Playful Python: Learning the language through games and puzzles

Ken Youens-Clark

Figure 1: The Playful Python

Contents

Introduction

> "The only way to learn a new programming language is by writing programs in it." - Dennis Ritchie

I believe you can learn serious things through silly games. I also think you will learn best by *doing*. This is a book of programming exercises. Each chapter includes a description of a program you should write with examples of how the program should work. Most importantly, each program includes tests so that you know if your program is working well enough.

I won't necessarily show you beforehand how to write each program. I'll describe what the program should do and provide some discussion about how to write it. I'll also create an appendix with short examples of how to do things like how to use `argparse`, how to read/write from/to a file, how to process all the files in a directory, how to extract k-mers from a string, etc. I'll provide some building blocks, but I want you to figure out how to put the pieces together.

When you are done with this books you be able to:

- Write command-line Python programs
- Process a variety of command-line arguments, options, and flags
- Write and run tests for your programs and functions
- Manipulate of Python data structures including strings, lists, tuples, sets, dictionaries
- Use higher-order functions like `map` and `filter`
- Write and use regular expressions
- Read, parse, and write various text formats
- Use and control of randomness
- Create and use graphs, kmers, Markov chains, Hamming distance, the Soundex algorithm, and more

Forking GitHub repo

First use the GitHub interface to "fork" this repository into your own account. Then do `git clone` of *your* repository to get a local copy. Inside that checkout, do:

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

This will allow you to `git pull upstream master` in order to get updates. When you create new files, `git add/commit/push` them to *your* repository. (Please do not create pull requests on *my* repository – unless, of course, you have suggestions for improving my repo!).

new.py

I provide some useful programs in the `bin` directory including one called `new.py` that will help you stub out new Python programs using the `argparse` module to parse the command line arguments and options for your programs. I recommend you start every new program with this program. For example, in the `article` directory the `README.md` wants you to create a program called `article.py`. You should do this:

```
$ cd article
$ new.py article
```

This will create a new file called `article.py` (that has been made executable with `chmod +x`, if your operating system supports that) that has example code for you to start writing your program.

$PATH

Your `$PATH` is a list of directories where your operating system will look for programs. To see what your `$PATH` looks like, do:

```
$ echo $PATH
```

Probably each directory is separated by a colon (`:`). *The order of the directories matters!* For instance, it's common to have more than one version of Python installed. When you type `python` on the command line, the directories in your `$PATH` are searched in order, and the first `python` found is the one that is used (and it's probably Python version 2!)

You could execute `new.py` by giving the full path to the program, e.g., `$HOME/work/playful_python/bin/new.py`, but that's really tedious. It's best to put `new.py` into one of the directories that is already in your `$PATH` like maybe `/usr/local/bin`. The problem is that you probably need administrator privileges to write to most of the directories that are in your `$PATH`. If you are working on your laptop, this is probably not a problem, but if you are on a shared system, you probably won't be able to copy the program into your `$PATH` directories.

An alternative is to alter your `$PATH` to include the directory where `new.py` is located. E.g., if `new.py` is in `$HOME/work/playful_python/bin/`, then add this directory to your `$PATH` – probably by editing `.bashrc` or `.bash_profile` located in your `$HOME` directory (if you use `bash`). See the documentation for your shell of choice to understand how to edit and persist your `$PATH`.

For what it's worth, I always create a `$HOME/.local` directory for local installations of software I need, so I add `$HOME/.local/bin` to my `$PATH`. Then I copy programs like `new.py` there and they are available to me anywhere on the system.

Testing your programs

Once you have stubbed out your new program, open it in your favorite editor and change the example arguments in `get_args` to suit the needs of your app, then add your code to `main` to accomplish the task described in the README. To run the test suite using `make`, you can type `make test` in the same directory as the `test.py` and `article.py` program. If your system does not have `make` or you just don't want to use it, type `pytest -v test.py`.

Your goal is to pass all the tests. The tests are written in an order designed to guide you in how break the problem down, e.g., often a test will ask you to alter one bit of text from the command line, and this it will ask you to read and alter the text from a file. I would suggest you solve the tests in order. The `make test` target in every Makefile executes `pytest -xv test.py` where the `-x` flag will have `pytest` halt testing after it finds one that fails. There's no point in running every test when one fails, so I think this is less frustrating that seeing perhaps hundreds of lines of failing tests shoot by.

A fair number of the program rely on a dictionary of English words. To be sure that you can reproduce my results, I include a copy of mine in `inputs/words.zip`.

Why Not Notebooks?

Notebooks are great for interactive exploration of data, especially if you want to visualize thing, but the downsides:

- Stored as JSON not line-oriented text, so no good `diff` tools
- Not easily shared
- Too easy to run cells out of order
- Hard to test
- No way to pass in arguments

I believe you can better learn how to create testable, **reproducible** software by writing command-line programs that always run from beginning to end and have a test suite. It's difficult to achieve that with Notebooks, but I do encourage you to explore Notebooks on your own.

Code examples, the REPL

I always love when a language has a good REPL (read-evaluate-print-loop) tool. Python and Haskell both excel in this respect. For simplicity's sake, I show the standard REPL when you execute `python3` on the command-line, but you won't be able to copy and paste the same code examples there. For your own purposes, I suggest using the iPython REPL (`ipython`) instead.

Author

Ken Youens-Clark is a Sr. Scientific Programmer in the lab of Dr. Bonnie Hurwitz at the University of Arizona. He started college as a music major at the University of North Texas but changed to English lit for his BA in 1995. He started programming at his first job out of college, working through several languages and companies before landing in bioinformatics in 2001. In 2019 he earned his MS in Biosystems Engineering, and enjoys helping people learn programming. When he's not working, he likes playing music, riding bikes, cooking, and being with his wife and children.

Copyright

Outline

I aim to have 40-50 programs complete with specs, examples, inputs, and test suites. They won't necessarily have a specific order, but they will be grouped into easiest/harder/hardest categories. As many programs use common ideas (e.g., regular expressions, graphs, infinite loops), there will be an appendix section with explanations of how to explore those ideas.

I have in mind a layout where each program gets four pages:

```
    1        2                3        4
+--------+--------+      +--------+--------+
|        |        |      |        |        |
|        |        |      |        |        |
|        |        |      |        |        |
| illus/ | specs  |      |solution| notes  |
| info   |        |      |        |        |
|        |        |      |        |        |
|        |        |      |        |        |
+--------+--------+      +--------+--------+
```

1. If a short program, perhaps an illustration; if longer, maybe some background or hints.
2. The `README.md` information (specs, example output)
3. The `solution.py` contents
4. Annotation of the solution with comments on lines, sections

Programs

The goal is to get the reader to become a *writer* – to try to solve the problems. One technique in teaching is to first present a problem without showing how to solve it. Once the student engages with the problem, they find they want and need the object of the lesson. Each program is intended to flex some programming technique or idea like playing with lists or contemplating regular expressions or using dictionaries. By using `argparse` for the programs, we also cover validation of user input.

Easiest

- **article**: Select "a" or "an" depending on the given argument
- **howler**: Uppercase input text so they YELL AT YOU LIKE "HOWLER" MESSAGES IN HARRY POTTER. (Could also be called "OWEN MEANY"?)
- **jump_the_five**: Numeric encryption based on "The Wire."

- **bottles_of_beer**: Produce the "Bottle of Beer on the Wall" song. Explores the basic idea of an algorithm and challenges the programmer to format strings.
- **picnic**: Write the picnic game. Uses input, lists.
- **apples_and_bananas**: Substitute vowels in text, e.g., "bananas" -> "bononos". While the concept is substitution of characters in a string which is actually trivial, it turns out there are many (at least 7) decent ways to accomplish this task!
- **gashlycrumb**: Create a morbid lookup table from text. Naturual use of dictionaries.
- **movie_reader**: Print text character-by-character with pauses like in the movies. How to read text by character, use STDOUT/flush, and pause the program.
- **palindromes**: Find palindromes in text. Reading input, manipulation of strings.
- **ransom_note**: Transform input text into "RaNSom cASe". Manipulation of text.
- **rhymer**: Produce rhyming "words" from input text.
- **rock_paper_scissors**: Write Rock, Paper, Scissors game. Infinite loops, dictionaries.

Harder

- **abuse**: Generate insults from lists of adjectives and nouns. Use of randomness, sampling, and lists.
- **bacronym**: Retrofit words onto acronyms. Use of randomness and dictionaries.
- **blackjack**: Play Blackjack (card game). Use of randomness, combinations, dictionaries.
- **family_tree**: Use GraphViz to visualize a family tree from text. Parsing text, creating graph structures, creating visual output.
- **gematria**: Calculate numeric values of words from characters. Manipulation of text, use of higher-order functions.
- **guess**: Write a number-guessing game. Use of randomness, validation/coercion of inputs, use of exceptions.
- **kentucky_fryer**: Turn text into Southern American English. Parsing, manipulation of text.
- **mad_libs**: TBD
- **markov_words**: Markov chain to generate words. Use of n-grams/k-mers, graphs, randomness, logging.
- **piggie**: Encode text in Pig Latin. Use of regular expressions, text manipulation.
- **sound**: Use Soundex to find rhyming words from a word list.
- **substring**: Write a game to guess words sharing a common substring. Dictionaries, k-mers/n-grams.

- **tictactoe**: Write a Tic-Tac-Toe game. Randomness, state.
- **twelve_days_of_christmas**: Produce the "12 Days of Christmas" song. Algorihtms, loops.
- **war**: Play the War card game. Combinations, randomness.
- **license_plates**: Explore how a regular expression engine works by creating alternate forms of license plates.

Hardest

- **anagram**: Find anagrams of text. Combinations, permutations, dictionaries.
- **hangman**: Write a Hangman (word/letter-guessing game). Randomness, game state, infinite loops, user input, validation.
- **markov_chain**: Markov chain to generate text. N-grams at word level, parsing text, list manipulations.
- **morse**: Write a Morse encoder/decoder. Dictionaries, text manipulation.
- **rot13**: ROT13-encode input text. Lists, encryption.

Chapter 1: Article Selector

Write a Python program called `article.py` that will select `a` or `an` for a given word depending on whether the word starts with a consonant or vowel, respectively.

When run with no arguments or the `-h|--help` flags, it should print a usage statement:

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

Article selector

positional arguments:
  str         Word

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

When run with a single positional argument, it should print the correct article and the given argument.

```
$ ./article.py bear
a bear
$ ./article.py Octopus
an Octopus
```

The tests will only give you words that start with an actual alphabetic character, so you won't have to detect numbers or punctuation or other weird stuff. Still, how might you extend the program to ensure that given argument only starts with one of the 26 characters of the English alphabest?

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), 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`.

- For the purposes of `x in y`, a string (`str`) is a `list` of characters, so you could ask if a character is in a string.
- Use the `print` function to print out the article joined to the argument. Put a single space in between.
- 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.

Solution

```
#!/usr/bin/env python3
"""Article selector"""

import argparse


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

    parser = argparse.ArgumentParser(
        description='Article selector',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

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

    return parser.parse_args()


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

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

    print('{} {}'.format(article, word))

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

Discussion

As with all the solutions presented, this assumes you have stubbed the program with `new.py` and that you are using the `argparse` module. I suggest putting this logic into a separate function which here is called `get_args` and which I like to define first so that I can see right away when I'm reading the program what the program expects as input. On line 12, I set the `description` for the program that will be displayed with the help documentation. On line 15, I indicate that the program expects just one *positional* argument, no more, no less. Since it is a "word" that I expect, I called the argument `word` which is also how I will access the value on line 25. I use the `metavar` on line 15 to let the user know that this should be a string.

The `get_args` function will `return` the result of parsing the command line arguments which I put into the variable `args` on line 24. I can now access the `word` by call `args.word`. Note the lack of parentheses – it's not `args.word()` – as this is not a function call. Think of it like a slot where the value lives.

On line 26, we need to figure out whether the `article` should be `a` or `an`. We'll use a very simple rule that any word that has a first character that is a vowel should get `an` and otherwise we choose `a`. This obviously misses actual pronunciations like in American English we don't pronounce the "h" in "herb" and so actually say "an herb" whereas the British *do* pronounce the "h" and so would say "an herb". (Even more bizarre to me is that the British leave off the article entirely for the word "hospital" as in, "The Queen is in hospital!") Nor will we consider words where the initial `y` acts like a vowel.

We can access the first character of the `word` with `word[0]` which looks the same as how we access the first element of a list. Strings are really list of characters, so this isn't so far-fetched, but we do have to remember that Python, like so many programming languages, starts numbering at `0`, so we often talked about the first element of a list as the "zeroth" element.

To decide if the given word starts with a vowel, we ask is `word[0].lower() in 'aeiou'`. So, to unpack that, `word[0]` returns a one-character-long `str` type which has the method `.lower()` which we call using the parentheses. Without the parens, this would just be the *idea* of the function that returns a lowercased version of the string. Understand that the `word` remains unchanged. The function does not lowercase `word[0]`, it only *returns a lowercase version* of that character.

```
>>> word = 'APPLE'
>>> word
'APPLE'
>>> word[0].lower()
'a'
>>> word
'APPLE'
```

The `x in y` form is a way to ask if element `x` is in the collection `y`:

```
>>> 'a' in 'abc'
True
>>> 'foo' in ['foo', 'bar']
True
>>> 3 in range(5)
True
>>> 10 in range(3)
False
```

The `if` *expression* (also called a "ternary" expression) is different from an `if` *statement*. An *expression* returns a value, and a statement does not. The `if` expression must have an `else`, but the `if` statement does not have this requirement. The first value is returned if the predicate (the bit after the `if`) evaluates to `True` in a Boolean context (cf. "Truthiness"), otherwise the last value is returned:

```
>>> 'Hooray!' if True else 'Shucks!'
'Hooray!'
```

The longer way to write this would have been:

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

Or more succinctly:

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

Cf. appendices: argparse, Truthiness

Chapter 2: Jump the Five

Figure 2: "When I get up, nothing gets me down." - D. L. Roth

Write a program called `jump.py` that will encode any number using "jump-the-five" algorithm that selects as a replacement for a given number one that is opposite on a US telephone pad if you jump over the 5. The numbers 5 and 0 will exchange with each other. So, "1" jumps the 5 to become "9," "6" jumps the 5 to become "4," "5" becomes "0," etc.

```
   1  2  3
   4  5  6
   7  8  9
   #  0  *
```

Print a usage statement for `-h|--help` or if there are no arguments.

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

Your program should replace numbers *anywhere* in the input string:

```
$ ./jump.py 555-1212
```

```
000-9898
$ ./jump.py 'Call 1-800-329-8044 today!'
Call 9-255-781-2566 today!
```

Hints:

- The numbers can occur anywhere in the text, so I recommend you think of how you can process the input character-by-character.
- To me, the most natural way to represent the subsitution table is in a `dict`.
- Read the documentation on Python's `str` class to see what you can do with a string. For instance, there is a `replace` method. Could you use that?

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')

    return parser.parse_args()


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

    args = get_args()
    text = args.text
    jumper = {'1': '9', '2': '8', '3': '7', '4': '6', '5': '0',
              '6': '4', '7': '3', '8': '2', '9': '1', '0': '5'}

    for char in text:
        print(jumper[char] if char in jumper else char, end='')

    print()


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

Discussion

On line 15, we indicate the one positional argument our program expects which is some `text` which we can retrieve on line 25. 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. Later problems will require much more complex arguments, so it's good to get used to this now.

I suggested you could represent the substitution table as a `dict` which is what I create on line 26. Each number `key` has its substitute as the `value` in the `dict`. 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. They are being stored as `str` values, not `int`. This is because we will be reading from a `str`. If we stored them as `int` keys and values, we would have to coerce the `str` types using the `int` function:

```
>>> type('4')
<class 'str'>
>>> type(4)
<class 'int'>
>>> type(int('4'))
<class 'int'>
```

To process the `text` by individual character (`char`), we can use a `for` loop on line 29. Like in the `article` solution, I decided to use an `if` *expression* where I look to see if the `char` is `in` the `jumper` dictionary. In the `article`, you saw we asked if a character was in the string `'aeiou'` (which can also be thought of as a `list` of characters). Here when we ask if a `char` (which is a string) is `in` a `dict`, Python looks to see if there is a **`key`** in the dictionary with that value. So if `char` is `'4'`, then we will print `jumper['4']` which is `'6'`. If the `char` is not in `jumper` (meaning it's not a digit), then we print `char`.

Another way you could have solved this would be to use the `str.translate` method which needs a translation table that you can make with the `str.maketrans` method:

```
>>> s = 'Jenny = 867-5309'
>>> s.translate(str.maketrans(jumper))
'Jenny = 243-0751'
```

Note that you could *not* use `str.replace` to change each number in turn as you would first change `1` to `9` and then you'd get to the `9`s that were in the original string and the `9`s that you changed from `1`s and you'd change them back to `1`s!

Chapter 3: Picnic

Write a Python program called `picnic.py` that accepts one or more positional arguments as the items to bring on a picnic. In response, print "You are bringing …" where "…" should be replaced according to the number of items where:

1. If one item, just state, e.g., if `chips` then "You are bringing chips."
2. If two items, put "and" in between, e.g., if `chips soda` then "You are bringing chips and soda."
3. If three or more items, place commas between all the items INCLUDING BEFORE THE FINAL "and" BECAUSE WE USE THE OXFORD COMMA, e.g., if `chips soda cupcakes` then "You are bringing chips, soda, and cupcakes."

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

Picnic game

positional arguments:
  str         Item(s) to bring

optional arguments:
  -h, --help  show this help message and exit
$ ./picnic.py chips
You are bringing chips.
$ ./picnic.py "potato chips" salad
You are bringing potato chips and salad.
$ ./picnic.py "potato chips" salad soda cupcakes
You are bringing potato chips, salad, soda, and cupcakes.
```

Solution

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

import argparse


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

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

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

    return parser.parse_args()


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

    args = get_args()
    items = args.item
    num = len(items)

    bringing = items[0] if num == 1 else ' and '.join(
        items) if num == 2 else ', '.join(items[:-1] + ['and ' + items[-1]])

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


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

Discussion

This program can accept a variable number of arguments which are all the same thing, so the most appropriate way to represent this with `argparse` is shown on lines 15-19 where we define an `item` agument with `nargs='+'` where `nargs` is the *number of arguments* and `'+'` means *one or more*. Remember, even if the user provides only one argument, you will still get a `list` with just one element.

We put the `items` into a variable on line 28. Note that I call it by the plural `items` because it's probably going to be more than one. Also, I call the variable something informative, not just `args` or something too generic. Lastly, I need to decide how to format the items. As in the article selector, I'm using an `if` *expression* rather than an `if` *statement that would look like this:

```
bringing = ''
if num == 1:
    bringing = items[0]
elif num == 2:
    bringing = ' and '.join(items)
else:
    bringing = ', '.join(items[:-1] + [ 'and ' + items[-1]])
```

But I chose to condense this down into a double `if` expression with the following form:

```
bringing = one_item if num == 1 else two_items if num == 2 else three_items
```

Finally to `print` the output, I'm using a format string where the `{}` indicates a placeholder for some value like so:

```
>>> 'I spy something {}!'.format('blue')
'I spy something blue!'
```

You can also put names inside the `{}` and pass in key/value pairs in any order:

```
>>> 'Give {person} the {thing}!'.format(thing='bread', person='Maggie')
'Give Maggie the bread!'
```

Depending on your version of Python, you may be able to use *f-strings*:

```
>>> color = 'blue'
>>> f'I spy something {color}!'
'I spy something blue!'
```

Chapter 4: Words Count

Write a Python program called `wc.py` that will emulate the venerable `wc` program in Unix that counts the lines, words, and characters in the given file arguments.

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

Argparse Python script

positional arguments:
  FILE        Input file(s)

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

```
$ ./wc.py ../inputs/scarlet.txt
    7035   68061  396320 ../inputs/scarlet.txt
$ ./wc.py ../inputs/*.txt
    1000    1000    5840 ../inputs/1000.txt
     100     100     657 ../inputs/1945-boys.txt
     100     100     684 ../inputs/1945-girls.txt
     872    7652   45119 ../inputs/const.txt
    2476    7436   41743 ../inputs/dickinson.txt
       1       9      45 ../inputs/fox.txt
      25     278    1476 ../inputs/gettysburg.txt
      37      91     499 ../inputs/issa.txt
       9      51     248 ../inputs/nobody.txt
       1      16      65 ../inputs/now.txt
       6      71     413 ../inputs/preamble.txt
    7035   68061  396320 ../inputs/scarlet.txt
      17     118     661 ../inputs/sonnet-29.txt
       3       7      45 ../inputs/spiders.txt
       9      34     192 ../inputs/the-bustle.txt
   37842   48990  369949 ../inputs/uscities.txt
     176    1340    8685 ../inputs/usdeclar.txt
```

Solution

```
#!/usr/bin/env python3
"""Emulate wc (word count)"""

import argparse


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

    parser = argparse.ArgumentParser(
        description='Argparse Python script',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

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

    return parser.parse_args()


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

    args = get_args()

    for fh in args.file:
        lines, words, chars = 0, 0, 0
        for line in fh:
            lines += 1
            chars += len(line)
            words += len(line.split())

        print('{:8}{:8}{:8} {}'.format(lines, words, chars, fh.name))


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

Chapter 5: Howler

Write a Python program `howler.py` that will uppercase all the text from the command line or from a file. The program should also take a named option of `-o|--outfile` to write the output. The default output should be *standard out* (STDOUT).

```
$ ./howler.py
usage: howler.py [-h] [-o str] STR
howler.py: error: the following arguments are required: STR
$ ./howler.py -h
usage: howler.py [-h] [-o str] STR

Howler (upper-case 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: )
$ ./howler.py 'One word: Plastics!'
ONE WORD: PLASTICS!
$ ./howler.py ../inputs/fox.txt
THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.
$ ./howler.py -o out.txt ../inputs/fox.txt
$ cat out.txt
THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.
```

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')

    parser.add_argument('-o',
                        '--outfile',
                        help='Output filename',
                        metavar='str',
                        type=str,
                        default='')

    return parser.parse_args()


# --------------------------------------------------
def main():
    """Make a jazz noise here"""
    args = get_args()
    text = args.text
    out_file = args.outfile

    if os.path.isfile(text):
        text = open(text).read().rstrip()

    out_fh = open(out_file, 'wt') if out_file else sys.stdout
    print(text.upper(), file=out_fh)
    out_fh.close()


```

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

Discussion

Cf. Truthiness, File Handles

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 that you should uppercase or it might be the name of a file. This pattern will come up repeatedly in this book, so commit these lines to memory:

```
if os.path.isfile(text):
    text = open(text).read().rstrip()
```

The first line looks on the file system to see 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. This is usually safe, but be careful if you write a program that could potentially read gigantic files. For instance, in bioinformatics we regularly deal with files with sizes in the 10s to 100s of gigabytes!

The result of `open(file).read()` is a `str` which itself has a *method* called `rstrip` that will return a copy of the string *stripped* of the whitespace off the *right* side of the string. The longer way to write the above would be:

```
if os.path.isfile(text):
    fh = open(text)
    text = fh.read()
    text = text.rstrip()
```

On line 39, we decide where to put the output of our program. The `if` expression will open `out_file` for writing text if `out_file` has been defined. The default value for `out_file` is the empty string which is effectively `False` when evaluated in a Boolean content. Unless the user provides a value, the output file handle `out_fh` will be `sys.stdout`.

To get uppercase, we can use the `text.upper` method. You can either `out_fh.write` this new text or use `print(..., file=...)`, noting which needs a newline and which does not. You can use `fh.close()` to close the file handle, but it's not entirely necessary as the program immediately ends after this. Still, it's good practice to close your file handles.

Chapter 6: Apples and Bananas

Perhaps you remember the children's song "Apples and Bananas"?

```
I like to eat, eat, eat apples and bananas
I like to eat, eat, eat apples and bananas

I like to ate, ate, ate ay-ples and ba-nay-nays
I like to ate, ate, ate ay-ples and ba-nay-nays

I like to eat, eat, eat ee-ples and bee-nee-nees
I like to eat, eat, eat ee-ples and bee-nee-nees
```

Figure 3: Apple and bananas go together like peas and carrots.

Write a Python program called `apples.py` that will turn all the vowels in some given text in a single positional argument into just one `-v|--vowel` (default `a`) like this song.

Replace all vowels with the given vowel, both lower- and uppercase.

If the program is run with no arguments or the `-h|--help` flags, print a usage statement:

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

```
Apples and bananas

positional arguments:
  str                   Input text or file

optional arguments:
  -h, --help            show this help message and exit
  -v str, --vowel str   The only vowel allowed (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')
```

The program should handle text on the command line:

```
$ ./apples.py foo
faa
$ ./apples.py foo -v i
fii
```

If the given text argument is a file, read the text from the file:

```
$ ./apples.py ../inputs/fox.txt
Tha qaack brawn fax jamps avar tha lazy dag.
$ ./apples.py --vowel u ../inputs/fox.txt
Thu quuck bruwn fux jumps uvur thu luzy dug.
```

Hints:

- See `choices` in the `argparse` documentation for how to constrain the `--vowel` options

Solution

```
#!/usr/bin/env python3
"""Apples and Bananas"""

import argparse
import os
import re


# --------------------------------------------------
def get_args():
    """get command-line arguments"""

    parser = argparse.ArgumentParser(
        description='Apples and bananas',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('text', metavar='str', help='Input text or file')

    parser.add_argument('-v',
                        '--vowel',
                        help='The vowel(s) allowed',
                        metavar='str',
                        type=str,
                        default='a',
                        choices=list('aeiou'))

    return parser.parse_args()


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

    args = get_args()
    text = args.text
    vowel = args.vowel

    if os.path.isfile(text):
        text = open(text).read()

    # Method 1: Iterate every character
    # new_text = []
    # for char in text:
```

```
    #     if char in 'aeiou':
    #         new_text.append(vowel)
    #     elif char in 'AEIOU':
    #         new_text.append(vowel.upper())
    #     else:
    #         new_text.append(char)
    # text = ''.join(new_text)

    # Method 2: str.replace
    # for v in 'aeiou':
    #     text = text.replace(v, vowel).replace(v.upper(), vowel.upper())

    # Method 3: str.translate
    # trans = str.maketrans('aeiouAEIOU', vowel * 5 + vowel.upper() * 5)
    # text = text.translate(trans)

    # Method 4: Use a list comprehension
    # new_text = [
    #     vowel if c in 'aeiou' else vowel.upper() if c in 'AEIOU' else c
    #     for c in text
    # ]
    # text = ''.join(new_text)

    # Method 5: Define a function, use list comprehension
    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])

    # Method 6: Use a `map` to iterate with a `lambda`
    # text = ''.join(
    #     map(
    #         lambda c: vowel if c in 'aeiou' else vowel.upper()
    #         if c in 'AEIOU' else c, text))

    # Method 7: `map` with the function
    text = ''.join(map(new_char, text))

    # Method 8: Regular expressions
    # text = re.sub('[aeiou]', vowel, text)
    # text = re.sub('[AEIOU]', vowel.upper(), text)

    print(text.rstrip())


# --------------------------------------------------
```

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

Discussion

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. Once again, I defer to `argparse` by defining the `text` positional argument and the `-v|--vowel` option with a default value of `'a'`. I additionally use the `choices` option to restrict the values to the `list('aeiou')`. Remember that calling `list` on a string will expand it into a `list` of characters:

```
>>> list('aeiou')
['a', 'e', 'i', 'o', 'u']
```

The next problem is detecting if `text` is the name of a file that should be read for the text or is the text itself. I use `os.path.isfile` to ask the operating system if `text` names a file on disk. If this returns `True`, then I use `open(text).read()` to open the file and read the entire contents of the opened file handle into the `text` variable.

Method 1: Iterate every character

You can use a `for` loop on a string to access each character:

```
>>> text = 'Apples and Bananas!'
>>> vowel = 'o'
>>> new_text = []
>>> for char in text:
...     if char in 'aeiou':
...         new_text.append(vowel)
...     elif char in 'AEIOU':
...         new_text.append(vowel.upper())
...     else:
...         new_text.append(char)
...
>>> text = ''.join(new_text)
>>> text
'Opplos ond Bononos!'
```

So we get each `char` (character) in the `text` and ask if the character is in the string `'aieou` to determine if it is a vowel. If it is, we instead use the `vowel` determined by the user. Likewise with checking for membership in `'AEIOU'` to see if it's an uppercase vowel and using the `vowel.uppper()`. If neither of those conditions is true, then we stick with the original character. Finally we overwrite `text` by joining the `new_text` on the empty string to make a new string with the vowels replaced.

Method 2: str.replace

The `str` class has a `replace` method that will return a new string with all instances of one string replaced by another. Note that the original string remains unchanged:

```
>>> s = 'foo'
>>> s.replace('o', 'a')
'faa'
>>> s.replace('oo', 'x')
'fx'
>>> s
'foo'
```

In this version:

```
>>> text = 'Apples and Bananas!'
>>> for v in 'aeiou':
...     text = text.replace(v, vowel).replace(v.upper(), vowel.upper())
...
>>> text
'Opplos ond Bononos!'
```

We use a `for` loop to iterate over each vowel in `'aeiou'` and then call `text.replace` to change that character to the indicated `vowel` from the user using both lower- and uppercase. If the character is not present, no action is taken.

Method 3: str.translate

There is a `str` method called `translate` that is very similar to `replace` that will "replace each character in the string using the given translation table." To create the translation table, you should call the `str.maketrans` method. I pass it the string of lower- and upper-case vowels (5 of each) and a string that has position-by-position what should be substituted which I create by concatenating the lowercase `vowel` repeated 5 times with the uppercase `vowel` repeated 5 times.

```
>>> vowel * 5
'ooooo'
>>> vowel * 5 + vowel.upper() * 5
'oooooOOOOO'
>>> trans = str.maketrans('aeiouAEIOU', vowel * 5 + vowel.upper() * 5)
```

The `trans` table is a `dict` where each character is represented by it's ordinal value. You can go back and forth from characters and their ordinal values by using `chr` and `ord`:

```
>>> chr(97)
```

```
'a'
>>> ord('a')
97
```

If you look at the `trans` table:

```
>>> 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}
```

you can see it's mapping all the lowercase vowels to the ordinal value `111` which is 'o' and the uppercase vowels to 79 which is 'O':

```
>>> chr(111)
'o'
>>> chr(79)
'O'
```

And so I hope you can see how this works now. Recall that the original `text` remains unchanged by the `translate` method, so we overwrite `text` with the new version:

```
>>> text = 'Apples and Bananas!'
>>> trans = str.maketrans('aeiouAEIOU', vowel * 5 + vowel.upper() * 5)
>>> text = text.translate(trans)
>>> text
'Opplos ond Bononos!'
```

Method 4: List comprehension

You can stick a modified `for` loop inside brackets `[]` to create what is called a "list comprehension" to create new list from an existing sequence (list/dict/generator/stream) in one line of code. (You can also do likewise with `{}` for a new `dict`.) For example, here is how you could generate a list of squared numbers:

```
>>> [n ** 2 for n in range(4)]
[0, 1, 4, 9]
```

Additionally, inside the list comprehension we can use an `if` *expression*. Let's say you wanted `list` of `tuple`s with a value and a string declaring if the value is "Even" or "Odd". The typical way to determine even/odd is looking at the remainder after dividing by 2 which we can do with the `modulo` (`%`) operator:

```
>>> 4 % 2
0
>>> 5 % 2
1
```

We can use Python's idea of "truthiness" to evaluate `0` as `False` and anything not `0` as `True`:

```
>>> 'Odd' if 4 % 2 else 'Even'
'Even'
>>> 'Odd' if 5 % 2 else 'Even'
'Odd'
```

Then use that inside a list comprehension:

```
>>> [(n, 'Odd' if n % 2 else 'Even') for n in range(4)]
[(0, 'Even'), (1, 'Odd'), (2, 'Even'), (3, 'Odd')]
```

We can chain `if` expressions to handle more than a binary decision. Perhaps you are programming an autonomous vehicle and want to decide how what to do at a traffic signal?

```
>>> color = 'red'
>>> 'STOP' if color == 'red' else 'Slow' if color == 'yellow' else 'Go'
'STOP'
>>> color = 'green'
>>> 'STOP' if color == 'red' else 'Slow' if color == 'yellow' else 'Go'
'Go'
```

In this version:

```
>>> text = 'Apples and Bananas!'
>>> new_text = [
...     vowel if c in 'aeiou' else vowel.upper() if c in 'AEIOU' else c
...     for c in text
... ]
>>> text = ''.join(new_text)
>>> text
'Opplos ond Bononos!'
```

You have to find the start of the `for` loop `for c in text` which is "for character in text." We then use our handy compound `if` *expression* to decide whether to return the chosen `vowel if c in 'aeiou'` or the same check with the uppercase version, and finally we default to the character `c` itself if it fails both of those conditions.

Method 5: List comprehension with function

We could define a small function that will decide whether to return the `vowel` or the original character:

```
>>> vowel = 'o'
>>> def new_char(c):
...     return vowel if c in 'aeiou' else vowel.upper() if c in 'AEIOU' else c
...
>>> new_char('a')
'o'
>>> new_char('b')
'b'
```

And then use our list comprehension to call that. To me, this code is far more readable:

```
>>> 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. Here 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`. 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. I could pass it as another argument, but the closure makes all this very compact and readable.

Method 6: `map` with a `lambda`

A `map` is essentially another way to write a list comprehension. Functions like `map` and another we'll use later called `filter` are in the class of "higher-order functions" because they take *other functions* as arguments, which is wicked cool. `map` applies another function to every member of a sequence. I like to think of `map` like a paint booth: You load up the booth with, say, blue paint, then unpainted cars go in, blue paint is applied, and blue cars come out.

I tend to think of this left-to-right:

```
car1, car2 -> paint_blue -> blue car1, blue car2
```

But the calling syntax moves right-to-left:

```
>>> paint_blue = lambda car: 'blue ' + car
>>> list(map(paint, ['car1', 'car2']))
['blue car1', 'blue car2']
```

Often you'll see the first argument to `map` starting with `lambda` to create an anonymous function using the `lambda` keyword. Think about regular named functions like `add1` that adds `1` to a value:

```
>>> def add1(n):
...     return n + 1
...
>>> add1(10)
11
>>> add1(11)
12
```

Here is the same idea using a `lambda`. Notice the function pretty much needs to fit on one line, can't really unpack complicated arguments, and doesn't need `return`:

```
>>> add1 = lambda n: n + 1
>>> add1(10)
11
>>> add1(11)
12
```

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 we just define the argument `n`. There is no difference in how you can use them. They are both functions:

```
>>> type(lambda x: x)
<class 'function'>
```

So I could define the `new_char` function using a `lambda` and it works just like the one created with `def new_char`:

```
>>> new_char = lambda c: vowel if c in 'aeiou' else \
...     vowel.upper() if c in 'AEIOU' else c
>>> new_char('a')
'o'
>>> new_char('b')
'b'
```

And here is how I can use it with `map`:

```
>>> text = 'Apples and Bananas!'
>>> text = ''.join(
...     map(
...         lambda c: vowel if c in 'aeiou' else vowel.upper()
...         if c in 'AEIOU' else c, text))
>>>
>>> text
'Opplos ond Bononos!'
```

Method 7: `map` with `new_char`

The previous version is not exactly easy to read, in my opinions, so instead of using `lambda` to make a function *inside* the `map`, I can use the `def new_char` version from above and `map` into that. In my opinion, this is the cleanest and most readable solution:

```
>>> text = 'Apples and Bananas!'
>>> text = ''.join(map(new_char, text))
>>> text
'Opplos ond Bononos!'
```

Notice that `map` takes `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'
```

What happens is that `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 `join` on the empty string to create a new version of `text`.

Method 8: Regular expressions

The last method I will introduce uses regular expressions which are a separate domain-specific language (DSL) you can use to describe patterns of text. They are incredibly powerful and well worth the effort to learn them. To use them in your program, you `import re` and then use methods like `search` to find a pattern in a string or here `sub` to substitute a pattern for a new string. We'll be using brackets `[]` to create a "character class" meaning anything matching one of these characters. The second argument is the string that will replace the found strings, and the third argument is the string on which to work. Note that this string remains unchanged by the operation:

```
>>> import re
>>> text = 'Apples and Bananas!'
>>> vowel = 'o'
>>> re.sub('[aeiou]', vowel, text)
'Applos ond Bononos!'
>>> text
'Apples and Bananas!'
```

That almost worked, but it missed the uppercase vowel "A". I could overwrite 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:

```
>>> text = 'Apples and Bananas!'
>>> text = re.sub('[AEIOU]', vowel.upper(), re.sub('[aeiou]', vowel, text))
>>> text
'Opplos ond Bononos!'
```

But I find that fairly hard to read.

Chapter 7: Telephone

Perhaps you remember the game of "Telephone" where a message is secretly passed through a series of intermediaries and then the result at the end of the chain is compared with how it started? This is like that, only we're going to take some `text` (from the command line or a file) and mutate it by some percentage `-m|--mutations` (a number between 0 and 1, default `0.1` or 10%) and then print out the resulting text.

Each mutation to the text should be chosen using the `random` module, so your program will also need to accept a `-s|--seed` option (default `None`) to pass to the `random.seed` function for testing purposes. Print the resulting text after making the appropriate number of mutations.

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

Telephone

positional arguments:
  str                   Input text or file

optional arguments:
  -h, --help            show this help message and exit
  -s str, --seed str    Random seed (default: None)
  -m float, --mutations float
                        Percent mutations (default: 0.1)
```

The program should not accept a bad `--mutations` argument:

```
$ ./telephone.py -m 10 foo
usage: telephone.py [-h] [-s str] [-m float] str
telephone.py: error: --mutations "10.0" must be b/w 0 and 1
```

It can be interesting to watch the accumulation of mutations:

```
$ ./telephone.py -s 1 ../inputs/fox.txt
Tho quick brown foa jumps oWer*the lazy dog.
$ ./telephone.py -s 1 -m .5 ../inputs/fox.txt
Thakqkrck&brow- fo[ jumps#oWe,*L/C lxdy dogos
```

Hints:

- To create a combined error/usage statement for the `--mutations` error, look at `parser.error` in `argparse`.

- To select a character position to change, I suggest using `random.choice` and a `range` from length of the incoming text. With that, you'll need to alter the character at that position, but you'll find that strings in Python are *immutable*. For instance, if I wanted to change "candle" into "handle":

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

- So, I need to create a *new string* that has `h` joined to the rest of the string `s` after the zeroth position. How could you do that?
- For the replacement value, you should use `random.choice` from the union of the `string` class's `ascii_letters` and `punctuation`:

```
>>> import string
>>> string.ascii_letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

Solution

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

import argparse
import os
import random
import string
import sys


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

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

    parser.add_argument('input',
                        metavar='str',
                        help='Input text or file')

    parser.add_argument('-s',
                        '--seed',
                        help='Random seed',
                        metavar='str',
                        type=str,
                        default=None)

    parser.add_argument('-m',
                        '--mutations',
                        help='Percent mutations',
                        metavar='float',
                        type=float,
                        default=0.1)

    args = parser.parse_args()

    if not 0 < args.mutations <= 1:
        msg = '--mutations "{}" must be b/w 0 and 1'.format(args.mutations)
        parser.error(msg)

    return args
```

```


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

    args = get_args()
    text = args.input
    random.seed(args.seed)

    if os.path.isfile(text):
        text = open(text).read()

    len_text = len(text)
    num_mutations = int(args.mutations * len_text)
    alpha = string.ascii_letters + string.punctuation

    for _ in range(num_mutations):
        i = random.choice(range(len_text))
        text = text[:i] + random.choice(alpha) + text[i+1:]

    print(text.rstrip())

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

Discussion

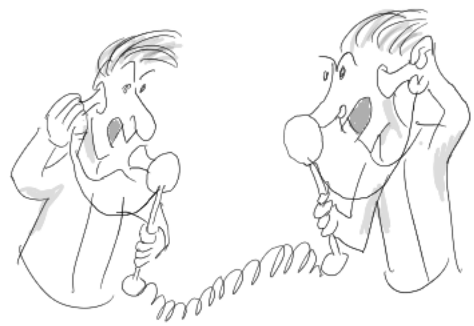


Figure 4: Telephones are for communication.

