(board):
    """Return the winner"""

    winning = [[0, 1, 2], [3, 4, 5], [6, 7, 8], [0, 3, 6], [1, 4, 7], ①
               [2, 5, 8], [0, 4, 8], [2, 4, 6]]

    for player in ['X', 'O']:
        for i, j, k in winning: ②
            combo = [board[i], board[j], board[k]]
            if combo == [player, player, player]:
                return player
```

- ① There are 8 winning positions — the three horizontal rows, the three vertical columns, and the two diagonals — so I decided to create a list where each element is also a list that contains the three cells in a winning configuration.
- ② It's typical to use `i` as a variable name for "integer" values, especially when their life is rather brief as here. When more similar names are needed in the same scope, it's also common to use `j`, `k`, `l`, etc. You may prefer to use names like `cell1`, `cell2`, and `cell3`, which are more descriptive but also longer to type. The unpacking of the cell values is exactly the same as the unpacking of the tuples in the above `enumerate()` code.

Figure 21.4. As with the above unpacking of the `enumerate()` tuples, each list of three elements can be unpacked into three variables in the `for` loop.



The rest of the code is checking if either X or O is the only character at each of the three positions. I worked out half a dozen ways to write this, but I'll just share this one alternate version that uses two of my favorite functions, `all()` and `map()`:

```
for combo in winning:
    group = list(map(lambda i: board[i], combo)) ①
    for player in ['X', 'O']:
        if all(x == player for x in group): ②
            return player ③
```

- ① Iterate through the tuple `combo` in `winning`.
- ② Use `map()` to get the value of `board` at each position in `combo`.
- ③ Check for each player X and O.
- ④ See if `all()` the values in the group are equal to the given player.

- ⑤ If so, return that player.

If a function has no explicit return or never executes a return as would be the case here when there is no winner, then Python will use the None value as the default return. We'll interpret that to mean there is no winner when we print the outcome of the game:

```
winner = find_winner(board)
print(f'{winner} has won!' if winner else 'No winner.')
```

That covers our version of the game that plays just one round of a game of Tic-Tac-Toe. In the next chapter, we'll expand these ideas into an interactive version that starts with a blank board and dynamically requests user input to play the game.

21.3 Review

- Our program uses a str value to represent the board of the Tic-Tac-Toe board with nine characters representing X, O, or . to indicate a taken or empty cell, respectively, but we sometimes convert that to a list to make it easier to modify.
- A regular expression is a handy way to validate the initial board. We can declaratively describe that it should be a string exactly nine characters long composed only of the characters ., X, and O.
- The any() function is like chaining or between multiple boolean values. It will return True if *any* of the values is "truthy."
- The all() function is like using and between multiple boolean values. It will return True only if every one of the values is "truthy."
- The enumerate() function will return the list index and value for each element in an iterable like a list.

21.4 Going further

- Write a game that will play one hand of a card game like Blackjack ("Twenty-one") or "War"

Tic-Tac-Toe Redux: An interactive version with type hints

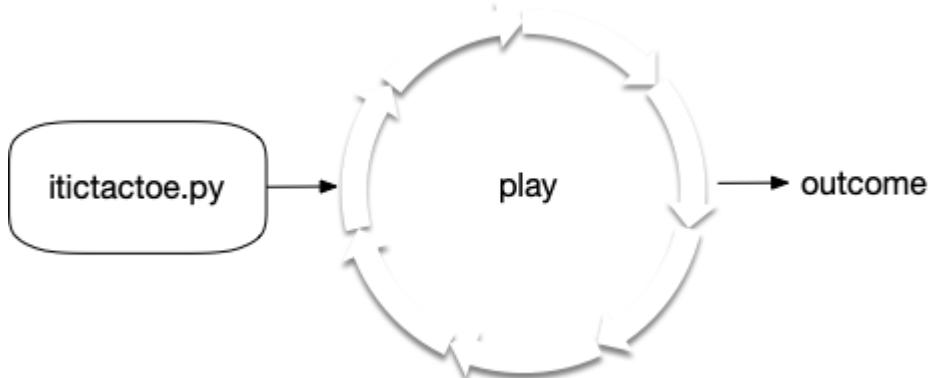
In this last exercise, we're going to revisit the Tic-Tac-Toe game from the previous chapter. That version played one round of the game by accepting some initial --board and then modifying it if there were also valid options for --player and --cell. It printed the one board and the winner, if any. We're going to extend those ideas into a version that will always start from an empty board and will play as many rounds as needed for the game to end with a winner or a draw. This program will be different from all the other programs in this book in that it will accept no command-line arguments. The game will always start with a blank "board" and with the X player going first. It will use the `input()` function to interactively ask each player, X and then O for a move. Any invalid move such as choosing an occupied or non-existing cell will be rejected. At the end of each round, the game will decide to stop if it determines there is a win or a draw.



22.1 Writing `itictactoe.py`

This is the one program where I don't provide an integration test since the program

doesn't take any arguments. This also makes it difficult to show a string diagram because the output of the program will be different depending on the moves you make. Still, here is an approximation of how we could think of the program starting with no inputs and then looping until some outcome is determined or the player quits:



I would encourage you to start off by running the `itictactoe.py` program to play a few rounds of the game. The first thing you may notice is that it should clear the screen of any text and show you an empty board along with a prompt for the X player's move. I'll type `1<Enter>`:

```

-----
| 1 | 2 | 3 |
-----
| 4 | 5 | 6 |
-----
| 7 | 8 | 9 |
-----
Player X, what is your move? [q to quit]: 1
  
```

Then you will see that the cell 1 is now occupied by X and the player has switched to 0:

```

-----
| X | 2 | 3 |
-----
| 4 | 5 | 6 |
-----
| 7 | 8 | 9 |
-----
Player 0, what is your move? [q to quit]:
  
```

If I choose 1 again, I am told that cell is already taken.

```

-----
| X | 2 | 3 |
-----
| 4 | 5 | 6 |
-----
| 7 | 8 | 9 |
-----
Cell "1" already taken
Player 0, what is your move? [q to quit]:
  
```

Note that the player is still 0 because the previous move was invalid. It's likewise if I put in some value that cannot be converted to an integer:

```
| X | 2 | 3 |
-----
| 4 | 5 | 6 |
-----
| 7 | 8 | 9 |
-----
Invalid cell "biscuit", please use 1-9
Player 0, what is your move? [q to quit]:
```

Or a valid integer that is out of range:

```
| X | 2 | 3 |
-----
| 4 | 5 | 6 |
-----
| 7 | 8 | 9 |
-----
Invalid cell "10", please use 1-9
Player 0, what is your move? [q to quit]:
```

You should be able to reuse many of the ideas from your previous version to validate the user input.

If I play the game to a conclusion where one player gets three in a row, it halts the game and proclaims the victor:

```
| X | 0 | 3 |
-----
| 4 | X | 6 |
-----
| 7 | 0 | 9 |
-----
Player X, what is your move? [q to quit]: 9
X has won!
```

22.1.1 Tuple talk

In this version, we'll write a interactive game that always starts with an empty grid and plays as many rounds as necessary to reach a conclusion with a win or a draw. The idea of "state" in the last game was limited to the board — which players were in which cells. This version requires us to track quite a few more variables in our game state:

1. The cells of the **board**, like ...X0...X0
2. The current **player**, either X or 0
3. Any **error** such as the player entering a cell that is occupied or does not exist or a value that cannot be converted to a number.
4. Whether the user wishes to **quit** the game early.
5. Whether the game is a **draw**, which is when all the cells of the grid are occupied but there is no winner and so we must quit.

6. The **winner**, if any, so I know when to quit.

You don't need to write your program exactly the way I wrote mine, but you still may find yourself needing to keep track of many items. A `dict` is a natural data structure for that, but I'd like to introduce a new data structure called a "named tuple" as it plays nicely with Python's type hints which will figure prominently in my solution.

We've encountered tuples throughout the exercises when they've been returned by something like `match.groups` when a regular expression contains capturing parentheses like in "Rhymer" or "Mad Libs" or using `zip` to combine two lists like in "WOD." A tuple is an immutable list, and we'll explore how that immutability can prevent us from introducing subtle bugs into our programs.

You create a tuple whenever you put commas between values:

```
>>> cell, player
(1, 'X')
```

It's most common to put parentheses around them to make it more explicit:

```
>>> (cell, player)
(1, 'X')
```

We could assign this to a variable called `state`:

```
>>> state = (cell, player)
>>> type(state)
<class 'tuple'>
```

We index into a tuple using list index values:

```
>>> state[0]
1
>>> state[1]
'X'
```

Unlike with a list, we cannot change any of the values inside the tuple:

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

It's going to be inconvenient remembering that the first position is the `cell` and the second position is the `player`, and it will get much worse when we add all the other fields. We could switch to using a `dict` so that we can use strings to access the values of `state`, but dictionaries are mutable, and it's also easy to misspell a key name.

22.1.2 **Named Tuples**

It would be nice to combine the safety of an immutable tuple with named fields, which is exactly what we get with the `namedtuple` function. First you must import it from the `collections` module:

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

The `namedtuple` function allows us to describe a new class for values. Let's say we want to create a class that describes the idea of State. A class is a group of variables, data, and functions that together can be used to represent some idea. The Python language itself, for example, has the `str` class that represents the idea of a sequence of characters that can be contained in a variable that has some `len` (length), which can be converted to uppercase with `str.upper()`, can be iterated with a `for` loop, and so forth. All these ideas are grouped into the `str` class, and we've used `help(str)` to read the documentation for that class inside the REPL.

It's common practice to capitalize class names. The class name is the first argument we pass to `namedtuple`, and the second argument is a list of the field names in the class:

```
>>> State = namedtuple('State', ['cell', 'player'])
```

We've just created a new type called `State`!

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

Just as there is a `list` type that is also a function called `list()`, we can use the function `State` to create a named tuple of the type `State` that has two named fields, `cell` and `player`:

```
>>> state = State(1, 'X')
>>> type(state)
<class '__main__.State'>
```

We can still access the fields with index values like any `list` or `tuple`:

```
>>> state[0]
1
>>> state[1]
'X'
```

But we can also use their names, which is much nicer. Notice there are no parentheses at the end as we are accessing a field, not calling a method:

```
>>> state.cell
1
>>> state.player
'X'
```

Because `state` is a `tuple`, we cannot mutate the value once it has been created:

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

This is actually *good* in many instances. It's often quite dangerous to change your data values once your program has started. **You should use tuples or named tuples whenever you want a list- or dictionary-like structure that cannot be accidentally modified.**

There is a problem, however, in that there's nothing to prevent us from instantiating a state with the fields out of order *and of the wrong types* — `cell` should be an `int` and `player` should be a `str`!

```
>>> state2 = State('0', 2)
>>> state2
State(cell='0', player=2)
```

In order to avoid that, you can use the field names such that their order no longer matters:

```
>>> state2 = State(player='0', cell=2)
>>> state2
State(cell=2, player='0')
```

And now you have data structure that looks like a `dict` but has the immutability of a `tuple`!

22.1.3 Adding type hints

We still have a big problem in that there's nothing preventing us from assigning a `str` to the `cell` which ought to be an `int` and vice versa for `int` and `player`:

```
>>> state3 = State(player=3, cell='X')
>>> state3
State(cell='X', player=3)
```

Starting in Python 3.6, the `typing` module allows you to add "type hints" to describe the data types for variables. You should read PEP 484 (www.python.org/dev/peps/pep-0484/) for more information, but the basic idea is that we can use this module to describe the appropriate types for variables and type signatures for functions.

The `typing` module defines a `NamedTuple` class that we can use as the base for a class of our own. We can create our own classes to represent ideas such as our game `State`. First we need to import from the `typing` module the classes we'll need such as `NamedTuple`, `List`, and `Optional`, the last of which describes a type that could be `None` or some other class like a `str`:

```
from typing import List, NamedTuple, Optional
```

And now we can specify a `State` class with named fields, types, and *even default values* to represent the initial state of the game where the board is empty (all dots) and the player X goes first. Note that I decided to store the board as a list of characters rather than a `str`:

```
class State(NamedTuple):
    board: List[str] = list('.' * 9)
    player: str = 'X'
    quit: bool = False
    draw: bool = False
    error: Optional[str] = None
    winner: Optional[str] = None
```

We can use the `State()` function to create a new value set to the initial state:

```
>>> state = State()
>>> state.board
['.', '.', '.', '.', '.', '.', '.', '.', '.']
>>> state.player
'X'
```

You can override any default value by providing the field name and a value. For instance we can start the game off with the player 0 by specifying `player='0'`. Any field we don't specify will use the default:

```
>>> state = State(player='0')
>>> state.board
['.', '.', '.', '.', '.', '.', '.', '.', '.']
>>> state.player
'0'
```

We get an exception if we misspell a field name like `playre` instead of `player`:

```
>>> state = State(playre='0')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __new__() got an unexpected keyword argument 'playre'
```

22.1.4 Type verification with mypy

As nice as all the above is, *Python will not generate a run-time error if we assign an incorrect type*. Here I can assign the `quit` to a `str` value `'True'` instead of the `bool` value `True` and nothing at all happens:

```
>>> state = State(quit='True')
>>> state.quit
'True'
```

The benefit to use type hints comes from using a program like `mypy` to check our code. Let's place all this code into a small program called `typehints.py` in the repo:

```
#!/usr/bin/env python3
""" Demonstrating type hints """

from typing import List, NamedTuple, Optional

class State(NamedTuple):
    board: List[str] = list('.' * 9)
    player: str = 'X'
    quit: bool = False      ①
    draw: bool = False
    error: Optional[str] = None
    winner: Optional[str] = None

state = State(quit='False') ②
print(state)
```

① `quit` is defined as a `bool`, which means it should only allow values of `True` and `False`.

- ② We are assigning the `str` value 'True' instead of the `bool` value `True`, which might be an easy mistake to make especially in a very large program. We'd like to know this type of error will be caught!

The program will execute *with no errors*:

```
$ ./typehints.py
State(board=['.', '.', '.', '.', '.', '.', '.', '.', '.'], player='X', \
quit='False', draw=False, error=None, winner=None)
```

But the `mypy` program will report the error of our ways:

```
$ mypy typehints.py
typehints.py:16: error: Argument "quit" to "State" has incompatible type "str"; expected
"bool"
Found 1 error in 1 file (checked 1 source file)
```

If I correct the program like so:

```
#!/usr/bin/env python3
""" Demonstrating type hints """

from typing import List, NamedTuple, Optional

class State(NamedTuple):
    board: List[str] = list('.' * 9)
    player: str = 'X'
    quit: bool = False ①
    draw: bool = False
    error: Optional[str] = None
    winner: Optional[str] = None

state = State(quit=True) ②
print(state)
```

① Again, the `quit` is a `bool` value.

② We have to assign an actual `bool` value in order to pass muster with `mypy`.

Now `mypy` will be satisfied:

```
$ mypy typehints2.py
Success: no issues found in 1 source file
```

22.1.5 Updating immutable structures

If one of the advantages to using `NamedTuples` is their *immutability*, then how will we keep track of changes to our program? Consider our initial state of an empty grid with the player X going first:

```
>>> state = State()
```

Imagine X takes cell "1", so we need to change the board to X..... and the player to 0. We can't directly modify `state`:

```
>>> state.board=list('X.....')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

I could use the `State` class to overwrite the existing state. That is, since I can't change anything *inside* the `state` variable, I can instead point the `state` to an entirely new value. We did this in exercises²⁷ where we needed to change a `str` value because they are also *immutable* in Python.