The number of mutations will be proportional to the length of the text

```
>>> text = 'The quick brown fox jumps over the lazy dog.'
>>> len_text = len(text)
>>> len_text
44
```

Since we chose the `--mutations` to be a `float` between 0 and 1, we can multiply that by the length to get the number of mutations to introduce. Since that number will likely be another `float` and we can introduce a partial number of mutations, we can use `int` to truncate the number to an integer value.

```
>>> mutations = .1
>>> int(mutations * len_text)
4
```

So we can use that number in a `for` loop with `range(4)` to modify four characters. To choose a character in the text to modify, I suggested to use `random.choice`:

```
>>> import random
>>> random.choice(range(len_text))
1
>>> random.choice(range(len_text))
22
```

If you assign that to a value like `i` (for "integer" and/or "index", it's pretty common to use `i` for this kind of value), then you could get the character at that position:

```
>>> i = random.choice(range(len_text))
```

```
>>> i
4
>>> text[i]
'q'
```

Now we saw earlier that we can't just change the `text`:

```
>>> text[i] = 'x'
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.

The bit before `i` is:

```
>>> text[:i]
'The '
```

And after `i` (skipping `i` itself, of course):

```
>>> text[i+1:]
'uick brown fox jumps over the lazy dog.'
```

There are many ways to join strings together into new strings, and the `+` operator is perhaps the simplest. So now we need some new character to insert in the middle which we can get with `random.choice` again, this time choosing from all the letters of the alphabet plus punctuation:

```
>>> import string
>>> alpha = string.ascii_letters + string.punctuation
>>> random.choice(alpha)
'n'
```

So to put it together, we overwrite the existing `text` so as to accumulate the changes over the iterations:

```
>>> text = text[:i] + random.choice(alpha) + text[i+1:]
>>> text
'The vuick brown fox jumps over the lazy dog.'
```

Mutations in DNA

For what it's worth, this is (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 are called them “in-dels” (insertion-deletions) in biology.

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! We can revisit the output of this program later by using the Hamming distance to find how many changes we’d need to make to the output to regain the input.

Chapter 8: Bottles of Beer Song

Write a Python program called `bottles.py` that takes a single option `-n|--num` which is an positive integer (default `10`) and prints the " bottles of beer on the wall song." The program should also respond to `-h|--help` with a usage statement:

```
$ ./bottles.py -h
usage: bottles.py [-h] [-n INT]

Bottles of beer song

optional arguments:
  -h, --help         show this help message and exit
  -n INT, --num INT  How many bottles (default: 10)
```

If the `--num` argument is not an integer value, print an error message and stop the program:

```
$ ./bottles.py -n foo
usage: bottles.py [-h] [-n INT]
bottles.py: error: argument -n/--num: invalid int value: 'foo'
$ ./bottles.py -n 2.4
usage: bottles.py [-h] [-n INT]
bottles.py: error: argument -n/--num: invalid int value: '2.4'
```

If the `-n` argument is less than 1, die with '–num () must be > 0'.

```
$ ./bottles.py -n -1
usage: bottles.py [-h] [-n INT]
bottles.py: error: --num (-1) must > 0
```

If the argument is good, then print the appropriate number of verses:

```
$ ./bottles.py -n 1
1 bottle of beer on the wall,
1 bottle of beer,
Take one down, pass it around,
0 bottles of beer on the wall!

$ ./bottles.py | head
10 bottles of beer on the wall,
10 bottles of beer,
Take one down, pass it around,
9 bottles of beer on the wall!

9 bottles of beer on the wall,
9 bottles of beer,
Take one down, pass it around,
```

```
8 bottles of beer on the wall!
```

Hints:

- Start with `new.py` and add a named *option* with `-n` for the "short" flag and `--num_bottles` for the "long" flag name. Be sure to choose `int` for the `type`. Note that the `metavar` is just for displaying to the user and has no effect on validation the arguments `type`.
- Look into `parser.error` for how to get `argparse` to printing an error message along with the usage and halt the program.
- Be sure to make the "bottle" into the proper singular or plural depending on the number in the phrase, e.g., "1 bottle" or "0 bottles."
- Either run your program or do `make test` after *every single change to your program* to ensure that it compiles and is getting closer to passing the tests. Do not change three things and then run it. Make one change, then run or test it.
- If you use `make test`, it runs `pytest -xv test.py` where the `-x` flag tells `pytest` to stop after the first test failure. The tests are written in a order to help you complete the program. For instance, the first test just ensures that the program exists. The next one that you have some sort of handling of `--help` which would probably indicate that you're using `argparse` and so have defined your arguments.
- Just try to pass each test in order. Focus on just one thing at a time. Create the program. Add the help. Handle bad arguments. Print just one verse. Print two verses. Etc.
- Read the next section on how to count down.

Counting down

You are going to need to count down, so you'll need to consider how to do that. You can use `range` to get a list of integers from some a "start" (default `0`, inclusive) to an "stop" (not inclusive). The `range` function is "lazy" in that it won't actually generate the list until you ask for the numbers, so I could create a `range` generator for an absurdly large number like `range(10**1000)` and the REPL returns immediately. Try it! To force *see* the list of numbers, I can coerce it into a `list`:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

OK, so maybe you were expecting the numbers 1-10? Welcome to "computer science" where we often starting counting at `0` and are quite often "off-by-one." To count 1 to 10, I have to do this:

```
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Cool, cool, but we actually need to count *down*. You saw that this function works differently depending on whether you give it one argument (`10`) or two (`1, 11`). It also will do something different if you give it a third argument that represents the "step" of the numbers. So, to list every other number:

```
>>> list(range(1, 11, 2))
[1, 3, 5, 7, 9]
```

And to count *down*, reverse the start and stop and use `-1` for the step:

```
>>> list(range(11, 1, -1))
[11, 10, 9, 8, 7, 6, 5, 4, 3, 2]
```

Wait, what? OK, the start number is inclusive and the stop is not. Try again:

```
>>> list(range(10, 0, -1))
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

There's a slightly easier way to get that list by using the `reversed` function:

```
>>> list(reversed(range(1, 11)))
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Solution

```
#!/usr/bin/env python3
"""Bottle 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',
                        '--num',
                        metavar='INT',
                        type=int,
                        default=10,
                        help='How many bottles')

    args = parser.parse_args()

    if args.num < 1:
        parser.error('--num ({}) must > 0'.format(args.num))

    return args


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

    args = get_args()
    tmpl = '\n'.join([
        '{} bottle{} of beer on the wall,',
        '{} bottle{} of beer,',
        'Take one down, pass it around,',
        '{} bottle{} of beer on the wall!',
    ])

    for bottle in reversed(range(1, args.num + 1)):
        next_bottle = bottle - 1
```

```
        s1 = '' if bottle == 1 else 's'
        s2 = '' if next_bottle == 1 else 's'
        print(tmpl.format(bottle, s1, bottle, s1, next_bottle, s2))
        if bottle > 1:
            print()


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

Discussion

Figure 5: "To alcohol! The cause of, and solution to, all of life's problems." - H. Simpson

If you used `new.py` and `argparse` to get started, then about 1/4 of the program is done for you. If you define an argument with the appropriate "short" (a dash plus one character) and "long" names (two dashes and a longer bit) with `type=int` and `default=10`, then `argparse` will do loads of hard work to ensure the user provides you with the correct input. We can't easily tell `argparse` that the number has to be a *positive* integer without defining a new "type", but it's fairly painless to add a check and use `parser.error` to both print an error message plus the usage and halt the execution of the program.

Earlier programs have the last line of `get_args` as:

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

But here we capture the arguments inside `get_args` and add a bit of validation. If `args.num_bottles` is less than one, we call `parser.error` with the message we want to tell the user. We don't have to tell the program to stop executing as `argparse` will exit immediately. Even better is that it will indicate a non-zero exit value to the operating system to indicate there was some sort of error. If you ever start writing command-line programs that chain together to make workflows, this is a way for one program to indicate failure and halt the entire process until the error has been fixed!

Once you get to the line `args = get_args()` in `main`, a great deal of hard work has already occurred to get and validate the input from the user. From here, I decided to create a template for the song putting `{}` in the spots that change from verse to verse. Then I use the `reversed(range(...))` bit we discussed before to count down, with a `for` loop, using the current number `bottle` and `next_bottle` to print out the verse noting the presence or absence of the `s` where appropriate.

I'd like to stress that there are literally hundreds of ways to solve this problem. The website http://www.99-bottles-of-beer.net/ claims to have 1500 variations in various languages, 15 in Python alone. As always, the solution you wrote and understand and that passes the test suite is the "right" solution.

Chapter 9: Gashlycrumb

Write a Python program called `gashlycrumb.py` that takes a letter of the alphabet as an argument and looks up the line in a `-f|--file` argument (default `gashlycrumb.txt`) and prints the line starting with that letter. It should generate usage with no arguments or for `-h|--help`:

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

Gashlycrumb

positional arguments:
  str                 Letter

optional arguments:
  -h, --help          show this help message and exit
  -f str, --file str  Input file (default: gashlycrumb.txt)
```

You can see the structure of the default "gashlycrumb.txt" file:

```
$ head -3 gashlycrumb.txt
A is for Amy who fell down the stairs.
B is for Basil assaulted by bears.
C is for Clara who wasted away.
```

Figure 6: D is for Donald, who died from gas.

You will use the first character of the line as a lookup value:

```
$ ./gashlycrumb.py a
```

```
A is for Amy who fell down the stairs.
$ ./gashlycrumb.py z
Z is for Zillah who drank too much gin.
```

If given a value that does not exist in the list of first characters on the lines from the input file (when searched with 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 provided a `--file` argument that does not exist, your program should exit with an error and message:

```
$ ./gashlycrumb.py -f sdfl b
usage: gashlycrumb.py [-h] [-f str] str
gashlycrumb.py: error: argument -f/--file: can't open 'sdfl': \
[Errno 2] No such file or directory: 'sdfl'
```

Hints:

- To validate that the `--filename` is actually a readable file, look into using `argparse.FileType('r')` to describe the `type` of the `--file` argument so that `argparse` will do the check and create the error.
- 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."
- Once you have an open file handle to the `--filename` (which is exactly what you get when use `argparse.FileType`), 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?
- Once you have constructed the dictionary of letters to lines, how can you check that the user's `letter` argument is `in` the dictionary?
- Can you solve this without a `dict`?

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', metavar='str', type=str)

    parser.add_argument('-f',
                        '--file',
                        help='Input file',
                        metavar='str',
                        type=argparse.FileType('r'),
                        default='gashlycrumb.txt')

    return parser.parse_args()


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

    args = get_args()
    letter = args.letter

    # lookup = {}
    # for line in args.file:
    #     lookup[line[0]] = line.rstrip()

    lookup = {line[0]: line.rstrip() for line in args.file}

    if letter.upper() in lookup:
        print(lookup[letter.upper()])
    else:
        print('I do not know "{}".'.format(letter))
```

```


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

Discussion

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 argument, 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 a valid, open file handle in the `args.file` slot. In the REPL, I can manually do what `argparse` has done by using `open` to get a file handle which I like to usually call `fh`:

```
>>> fh = open('gashlycrumb.txt')
```

I can use a `for` loop to read each line of text and get the first letter using `line[0]` and set a `dict` called `lookup` with the value for the `line`:

```
>>> lookup = {}
>>> for line in fh:
...     lookup[line[0]] = line.rstrip()
...
>>> from pprint import pprint as pp
>>> pp(lookup)
{'A': 'A is for Amy who fell down the stairs.',
 'B': 'B is for Basil assaulted by bears.',
 'C': 'C is for Clara who wasted away.',
 'D': 'D is for Desmond thrown out of a sleigh.',
 'E': 'E is for Ernest who choked on a peach.',
 'F': 'F is for Fanny sucked dry by a leech.',
 'G': 'G is for George smothered under a rug.',
 'H': 'H is for Hector done in by a thug.',
 'I': 'I is for Ida who drowned in a lake.',
 'J': 'J is for James who took lye by mistake.',
 'K': 'K is for Kate who was struck with an axe.',
 'L': 'L is for Leo who choked on some tacks.',
 'M': 'M is for Maud who was swept out to sea.',
 'N': 'N is for Neville who died of ennui.',
 'O': 'O is for Olive run through with an awl.',
 'P': 'P is for Prue trampled flat in a brawl.',
 'Q': 'Q is for Quentin who sank on a mire.',
 'R': 'R is for Rhoda consumed by a fire.',
 'S': 'S is for Susan who perished of fits.',
 'T': 'T is for Titus who flew into bits.',
 'U': 'U is for Una who slipped down a drain.',
 'V': 'V is for Victor squashed under a train.',
```

```
 'W': 'W is for Winnie embedded in ice.',
 'X': 'X is for Xerxes devoured by mice.',
 'Y': 'Y is for Yorick whose head was bashed in.',
 'Z': 'Z is for Zillah who drank too much gin.'}
```

We've seen list comprehensions by essentially sticking a `for` inside brackets `[]`, and we can use a dictionary comprehension by doing the same with a `for` loop inside curlies `{}`. If you are following along by pasting code into the REPL, note that we have exhausted the file handle `fh` just above by reading it. I need to `open` it again for this next bit:

```
>>> fh = open('gashlycrumb.txt')
>>> lookup = {line[0]: line.rstrip() for line in fh}
```

If you `pprint` it 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).

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 just 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".
```

I don't have to use a `dict`. I could, for example, use a `list` of `tuple` values:

```
>>> fh = open('gashlycrumb.txt')
>>> lookup = [(line[0], line.rstrip()) for line in fh]
>>> pp(lookup[:2])
[('A', 'A is for Amy who fell down the stairs.'),
 ('B', 'B is for Basil assaulted by bears.')]
```

I can get the letters with a list comprehension:

```
>>> [char for char, line in lookup][:3]
['A', 'B', 'C']
```

And then use `in` to see if my `letter` is present:

```
>>> letter = 'a'
>>> letter.upper() in [char for char, line in lookup]
True
```

And get the value like so:

```
>>> [line for char, line in lookup if char == letter.upper()]
['A is for Amy who fell down the stairs.']
```

The problem is that the cost of the search is proportional to the number of values. That is, if we were searching a million keys in a list, then Python starts searching at the beginning of the list and goes until it finds the value. When you store items in a `dict`, the search time for a key can be much shorter, often nearly instantaneous. It's well worth your time to learn dictionaries very well!

Edward Gorey

If you are not familiar with the work of Edward Gorey, please go read about him immediately, e.g. https://www.brainpickings.org/2011/01/19/edward-gorey-the-gashlycrumb-tinies/!

Alternate text

Write your own version of Gorey's text and pass in your version as the `--file`. I include my own `alternate.txt` which I used the simple and Soundex rhymers to help me find words.

Interactive version

Write an interactive version that takes input directly from the user.

```
$ ./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
```

Hint: Use `while True` to set up an infinite loop and keep using `input` to get the user's next `letter`.

Chapter 10: Ransom

Figure 7: A ransom note.

Create a Python program called `ransom.py` that will randomly capitalize the letters in a text. The program should take a `-s|--seed` argument for the `random.seed` to control randomness for the test suite. It should print usage when given no arguments or `-h|--help`.

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

Ransom Note

positional arguments:
  str                 Input text or file

optional arguments:
  -h, --help          show this help message and exit
  -s int, --seed int  Random seed (default: None)
```

The text can be given on the command line:

```
$ ./ransom.py -s 2 'The quick brown fox jumps over the lazy dog.'
the qUIck BROWN fOX JUmps ovEr ThE LAZY DOg.
```

Or in a file:

```
$ cat ../inputs/fox.txt
The quick brown fox jumps over the lazy dog.
$ ./ransom.py --seed 2 ../inputs/fox.txt
```

```
the qUIck BROWN fOX JUmps ovEr ThE LAZY DOg.
```

Hints:

- You can iterate each character in the input string with a `for` loop
- For each character, can use the `random.choice` function to decide whether to force the character to upper or lower case using methods from the `str` class

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='str', help='Input text or file')

    parser.add_argument('-s',
                        '--seed',
                        help='Random seed',
                        metavar='int',
                        type=int,
                        default=None)

    args = parser.parse_args()

    if os.path.isfile(args.text):
        args.text = open(args.text).read().rstrip()

    return args

# --------------------------------------------------
def choose(c):
    """Randomly choose an upper or lowercase letter to return"""

    return c.upper() if random.choice([0, 1]) else c.lower()

# --------------------------------------------------
def main():
    """Make a jazz noise here"""
    args = get_args()
    text = args.text
```

```
    random.seed(args.seed)

    # Method 1: Iterate each character, add to list
    # ransom = []
    # for char in text:
    #     ransom.append(char.upper() if random.choice([0, 1]) else char.lower())

    # Method 2: List comprehension
    #ransom = [c.upper() if random.choice([0, 1]) else c.lower() for c in text]

    # Method 3: List comprehension with function
    #ransom = [choose(c) for c in text]

    # Method 4: map with lambda
    # ransom = map(lambda c: c.upper() if random.choice([0, 1]) else c.lower(),
    #              text)

    # Method 5: map with function
    ransom = map(choose, text)

    print(''.join(ransom))


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

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?

It's a common pattern in many of these problems that the input can either be given on the command line or in a file, so I have to defined the `text` argument as having `type=str`. In this version of the program, I decided to check in the `get_args` if the `text` is a file (`os.path.isfile(text)`), and, if so, to override the value of `args.text` with the result of reading the contents of the file. That way when I get to the `args = get_args()` line in my program, I've already gotten the text from the user, whether given on the command line or in a file.

I set the `--seed` optional `default` to Python's special `None` value which means nothing at all. As such, I can pass it directly to `random.seed` because setting the seed to `None` is the same as not setting it. Only if the user indicates a `--seed` value (which must be an `int` and which `argparse` will validate) will this affect the behavior of the program.

Assume that we have the following:

```
>>> text = 'The quick brown fox jumps over the lazy dog.'
```

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. Here's one way to print an uppercase version of the `text`

```
>>> for char in text:
...     print(char.upper(), end='')
...
THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.
```

Let's use `random.choice` to make a binary selection:

```
>>> import random
>>> random.choice([True, False])
False
>>> random.choice([0, 1])
0
>>> random.choice(['blue', 'green'])
'blue'
```

Now use that to select whether to take the upper- or lowercase character. Note that this version relies on the idea of "truthiness" (cf appendix) where `0` is considered `False` and anything not zero (like `1`) is `True`. So if `random.choice([0, 1])` returns a `1` (or `True`) then we take `char.upper()` otherwise we take `char.lower()`:

```
>>> ransom = []
```

```
>>> for char in text:
...     ransom.append(char.upper() if random.choice([0, 1]) else char.lower())
...
>>> ''.join(ransom)
'The quIck brOwn Fox JumpS over ThE lAZY dOG.'
```

We can shorten this to one line of code if we use a list comprehension, essentially putting the `for` loop inside the brackets `[]` that create the `ransom` list:

```
>>> ransom = [c.upper() if random.choice([0, 1]) else c.lower() for c in text]
>>> ''.join(ransom)
'thE quicK bRowN foX JUmPs OVER tHe lAzY dog.'
```

All the code for deciding which case could go into a very small function which you could either write as a `lambda`:

```
>>> choose = lambda c: c.upper() if random.choice([0, 1]) else c.lower()
>>> choose('t')
'T'
```

Or the more standard `def` version:

```
>>> def choose(c):
...     return c.upper() if random.choice([0, 1]) else c.lower()
...
>>> choose('t')
't'
```

And then use that in your list comprehension. This version reads very well as is perhaps my favorite:

```
>>> ransom = [choose(c) for c in text]
>>> ''.join(ransom)
'thE QUicK broWN fOx JUmPS OVeR the lAZy doG.'
```

But I also quite like the `map` function which takes another function as the first argument which is applied to all the elements of second argument which is an iterable:

```
>>> ransom = map(lambda c: c.upper() if random.choice([0, 1]) else c.lower(), text)
>>> ''.join(ransom)
'ThE qUiCk BROwn FoX JUMps oVER THe lAzY dog.'
```

And that cleans up very nicely if instead we used our named function. This version is the shortest and perhaps cleanest but does require the reader to understand `map`:

```
>>> ransom = map(choose, text)
>>> ''.join(ransom)
'thE QUIck BrOwN FOX jumPs oVeR thE lAZY dOg.'
```

It may seem silly to spend so much time working through five ways to solve what is an essientially 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, c-like solution while the list comprehensions are very Pythonic and the `map` versions borrow from the world of purely functional languages like Haskell.

Chapter 11: Simple Rhymer

Write a Python program called `rhymer.py` that will create new words by removing the consonant(s) from the beginning (if any) of a given word and then create new words by prefixing the remainder with all the consonants and clusters that were not at the beginning. That is, prefix with all the consonants in the alphabet plus these clusters:

```
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
```

If given no arguments or the `-h|--help` flags, print a usage statement:

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

Make rhyming "words"

positional arguments:
  str         A word

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

If the word starts with a vowel, use the word as-is:

```
$ ./rhymer.py apple | head -3
bapple
capple
dapple
```

If the word begins with any consonants, remove them and append all the prefixes above making sure not to include any prefixes that match what you removed:

```
$ ./rhymer.py take | head -3
bake
cake
dake
$ ./rhymer.py take | grep take
stake
```

If the word doesn't match one of the above conditions, e.g., it is entirely consonants, print a message that you cannot rhyme it.

```
$ ./rhymer.py RDNZL
Cannot rhyme "RDNZL"
```

Hints:

The heart of the program for me is the stemming of the word. Do you even need to stemp it? Not if it begins with a vowel, so how can you detect that? I ended up writing a function called `stemmer` and inserted this into my `rhymer.py`:

```
def test_stemmer():
    """Test the stemmer"""

    assert ('c', 'ake') == stemmer('cake')
    assert ('ch', 'air') == stemmer('chair')
    assert ('', 'apple') == stemmer('apple')
    assert stemmer('bbb') is None
```

If you notice the `make test` target also include `rhymer.py`:

```
pytest -xv rhymer.py test.py
```

I wrote my `stemmer(word)` to return a tuple of `(prefix, stem)` where `prefix` will be the empty string when the `word` starts with a vowel. If the word starts with a consonant and can be split, I return the two parts of the word e.g., `chair` become `('ch', 'air')`. Otherwise I return `None` to indicate a failure to communicate.

If you choose to do the same, you can add the `test_stemmer` to your program and `pytest` will find any function with a name starting with `test_` to run. You can use this to verify that your `stemmer` does what you expect.

Solution

```
#!/usr/bin/env python3
"""Make rhyming words"""

import argparse
import re
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='str', help='A word to rhyme')

    return parser.parse_args()


# --------------------------------------------------
def stemmer(word):
    """Return leading consonants (if any), and 'stem' of word"""


def stemmer(word):
    vowels = 'aeiou'
    consonants = ''.join(
        filter(lambda c: c not in vowels, string.ascii_lowercase))
    match = re.match('^([' + consonants + ']*)([' + vowels + '].*)', word)
    if match:
        return match.groups()
    return None


# --------------------------------------------------
def test_stemmer():
    """Test the stemmer"""

    assert ('c', 'ake') == stemmer('cake')
    assert ('ch', 'air') == stemmer('chair')
    assert ('', 'apple') == stemmer('apple')
```

```
    assert stemmer('bbb') is None


# --------------------------------------------------
def main():
    """Make a jazz noise here"""
    args = get_args()
    word = args.word
    stemmed = stemmer(word.lower())
    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()

    if stemmed:
        start, rest = stemmed
        print('\n'.join([p + rest for p in prefixes if p != start]))
    else:
        print('Cannot rhyme "{}"'.format(word))


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

Discussion

As stated in the description, I spent most of my time working out how to stem a word. Some other programs in the book require this idea (Soundex rhymer, Runny Babbit), so you might look there, too. I decided to write a function `stemmer(word)` that will return a tuple of `(prefix, stem)`.

We need to check if the word can be split into one or more consonants followed by at least one vowel and maybe some other stuff, e.g., `'ha'` could be `('h', 'a')`. The easiest way is to write a regular expression using the `re` module. We've already defined the `vowels`, so we can use those to find the complement of `consonants`. I can iterate through the letters of the alphabet by using `string.ascii_lowercase` and find those not in the `vowels`:

```
>>> import string
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
>>> vowels = 'aeiou'
>>> consonants = ''.join(
...     filter(lambda c: c not in vowels, string.ascii_lowercase))
>>> consonants
'bcdfghjklmnpqrstvwxyz'
```

Here we see the use of `filter` which is a "higher-order" function takes a *another function* as the first argument and an iterable as the second argument. The `lambda c` keyword creates an anonymous function with a single argument I call `c` (for "character") which can then be referenced in the function body.

The more Pythonic way to write this would be a list comprehension:

```
>>> consonants = ''.join([c for c in string.ascii_lowercase if c not in vowels])
>>> consonants
'bcdfghjklmnpqrstvwxyz'
```

Both ways are fine. It's mostly preference, though true Pythonistas would probably disagree. If nothing else, the `filter` might be slower than a comprehension, especially if the iterable were large, so choose whichever way makes more sense for your style and application.

The regular expression is a bit tricky. We want to find consonants at the beginning, so we can use the caret (`^`) to anchor the regex to the start of the string.

```
>>> r = '^'
>>> r
'^'
```

Then we create a "character class" using `[]` and enumerate inside all the characters that are allowed:

```
>>> r = '^[' + consonants + ']'
>>> r
'^[bcdfghjklmnpqrstvwxyz]'
```

We will want to "capture" these so we can extract them later, so we put parentheses `()` around the character class to group them:

```
>>> r = '^([' + consonants + '])'
>>> r
'^([bcdfghjklmnpqrstvwxyz])'
```

Let's try that and see what we get:

```
>>> import re
>>> re.search(r, 'chair')
<re.Match object; span=(0, 1), match='c'>
```

Hmm, it didn't match `ch` because we didn't tell the regex *how many* to match, so it just matched one. We can add `*` to indicate "zero or more":

```
>>> r = '^([' + consonants + ']*)'
>>> r
'^([bcdfghjklmnpqrstvwxyz]*)'
>>> re.search(r, 'chair')
<re.Match object; span=(0, 2), match='ch'>
```

Very nice. Sometimes you'll see `+` to mean that a pattern can be repeated, but that one means "one or more." By using `*`, I'm relying on the fact that "zero" matches will always be true, so this will also help me find any `word` that begins with a vowel (although it doesn't seem like it just yet):

```
>>> re.search(r, 'apple')
<re.Match object; span=(0, 0), match=''>
```

Now I want to say that after some optional consonant prefix there must be at least one vowel:

```
>>> r = '^([' + consonants + ']*)' + '([' + vowels + '])'
>>> r
'^([bcdfghjklmnpqrstvwxyz]*)([aeiou])'
>>> re.search(r, 'chair')
<re.Match object; span=(0, 3), match='cha'>
>>> re.search(r, 'apple')
<re.Match object; span=(0, 1), match='a'>
```

Getting closer, but we need the regular expression to reach the end of the word now, so we add `.*` where `.` means "one of anything" and `*` means "zero or more":

```
>>> r = '^([' + consonants + ']*)' + '([' + vowels + '].*)'
>>> r
'^([bcdfghjklmnpqrstvwxyz]*)([aeiou].*)'
```

```
>>> re.search(r, 'chair')
<re.Match object; span=(0, 5), match='chair'>
>>> re.search(r, 'apple')
<re.Match object; span=(0, 5), match='apple'>
```

Great! We're matching the entire word. The true magic comes in when we look at the capture `groups`:

```
>>> re.search(r, 'chair').groups()
('ch', 'air')
>>> re.search(r, 'apple').groups()
('', 'apple')
```

That is exactly what I wanted to return! For what it's worth, I can get each `group` individually by referencing their order:

```
>>> re.search(r, 'chair').group(1)
'ch'
>>> re.search(r, 'apple').group(2)
'apple'
```

If I can't `match` a string:

```
>>> type(re.search(r, 'RDNZL'))
<class 'NoneType'>
```

I return `None` from my function:

```
>>> def stemmer(word):
...     vowels = 'aeiou'
...     consonants = ''.join(
...         filter(lambda c: c not in vowels, string.ascii_lowercase))
...     match = re.match('^([' + consonants + ']*)([' + vowels + '].*)', word)
...     if match:
...         return match.groups()
...     return None
...
>>> stemmer('apple')
('', 'apple')
>>> stemmer('chair')
('ch', 'air')
>>> stemmer('RDNZL')
```

So, given a working `stemmer` I try to stem a given `word`. If there is no result, I print the message that I cannot rhyme the word. Otherwise I iterate over all the prefixes:

```
>>> 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()
```

And add them to the stem of the word, being sure to avoid any prefix that was the same as the original word:

```
>>> start, rest = stemmer('chair')
>>> start
'ch'
>>> rest
'air'
>>> [p + rest for p in prefixes if p != start][:3]
['bair', 'cair', 'dair']
```

Chapter 12: Rock, Paper, Scissors

Write a Python program called `rps.py` that will play the ever-popular "Rock, Paper, Scissors" game. As often as possible, insult the player by combining an adjective and a noun from the following lists:

Adjectives = truculent fatuous vainglorious fatuous petulant moribund jejune feckless antiquated rambunctious mundane misshapen glib dreary dopey devoid deleterious degrading clammy brazen indiscreet indecorous imbecilic dysfunctional dubious drunken disreputable dismal dim deficient deceitful damned daft contrary churlish catty banal asinine infantile lurid morbid repugnant unkempt vapid decrepit malevolent impertinent decrepit grotesque puerile

Nouns = abydocomist bedswerver bespawler bobolyne cumberworld dalcop dew-beater dorbel drate-poke driggle-draggle fopdoodle fustylugs fustilarian gillie-wet-foot gnashgab gobermouch gowpenful-o'-anything klazomaniac leasing-monger loiter-sack lubberwort muck-spout mumblecrust quisby raggabrash rakefire roiderbanks saddle-goose scobberlotcher skelpie-limmer smell-feast smellfungus snoutband sorner stampcrab stymphalist tallowcatch triptaker wandought whiffle-whaffle yaldson zoilist

The program should accept a `-s|--seed` to pass to `random`.

```
$ ./rps.py
1-2-3-Go! [rps|q] r
You: Rock
Me : Scissors
You win. You are a clammy drate-poke.
1-2-3-Go! [rps|q] t
You dysfunctional dew-beater! Please choose from: p, r, s.
1-2-3-Go! [rps|q] p
You: Paper
Me : Rock
You win. You are a dismal gillie-wet-foot.
1-2-3-Go! [rps|q] q
Bye, you imbecilic fopdoodle!
```

Solution

```
#!/usr/bin/env python3
"""Rock, Paper, Scissors"""

import argparse
import os
import random
import sys


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

    parser = argparse.ArgumentParser(
        description='Rock, Paper, Scissors',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('-s',
                        '--seed',
                        help='Random seed',
                        metavar='int',
                        type=int,
                        default=None)

    return parser.parse_args()


# --------------------------------------------------
def insult():
    adjective = """
    truculent fatuous vainglorious fatuous petulant moribund jejune
    feckless antiquated rambunctious mundane misshapen glib dreary
    dopey devoid deleterious degrading clammy brazen indiscreet
    indecorous imbecilic dysfunctional dubious drunken disreputable
    dismal dim deficient deceitful damned daft contrary churlish
    catty banal asinine infantile lurid morbid repugnant unkempt
    vapid decrepit malevolent impertinent decrepit grotesque puerile
    """.split()

    noun = """
    abydocomist bedswerver bespawler bobolyne cumberworld dalcop
    dew-beater dorbel drate-poke driggle-draggle fopdoodle fustylugs
    fustilarian gillie-wet-foot gnashgab gobermouch
```

```
    gowpenful-o'-anything klazomaniac leasing-monger loiter-sack
    lubberwort muck-spout mumblecrust quisby raggabrash rakefire
    roiderbanks saddle-goose scobberlotcher skelpie-limmer
    smell-feast smellfungus snoutband sorner stampcrab stymphalist
    tallowcatch triptaker wandought whiffle-whaffle yaldson zoilist
    """.split()

    return ' '.join([random.choice(adjective), random.choice(noun)])


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

    args = get_args()
    random.seed(args.seed)

    valid = set('rps')
    beats = {'r': 's', 's': 'p', 'p': 'r'}
    display = {'r': 'Rock', 'p': 'Paper', 's': 'Scissors'}

    while True:
        play = input('1-2-3-Go! [rps|q] ').lower()

        if play.startswith('q'):
            print('Bye, you {}!'.format(insult()))
            sys.exit(0)

        if play not in valid:
            print('You {}! Please choose from: {}.'.format(
                insult(), ', '.join(sorted(valid))))
            continue

        computer = random.choice(list(valid))

        print('You: {}\nMe : {}'.format(display[play], display[computer]))

        if beats[play] == computer:
            print('You win. You are a {}.'.format(insult()))
        elif beats[computer] == play:
            print('You lose, {}!'.format(insult()))
        else:
            print('Draw, you {}.'.format(insult()))


# --------------------------------------------------
```

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

Chapter 13: Dial-A-Curse

Write a Python program called `abuse.py` that generates some `-n|--number` of insults (default `3`) by randomly combining some number of `-a|--adjectives` (default `2`) with a noun (see below). Be sure your program accepts a `-s|--seed` argument (default `None`) to pass to `random.seed`.

The are the 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 barbermonger 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

If run with the `-h|--help` flag, the program should generate usage:

```
$ ./abuse.py -h
usage: abuse.py [-h] [-a int] [-n int] [-s int]

Argparse Python script

optional arguments:
  -h, --help            show this help message and exit
  -a int, --adjectives int
                        Number of adjectives (default: 2)
  -n int, --number int  Number of insults (default: 3)
  -s int, --seed int    Random seed (default: None)
```

When run with no arguments, the program should generate insults using the defaults:

```
$ ./abuse.py
You slanderous, rotten block!
You lubbery, scurilous ratcatcher!
You rotten, foul liar!
```

It's unlikely you'll get the same output above when you run yours because no seed was set. The following, however, should be exactly reproducible due to the `--seed`:

```
$ ./abuse.py -s 1 -n 2 -a 1
You rotten rogue!
```

```
You lascivious ape!
$ ./abuse.py -s 2 -n 4 -a 4
You scurilous, foolish, vile, foul milksop!
You cullionly, lubbery, heedless, filthy lunatic!
You foul, lecherous, infected, slanderous degenerate!
You base, ruinous, slanderous, false liar!
```

If run with a `--number` less than 1, exit with an error code and message, preferably with the usage:

```
$ ./abuse.py -n -4
usage: abuse.py [-h] [-a int] [-n int] [-s int]
abuse.py: error: --number "-4" cannot be less than 1
```

Hints:

- You can use three single or double quotes (""") to create a multi-line string and then `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).
- Perform the check for `--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 for `args.seed` to `None` while using a `type=int`, you should be able to directly pass the argument's value to `random.seed` to control testing.
- 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 `sample` and `choice` functions in the `random` module for help in selecting some adjectives and a noun.
- To construct an insult string to print, you can use the `+` operator to concatenate strings, use the `str.join` method, or use format strings (and maybe other methods?).

Solution

```
#!/usr/bin/env python3
"""Heap abuse"""

import argparse
import random
import sys

adjectives = """
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
""".strip().split()

nouns = """
Judas Satan ape ass barbermonger 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()


# --------------------------------------------------
def get_args():
    """get command-line arguments"""

    parser = argparse.ArgumentParser(
        description='Heap abuse',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('-a',
                        '--adjectives',
                        help='Number of adjectives',
                        metavar='int',
                        type=int,
                        default=2)

    parser.add_argument('-n',
                        '--number',
                        help='Number of insults',
                        metavar='int',
                        type=int,
```

```
                         default=3)

    parser.add_argument('-s',
                        '--seed',
                        help='Random seed',
                        metavar='int',
                        type=int,
                        default=None)

    args = parser.parse_args()

    if args.number < 1:
        parser.error('--number "{}" cannot be less than 1'.format(args.number))

    return args


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

    args = get_args()
    num_adj = args.adjectives
    num_insults = args.number
    random.seed(args.seed)

    for _ in range(num_insults):
        adjs = random.sample(adjectives, k=num_adj)
        noun = random.choice(nouns)
        print('You {} {}!'.format(', '.join(adjs), noun))


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

Discussion

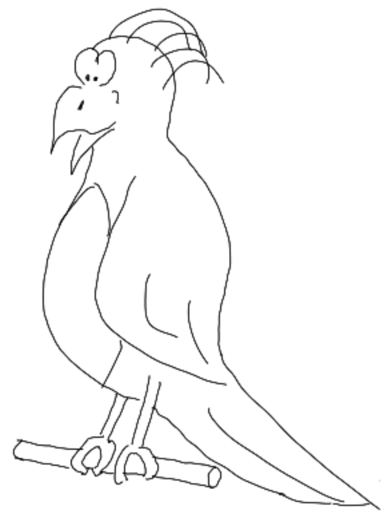


Figure 8: Erak! The captain is a corrupt, irksome fiend!

get_args

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. You can use the `type` function in Python to find out how Python represents a value:

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

For `--adjectives` and `--number`, I can set reasonable defaults so that no input is *required* from the user but the values are easily overridden. 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.

Another reason I quite like `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

For instance, 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 call `args = get_args()` in my `main` function, I know that all the arguments have been validated. I could have also added a similar check for `--adjectives`, but the main point was to highlight that such a thing is possible. As you write your own programs, you'll have to decide how much validation of user input you feel is necessary.

main

Once I'm in `main` and have my arguments, I can control the randomness of the program by calling `random.seed(args.seed)` because:

1. The default value of the `seed` is `None`, and setting `random.seed` to `None` is the same as not setting it at all.
2. The `type` of `args.seed` is `int` which is the proper type for `random.seed`. I do not have to validate the argument further. Negative integers are valid values.