To do this, I can copy all the current values that haven't changed for `quit` and such:

```
>>> state = State(board=list('X.....'), player='0', quit=state.quit, \
    draw=state.draw, error=state.error, winner=state.winner)
```

The `namedtuple._replace()` method, however, provides a much simpler way to do this. Only the values provided are changed, and the result is a new `State`:

```
>>> state = state._replace(board=list('X.....'), player='0')
```

We overwrite our existing `state` with the return from `state._replace` because the original `state` is not changed.

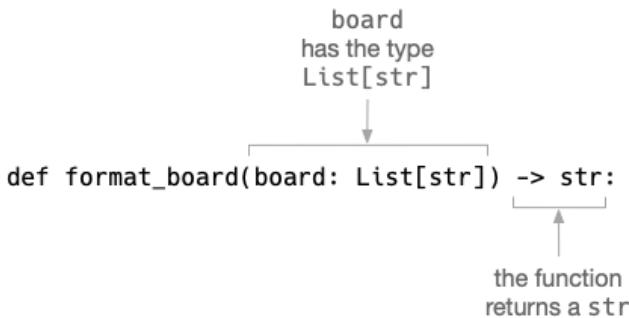
```
>>> state
State(board=['X', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.'], player='0', \
    quit=False, draw=False, error=None, winner=None)
```

This is much more convenient than having to list all the fields — we only need to specify the fields that changed. We are also prevented from accidentally modifying any of the other fields, and we are likewise prevented from forgetting any fields or setting them to the wrong types!

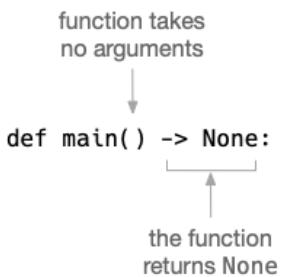
22.1.6 Adding type hints to function definitions

Now let's look at how we can add type hints to our function definitions! For an example, we can modify our `format_board()` function to indicate that it takes the parameter called `board` which is a list of string values by adding `board: List[str]`. Additionally, the function returns a `str` value, so we add `→ str` after the colon on the `def` to indicate this:

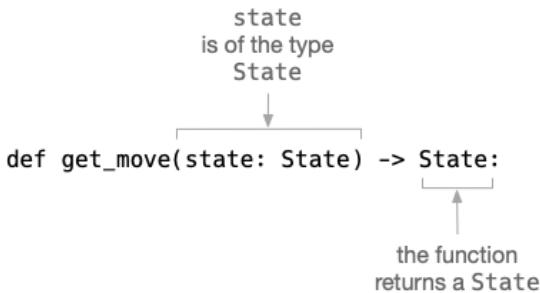
²⁷ See "Apples and Bananas" solution 2.



The annotation for `main()` indicates that the `None` value is returned:



What's really terrific is that we can define a function that takes a value of the type `State`, and `mypy` would check that this kind of value is actually being passed!



Let's look at an interactive solution that incorporates all of these ideas to create a program that has the additional benefits of data immutability and type safety!

22.2 Interactive solution

```

#!/usr/bin/env python3
""" Interactive Tic-Tac-Toe using NamedTuple """
from typing import List, NamedTuple, Optional
class State(NamedTuple):
    board: List[str] = list('.' * 9)
    player: str = 'X'
  
```

(1)

(2)

```

quit: bool = False
draw: bool = False
error: Optional[str] = None
winner: Optional[str] = None

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

    state = State()                                ③

    while True:
        print("\033[H\033[J")
        print(format_board(state.board))            ⑤

        if state.error:                            ⑦
            print(state.error)

        state = get_move(state)                   ⑧

        if state.quit:                           ⑨
            print('You lose, loser!')
            break
        elif state.winner:                      ⑩
            print(f'{state.winner} has won!')
            break
        elif state.draw:                        ⑪
            print("All right, we'll call it a draw.")
            break

# -----
def get_move(state: State) -> State:
    """Get the player's move"""                  ⑫

    player = state.player                      ⑬
    cell = input(f'Player {player}, what is your move? [q to quit]: ') ⑭

    if cell == 'q':                           ⑮
        return state._replace(quit=True)         ⑯

    if not (cell.isdigit() and int(cell) in range(1, 10)): ⑰
        return state._replace(error=f'Invalid cell "{cell}", please use 1-9') ⑱

    cell_num = int(cell)                     ⑲
    if state.board[cell_num - 1] in 'XO':      ⑳
        return state._replace(error=f'Cell "{cell}" already taken') ⑳

    board = state.board                      ㉑
    board[cell_num - 1] = player             ㉒
    return state._replace(board=board,
                          player='O' if player == 'X' else 'X',
                          winner=find_winner(board),
                          draw='.' not in board,
                          error=None)           ㉔

# -----

```

```

def format_board(board: List[str]) -> str:          ㉕
    """Format the board"""

    cells = [str(i) if c == '.' else c for i, c in enumerate(board, 1)]
    bar = '-----'
    cells_tmpl = '| {} | {} | {} |'
    return '\n'.join([
        bar,
        cells_tmpl.format(*cells[:3]), bar,
        cells_tmpl.format(*cells[3:6]), bar,
        cells_tmpl.format(*cells[6:]), bar
    ])

# -----
def find_winner(board: List[str]) -> Optional[str]: ㉖
    """Return the winner"""

    winning = [[0, 1, 2], [3, 4, 5], [6, 7, 8], [0, 3, 6], [1, 4, 7],
               [2, 5, 8], [0, 4, 8], [2, 4, 6]]

    for player in ['X', 'O']:
        for i, j, k in winning:
            combo = [board[i], board[j], board[k]]
            if combo == [player, player, player]:
                return player

    return None

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

```

- ① Import the classes we'll need from the `typing` module.
- ② Declare a class that is based on the `NamedTuple` class. Define field names, types, and defaults for the values this class can hold.
- ③ Instantiate our initial state as an empty grid and the first player as X.
- ④ Start an infinite loop using `while True`. When we have a reason to stop, we can break out of the loop.
- ⑤ Print a special sequence that most terminals will interpret as a command to clear the screen.
- ⑥ Print the current state of the board.
- ⑦ If there is an error (such as the user didn't choose a valid cell), print that.
- ⑧ Get the next move from the player. The `get_move()` accepts a `State` type and returns one, too. We overwrite our existing state variable each time through the loop.
- ⑨ If the user has decided to withdraw from the game prematurely, insult them and break from the loop.
- ⑩ If there is a winner, proclaim the victor and break from the loop.
- ⑪ If we have reached a stalemate where all cells are occupied but there is no winner, declare a draw and break from the loop.
- ⑫ Define a `get_move()` function that takes and returns a `State` type.
- ⑬ Copy the player from the state since I'll refer to it several times in the function body.
- ⑭ Use the `input()` function to ask the player for their next move. Tell them how to quit the game early so they don't have to use Ctrl-C (control key plus "c") to interrupt the program.
- ⑮ First check if they want to quit.

- ⑯ If so, replace the quit value of our state with True and return with the new state. Note that no other values in the state are modified.
- ⑰ Check if the user gave us a value that can be converted to a digit using `str.isdigit()` and if the `int` version of the value is in the valid range.
- ⑱ If not, return an updated state that has an error. Note that the current state and player remain unchanged so that the same player has a retry with the same board until they provide valid input.
- ⑲ After we have verified that `cell` is a valid integer value, convert it to an `int`.
- ⑳ See if the board is open at the indicated `cell`.
- ㉑ If not, return an updated state with an error. Again, nothing else about the state is changed, so we retry the round with the same player and state.
- ㉒ Copy the current board because we need to modify it and the `state.board` is immutable.
- ㉓ Use the `cell` value to update the board with the current player.
- ㉔ Return a new state value where we update the board, switch the player to the other, and check if there is a winner or a draw.
- ㉕ The only change from the previous version of this function is the addition of type hints. The function accepts a list of string values (the current board) and returns a formatted grid of the board state.
- ㉖ This is also the same function as before but with type hints. The function accepts the board as a list of strings and returns an optional `str` value which means it could also return `None`.

22.3 A version using TypedDict

New to Python 3.8 is the `TypedDict` class that looks very similar to a `NamedTuple`. One crucial difference is that you cannot (yet) set default values for the fields:

```
#!/usr/bin/env python3
"""
Interactive Tic-Tac-Toe using TypedDict """
from typing import List, Optional, TypedDict ①

class State(TypedDict): ②
    board: str
    player: str
    quit: bool
    draw: bool
    error: Optional[str]
    winner: Optional[str]
```

- ① Import `TypedDict` instead of `NamedTuple`.
- ② Base our `State` on a `TypedDict`.

We have to set our initial values when we instantiate a new `state`:

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

    state = State(board='.' * 9,
                  player='X',
```

```
quit=False,
draw=False,
error=None,
winner=None)
```

Syntactically, I prefer using `state.board` with the named tuple rather than the dictionary access of `state['board']`:

```
while True:
    print("\033[H\033[J")
    print(format_board(state['board']))

    if state['error']:
        print(state['error'])

    state = get_move(state)

    if state['quit']:
        print('You lose, loser!')
        break
    elif state['winner']:
        print(f"{state['winner']} has won!")
        break
    elif state['draw']:
        print('No winner.')
        break
```

Beyond the convenience of accessing the fields, I prefer the read-only nature of the `NamedTuple` to the mutable `TypedDict`. Note how in the `get_move()` function, I can change the `state`:

```
def get_move(state: State) -> State:
    """Get the player's move"""

    player = state['player']
    cell = input(f'Player {player}, what is your move? [q to quit]: ')

    if cell == 'q':
        state['quit'] = True
        return state

    if not (cell.isdigit() and int(cell) in range(1, 10)):
        state['error'] = f'Invalid cell "{cell}", please use 1-9'
        return state

    cell_num = int(cell)
    if state['board'][cell_num - 1] in 'XO':
        state['error'] = f'Cell "{cell}" already taken'
        return state

    board = list(state['board'])
    board[cell_num - 1] = player

    return State(
        board=''.join(board),
        player='O' if player == 'X' else 'X',
        winner=find_winner(board),
        draw='.' not in board,
        error=None,
        quit=False,
```

)

In my opinion, a `NamedTuple` has nicer syntax, default values, and immutability over the `TypedDict` version, so I prefer it. Regardless of which you might choose, the greater lesson I hope to impart is that we try to be explicit about the "state" of the program and when and how it changes.

22.3.1 Thinking about state

The idea of program state describes how a program can remember changes to variables over time. Our first version accepted a given `-state` and possibly values for `-cell` and `-player` that might alter the state. Then the board prints a representation of the state. In the interactive version, the state always begins as an empty grid, and the state changes with each round which we modeled as an infinite loop.



It is common in programs like this to see programmers use "global variables" that are declared at the top of a program outside of any function definitions so that they are *globally* visible throughout the program. While common, it's not considered a "best practice," and I would discourage you from ever using globals unless you can see no other way. I would suggest, instead, that you stick to small functions that accept all the values required and return a single type of value. I would also suggest you use data structures like typed, named tuples to represent program state, and that you guard the changes to state very carefully!

22.4 Review

- The first program used a `str` value to represent the "state" of the Tic-Tac-Toe board with nine characters representing X, O, or . to indicate a taken or empty cell, respectively.
- The second program used a `NamedTuple` to create a data structure for a much expanded idea of "state" that included many variables such as the current board, the current player, any errors, etc. The `NamedTuple` behaves a bit like a `dict`, a bit like an object, but retains the immutability of tuples.
- Type hints allow you to annotate variables and function parameters and return values.
- Python itself will ignore type hints at run-time, but `mypy` can use type hints to find logical errors in your code before you ever run it.
- Both `NamedTuple` and `TypedDict` allow you to create a novel type with defined fields and types that you can use as type hints to your own functions.

22.5 *Going further*

- Incorporate spicier insults. Maybe bring in your Shakespearean generator?
- Write a version that allows the user to play more games without quitting and restarting the program.
- Write other games like Hangman.

23

Epilogue

Well, that's the whole book. We came a long way from writing the "Crow's Nest" to an interactive Tic-Tac-Toe that incorporates a custom class based on named tuples and uses type hints! I hope you can see now how much we can do with Python's strings, lists, tuples, dictionaries, sets, and functions. I especially hope I've convinced you that, above all you should always write programs that are:

1. **Flexible** by taking command-line arguments
2. **Documented** because you use something like `argparse` to parse your arguments and produce "usage" statements
3. **Tested** by writing both *unit* tests for your functions and *integration* tests for your program as a whole

The people using your programs will really appreciate knowing how to use your program, how to make it behave differently, and that you took the time to verify your program is correct. Let's be honest, though — the person most likely to be using and modifying your programs will be you several months from now. I've heard it said that "Documentation is a love letter to your future self." All this work you put into make your programs good will be very appreciated by you when you come back to your code.

Now that you've worked through all the exercises and seen how to use the tests I've written, I would challenge you to go back to the beginning and read the `test.py` programs. If you intend to adopt test-driven development, you may find you can steal many ideas and techniques from those programs.

Further, each chapter included suggestions for how to extend the ideas and exercises presented. Go back and think about how you can use ideas you learned later in the book to improve or extend earlier programs. Here are some ideas:

- Ch 1 (Hello): Add an option to randomly select a greeting other than "Hello" from a list like "Hello," "Hola," "Salut," and "Ciao."
- Ch 2 (Crow's Nest): Allow the program to take one or more options and incorporate those into the output with the correct articles for each item joined on the Oxford comma.
- Ch 6 (Word Count): Add flags for `-l|-lines`, `-w|-words`, and `-c|-char` like the original `wc` such that only those columns will show in the output. That is, if only `-lines` is present, then the numbers of words and bytes will not be shown.
- Ch 7 (Gashlycrumb): Download "The Devil's Dictionary" by Ambrose Bierce from Project Gutenberg. Write a program to parse it into a program that will look up a word's definition if it appears in the text.
- Ch 16 (Scrambler): Use the scrambled text as the basis for encrypting messages. Force the scrambled words to uppercase, remove all the punctuation and spaces, then format the text into "words" of 5 characters followed by a space, with no more than 5 per line. Pad the end so that the text completely fills the last line. Can you make sense of the output?
- Ch 18 (Gemetria): Find all the words in the English dictionary that encode to the same value. Find the sums that have the highest and lowest number of words. How many words have the value 666?

I hope you've had as much fun writing the programs as I have had creating and teaching them. I want you to feel you now have dozens of programs and tests with ideas and functions you can steal to create even more programs.

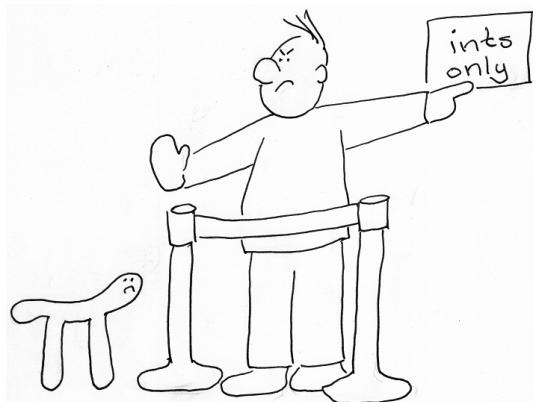
All the best to you in your coding adventures!



Ken

Using argparse

Often getting the right data into your program is a real chore. The argparse module makes it much easier to validate arguments from the user and generate useful error messages when they provide bad input. It's like our program's "bouncer," only allowing the right kinds of values into our program. Often half or more of the programs in this book can be handled simply by defining the arguments properly with argparse!



In Chapter 1, we ended up writing a very flexible program that could extend warm salutations to an optionally named entity such as the "World" or "Universe":