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 _, e.g., here I can unpack a 3-tuple so as to get the middle value:

```
>>> x = 'Jesus', 'Mary', 'Joseph'
>>> _, name, _ = x
>>> name
'Mary'
```

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 lubbery old
... peevish rascaly rotten ruinous scurilous scurvy slanderous
... sodden-witted thin-faced toad-spotted unmannered vile wall-eyed
... """.strip().split()
>>> nouns = """
... Judas Satan ape ass barbermonger 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 concatenante them together, I need to put `', '` (a comma and a space) between each of the adjectives, and I 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:

```
>>> noun = random.choice(nouns)
>>> print('You {} {}!'.format(', '.join(adjs), noun))
You thin-faced, scurvy, sodden-witted liar!
```

And now you have a handy way to make enemies and influence people.

Chapter 14: Scrambler

Write a Python program called `scrambler.py` that will take a single position positional argument that is text or a text file and then convert each word into a scrambled version. The scrambling should only work on words greater than 3 characters in length and should only scramble the letters in the middle, leaving the first and last characters unchanged. The program should take a `-s|--seed` argument (default `None`) to pass to `random.seed`.

Cf. Typoglycemia https://www.dictionary.com/e/typoglycemia/

We'll need to use the same algorithm for scrambling the words. I used the `random.shuffle` method to mix up the letters in the middle, being sure that the word that gets created is not the same as the word that you are given. If the word is 3 characters or shorter, just return the word unchanged.

Another very tricky bit is that we want to scramble all the "words" on each line and leave everything that's not a "word" unchanged. We'll use a regular expression that looks for strings composed only of the characters a-z, A-Z, and the single quote so we can find words like `can't` or `Susie's`. Everything else will be consider not a word. Here is the regex you should use:

```
>>> import re
>>> text = 'this is  a\n"sentence?"'
>>> re.split(r'(\W+)', text)
['this', ' ', 'is', '  ', 'a', '\n"', 'sentence', '?"', '']
```

Now scramble all the things that are "words"!

Here is how the program should perform:

```
$ ./scrambler.py
usage: scrambler.py [-h] [-s int] STR
scrambler.py: error: the following arguments are required: STR
$ ./scrambler.py -h
usage: scrambler.py [-h] [-s int] STR

Scramble the letters of words

positional arguments:
  STR                 Input text or file

optional arguments:
  -h, --help          show this help message and exit
  -s int, --seed int  Random seed (default: None)
```

It should handle text on the command line:

```
$ ./scrambler.py -s 1 foobar
faobor
```

```
$ ./scrambler.py -s 1 "foobar bazquux"
faobor buuzaqx
```

Or from a file:

```
$ ./scrambler.py -s 1 ../inputs/the-bustle.txt
The blutse in a hsoue
The monrnig atefr dteah
Is snleoemst of iusinedrts
Eatcend uopn etarh,--

The sewnpeig up the hreat,
And ptunitg lvoe aawy
We slahl not wnat to use agian
Utnil ertiteny.
```

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='STR', help='Input text or file')

    parser.add_argument('-s',
                        '--seed',
                        help='Random seed',
                        metavar='int',
                        type=int,
                        default=None)

    args = parser.parse_args()

    if os.path.isfile(args.text):
        args.text = open(args.text).read()

    return args


# --------------------------------------------------
def scramble(word):
    """For words over 3 characters, shuffle the letters in the middle"""

    if len(word) > 3 and re.match(r'\w+', word):
        orig = list(word[1:-1])
        middle = orig.copy()
        if len(set(middle)) > 1:
            while middle == orig:
```

```
                random.shuffle(middle)
        word = word[0] + ''.join(middle) + word[-1]

    return word


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

    args = get_args()
    text = args.text
    random.seed(args.seed)

    for line in text.splitlines():
        print(''.join(map(scramble, re.split(r'\b', line))))


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

Discussion

As with several other programs, we want to take our `text` either from the command line or from a file. I decided to put this logic into the `get_args` function and detect in there if `args.text` is a file and read it so that by the time I call `get_args` I already have the `text` I need. Since `--seed` has a default of `None`, I can directly pass it to `random.seed`. If the seed is `None`, it's the same as not setting the seed. If the seed is defined (and it must be an `int` because of the constraint in `argparse`), then it sets the seed properly.

Because I want to preserve the shape of the input text, I decided to handle the `text` line-by-line by calling `text.splitlines()`. If we are reading the "spiders" haiku, the first line is:

```
>>> line = 'Don't worry, spiders,'
```

We need to break the line into "words" which we often do with `str.split`:

```
>>> line.split()
['Don't', 'worry,', 'spiders,']
```

But that leaves punctation stuck to our words. Instead, we'll `import re` to get the regular expression module and split on word boundaries (`\b`):

```
>>> re.split(r'\b', line)
['', 'Don', ''', 't', ' ', 'worry', ', ', 'spiders', ',']
```

That actually breaks "Don't" into two words, but we'll just not worry about that. So let's think about how we'll scramble our words by starting with just one word.

Scrambling one word

Given any particular "word" that:

1. looks like a string
2. is longer than 3 characters

We want to scramble the middle of any string, where the "middle" is everything after the first character up to the second to last character.

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. The position of any string is `-1`:

```
>>> word = 'spiders'
>>> word[0]
's'
>>> word[-1]
's'
```

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
'pider'
```

We can `import random` to get access to the `shuffle` method. You have to know that this method mutates the given list **in-place**, and that's going to cause a problem:

```
>>> import random
>>> 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
```

Hey, wha' happan? This is a bit tricky to understand, but basically when we defined the `middle` variable, we were just *pointing to a part of a string*, and strings are immutable. We'd get same error if we did `random.shuffle('ooba')`. We need `middle` make a new `list` of the characters from `word`:

```
>>> middle = list(word[1:-1])
>>> middle
['p', 'i', 'd', 'e', 'r']
```

And that is something we can `shuffle`:

```
>>> random.shuffle(middle)
>>> middle
['r', 'e', 'p', 'i', 'd']
```

While writing the program, I found that the shuffling didn't always result in a different order, so I added a bit of logic to keep shuffling until I get something new. Another problem I encountered was creating an infinite loop while comparing my shuffled string to the original when the word was "keep" because the middle is "ee" and no matter how many times you shuffle that it will always be "ee", so I added a check that the unique `set` of letters in the middle is greater than 1:

```
>>> orig = list(word[1:-1])
>>> middle = orig.copy()
>>> if len(set(middle)) > 1:
...     while middle == orig:
...         random.shuffle(middle)
...
>>> middle
['p', 'i', 'd', 'r', 'e']
```

```
>>> middle
['r', 'e', 'p', 'i', 'd']
```

And now I can put the word back together with the original first and last characters:

```
>>> word = word[0] + ''.join(middle) + word[-1]
>>> word
'srepids'
```

Scrambling all the words

Now I have a function `scramble(word)`:

```
>>> def scramble(word):
...     """For words over 3 characters, shuffle the letters in the middle"""
...     if len(word) > 3 and re.match(r'\w+', word):
...         orig = list(word[1:-1])
...         middle = orig.copy()
...         if len(set(middle)) > 1:
...             while middle == orig:
...                 random.shuffle(middle)
...         word = word[0] + ''.join(middle) + word[-1]
...     return word
...
```

And a way to break up each line into word-like piecies (using `re.split`).

I need to apply a function to a list, and that is exactly what `map` does. For instance, I could `split` the line into words:

```
>>> line.split()
['Don’t', 'worry,', 'spiders,']
```

And `map` that into the `len` function to find the lengths of each element in the list. In order to evaluate the resulting `map` object, I have to use `list` in the REPL (but not in actual code):

```
>>> list(map(len, line.split()))
[5, 6, 8]
```

Instead, we'll `map` into our `scramble` function and split on word boundaries:

```
>>> list(map(scramble, re.split(r'\b', line)))
['', 'Don', '’', 't', ' ', 'wrory', ', ', 'sdeirps', ',']
```

And then put that `list` back together by joining on the empty string:

```
>>> ''.join(map(scramble, re.split(r'\b', line)))
'Don’t wrroy, sperdis,'
```

I do this for each line of text, printing the scrambled line, and that is the whole program.

If you don’t like `map`, you can accomplish the same thing with a list comprehension:

```
>>> ''.join([scramble(w) for w in re.split(r'\b', line)])
'Don’t wrory, sdepris,'
```

Chapter 15: Bacronym

Write a Python program called `bacronym.py` that takes a string like "FBI" and retrofits some `-n|--num` (default `5`) of acronyms by reading a `-w|--wordlist` argument (default `/usr/share/dict/words`), skipping over words to `-e|--exclude` (default `a, an, the`) and randomly selecting words that start with each of the letters. Be sure to include a `-s|--seed` argument (default `None`) to pass to `random.seed` for the test suite.

If provided the `-h|--help` flags or no arguments, the program should print a usage:

```
$ ./bacronym.py
usage: bacronym.py [-h] [-n NUM] [-w STR] [-x STR [STR ...]] [-s INT] STR
bacronym.py: error: the following arguments are required: STR
$ ./bacronym.py -h
usage: bacronym.py [-h] [-n NUM] [-w STR] [-x STR [STR ...]] [-s INT] STR

Explain acronyms

positional arguments:
  STR                   Acronym

optional arguments:
  -h, --help            show this help message and exit
  -n NUM, --num NUM     Maximum number of definitions (default: 5)
  -w STR, --wordlist STR
                        Dictionary/word file (default: /usr/share/dict/words)
  -x STR [STR ...], --exclude STR [STR ...]
                        List of words to exclude (default: ['a', 'an', 'the'])
  -s INT, --seed INT    Random seed (default: None)
```

Because I'm including a `--seed` argumment, you should get this exact output if using the same `--wordlist`:

```
$ ./bacronym.py -s 1 FBI
FBI =
 - Fecundity Brokage Imitant
 - Figureless Basketmaking Ismailite
 - Frumpery Bonedog Irregardless
 - Foxily Blastomyces Inedited
 - Fastland Bouncingly Idiospasm
```

The program should create errors and usage for `--num` less than 1:

```
$ ./bacronym.py -n -3 AAA
usage: bacronym.py [-h] [-n NUM] [-w STR] [-x STR [STR ...]] [-s INT] STR
bacronym.py: error: --num "-3" must be > 0
```

And for a bad `--wordlist`:

```
$ ./bacronym.py -w mnvdf AAA
usage: bacronym.py [-h] [-n NUM] [-w STR] [-x STR [STR ...]] [-s INT] STR
bacronym.py: error: argument -w/--wordlist: can't open 'mnvdf': \
[Errno 2] No such file or directory: 'mnvdf'
```

The acronym must be composed entirely of letters:

```
$ ./bacronym.py 666
usage: bacronym.py [-h] [-n NUM] [-w STR] [-x STR [STR ...]] [-s INT] STR
bacronym.py: error: Acronym "666" must be >1 in length, only use letters
```

And be greater than 1 character in length:

```
$ ./bacronym.py A
usage: bacronym.py [-h] [-n NUM] [-w STR] [-x STR [STR ...]] [-s INT] STR
bacronym.py: error: Acronym "A" must be >1 in length, only use letters
```

Hints:

- See how much error checking you can put into the `get_args` function and use `parser.error` to throw the errors
- The `--wordlist` need not be a system dictionary file with one lower-case word on each line. Assume that you can read any file with many words on each line and that might include punctuation. I suggest you use a regualar expression to remove anything that is not an alphabet character with `re.sub('[^a-z]', '')`. Be sure that words are only represented once in your list.
- In my version, I write two important functions: one (`group_words`) that reads the wordlist and returns a grouping of words by their first letter, and another (`make_definitions`) that produces plausible definitions from that grouping of words by letters for a given acronym. I place the following test functions into my program and run `pytest` to verify that the functions work properly.

```
def test_group_words():
    """Test group_words()"""

    words = io.StringIO('apple, "BANANA," The Coconut! Berry; A cabbage.')
    stop = 'a an the'.split()
    words_by_letter = group_words(words, stop)
    assert words_by_letter['a'] == ['apple']
    assert words_by_letter['b'] == ['banana', 'berry']
    assert words_by_letter['c'] == ['coconut', 'cabbage']
    assert 't' not in words_by_letter

def test_make_definitions():
    """Test make_definitions()"""
```

```python
words = {
    'a': ['apple'],
    'b': ['banana', 'berry'],
    'c': ['coconut', 'cabbage']
}

random.seed(1)
assert make_definitions('ABC', words) == ['Apple Banana Cabbage']
random.seed(2)
assert make_definitions('ABC', words) == ['Apple Banana Coconut']
random.seed(3)
assert make_definitions('AAA', words) == ['Apple Apple Apple']
random.seed(4)
assert make_definitions('YYZ', words) == ['? ? ?']
random.seed(None)
```

Solution

```
#!/usr/bin/env python3
"""Explain acronyms"""

import argparse
import io
import sys
import os
import random
import re
from collections import defaultdict
from functools import partial


# --------------------------------------------------
def get_args():
    """get arguments"""

    parser = argparse.ArgumentParser(
        description='Explain acronyms',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('acronym', help='Acronym', type=str, metavar='STR')

    parser.add_argument('-n',
                        '--num',
                        help='Maximum number of definitions',
                        type=int,
                        metavar='NUM',
                        default=5)

    parser.add_argument('-w',
                        '--wordlist',
                        help='Dictionary/word file',
                        type=argparse.FileType('r'),
                        metavar='STR',
                        default='/usr/share/dict/words')

    parser.add_argument('-x',
                        '--exclude',
                        help='List of words to exclude',
                        type=str,
                        metavar='STR',
                        nargs='+',
```

```
                        default='a an the'.split())

    parser.add_argument('-s',
                        '--seed',
                        help='Random seed',
                        type=int,
                        metavar='INT',
                        default=None)

    args = parser.parse_args()

    if args.num < 1:
        parser.error('--num "{}" must be > 0'.format(args.num))

    if not re.search(r'^[A-Z]{2,}$', args.acronym.upper()):
        msg = 'Acronym "{}" must be >1 in length, only use letters'.format(
            args.acronym)
        parser.error(msg)

    return args


# --------------------------------------------------
def group_words(file, stop_words=set()):
    """Groups words in file by first letter"""

    good = partial(re.search, r'^[a-z]{2,}$')
    seen = set()
    words_by_letter = defaultdict(list)
    clean = lambda word: re.sub('[^a-z]', '', word)
    for word in filter(good, map(clean, file.read().lower().split())):
        if word not in seen and word not in stop_words:
            seen.add(word)
            words_by_letter[word[0]].append(word)

    return words_by_letter


# --------------------------------------------------
def test_group_words():
    """Test group_words()"""

    words = io.StringIO('apple, "BANANA," The Coconut! Berry - APPLE; A cabbage.')
    stop = 'a an the'.split()
    words_by_letter = group_words(words, stop)

```

```
    assert words_by_letter['a'] == ['apple']
    assert words_by_letter['b'] == ['banana', 'berry']
    assert words_by_letter['c'] == ['coconut', 'cabbage']
    assert 't' not in words_by_letter


# --------------------------------------------------
def make_definitions(acronym, words_by_letter, limit=1):
    """Find definitions an acronym given groupings of words by letters"""

    definitions = []
    for _ in range(limit):
        definition = []
        for letter in acronym.lower():
            opts = words_by_letter.get(letter.lower(), [])
            definition.append(random.choice(opts).title() if opts else '?')
        definitions.append(' '.join(definition))

    return definitions


# --------------------------------------------------
def test_make_definitions():
    """Test make_definitions()"""

    words = {
        'a': ['apple'],
        'b': ['banana', 'berry'],
        'c': ['coconut', 'cabbage']
    }

    random.seed(1)
    assert make_definitions('ABC', words) == ['Apple Banana Cabbage']
    random.seed(2)
    assert make_definitions('ABC', words) == ['Apple Banana Coconut']
    random.seed(3)
    assert make_definitions('AAA', words) == ['Apple Apple Apple']
    random.seed(4)
    assert make_definitions('YYZ', words) == ['? ? ?']
    random.seed(None)


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

```

```
    args = get_args()
    acronym = args.acronym
    stop = set(map(str.lower, args.exclude))
    random.seed(args.seed)

    words_by_letter = group_words(args.wordlist, stop)
    definitions = make_definitions(acronym, words_by_letter, args.num)

    if definitions:
        print(acronym.upper() + ' =')
        for definition in definitions:
            print(' - ' + definition)
    else:
        print('Sorry I could not find any good definitions')


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

Discussion

Handling arguments

As suggested in the introduction, I check that the `--num` argument is a positive integer and that the given acronym is composed entirely of letters and is at least two characters in length. The second is achieved with a regular expression which returns `None` when it fails to match:

```
>>> import re
>>> acronym = 'A'
>>> type(re.search(r'^[A-Z]{2,}$', acronym.upper()))
<class 'NoneType'>
>>> acronym = '4E9'
>>> type(re.search(r'^[A-Z]{2,}$', acronym.upper()))
<class 'NoneType'>
>>> acronym = 'ABC'
>>> type(re.search(r'^[A-Z]{2,}$', acronym.upper()))
<class 're.Match'>
```

If any errors with the arguments are detected, I use `parser.error` to cause `argparse` to do the following:

1. Print the short usage
2. Print an error message
3. Exit the program with a non-zero exit value to indicate a failure

If you inspect the `test.py`, you can see that the tests for these bad inputs verify that the `rv` (return value) for these calls is not `0`. If you write "pipelines" on the command line where the output of one program is the input for the next, it is important to stop the process when a program exits with an error. Non-zero exit values can be used by tools like `make` to halt a larger execution of programs.

I defined my `--exclude` words with `nargs='+'` to indicate one or more string values, so I set the `default='a an the'.split()` which creates a list more easily than typing each individual word in quotes and `[]`:

```
>>> 'a an the'.split()
['a', 'an', 'the']
```

I can take that list and lowercase each word by mapping the values into the `str.lower` function. Note that `map` is a "lazy" function that only produces values when needed, so I have to use `list` in the REPL if I want to see the evaluated `list`:

```
>>> list(map(str.lower, 'A AN THE'.split()))
['a', 'an', 'the']
```

Then I can use `set` to create a unique list of stop words:

```
>>> exclude = 'a an the'.split()
>>> stop = set(map(str.lower, exclude))
>>> stop
{'the', 'an', 'a'}
```

Because I define the `--seed` to be `type=int` with a `default=None`, I can pass `args.seed` directly to `random.seed`. I also define `--wordlist` using `type=argparse.FileType('r')` to ensure that the value is a readable file, and I set the default to my system dictionary.

Grouping words by first letters

After validating the arguments, the next big conceptual task is reading the `--wordlist` and grouping the words by their first letters. A dictionary is perfect for this sort of task. Many times we associate some single value like a string to some other single value like another string or a number, e.g., a last name to a first name or a name to an age. Here, though, we want to link a letter like `a` to a `list` of words that start with that letter.

It's tedious to check for the existence of a key and then create a new `list` if that key doesn't exist, so let's use the `defaultdict` for this. The argument to `defaultdict` is the **type** of data we want to use for the **value** of a new entry. That is, if we start with an empty `dict` called `words` and try to access `words['a']`, it will blow up:

```
>>> words = dict()
>>> words['a']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'a'
```

If instead we create `words` as a `defaultdict` that initializes an undefined key with an empty list, we get this instead:

```
>>> from collections import defaultdict
>>> words = defaultdict(list)
>>> words['a']
[]
```

Which means we can call `list` methods on the values of elements in the `dict`, methods like `append`:

```
>>> words['b'].append('banana')
>>> words
defaultdict(<class 'list'>, {'a': [], 'b': ['banana']})
```

Since we defined the `--wordlist` to be a readable file, `argparse` has already delivered to us an open file handle upon which we can call `read`. I also want to lowercase the line and then `split` it into word-like units. I'm going to create an `io.StringIO` object here that I also use in the test to create a string that will behave like an open filehandle:

```
>>> fh = io.StringIO('apple, "BANANA," The Coconut! Berry - APPLE; A cabbage.')
>>> type(fh)
<class '_io.StringIO'>
```

My goal is to turn this into a data structure where the words are grouped by their first, lowercased letter, something like this:

```
'a' = ['apple']
'b' = ['banana']
'c' = ['cabbage', 'coconut']
```

I can chain the methods `read`, `lower`, and `split` to get word-like units. Note that I can only `read` an `io.StringIO` object once. Just like a file handle, once it is exhausted it has to be opened again:

```
>>> words = fh.read().lower().split()
>>> words
['apple,', '"banana,"', 'the', 'coconut!', 'berry', '-', 'apple;', 'a', 'cabbage.']
```

We're getting closer, but we still need to remove anything that's not a letter from each word. We can create a `clean` function to do this. It's really just one line of code, so I can actually make it like so:

```
>>> clean = lambda word: re.sub('[^a-z]', '', word)
```

And I can `map` all the words into that function to get actual "words":

```
>>> words = list(map(clean, fh.read().lower().split()))
>>> words
['apple', 'banana', 'the', 'coconut', 'berry', '', 'apple', 'a', 'cabbage']
```

We only want to take words that are at least 2 characters long, so we can create a regular expression for this:

```
>>> good = re.compile(r'^[a-z]{2,}$')
```

And we can use it like so:

```
>>> type(good.search('banana'))
<class 're.Match'>
>>> type(good.search('i'))
<class 'NoneType'>
```

I'd actually like to use it as a `filter` for the elements coming out of the `map`, but there's a problem in that we can't use it like it is:

```
>>> words = list(filter(good, map(clean, fh.read().lower().split())))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 're.Pattern' object is not callable
```

It’s a bit cryptic to figure this out, but the problem is with the fact that `good` is not a function, it’s a compiled regular expression:

```
>>> type(good)
<class 're.Pattern'>
```

What we want is something that is something that uses `re.match` with a regex to filter the elements. The `re.match` function takes two arguments, and the `filter` will automatically feed in the words, so what we need is a *partially applied* function where the first argument (the regex pattern) is already bound. We can use `functools.partial` for this:

```
>>> good = partial(re.search, r'^[a-z]{2,}$')
>>> type(good('banana'))
<class 're.Match'>
>>> type(good('x'))
<class 'NoneType'>
```

And it can now be a part of our chain:

```
>>> words = list(filter(good, map(clean, fh.read().lower().split())))
>>> words
['apple', 'banana', 'the', 'coconut', 'berry', 'apple', 'cabbage']
```

So we’re just read the input file, split it into words, removed any bad characters, and filtered out unwanted strings in one line of code that is extremely readable! To avoid adding words more than once, I created a `seen` variable from a `set` to check if a word has been seed before. I was also given a list of `stop` words to avoid which is also a `set` (or it could just as easily be a `list`), so I need to check that any given `word` is not in either of these.

```
>>> stop
{'the', 'an', 'a'}
>>> seen = set()
>>> for word in words:
...     if not any([word in stop, word in seen]):
...         print(word)
...         seen.add(word)
...
apple
banana
coconut
berry
cabbage
```

And you can see that “apple” was only printed once. Returning to the end goal of making a list of words by first letter, we return to our `defaultdict(list)` that we started off with:

```
>>> words_by_letter = defaultdict(list)
```

```
>>> for word in words:
...     if not any([word in stop, word in seen]):
...         words_by_letter[word[0]].append(word)
...         seen.add(word)
...
>>> from pprint import pprint as pp
>>> pp(words_by_letter)
defaultdict(<class 'list'>,
            {'a': ['apple'],
             'b': ['banana', 'berry'],
             'c': ['coconut', 'cabbage']})
```

Finally we can make all this a function called `group_words`. Note that I'll make the `stop` words an option by adding a default value:

```
>>> def group_words(file, stop_words=set()):
...     """Groups words in file by first letter"""
...     good = partial(re.search, r'^[a-z]{2,}$')
...     seen = set()
...     words_by_letter = defaultdict(list)
...     clean = lambda word: re.sub('[^a-z]', '', word)
...     for word in filter(good, map(clean, file.read().lower().split())):
...         if word not in seen and word not in stop_words:
...             seen.add(word)
...             words_by_letter[word[0]].append(word)
...     return words_by_letter
...
```

I can call it with an open file handle:

```
>>> pp(group_words(open('../inputs/fox.txt')))
defaultdict(<class 'list'>,
            {'b': ['brown'],
             'd': ['dog'],
             'f': ['fox'],
             'j': ['jumps'],
             'l': ['lazy'],
             'o': ['over'],
             'q': ['quick'],
             't': ['the']})
```

Most importantly, I can write a test which I'll call `test_group_words` (included in the introduction) so that `pytest` will execute it. My test sends in a fake (or "mock" in testing parlance) file handle and checks that the expected words are present and absent. It may seem like overkill to put just a few lines of code into a function, but it's very important to write small functions that do essentially one thing and which can be tested!

Making definitions

Similarly, I made a small function that takes the grouped words, an acronym, and number and returns a list of that number of plausible definitions. I can use a `for` loop with `range(n)` to iterate `n` times through some code. Since I don't need the number for each loop, I can use an underscore (`_`) to throwaway the value:

```
>>> limit = 2
>>> for _ in range(limit):
...     print('hi')
...
hi
hi
```

So, for however many definitions I want, I need to loop through each letter of the acronym and select some word from the grouped words:

```
>>> pp(words_by_letter)
defaultdict(<class 'list'>,
            {'a': ['apple'],
             'b': ['banana', 'berry'],
             'c': ['coconut', 'cabbage']})
>>> import random
>>> definition = []
>>> acronym = 'ABC'
>>> for letter in acronym:
...     opts = words_by_letter.get(letter.lower(), [])
...     definition.append(random.choice(opts).title() if opts else '?')
...
>>>
>>> definition
['Apple', 'Berry', 'Coconut']
```

Depending on the wordlist I read, a given letter may not exist, so I use the `dict.get` method to safely look for a `letter` with the default return value being an empty list `[]`. Then I can use an `if` *expression* to use `random.choice` to select from those options if they exists or use the question mark `?` to indicate no possible value. I can put all this into a function:

```
>>> def make_definitions(acronym, words_by_letter, limit=1):
...     definitions = []
...     for _ in range(limit):
...         definition = []
...         for letter in acronym.lower():
...             opts = words_by_letter.get(letter.lower(), [])
...             definition.append(random.choice(opts).title() if opts else '?')
...         definitions.append(' '.join(definition))
```

```
...     return definitions
...
>>> make_definitions('ABC', words_by_letter)
['Apple Berry Coconut']
>>> make_definitions('ABX', words_by_letter)
['Apple Berry ?']
```

The `test_make_definitions` function included in the introduction ensures that this function works properly.

Putting it together

To recap, so far we've written three central functions to:

1. Parse and validate the user arguments
2. Read the word list file and group the words by their first letters
3. Make definitions from the grouped words for the given acronym

Now we can write very understandable, almost self-documenting code:

```
>>> words_by_letter = group_words(open('/usr/share/dict/words'), stop)
>>> definitions = make_definitions('YYZ', words_by_letter, 2)
>>> definitions
['Yearock Yon Zone', 'Yacca Yengee Zincalo']
```

If we are able to make some `definitions`, we will print them out; otherwise we can apologize:

```
>>> if definitions:
...     print(acronym.upper() + ' =')
...     for definition in definitions:
...         print(' - ' + definition)
... else:
...     print('Sorry I could not find any good definitions')
...
ABC =
 - Yearock Yon Zone
 - Yacca Yengee Zincalo
```

Testing

In the introduction, I encouraged you to write a couple of functions that included specific tests that live *inside* your program. Such tests help you know that the building blocks of your code work – what are often called "unit tests."

Additionally, you have been provided a test suite that checks that the program works from the *outside*. However you implement the logic of the code, these tests check that the whole program works – what might be called "integration tests."

As you write your own programs, you should think about writing very small functions that do *one thing* and then writing tests to be sure they actually do the thing you think and always continue to do that thing as you change your program. Additionally, you need to write tests to make sure that all the parts work together to accomplish the larger task at hand. While writing and refactoring this program, I repeatedly updated and used my test suite to ensure I wasn't introducing bugs!

Chapter 16: Workout Of (the) Day (WOD)

Write a Python program called `wod.py` that will create a Workout Of (the) Day (WOD) from a list of exercises provided in CSV format (default `wod.csv`). Accept a `-n|--num_exercises` argument (default 4) to determine the sample size from your exercise list. Also accept a `-e|--easy` flag to indicate that the reps should be cut in half. Finally accept a `-s|--seed` argument to pass to `random.seed` for testing purposes. You should use the `tabulate` module to format the output as expected.

The input file should be comma-separated values with headers for "exercise" and "reps," e.g.:

```
$ tablify.py wod.csv
+---------------+--------+
| exercise      | reps   |
|---------------+--------|
| Burpees       | 20-50  |
| Situps        | 40-100 |
| Pushups       | 25-75  |
| Squats        | 20-50  |
| Pullups       | 10-30  |
| HSPU          | 5-20   |
| Lunges        | 20-40  |
| Plank         | 30-60  |
| Jumprope      | 50-100 |
| Jumping Jacks | 25-75  |
| Crunches      | 20-30  |
| Dips          | 10-30  |
+---------------+--------+
```

You should use the range of reps to choose a random integer value in that range.

```
$ ./wod.py -h
usage: wod.py [-h] [-f str] [-s int] [-n int] [-e]

Create Workout Of (the) Day (WOD)

optional arguments:
  -h, --help            show this help message and exit
  -f str, --file str    CSV input file of exercises (default: wod.csv)
  -s int, --seed int    Random seed (default: None)
  -n int, --num_exercises int
                        Number of exercises (default: 4)
  -e, --easy            Make it easy (default: False)
$ ./wod.py
Exercise      Reps
```

```
----------  ------
Crunches        26
HSPU             9
Squats          43
Pushups         36
$ ./wod.py -s 1
Exercise         Reps
-------------  ------
Pushups            32
Jumping Jacks      56
Situps             88
Pullups            24
$ ./wod.py -s 1 -e
Exercise         Reps
-------------  ------
Pushups            15
Jumping Jacks      27
Situps             44
Pullups            12
$ ./wod.py -f wod2.csv -n 5
Exercise                Reps
--------------------  ------
Erstwhile Lunges           9
Existential Earflaps      32
Rock Squats               21
Squatting Chinups         49
Flapping Leg Raises       17
```

Hints:

- Use the `csv` module's `DictReader` to read the input CSV files
- Break the `reps` field on the `-` character, coerce the low/high values to `int` values, and then use the `random` module to choose a random integer in that range. Also see if the `random` module can help you sample some exercises.
- Read the docs on the `tabulate` module to figure out to get it to print your data

Solution

```
#!/usr/bin/env python3
"""Create Workout Of (the) Day (WOD)"""

import argparse
import csv
import os
import random
from tabulate import tabulate
from dire import die


# --------------------------------------------------
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='str',
                        type=argparse.FileType('r'),
                        default='wod.csv')

    parser.add_argument('-s',
                        '--seed',
                        help='Random seed',
                        metavar='int',
                        type=int,
                        default=None)

    parser.add_argument('-n',
                        '--num_exercises',
                        help='Number of exercises',
                        metavar='int',
                        type=int,
                        default=4)

    parser.add_argument('-e',
                        '--easy',
                        help='Make it easy',
```

```
                        action='store_true')

    return parser.parse_args()


# --------------------------------------------------
def read_csv(fh):
    """Read the CSV input"""

    exercises = []

    for row in csv.DictReader(fh, delimiter=','):
        name = row['exercise']
        low, high = row['reps'].split('-')
        exercises.append((name, int(low), int(high)))

    return exercises


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

    args = get_args()
    random.seed(args.seed)
    exercises = read_csv(args.file)
    table = []

    for name, low, high in random.sample(exercises, k=args.num_exercises):
        if args.easy:
            low = int(low / 2)
            high = int(high / 2)

        table.append((name, random.randint(low, high)))

    print(tabulate(table, headers=('Exercise', 'Reps')))


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

Discussion

As usual, I start with my `get_args` first to define what the program expects. Most important is a `file` which is not required since it has a `default` value of the `wod.csv` file, so I make it an optional named argument. I use the `type=argparse.FileType('r')` so I can offload the validation of the argument to `argparse`. The `--seed` and `--num_exercises` options must to be `type=int`, and the `--easy` option is a `True/False` flag.

Reading the WOD file

Since I know I will return a `list` of exercises and low/high ranges, I first set `exercises = []`. I recommended you use the `csv.DictReader` module to parse the CSV files into a list of dictionaries that represent each rows values merged with the column names in the first row. If the file looks like this:

```
$ head -3 wod.csv
exercise,reps
Burpees,20-50
Situps,40-100
```

You can read it like so:

```
>>> import csv
>>> fh = open('wod.csv')
>>> rows = list(csv.DictReader(fh, delimiter=','))
>>> rows[0]
OrderedDict([('exercise', 'Burpees'), ('reps', '20-50')])
```

On line 55-58, I iterate the rows, `split` the `reps` values like `20-50` into a `low` and `high` values, coerce them into `int` values. I want to `return` a `list` of tuples containing the exercise name along with the minimum and maximum reps.

For the purposes of this exercise, you can assume the CSV files you are given will have the correct headers and the reps can be safely converted.

Choosing the exercises

Before I use the `random` module, I need to be sure to set the `random.seed` with any input from the user. The output will be formatted using the `tabulate` module which wants the data as a single `list` of rows to format, so I first create a `table` to hold the chosen exercises and reps. Then I get the workout options and reps from the file (line 69) which looks like this:

```
>>> from pprint import pprint as pp
>>> pp(exercises)
[('Burpees', 20, 50),
```

```
 ('Situps', 40, 100),
 ('Pushups', 25, 75),
 ('Squats', 20, 50),
 ('Pullups', 10, 30),
 ('HSPU', 5, 20),
 ('Lunges', 20, 40),
 ('Plank', 30, 60),
 ('Jumprope', 50, 100),
 ('Jumping Jacks', 25, 75),
 ('Crunches', 20, 30),
 ('Dips', 10, 30)]
```

and can then then use `random.sample` to select some `k` number given by the user from the `exercises`:

```
>>> import random
>>> random.sample(exercises, 3)
[('Dips', 10, 30), ('Jumprope', 50, 100), ('Lunges', 20, 40)]
```

The sampling returns a `list` from `exercises` which holds tuples with three values each, so I can iterate over those tuples and unpack them all on line 72. If `args.easy` is `True`, then I halve the `low` and `high` values.

```
>>> random.randint(5, 10)
6
>>> random.randint(5, 10)
8
```

Printing the table

Then I can `append` to the `table` a new tuple containing the `name` of the exercise and a `randint` (random integer) selected from the range given by `low` and `high`. Finally I can `print` the result of having the `tabulate` module create a text table using the given `headers`. You can explore the documentation of the `tabulate` module to discover the many options the module has.

Chapter 17: Blackjack

What's a games book without a card game? Let's write a Python program called `blackjack.py` that plays an abbreviated game of Blackjack. Your program should accept a `-S|--stand` option (default `18`) for the value to "stand" on (not "hit" or take another card). Your program should also accept two flags (Boolean values) for `-p|--player_hits` and `-d|--dealer_hits` which will be explained shortly. You will need to accept a `-s|--seed` (default `None`) to set `random.seed`. As usual, you will also have a `-h|--help` option for usage statement:

```
$ ./blackjack.py -h
usage: blackjack.py [-h] [-d] [-p] [-S int] [-s int]

Blackjack

optional arguments:
  -h, --help            show this help message and exit
  -d, --dealer_hits     Dealer hits (default: False)
  -p, --player_hits     Player hits (default: False)
  -S int, --stand int   Stand on value (default: 18)
  -s int, --seed int    Random seed (default: None)
```

The program will create a deck of cards by combining symbols "H," "D," "S", and "C" for the suites "hearts," "diamonds," "spades," and "clubs," respectively, with the numbers 2-10 and the letters "A", "J", "Q," and "K". In order to pass the tests, you will need to sort your deck and then use the `random.shuffle` method so that your cards will be in the order the tests expect.

To deal, keep in mind how cards are actually dealt – first one card to each of the players, then one to the dealer, then the players, then the dealer, etc. You might be tempted to use `random.choice` or something like that to select your cards, but you need to keep in mind that you are modeling an actual deck and so selected cards should no longer be present in the deck. If the `--player_hits` flag is present, deal an additional card to the player; likewise with the `--dealer_hits` flag.

When the program runs with no arguments, display the dealer and players hand along with a sum of the values of the cards. In Blackjack, number cards are worth their value, face cards are worth 10, and the Ace will be worth 1 for our game (though in the real game it can alternate between 1 and 11).