```
$ ./hello.py          ①
Hello, World!
$ ./hello.py --name Universe ②
Hello, Universe!
```

- ① When the program runs with no input values, it will use "World" for the entity to greet.
- ② The program can take an optional --name value to override the default.

The program would respond to the -h and --help flags with helpful documentation:

```
$ ./hello.py -h      ①
```

```
usage: hello.py [-h] [-n str] ②
Say hello ③
optional arguments:
  -h, --help      show this help message and exit ④
  -n str, --name str  The name to greet (default: World) ⑤
```

- ① The argument to the program is `-h`, which is the "short" flag to ask for help.
- ② This line shows a summary of all the options the program accepts. The square brackets `[]` around the arguments show that they are optional.
- ③ We set this as the description of the program.
- ④ We can use either the "short" name `-h` or the "long" name `--help` to ask the program for help on how to run it.
- ⑤ Our optional "name" parameter also has short and long names of `-n` and `--name`.

All of this is created by just two lines of code in the `hello.py` program:

```
parser = argparse.ArgumentParser(description='Say hello')
parser.add_argument('-n', '--name', default='World', help='Name to greet') ① ②
```

- ① The parser will parse the arguments for us. If the user provides unknown arguments or the wrong number of arguments, the program will halt with a usage statement.
- ② The only argument to this program is an optional `--name` value.

NOTE

You do not need to define the `-h` or `--help` flags. Those are generated automatically for you by `argparse`. In fact, you should never try to use those for other values because they are almost universal options that most users will expect.

The `argparse` module helps us define a parser for the arguments and generates help messages, saving us loads of time and making our programs look professional. Every program in this book is tested on different inputs, so you'll really understand how to use this module by the end. I would recommend you look over the documentation (docs.python.org/3/library/argparse.html). Now let's dig further into what this module can do for us. In this appendix, we will:

- Learn how to use `argparse` to handle positional parameters, options, and flags.
- Set default values for options.
- Use `type` to force the user to provide values like numbers or files.
- Use `choices` to restrict the values for an option.

A.1 Types of arguments

Command-line arguments can be classified as follows:

- **Positional arguments:** The order and number of the arguments is what determines their meaning. Some programs might expect, for instance, a file name as the first argument and an output directory as the second. Typically positional arguments are always required. Making them optional is difficult — how would

you write a program that accepts 2 or 3 arguments where the second and third ones are independent and optional? In the first version of `hello.py`, the name to greet was provided as a positional argument.

- **Named options:** Most command-line programs define a "short" name like `-n` (one dash and a single character) or a "long" name like `--name` (two dashes and a word) followed by some value like the name for the `hello.py` program. Named options allow for arguments to be provided in any order, so their *position* is not relevant; hence they are the right choice when the user is not required to provide them (they are "options," after all). It's good to provide reasonable default values for options. When we changed the required, positional name argument of `hello.py` to the optional `--name`, we used "World" for the default so that the program could run with no input from the user. Note that some languages like Java might define "long" names with a single dash like `-jar`.
- **Flags:** A "Boolean" value like "yes"/"no" or `True/False` is indicated by something that starts off looking like a named option but there is no value after the name, for example, `-d` or `--debug` to turn on debugging. Typically the presence of the flag indicates a `True` value for the argument; therefore, its absence would mean `False`, so `--debug` turns *on* debugging while its absence means it is off.

A.2 Starting off with `new.py`

Let's start a new program using either the `new.py` program. From the top level of the repository, you can execute this command:

```
$ bin/new.py foo.py
```

Or you could copy the template:

```
$ cp template/template.py foo.py
```

The resulting program will show you how to declare each of these argument types. Additionally, we can use `argparse` to validate the input like making sure that some argument is a number while some other argument is a file.

Let's look at the help generated by our new program:

```
$ ./foo.py -h ①
usage: foo.py [-h] [-a str] [-i int] [-f FILE] [-o] str ②

Rock the Casbah      ③

positional arguments: ④
  str                A positional argument

optional arguments: ⑤
  -h, --help          show this help message and exit ⑥
  -a str, --arg str   A named string argument (default: ) ⑦
  -i int, --int int   A named integer argument (default: 0) ⑧
  -f FILE, --file FILE A readable file (default: None) ⑨
  -o, --on            A boolean flag (default: False) ⑩
```

- ① Every program should respond to -h and --help with a help message.
- ② This is a brief summary of all the options that are described in greater detail below.
- ③ This is the description of our entire program.
- ④ This program defines one positional parameters, but we can have many more. We'll see below how to define those.
- ⑤ Optional arguments can be left out, so you should provide reasonable default values for them.
- ⑥ The -h and --help are always present when you use argparse, and you do not need to define them.
- ⑦ The -a or --arg option accepts some text which is often called a "string."
- ⑧ The -i or --int option must be an integer value. If the user provides "one" or "4.2," these will be rejected.
- ⑨ The -f or --file option must be a valid, readable file.
- ⑩ The -o or --on is a flag. Notice how the -f FILE description shows that some "FILE" value should follow the -f, but here there is no value that follows the option. The flag is either present or absent, and so it's either True or False, respectively.

A.3 Using argparse

The code to generate this is found in a function called `get_args` that looks like this. You are not required to have a function for this. You are welcome to put this code wherever you like, but sometimes just defining and validating the arguments can get rather long. I like to separate this idea, and I always call this function `get_args`, and I always define the function first in my program so that I can see it immediately when I'm reading the source code:

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

    parser = argparse.ArgumentParser(
        description='Rock the Casbah',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('positional',
                        metavar='str',
                        help='A positional argument')

    parser.add_argument('-a',
                        '--arg',
                        help='A named string argument',
                        metavar='str',
                        type=str,
                        default='')

    parser.add_argument('-i',
                        '--int',
                        help='A named integer argument',
                        metavar='int',
                        type=int,
                        default=0)

    parser.add_argument('-f',
                        '--file',
                        help='A readable file',
                        metavar='FILE',
```

```

        type=argparse.FileType('r'),
        default=None)

parser.add_argument('-o',
                    '--on',
                    help='A boolean flag',
                    action='store_true')

return parser.parse_args()

```

The `get_args` function which is defined like this:

```

def get_args():  
    """Get command-line arguments"""  


```

- ① The `def` keyword defines a new function. The arguments to the function are listed in the parentheses. Even though the `get_args` function takes no arguments, the parentheses are still required.
- ② The triple-quoted line after the function `def` is the "docstring" which serves as a bit of documentation for the function. Docstrings are not required, but they are good style, and `pylint` will complain if you leave them out.