```
$ ./blackjack.py -s 1
Dealer [15]: HJ C5
Player [10]: C9 SA
Dealer should hit.
Player should hit.
```

Here we see that both the dealer and player fall below the `--stand` value of `18`.

Run again and have both players hit:

```
$ ./blackjack.py -s 1 -d -p
Dealer [23]: HJ C5 C8
Player [14]: C9 SA D4
Dealer busts.
```

Here the dealer's hand went above 21, so he "busts." The player could stand to hit again, but, of course, need not since the dealer busted.

If we run with a different seed, we see different results:

```
$ ./blackjack.py -s 3
Dealer [19]: HK C9
Player [12]: D3 H9
Player should hit.
```

Here the dealer is recommended to stand because they have more than 18. Run with a higher `--stand` to change that:

```
$ ./blackjack.py -s 3 -S 20
Dealer [19]: HK C9
Player [12]: D3 H9
Dealer should hit.
Player should hit.
```

Now the dealer is recommended to hit, which seems unwise.

After dealing all the required cards and displaying the hands, the code should do (in order):

1. Check if the player has more than 21; if so, print 'Player busts! You lose, loser!' and `exit(0)`
2. Check if the dealer has more than 21; if so, print 'Dealer busts.' and `exit(0)`
3. Check if the player has exactly 21; if so, print 'Player wins. You probably cheated.' and `exit(0)`
4. Check if the dealer has exactly 21; if so, print 'Dealer wins!' and `exit(0)`
5. If the either the dealer or the player has less than 18, you should indicate "X should hit."

Hints:

- Use `itertools.product` to combine the suites and cards to make your deck.

Solution

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

import argparse
import random
import re
import sys
from itertools import product


# --------------------------------------------------
def get_args():
    """get command-line arguments"""

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

    parser.add_argument('-d',
                        '--dealer_hits',
                        help='Dealer hits',
                        action='store_true')

    parser.add_argument('-p',
                        '--player_hits',
                        help='Player hits',
                        action='store_true')

    parser.add_argument('-S',
                        '--stand',
                        help='Stand on value',
                        metavar='int',
                        type=int,
                        default=18)

    parser.add_argument('-s',
                        '--seed',
                        help='Random seed',
                        metavar='int',
                        type=int,
                        default=None)

    return parser.parse_args()
```

```python


# --------------------------------------------------
def bail(msg):
    """print() and exit(0)"""

    print(msg)
    sys.exit(0)


# --------------------------------------------------
def card_value(card):
    """card to numeric value"""

    val = card[1:]
    faces = {'A': 1, 'J': 10, 'Q': 10, 'K': 10}
    return int(val) if val.isdigit() else faces[val] if val in faces else None


# --------------------------------------------------
def test_card_value():
    """Test card_value"""

    assert card_value('HA') == 1

    for face in 'JQK':
        assert card_value('D' + face) == 10

    for num in range(1, 11):
        assert card_value('S' + str(num)) == num


# --------------------------------------------------
def make_deck():
    """Make a deck of cards"""

    suites = list('HDSC')
    values = list(range(2, 11)) + list('AJQK')
    cards = sorted(map(lambda t: '{}{}'.format(*t), product(suites, values)))
    random.shuffle(cards)
    return cards


# --------------------------------------------------
def test_make_deck():
    """Test for make_deck"""
```

```

    deck = make_deck()
    assert len(deck) == 52

    for suite in 'HDSC':
        cards = list(filter(lambda c: c[0] == suite, deck))
        assert len(cards) == 13
        num_cards = list(filter(lambda c: re.match('\d+', c[1:]), deck))


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

    args = get_args()
    stand_on = args.stand
    random.seed(args.seed)
    cards = make_deck()

    p1, d1, p2, d2 = cards.pop(), cards.pop(), cards.pop(), cards.pop()
    player = [p1, p2]
    dealer = [d1, d2]

    if args.player_hits:
        player.append(cards.pop())
    if args.dealer_hits:
        dealer.append(cards.pop())

    player_hand = sum(map(card_value, player))
    dealer_hand = sum(map(card_value, dealer))

    print('Dealer [{:2}]: {}'.format(dealer_hand, ' '.join(dealer)))
    print('Player [{:2}]: {}'.format(player_hand, ' '.join(player)))

    blackjack = 21
    if player_hand > blackjack:
        bail('Player busts! You lose, loser!')

    if dealer_hand > blackjack:
        bail('Dealer busts.')

    if player_hand == blackjack:
        bail('Player wins. You probably cheated.')

    if dealer_hand == blackjack:
        bail('Dealer wins!')
```

```

    if dealer_hand < stand_on:
        print('Dealer should hit.')

    if player_hand < stand_on:
        print('Player should hit.')


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

Chapter 18: Family Tree

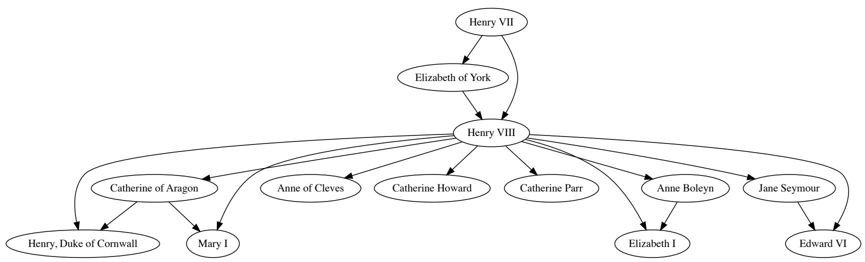


Figure 9: Partial Tudor family tree

Write a program called `tree.py` that will take an input file as a single positional argument and produce an `-o|--outfile` graph of the family tree described therein. There should be a '-v|–view' flag to have the image opened when done (default `False`). The program should produce a usage with no arguments or if given `-h|--help` flags:

```
$ ./tree.py
usage: tree.py [-h] [-o str] [-v] FILE
tree.py: error: the following arguments are required: FILE
$ ./tree.py -h
usage: tree.py [-h] [-o str] [-v] FILE

Display a family tree

positional arguments:
  FILE                  File input

optional arguments:
  -h, --help            show this help message and exit
  -o str, --outfile str
                        Output filename (default: )
  -v, --view            View image (default: False)
```

The input file can have only three kinds of statements:

1. `INITIALS = Full Name`
2. `INITIALS married INTIALLS`
3. `INITIALS and INITIALS begat INITIALS[, INITIALS...]`

Use the `graphviz` module to generate a graph like the one shown above from the following input:

```
$ cat tudor.txt
```

```
H7 = Henry VII
EOY = Elizabeth of York
H8 = Henry VIII
COA = Catherine of Aragon
AB = Anne Boleyn
JS = Jane Seymour
AOC = Anne of Cleves
CH = Catherine Howard
CP = Catherine Parr
HDC = Henry, Duke of Cornwall
M1 = Mary I
E1 = Elizabeth I
E6 = Edward VI

H7 married EOY
H7 and EOY begat H8
H8 married COA
H8 married AB
H8 married JS
H8 married AOC
H8 married CH
H8 married CP
H8 and COA begat HDC, M1
H8 and AB begat E1
H8 and JS begat E6
```

If given no `-o|--outfile`, the default should be the name of the input file with `.gv` appended:

```
$ ./tree.py tudor.txt
Done, see output in "tudor.txt.gv".
```

Technically your input file doesn't need the "INITIALS = Full Name" lines. Those are just to make it a bit easier to spell out all the marrying and begetting that people do. Here is a very simple tree:

```
$ cat joanie.txt
Joanie married Chachi
$ ./tree.py joanie.txt
Done, see output in "joanie.txt.gv".
```

Joanie Loves Chachi

Graphs

You are creating a graph that describes the relationships among entities. Graphs have "nodes" (or "vertices") and "edges" that connect them. In the phrase "My best friend's sister's boyfriend's brother's girlfriend heard from this guy who knows this kid who's going with the girl who saw Ferris pass out at 31 Flavors last night," there are 10 nodes:

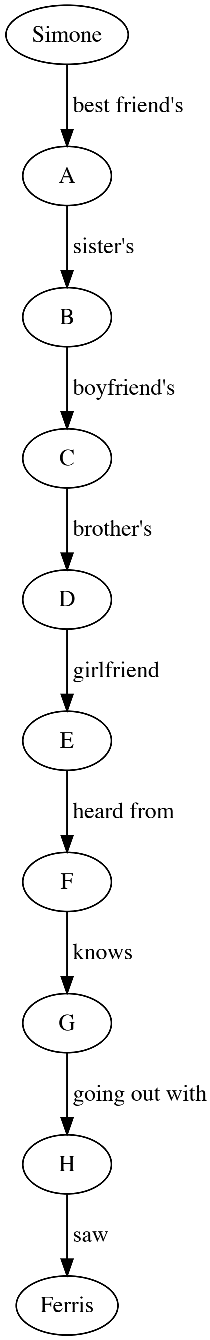
1. the speaker (Simone)
2. my best friend
3. sister
4. boyfriend
5. brother
6. girlfriend
7. this guy
8. this kid
9. the girl
10. Ferris

If we call all the unnamed people by a letter like `A`, then we could write code to visualize this graph:

```
from graphviz import Digraph
nodes = ('Simone', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'Ferris')
labels = ("best friend's", "sister's", "boyfriend's", "brother's",
          "girlfriend", "heard from", "knows", "going out with", "saw")

dot = Digraph('Simone')
for k in range(len(nodes) - 1):
    dot.edge(nodes[k], nodes[k+1], label=' ' + labels[k])

dot.render('simone.gv', view=True)
```


Simone
best friend's
A
sister's
B
boyfriend's
C
brother's
D
girlfriend
E
heard from
F
knows
G
going out with
H
saw
Ferris

Solution

```
#!/usr/bin/env python3
"""Display a family tree"""

import argparse
import os
import re
from dire import die
from graphviz import Digraph


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

    parser = argparse.ArgumentParser(
        description='Display a family tree',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file',
                        metavar='FILE',
                        type=argparse.FileType('r'),
                        help='File input')

    parser.add_argument('-o',
                        '--outfile',
                        help='Output filename',
                        metavar='str',
                        type=str,
                        default='')

    parser.add_argument('-v', '--view', help='View image', action='store_true')

    return parser.parse_args()


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

    args = get_args()
    fh = args.file
    out_file = args.outfile or os.path.basename(fh.name) + '.gv'

```

```
    nodes, edges = parse_tree(fh)

    if not nodes and not edges:
        die('No nodes or edges in "{}".'.format(fh.name))

    dot = Digraph(comment='Tree')

    # keys are initials which we don't need
    for _, name in nodes.items():
        dot.node(name)

    for n1, n2 in edges:
        # see if node has alias in nodes, else use node itself
        n1 = nodes.get(n1, n1)
        n2 = nodes.get(n2, n2)
        dot.edge(n1, n2)

    dot.render(out_file, view=args.view)

    print('Done, see output in "{}".'.format(out_file))


# --------------------------------------------------
def parse_tree(fh):
    """parse input file"""

    name_patt = r'(.+)\s*=\s*(.+)'
    married_patt = r'(.+)\s+married\s+(.+)'
    begat_patt = r'(.+)\s+and\s+(.+)\s+begat\s+(.+)'

    edges = set()
    nodes = {}

    for line in fh:
        name_match = re.match(name_patt, line)
        begat_match = re.match(begat_patt, line)
        married_match = re.match(married_patt, line)

        if name_match:
            initials, name = name_match.groups()
            nodes[initials.strip()] = name.strip()
        elif married_match:
            p1, p2 = married_match.groups()
            edges.add((p1.strip(), p2.strip()))
        elif begat_match:
            p1, p2, begat = begat_match.groups()
```

```
            children = re.split(r'\s*,\s*', begat)
            for parent in p1, p2:
                for child in children:
                    edges.add((parent.strip(), child.strip()))

    return nodes, edges


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

Discussion

There are two main parts to the program:

1. Parsing the input file for nodes and edges
2. Using nodes and edges with the `graphviz` module to produce a graph

Parsing input file

Since there are only three types of statements we expect in the input file, I will create three regular expressions to match each.

Parsing name line

The first allowed expression is something along the lines of "INITIALS = Full Name":

```
>>> line = 'H7 = Henry VII'
```

I could look for an equal sign in the line and `split` it if found:

```
>>> if '=' in line:
...     line.split('=')
...
['H7 ', ' Henry VII']
```

Or I could `import re` to bring in the regular expression module and write a pattern to match "something, an equal sign, something else". The dot `.` means "anything" and I can use `.+` to say "one or more of anything". This regex matches the whole line:

```
>>> import re
>>> re.match('.+', line)
<re.Match object; span=(0, 14), match='H7 = Henry VII'>
```

There may or may not be whitespace around the equal signs. Whitespace is `\s`, and we can use `\s*` to indicate "zero or more whitespace". The equal sign is a literal `=`, and then more optional whitespace. This regex takes up to the space after the `=`:

```
>>> re.match('.+\s*=\s*', line)
<re.Match object; span=(0, 5), match='H7 = '>
```

We can finish it off with the same pattern at the beginning and put parentheses `()` around the parts of the pattern we want to capture:

```
>>> re.match('(.+)\s*=\s*(.+)', line)
<re.Match object; span=(0, 14), match='H7 = Henry VII'>
>>> match = re.match('(.+)\s*=\s*(.+)', line)
```

```
>>> match.groups()
('H7 ', 'Henry VII')
```

There is some trailing whitespace around the first group, so I'll be sure to `strip` it to remove spaces from the beginning and end.

Parsing married line

The "A married B" line can be found in a very similar fashion. Instead of `=` we can substitute `married`:

```
>>> line = 'H8 married COA'
>>> re.match(r'(.+)\s+married\s+(.+)', line)
<re.Match object; span=(0, 14), match='H8 married COA'>
```

And get the parts from `groups`:

```
>>> match = re.match(r'(.+)\s+married\s+(.+)', line)
>>> match.groups()
('H8', 'COA')
```

Parsing begat line

The previous patterns could have just as easily been handled by looking for the `=` or `married` in the `line` and using `line.split` on the string. The "begat" line is the most complicated and really makes use of regular expressions.

The pattern still looks similar:

```
>>> line = 'H8 and COA begat HDC, M1'
>>> re.match(r'(.+)\s+and\s+(.+)\s+begat\s+(.+)', line).groups()
('H8', 'COA', 'HDC, M1')
```

The parents are groups 1 and 2, and the children (group 3) can be split with another regex:

```
>>> re.split('\s*,\s*', 'HDC, M1')
['HDC', 'M1']
```

Building the graph

I chose to represent my graph with two structures:

1. nodes: a `dict` from initials to full names
2. edges: a `set` of 2-tuples of node names

I said in the intro that the "INTIALS = Full Name" was optional, and so technically the "nodes" can be empty. You saw in the `simone.py` example that

Graphviz will automatically create nodes as needed when you add edges that name nodes that do not yet exist.

When parsing the input file, I decided to create a `parse_tree` function that takes the input file handle, reads it line-by-line, and tries to match each line to the three regular expressions described above. If I match the "initials" line, I add the initials and names to the `nodes` dictionary. If I find a "married" line, I add the two nodes to the `edges` set. If I find a "begat" line, I add an edge from each parent to each child.

Using graphviz

The `graphviz` module is an interface to the `graphviz` program which is a stand-alone program you can use directly. Mostly the Python module makes it fairly easy to write the graph structure that `graphviz` expects, giving us an interface to add nodes and edges using the objects provided by the module. Here is a very simple tree:

```
>>> from graphviz import Digraph
>>> dot = Digraph()
>>> dot.edge('Joanie', 'Chachi')
>>> dot.render(view=True)
'Digraph.gv.pdf'
```

To make a more complicated graph, I added the full names from my `nodes` dictionary, and then use those full names to expand the initials from the `edges`, if present. In the end, the code isn't much more complicated that these few lines.

Chapter 19: Gematria: Numeric encoding of text

Write a Python program called `gematria.py` that will numerically encode each word in a given text. The name of this program comes from gematria, a system for assigning a number to a word by summing the numeric values of each of the letters as defined by the Mispar godol (https://en.wikipedia.org/wiki/Gematria). For English characters, we can use the ASCII table (https://en.wikipedia.org/wiki/ASCII). Python provides these value throug the `ord` function to convert a character to its "ordinal" (order in the ASCII table) value as well as the `chr` function to convert a number to its "character."

```
>>> ord('A')
65
>>> ord('a')
97
>>> chr(88)
'X'
>>> chr(112)
'p'
```

To implement an ASCII version of gematria in Python, for each word in a text we need to turn each letter into a number and add them all together. So, to start, note that Python can use a `for` loop to cycle through all the characters in a string:

```
>>> for char in 'python':
...     print(ord(char))
...
112
121
116
104
111
110
```

We've seen before how you can put a `for` loop inside brackets `[]` for a list comprehension. Do that and then `sum` the list.

The program should print a usage if given no arguments or the `-h|--help` flag:

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

Gematria
```

```
positional arguments:
  str         Input text or file

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

The text may be given on the command line:

```
$ ./gematria.py 'foo bar baz'
324 309 317
```

Or in a file:

```
$ ./gematria.py ../inputs/fox.txt
289 541 552 333 559 444 321 448 314
```

Hints:

- You'll want to read the input line-by-line because the tests are expecting lines of output where each word has been encoded.
- Can you write a function that can encode just one word? E.g, "gematria" = 842.
- Be sure you only encode the words themselves and not any punctuation that might be next to a word. E.g., if you use `str.split` to break text on whitespaces, quotes/commas/periods and such will still be attached to the words. Additionally, you should remove any internal punctuation like apostrophes. Maybe look into the `re` module to use regular expressions.
- Now can you apply that function to each word in a line of text?

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='str', help='Input text or file')

    args = parser.parse_args()

    if os.path.isfile(args.text):
        args.text = open(args.text).read()

    return args


# --------------------------------------------------
def word2num(word):
    """Sum the ordinal values of all the characters"""

    word = re.sub('[^a-zA-Z0-9]', '', word)
    return str(sum(map(ord, word)))


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

    args = get_args()
    text = args.text

    for line in text.splitlines():
        print(' '.join(map(word2num, line.split())))
```

```


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

Discussion

The `text` argument for the program might be taken directly from the command line or from a named file. I chose to handle inside the `get_args` the reading of a file argument so that by the time I call `args = get_args()` I have the actual `text` I need to process.

Reading lines of text

As mentioned in the description of the program, the test suite is looking for the lines of input text to be maintained in the output. It's straightforward to `read` an `open` file line-by-line:

```
>>> file = '../inputs/spiders.txt'
>>> for line in open(file):
...     print(line, end='')
...
Don't worry, spiders,
I keep house
casually.
```

If I've taken the `text` from a file, then it's all just now just one string and a `for` loop on a string will iterate over each *character* not each line.

```
>>> text = open(file).read()
>>> text
'Don't worry, spiders,\nI keep house\ncasually.\n'
>>> for char in text:
...     print(char, end='-')
...
D-o-n-'-t- -w-o-r-r-y-,- -s-p-i-d-e-r-s-,-
-I- -k-e-e-p- -h-o-u-s-e-
-c-a-s-u-a-l-l-y-.-
```

So instead we can use `str.splitlines()` to break `text` on the newlines:

```
>>> for line in text.splitlines():
...     print(line)
...
Don't worry, spiders,
I keep house
casually.
```

List comprehensions vs map

Like several other programs, we now are left with applying some function to each member of a list. That is, we want to turn each word into a number, and we've seen there are several ways to go about this. I will focus on two methods which I use twice each: list comprensions and the `map` function.

Encoding one word

First let's take just one word. We've seen how we can use `ord` to turn one character into a number:

```
>>> ord('g')
103
```

We can use a list comprehension to do this for every character in a word:

```
>>> word = 'gematria'
>>> [ord(char) for char in word]
[103, 101, 109, 97, 116, 114, 105, 97]
```

And then the `sum` function will add those for us:

```
>>> sum([ord(char) for char in word])
842
```

We can do the same thing with `map`. The first argument to `map` is a function which is applied to every element in the second argument which must be something *iterable* like a `list` or a generator. Because the second argument is iterable, we don't have to spell out `for char in`.

```
>>> map(ord, word)
<map object at 0x105c3c550>
```

We see this because `map` is a "lazy" function that doesn't actually produce results until they are actually required. For purposes of viewing in the REPL only, we can use `list` to see the values. (You do not have to use `list` in your actual code!)

```
>>> list(map(ord, word))
[103, 101, 109, 97, 116, 114, 105, 97]
```

And now we can `sum` that. Note that `sum` will consume the `map` object, so we don't have to use `list`. To me, this is an extremly clean bit of code:

```
>>> sum(map(ord, word))
842
```

Since ultimately I will be giving these numbers to `print` in a way that will expect strings, I will additionally coerce the number using `str`. We can put this into a function either writing it with `lambda` on one line:

```
>>> word2num = lambda word: str(sum(map(ord, word)))
>>> word2num('gematria')
'842'
```

Or using `def`:

```
>>> def word2num(word):
...     return str(sum(map(ord, word)))
...
>>> word2num('gematria')
'842'
```

Finding the words

Just above we were applying the `ord` function to every character in a word. Now we want to apply our new `word2num` function to every word in a line. I hope you see it's the exact same problem, and both list comprehensions and `map` will serve equally.

So how to find "words" in a line? We know that we can use `str.split()` to break each `line` into `words`:

```
>>> for line in text.splitlines():
...     words = line.split()
...     print(words)
...
['Don’t', 'worry,', 'spiders,']
['I', 'keep', 'house']
['casually.']
```

There's a small problem, though. Notice that we get `worry,` and not `worry` and `spiders,` instead of `spiders`. We don't want to encode the punctuation that is still attached to the words. Also, let's just say we also don't want to encode the apostrophe in `Don't`. So how can we remove these offending characters? The first step is in identifying what they are. If we say "remove anything that is not the a letter in the set A-Z or a number in the list 0-9", that helps. We can use regular expressions to describe that exactly using `[]` to create a "character class" and putting the allowed characters in there. Notice this filters out the unwanted characters:

```
>>> import re
>>> re.findall('[a-zA-Z0-9]', "Don't")
['D', 'o', 'n', 't']
>>> re.findall('[a-zA-Z0-9]', "spiders,")
['s', 'p', 'i', 'd', 'e', 'r', 's']
```

Or we could use the `re.sub` function to "substitute" any matches. We can negate our character class by putting a caret (`^`) just *inside* the start of the brackets to indicate we want to find anything that's *not* an English alphabet character or an Arabic number and replace it with the empty string:

```
>>> re.sub('[^a-zA-Z0-9]', '', "Don't")
'Dont'
>>> re.sub('[^a-zA-Z0-9]', '', "spiders,")
'spiders'
```

Let's put that into a function:

```
>>> def clean(word):
...     return re.sub('[^a-zA-Z0-9]', '', word)
...
>>> clean("Don't")
'Dont'
>>> clean("spiders,")
'spiders'
```

Compare this with the earlier version to see that we now have "clean" words to encode:

```
>>> for line in text.splitlines():
...     words = map(clean, line.split())
...     print(list(words))
...
['Dont', 'worry', 'spiders']
['I', 'keep', 'house']
['casually']
```

For convenience, let's update the `word2num` function to use that:

```
def word2num(word):
    word = re.sub('[^a-zA-Z0-9]', '', word)
    return str(sum(map(ord, word)))
>>> word2num('spiders,')
'762'
>>> word2num('spiders')
'762'
```

Encoding all words

So we're finally to the point where we have lines of text and lists of words to encode. As we've seen, a list comprehension works adequately:

```
>>> words = ['Dont', 'worry', 'spiders']
>>> [word2num(word) for word in words]
['405', '579', '762']
```

But `map` is cleaner:

```
>>> list(map(word2num, words))
['405', '579', '762']
```

All that is left is to `print` the encoded words back out:

```
>>> for line in text.splitlines():
...     print(' '.join(map(word2num, line.split())))
...
405 579 762
73 421 548
862
```

Chapter 20: I Get Histy

Write a Python program called `histy.py` that takes a single positional argument that may be plain text or the name of a file to read for the text. Count the frequency of each character (not spaces) and print a histogram of the data. By default, you should order the histogram by the characters but include `-f|--frequency_sort` option to sort by the frequency (in descending order). Also include a `-s|--symbol` option (default `|`) to represent a mark in the histogram, a `-m|--minimum` option (default `1`) to include a character in the output, a `-w|--width` option (default `70`) to limit the size of the histogram, and a `-i|--case_insensitive` flag to force all input to uppercase.

When run with no arguments or the `-h|--help` flag, print a usage:

```
$ ./histy.py
usage: histy.py [-h] [-s str] [-m int] [-w int] [-i] [-f] str
histy.py: error: the following arguments are required: str
$ ./histy.py -h
usage: histy.py [-h] [-s str] [-m int] [-w int] [-i] [-f] str

Histogrammer

positional arguments:
  str                   Input text or file

optional arguments:
  -h, --help            show this help message and exit
  -s str, --symbol str  Symbol for marks (default: |)
  -m int, --minimum int
                        Minimum frequency to print (default: 1)
  -w int, --width int   Maximum width of output (default: 70)
  -i, --case_insensitive
                        Case insensitive search (default: False)
  -f, --frequency_sort  Sort by frequency (default: False)
```

Error out if the `--symbol` is not a single character:

```
$ ./histy.py -s XX foobar
usage: histy.py [-h] [-s str] [-m int] [-w int] [-i] [-f] str
histy.py: error: --symbol "XX" must be one character
```

Accept text on the command line. Note that the default sorting should be *case-insensitive* so that `I` follows immediately here after `h`:

```
$ ./histy.py "I don't want the world, I just want your half."
a      3 |||
d      2 ||
e      1 |
```

```
f       1 |
h       2 ||
I       2 ||
j       1 |
l       2 ||
n       3 |||
o       3 |||
r       2 ||
s       1 |
t       5 |||||
u       2 ||
w       3 |||
y       1 |
```

Or an input file:

```
$ ./histy.py -i ../inputs/fox.txt -m 2
E       3 |||
H       2 ||
O       4 ||||
R       2 ||
T       2 ||
U       2 ||
```

Note that short flags can be combined:

```
$ ./histy.py ../inputs/const.txt -fim 100 -w 50 -s '#'
E      50 ##################################################
T      36 ####################################
S      26 ##########################
O      26 ##########################
A      26 ##########################
N      25 #########################
I      23 #######################
R      21 #####################
H      19 ###################
L      14 ##############
D      12 ############
C      11 ###########
F       9 #########
U       8 ########
P       7 #######
M       7 #######
B       5 #####
Y       4 ####
V       4 ####
G       4 ####
W       3 ###
```

Counting, filtering, scaling and sorting text

I chose to create a single function called `count(text, minimum=0, width=0, frequency_sort=False)` that filters the input text to only characters, counts the letter frequencies, filters by a minimum count, scales the numbers down by a maximum width, and sorts the values by the character or by the frequency. I put the following function inside my `histy.py` program that verifies that I get expected results. I encourage you to do the same and then run `pytest -v histy.py` on your program to check your function:

```
def test_count():
    """Test count"""

    text = '"ab,Bc CC: dd_d-d"'
    assert count(text) == [('a', 1), ('B', 1), ('b', 1), ('C', 2), ('c', 1),
                           ('d', 4)]

    assert count(text, minimum=2) == [('C', 2), ('d', 4)]

    assert count(text, frequency_sort=True) == [('d', 4), ('C', 2), ('c', 1),
                                                ('b', 1), ('a', 1), ('B', 1)]

    assert count(text, frequency_sort=True, minimum=2) == [('d', 4), ('C', 2)]

    assert count(text, width=3) == [('a', 0), ('B', 0), ('b', 0), ('C', 1),
                                    ('c', 0), ('d', 3)]
```

Once you have the results of this, you need to create the histogram by printing the `--symbol` the number of times shown for each character.

Hints

- Put all your input validation into `get_args` and use `parser.error` to error out
- A regular expression plus the `filter` function can help you remove any characters from the input text that are not in the set of ASCII characters "a-zA-Z"
- Look at `collections.Counter` for counting the characters

Solution

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

import argparse
import os
import re
from collections import Counter


# --------------------------------------------------
def get_args():
    """get command-line arguments"""

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

    parser.add_argument('text', metavar='str', help='Input text or file')

    parser.add_argument('-s',
                        '--symbol',
                        help='Symbol for marks',
                        metavar='str',
                        type=str,
                        default='|')

    parser.add_argument('-m',
                        '--minimum',
                        help='Minimum frequency to print',
                        metavar='int',
                        type=int,
                        default=1)

    parser.add_argument('-w',
                        '--width',
                        help='Maximum width of output',
                        metavar='int',
                        type=int,
                        default=70)

    parser.add_argument('-i',
                        '--case_insensitive',
                        help='Case insensitive search',
```

```
                        action='store_true')

    parser.add_argument('-f',
                        '--frequency_sort',
                        help='Sort by frequency',
                        action='store_true')

    args = parser.parse_args()

    if os.path.isfile(args.text):
        args.text = open(args.text).read()

    if args.case_insensitive:
        args.text = args.text.upper()

    if len(args.symbol) != 1:
        parser.error('--symbol "{}" must be one character'.format(args.symbol))

    return args


# --------------------------------------------------
def count(text, minimum=0, width=0, frequency_sort=False):
    """Count characters in text"""

    freqs = Counter(re.findall(r'[a-zA-Z]', text))

    if minimum > 1:
        freqs = {k:v for k,v in freqs.items() if v >= minimum}

    high = max(freqs.values())
    if width > 0 and high > width:
        scale = width / high
        freqs = {k:int(v * scale) for k,v in freqs.items()}

    if frequency_sort:
        return list(
            map(lambda t: (t[1], t[0]),
                sorted([(v, k) for k, v in freqs.items()], reverse=True)))
    else:
        return list(
            map(lambda t: (t[1], t[2]),
                sorted([(k.lower(), k, v) for k, v in freqs.items()])))


# --------------------------------------------------
```

```
def test_count():
    """Test count"""

    text = '"ab,Bc CC: dd_d-d"'
    assert count(text) == [('a', 1), ('B', 1), ('b', 1), ('C', 2), ('c', 1),
                           ('d', 4)]

    assert count(text, minimum=2) == [('C', 2), ('d', 4)]

    assert count(text, frequency_sort=True) == [('d', 4), ('C', 2), ('c', 1),
                                                ('b', 1), ('a', 1), ('B', 1)]

    assert count(text, frequency_sort=True, minimum=2) == [('d', 4), ('C', 2)]

    assert count(text, width=3) == [('a', 0), ('B', 0), ('b', 0), ('C', 1),
                                    ('c', 0), ('d', 3)]


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

    args = get_args()
    freqs = count(args.text, args.minimum, args.width, args.frequency_sort)

    for c, num in freqs:
        print('{} {:6} {}'.format(c, num, args.symbol * num))


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

Discussion

As suggested, I wrote my `get_args` to handle all the user input validation. It's easy enough to check, for instance, if the `text` argument is actually a file and `open/read` it so as to return exactly what the rest of the program requires. I can also see if the `--symbol` is not exactly one character and use `parser.error` to print an error with the usage and then exit with a non-zero status.

Inside my `main`, I pass most of the arguments to the `count` function I described in the intro which is where most of the logic of the program is contained. If you wrote several functions to do each part or if you simply put all the logic in the `main`, that's fine (as long as you are passing the test suite). The advantage to me is that I can write a `test_count` function where I pass in some known text and ensure that I'm getting back the filtered, sorted results I expect.

The first thing to notice about the `test_count` function is the text that it passes in:

```
>>> text = '"ab,Bc CC: dd_d-d"'
```

I only want to count the alphabetic characters, so how can I remove all the other stuff? I love to use regular expressions for this. I create a character class with `r'[a-zA-Z]'` (where `r''` creates a "raw" string), the brackets `[]` enclose the class, and `a-z` is the range of letters from lowercase `a` to lowercase `z` and then `A-Z` is the uppercase letters. The `re.findall` method will return a `list` of matching characters:

```
>>> import re
>>> re.findall(r'[a-zA-Z]', text)
['a', 'b', 'B', 'c', 'C', 'C', 'd', 'd', 'd', 'd']
```

Which is exactly what we need for `collections.Counter`:

```
>>> freqs = Counter(re.findall(r'[a-zA-Z]', text))
>>> freqs
Counter({'d': 4, 'C': 2, 'a': 1, 'b': 1, 'B': 1, 'c': 1})
```

So, in one line of code we filtered and counted the input text. Next, we can remove letters that are found too infrequently. A dictionary comprehension works well if we use `freq.items()` to get a list of tuples and only take those above the minimum:

```
>>> minimum = 2
>>> freqs = {k:v for k,v in freqs.items() if v >= minimum}
>>> freqs
{'C': 2, 'd': 4}
```

We need to find the highest value which is what the `max` function will do. Note that you should not call your variable `max` as it will then *overwrite the actual function* called `max`!

```
>>> high = max(freqs.values())
>>> highest
4
```

If the `highest` value is greater than the `width` argument, we should scale the values down. Again a dictionary comprehension works well:

```
>>> freqs = {k:int(v * scale) for k,v in freqs.items()}
>>> freqs
{'C': 1, 'd': 3}
```

Finally we need to sort the values either by the characters (which are the `keys`) or their frequencies (which are the `values`). We can use the `sorted` function on the `freqs.items()` to sort by key and value, but that would be case-senstive. To sort without regard to case, we need to convert the keys to upper- or lowercase first, but we still want to display the proper case. Note that `dict.items` returns a list of tuples:

```
>>> freqs.items()
dict_items([('C', 1), ('d', 3)])
```

We could add another field to the tuple that is the first value in lowercase:

```
>>> s = list(map(lambda t: (t[0].lower(), t[0], t[1]), freqs.items()))
>>> s
[('c', 'C', 1), ('d', 'd', 3)]
```

Then call `sorted` on that. We can then remove the first field from each tuple:

```
>>> list(map(lambda t: (t[1], t[2]), s))
[('C', 1), ('d', 3)]
```

The other sort is by values, so we can use a dictionary comprehension to reverse the keys and values before calling `sorted`:

```
>>> s = sorted([(v, k) for k, v in freqs.items()], reverse=True)
>>> s
[(3, 'd'), (1, 'C')]
```

And then reverse the tuples to put them back with `(character, count)`:

```
>>> list(map(lambda t: (t[1], t[0]), s))
[('d', 3), ('C', 1)]
```

Now it's a matter to `print` each item. I can use the `*` operator to repeat the `symbol` argument by the scaled count:

```
>>> symbol = '|'
>>> for c, num in freqs:
...     print('{} {:6} {}'.format(c, num, symbol * num))
...
d      3 |||
C      1 |
```

Further

- Turn the histogram 90 degrees so that the characters are listed on the bottom and the bars go up

Chapter 21: Mommy's Little (Crossword) Helper

Write a Python program called `helper.py` that finds all words matching a given `-p|--pattern` such as one might use to complete a crossword puzzle to find words matching from a given `-w|--wordlist` (default `/usr/share/dict/words`). E.g., all 5-letter words with a "t" as the second character and ending in "ed". I could do this on the command line like so:

```
$ grep '^.t' /usr/share/dict/words | grep 'ed$' | awk 'length($0) == 5'
steed
```

Here is how a program could look:

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

Crossword helper

positional arguments:
  str                   The pattern to search

optional arguments:
  -h, --help            show this help message and exit
  -w str, --wordlist str
                        Wordlist to search (default: /usr/share/dict/words)
```

We'll use an underscore (`_`) to indicate a blank and supply any known letters, e.g., the example above would be `_t_ed`:

```
$ ./helper.py _t_ed
  1: steed
```

Or 6-letter words beginning with "ex" and ending in "s":

```
$ ./helper.py ex___s
  1: excess
  2: excuss
  3: exitus
  4: exodos
  5: exodus
  6: exomis
```

Hints

- If you know about regular expressions, that is a natural way to solve this problem. See how elegantly you can solve the problem.
- Even if you do know how to solve use regexes, try solving without them.

Solution

```
#!/usr/bin/env python3
"""Crossword helper"""

import argparse
import os
import re
import sys
from typing import List, TextIO


# --------------------------------------------------
def get_args() -> argparse.Namespace:
    """Get command-line arguments"""

    parser = argparse.ArgumentParser(
        description='Crossword helper',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('pattern', metavar='str', help='The pattern to search')

    parser.add_argument('-w',
                        '--wordlist',
                        help='Wordlist to search',
                        metavar='str',
                        type=argparse.FileType('r'),
                        default='/usr/share/dict/words')

    return parser.parse_args()


# --------------------------------------------------
def regex_solution(pattern: str, wordlist: TextIO) -> List[str]:
    """Using regular expressions"""

    regex = r'\b{}\b'.format(pattern.replace('_', '.'))
    return re.findall(regex, wordlist.read())


# --------------------------------------------------
def manual_solution(pattern: str, wordlist: TextIO) -> List[str]:
    """Not using regular expressions"""

    letters = [t for t in enumerate(pattern) if t[1] != '_']
```

```
    #letters = filter(lambda t: t[1] != '_', enumerate(pattern))
    wanted_len = len(pattern)
    words = []

    for word in wordlist.read().split():
        if len(word) == wanted_len and all(
            [word[i] == char for i, char in letters]):
            words.append(word)

    return words


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

    args = get_args()
    words = regex_solution(args.pattern, args.wordlist)
    #words = manual_solution(args.pattern, args.wordlist)

    if words:
        for i, word in enumerate(words, start=1):
            print('{:3}: {}'.format(i, word))
    else:
        print('Found no words matching "{}".'.format(args.pattern))


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

Discussion

I rely on `argparse` so very much, and this example is no different. I define a `pattern` as a positional argument and a the `--wordlist` option as a readable file type that has a reasonable default. With this definition, I can safely `read()` the word list argument to get the entire contents of the file. I decided to show two ways to solve the problem, both of which take the `pattern` (a `str`) and the `wordlist` as an open file handle (`TextIO`).

Regular Expressions

The `regex_solution` could be one line, but I wrote it in two for readability. The `pattern` uses underscores (`_`) to indicate a character. In regular expressions, the `.` is how we represent one of any character, so we can use `str.replace` to change those:

```
>>> pattern = '_t_ed'
>>> pattern.replace('_', '.')
'.t.ed'
```

I could have chosen to use `wordlist.read().split()` to get a list of each word (`List[str]`) and then used a pattern that anchors the above to the beginning (`^`) and end (`$`) of each word:

```
>>> regex = '^{}$'.format(pattern.replace('_', '.'))
>>> regex
'^.t.ed$'
```

So that I could apply this to each word individually:

```
>>> import re
>>> wordlist = open('/usr/share/dict/words')
>>> [w for w in wordlist.read().split() if re.search(regex, w)]
['steed']
```

That works just fine, but I chose instead to use the "word boundary" metacharacter `\b` to anchor the pattern to the beginning and end of each word so that I could `read()` the entire file as a stream. Note that it's important to enclose this pattern in a "raw" string with `r''` so that the `\b` is interpreted correctly. The `re.findall` method will return every match of the given pattern in a body of text.

```
>>> wordlist = open('/usr/share/dict/words')
>>> regex = r'\b{}\b'.format(pattern.replace('_', '.'))
>>> re.findall(regex, wordlist.read())
['steed']
```

If I needed to get each `match` object, maybe to use the position of the match or whatnot, I would not use `re.findall`, but for this purpose it was exactly the right function.

Manual Matching

Trying to solve this without regular expressions can give you a real appreciation for exactly how much time regular expressions can save us. For my manual solution, I thought I would use two criteria to find matching words:

1. The length of a word matches the length of the pattern
2. The word has characters matching in the same positions as in the pattern

For the second point, I thought a list of tuples show the position of each character that is not an underscore would be perfect. We can use `enumerate` on any list to give us position and value of each element. Note that I only need to use `list` here to force the REPL to evaluate the generator.

```
>>> pattern = '_t_ed'
>>> list(enumerate(pattern))
[(0, '_'), (1, 't'), (2, '_'), (3, 'e'), (4, 'd')]
```

You don't need to use `list` in your code unless you will need to iterate the generated list more than once. This is because generators are lazy, hence they won't generate their values unless forced, and they can only be iterated once:

```
>>> g = enumerate(pattern)
>>> list(g)
[(0, '_'), (1, 't'), (2, '_'), (3, 'e'), (4, 'd')]
>>> list(g)
[]
```

I only care about the positions of the characters that are *not* underscores, so I can `filter` out the underscores. One limitation to the `lambda` is that is cannot unpack the tuple, so I use `t` to remind me of the type and use `[1]` to indicate the second part of the tuple which is the character. The `filter` will only allow those list elements to pass through for which the predicate (`lambda`) returns something "truthy."

```
>>> list(filter(lambda t: t[1] != '_', enumerate(pattern)))
[(1, 't'), (3, 'e'), (4, 'd')]
```

If you don't care for `filter`, the same idea can be done with a list comprehension:

```
>>> [t for t in enumerate(pattern) if t[1] != '_']
[(1, 't'), (3, 'e'), (4, 'd')]
```

One of the nicer things about this syntax is that you *can* unpack the tuple (but we need to return the tuple all the same):

```
>>> [(i, char) for i, char in enumerate(pattern) if char != '_']
[(1, 't'), (3, 'e'), (4, 'd')]
```

For this solution, I do want to look at each word individually, so I call `for word in wordlist.read().split()` and then check first for the length. The second condition is a little trickier and worth exploring. I decided to use the `all` function to find if *all* the characters in the `pattern` are the same in the `word`. Here I use the list comprehension syntax to unpack the list of tuples in `letters` to get their positions (`i`) and characters (`char`) and check if the `word` at that position matches the character (`word[i] == char`):

```
>>> word = 'steed'
>>> [word[i] == char for i, char in letters]
[True, True, True]
>>> word = 'steer'
>>> [word[i] == char for i, char in letters]
[True, True, False]
```

And then `all` will reduce it to a single value:

```
>>> word = 'steed'
>>> all([word[i] == char for i, char in letters])
True
>>> word = 'steer'
>>> all([word[i] == char for i, char in letters])
False
```

If both conditions are `True` (same length, all characters the same), then I `append` the `word` to the list of `words` I finally `return` from the function.

Summary

All that is left is to check if any words matched. If so, we print them out, numbered and nicely aligned; otherwise, we let the user know that no matches were found. I hope you tried solving this problem with and without regular expressions as there is much to learn by each method.

Chapter 22: Kentucky Friar

Write a Python program called `friar.py` that reads some input text from a single positional argument on the command line (which could be a file to read) and transforms the text by dropping the "g" from words two-syllable words ending in "-ing" and also changes "you" to "y'all". Be mindful to keep the case the same on the first letter, e.g, "You" should become "Y'all," "Hunting" should become "Huntin"'.

Figure 10: The friar is fixin' ta do some cookin'!

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

Southern fry text

positional arguments:
  str         Input text or file

optional arguments:
  -h, --help  show this help message and exit
$ ./friar.py you
y'all
$ ./friar.py Fishing
Fishin'
```

```
$ ./friar.py string
string
$ cat tests/input1.txt
So I was fixing to ask him, "Do you want to go fishing?" I was dying
to go for a swing and maybe do some swimming, too.
$ ./friar.py tests/input1.txt
So I was fixin' to ask him, "Do y'all want to go fishin'?" I was dyin'
to go for a swing and maybe do some swimmin', too.
```

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='str', help='Input text or file')

    return parser.parse_args()


# --------------------------------------------------
def fry(word):
    """
    Drop the 'g' from '-ing' words, change "you" to "y'all"
    """

    ing_word = re.search('(.+)ing$', word)
    you = re.match('([Yy])ou$', word)

    if ing_word:
        prefix = ing_word.group(1)
        if re.search('[aeiouy]', prefix):
            return prefix + "in'"
    elif you:
        return you.group(1) + "'all"

    return word


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

```

```
    args = get_args()
    text = args.text

    if os.path.isfile(text):
        text = open(text).read()

    for line in text.splitlines():
        print(''.join(map(fry, re.split(r'(\W+)', line.rstrip()))))


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

Discussion

The heart of this program for me is the `fry` function. The `main` and `get_args` should look pretty standard by now. We get some argument that is either the text or the name of a file with the text. I chose to handle the input line-by-line because of the need to `print` the output. I don't want to worry about messing up the existing new lines, so I decided to read a line, strip off the newline, process the words, and `print` it all back out.

I wouldn't want to try to solve this problem without regular expressions, so I didn't really bother exploring a way that doesn't use them. For one thing, I use `re.split` to split the text on things that do and do not look like words. The first argument to this function is the regex that matches the thing you want to split on. Normally this is thrown away, for instance, if I `split` on any amount of whitespace, then the whitespace is not included:

```
>>> import re
>>> s = 'I said, "How do you do?"'
>>> re.split('\s+', s)
['I', 'said,', '"How', 'do', 'you', 'do?"']
```

It's a funny trick with this method that if you put the regex in capturing parens, it will return both the splitting text and the bits in between. The expression `\w` is any "word"-like character, so `\W` is the complement (non-word characters). The plus sign means "one or more", and so it finds all the non-word characters between the words. This is important because I don't want to lose them!

```
>>> re.split(r'(\W+)', s)
['I', ' ', 'said', ', "', 'How', ' ', 'do', ' ', 'you', ' ', 'do', '?"', '']
```

Now I need to process any string that ends in "ing":

```
>>> re.search('(.+)ing$', 'spam')
>>> re.search('(.+)ing$', 'fishing')
<re.Match object; span=(0, 7), match='fishing'>
```

I only want to remove the "g" from two-syllable words, though. A rough guess is to look for a vowel in the part of the work before the "ing", so I wrote the regex to capture the first part:

```
>>> match = re.search('(.+)ing$', 'fishing')
>>> prefix = match.group(1)
>>> prefix
'fish'
>>> re.search('[aeiouy]', prefix)
<re.Match object; span=(1, 2), match='i'>
```

But a word like "swing" would not work:

```
>>> match = re.search('(.+)ing$', 'swing')
```

```
>>> prefix = match.group(1)
>>> prefix
'sw'
>>> re.search('[aeiouy]', prefix)
```

If all the conditions are true, I `return` the `prefix` of the word with "in"'.

The other word to match is "you" either with an upper- or lowercase "y" which I can represent with a character class `[Yy]` for "either 'Y' or 'y' " which I additionally capture so as to reuse it and maintain the proper case:

```
>>> match = re.match('([Yy])ou$', 'You')
>>> match.group(1) + "'all"
"Y'all"
```

Finally we need to apply our `fry` function to all the pieces we got from splitting the input text. I know that a list comprehension is more "Pythonic," but I just prefer how `map` reads. I also understand that `map` is a bit slower due to the overhead of calling another function, but I don't usually choose Python for performace.

```
>>> def fry(word):
...     ing_word = re.search('(.+)ing$', word)
...     you = re.match('([Yy])ou$', word)
...     if ing_word:
...         prefix = ing_word.group(1)
...         if re.search('[aeiouy]', prefix):
...             return prefix + "in'"
...     elif you:
...         return you.group(1) + "'all"
...     return word
...
>>> s = "Hunting and fishing all you care about."
>>> ''.join([fry(w) for w in re.split(r'(\W+)', s)])
"Huntin' and fishin' all y'all care about."
>>> ''.join(map(fry, re.split(r'(\W+)', s)))
"Huntin' and fishin' all y'all care about."
```

Chapter 23: Mad Libs

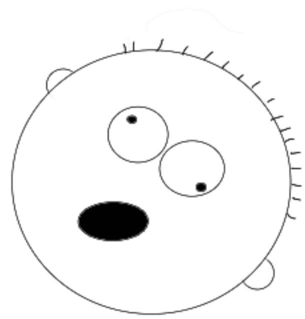


Figure 11: This definitely not a copyright infringment.

Write a Python program called `mad_lib.py` that will read a file given as a positional argument and find all the placeholders noted in <>, e.g., <verb>, prompt the user for the part of speech being requested, e.g., a "verb", and then substitute that into the text of the file, finally printing out all the placeholders replaced by the user's inputs. 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|--inputs` so the test suite can pass in all the answers and bypass the `input` calls.

Given no arguments or the `-h|--help` flag, the program should print a usage:

```
$ ./mad_lib.py
usage: mad_lib.py [-h] [-i str [str ...]] FILE
mad_lib.py: error: the following arguments are required: FILE
$ ./mad_lib.py -h
usage: mad_lib.py [-h] [-i str [str ...]] FILE

Mad Libs

positional arguments:
  FILE                  Input file

optional arguments:
  -h, --help            show this help message and exit
  -i str [str ...], --inputs str [str ...]
                        Inputs (for testing) (default: None)
```

The structure of the input file has a part of speech enclosed in angle brackets, e.g., <verb>:

```
$ cat help.txt
<exclamation>! I need <noun>!
<exclamation>! Not just <noun>!
<exclamation>! You know I need <noun>!
<exclamation>!
```

When this is the input for the program, you should ask for each part of speech in order using the `input` command to ask the user for some text. When you've gotten all the text you need, print out the result of putting the user's answers into the placeholders:

```
$ ./mad_lib.py help.txt
exclamation: Hey
noun: tacos
exclamation: Oi
noun: fish
exclamation: Ouch
noun: pie
exclamation: Dang
Hey! I need tacos!
Oi! Not just fish!
Ouch! You know I need pie!
Dang!
```

The default mode is to be interactive, which is difficult to test, so take all the `--inputs` from the command line, skip the `input` prompts, and just show the resulting text:

```
$ ./mad_lib.py romeo_juliet.txt -i cars Detroit oil pistons \
> "stick shift" furious accelerate 42 foot hammer
Two cars, both alike in dignity,
In fair Detroit, where we lay our scene,
From ancient oil break to new mutiny,
Where civil blood makes civil hands unclean.
From forth the fatal loins of these two foes
A pair of star-cross'd pistons take their life;
Whose misadventur'd piteous overthrows
Doth with their stick shift bury their parents' strife.
The fearful passage of their furious love,
And the continuance of their parents' rage,
Which, but their children's end, nought could accelerate,
Is now the 42 hours' traffic of our stage;
The which if you with patient foot attend,
What here shall hammer, our toil shall strive to mend.
```

Hints:

- Definitely look into using regular expressions!

Solution

```
#!/usr/bin/env python3
"""Mad Libs"""

import argparse
import re
from dire import die


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

    parser = argparse.ArgumentParser(
        description='Mad Libs',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file',
                        metavar='FILE',
                        type=argparse.FileType('r'),
                        help='Input file')

    parser.add_argument('-i',
                        '--inputs',
                        help='Inputs (for testing)',
                        metavar='str',
                        type=str,
                        nargs='+',
                        required=False)

    return parser.parse_args()


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

    args = get_args()
    inputs = args.inputs
    regex = re.compile('(<([^>])+>)')
    text = args.file.read().rstrip()
    blanks = list(regex.finditer(text))

    if not blanks:
```

```
        die('File "{}" has no placeholders'.format(args.file.name))

    for match in blanks:
        placeholder = match.group(1)
        name = match.group(2)
        answer = inputs.pop(0) if inputs else input('{}: '.format(name))
        text = re.sub(placeholder, answer, text, count=1)

    print(text)


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

Discussion

If you define the `file` with `type=argparse.FileType('r')`, then `argparse` with 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 none are 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`.

The first thing we need to do is `read` the input file:

```
>>> from pprint import pprint as pp
>>> text = open('help.txt').read()
>>> pp(text)
('<exclamation>! I need <noun>!\n'
 '<exclamation>! Not just <noun>!\n'
 '<exclamation>! You know I need <noun>!\n'
 '<exclamation>!\n')
```

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=(0, 1), match='<'>
```

Now let's find it's mate pair. The `.` means "anything," and we can add a `+` after it to mean "one or more":

```
>>> re.search('<.+>', text)
<re.Match object; span=(0, 28), match='<exclamation>! I need <noun>'>
```

Hmm, that matched all the way to the end of the string instead of stopping at the first available `>`. Often we find that regexes are "greedy" in that they will keep matching beyond where we want them to. If we add a `?` after the `+`, that will make it "non-greedy":

```
>>> re.search('<.+?>', text)
<re.Match object; span=(0, 13), match='<exclamation>'>
```

Another way is to say that we want to match one or more of anything that is *not* a `>`. We can create a character class `[>]` and then put a caret (`^`) *inside* it to negate the class. We'll add parens `()` to capture the whole mess as well as parens to capture the bit inside the `<>`

```
>>> re.search('(<([^>]+)>)', text)
<re.Match object; span=(0, 13), match='<exclamation>'>
```

Now we can `re.findall` if we just want the matching text:

```
>>> pp(re.findall('(<([^>]+)>)', text))
```

```
['<exclamation>',
 '<noun>',
 '<exclamation>',
 '<noun>',
 '<exclamation>',
 '<noun>',
 '<exclamation>']
```

But I wanted to use each match object, so I used `re.finditer` to return an iterator over the matches:

```
>>> it = re.finditer('(<([^>]+)>)', text)
>>> type(it)
<class 'callable_iterator'>
```

Iterables are "lazy", so if I `print` it, I'll just get a message about it being an object:

```
>>> pp(it)
<callable_iterator object at 0x1015a30f0>
```

In the REPL, I can call `list` to force Python to evaluate the iterable to the end:

```
>>> pp(list(it))
[<re.Match object; span=(0, 13), match='<exclamation>'>,
 <re.Match object; span=(22, 28), match='<noun>'>,
 <re.Match object; span=(30, 43), match='<exclamation>'>,
 <re.Match object; span=(54, 60), match='<noun>'>,
 <re.Match object; span=(62, 75), match='<exclamation>'>,
 <re.Match object; span=(93, 99), match='<noun>'>,
 <re.Match object; span=(101, 114), match='<exclamation>'>]
```

But note that it has not been exhausted and cannot be iterated again:

```
>>> pp(list(it))
[]
```

Which is why I convert it to a `list` right away so that I can evaluate if I found any (by inspecting the length) and then iterate over them:

```
>>> blanks = list(re.finditer('(<([^>]+)>)', text))
>>> len(blanks)
7
>>> for match in blanks:
...     print(match.group(1))
...
<exclamation>
<noun>
<exclamation>
<noun>
```

```
<exclamation>
<noun>
<exclamation>
```

The groups are defined in the order that you create them by counting the opening parenthesis. In our case, the second group is inside the first group, which is fine. If you start getting lots of groups, it might be best to name them:

```
>>> blanks = list(re.finditer(r'(?P<placeholder><(?P<name>[^>]+)>)', text))
>>> for match in blanks:
...     print(match.group('name'))
...
exclamation
noun
exclamation
noun
exclamation
noun
exclamation
```

Now to get the values from the user. We can use the name as the prompt for input:

```
>>> blanks = list(re.finditer(r'(<([^>]+)>)', text))
>>> for match in blanks:
...     name = match.group(2)
...     answer = input('{}: '.format(name))
...
exclamation: Dude!
```

But I will make it a bit more flexible by using an if *expression to take pop a value from the inputs list if that is available, otherwise I will use the input:

```
>>> inputs = ['Wow']
>>> answer = inputs.pop(0) if inputs else input('{}: '.format(name))
>>> answer
'Wow'
```

Now we need to put the user's answer into the original text which we can do with re.sub (substitute):

```
>>> pp(text)
('<exclamation>! I need <noun>!\n'
 '<exclamation>! Not just <noun>!\n'
 '<exclamation>! You know I need <noun>!\n'
 '<exclamation>!\n')
>>> placeholder = '<exclamation>'
>>> text = re.sub(placeholder, answer, text, count=1)
>>> pp(text)
('Wow! I need <noun>!\n'
```

```
'<exclamation>! Not just <noun>!\n'
'<exclamation>! You know I need <noun>!\n'
'<exclamation>!\n')
```

The `count=1` is necessary to prevent `re.sub` from replacing *every* instance of the pattern. After doing that for every placeholder in the `text`, we can `print(text)` and we are done!

Chapter 24: License Plates

Write a Python program called `license.py` that will create a regular expression for a license plate that accounts for characters and numbers which might be confused according to the following list:

- 5 S
- X K Y
- 1 I
- 3 E
- D 0 O Q
- M N
- U V W
- 2 8

Print the plate, the regular expression that would match that plate with all possible ambiguities, and then print all possible combinations of plates that includes the options along with the result of comparing the regular expression you created to the generated plate.

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

License plate regular expression

positional arguments:
  PLATE       License plate

optional arguments:
  -h, --help  show this help message and exit
$ ./license.py ABC1234
plate = "ABC1234"
regex = "^ABC[1I][27][3E]4$"
ABC1234 OK
ABC12E4 OK
ABC1734 OK
ABC17E4 OK
ABCI234 OK
ABCI2E4 OK
ABCI734 OK
ABCI7E4 OK
$ ./license.py 123456
plate = "123456"
regex = "^[1I][27][3E]4[5S]6$"
```

```
123456 OK
1234S6 OK
12E456 OK
12E4S6 OK
173456 OK
1734S6 OK
17E456 OK
17E4S6 OK
I23456 OK
I234S6 OK
I2E456 OK
I2E4S6 OK
I73456 OK
I734S6 OK
I7E456 OK
I7E4S6 OK
```

Owing to the vagaries of the typefaces chosen by different states as well as the wear of the plates themselves, it would seem to me that people might easily confuse certain letters and numbers on plates. In the above example, `ABC1234`, the number `1` might look like the letter `I`, so the plate could be `ABD1234` or `ABCI234`. Granted, most license plates follow a pattern of using only letters in some spots and numbers in others, e.g., 3 letters plus 4 numbers, but I want to focus on all possibilities in this problem both because it makes the problem a bit easier and also because it doesn't have to worry about how each state formats their plates. Additionally, I want to account for customized plates that do not follow any pattern and might use any combination of characters.

I represented the above confusion table as a list of tuples. At first I though I might use a dictionary, but there is a problem when three characters are involved, e.g., `0`, `O`, and `Q`. I iterate through each character in the provided plate and decide if the character exists in any of the tuples. If so, I represent that position in the regular expression as a choice; if not, it is just the character.

If you think about a regular expression as a graph, it starts with the first character, e.g., `A` which must be followed by `B` which must be followed by `C` which must be followed by either a `1` or an `I` which must be followed by a `2` or a `7`, etc.

```
                  1         2         3
A -> B -> C -> <     > -> <     > -> <     > -> 4
                  I         7         E
```

In creating all the possible plates from your regular expression, you are making concrete what the regular expression is, well, … expressing. I find `itertools.product` to be just the ticket for creating all those possibilites, which must be sorted for the sake of the test.

Solution

```
#!/usr/bin/env python3
"""License plate regular expression"""

import argparse
import re
import sys
from itertools import product


# --------------------------------------------------
def get_args():
    """get command-line arguments"""
    parser = argparse.ArgumentParser(
        description='License plate regular expression',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('plate', metavar='PLATE', help='License plate')

    return parser.parse_args()


# --------------------------------------------------
def main():
    """Make a jazz noise here"""
    args = get_args()
    plate = args.plate
    mixups = [('5', 'S'), ('X', 'K', 'Y'), ('1', 'I'), ('3', 'E'),
              ('D', 'O', 'O', 'Q'), ('M', 'N'), ('U', 'V', 'W'), ('2', '7')]

    chars = []
    for char in plate:
        group = list(filter(lambda t: char in t, mixups))
        if group:
            chars.append(group[0])
        else:
            chars.append((char, ))

    regex = '^{}$'.format(''.join(
        map(lambda t: '[' + ''.join(t) + ']' if len(t) > 1 else t[0], chars)))

    print('plate = "{}"'.format(plate))
    print('regex = "{}"'.format(regex))

```

```
    for possible in sorted(product(*chars)):
        s = ''.join(possible)
        print(s, 'OK' if re.search(regex, s) else 'NO')


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

Chapter 25: Gibberish Generator

Write a Python program called `gibberish.py` that uses the Markov chain algorithm to generate new words from the words in a set of training files. The program should take one or more positional arguments which are files that you read, word-by-word, and note the options of letters after a given `-k|--kmer_size` (default 2) grouping of letters. E.g., in the word "alabama" with `k=1`, the frequency table will look like:

```
a = l, b, m
l = a
b = a
m = a
```

That is, given this training set, if you started with `l` you could only choose an `a`, but if you have `a` then you could choose `l`, `b`, or `m`.

The program should generate `-n|--num_words` words (default `10`), each a random size between `k+2` and a `-m|--max_word` size (default `12`). Be sure to accept `-s|--seed` to pass to `random.seed`. My solution also takes a `-d|--debug` flag that will emit debug messages to `.log` for you to inspect.

If provided no arguments or the `-h|--help` flag, generate a usage:

```
$ ./gibberish.py
usage: gibberish.py [-h] [-n int] [-k int] [-m int] [-s int] [-d] FILE [FILE ...]
gibberish.py: error: the following arguments are required: FILE
$ ./gibberish.py -h
usage: gibberish.py [-h] [-n int] [-k int] [-m int] [-s int] [-d] FILE [FILE ...]

Markov chain for characters/words

positional arguments:
  FILE                  Training file(s)

optional arguments:
  -h, --help            show this help message and exit
  -n int, --num_words int
                        Number of words to generate (default: 10)
  -k int, --kmer_size int
                        Kmer size (default: 2)
  -m int, --max_word int
                        Max word length (default: 12)
  -s int, --seed int    Random seed (default: None)
  -d, --debug           Debug to ".log" (default: False)
```

Create new English words by training on a dictionary:

```
$ ./gibberish.py /usr/share/dict/words -s 1 -n 5
```

```
  1: salva
  2: xeroolizati
  3: upst
  4: azeconi
  5: woco
```

Or train on the US Constitution:

```
$ ./gibberish.py ../inputs/const.txt -s 2 -k 3 -n 4
  1: lfare
  2: sachmentit
  3: such
  4: rcessadopti
```

Generate new names for boys:

```
$ ./gibberish.py -k 2 -n 6 ../inputs/1945-boys.txt
  1: marthomart
  2: danie
  3: muel
  4: osep
  5: tomandrenny
  6: alberber
```

Chose the best words and create definitions for them:

- yulcogicism: the study of Christmas gnostics
- umjamp: skateboarding trick
- callots: insignia of officers in Greek army
- urchenev: fungal growth found under cobblestones

Kmers

To create the Markov chains, first you'll need to read all the words from each file. Use `str.lower` to lowercase all the text and then remove any character that are not in the regular English alphabet (a-z). A regular expression is handy for that:

```
>>> import re
>>> re.sub('[^a-z]', '', 'H48,`b09e3!"')
'be'
```

You'll need to extract "k-mers" or "n-grams" from each word. In the text "abcd," if `k=2` then the 2-mers are "ab," "bc," and "cd." If `k=3`, then the 3-mers are "abc" and "bcd." It may be helpful to know the number `n` of kmers `k` is proportional to the length `l` of the string `n = l - k + 1`.

Consider writing a function `get_kmers(text, k=1)` that only extracts kmers from some text, and then add this function to your program:

```
def test_get_kmers():
    """Test get_kmers"""

    assert get_kmers('abcd') == list('abcd')
    assert get_kmers('abcd', 2) == ['ab', 'bc', 'cd']
    assert get_kmers('abcd', 3) == ['abc', 'bcd']
    assert get_kmers('abcd', 4) == ['abcd']
    assert get_kmers('abcd', 5) == []
```

Run your program with `pytest -v gibberish.py` and see if it passes.

Chains

To create the Markov chains, you'll need to get all the kmers for `k+1` for all the words in all the texts. That is, if `k=3` you need to find all the 4-mers so that you can find the character *after* the 3-mers in the texts. For example, in the text "The quick brown fox jumps over the lazy dog.", we need to create a data structure that looks like this:

```
>>> from pprint import pprint as pp
>>> pp(chains)
{'bro': ['w'],
 'jum': ['p'],
 'laz': ['y'],
 'ove': ['r'],
 'qui': ['c'],
 'row': ['n'],
 'uic': ['k'],
 'ump': ['s']}
```

For every 3-mer, we need to know all the characters that follow each. Obviously this is not very exciting given the small size of the input text. If `k=2`, then you will see that `th` has two options, `e` and `e`. It's important to note how you will represent the choices for a given kmer. Will you use a `list`, a `set`, or a `collections.Counter`? Consider the implications. A `set` is smaller as it will represent only the *unique* letters but you will lose information about the *frequency* of letters. A `Counter` would store letters and counts, but how will you sample from that in a way that takes into account frequency? A `list` is probably the easiest structure.

Consider writing a function `read_training(fhs, k=1)` that reads the input training files and returns a dictionary of kmer chains. Then add this function to test that is works properly:

```
def test_read_training():
    """Test read_training"""
```

```
text = 'The quick brown fox jumps over the lazy dog.'

expected3 = {
    'qui': ['c'],
    'uic': ['k'],
    'bro': ['w'],
    'row': ['n'],
    'jum': ['p'],
    'ump': ['s'],
    'ove': ['r'],
    'laz': ['y']
}
assert read_training([io.StringIO(text)], k=3) == expected3

expected4 = {'quic': ['k'], 'brow': ['n'], 'jump': ['s']}
assert read_training([io.StringIO(text)], k=4) == expected4
```

Making new words

Once you have the chains of letters that follow each kmer, you need can use `random.choice` to find a starting kmer from the `keys` of your chain dictionary. Also use that function to select a length for your new word from the range of `k+2` to the `args.max_word` (which defaults to `12`). Build up your new word by again using `random.choice` to select from the possibilities for the kmer which will change through each iteration.

That is, if `k=3` and you start with the randomly selected kmer `ero`, you might get `n` as your next letter. On the next iteration of the loop, the `kmer` will be `ron` and you will look to see what letters follow that 3-mer. You might get `d`, and so the next time you would look for those letters following `ond`, and so forth. Continue until you've built a word that is the length you selected.

Hints:

- Define the input files with `type=argparse.FileType('r')` so that `argparse` will validate the user provides readable files and then will `open` them for you.
- Consider using the `logging` module to print out debugging messages. Run the `solution.py` with the `-d` flag and then inspect the `.log` file.

Solution

```
#!/usr/bin/env python3
"""Markov chain word generator"""

import argparse
import io
import logging
import random
import re
from collections import defaultdict


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

    parser = argparse.ArgumentParser(
        description='Markov chain word generator',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file',
                        metavar='FILE',
                        nargs='+',
                        type=argparse.FileType('r'),
                        help='Training file(s)')

    parser.add_argument('-n',
                        '--num_words',
                        help='Number of words to generate',
                        metavar='int',
                        type=int,
                        default=10)

    parser.add_argument('-k',
                        '--kmer_size',
                        help='Kmer size',
                        metavar='int',
                        type=int,
                        default=2)

    parser.add_argument('-m',
                        '--max_word',
                        help='Max word length',
                        metavar='int',
```

```
                        type=int,
                        default=12)

    parser.add_argument('-s',
                        '--seed',
                        help='Random seed',
                        metavar='int',
                        type=int,
                        default=None)

    parser.add_argument('-d',
                        '--debug',
                        help='Debug to ".log"',
                        action='store_true')

    return parser.parse_args()


# --------------------------------------------------
def get_kmers(text, k=1):
    """Return k-mers from text"""

    return [text[i:i + k] for i in range(len(text) - k + 1)]


# --------------------------------------------------
def test_get_kmers():
    """Test get_kmers"""

    assert get_kmers('abcd') == list('abcd')
    assert get_kmers('abcd', 2) == ['ab', 'bc', 'cd']
    assert get_kmers('abcd', 3) == ['abc', 'bcd']
    assert get_kmers('abcd', 4) == ['abcd']
    assert get_kmers('abcd', 5) == []


# --------------------------------------------------
def read_training(fhs, k=1):
    """Read training files, return chains"""

    chains = defaultdict(list)
    clean = lambda w: re.sub('[^a-z]', '', w.lower())

    for fh in fhs:
        for word in map(clean, fh.read().split()):
            for kmer in get_kmers(word, k + 1):
```

```
                chains[kmer[:-1]].append(kmer[-1])

    return chains


# --------------------------------------------------
def test_read_training():
    """Test read_training"""

    text = 'The quick brown fox jumps over the lazy dog.'

    expected3 = {
        'qui': ['c'],
        'uic': ['k'],
        'bro': ['w'],
        'row': ['n'],
        'jum': ['p'],
        'ump': ['s'],
        'ove': ['r'],
        'laz': ['y']
    }
    assert read_training([io.StringIO(text)], k=3) == expected3

    expected4 = {'quic': ['k'], 'brow': ['n'], 'jump': ['s']}
    assert read_training([io.StringIO(text)], k=4) == expected4


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

    args = get_args()
    k = args.kmer_size
    random.seed(args.seed)

    logging.basicConfig(
        filename='.log',
        filemode='w',
        level=logging.DEBUG if args.debug else logging.CRITICAL)

    chains = read_training(args.file, k)
    logging.debug(chains)

    kmers = list(chains.keys())
    for i in range(args.num_words):
        word = random.choice(kmers)
```

```
        length = random.choice(range(k + 2, args.max_word))
        logging.debug('Length "%s" starting with "%s"', length, word)

        while len(word) < length:
            kmer = word[-1 * k:]
            if not chains[kmer]:
                break

            char = random.choice(list(chains[kmer]))
            logging.debug('char = "%s"', char)
            word += char

        logging.debug('word = "%s"', word)
        print('{:3}: {}'.format(i + 1, word))


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

Discussion

As recommended in the description, I define my arguments in `get_args` to rely on `argparse` to validate as much as possible, e.g. verify that I get `int` values and readable files as well as provide reasonable defaults for everything but the required `file` argument. I additionally define a `-d|--debug` flag that is only `True` when present so that I can add this bit of code:

```
logging.basicConfig(
    filename='.log',
    filemode='w',
    level=logging.DEBUG if args.debug else logging.CRITICAL)
```

This is a simple and effective way to turn debugging messages on and off. I usually write to a `.log` file, being sure to choose a name that starts with a `.` so that it will normally be hidden when I `ls` the directory. Since the `filemode='w'`, the file will be overwritten on each run. I set the threshold to `logging.DEBUG` if the `debug` flag is `True`; otherwise the `logging` module will only emit those at the `CRITICAL` level. As I don't have any "critical" messages, the `.log` file will be empty unless the `--debug` is present. Then I have `logging.debug()` calls throughout my code which will only log messages when I ask. This is easier than putting `print` statements in your code which you have to remove or comment out when you are done debugging.

Finding kmers in text

If you followed my advice about breaking down the problem, then you probably created a `kmers` function with the formula for the number of kmers in a given test (`n = l - k + 1`):

```
>>> def get_kmers(text, k=1):
...     return [text[i:i + k] for i in range(len(text) - k + 1)]
...
```

Using the formula given in the intro for the number of kmers in a string, I use the `range` function to get the start position of each of those kmers and then get the slice of the `text` from that position to the position `k` away.

I can verify it works in the REPL:

```
>>> get_kmers('abcd', 2)
['ab', 'bc', 'cd']
>>> get_kmers('abcd', 3)
['abc', 'bcd']
```

But more importantly, I can write a `test_kmers` function that I embed in my code and run with `pytest`!

Reading the training files

Since I used the `argparse.FileType` to define the `file` with `nargs='+'`, I have a `list` of *open file handles* that can be read. I defined a `read_training` function that iterates over all the words in each file by calling `fh.read().split()`. As this breaks the text on spaces, various bits of punctuation may still be attached:

```
>>> fh = open('../inputs/spiders.txt')
>>> fh.read().split()
['Don't', 'worry,', 'spiders,', 'I', 'keep', 'house', 'casually.']
```

So I use a regular expression to remove anything that is *not* in the set of letters "a-z" by defining a negated character class `[^a-z]`. I create a one-line function to `lower` the word and `clean` it:

```
>>> import re
>>> clean = lambda word: re.sub('[^a-z]', '', word.lower())
>>> clean('"Hey!"')
'hey'
```

Now I can get cleaned, lowercase text:

```
>>> fh = open('../inputs/spiders.txt')
>>> list(map(clean, fh.read().split()))
['dont', 'worry', 'spiders', 'i', 'keep', 'house', 'casually']
```

I can now get all the kmers for each word by using my `kmers` function. I put all this into a function called `read_training`. It takes a `list` of open file handles (which I get from `argparse`) and a `k` which defaults to `1`:

```
>>> def read_training(fhs, k=1):
...     chains = defaultdict(list)
...     clean = lambda word: re.sub('[^a-z]', '', word.lower())
...     for fh in fhs:
...         for word in map(clean, fh.read().split()):
...             for kmer in get_kmers(word, k + 1):
...                 chains[kmer[:-1]].append(kmer[-1])
...     return chains
...
```

Note the handling of the kmers. I actually request `k+1`-mers and then slice `kmer[:-1]` to get the actual `k`-mer (everything up to the penultimate letter) and then `append` `kmer[-1]` (the last letter) to the `chains` for that `k`-mer.

I can verify it works:

```
>>> from collections import defaultdict
>>> from pprint import pprint as pp
>>> pp(read_training([open('../inputs/spiders.txt')], k=5))
defaultdict(<class 'list'>,
```

```
{'asual': ['l'],
 'casua': ['l'],
 'pider': ['s'],
 'spide': ['r'],
 'suall': ['y']})
```

But, again, *more importantly is that I can write a test that verifies it works*! If you copy in the `test_read_training` function, you have the assurange that you are creating valid chains.

Making new words

Once I have the chains from all the input files, I need to use a `for` loop for the `range(args.num_words)`. Each time through the loop, I need to choose a starting kmer for a new word and a length

```
>>> k = 3
>>> max_word = 12
>>> chains = read_training([open('../inputs/spiders.txt')], k)
>>> kmers = list(chains.keys())
>>> num_words = 3
>>> for i in range(num_words):
...     word = random.choice(kmers)
...     length = random.choice(range(k + 2, max_word))
...     print('Length "{}" starting with "{}"'.format(length, word))
...
Length "9" starting with "pid"
Length "7" starting with "cas"
Length "8" starting with "orr"
```

OK, that's our starting point. Given a starting kmer like `'pid'`, we need to create a `while` loop that will continue as long as the `len(word)` is less than the `length` we chose for the word. Each time through the loop, I'll set the current `kmer` to the last `k` letters of the `word`. I use `random.choice` to select from `chains[kmer]` to find the next `char` (character) and append that to the `word`:

```
>>> while len(word) < length:
...     kmer = word[-1 * k:]
...     if not chains[kmer]: break
...     char = random.choice(list(chains[kmer]))
...     word += char
...
>>> print(word)
piders
```

It can happen sometimes that there are no options for a given `kmer`. That is, `chains[kmer]` is an empty list, so I in my code I add a check to `break` out of the `while` loop if this evaluates to `False`.

Finally I `print` out the number of the word and the word itself using a format string to align the numbers and text:

```
>>> print('{:3}: {}'.format(i+1, word))
  3: piders
```

Machine Learning

If you didn't realize it, you just implemented a basic machine learning algorithm. Your program predicts the next letter after a given sequence based on the frequencies of patterns you "learned" from the training files! The kmers you extracted from the text could become vectors for other machine learning techniques. You could, for instance, train on texts that are labeled by language source (e.g., English, German, French) and then, given a new unlabled text, predict the language by the kmer frequencies.

What next

Now you can talk the "Markov Chain" problem that moves to the level of words and generates novel texts!

Chapter 26: Piggy (Pig Latin)

Write a Python program named `piggy.py` that takes one or more file names as positional arguments and converts all the words in them into "Pig Latin" (see rules below). Write the output to a directory given with the flags `-o|--outdir` (default `out-yay`) using the same basename as the input file, e.g., `input/foo.txt` would be written to `out-yay/foo.txt`.

If an argument names a non-existent file, print a warning to STDERR and skip that file. If the output directory does not exist, create it.

To create "Pig Latin":

1. If the word begins with consonants, e.g., "k" or "ch", move them to the end of the word and append "ay" so that "mouse" becomes "ouse-may" and "chair" becomes "air-chay."
2. If the word begins with a vowel, simple append "-yay" to the end, so "apple" is "apple-yay."

Figure 12: He's speaking "pig" Latin. Get it?

The program should print a usage if given no arguments or the `-h|--help` flag:

```
$ ./piggy.py
usage: piggy.py [-h] [-o str] FILE [FILE ...]
piggy.py: error: the following arguments are required: FILE
[cholla@~/work/python/playful_python/piggie]$ ./piggy.py -h
usage: piggy.py [-h] [-o str] FILE [FILE ...]

Convert to Pig Latin

positional arguments:
  FILE                  Input file(s)
```

```
optional arguments:
  -h, --help            show this help message and exit
  -o str, --outdir str  Output directory (default: out-yay)
```

If given a bad input file, it should complain and indicate an error:

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

For each file, write a new output file into the `--outdir`:

```
$ ./piggy.py ../inputs/sonnet-29.txt
  1: sonnet-29.txt
Done, wrote 1 file to "out-yay".
$ head -6 out-yay/sonnet-29.txt
onnet-Say 29
illiam-Way akespeare-Shay

en-Whay, in-yay isgrace-day ith-way ortune-fay and-yay en-may’s-yay eyes-yay,
I-yay all-yay alone-yay eweep-bay my-yay outcast-yay ate-stay,
And-yay ouble-tray eaf-day eaven-hay ith-way my-yay ootless-bay ies-cray,
$ ./piggy.py ../inputs/s*.txt
  1: scarlet.txt
  2: sonnet-29.txt
  3: spiders.txt
Done, wrote 3 files to "out-yay".
```

Hints:

- For the `file` argument, use `type=argparse.FileType('r')`
- First write a function that will create a Pig Latin version of just one word; write tests to verify that it does the right thing with words starting with vowels and with consonants
- Write a loop that prints the names of each input file
- Then write a loop inside that to read and print each line from a file
- Then figure out how to print each word on the line
- Then figure out how to print the Pig Latin version of each word on the line

Solution

```
#!/usr/bin/env python3
"""Convert text to Pig Latin"""

import argparse
import os
import re
import string
import textwrap


# --------------------------------------------------
def get_args():
    """get command-line arguments"""

    parser = argparse.ArgumentParser(
        description='Convert to Pig Latin',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

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

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

    return parser.parse_args()


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

    args = get_args()
    out_dir = args.outdir

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

```

    splitter = re.compile("([a-zA-Z](?:[a-zA-Z']*[a-zA-Z])?)")

    num_files = 0
    for i, fh in enumerate(args.file, start=1):
        basename = os.path.basename(fh.name)
        out_file = os.path.join(out_dir, basename)
        print('{:3}: {}'.format(i, basename))

        num_files += 1
        out_fh = open(out_file, 'wt')
        for line in fh:
            out_fh.write(''.join(map(pig, splitter.split(line))))
        out_fh.close()

    print('Done, wrote {} file{} to "{}".'.format(
        num_files, '' if num_files == 1 else 's', out_dir))


# --------------------------------------------------
def pig(word):
    """Create Pig Latin version of a word"""
    if re.match(r"^[\w']+$", word):
        vowels = 'aeiouAEIOU'
        consonants = re.sub('[' + vowels + ']', '', string.ascii_letters)
        match = re.match('^([' + consonants + ']+)(['+ vowels + '].*)', word)
        if match:
            word = '-'.join([match.group(2), match.group(1) + 'ay'])
        else:
            word = word + '-yay'
    return word

# --------------------------------------------------
def test_pig():
    """Test pig"""

    assert pig(' ') == ' '
    assert pig(', ') == ', '
    assert pig('\n') == '\n'
    assert pig('a') == 'a-yay'
    assert pig('i') == 'i-yay'
    assert pig('apple') == 'apple-yay'
    assert pig('cat') == 'at-cay'
    assert pig('chair') == 'air-chay'
    assert pig('the') == 'e-thay'
    assert list(map(pig, ['foo', '\n'])) == ['oo-fay', '\n']
```

```


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

Discussion

As with so many other exercises that want files as input, I'm going to rely on `argparse` to verify that the `type=argparse.FileType('r')` for the `file` argument. I will also specify `nargs='+'` to indicate "one or more." The `--outdir` is just a `str` and the directory it names may or may not exist, so there's really nothing to validate. I set the `default='out-yay'`.