A.3.1 ***Creating the parser***

The following line creates a parser that will deal with the arguments from the command line. To "parse" here means to derive some meaning from the order and syntax of the bits of text provided as arguments:

```

parser = argparse.ArgumentParser(  
    description='Argparse Python script',  
    formatter_class=argparse.ArgumentDefaultsHelpFormatter)  


```

- ① Call the `argparse.ArgumentParser` function to create a new parser.
- ② A short summary of your program's purpose.
- ③ The `formatter_class` argument tells `argparse` to show the default values in usage.

You should read the documentation for `argparse` to see all the other options you can use to define a parser or the parameters. In the REPL, you could start with `help(argparse)`, or you could look up the docs on the Internet at docs.python.org/3/library/argparse.html.

A.3.2 ***A positional parameter***

The following line will create a new *positional* parameter:

```

parser.add_argument('positional',  
    metavar='str',  
    help='A positional argument')  


```

- ① The lack of leading dashes makes this a positional parameter, not the name "positional."
- ② A hint to the user for the data type. By default, all arguments are strings.
- ③ A brief description of the parameter for the usage.

Remember that the parameter is not positional because the *name* is "positional." That's just there to remind you that it *is* a positional parameter. The argparse interprets the string 'positional' as a positional parameter *because the name does start any dashes*.

A.3.3 An optional string parameter

The following line creates an *optional* parameter with a short name of -a and a long name of --arg that will be a str with a default value of '' (the empty string). Note that you can leave off either the short or long name in your own programs, but it's good form to provide both. Most of the tests for the exercises will use both short and long option names.

```
parser.add_argument('-a',  
                  '--arg',  
                  help='A named string argument',  
                  metavar='str',  
                  type=str,  
                  default='')
```

- ① The short name.
- ② The long name.
- ③ Brief description for the usage.
- ④ Type hint for usage.
- ⑤ The actual Python data type (note the lack of quotes around str).
- ⑥ The default value.

If you wanted to make this a required, named parameter, you would remove the default and add required=True.

A.3.4 An optional numeric parameter

The following line creates the option called -i or --int that accepts an int (integer) with a default value of 0. If the user provides anything that cannot be interpreted as an integer, the argparse module will stop processing the arguments and will print an error message and a short usage statement:

```
parser.add_argument('-i',  
                  '--int',  
                  help='A named integer argument',  
                  metavar='int',  
                  type=int,  
                  default=0)
```

- ① The short name.
- ② The long name.
- ③ Brief description for usage.
- ④ Type hint for usage.
- ⑤ Python data type that the string must be converted to. You can also use float for a floating point value (a number with a fractional component like 3.14).
- ⑥ The default value.

One of the big reasons to define numeric arguments in this way is that `argparse` will convert the input to the correct type. That is, all values coming from the command are strings. It's the job of the program to convert the value to an actual numeric value. If you tell `argparse` that the option should be `type=int`, then when you ask the parser for the value, it will have already been converted to an actual `int` value. If the value provided by the user cannot be converted to an `int`, then the value will be rejected. That saves you a lot of time and effort!

A.3.5 An optional file parameter

The following line creates an option called `-f` or `--file` that will only accept a valid, readable file. This argument alone is worth the price of admission as it will save you oodles of time validating the input from your user. Note that pretty much every exercise that has a file input will have tests that pass *invalid* file arguments to ensure that your program rejects them.

```
parser.add_argument('-f',  
                  '--file',  
                  help='A readable file',  
                  metavar='FILE',  
                  type=argparse.FileType('r'),  
                  default=None)
```

- ① The short name.
- ② The long name.
- ③ Brief usage.
- ④ Type suggestion.
- ⑤ Says that the argument must name a readable ('r') file.
- ⑥ Default value.

The person running the program is responsible for providing the location of the file. For instance, if you created the `foo.py` program in the top level of the repository, there is a `README.md` file there. We could use that as the input to our program, and it will be accepted as a valid argument:

```
$ ./foo.py -f README.md foo  
str_arg = ""  
int_arg = "0"  
file_arg = "README.md"  
flag_arg = "False"  
positional = "foo"
```

If we provide a bogus `--file` argument like "blaragh," we will get an error message:

```
$ ./foo.py -f blaragh foo  
usage: foo.py [-h] [-a str] [-i int] [-f FILE] [-o] str  
foo.py: error: argument -f/--file: can't open 'blaragh': \\\n[Errno 2] No such file or directory: 'blaragh'
```

A.3.6 A flag

The flag option is slightly different in that it does not take a value like an string or

integer. Flags are either present or not, and they *usually* indicate that some idea is True or False. For instance, common flag is `--debug` to turn *on* debugging statements or `--verbose` to print extra messages to the user. When `--debug` is not present, the default value is "off" (or False), which we can represent using `action='store_true'`:

```
parser.add_argument('-o',
                   '--on',
                   help='A boolean flag',
                   action='store_true')
```

- ① Short name.
- ② Long name.
- ③ Brief usage.
- ④ What to do when this is present. When this flag is present, use the value True for on. The default value will be False when the flag is not present.

You could instead use `action='store_false'`, in which case on would be False when the flag is present and so the default value would be True. You could also store one or more constant values when the flag is present. Read the documentation for the various ways you can define this parameter. For the purposes of this book, we will only use a flag to turn "on" some behavior.

A.3.7 **Returning from get_args**

The final statement in `get_args` is to return the result of having the `parser` object parse the arguments. That is, the code that calls `get_args` will receive this value back:

```
return parser.parse_args()
```

This could fail because `argparse` finds that the user provided invalid arguments, for example, a string value when it expected a `float` or perhaps a misspelled filename. If the parsing succeeds, then we will have a way in our code to access all the values the user provided. Additionally, those values will be of the *types* that we indicated. That is, if we indicate that the `--int` argument should be an `int`, then when we ask for `args.int`, it will already be an `int`. If we define a file argument, we'll get an *open file handle*. That may not seem impressive now, but it's really enormously helpful.

If we refer to the `foo.py` program we generated, we see that the `main()` function calls `get_args`, so the the return from `get_args` goes back to the `main()`. From there, we can access all the values we just defined using the name of the positional parameters or the "long" name of the optional parameters:

```
def main():
    args = get_args()
    str_arg = args.arg
    int_arg = args.int
    file_arg = args.file
    flag_arg = args.on
    pos_arg = args.positional
```

A.4 Examples using argparse

Many of the program tests can be satisfied by learning how to use argparse effectively to validate the arguments to your programs. I think of the command line as the boundary of your program, and you need to be judicious about what you let in. You should always expect and defend against every argument being wrong.²⁸ Our hello.py program was an example of a single, positional argument and then a single, optional argument. Let's look at some more examples of how you can use argparse.

A.4.1 A single, positional argument

This is first version of our hello.py program that required a single argument of the name to greet:

```
#!/usr/bin/env python3
"""A single positional argument"""

import argparse

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

    parser = argparse.ArgumentParser(
        description='A single positional argument',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('name', metavar='name', help='The name to greet') ①

    return parser.parse_args()

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

    args = get_args()
    print('Hello, ' + args.name + '!') ②

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

- ① The name parameter does not start with dashes, so this is a *positional* parameter. The metavar will show up in the help to let the user know what this argument is supposed to be.
- ② Whatever is provided as the first positional argument to the program will be available in the args.name slot.

This program will not print the "Hello" line if it's not provided exactly one argument. If given nothing, it will print a brief usage about the proper way to invoke the program:

²⁸ I always think of the kid who will type "fart" for every input.

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

If we provide more than one argument, it complains again. Here "Emily" and "Bronte" are two arguments because spaces separate arguments on the command line, and so the program complains about getting a second argument that it does not have defined:

```
$ ./one_arg.py Emily Bronte
usage: one_arg.py [-h] name
one_arg.py: error: unrecognized arguments: Bronte
```

Only when we give the program exactly one argument will it run:

```
$ ./one_arg.py "Emily Bronte"
Hello, Emily Bronte!
```

While it may seem like overkill to use `argparse` for such a simple program, it actually means that `argparse` does quite a bit of error checking and validation of the arguments for us!

A.4.2 Two different positional arguments

Imagine you want two *different* positional arguments, like the `color` and `size` of an item to order. The color should be a `str`, and the size should be an `int` value. When you define them positionally, the order in which you declare them is the order in which the user must supply the arguments. Here we define `color` first and then `size`:

```
#!/usr/bin/env python3
"""Two positional arguments"""

import argparse

# -----
def get_args():
    """get args"""

    parser = argparse.ArgumentParser(
        description='Two positional arguments',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('color', ①
                        metavar='color',
                        type=str,
                        help='The color of the garment')

    parser.add_argument('size', ②
                        metavar='size',
                        type=int,
                        help='The size of the garment')

    return parser.parse_args()

# -----
def main():
    """main"""
```

```

args = get_args()
print('color =', args.color) ③
print('size =', args.size) ④

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

```

- ① This will be the first of the positional arguments because it is defined first. Notice that metavar has been set to 'color' instead of 'str' as it's more descriptive of the *kind* of string we expect — one that describes the "color" of the garment.
- ② This will be the second of the position arguments. Here metavar='size' which could be a number like 4 or a string like 'small', so it's still ambiguous!
- ③ The "color" argument is accessed via the name of the parameter color.
- ④ The "size" argument is accessed via the name of the parameter size.

Again, the user must provide exactly two positional arguments. No arguments triggers a short usage:

```
$ ./two_args.py
usage: two_args.py [-h] color size
two_args.py: error: the following arguments are required: color, size
```

Just one won't cut it. We are told that "size" is missing:

```
$ ./two_args.py blue
usage: two_args.py [-h] color size
two_args.py: error: the following arguments are required: size
```

If we give two strings like "blue" for the color and "small" for the size, the size value will be rejected because it needs to be an integer value:

```
$ ./two_args.py blue small
usage: two_args.py [-h] color size
two_args.py: error: argument size: invalid int value: 'small'
```

If we give two arguments, the second of which can be interpreted as an *int*, all is well:

```
$ ./two_args.py blue 4
color = blue
size = 4
```

Remember that *all* the arguments coming from the command line are strings. The command line doesn't require quotes around blue or the 4 to make them strings the way that Python does. On the command line, everything is a string, and all arguments are passed to Python as strings! When we tell argparse that the second argument needs to be an *int*, then argparse will do the work to attempt the conversion of



the string '4' to the integer 4. If you provide 4.1, that will be rejected, too:

```
$ ./two_args.py blue 4.1
usage: two_args.py [-h] str int
two_args.py: error: argument int: invalid int value: '4.1'
```

Positional arguments have the problem that the user is required to remember the correct order. In the case of switching a `str` and `int`, `argparse` will detect invalid values:

```
$ ./two_args.py 4 blue
usage: two_args.py [-h] COLOR SIZE
two_args.py: error: argument SIZE: invalid int value: 'blue'
```



Imagine, however, a case of two strings or two numbers which represent two *different* values like a car's make and model or a person's height and weight. How could you detect that the arguments are reversed? Generally speaking, I only ever create programs that take exactly one positional argument or one or more *of the same thing* like a list of files to process.

A.4.3 **Restricting values using choices**

In our previous example, note that there's nothing stopping the user from providing *two integer values*:

```
$ ./two_args.py 1 2
color = 1
size = 2
```

The 1 is a "string." It may look like a "number" to you, but it is actually the *character* "1." That is a valid string value, and so our program accepts it.

Our program would also accept a "size" of -4, which clearly is not a valid size:

```
$ ./two_args.py blue -4
color = blue
size = -4
```

How could we ensure that the user provides both a valid "color" and "size"? Let's say we only offer shirts in only primary colors. We can pass in a `list` of valid values using the `choices` option. Here we restrict the `color` to one of "red," "yellow," or "blue." Additionally, we can use `range(1, 11)` to generate a list of numbers from 1-10 (11 isn't included!) as the valid sizes for our shirts:

```
#!/usr/bin/env python3
"""
Choices"""

import argparse
```

```

# -----
def get_args():
    """get args"""

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

    parser.add_argument('color',
                        metavar='str',
                        help='Color',
                        choices=['red', 'yellow', 'blue']) ①

    parser.add_argument('size',
                        metavar='size',
                        type=int,
                        choices=range(1, 11), ②
                        help='The size of the garment')

    return parser.parse_args()

# -----
def main():
    """main"""

    args = get_args()
    print('color =', args.color) ③
    print('size =', args.size) ④

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

```

- ① The choices option takes a list of values. argparse stop the program if the user fails to supply one of these.
- ② Our user must choose from the numbers 1-10 or argparse will stop with an error.
- ③ If our program makes it to this point, we know that args.color will definitely be one of those values and that args.size is an integer value in the range of 1-10. The program will never get to this point unless both arguments are valid!

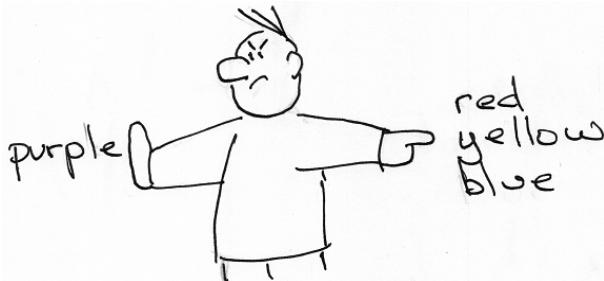
Any value not present in the list will be rejected and the user will be shown the valid choices. Again, no value is rejected:

```
$ ./choices.py
usage: choices.py [-h] color size
choices.py: error: the following arguments are required: color, size
```

If we provide "purple," it will be rejected because it is not in choices we defined. The error message that argparse produces tells the user the problem ("invalid choice") and even lists the acceptable colors!

```
$ ./choices.py purple 1
usage: choices.py [-h] color size
choices.py: error: argument color: \
```

```
invalid choice: 'purple' (choose from 'red', 'yellow', 'blue')
```



Likewise with a negative "size" argument:

```
$ ./choices.py red -1
usage: choices.py [-h] color size
choices.py: error: argument size: \
invalid choice: -1 (choose from 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

Only when both arguments are valid may we continue:

```
$ ./choices.py red 4
color = red
size = 4
```

That's really quite a bit of error checking and feedback that you never have to write. The best code is code you don't write!

A.4.4 **Two of the same positional arguments**

If you were writing a program that adds two numbers, you could define them as two positional arguments, like `number1` and `number2`. Since they are the same kinds of arguments (two numbers that we will add), it might make more sense to use the `nargs` option to tell `argparse` that you want exactly two of some thing:

```
#!/usr/bin/env python3
"""nargs=2"""

import argparse

# -----
def get_args():
    """get args"""

    parser = argparse.ArgumentParser(
        description='nargs=2',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('numbers',
                        metavar='int',
                        nargs=2, ①
                        type=int, ②
                        help='Numbers')
```

```

    return parser.parse_args()

# -----
def main():
    """main"""

    args = get_args()
    n1, n2 = args.numbers
    print(f'{n1} + {n2} = {n1 + n2}') ④

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

```

- ① The nargs=2 will require exactly 2 values.
- ② Each value must be parseable as an integer value or the program will error out.
- ③ Since we defined that there are exactly two values for numbers, we can copy them into two variables.
- ④ Because these are actual int values, the result of + will be numeric addition and not string concatenation.