Testing to see if a string names a directory is rather straightforward:

```
>>> import os
>>> out_dir = 'out-yay'
>>> os.path.isdir(out_dir)
False
```

If it doesn't exist, we use `os.makedirs` which is equivalent to `mkdir -p` on the command line in that parent directories will be created along the way if needed. That is, if the user specifies `~/python/pigsty/out_files` and you try to use `os.mkdir`, it will fail. This is why I never use `os.mkdir`:

```
>>> out_dir = '~/python/pigsty/out_files'
>>> os.path.isdir(out_dir)
False
>>> os.mkdir(out_dir)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: \
'~/python/pigsty/out_files'
```

The Pigifier

Let's start with the actual Pig Latinification of any given word. According to the rules, we add "-yay" to words starting with vowels, so "apple" becomes "apple-yay"; otherwise, we move consonant sounds to the end and add "ay", so "chair" becomes "air-chay." We've seen this same problem in other exercises like Runny Babbit and the rhymers. As in those solutions, to identify consonants I will complement the set of vowels:

```
>>> import string, re
>>> vowels = 'aeiouAEIOU'
>>> consonants = re.sub('[' + vowels + ']', '', string.ascii_letters)
>>> consonants
'bcdfghjklmnpqrstvwxyzBCDFGHJKLMNPQRSTVWXYZ'
```

I'm looking for the start of a string, so I will use the caret (^) to anchor a character class of consonants. I will verify this works:

```
>>> regex = '^[' + consonants + ']+'
```

```
>>> regex
'^[bcdfghjklmnpqrstvwxyzBCDFGHJKLMNPQRSTVWXYZ]+'
>>> re.search(regex, 'chair')
<re.Match object; span=(0, 2), match='ch'>
```

There needs to be at least one vowel after this:

```
>>> regex = '^[' + consonants + ']+[' + vowels + ']'
>>> regex
'^[bcdfghjklmnpqrstvwxyzBCDFGHJKLMNPQRSTVWXYZ]+[aeiouAEIOU]'
>>> re.search(regex, 'chair')
<re.Match object; span=(0, 3), match='cha'>
```

And then anything else which is represented with a dot `.` and any number which is a star `*`:

```
>>> regex = '^[' + consonants + ']+[' + vowels + '].*'
>>> re.search(regex, 'chair')
<re.Match object; span=(0, 5), match='chair'>
```

Finally we want to capture the first thing and the second thing, so we add parentheses so we can access the `match.groups()` method:

```
>>> regex = '^([' + consonants + ']+)([' + vowels + '].*)'
>>> regex
'^([bcdfghjklmnpqrstvwxyzBCDFGHJKLMNPQRSTVWXYZ]+)([aeiouAEIOU].*)'
>>> re.search(regex, 'chair')
>>> re.search(regex, 'chair').groups()
('ch', 'air')
```

So *if* the `search` succeeds and we get a `match` object, we can use the `groups` to move the first group to the end with "ay"; otherwise, we just add "-yay" to the end:

```
>>> word = 'chair'
>>> match = re.match('^([' + consonants + ']+)(['+ vowels + '].*)', word)
>>> if match:
...     word = '-'.join([match.group(2), match.group(1) + 'ay'])
... else:
...     word = word + '-yay'
...
>>> word
'air-chay'
```

Let's make this into a function:

```
>>> def pig(word):
...     """Create Pig Latin version of a word"""
...     if re.match(r"^[\w']+$", word):
...         vowels = 'aeiouAEIOU'
...         consonants = re.sub('[' + vowels + ']', '', string.ascii_letters)
```

```
...         match = re.match('^([' + consonants + ']+)(['+ vowels + '].*)', word)
...         if match:
...             word = '-'.join([match.group(2), match.group(1) + 'ay'])
...         else:
...             word = word + '-yay'
...     return word
...
```

And manually verify it:

```
>>> pig('chair')
'air-chay'
>>> pig('apple')
'apple-yay'
```

In my solution, I added a `test_pig` function that uses the `assert` function to verify that `pig('apple')` return "apple-yay" and so forth. If I run `pytest` on my `piggy.py` program, it will execute any function that starts with `test_`. This way I can be sure that my function continues to work if I make changes to the program.

Pigification of words

Now we have a problem we've seen in many other examples: we want to apply our `pig` function to all the words in a file. I've shown my preference for the `map` function, but we can also use `for` loops or list comprehensions. They will all accomplish the task.

First we have to deal with fact that we can have several input files. Since I defined the `file` argument with `nargs='+'`, then `args.file` will be a `list`. Even if the user defined only one file, `args.file` will be a `list` with one element. I can use a `for` loop to iterate through the files. Additionally, I want to list the number of the file as I'm processing them, so I will use the `enumerate` function to give me both the position in the list and the name of each file. Because I don't want to start counting at `0`, I'll use the `start=1` option:

```
>>> files = ['../inputs/spiders.txt', '../inputs/fox.txt']
>>> list(enumerate(files, start=1))
[(1, '../inputs/spiders.txt'), (2, '../inputs/fox.txt')]
```

That returns a `list` of tuples which I can unpack into a `i` ("integer", a common throwaway name for an incrementing counter) and `fh` ("file handle" which it is because `argparse` has already performed an `open` on the file). I will use a format string to print the `i` in a space three characters wide (right-justified), followed by a colon and then the name of the file:

```
>>> files = map(open, ['../inputs/spiders.txt', '../inputs/fox.txt'])
>>> for i, fh in enumerate(files, start=1):
```

```
...     print('{:3}: {}'.format(i, fh.name))
...
  1: ../inputs/spiders.txt
  2: ../inputs/fox.txt
```

I want to maintain the original line endings for each file, so I will read each file line-by-line using a `for` loop:

```
>>> fh = open('../inputs/spiders.txt')
>>> for line in fh:
...     print(line, end='')
...
Don’t worry, spiders,
I keep house
casually.
```

Now here is a problem: We can’t just use `split` to read the input text because punctuation will still be attached:

```
>>> line = "Don't worry, spiders,"
>>> line.split()
["Don't", 'worry,', 'spiders,']
```

Our `pig` fails with this input:

```
>>> pig('worry,')
'worry,'
>>> list(map(pig, line.split()))
["on't-Day", 'worry,', 'spiders,']
```

We could use regular expressions to remove anything not a character, but then we’d lose the original structure of the document. We need to find just the words themselves but not lose anything along the way. We can use `re.split` on `\W+` to define “one or more of any non-word character”:

```
>>> re.split('\W+', line)
['Don', 't', 'worry', 'spiders', '']
```

But that splits “Don’t” into two words, and we lose all the punctionation. Oddly, we can add capturing parens around the split pattern to get all the parts of the string:

```
>>> re.split('(\W+)', line)
['Don', "'", 't', ' ', 'worry', ', ', 'spiders', ',', '']
```

But that doesn’t stop “Don’t” being split. We need a far more complicated pattern:

```
>>> splitter = re.compile("([a-zA-Z](?:[a-zA-Z']*[a-zA-Z])?)")
>>> splitter.split(line)
['', "Don't", ' ', 'worry', ', ', 'spiders', ',']
```

Wow! I'll confess that I did not create that myself. I found it on StackOverflow, but the magical thing is that it was from a Java question. Regular expressions, however, is an idea separate from any one programming language. They are mostly compatible from Perl to Ruby to Rust. I searched for "regex split text word boundaries apostrophe" because I wanted a pattern that wouldn't split on a single quote (an apostrophe). I was able to use the exact pattern from an answer in Java because the regex is (usually) the same no matter where you use it!

Now I can `map` the `pig` function onto each part of the split line:

```
>>> list(map(pig, splitter.split(line)))
['', "on't-Day", ' ', 'orry-way', ', ', 'iders-spay', ',']
```

Or a list comprehension:

```
>>> [pig(w) for w in splitter.split(line)]
['', "on't-Day", ' ', 'orry-way', ', ', 'iders-spay', ',']
```

This is a rare exercise where you are required to write an output file. To create the ouput file name, we need the "basename" of the file which we can get with `os.path.basename`. Then use `os.path.join` to add the `out_dir` (which we created if needed) to the basename. Then we `open` that with the flags `w` for "write" and `t` for "text" which can be combined `wt`:

```
>>> fh.name
'../inputs/spiders.txt'
>>> basename = os.path.basename(fh.name)
>>> basename
'spiders.txt'
>>> out_file = os.path.join(out_dir, basename)
>>> out_file
'~/python/pigsty/out_files/spiders.txt'
>>> out_fh = open(out_file, 'wt')
```

I've called my output file handle `out_fh` so I can remember what it is. For each line of input text, we use `out_fh.write()` to print our text. It's important to remember that `print` will add a newline unless you tell it not to (using `end=''`), but `fh.write` will *not* add a newline unless you tell it to (by adding `+ '\n'` to your output string). In our case, the lines we are reading have a newline still attached, so we don't need to add another. Be sure to `close` the file handle when you are done.

```
>>> out_fh = open(out_file, 'wt')
>>> for line in fh:
...     out_fh.write(''.join(map(pig, splitter.split(line))))
...
>>> out_fh.close()
```

That is the crux of the program. All that is left is to report to the user how many files were processed and to remind them of the output directory.

Chapter 27: Soundex Rhymer

Write a Python program called `rhymer.py` that uses the Soundex algorithm/module to find words that rhyme with a given input word. When comparing words, you sometimes want to discount any leading consonants, e.g., the words "listen" and "glisten" rhyme but only if you compare the "isten" part, so the program should have an optional flag `-s|--stem` to indicate that the given word and the words you compare should both be trimmed to the "stem". The program should take an optional `-w|--wordlist` argument (default `/usr/share/dict/words`) for the comparisons and should respond, as always, to `-h|--help` for usage.

For more background on the Soundex algorithm, I recommend the Wikipedia page and the PyPi module documentation for `soundex`.

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

Find rhyming words using the Soundex

positional arguments:
  str                   Word

optional arguments:
  -h, --help            show this help message and exit
  -w str, --wordlist str
                        Wordlist (default: /usr/share/dict/words)
  -s, --stem            Stem the word (remove starting consonants (default:
                        False)
```

With my words list, I can find 37 words that rhyme with "listen" and 161 words that rhyme with the "isten" part:

```
$ ./rhymer.py listen | wc -l
      37
$ ./rhymer.py -s listen | wc -l
     161
```

I can verify that "glisten" only turns up when stemming is on:

```
$ ./rhymer.py listen | grep glisten
$ ./rhymer.py -s listen | grep glisten
glisten
```

Here is a sample of the words that my version finds:

```
$ ./rhymer.py listen | head -3
lackeydom
lactam
```

```
lactation
```

This program could be useful in creating custom input for the Gashlycrumb program.

Hints:

- You need to be sure that the given `word` actually has a vowel.
- If you are going to remove consonants from the beginning of a string, it might be easiest to find a regular expression to find things that are not vowels (because there are fewer of them to list).
- Another way to remove leading consonants would be to manually find the position of the first vowel in the string and then use a list slice on the given word to take the substring from that position to the end
- I suggest you use the `soundex` module

Testing the stemmer

I found the stemming part somewhat challenging, especially as I explored three different methods. I added the following test inside my `rhymer.py`:

```
def test_stemmer():
    """test stemmer"""

    assert stemmer('listen', True) == 'isten'
    assert stemmer('listen', False) == 'listen'
    assert stemmer('chair', True) == 'air'
    assert stemmer('chair', False) == 'chair'
    assert stemmer('apple', True) == 'apple'
    assert stemmer('apple', False) == 'apple'
    assert stemmer('xxxxxx', True) == 'xxxxxx'
    assert stemmer('xxxxxx', False) == 'xxxxxx'

    assert stemmer('LISTEN', True) == 'ISTEN'
    assert stemmer('LISTEN', False) == 'LISTEN'
    assert stemmer('CHAIR', True) == 'AIR'
    assert stemmer('CHAIR', False) == 'CHAIR'
    assert stemmer('APPLE', True) == 'APPLE'
    assert stemmer('APPLE', False) == 'APPLE'
    assert stemmer('XXXXXX', True) == 'XXXXXX'
    assert stemmer('XXXXXX', False) == 'XXXXXX'
```

And them I modified `make test` to include `rhymer.py` in the list of files to test. The `pytest` module looks for any function name that starts with `test_` and runs them. The `assert` will halt execution of the program if the test fails.

Some of the words in my system dictionary don't have vowels, so some of methods that assumed the presence of a vowel failed. Writing a test just for this one function really helped me find errors in my code.

Solution

```
#!/usr/bin/env python3
"""Find rhyming words using the Soundex"""

import argparse
import re
import string
from soundex import Soundex


# --------------------------------------------------
def get_args():
    """get command-line arguments"""

    parser = argparse.ArgumentParser(
        description='Find rhyming words using the Soundex',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

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

    parser.add_argument('-w',
                        '--wordlist',
                        metavar='str',
                        help='Wordlist',
                        type=argparse.FileType('r'),
                        default='/usr/share/dict/words')

    parser.add_argument('-s',
                        '--stem',
                        help='Stem the word (remove starting consonants',
                        action='store_true')

    args = parser.parse_args()

    #if not any([c in 'aeiouy' for c in args.word.lower()]):
    if not re.search('[aeiouy]', args.word, re.IGNORECASE):
        msg = 'word "{}" must contain at least one vowel'
        parser.error(msg.format(args.word))

    return args


# --------------------------------------------------
def stemmer(s: str, stem: bool) -> str:
```

```
    """Use regular expressions"""

    if stem:
        match = re.search(r'^[^aeiou]+([aeiou].*)', s, re.IGNORECASE)
        return match.group(1) if match else s
    return s


# --------------------------------------------------
# def stemmer(s: str, stem: bool) -> str:
#     """Manually `find` first vowel"""

#     if stem:
#         positions = list(
#             filter(lambda p: p >= 0, [s.lower().find(v) for v in 'aeiou']))
#         if positions:
#             first = min(positions)
#             return s[first:] if first else s
#     return s

# --------------------------------------------------
# def stemmer(s: str, stem: bool) -> str:
#     """Manually find first vowel with generator/next"""

#     if stem:
#         first = next(
#             (t[0] for t in enumerate(s) if t[1].lower() in 'aeiou'), False)
#         return s[first:] if first else s
#     return s


# --------------------------------------------------
def test_stemmer():
    """test stemmer"""

    assert stemmer('listen', True) == 'isten'
    assert stemmer('listen', False) == 'listen'
    assert stemmer('chair', True) == 'air'
    assert stemmer('chair', False) == 'chair'
    assert stemmer('apple', True) == 'apple'
    assert stemmer('apple', False) == 'apple'
    assert stemmer('xxxxxx', True) == 'xxxxxx'
    assert stemmer('xxxxxx', False) == 'xxxxxx'

    assert stemmer('LISTEN', True) == 'ISTEN'
    assert stemmer('LISTEN', False) == 'LISTEN'
```

```
    assert stemmer('CHAIR', True) == 'AIR'
    assert stemmer('CHAIR', False) == 'CHAIR'
    assert stemmer('APPLE', True) == 'APPLE'
    assert stemmer('APPLE', False) == 'APPLE'
    assert stemmer('XXXXXX', True) == 'XXXXXX'
    assert stemmer('XXXXXX', False) == 'XXXXXX'


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

    args = get_args()
    given = args.word
    words = args.wordlist.read().split()

    def sndx(s):
        return Soundex().soundex(stemmer(s, args.stem))

    wanted = sndx(given)

    for word in words:
        if given != word and sndx(word) == wanted:
            print(word)

    # print('\n'.join(
    #     filter(lambda word: given != word and sndx(word) == wanted, words)))

    # print('\n'.join([
    #     word for word in words if given != word and sndx(word) == wanted
    # ]))


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

Discussion

The first thing to check is that the given word contains a vowel which is simple enough if you use regular expressions. We'll include "y" for this purpose:

```
>>> re.search('[aeiouy]', 'YYZ', re.IGNORECASE) or 'Fail'
<re.Match object; span=(0, 1), match='Y'>
>>> re.search('[aeiouy]', 'bbbb', re.IGNORECASE) or 'Fail'
'Fail'
```

Another way that doesn't use a regex could use a list comprehension to iterate over character in the given word to see if it is `in` the `list` of vowels 'aeiouy':

```
>>> [c in 'aeiouy' for c in 'CAT'.lower()]
[False, True, False]
```

You can then ask if `any` of these tests are true:

```
>>> any([c in 'aeiouy' for c in 'CAT'.lower()])
True
>>> any([c in 'aeiouy' for c in 'BCD'.lower()])
False
```

By far the regex version is simpler, but it's always interesting to think about other ways to accomplish a task. Anyway, if the given `word` does not have a vowel, I throw a `parser.error`.

Using Soundex

The `soundex` module has you create a `Soundex` object and then call a `soundex` function, which all seems a bit repetitive. Still, it gives us a way to get a Soundex value for a given word:

```
>>> from soundex import Soundex
>>> sndx = Soundex()
>>> sndx.soundex('paper')
'p16'
```

The problem is that sometimes we want the stemmed version of the word:

```
>>> sndx.soundex('aper')
'a16'
```

So I wrote a `stemmer` function that does (or does not) stem the word using the value of the `--stem` option which I defined in `argparse` as a Boolean value. I tried to find a way to remove leading consonants both with and without regular expressions. The regex version builds a somewhat complicated regex. Let's start with how to match something at the start of a string that is *not* a vowel (again, because there are only 5 to list):

```
>>> import re
>>> re.search(r'^[^aeiou]+', 'chair')
<re.Match object; span=(0, 2), match='ch'>
```

So we saw earlier that `[aeiou]` is the character class that matches vowels, so we can *negate* the class with `^` **inside** the character class. It's a bit confusing because there is also a `^` at the beginning of the `r''` (raw) string that anchors the expression to the beginning of the string.

OK, so that find the non-vowels leading the word, but we want the bit afterwards. It seems like we could just write something like this:

```
>>> re.search(r'^[^aeiou]+(.+)$', 'chr')
<re.Match object; span=(0, 3), match='chr'>
```

Which seems to say "one or more non-vowels followed by one or more of anything" and it looks to work, but look further:

```
>>> re.search(r'^[^aeiou]+(.+)$', 'chr').groups()
('r',)
```

It finds the last `r`. We need to specify that after the non-vowels there needs to be at least one vowel:

```
>>> re.search(r'^[^aeiou]+([aeiou].*)', 'chr')
```

And now it works:

```
>>> re.search(r'^[^aeiou]+([aeiou].*)', 'chr')
>>> re.search(r'^[^aeiou]+([aeiou].*)', 'car')
<re.Match object; span=(0, 3), match='car'>
>>> re.search(r'^[^aeiou]+([aeiou].*)', 'car').groups()
('ar',)
```

So the `stemmer` works by first looking to see if we should even attempt to `stem`. If so, it attempts to match the regular expression. If that succeeds, then it returns the match. The `else` for everything is to return the original string `s`.

The two other versions of `stemmer` rely on some things I'll discuss later.

As stated in the intro, it was most helpful to me to add the `test_stemmer` function to ensure that all my versions of the `stemmer` function actually had the same behavior.

Once I have the `stemmer` function, I can apply it to the given `word` and every word in the `--wordlist` and then call the "

Chapter 28: Tic-Tac-Toe Outcome

Create a Python program called `outcome.py` that takes a given Tic-Tac-Toe state as it's only (positional) argument and reports if X or O has won or if there is no winner. The state should only contain the characters ".", "O", and "X", and must be exactly 9 characters long. If there is not exactly one argument, print a "usage" statement.

```
$ ./outcome.py
Usage: outcome.py STATE
$ ./outcome.py ..X.OA..X
State "..X.OA..X" must be 9 characters of only ., X, O
$ ./outcome.py ..X.OX...
No winner
$ ./outcome.py ..X.OX..X
X has won
```

Solution

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

import os
import re
import sys


# --------------------------------------------------
def main():
    args = sys.argv[1:]

    if len(args) != 1:
        print('Usage: {} STATE'.format(os.path.basename(sys.argv[0])))
        sys.exit(1)

    state = args[0]

    if not re.search('^[.XO]{9}$', state):
        print('State "{}" must be 9 characters of only ., X, O'.format(state),
              file=sys.stderr)
        sys.exit(1)

    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]]

    winner = 'No winner'

    # for player in ['X', 'O']:
    #     for combo in winning:
    #         i, j, k = combo
    #         if state[i] == player and state[j] == player and state[k] == player:
    #             winner = player
    #             break

    # for player in ['X', 'O']:
    #     for combo in winning:
    #         chars = []
    #         for i in combo:
    #             chars.append(state[i])

    #         if ''.join(chars) == player * 3:
    #             winner = player
    #             break
```

```

    # for player in ['X', 'O']:
    #     for i, j, k in winning:
    #         chars = ''.join([state[i], state[j], state[k]])
    #         if ''.join(chars) == '{}{}{}'.format(player, player, player):
    #             winner = player
    #             break

    for player in ['X', 'O']:
        for i, j, k in winning:
            combo = [state[i], state[j], state[k]]
            if combo == [player, player, player]:
                winner = '{} has won'.format(player)
                break

    # for combo in winning:
    #     group = list(map(lambda i: state[i], combo))
    #     for player in ['X', 'O']:
    #         if all(x == player for x in group):
    #             winner = player
    #             break

    print(winner)


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

Chapter 29: Twelve Days of Christmas

Write a Python program called `twelve_days.py` that will generate the "Twelve Days of Christmas" song up to the `-n|--number_days` argument (default `12`), writing the resulting text to the `-o|--outfile` argument (default STDOUT).

```
$ ./twelve_days.py -h
usage: twelve_days.py [-h] [-o str] [-n int]

Twelve Days of Christmas

optional arguments:
  -h, --help            show this help message and exit
  -o str, --outfile str
                        Outfile (STDOUT) (default: )
  -n int, --number_days int
                        Number of days to sing (default: 12)
$ ./twelve_days.py -n 1
On the first day of Christmas,
My true love gave to me,
A partridge in a pear tree.

$ ./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.

$ ./twelve_days.py -o out
$ wc -l out
     113 out
```

Solution

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

import argparse
import sys
from dire import die


# --------------------------------------------------
def get_args():
    """get command-line arguments"""
    parser = argparse.ArgumentParser(
        description='Twelve Days of Christmas',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('-o',
                        '--outfile',
                        help='Outfile (STDOUT)',
                        metavar='str',
                        type=str,
                        default='')

    parser.add_argument('-n',
                        '--number_days',
                        help='Number of days to sing',
                        metavar='int',
                        type=int,
                        default=12)

    return parser.parse_args()


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

    args = get_args()
    out_file = args.outfile
    num_days = args.number_days
    out_fh = open(out_file, 'wt') if out_file else sys.stdout

    days = {
        12: 'Twelve drummers drumming',
        11: 'Eleven pipers piping',
```

```
        10: 'Ten lords a leaping',
        9: 'Nine ladies dancing',
        8: 'Eight maids a milking',
        7: 'Seven swans a swimming',
        6: 'Six geese a laying',
        5: 'Five gold rings',
        4: 'Four calling birds',
        3: 'Three French hens',
        2: 'Two turtle doves',
        1: 'a partridge in a pear tree',
    }

    ordinal = {
        12: 'twelfth', 11: 'eleven', 10: 'tenth',
        9: 'ninth', 8: 'eighth', 7: 'seventh',
        6: 'sixth', 5: 'fifth', 4: 'fourth',
        3: 'third', 2: 'second', 1: 'first',
    }

    if not num_days in days:
        die('Cannot sing "{}" days'.format(num_days))

    for i in range(1, num_days + 1):
        first = 'On the {} day of Christmas,\nMy true love gave to me,'
        out_fh.write(first.format(ordinal[i]) + '\n')
        for j in reversed(range(1, i + 1)):
            if j == 1:
                if i == 1:
                    out_fh.write('{}.\n'.format(days[j].title()))
                else:
                    out_fh.write('And {}.\n'.format(days[j]))
            else:
                out_fh.write('{},\n'.format(days[j]))

        if i < max(days.keys()):
            out_fh.write('\n')


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

Chapter 30: Anagram

Write a program called `presto.py` that will find anagrams of a given positional argument. The program should take an optional `-w|--wordlist` (default `/usr/share/dict/words`) and produce output that includes combinations of `-n|num_combos` words (default `1`) that are anagrams of the given input.

It should provide a usage with no input or the `-h|--help` flags:

```
$ ./presto.py
usage: presto.py [-h] [-w str] [-n int] [-d] str
presto.py: error: the following arguments are required: str
$ ./presto.py -h
usage: presto.py [-h] [-w str] [-n int] [-d] str

Find anagrams

positional arguments:
  str                   Input text

optional arguments:
  -h, --help            show this help message and exit
  -w str, --wordlist str
                        Wordlist (default: /usr/share/dict/words)
  -n int, --num_combos int
                        Number of words combination to test (default: 1)
  -d, --debug           Debug (default: False)
```

Be default, it should search the `--wordlist` file for other words of the same length as the input that have the same letters in the same frequency:

```
$ ./presto.py presto
presto =
   1. poster
   2. repost
   3. respot
   4. stoper
$ ./presto.py listen
listen =
   1. enlist
   2. silent
   3. tinsel
```

If `-n` is greater than 1 (the default), then the program should additionally find all combinations of two words that together create the original word.

```
$ ./presto.py listen -n 2 | tail
  82. sten li
```

```
83. te nils
84. ten lis
85. ten sil
86. ti lens
87. til ens
88. til sen
89. tin els
90. tin les
91. tinsel
```

Hints:

- How will you determine that two strings are anagrams? That is, what is the code you will use to compare two strings and return `True` or `False` that they are anagrams? Start there.
- You can assume a strict dictionary-type input file like the default, but you might also consider mining some other text as the source for anagrams, one that might have punctuation and mixed-case letters.
- When you move to `-n > 1`, you may find you quickly have an overwhelming number of combinations to consider. My `/usr/share/dict/words` has 235886 words. At `n=2`, that could produce over 55 *billion* combinations of words. Obviously I don't need to consider the entire Cartesian product of the list, only those whose lengths sum to equal the length of the input word. How can you find all the combinations of numbers that sum to that length? E.g, for 5, you can add 0 + 5, 1 + 4, and 2 + 3. How can you segregate all the input words by their lengths?

Solution

```
#!/usr/bin/env python3
"""Find anagrams"""

import argparse
import logging
import re
from collections import defaultdict, Counter
from itertools import combinations, permutations, product, chain


# --------------------------------------------------
def get_args():
    """get command-line arguments"""

    parser = argparse.ArgumentParser(
        description='Find anagrams',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('text', metavar='str', help='Input text')

    parser.add_argument('-w',
                        '--wordlist',
                        help='Wordlist',
                        metavar='str',
                        type=argparse.FileType('r'),
                        default='/usr/share/dict/words')

    parser.add_argument('-n',
                        '--num_combos',
                        help='Number of words combination to test',
                        metavar='int',
                        type=int,
                        default=1)

    parser.add_argument('-d', '--debug', help='Debug', action='store_true')

    return parser.parse_args()


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

```

```
    args = get_args()
    text = args.text

    logging.basicConfig(
         filename='.log',
        filemode='w',
        level=logging.DEBUG if args.debug else logging.CRITICAL)

    words = defaultdict(set)
    regex = re.compile('[^a-z0-9]')
    for line in args.wordlist:
        for word in line.split():
            clean = regex.sub('', word.lower())
            if len(clean) == 1 and clean not in 'ai':
                continue
            words[len(clean)].add(clean)

    text_len = len(text)
    counts = Counter(text)
    anagrams = set()
    lengths = list(words.keys())
    for n in range(1, args.num_combos + 1):
        key_combos = list(
            filter(
                lambda t: sum(t) == text_len,
                set(
                    map(lambda t: tuple(sorted(t)),
                        combinations(chain(lengths, lengths), n)))))
        logging.debug('key combos = %s', key_combos)

        for keys in key_combos:
            logging.debug('Searching keys %s', keys)
            word_combos = list(product(*list(map(lambda k: words[k], keys))))

            for t in word_combos:
                if Counter(''.join(t)) == counts:
                    logging.debug('combo = %s', t)
                    for s in [' '.join(l) for l in permutations(t)]:
                        if s != text:
                            anagrams.add(s)

            logging.debug('# anagrams = %s', len(anagrams))

    logging.debug('Finished searching')

    if anagrams:
```

```
        print('{} ='.format(text))
        for i, t in enumerate(sorted(anagrams), 1):
            print('{:4}. {}'.format(i, t))
    else:
        print('No anagrams for "{}".'.format(text))


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

Discussion

I rely on `type=argparse.FileType('r')` for any "file" argument, so my `get_args` once again uses that to define the input `--wordlist`. Likewise, I defined `--num_combos` as an `int` and let `argparse` handle argument validation for me.

Logging

My solution also incorporates the `logging` I used while solving this problem for myself. I tend copy and paste this block all the time:

```
logging.basicConfig(
    filename='.log',
    filemode='w',
    level=logging.DEBUG if args.debug else logging.CRITICAL)
```

If I define `args.debug` as a Boolean, then I can effectively turn `logging` on and off because I tend not to write `CRITICAL` messages. Since I use `filemode='w'` to overwrite the `.log` file, then that file will be empty after every run that `--debug` isn't on (and the default is that it is not). Also, I like to use a filename starting with a `.` (e.g., `.log`) as it will be hidden in most Unix-style `ls` commands. This makes logging as transparent and easy as I can think.

Reading wordlist

First I handle getting a wordlist. I wrote a rather verbose way to process what could be a large input file. Rather than called `args.wordlist.read().split()` which reads the *entire file into memory*, I chose to read each line one-by-one into memory and call `line.split()` on that. If you have to deal with large input file (e.g., I regularly deal with files in the gigabytes in biology!), it's best to read line-by-line.

I iterate `for` each `word` in the line and clean it up with a regular expression that defines a character class of all the characters `a-z` and `0-9` with `[a-z0-9]` and then uses a caret **inside** the character class to negate it. Then I use the `sub` (substitute) function to replace characters that match with the empty string:

```
>>> import re
>>> regex = re.compile('[^a-z0-9]')
>>> regex.sub('', '"hey!"')
'hey'
```

If the remaining word is only 1 character long, I only accept it if it is "a" or "i".

To store the words, I decided to use a dictionary where the key is the length of a given word and the value is a `set` of the words of that length. I chose a set in case I was reading a file other than a standard dictionary-type file where words might be repeated. I use the length of the each word as the key so that I can select the combinations of words whose lengths sum to the desired lenghts. That is, if my input word is 5 characters long, there is no reason to look at words longer than 5 characters.

defaultdict

To define this data structure, I used `words = defaultdict(set)` where `defaultdict` takes a datatype like `str` or `list` as the default *value* to initiate when a given key does not exist. For instance, using `int` will create a new entry in the dictionary with a value of `0`:

```
>>> from collections import defaultdict
>>> d = defaultdict(int)
>>> d
defaultdict(<class 'int'>, {})
>>> d['foo'] += 1
>>> d
defaultdict(<class 'int'>, {'foo': 1})
```

If you use `str`, the empty string will be used:

```
>>> d = defaultdict(str)
>>> d
defaultdict(<class 'str'>, {})
>>> d['foo'] += 'a'
>>> d
defaultdict(<class 'str'>, {'foo': 'a'})
>>> d['foo'] += 'b'
>>> d
defaultdict(<class 'str'>, {'foo': 'ab'})
```

Likewise with a `list`, you get an empty list:

```
>>> d = defaultdict(list)
>>> d
defaultdict(<class 'list'>, {})
>>> d['foo'] += 'a'
>>> d['foo'] += 'b'
>>> d
defaultdict(<class 'list'>, {'foo': ['a', 'b']})
```

And so, with `set` you get an empty set to which you can `add`:

```
>>> d = defaultdict(set)
>>> d
defaultdict(<class 'set'>, {})
```

```
>>> d['foo'].add('a')
>>> d['foo'].add('b')
```

Note that the argument to `defaultdict` is *not* in quotes. You are passing the class `set` not the string `'set'`!

So, with all that, I end up adding words like so:

```
>>> words = defaultdict(set)
>>> word = 'apple'
>>> words[len(word)].add(word)
>>> word = 'bear'
>>> words[len(word)].add(word)
>>> words
defaultdict(<class 'set'>, {5: {'apple'}, 4: {'bear'}})
```

Identifying anagrams

In the intro to the problem, I mentioned my algorithm for finding an anagram:

1. Same length as the given word
2. Same frequency of characters as the given word

The first one is easy enough to find using `len`. If our given word is "listen," then we only need to look at words of length 6 or less:

```
>>> given = 'listen'
>>> len(given)
6
```

How about the character frequency? There are many ways to find this, but I know of no easier method than to use the `Counter` from the `collections` module:

```
>>> from collections import Counter
>>> Counter('listen')
Counter({'l': 1, 'i': 1, 's': 1, 't': 1, 'e': 1, 'n': 1})
```

If we are looking at the word "tinsel," we see that we have found an anagram:

```
>>> word = 'tinsel'
>>> len(given) == len(word)
True
>>> Counter(given) == Counter(word)
True
```

Selecting words to compare

My first implementation of this program was quite naive and yet worked fine for find all other single words that were anagrams. Everything came crashing down when I attempted to find combinations. I suddenly realized the number of 2-word combinations I needed to check (that 55 *billion* I mentioned before). As it happened, I then rewatched "The Imitation Game" about Alan Turing and the creation of his machine ("Christopher") to crack the Enigma code which has a possible 150 million million possible states. He was unable to build a machine that could churn through that many possibilities in the 18 hours or so per day they had to find the right combination, so they had to find a way to cut down the number of combinations they attempted. Similarly, I realized I only needed to look at combinations of words whose lengths sum to the length of the given word; hence my decision to store `words` using the word length as the key and then as a `set` of words that length.

Next I needed to find all combinations of numbers that add up to that number. Let's consider we are using "listen" as the `text`:

```
>>> text = 'listen'
>>> text_len = len(text)
>>> text_len
6
```

I need to do quite a few complex operations for which the `itertools` module provides very handy functions:

```
>>> from itertools import combinations, permutations, product, chain
```

First assume that we had words ranging from 1 to 10 characters in our word list file:

```
>>> lengths = list(range(1, 11))
>>> lengths
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Now find all `combinations` of all the different lengths, I first need to `chain` the `lengths` to add it to itself:

```
>>> list(chain(lengths, lengths))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

And then find the combinations of which there are many:

```
>>> len(list(combinations(chain(lengths, lengths), 2)))
190
>>> list(combinations(chain(lengths, lengths), 2))[:5]
[(1, 2), (1, 3), (1, 4), (1, 5), (1, 6)]
```

It turns out the list is longer than necessary because the tuples are unique, so we can fix that with a `set`:

```
>>> combos = combinations(chain(lengths, lengths), 2)
>>> uniq_combos = set(map(lambda t: tuple(sorted(t)), combos))
>>> len(uniq_combos)
55
>>> list(uniq_combos)[:3]
[(5, 9), (4, 7), (1, 3)]
```

And then find those where the sum is `6`:

```
>>> list(filter(lambda t: sum(t) == 6, uniq_combos))
[(3, 3), (1, 5), (2, 4)]
```

If we put it all together and look for combinations of 3 numbers that sum to 6:

```
>>> n = 3
>>> text_len = 6
>>> key_combos = list(
...     filter(
...         lambda t: sum(t) == text_len,
...         set(
...             map(lambda t: tuple(sorted(t)),
...                 combinations(chain(lengths, lengths), n)))))
>>>
>>> key_combos
[(1, 1, 4), (1, 2, 3)]
```

Now I have the *keys* for the `words` to look to check for word combinations!

```
>>> key_combos
[(3, 3), (1, 5), (2, 4)]
```

Let's take the first combo:

```
>>> keys = key_combos[0]
>>> keys
(3, 3)
```

And pretend we have a very small `words` list:

```
>>> words = defaultdict(set)
>>> words[3].add('les')
>>> words[3].add('tin')
>>> words[4].add('lest')
>>> words[4].add('list')
>>> words[3].add('len')
>>> words[3].add('its')
>>> words
defaultdict(<class 'set'>, {3: {'len', 'its', 'tin', 'les'}, 4: {'list', 'lest'}})
```

I can `map` to find to find all the words for those lengths.

```
>>> list(map(lambda k: words[k], keys))
```

```
[{'len', 'its', 'tin', 'les'}, {'len', 'its', 'tin', 'les'}]
```

And then use `product` to get the Cartesian combination:

```
>>> word_combos = list(product(*list(map(lambda k: words[k], keys))))
>>> word_combos[:3]
[('len', 'len'), ('len', 'its'), ('len', 'tin')]
```

Which I can then iterate and apply my algorithm described above to decide if there are any anagrams:

```
>>> counts = Counter('listen')
>>> for t in word_combos:
...     if Counter(''.join(t)) == counts:
...         for s in [' '.join(l) for l in permutations(t)]:
...             if s != text:
...                 print(s)
...
len its
its len
its len
len its
tin les
les tin
les tin
tin les
```

Some are repeated which is why I chose to create my `anagrams` holder as a `set` to make them unique.

In the end, I look to see how many `anagrams` I found using `len(anagrams)`. If there are some, I report how many and what they are in `sorted` order; otherwise I let the user know that none were found.

Chapter 31: Hangman

Write a Python program called `hangman.py` that will play a game of Hangman which is a bit like "Wheel of Fortune" where you present the user with a number of elements indicating the length of a word. For our game, use the underscore `_` to indicate a letter that has not been guessed. The program should take `-n|--minlen` minimum length (default `5`) and `-l|--maxlen` maximum length options (default `10`) to indicate the minimum and maximum lengths of the randomly chosen word taken from the `-w|--wordlist` option (default `/usr/share/dict/words`). It also needs to take `-s|--seed` to for the random seed and the `-m|--misses` number of misses to allow the player.

The game is intended to be interactive, but I want you to additionally take an `-i|--inputs` option that is a string of letters to use as guesses so that we can write a test.

When run with the `-h|--help` flags, it should present a usage statement:

```
$ ./hangman.py -h
usage: hangman.py [-h] [-l MAXLEN] [-n MINLEN] [-m MISSES] [-s SEED]
                  [-w WORDLIST] [-i INPUTS]

Hangman

optional arguments:
  -h, --help            show this help message and exit
  -l MAXLEN, --maxlen MAXLEN
                        Max word length (default: 10)
  -n MINLEN, --minlen MINLEN
                        Min word length (default: 5)
  -m MISSES, --misses MISSES
                        Max number of misses (default: 10)
  -s SEED, --seed SEED  Random seed (default: None)
  -w WORDLIST, --wordlist WORDLIST
                        Word list (default: /usr/share/dict/words)
  -i INPUTS, --inputs INPUTS
                        Input choices (default: )
```

If given a bad `--wordlist`, error out (print the problem and exit with a non-zero status) with a message like so:

```
$ ./hangman.py -w kdfkj
usage: hangman.py [-h] [-l MAXLEN] [-n MINLEN] [-m MISSES] [-s SEED]
                  [-w WORDLIST] [-i INPUTS]
hangman.py: error: argument -w/--wordlist: can't open 'kdfkj': [Errno 2] \
No such file or directory: 'kdfkj'
```

If given a value less than 1 for `--minlen`, error out:

```
$ ./hangman.py -n -4
usage: hangman.py [-h] [-l MAXLEN] [-n MINLEN] [-m MISSES] [-s SEED]
                  [-w WORDLIST] [-i INPUTS]
hangman.py: error: --minlen "-4" must be positive
```

If given a `--maxlen` value greater than 20, error out:

```
$ ./hangman.py -l 30
usage: hangman.py [-h] [-l MAXLEN] [-n MINLEN] [-m MISSES] [-s SEED]
                  [-w WORDLIST] [-i INPUTS]
hangman.py: error: --maxlen "30" must be < 20
```

Error out if the `--minlen` is greater than the `--maxlen`:

```
$ ./hangman.py -l 5 -n 10
usage: hangman.py [-h] [-l MAXLEN] [-n MINLEN] [-m MISSES] [-s SEED]
                  [-w WORDLIST] [-i INPUTS]
hangman.py: error: --minlen "10" is greater than --maxlen "5"
```

To play, you will initiate an inifinite loop and keep track of the game state, e.g., the word to guess, the letters already guessed, the letters found, the number of misses. As this is an interactive game, you will normally use the `input` function to get a letter from the user. If given `--inputs`, bypass the `input` prompt and instead use those letters in turn.

If the user guesses a letter that is in the word, replace the `_` characters with the letter. If the user guesses the same letter twice, admonish them. If the user guesses a letter that is not in the word, increment the misses and let them know they missed. If the user guesses too many times, exit the game and insult them. If they correctly guess the word, let them know and exit the game.

```
$ ./hangman.py -s 2
_ _ _ _ _ _ _ (Misses: 0)
Your guess? ("?" for hint, "!" to quit) a
There is no "a"
_ _ _ _ _ _ _ (Misses: 1)
Your guess? ("?" for hint, "!" to quit) i
There is no "i"
_ _ _ _ _ _ _ (Misses: 2)
Your guess? ("?" for hint, "!" to quit) e
_ _ _ _ _ _ e (Misses: 2)
Your guess? ("?" for hint, "!" to quit) o
o _ o _ _ _ e (Misses: 2)
Your guess? ("?" for hint, "!" to quit) z
o z o _ _ _ e (Misses: 2)
Your guess? ("?" for hint, "!" to quit) t
o z o t _ _ e (Misses: 2)
Your guess? ("?" for hint, "!" to quit) p
o z o t _ p e (Misses: 2)
```

```
Your guess? ("?" for hint, "!" to quit) y
You win. You guessed "ozotype" with "2" misses!
```

Play the `solution.py` a few times to get a feel for how the game should work.

Hints:

- Leverage `get_args` and `argparse` to validate inputs. Use `type=argparse.FileType('r')` for the `--wordlist`. Check the value of `--minlen` and `--maxlen` inside `get_args` and use `parser.error` to error out.

Solution

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

import argparse
import io
import random
import re
import sys


# --------------------------------------------------
def get_args():
    """parse arguments"""

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

    parser.add_argument('-l',
                        '--maxlen',
                        help='Max word length',
                        type=int,
                        default=10)

    parser.add_argument('-n',
                        '--minlen',
                        help='Min word length',
                        type=int,
                        default=5)

    parser.add_argument('-m',
                        '--misses',
                        help='Max number of misses',
                        type=int,
                        default=10)

    parser.add_argument('-s',
                        '--seed',
                        help='Random seed',
                        type=str,
                        default=None)

    parser.add_argument('-w',
```

```
                        '--wordlist',
                        help='Word list',
                        type=argparse.FileType('r'),
                        default='/usr/share/dict/words')

    parser.add_argument('-i',
                        '--inputs',
                        help='Input choices',
                        type=str,
                        default='')

    args = parser.parse_args()

    if args.minlen < 1:
        parser.error('--minlen "{}" must be positive'.format(args.minlen))

    if args.maxlen > 20:
        parser.error('--maxlen "{}" must be < 20'.format(args.maxlen))

    if args.minlen > args.maxlen:
        parser.error('--minlen "{}" is greater than --maxlen "{}"'.format(
            args.minlen, args.maxlen))

    return args


# --------------------------------------------------
def get_words(wordlist, min_len, max_len):
    """Read wordlist (file handle), return words in range(min_len, max_len)"""

    good = '^[a-z]{' + str(min_len) + ',' + str(max_len) + '}$'
    is_good = lambda w: re.match(good, w)
    return list(filter(is_good, wordlist.read().lower().split()))


# --------------------------------------------------
def test_get_words():
    """Test get_words"""

    text = 'Apple banana COW da epinephrine'
    assert get_words(io.StringIO(text), 1, 20) == text.lower().split()
    assert get_words(io.StringIO(text), 5, 10) == ['apple', 'banana']
    assert get_words(io.StringIO(text), 3, 10) == ['apple', 'banana', 'cow']


# --------------------------------------------------
```

```
def main():
    """main"""

    args = get_args()
    words = get_words(args.wordlist, args.minlen, args.maxlen)
    random.seed(args.seed)
    result = play({
        'word': random.choice(words),
        'max_misses': args.misses,
        'inputs': list(args.inputs),
    })

    print('You win!' if result else 'You lose, loser!')


# --------------------------------------------------
def play(state):
    """Play a round given a `dict` of the current state of the game"""

    word = state.get('word')
    if not word:
        print('No word!')
        return False

    guessed = state.get('guessed', list('_' * len(word)))
    prev_guesses = state.get('prev_guesses', set())
    num_misses = state.get('num_misses', 0)
    max_misses = state.get('max_misses', 10)
    inputs = state.get('inputs', [])

    if ''.join(guessed) == word:
        msg = 'You guessed "{}" with "{}" miss{}.'
        print(msg.format(word, num_misses, '' if num_misses == 1 else 'es'))
        return True

    if num_misses >= max_misses:
        print('The word was "{}."'.format(word))
        return False

    print('{} (Misses: {})'.format(' '.join(guessed), num_misses))

    get_char = lambda: input('Your guess? ("?" for hint, "!" to quit) ').lower()
    new_guess = inputs.pop(0) if inputs else get_char()

    if new_guess == '!':
        print('Better luck next time.')
```

```
        return False
    elif new_guess == '?':
        new_guess = random.choice([c for c in word if c not in guessed])
        num_misses += 1

    if not re.match('^[a-z]$', new_guess):
        print('"{}" is not a letter.'.format(new_guess))
        num_misses += 1
    elif new_guess in prev_guesses:
        print('You already guessed that.')
    elif new_guess in word:
        prev_guesses.add(new_guess)
        last_pos = 0
        while True:
            pos = word.find(new_guess, last_pos)
            if pos < 0:
                break
            elif pos >= 0:
                guessed[pos] = new_guess
                last_pos = pos + 1
    else:
        print('There is no "{}."'.format(new_guess))
        num_misses += 1

    return play({
        'word': word,
        'guessed': guessed,
        'num_misses': num_misses,
        'prev_guesses': prev_guesses,
        'max_misses': max_misses,
        'inputs': inputs,
    })


# --------------------------------------------------
def test_play():
    """Test play"""

    assert play({'word': 'banana', 'inputs': list('abn')}) == True
    assert play({'word': 'banana', 'inputs': list('abcdefghijklm')}) == False
    assert play({'word': 'banana', 'inputs': list('???')}) == True
    assert play({'word': 'banana', 'inputs': list('!')}) == False


# --------------------------------------------------
if __name__ == '__main__':
```

```
    main()
```

Discussion

As suggested in the specs, I put all the validation of user inputs into my `get_args` function, relying on `argparse` to validate that the `--wordlist` is a readable file and using `parser.error` to throw any problems with `--minlen` and `--maxlen`. By the time I have my `args` from `get_args`, I know I have all the right types and values for my inputs. I immediately set `random.seed` with the value of `args.seed` knowing that it is either a valid `int` or the `None` value which is essentially the same as not setting the seed.

Getting the words

This program assumes a dictionary-type file like the standard `/usr/share/dict/words` file with words and no punctuation, so I use `read().lower().split()` to read the wordlist argument which will be an open file handle because of how I defined it with `argparse`. I decided to write a small function to read the file:

```
def get_words(wordlist, min_len, max_len):
    good = '^[a-z]{' + str(min_len) + ',' + str(max_len) + '}$'
    is_good = lambda w: re.match(good, w)
    return list(filter(is_good, wordlist.read().lower().split()))
```

I can run my function to see if it works, faking an open file handle using `io.StringIO`:

```
>>> import re, io
>>> text = 'Apple banana COW da epinephrine'
>>> get_words(io.StringIO(text), 1, 20)
['apple', 'banana', 'cow', 'da', 'epinephrine']
>>> get_words(io.StringIO(text), 5, 10)
['apple', 'banana']
```

And then I can write a `test_get_words` function:

```
>>> def test_get_words():
...     text = 'Apple banana COW da epinephrine'
...     assert get_words(io.StringIO(text), 1, 20) == text.lower().split()
...     assert get_words(io.StringIO(text), 5, 10) == ['apple', 'banana']
...     assert get_words(io.StringIO(text), 3, 10) == ['apple', 'banana', 'cow']
...
```

Selecting a word

Once I have my `words`, I use `random.choice` to make a selection:

```
>>> import random
>>> random.seed(1)
```

```
>>> word = random.choice(get_words(io.StringIO(text), 1, 20))
>>> word
'banana'
```

Recursion vs Infinite Loops

From here, I could use the normal `while True` idiom to create an infinite loop from which I can `break` when the game is over or `continue` when I want to skip to the next iteration. For this example, I wanted to show how to write a *recursive* function – a function that calls itself, like a snake head eating the head on the opposite side. I have two reasons to do this:

1. I want to avoid using variables outside the loop to maintain the "state" of the program
2. Recursive functions are cool and it's fun to play with them

For what it's worth, my experience programming web interfaces with the Elm language (a dialect of Haskell that compiles to JavaScript) greatly influenced my decision to write the program this way. In Elm, there is a single `Model` that holds the state of the program where "state" means the values of everything in the program. There is a single `update` function that changes the state which is immediately followed by a `view` function to show the current state of the program.

Maintaining state

For our "Hangman," we need to keep track of the following:

1. The randomly selected word that is being guessed
2. The letters of the word which have been correctly guessed
3. The letters the user guessed which were not found in the word
4. The number of misses the user had made
5. The maximum number of misses the user is allowed
6. Any characters provided for the `inputs`

I *could* create 6 variables outside of a `while` loop and mutate those each time through the loop to know how the game is progressing. Instead, I created the function `play` and pass in a `dict` that holds all these values. It's perhaps the single longest function in the book running just over 60 lines. I usually try to keep every function short enough to fit into 80 characters wide and 50 lines long (the default size of my terminal windows). I strongly believe you should be able to see the entire function in one screen, but this one needs to be just a little longer.

Shall we play a game?

I wrote `play` to eventually `return` either `True` or `False` to indicate if the user won or not. I want to show you another way to debug a program. If you are in the same directory as your `hangman.py` program (and you created it with the `new.py` so that it has the `if __name__ == '__main__'` bit so that it won't immediately try to execute code), then you can actually `import` your entire program and `play` the game like so:

```
>>> import hangman
>>> hangman.play({'word': 'banana'})
_ _ _ _ _ _ (Misses: 0)
Your guess? ("?" for hint, "!" to quit) a
_ a _ a _ a (Misses: 0)
Your guess? ("?" for hint, "!" to quit) b
b a _ a _ a (Misses: 0)
Your guess? ("?" for hint, "!" to quit) n
You guessed "banana" with "0" misses.
True
```

Or pass in the `inputs`:

```
>>> hangman.play({'word': 'banana', 'inputs': list('ban')})
_ _ _ _ _ _ (Misses: 0)
b _ _ _ _ _ (Misses: 0)
b a _ a _ a (Misses: 0)
You guessed "banana" with "0" misses.
True
```

Now, how can I write a `play` to do that? The function takes just one argument which I call `state` which is a regular `dict` to hold key/value pairs. As I demonstrate above, not all of the 6 variables listed above need to be passed in for it to work. I use the `dict.get` function to return the `value` for a given `key` if the `key` exists, otherwise return a default value passed as the second argument:

```
>>> state = {'word': 'banana', 'inputs': list('ban')}
>>> guessed = state.get('guessed', list('_' * len(word)))
>>> guessed
['_', '_', '_', '_', '_', '_']
```

Here I represent the state of those letters that have been guessed or not as a string the same length as the `word` where there are underscores (`_`) for letters not yet guessed and those which have been guessed are present in their correct locations.

The previous guesses I store as a `set` because I only care about the unique letters:

```
>>> prev_guesses = state.get('prev_guesses', set())
```

```
>>> prev_guesses
set()
```

The default value for the number of misses is `0`:

```
>>> num_misses = state.get('num_misses', 0)
>>> num_misses
0
```

And the upper limit for maximum guesses is `10`:

```
>>> max_misses = state.get('max_misses', 10)
>>> max_misses
10
```

Finally, the `inputs` will default to an empty `list`:

```
>>> inputs = state.get('inputs', [])
>>> inputs
['b', 'a', 'n']
```

First I check if the `guessed` is the same as the `word`. If so, the user has won and I can `return True` to indicate this. Right now, this is not so:

```
>>> ''.join(guessed) == word
False
```

Then I check if they have guessed too many times:

```
>>> num_misses >= max_misses
False
```

So far, so good. Now I either want to get some character from the user for the next guessed letter or take it from the `inputs`:

```
>>> get_char = lambda: input('Your guess? ("?" for hint, "!" to quit) ').lower()
>>> new_guess = inputs.pop(0) if inputs else get_char()
>>> new_guess
'b'
```

The `lambda` is to create an function that I can call with `get_char()` if I need it. I could have written it all on one line, but I detest lines over 80 characters.

The user is allowed to exit the game early with a `!`, so I check if the `new_guess` is that and `return False` if so. They can also request a free letter with `?`. To select a free letter, I need to choose from the letters in the `word` if they are not present in the ones that have been `guessed`:

```
>>> new_guess = random.choice([c for c in word if c not in guessed])
>>> new_guess
'n'
```

At this point, I should have something from the user, whether they responded to the `input` or I took it from the `inputs` or they requested a hint. I need to verify that they gave me something that looks like just one character from the set `a-z`. I hope you immediately think of using a regular expression with a character class:

```
>>> re.match('^[a-z]$', new_guess)
<re.Match object; span=(0, 1), match='n'>
```

Where the caret (`^`) will anchor the regex to the start of the string, the `[a-z]` creates a character class comprised only of the lowercase letters from `a` to `z`, and the `$` anchors the pattern to the end of the string. Because I didn't indicate any number of matches with `*` (zero or more) or `+` (one or more) or `{n}` (`n` exactly), etc. it will match only one character. If this *fails* to match, then we do not have a valid input.

I check if `new_guess in prev_guesses` and let the user know if that is so. Then I check if `new_guess in word`. If so, they have guessed a letter correctly! I need to update the `guessed` list with the `new_guess` using the positions of that letter in the `word`. This is a bit tricky, and perhaps you chose to handle this differently. Here's what I do.

Let's say the `word='banana'` and `new_guess='n'`. I need to know the positions of `'n'` in `'banana'`:

```
>>> word
'banana'
>>> word.find('n')
2
```

Yes, there is an "n" at index 2:

```
>>> word[2]
'n'
```

So I can update `guessed` with that information:

```
>>> guessed
['_', '_', '_', '_', '_', '_']
>>> guessed[2] = 'n'
>>> guessed
['_', '_', 'n', '_', '_', '_']
```

If I call `word.find('n')` again, I'll get `2` again because it always starts searching from the beginning of the string. Note that `str.index` works the same way. I can pass a second optional argument to indicate the starting search index. Note that this need to be one greater than what we just found:

```
>>> word.find('n', 3)
4
```

I can see this is correct:

```
>>> word[4]
```

```
'n'
```

And can update guessed again:

```
>>> guessed[4] = 'n'
>>> guessed
['_', '_', 'n', '_', 'n', '_']
```

When I call word.find next, I'll get -1 to indicate the character is not found:

```
>>> word.find('n', 5)
-1
```

Contrast this with str.index to see that it creates an exception which is why we use find:

```
>>> word.index('n', 5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

If all the previous checks failed, then the new_guess was not found in the word, so I increment the num_misses and let the user know.

Finally, I call play again, passing it the new state as a new dict. Anytime a return statement is called, the recursion stops and execution returns to the first place I called play. Using the final return value which should be a bool, I can decide whether to print "You win!" or "You lose, loser!"

Testing the play

I can test my play with a function:

```
>>> from hangman import play
>>> def test_play():
...     assert play({'word': 'banana', 'inputs': list('abn')}) == True
...     assert play({'word': 'banana', 'inputs': list('abcdefghijklm')}) == False
...     assert play({'word': 'banana', 'inputs': list('???')}) == True
...     assert play({'word': 'banana', 'inputs': list('!')}) == False
...
>>> test_play()
...
```

Further

Here are some changes you could make to your program:

- Read a wordlist that has punctuation and use only a unique list for your words

- Add a limit to the number of hints the user can request with `?`
- Add a random insult every time the user asks for a hint
- Add a `quiet` flag to keep `play` from executing any `print` statements

Chapter 32: First Bank of Change

Write a Python program called `fboc.py` that will figure out all the different combinations of pennies, nickels, dimes, and quarters in a given `value` provided as a single positional argument. The value must be greater than 0 and less than or equal to 100. It should provide a usage if given no arguments or the `-h|--help` flag:

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

First Bank of Change

positional arguments:
  int         Sum

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

It should throw an error if the value is not greater than 0 and less than or equal to 100:

```
$ ./fboc.py 0
usage: fboc.py [-h] int
fboc.py: error: value "0" must be > 0 and <= 100
$ ./fboc.py 124
usage: fboc.py [-h] int
fboc.py: error: value "124" must be > 0 and <= 100
```

For valid values, it should print out all the combinations, always in order from largest to smalled denominations:

```
$ ./fboc.py 1
If you give me 1 cent, I can give you:
  1: 1 penny
$ ./fboc.py 4
If you give me 4 cents, I can give you:
  1: 4 pennies
$ ./fboc.py 6
If you give me 6 cents, I can give you:
  1: 6 pennies
  2: 1 nickel, 1 penny
$ ./fboc.py 13
If you give me 13 cents, I can give you:
  1: 13 pennies
```

```
  2: 1 dime, 3 pennies
  3: 1 nickel, 8 pennies
  4: 2 nickels, 3 pennies
$ ./fboc.py 27
If you give me 27 cents, I can give you:
  1: 27 pennies
  2: 1 quarter, 2 pennies
  3: 1 dime, 17 pennies
  4: 2 dimes, 7 pennies
  5: 1 nickel, 22 pennies
  6: 1 dime, 1 nickel, 12 pennies
  7: 2 dimes, 1 nickel, 2 pennies
  8: 2 nickels, 17 pennies
  9: 1 dime, 2 nickels, 7 pennies
 10: 3 nickels, 12 pennies
 11: 1 dime, 3 nickels, 2 pennies
 12: 4 nickels, 7 pennies
 13: 5 nickels, 2 pennies
```

Solution

```
#!/usr/bin/env python3
"""Coin combos for value"""

import argparse
from itertools import product
from functools import partial


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

    parser = argparse.ArgumentParser(
        description='First Bank of Change',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('value', metavar='int', type=int, help='Sum')

    args = parser.parse_args()

    if not 0 < args.value <= 100:
        parser.error('value "{}" must be > 0 and <= 100'.format(args.value))

    return args


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

    args = get_args()
    value = args.value
    nickels = range((value // 5) + 1)
    dimes = range((value // 10) + 1)
    quarters = range((value // 25) + 1)
    fig = partial(figure, value)
    combos = [c for c in map(fig, product(nickels, dimes, quarters)) if c]

    print('If you give me {} cent{}, I can give you:'.format(
        value, '' if value == 1 else 's'))

    for i, combo in enumerate(combos, 1):
        print('{:3}: {}'.format(i, fmt_combo(combo)))
```

```


# --------------------------------------------------
def fmt_combo(combo):
    """English version of combo"""

    out = []
    for coin, val in zip(('quarter', 'dime', 'nickel', 'penny'), combo):
        if val:
            plural = 'pennies' if coin == 'penny' else coin + 's'
            out.append('{} {}'.format(val, coin if val == 1 else plural))

    return ', '.join(out)


# --------------------------------------------------
def figure(value, coins):
    """
    If there is a valid combo of 'coins' in 'value',
    return a tuple of ints for (quarters, dimes, nickels, pennies)
    """

    nickels, dimes, quarters = coins
    big_coins = sum([5 * nickels, 10 * dimes, 25 * quarters])

    if big_coins <= value:
        return (quarters, dimes, nickels, value - big_coins)


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

Discussion

Let's start with a short look at `get_args` where I've decided to move the validation of the single `value` argument into this function rather than getting the arguments in `main` and checking there. We can use `argparse` to ensure the user provides an `int` value, but there's no `type` to say that it must be in our desired range; however, I can use the `parser.error` function on line 22 to trigger the normal fail-with-usage behaviour we normally get from `argparse`. From the standpoint of the calling code on line 32, all the work to coerce and validate the user happens in `get_args`. If we make it past line 32, then all must have been good and we can just focus on the task at hand.

I'd like to mention that I worked for a couple of days on this solution. I tried many different approaches before settling on the way I solved this problem, so what I do next may not be at all how you solved the problem. My idea was to find how many possible nickels, dimes, and quarters are in the given `value` and then find every combination of those values to see which ones sum to the `value` or less. To do this, I can use the `//` operator to find the integer division of the `value` by each of 5, 10, and 25 for nickels, dimes, and quarters, e.g.:

```
>>> value = 13
>>> value // 5
2
```

Finds there are two nickels in 13 cents. I construct a range that includes 0, 1, and 2 like so:

```
>>> nickels = range((value // 5) + 1)
>>> nickels
range(0, 3)
>>> list(nickels)
[0, 1, 2]
```

I used the `itertools.product` function and three ranges for nickels, dimes, and quarters to find every possible combination of every number of coins

```
>>> dimes = range((value // 10) + 1)
>>> quarters = range((value // 25) + 1)
>>> from itertools import product
>>> list(product(nickels, dimes, quarters))
[(0, 0, 0), (0, 1, 0), (1, 0, 0), (1, 1, 0), (2, 0, 0), (2, 1, 0)]
```

I want to include 0 of every coin so that I can make up the remainder in pennies. Let's jump ahead to the `figure` function to see how I wanted to use these values. Because `product` gives me a list of 3-tuples, I decided to pass `figure` the `value` and then a `coins` tuple that I unpack on line 66. I `sum` the values of the `nickels`, `dimes`, and `quarters` on line 67 and see if that is less than or equal to the `value`. If so, I get the number of pennies by subtracting the sum of the larger coins and return a 4-tuple with the number of each coin. If the previous sum was larger than the `value`, we don't bother defining the `return` of the function and so `None` is used.

Going back to line 37 where I want to call `figure` for each of the combinations returned by `product`, I use a list comprehension combined with a `map` which may seem rather dense but works quite well. The `map` wants a function and a list of items to apply the function. There's a slight problem in that the `figure` function wants 2 arguments – the `value` and the 3-tuple. I could have written the `map` using a `lambda`:

```
>>> def figure(value, coins):
...     nickels, dimes, quarters = coins
...     big_coins = sum([5 * nickels, 10 * dimes, 25 * quarters])
...     if big_coins <= value:
...         return (quarters, dimes, nickels, value - big_coins)
...
>>> list(map(lambda c: figure(value, c), product(nickels, dimes, quarters)))
[(0, 0, 0, 13), (0, 1, 0, 3), (0, 0, 1, 8), None, (0, 0, 2, 3), None]
```

But I thought it would be cleaner to create a partial application of the `figure` function with the `value` already bound. The `functools.partial` is exactly the tool we need and then we only need to pass in the 3-tuple of the coins:

```
>>> from functools import partial
>>> fig = partial(figure, value)
>>> fig((1,0,0))
(0, 0, 1, 8)
```

And so now I can use this `partial` function in my `map`:

```
>>> list(map(fig, product(nickels, dimes, quarters)))
[(0, 0, 0, 13), (0, 1, 0, 3), (0, 0, 1, 8), None, (0, 0, 2, 3), None]
```

Notice how we get some `None` values returned. Remember, this is because some of the combinations we are trying are too large, e.g., the maximum number of all the coins will be too large. So, to filter out those value, I can use a list comprehension with a guard at the end:

```
>>> combos = [c for c in map(fig, product(nickels, dimes, quarters)) if c]
>>> combos
[(0, 0, 0, 13), (0, 1, 0, 3), (0, 0, 1, 8), (0, 0, 2, 3)]
```

I could have used a `filter` for this, but it just doesn't seem to read as well:

```
>>> list(filter(lambda c: c, map(fig, product(nickels, dimes, quarters))))
[(0, 0, 0, 13), (0, 1, 0, 3), (0, 0, 1, 8), (0, 0, 2, 3)]
```

This is a list of 4-tuples representing the number of quarters, dimes, nickels, and pennies that will sum to `13`. We still need to report back to the user, so that is the purpose of the `fmt_combo` function. Given that 4-tuple, I want to report, e.g., "1 quarter" or "3 dimes", so I need to know the value of the denomination and the singular/plural versions of name of the denomination. I use the `zip` function to pair the coin denominations with their values:

```
>>> combo = (0, 0, 0, 13)
>>> list(zip(('quarter', 'dime', 'nickel', 'penny'), combo))
[('quarter', 0), ('dime', 0), ('nickel', 0), ('penny', 13)]
```

The `plural` version of each name is made by adding `s` except for `penny`, so line 53 handles that. If the denomination is not in the `combo` (e.g., here we have only pennies), then we skip those by using `if val` where `val` will be the number of coins. The integer value `0` will evaluate to `False` in a Boolean context, so only those with a non-zero value will be included. I decided to create a `list` of the strings for each denomination, so I `append` to that list the `val` plus the correct singular or plural version of the name, finally returning that list joined on comma-space (`', '`).

Finally lines 39-43 are left to formatting the report to the user, being sure to provide feedback that includes the original `value` ("If you give me …") and an enumerated list of all the possible ways we could make change. The test suite does not bother to check the order in which you return the combinations, only that the correct number are present and they are in the correct format.

Chapter 33: Modeling the Pareto Principle

Vilfredi Pareto was an Italian economist who noted in the late 1800s that roughly 80% of the land in Italy was owned by about 20% of the population. This 80/20 rule has been noted in many other contexts, but it stands out in wealth inequalities where it has tilted ever further to 90/10 or even 99/1. This exercise is designed to simulate the move of resources to an ever shrinking segment of a population through random events.

In our exercise, we will create variables to represent the following:

1. The number of actors in the simulation
2. The number of units of some resource
3. The percent of unequal distribution to stop the simulation
4. The number of iterations of the simulation to run

Create a program called `pareto.py` that accepts options for the number of `-a|--actors` (default `50`), the number of `-u|--units` (default `500`), the target `-d|--distribution` (default `0.8`), the number of `-r|--rounds` to simulate (default `10`). You also need to accept a `-s|--seed` option to pass to `random.seed` (default `None`) and an option to `-g|--graph` the result of each simulation as a text histogram to some file base name (default `None`).

The "resource" at play could be thought of as some measure of wealth or some resource like food. Ours is a zero-sum simulation where all the actors are randomly paired with each other and a coin is flipped to determine a winner. Each loser gives up one unit to the winner. Actors with no more units can no longer participate, so they cannot go into negative values ("debt") but neither can they ever re-enter the game.

The simulation

At the beginning of each simulation, the `--units` should evenly distributed to all the `--actors`. Create an infinite loop where the actors are shuffled and then paired up. I suggest you represent the actors as a `list` that you can pass to the `random.shuffle` function. To create the pairs, think about using the `range` function with a step value of `2` to get the position of every other actor, and then the mate for each will be the one just after it. "Flip" a coin by using `random.choice([True, False])`. If `True`, give the first player one unit and subtract one unit from the second; if `False,` vice versa. Consider using a `dict` to keep track of the number of units for each actor.

Each simulation is done when the designated `distribution` (default `0.8` or 80%) is controlled by `1 - distribution` (`0.2` or 20%). So when 80% of the units are controlled by 20% of the actors, return the number of iterations through the infinite loop it took to reach the target.

Calculating the distribution

If there are 10 actors and 10 units, the distribution starts out looking like this:

```
>>> units = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

You should stop when 2 actors control all the assets, so something like this:

```
>>> units = [0, 0, 7, 0, 0, 3, 0, 0, 0, 0]
```

Think about sorting the units from largest to smallest and then checking cumulative sums starting from the start. When the sum is greater than the target percentage, divide the number of values that went in by the total number of values to find the percentage controlling the target amount.

Consider adding a test, e.g., if the function is called `get_dist` then you can create the following to run with `pytest`:

```
def test_get_dist():
    """Test get_dist"""

    tests = [
        ([1, 1, 1, 1, 1, 1, 1, 1, 1, 1], 0.8, 0.8),
        ([2, 2, 2, 1, 0, 0, 0, 0, 0, 0], 0.8, 0.3),
        ([2, 2, 2, 2, 2, 0, 0, 0, 0, 0], 1.0, 0.5),
        ([0, 0, 7, 0, 0, 3, 0, 0, 0, 0], 0.8, 0.2),
        ([0, 0, 9, 0, 0, 1, 0, 0, 0, 0], 0.9, 0.1),
    ]

    for vals, perc, target in tests:
        dist = {k:v for k,v in enumerate(vals, 1)}
        assert get_dist(dist, perc) == target
```

Hints:

- For the `--graph` option, you should be able to use your `histy` code exactly.
- Consider make a function that runs one simulation, then call that function for the number in `--rounds`

Solution

```
#!/usr/bin/env python3
"""Simulation of Pareto distribution through random simulation"""

import argparse
import random


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

    parser = argparse.ArgumentParser(
        description='Argparse Python script',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('-a',
                        '--actors',
                        help='Number of actors',
                        metavar='int',
                        type=int,
                        default=50)

    parser.add_argument('-u',
                        '--units',
                        help='Number of units',
                        metavar='int',
                        type=int,
                        default=500)

    parser.add_argument('-d',
                        '--distribution',
                        help='Target distribution',
                        metavar='float',
                        type=float,
                        default=0.8)

    parser.add_argument('-r',
                        '--rounds',
                        help='Number of rounds',
                        metavar='int',
                        type=int,
                        default=10)

```

```
    parser.add_argument('-s',
                        '--seed',
                        help='Random seed',
                        metavar='int',
                        type=int,
                        default=None)

    parser.add_argument('-g',
                        '--graph',
                        help='Plot histograms to file',
                        metavar='FILE',
                        type=str,
                        default=None)

    return parser.parse_args()


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

    args = get_args()
    random.seed(args.seed)

    results = []
    for i in range(args.rounds):
        res, dist = sim(num_actors=args.actors,
                        num_units=args.units,
                        distribution=args.distribution)

        if args.graph:
            fh = open(args.graph + '-{}.txt'.format(i+1), 'wt')
            for units, actor in sorted([(v, k) for k, v in dist.items()]):
                fh.write('{:3}: {:3} {}\n'.format(actor, units, '#' * units))
            fh.close()

        print('{:3}: {} iterations'.format(i + 1, res))
        results.append(res)

    print('Average = {:,d} iterations'.format(int(sum(results) /
                                                  len(results))))


# --------------------------------------------------
def sim(num_actors, num_units, distribution):
    """Run a simulation"""
```

```

    actors = list(range(1, num_actors + 1))
    units_per_actor = int(num_units / num_actors)
    assert units_per_actor > 0, 'Not enough units per actor'
    dist = {actor: units_per_actor for actor in actors}
    rounds = 0

    while True:
        rounds += 1
        random.shuffle(actors)
        for i in range(0, len(actors), 2):
            a1, a2 = actors[i], actors[i + 1]
            if all([dist[a1], dist[a2]]):
                if random.choice([True, False]):
                    dist[a1] += 1
                    dist[a2] -= 1
                else:
                    dist[a1] -= 1
                    dist[a2] += 1

        if get_dist(dist, percentile=distribution) <= 1 - distribution:
            return rounds, dist

    return 0


# --------------------------------------------------
def get_dist(dist, percentile):
    """Calculate the distribution of units to actors"""

    values = sorted(list(dist.values()), reverse=True)
    total = sum(values)
    assert total > 0
    num_actors = len(values)

    for i in range(1, num_actors + 1):
        cum_sum = sum(values[:i])
        perc_actors = i / num_actors
        if cum_sum / total >= percentile:
            return i / num_actors

    return 0


# --------------------------------------------------
def test_get_dist():
```

```
    """Test get_dist"""

    tests = [
        ([1, 1, 1, 1, 1, 1, 1, 1, 1, 1], 0.8, 0.8),
        ([2, 2, 2, 1, 0, 0, 0, 0, 0, 0], 0.8, 0.3),
        ([2, 2, 2, 2, 2, 0, 0, 0, 0, 0], 1.0, 0.5),
        ([0, 0, 7, 0, 0, 3, 0, 0, 0, 0], 0.8, 0.2),
        ([0, 0, 9, 0, 0, 1, 0, 0, 0, 0], 0.9, 0.1),
    ]

    for vals, perc, target in tests:
        dist = {k:v for k,v in enumerate(vals, 1)}
        assert get_dist(dist, perc) == target


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

Discussion

As usual, I use `argparse` to validate all the user arguments and provide reasonable defaults such that the program can run with no input from the user. I pass the `--seed` argumnt directly to `random.seed` and then can worry about how to create and run my simulations.

I decided to create a function `sim` that I would call the correct number of `--round` using a `for` loop. The `sim` function needs to be passed the number of actors, the number of resources, and the target distribution to stop the simulation and will return the number of times through the simulation necessary to reach that target inequality of resource distribution. That is, if there are 10 actors and 10 units, then I stop when 8 units are controlled by no more than 2 actors.