The help indicates we want two numbers:

```
$ ./nargs2.py
usage: nargs2.py [-h] int int
nargs2.py: error: the following arguments are required: int
```

When we provide two good integer values, we get their sum:

```
$ ./nargs2.py 3 5
3 + 5 = 8
```



Notice that argparse converts the n1 and n2 values to actual integer values. Change the type=int to type=str and you'll see that the program will print 35 instead of 8 because the + operator in Python both adds numbers and concatenates strings!

```
>>> 3 + 5
8
>>> '3' + '5'
'35'
```

A.4.5 One or more of the same positional arguments

You could expand your 2-number adder into one that sums as many numbers as you provide. When you want *one or more* of some argument, you can use nargs='+' :

```
#!/usr/bin/env python3
"""nargs='+'"""

import argparse
```

```

# -----
def get_args():
    """get args"""

    parser = argparse.ArgumentParser(
        description='nargs=+',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('numbers',
                        metavar='INT',
                        nargs='+', ①
                        type=int, ②
                        help='Numbers')

    return parser.parse_args()

# -----
def main():
    """main"""

    args = get_args()
    numbers = args.numbers ③

    print('{} = {}'.format(' + '.join(map(str, numbers)), sum(numbers))) ④

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

```

① The + will make nargs accept one or more values.

② The int means that all the values must be integer values.

③ numbers will be a list with at least one element.

④ Don't worry if you don't understand this line. You will by the end of the book!

Note that this will mean args.numbers is always a list. Even if the user provides just one argument, args.numbers will be a list containing that one value:

```

$ ./nargs+.py 5
5 = 5
$ ./nargs+.py 1 2 3 4
1 + 2 + 3 + 4 = 10

```

You can also use nargs='*' to indicate *zero or more* of an argument, and nargs='?' means *zero or one*.

A.4.6 File arguments

So far we've seen that we can define that an argument should be of a type like str (which is the default), int, or float. There are many exercises that require a file as input, and you can use the type of argparse.FileType('r') to indicate that an argument must be a *file* which is *readable* (the 'r' part).

Here is an example showing an implementation in Python of the command cat -

n where cat will *concatenate* files and the -n says to *number* the lines of output:

```
#!/usr/bin/env python3
"""Python version of `cat -n`"""

import argparse

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

    parser = argparse.ArgumentParser(
        description='Python version of `cat -n`',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file',
                        metavar='FILE',
                        type=argparse.FileType('r'), ①
                        help='Input file')

    return parser.parse_args()

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

    args = get_args()

    for i, line in enumerate(args.file, start=1): ②
        print(f'{i:6} {line}', end='')

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

① The argument will be rejected if it does not name a valid, readable file.

② The value of args.file is an open file handle that we can directly read. Again, don't worry if you don't understand this code. We'll talk all about file handles in the exercises!

When I define an argument as type=int, I get back an actual int value. Here, I define the file argument as a file type, and so I receive an *open file handle*. If I had defined the file argument as a string, I would have to manually check if it were a file and then use open to get a file handle:

```
#!/usr/bin/env python3
"""Python version of `cat -n`, manually checking file argument"""

import argparse
import os

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

    parser = argparse.ArgumentParser(
```

```

        description='Python version of `cat -n`',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file', metavar='str', type=str, help='Input file')

    args = parser.parse_args()          ①

    if not os.path.isfile(args.file): ②
        parser.error(f'"{args.file}" is not a file') ③

    args.file = open(args.file)        ④

    return args

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

    args = get_args()

    for i, line in enumerate(args.file, start=1):
        print(f'{i:6} {line}', end='')

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

```

- ① Intercept the arguments.
- ② Check if the file argument is *not* a file.
- ③ Print an error message and exit the program with a non-zero value.
- ④ Replace the file with an open file handle.

With the file type definition, you don't have to write any of this code.

You can also use `argparse.FileType('w')` to indicate that you want the name of a file that can be opened for *writing* (the '`w`'). You can pass additional arguments for how to open the file like the encoding. See the documentation for more information.

A.4.7 ***Manually checking arguments***

It's also possible to manually validate arguments before you return from `get_args`. For instance, we can define that `--int` should be an `int` but how can we require that it must be between 1 and 10? One fairly simple way to do this is to manually check the value. If there is a problem, you can use the `parser.error` function to halt execution of the program, print an error message along with the short usage, and then exit with an error:

```

#!/usr/bin/env python3
"""Manually check an argument"""

import argparse

# -----

```

```

def get_args():
    """Get command-line arguments"""

    parser = argparse.ArgumentParser(
        description='Manually check an argument',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('-v',
                        '--val',
                        help='Integer value between 1 and 10',
                        metavar='int',
                        type=int,
                        default=5)

    args = parser.parse_args() ①
    if not 1 <= args.val <= 10: ②
        parser.error(f"--val '{args.val}' must be between 1 and 10") ③

    return args ④

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

    args = get_args()
    print(f'val = "{args.val}"')

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

```

- ① Parse the arguments.
- ② Check if the args.int value is *not* between 1 and 10.
- ③ Call parser.error with an error message. The entire program will stop, the error message and the brief usage will be shown to the user.
- ④ If we get here, then everything was OK, and the program will continue as normal.

If we provide a good --val, all is well:

```
$ ./manual.py -v 7
val = "7"
```

If we run this program with a value like 20, we get an error message:

```
$ ./manual.py -v 20
usage: manual.py [-h] [-v int]
manual.py: error: --val "20" must be between 1 and 10
```

It's not possible to tell here, but the parser.error also caused the program to exit with a non-zero status. In the Unix world, an exit status of 0 indicates "zero errors," so anything not 0 is considered an error. You may not realize just yet how wonderful that is, so just trust me. It is.

A.4.8 Automatic help

When you define a program's parameters using argparse, the `-h` and `--help` flags will be reserved for generating help documentation. You do not need to add these nor are you allowed to use these flags for other purposes.

I think of this documentation like a door to your program. Doors are how we get into buildings and cars and such. Have you ever come across a door that you can't figure out how to open? Or one that requires a "PUSH" sign when clearly the handle is design to "pull"? The book *The Design of Everyday Things* by Don Norman uses the term "affordances" to describe the interfaces that objects present to us which do or do not inherently describe how we should use them.

The usage statement of your program is like the handle of the door. It should let me know exactly how to use it. When I encounter a program I've never used, I either run it with no arguments or with `-h` or `--help`. I *expect* to see some sort of usage statement. The only alternative would be to open the source code itself and study how to make the program run and how I can alter it, and this is a truly unacceptable way to write and distribute software!



When you start a new program with `new.py foo.py`, this is the help that will be generated:

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

Rock the Casbah

positional arguments:
  str                  A positional argument

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

Without writing a single line of code, you have

1. an executable Python program
2. that accepts a variety of command line arguments
3. generates a standard and useful help message

This is the "handle" to your program, and you don't have to write a single line of code to get it!

A.5 **Summary**

- Positional parameters typically are always required. If you have more than two or more positional parameters representing different ideas, it would be better to make them into named options.
- Optional parameters can be named like `--file fox.txt` where `fox.txt` is the value for the `--file` option. It is recommended to always define a default value for options.
- `argparse` can enforce many types for arguments including numbers like `int` and `float` or even files.
- Flags like `--help` do not have an associated value. They are considered `True` if present and `False` if not.
- The `-h` and `--help` flags are reserved for use by `argparse`. If you use `argparse`, then your program will automatically respond to these flags with a usage statement.