The simulation

As suggested in the description, I make a `list` to represent the actors:

```
>>> num_actors = 10
>>> num_units = 10
>>> distribution = .8
>>> actors = list(range(1, num_actors + 1))
>>> actors
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

I need to figure out how many units to initially give each actor:

```
>>> units_per_actor = int(num_units / num_actors)
>>> units_per_actor
1
```

And then `assert` that there is something to distribute:

```
>>> assert units_per_actor > 0, 'Not enough units per actor'
```

I use a dictionary comprehension to create a `dict` that tracks the number of units for each actor:

```
>>> dist
{1: 1, 2: 1, 3: 1, 4: 1, 5: 1, 6: 1, 7: 1, 8: 1, 9: 1, 10: 1}
```

And then set up a `while True` infinite loop. Each time through the loop, I shuffle the actors:

```
>>> random.shuffle(actors)
>>> actors
[7, 1, 9, 2, 10, 3, 5, 6, 8, 4]
```

To create pairs, I use `range` to get every other position and then add `1` to that for the next actor:

```
>>> for i in range(0, len(actors), 2):
...     print(actors[i], actors[i + 1])
...
7 1
9 2
10 3
5 6
8 4
```

Actors can never go into negative values, so I need to ensure both actors still have units:

```
>>> all([dist[a1], dist[a2]])
True
```

If one has no more units, I simply move to the next pair. If they both have assests, I use `random.choice([True, False])` and give 1 unit to the first actor and take 1 from the second if `True`, vice versa if not.

The distribution

After doing this for all the pairs, I then check the resource distribution with a function called `get_dist` that takes the `dict` of actors/units and the target distribution. Suppose a distribution looks like this:

```
>>> dist
{1: 0, 2: 0, 3: 7, 4: 0, 5: 0, 6: 3, 7: 0, 8: 0, 9: 0, 10: 0}
```

I sort the `values` of the `dict` and find the `sum`

```
>>> values = sorted(list(dist.values()), reverse=True)
>>> values
[7, 3, 0, 0, 0, 0, 0, 0, 0, 0]
>>> total = sum(values)
>>> total
10
```

I figure out how many actors there are and start creating cumulative sums from the beginning of my sorted `values` list, counting how many actors are needed to account for whatever percentage of the total are present:

```
>>> percentile = .8
>>> num_actors = len(values)
>>> for i in range(1, num_actors + 1):
...     cum_sum = sum(values[:i])
...     perc_actors = i / num_actors
```

```
...     if cum_sum / total >= percentile:
...         print(i / num_actors)
...         break
...
0.2
```

I can put this into a function:

```
>>> import pareto, inspect
>>> print(inspect.getsource(pareto.get_dist))
def get_dist(dist, percentile):
    """Calculate the distribution of units to actors"""

    values = sorted(list(dist.values()), reverse=True)
    total = sum(values)
    assert total > 0
    num_actors = len(values)

    for i in range(1, num_actors + 1):
        cum_sum = sum(values[:i])
        perc_actors = i / num_actors
        if cum_sum / total >= percentile:
            return i / num_actors

    return 0
```

And test it using the `test_get_dist` function shown earlier.

Graphing the distribution

Data visualization is a vital part of checking the accuracy of your work. You can use the code you wrote for `histy` to see that, indeed, 80% of the actors have nothing while 20% control everything all due to nothing more than random coin flips:

```
$ ./pareto.py -r 1 -a 10 -u 10 -g graph
  1: 50 iterations
Average = 50 iterations
$ cat graph-1.txt
  1:   0
  2:   0
  4:   0
  5:   0
  6:   0
  7:   0
  8:   0
 10:   0
```

```
  9:   2 ##
  3:   8 ########
```

If you run it again, you will most like see that some other two actors were the winners:

```
$ ./pareto.py -r 1 -a 10 -u 10 -g graph
  1: 97 iterations
Average = 97 iterations
$ cat graph-1.txt
  1:   0
  3:   0
  4:   0
  6:   0
  7:   0
  8:   0
  9:   0
 10:   0
  5:   2 ##
  2:   8 ########
```

It doesn't take long for an even distribution to become very skewed. Imagine how much more quickly the imbalance would be achieved if the resources were unevenly distributed at the beginning!

More

- Find a way to animate the changes to the histogram during each challenge inside the simulation; e.g., `matplotlib.animation` or create a series of GIFs or PNGs that you stitch together to create a short movie to visualize how the resource distribution changes over time.

Chapter 34: Runny Babbit

Are you familiar with Spoonerisms where the initial consonant sounds of two words are switched? According to Wikipedia, they get their name from William Archibald Spooner who did this often. The author Shel Silverstein wrote a wonderful book called *Runny Babbit* ("bunny rabbit") based on this. So, let's write a Python program called `runny_babbit.py` that will read some text or an input file given as a single positional argument and finds neighboring words with initial consonant sounds to swap. As we'll need to look at pairs of words and in such as way that it will make it difficult to remember the original formatting of the text, let's also take a `-w|--width` (default `70`) to format the output text to a maximum width.

As usual, the program should show usage with no arguments or for `-h|--help`:

```
$ ./runny_babbit.py
usage: runny_babbit.py [-h] [-w int] str
runny_babbit.py: error: the following arguments are required: str
$ ./runny_babbit.py -h
usage: runny_babbit.py [-h] [-w int] str

Introduce Spoonerisms

positional arguments:
  str                  Input text or file

optional arguments:
  -h, --help           show this help message and exit
  -w int, --width int  Output text width (default: 70)
```

It should handle text from the command line:

```
$ ./runny_babbit.py 'the bunny rabbit'
the runny babbit
```

Or a named file:

```
$ cat input1.txt
The bunny rabbit is cute.
$ ./runny_babbit.py input1.txt
The runny babbit is cute.
```

We'll use a set of "stop" words to prevent the switching of sounds when one of the words is in the following list:

before behind between beyond but by concerning despite down during following for from into like near plus since that the through throughout to towards which with within without

The results are endlessly entertaining:

```
$ ./runny_babbit.py ../inputs/preamble.txt
When, in the course of human events, it necomes becessary for one
people to dissolve the bolitical pands hich whave thonnected cem with
another, and to assume among the powers of the earth, the separate and
equal station to which the laws of nature and of Gature's nod entitle
them, a recent despect to the opinions of rankind mequires that shey
thould declare the whauses cich impel them to the separation.
```

Hints:

- You'll need to consider all the words in the input as pairs, like `[(0, 1), (1, 2)]` up to `n` (number of words) etc. How can you create such a list where instead of `0` and `1` you have the actual words, e.g., `[('The', 'bunny'), ('bunny', 'rabbit')]`?
- There are several exercises where we try to break words into initial consonant sounds and whatever else that follows. Can you reuse code from elsewhere? I'd recommend using regular expressions!
- Be sure you don't use a word more than once in a swap. E.g., in the phrase "the brown, wooden box", we'd skip "the" and consider the other two pairs of words `('brown', 'wooden')` and `('wooden', 'box')`. If we swap the first pair to make `('wown', 'brooden')`, we would not want to consider the next pair because 'wooden' has already been used.
- Use the `textwrap` module to handle the formatting of the ouput text to a maximum `--width`

Solution

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

import argparse
import os
import re
import string
import textwrap


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

    parser = argparse.ArgumentParser(
        description='Introduce Spoonerisms',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('text',
                        metavar='str',
                        help='Input text or file')

    parser.add_argument('-w',
                        '--width',
                        help='Output text width',
                        metavar='int',
                        type=int,
                        default=70)

    args = parser.parse_args()

    if os.path.isfile(args.text):
        args.text = open(args.text).read()

    return args


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

    args = get_args()
    text = args.text
```

```
    words = text.split()
    pairs = []

    for k in range(len(words) - 1):
        pairs.append((words[k], words[k+1]))

    vowels = 'aeiouAEIOU'
    consonants = ''.join([c for c in string.ascii_letters if c not in vowels])
    regex = re.compile('^([' + consonants + ']+)([' + vowels + '].*)')
    stop = set('before behind between beyond but by concerning'
               'despite down during following for from into like near'
               'plus since that the through throughout to towards'
               'which with within without'.split())
    skip = set()

    for i, pair in enumerate(pairs):
        w1, w2 = pair
        if set([w1.lower(), w2.lower()]).intersection(stop):
            continue

        i1, i2 = i, i + 1
        if i1 in skip or i2 in skip:
            continue

        m1 = regex.search(w1)
        m2 = regex.search(w2)
        if m1 and m2:
            prefix1, suffix1 = m1.groups()
            prefix2, suffix2 = m2.groups()
            words[i1] = prefix2 + suffix1
            words[i2] = prefix1 + suffix2
            skip.add(i1)
            skip.add(i2)

    print('\n'.join(textwrap.wrap(' '.join(words), width=args.width)))

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

Discussion

Figure 13: Also definitely not copyright infringement.

For this exercise, I thought I might move the logic to read an optionally named input *file* into the `get_args` function so that by the time I call `args = get_args()` the `args.text` really is just whatever "text" I need to consider, regardless if the source was the command line or a file. If I'm using `input1.txt`, then I essentially have this:

```
>>> text = open('input1.txt').read()
>>> text
'The bunny rabbit is cute.\n'
```

I need all the pairs of words, so that means I first need all the "words" which I'll get by naively using `str.split` (that is, I won't worry about punctation and such):

```
>>> words = text.split()
>>> words
['The', 'bunny', 'rabbit', 'is', 'cute.']
```

Now I need all *pairs* of words which I can get by going from the zeroth word to the second to last word:

```
>>> pairs = []
>>> for k in range(len(words) - 1):
...     pairs.append((words[k], words[k+1]))
...
>>> pairs
[('The', 'bunny'), ('bunny', 'rabbit'), ('rabbit', 'is'), ('is', 'cute.')
```

I need to find all the pairs where both words start with some consonant sounds and where neither of them is in my stop list, which I'll create like so:

```
>>> stop = set('before behind between beyond but by concerning'
...            'despite down during following for from into like near'
...            'plus since that the through throughout to towards'
...            'which with within without'.split())
```

How will I find words that start with consonants? I can easily list all the vowels:

```
>>> vowels = 'aeiouAEIOU'
```

And then create the complement from `string.ascii_lowercase`:

```
>>> import string
>>> consonants = ''.join([c for c in string.ascii_letters if c not in vowels])
>>> consonants
'bcdfghjklmnpqrstvwxyzBCDFGHJKLMNPQRSTVWXYZ'
```

And then build a regular expression that looks for the start of a string ^ followed by a character class of all the `consonants` followed by the character class of `vowels` maybe followed by something else. I'll use parentheses `()` to capture both parts:

```
>>> import re
>>> regex = re.compile('^([' + consonants + ']+)([' + vowels + '].*)')
>>> regex.search('chair')
<re.Match object; span=(0, 5), match='chair'>
>>> regex.search('chair').groups()
('ch', 'air')
```

Now I can iterate over the `pairs`. First I check if the either of the words is in the `stop` set by using the `set.intersection` function. For the first pair `('The', 'bunny')` we see there is an intersection:

```
>>> w1 = 'The'
>>> w2 = 'bunny'
>>> set([w1.lower(), w2.lower()]).intersection(stop)
{'the'}
```

For the next pair, there is not:

```
>>> w1 = 'bunny'
>>> w2 = 'rabbit'
>>> set([w1.lower(), w2.lower()]).intersection(stop)
set()
```

The next check in my code is whether I've previously determined that I need to skip these words, so I have to know their positions in the original list. I decided to use `enumerate` over the `words` to get the number of the pair which will equal the position of the first word of each tuple in the original list of `words`.

Next I need to see if *both* words match my regular expression:

```
>>> m1 = regex.search(w1)
```

```
>>> m2 = regex.search(w2)
>>> m1
<re.Match object; span=(0, 5), match='bunny'>
>>> m2
<re.Match object; span=(0, 6), match='rabbit'>
```

They do! So I can use their `groups` to get the parts of each word to swap:

```
>>> m1.groups()
('b', 'unny')
>>> m2.groups()
('r', 'abbit')
>>> prefix1, suffix1 = m1.groups()
>>> prefix2, suffix2 = m2.groups()
```

This is the 2nd pair, so `i` would be equal to `1` in the actual code. I can use this to go mutate the `words` at positions `i` and `i + 1`:

```
>>> i = 1
>>> words[i] = prefix2 + suffix1
>>> words[i + 1] = prefix1 + suffix2
>>> words
['The', 'runny', 'babbit', 'is', 'cute.']
```

I need to be sure to add those positions to the `skip` set I created for the check that I discussed just above.

Finally we need to `print` the `words` back out, joining them on a blank and using `textwrap.wrap` with the `--width` argument to make it pretty:

```
>>> import textwrap
>>> print('\n'.join(textwrap.wrap(' '.join(words), width=70)))
The runny babbit is cute.
```

Chapter 35: Markov Chain

Write a Python program called `markov.py` that takes one or more text files as positional arguments for training. Use the `-n|--num_words` argument (default `2`) to find clusters of words and the words that follow them, e.g., in "The Bustle" by Emily Dickinson:

```
The bustle in a house
The morning after death
Is solemnest of industries
Enacted upon earth,-

The sweeping up the heart,
And putting love away
We shall not want to use again
Until eternity.
```

If `n=1`, then we find that "The" can be followed by "bustle," "morning," and "sweeping." There is a "the" followed by "heart," but we're not going to alter the text in any way, including removing punctuation, so just use `str.split` on the text to break up the words.

To begin your text, choose a random word (or words) that begin with an uppercase letter. Then randomly select the next word in the chain, keep track of the floating window of the `-n` words, and keep selecting the next words until you have matched or exceeded the `-l|--length` argument of the number of characters (default 500) to emit at which point you should stop when you find a word that terminates with `.`, `!`, or `?`.

If you use `str.split` to get the words from the training text, you'll be removing any newlines from the text, so use a `-w|--text_width` argument (default 70) to introduce newlines in the output before the text exceeds that number of characters on the line. I recommend you use the `textwrap` module for this.

Because of the use of randomness, you should include a `-s|--seed` argument (default `None`) to pass to `random.seed`.

Occassionally you may chose a path that terminates. That is, in selecting the next word, you may find there is no next-next word. In that case, just exit the program.

My implementation includes a `-d|--debug` option that will write a `.log` file so you can inspect my data structures and logic as you write your own version.

You should find many diverse texts and use them all as training files with varying numbers for `-n` to see how the texts will be mixed. The results are endlessly entertaining.

```
$ ./markov.py
usage: markov.py [-h] [-l int] [-n int] [-s int] [-w int] [-d] FILE [FILE ...]
```

```
markov.py: error: the following arguments are required: FILE
$ ./markov.py -h
usage: markov.py [-h] [-l int] [-n int] [-s int] [-w int] [-d] FILE [FILE ...]

Markov Chain

positional arguments:
  FILE                  Training file(s)

optional arguments:
  -h, --help            show this help message and exit
  -l int, --length int  Output length (characters) (default: 500)
  -n int, --num_words int
                        Number of words (default: 2)
  -s int, --seed int    Random seed (default: None)
  -w int, --text_width int
                        Max number of characters per line (default: 70)
  -d, --debug           Debug to ".log" (default: False)
$ ./markov.py ../inputs/const.txt -s 1
States, shall have no Vote, unless they shall meet in their respective
Numbers, which shall abridge the privileges or immunities of citizens
of the Militia to execute the Laws thereof, escaping into another,
shall, in the land and naval Forces; To provide for the loss or
emancipation of any slave; but all such Laws shall be bound thereby,
any Thing in the case wherein neither a President or Vice President
and Vice-President, or hold any office, civil or military, under the
United States; he may adjourn them to such Time as he shall have
failed to qualify, then the Vice-President chosen for the purpose
shall consist of a term to which the United States of America.
$ ./markov.py -s 2 ../inputs/dickinson.txt -w 30 -l 100
His knowledge to unfold On
what concerns our mutual mind,
The literature of old; What
interested scholars most, What
competitions ran When Plato
was a living girl, And
Beatrice wore The gown that
Dante deified.
```

Solution

```
#!/usr/bin/env python3
"""Markov Chain"""

import argparse
import logging
import random
import textwrap
from pprint import pformat as pf
from collections import defaultdict


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

    parser = argparse.ArgumentParser(
        description='Markov Chain',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('training',
                        metavar='FILE',
                        nargs='+',
                        type=argparse.FileType('r'),
                        help='Training file(s)')

    parser.add_argument('-l',
                        '--length',
                        help='Output length (characters)',
                        metavar='int',
                        type=int,
                        default=500)

    parser.add_argument('-n',
                        '--num_words',
                        help='Number of words',
                        metavar='int',
                        type=int,
                        default=2)

    parser.add_argument('-s',
                        '--seed',
                        help='Random seed',
                        metavar='int',
```

```
                        type=int,
                        default=None)

    parser.add_argument('-w',
                        '--text_width',
                        help='Max number of characters per line',
                        metavar='int',
                        type=int,
                        default=70)

    parser.add_argument('-d',
                        '--debug',
                        help='Debug to ".log"',
                        action='store_true')

    return parser.parse_args()


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

    args = get_args()
    char_max = args.length
    random.seed(args.seed)
    num_words = args.num_words

    logging.basicConfig(
        filename='.log',
        filemode='w',
        level=logging.DEBUG if args.debug else logging.CRITICAL)

    training = read_training(args.training, num_words)
    logging.debug('training = %s', pf(training))

    # Find a word starting with a capital letter
    words = list(
        random.choice(
            list(filter(lambda t: t[0][0].isupper(), training.keys()))))

    logging.debug('starting with "%s"', words)
    logging.debug(training[tuple(words)])

    while True:
        # get last two words
        prev = tuple(words[-1 * num_words:])
```

```

        # bail if dead end
        if not prev in training:
            break

        new_word = random.choice(training[prev])
        logging.debug('chose "{}" from {}'.format(new_word, training[prev]))
        words.append(new_word)

        # try to find ending punctuation if we've hit wanted char count
        char_count = sum(map(len, words)) + len(words)
        if char_count >= char_max and new_word[-1] in '.!?':
            break

    print('\n'.join(textwrap.wrap(' '.join(words), width=args.text_width)))
    logging.debug('Finished')


# --------------------------------------------------
def read_training(fhs, num_words):
    """Read training files, return dict of chains"""

    all_words = defaultdict(list)
    for fh in fhs:
        words = fh.read().split()

        for i in range(0, len(words) - num_words):
            l = words[i:i + num_words + 1]
            all_words[tuple(l[:-1])].append(l[-1])

    return all_words


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

Discussion

As usual, I like to start my program by defining the options to my program with `get_args`. There will be one or more positional arguments which are `training` files, so I defined this argument with `narg='+'` and the `type=argparse.FileType('r')` so that `argparse` will validate the user input. Per the README, I define four other `int` arguments for the `--length` of the output, the `--num_words` in the patterns, the random `--seed`, and the `--text_width` of each line of output.

I also define a `--debug` option that will turn on `logging` to a `.log` file. Lines 71-74 initialize the `logging` module with `filemode='w'` so that it will overwrite an existing file and only emitting `DEBUG`-level messages if `--debug` is present; otherwise, only `CRITICAL` messages are shown and, since I have no calls to `logging.critical`, nothing will go into the logfile.

On line 76, I call a function to read the training files which I pass as a `list` as the first argument and the `args.num_words` as the second. While I could have put these few lines of code in the `main`, I prefer having short functions that do one thing. One of the hardest things to figure out for this program was the data structure I needed to represent a Markov chain. I settled on using a `dict` that would have as keys a tuple of word pairs and as values a list of words that follow that word pair. I call this `all_words` on line 114 and create it using the `collections.defaultdict(list)`. The advantage to `defaultdict` is that keys are created automatically using a default value for the indicated data type – an empty string for `str`, the value `0` for `int`, and the empty list `[]` for `list`. (If you're into category theory, these are the "empty" values for the monoids of strings, integers, and lists.)

On line 115, I iterate over the file handles that `argparse` opened for me. Note I call each file handle `fh` and the list of file handles `fhs` (the "plural" of `fh`). On line 116, the call to `fh.read().split()` will read the *entire* file and split it into "words" which I quote because I'm specifically not removing any sort of punctuation as I decided to follow the example in the Kernighan/Pike book where quotes and punctuation from the original text will determine the same kinds of patterns in the resulting text. Of course, this can lead to mismatched quotes and randomly distributed punctuation, but c'est la vie.

To create the chains, I want to select continuous sequences of words of length `--num_words` plus the word that follows. So, if `--num_words` is `1`, I will use the first word as my key, then look ahead at the next word as a choice of what can come next. Given a phrase like "The Lion, The Witch and The Wardrobe", we can see that "The" may be followed by either "Lion," "Witch," or "Wardrobe." To write this in code, we use the same idea as extracting k-mers from a word but instead think of "mers" as words and select "k-words" from a string:

```
>>> words = 'The Lion, The Witch and The Wardrobe'.split()
>>> words
```

```
['The', 'Lion,', 'The', 'Witch', 'and', 'The', 'Wardrobe']
>>> from collections import defaultdict
>>> all_words = defaultdict(list)
>>> num_words = 1
>>> for i in range(0, len(words) - num_words):
...     l = words[i:i + num_words + 1]
...     all_words[tuple(l[:-1])].append(l[-1])
```

In the resulting `all_words` structure, we see that 'The' has the expected three options:

```
>>> from pprint import pprint as pp
>>> pp(all_words)
defaultdict(<class 'list'>,
            {('Lion,',): ['The'],
             ('The',): ['Lion,', 'Witch', 'Wardrobe'],
             ('Witch',): ['and'],
             ('and',): ['The']})
```

In creating the list of options, I chose not to unique the values. Some texts may have the same word following a given sequence many times which will result in that word being randomly selected more often, but this is a consequence of the training data influencing the outcome. If you used a `set` instead of a `list`, you would lose that influence. The data structure matters!

On line 80, I need to find a place to start. I use `random.choice` to select from the list of `training` words that start with a capital letter. I can `filter` the the keys of the `training` dict which you should recall are tuples. In the `lambda`, I reference `t[0][0]` (`t` for "tuple") to index the zeroth element in the tuple and then the zeroth character of that word. This will return a `str` object which I can use to call the `isupper` method to tell me if the character is an uppercase letter. Remember that `filter` will only allow to pass those elements for which the `lambda` evaluates to `True`.

The `while True` begins the actual generation of text. I get the previous `num_words` by multiplying that value by `-1` and indexing from the end of the `words` list. I need to turn that list into a `tuple` to use for the key to `training`. (A note here that you cannot use a `list` as a key to a `dict` because it's not immutable whereas strings and tuples are.) I need to `break` out of the loop if I happened down deadend; otherwise, I can use `random.choice` again to select a new word from the list of options and `append` that to the `words` I've selected already.

To figure out if we've gone far enough, I need to count up how many characters I've got in `words`, so I `map` the `words` into the `len` function and `sum` them up:

```
>>> words
['The', 'Lion,', 'The', 'Witch', 'and', 'The', 'Wardrobe']
>>> sum(map(len, words))
```

```
30
```

But there will be spaces in between each word, so I account for them by adding on the `len(words)`. If I have matched or exceeded the `char_max`, then I need to find a stopping point by looking to see if the `new_word` I've selected ends with an ending punctuation like `.`, `!`, or `?`. If so, we `break` out of the loop.

At this point, the `words` list needs to be turned into text. It would be ugly to just `print` out one long string, so I use the `textwrap.wrap` to break up the long string into lines that are no longer than the given `text_width`. That function returns a list of lines that need to be joined on newlines to print.

Chapter 36: Hamming Chain

Write a Python program called `chain.py` that takes a `-s|--start` word and searches a `-w|--wordlist` argument (default `/usr/local/share/dict`) for words no more than `-d|--max_distance` Hamming distance for some number of `-i|--iteration` (default 20). Be sure to accept a `-S|--seed` for `random.seed`.

If the given word is not found in the word list, exit with an error and message. While searching for the next word in the chain, be sure not to repeat any words previously found or you might just go in circles! If you fail to find any new words before the end of the iterations, exit with an error and message as such.

```
$ ./chain.py -h
usage: chain.py [-h] [-s START] [-w FILE] [-d int] [-i int] [-S int] [-D]

Hamming chain

optional arguments:
  -h, --help            show this help message and exit
  -s START, --start START
                        Starting word (default: )
  -w FILE, --wordlist FILE
                        File input (default: /usr/share/dict/words)
  -d int, --max_distance int
                        Maximum Hamming distance (default: 1)
  -i int, --iterations int
                        Random seed (default: 20)
  -S int, --seed int    Random seed (default: None)
  -D, --debug           Debug (default: False)
$ ./chain.py -s foobar
Unknown word "foobar"
$ ./chain.py -s bike -S 1 -i 5
  1: bike
  2: bikh
  3: Sikh
  4: sith
  5: sithe
$ ./chain.py -s bike -S 1 -i 5 -d 2
  1: bike
  2: bit
  3: net
  4: yot
  5: ye
$ ./chain.py -S 1 -s bicycle
Failed to find more words!
```

```
  1: bicycle
  2: bicycler
$ ./chain.py -S 1 -s bicycle -d 2 -i 5
  1: bicycle
  2: bicyclic
  3: bicyclism
  4: dicyclist
  5: bicyclist
```

Use the `uscities.txt` file to plan a trip!

```
$ ./chain.py -S 1 -w ../inputs/uscities.txt -s Clinton -d 3
  1: Clinton
  2: Flint
  3: Fritz
  4: Unity
  5: Union
  6: Mason
  7: Oasis
  8: Nash
  9: Zag
 10: Guy
 11: Gaza
 12: Jay
 13: Ely
 14: Egan
 15: Aden
 16: Alta
 17: Ada
 18: Nyac
 19: Pyatt
 20: Plato
$ ./chain.py -S 1 -w ../inputs/uscities.txt -s 'Calumet City' -d 4
Failed to find more words!
  1: Calumet City
  2: Calumet Park
  3: Palomar Park
  4: Hanover Park
  5: Langley Park
  6: Stanley Park
  7: Kearney Park
```

Solution

```
#!/usr/bin/env python3
"""Hamming chain"""

import argparse
import logging
import random
import re
from dire import die, warn


# --------------------------------------------------
def get_args():
    """get command-line arguments"""

    parser = argparse.ArgumentParser(
        description='Hamming chain',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('-s', '--start', type=str, help='Starting word', default='')

    parser.add_argument('-w',
                        '--wordlist',
                        metavar='FILE',
                        type=argparse.FileType('r'),
                        help='File input',
                        default='/usr/share/dict/words')

    parser.add_argument('-d',
                        '--max_distance',
                        metavar='int',
                        type=int,
                        help='Maximum Hamming distance',
                        default=1)

    parser.add_argument('-i',
                        '--iterations',
                        metavar='int',
                        type=int,
                        help='Random seed',
                        default=20)

    parser.add_argument('-S',
                        '--seed',
```

```
                        metavar='int',
                        type=int,
                        help='Random seed',
                        default=None)

    parser.add_argument('-D', '--debug', help='Debug', action='store_true')

    return parser.parse_args()


# --------------------------------------------------
def dist(s1, s2):
    """Given two strings, return the Hamming distance (int)"""

    return abs(len(s1) - len(s2)) + sum(
        map(lambda p: 0 if p[0] == p[1] else 1, zip(s1.lower(), s2.lower())))


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

    args = get_args()
    start = args.start
    fh = args.wordlist
    distance = args.max_distance

    random.seed(args.seed)

    logging.basicConfig(
        filename='.log',
        filemode='w',
        level=logging.DEBUG if args.debug else logging.CRITICAL)

    logging.debug('file = %s', fh.name)

    words = fh.read().splitlines()

    if not start:
        start = random.choice(words)

    if not start in words:
        die('Unknown word "{}"'.format(start))

    def find_close(word):
        l = len(word)
```

```
        low, high = l - distance, l + distance
        test = filter(lambda w: low <= len(w) <= high, words)
        return filter(lambda w: dist(word, w) <= distance, test)

    chain = [start]
    for _ in range(args.iterations - 1):
        close = list(filter(lambda w: w not in chain, find_close(chain[-1])))
        if not close:
            warn('Failed to find more words!')
            break

        next_word = random.choice(close)
        chain.append(next_word)

    for i, link in enumerate(chain, start=1):
        print('{:3}: {}'.format(i, link))

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

Chapter 37: Morse Encoder/Decoder

Write a Python program called `morse.py` that will encrypt/decrypt text to/from Morse code. The program should expect a single positional argument which is either the name of a file to read for the input or the character `-` to indicate reading from STDIN. The program should also take a `-c|--coding` option to indicate use of the `itu` or standard `morse` tables, `-o|--outfile` for writing the output (default STDOUT), and a `-d|--decode` flag to indicate that the action is to decode the input (the default is to encode it).

```
$ ./morse.py
usage: morse.py [-h] [-c str] [-o str] [-d] [-D] FILE
morse.py: error: the following arguments are required: FILE
$ ./morse.py -h
usage: morse.py [-h] [-c str] [-o str] [-d] [-D] FILE

Encode and decode text/Morse

positional arguments:
  FILE                  Input file or "-" for stdin

optional arguments:
  -h, --help            show this help message and exit
  -c str, --coding str  Coding version (default: itu)
  -o str, --outfile str
                        Output file (default: None)
  -d, --decode          Decode message from Morse to text (default: False)
  -D, --debug           Debug (default: False)
$ ./morse.py ../inputs/fox.txt
- .... .  --.- ..- .. -.-. -.-  -... .-. --- .-- -.  ..-. --- -..-  .--- ..- -- .--. ...  --
[cholla@~/work/python/playful_python/morse]$ ./morse.py ../inputs/fox.txt | ./morse.py -d -
THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.
```

Solution

```
#!/usr/bin/env python3
"""Morse en/decoder"""

import argparse
import logging
import random
import re
import string
import sys


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

    parser = argparse.ArgumentParser(
        description='Encode and decode text/Morse',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('input',
                        metavar='FILE',
                        help='Input file or "-" for stdin')

    parser.add_argument('-c',
                        '--coding',
                        help='Coding version',
                        metavar='str',
                        type=str,
                        choices=['itu', 'morse'],
                        default='itu')

    parser.add_argument('-o',
                        '--outfile',
                        help='Output file',
                        metavar='str',
                        type=str,
                        default=None)

    parser.add_argument('-d',
                        '--decode',
                        help='Decode message from Morse to text',
                        action='store_true')

```

```
    parser.add_argument('-D', '--debug', help='Debug', action='store_true')

    return parser.parse_args()


# --------------------------------------------------
def encode_word(word, table):
    """Encode word using given table"""

    coded = []
    for char in word.upper():
        logging.debug(char)
        if char != ' ' and char in table:
            coded.append(table[char])

    encoded = ' '.join(coded)
    logging.debug('endoding "{}" to "{}"'.format(word, encoded))

    return encoded


# --------------------------------------------------
def decode_word(encoded, table):
    """Decode word using given table"""

    decoded = []
    for code in encoded.split(' '):
        if code in table:
            decoded.append(table[code])

    word = ''.join(decoded)
    logging.debug('dedoding "{}" to "{}"'.format(encoded, word))

    return word


# --------------------------------------------------
def test_encode_word():
    """Test Encoding"""

    assert encode_word('sos', ENCODE_ITU) == '... --- ...'
    assert encode_word('sos', ENCODE_MORSE) == '... .,. ...'


# --------------------------------------------------
def test_decode_word():
```

```
    """Test Decoding"""

    assert decode_word('... --- ...', DECODE_ITU) == 'SOS'
    assert decode_word('... .,. ...', DECODE_MORSE) == 'SOS'


# --------------------------------------------------
def test_roundtrip():
    """Test En/decoding"""

    random_str = lambda: ''.join(random.sample(string.ascii_lowercase, k=10))
    for _ in range(10):
        word = random_str()
        for encode_tbl, decode_tbl in [(ENCODE_ITU, DECODE_ITU),
                                       (ENCODE_MORSE, DECODE_MORSE)]:

            assert word.upper() == decode_word(encode_word(word, encode_tbl),
                                               decode_tbl)


# --------------------------------------------------
def main():
    """Make a jazz noise here"""
    args = get_args()
    action = 'decode' if args.decode else 'encode'
    output = open(args.outfile, 'wt') if args.outfile else sys.stdout
    source = sys.stdin if args.input == '-' else open(args.input)

    coding_table = ''
    if args.coding == 'itu':
        coding_table = ENCODE_ITU if action == 'encode' else DECODE_ITU
    else:
        coding_table = ENCODE_MORSE if action == 'encode' else DECODE_MORSE

    logging.basicConfig(
        filename='.log',
        filemode='w',
        level=logging.DEBUG if args.debug else logging.CRITICAL)

    word_split = r'\s+' if action == 'encode' else r'\s{2}'

    for line in source:
        for word in re.split(word_split, line):
            if action == 'encode':
                print(encode_word(word, coding_table), end='  ')
            else:
```

```
                print(decode_word(word, coding_table), end=' ')
        print()


# --------------------------------------------------
def invert_dict(d):
    """Invert a dictionary's key/value"""

    #return dict(map(lambda t: list(reversed(t)), d.items()))
    return dict([(v, k) for k, v in d.items()])


# --------------------------------------------------
# GLOBALS

ENCODE_ITU = {
    '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': '--..', '0': '-----', '1': '.----', '2': '..---', '3':
    '...--', '4': '....-', '5': '.....', '6': '-....', '7': '--...', '8':
    '---..', '9': '----.', '.': '.-.-.-', ',': '--..--', '?': '..--..', '!':
    '-.-.--', '&': '.-...', ';': '-.-.-.', ':': '---...', "'": '.----.', '/':
    '-..-.', '-': '-....-', '(': '-.--.', ')': '-.--.-',
}

ENCODE_MORSE = {
    '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': '...,.', '0': '+++++', '1': '.--.', '2': '..-..', '3':
    '...-.', '4': '....-', '5': '---', '6': '......', '7': '--..', '8':
    '-....', '9': '-..-', '.': '..--..', ',': '.-.-', '?': '-..-.', '!':
    '---.', '&': '.,...', ';': '...,..', ':': '-.-,.,.', "'": '..-.,.-..', '/':
    '..-,-', '-': '....,.-..', '(': '.....,.-..', ')': '.....,..,..',
}

DECODE_ITU = invert_dict(ENCODE_ITU)
DECODE_MORSE = invert_dict(ENCODE_MORSE)

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

Chapter 38: ROT13 (Rotate 13)

Write a Python program called `rot13.py` that will encrypt/decrypt input text by shifting the text by a given `-s|--shift` argument or will move each character halfway through the alphabet, e.g., "a" becomes "n," "b" becomes "o," etc. The text to rotate should be provided as a single positional argument to your program and can either be a text file, text on the command line, or `-` to indicate STDIN so that you can round-trip data through your program to ensure you are encrypting and decrypting properly.

The way I approached the solution is to think of adding time. If it's 8 in the morning and I want to know the time in 6 hours on a 12-hour (not military/24-hour) clock, I need to think in terms of 12 when the clock rolls over from AM to PM. To do that, I need to know the remainder of dividing by 12, which is given by the modulus `%` operator:

```
>>> now = 8
>>> (now + 6) % 12
2
```

And 6 hours from 8AM is, indeed, 2PM.

Similarly if I want to know how many hours (in decimal) are a particular number of minutes, I need to mod by 60:

```
>>> minutes = 90
>>> int(minutes / 60) + (minutes % 60) / 60
1.5
>>> minutes = 204
>>> int(minutes / 60) + (minutes % 60) / 60
3.4
```

If you `import string`, you can see all the lower/uppercase letters

```
>>> import string
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
>>> string.ascii_uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

So I think about "rot13" like adding 13 (or some other shift interval) to the position of the letter in the list and modding by the length of the list to wrap it around. If the shift is 13 and we are at "a" and want to know what the letter 13 way is, we can use `pos` to find "a" and add 13 to that:

```
>>> lcase = list(string.ascii_lowercase)
>>> lcase.index('a')
0
>>> lcase[lcase.index('a') + 13]
'n'
```

But if we want to know the value for something after the 13th letter in our list, we are in trouble!

```
>>> lcase[lcase.index('x') + 13]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

`%` to the rescue!

```
>>> lcase[(lcase.index('x') + 13) % len(lcase)]
'k'
```

It's not necessary in this algorithm to shift by any particular number. 13 is special because it's halfway through the alphabet, but we could shift by just 2 or 5 characters. If we want to round-trip our text, it's necessary to shift in the opposite direction on the second half of the trip, so be sure to use the negative value there!

```
$ ./rot13.py
usage: rot13.py [-h] [-s int] str
rot13.py: error: the following arguments are required: str
$ ./rot13.py -h
usage: rot13.py [-h] [-s int] str

Argparse Python script

positional arguments:
  str                  Input text, file, or "-" for STDIN

optional arguments:
  -h, --help           show this help message and exit
  -s int, --shift int  Shift arg (default: 0)
$ ./rot13.py AbCd
NoPq
$ ./rot13.py AbCd -s 2
CdEf
$ ./rot13.py fox.txt
Gur dhvpx oebja sbk whzcf bire gur ynml qbt.
$ ./rot13.py fox.txt | ./rot13.py -
The quick brown fox jumps over the lazy dog.
$ ./rot13.py -s 3 fox.txt | ./rot13.py -s -3 -
The quick brown fox jumps over the lazy dog.
```

Solution

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

import argparse
import os
import re
import string
import sys


# --------------------------------------------------
def get_args():
    """get command-line arguments"""
    parser = argparse.ArgumentParser(
        description='ROT13 encryption',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('text',
                        metavar='str',
                        help='Input text, file, or "-" for STDIN')

    parser.add_argument('-s',
                        '--shift',
                        help='Shift arg',
                        metavar='int',
                        type=int,
                        default=0)

    return parser.parse_args()


# --------------------------------------------------
def main():
    """Make a jazz noise here"""
    args = get_args()
    text = args.text

    if text == '-':
        text = sys.stdin.read()
    elif os.path.isfile(text):
        text = open(text).read()

    lcase = list(string.ascii_lowercase)
    ucase = list(string.ascii_uppercase)
```

```
    num_lcase = len(lcase)
    num_ucase = len(ucase)
    lcase_shift = args.shift or int(num_lcase / 2)
    ucase_shift = args.shift or int(num_ucase / 2)

    def rot13(char):
        if char in lcase:
            pos = lcase.index(char)
            rot = (pos + lcase_shift) % num_lcase
            return lcase[rot]
        elif char in ucase:
            pos = ucase.index(char)
            rot = (pos + ucase_shift) % num_ucase
            return ucase[rot]
        else:
            return char

    print(''.join(map(rot13, text)).rstrip())


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

Chapter 39: Word Search

Write a Python program called `search.py` that takes a file name as the single positional argument and finds the words hidden in the puzzle grid.

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

Word search

positional arguments:
  FILE        The puzzle

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

If given a non-existent file, it should complain and exit with a non-zero status:

```
$ ./search.py lkdfak
usage: search.py [-h] FILE
search.py: error: argument FILE: can't open 'lkdfak': [Errno 2] No such file or directory:
```

The format of the puzzle file will be a grid of letters followed by an empty line followed by a list of words to find delimited by newlines, e.g.:

```
$ cat puzzle06.txt
ABC
DEF
GHI

DH
```

If the input grid is uneven, the program should error out:

```
$ cat bad_grid.txt
ABC
DEFG
HIJ

XYZ
$ ./search.py bad_grid.txt
Uneven number of columns
```

The output should be the input puzzle with only the letters showing for the words that are found replacing all the other letters with `.` (a period):

```
$ ./search.py puzzle06.txt
```

```
...
D..
.H.
$ cat ice_cream.txt
YMTRLCHOCOLATE
ASKCARTESOOMET
PYVANILLASNOTE
MKDETDEACFANAA
CATNLINNAOCOOE
OKPOAAGODKEAET
ECULNCAEFOPLRN
DOTAEENORYWEEE
OCBOAWYOTTEOIE
COIEAAARTSAOAR
RNTTCRALETNIAG
EEGDUFOSNIOVLT
DAORYKCORUACGT
AEETUNOCOCTPES

COTTON CANDY
MAPLE WALNUT
PECAN
BANANA
TIGER TAIL
MOOSE TRACKS
COCONUT
ROCKY ROAD
GREEN TEA
FUDGE
REESES
CHOCOLATE
VANILLA
$ ./search.py ice_cream.txt
.....CHOCOLATE
.SKCARTESOOM..
.YVANILLA.N...
M.D.T..A..A..A
.A.N.IN...C..E
..P.AAG...E..T
...LNC.E..P.RN
...AE.N.R..E.E
..B..W.O.TE..E
......A.TSA..R
.......LET.I.G
.EGDUF.SN.O.L.
DAORYKCORU.C..
```

...TUNOCOCT...

Solution

```
#!/usr/bin/env python3
"""Word Search"""

import argparse
from dire import die


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

    parser = argparse.ArgumentParser(
        description='Word search',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file',
                        metavar='FILE',
                        type=argparse.FileType('r'),
                        help='The puzzle')

    return parser.parse_args()


# --------------------------------------------------
def read_puzzle(fh):
    """Read the puzzle file"""

    puzzle, words = [], []
    cell = 0
    read = 'puzzle'
    for line in map(str.rstrip, fh):
        if line == '':
            read = 'words'
            continue

        if read == 'puzzle':
            row = []
            for char in list(line):
                cell += 1
                row.append((char, cell))

            puzzle.append(row)
        else:
```

```
            words.append(line.replace(' ', ''))

    return puzzle, words


# --------------------------------------------------
def all_combos(puzzle):
    """Find all combos in puzzle"""

    num_rows = len(puzzle)
    num_cols = len(puzzle[0])

    if not all([len(row) == num_cols for row in puzzle]):
        die('Uneven number of columns')

    combos = []

    # Horizontal
    for row in puzzle:
        combos.append(row)

    # Vertical
    for col_num in range(num_cols):
        col = [puzzle[row_num][col_num] for row_num in range(num_rows)]
        combos.append(col)

    # Diagonals Up
    for row_i in range(0, num_rows):
        diag = []
        col_num = 0
        for row_j in range(row_i, -1, -1):
            diag.append(puzzle[row_j][col_num])
            col_num += 1

        if diag:
            combos.append(diag)

    for col_i in range(1, num_cols):
        diag = []

        col_num = col_i
        for row_num in range(num_rows - 1, -1, -1):
            diag.append(puzzle[row_num][col_num])
            col_num += 1
            if col_num == num_cols:
                break
```

```python

        if diag:
            combos.append(diag)

    # Diagonals Down
    for row_i in range(0, num_rows):
        diag = []
        col_num = 0
        for row_j in range(row_i, num_rows):
            diag.append(puzzle[row_j][col_num])
            col_num += 1
            if col_num == num_cols:
                break

        if diag:
            combos.append(diag)

    for col_i in range(0, num_cols):
        diag = []

        col_num = col_i
        for row_num in range(0, num_rows):
            diag.append(puzzle[row_num][col_num])
            col_num += 1
            if col_num == num_cols:
                break

        if diag:
            combos.append(diag)

    combos.extend([list(reversed(c)) for c in combos])
    return combos


# --------------------------------------------------
def fst(t):
    """Return first element of a tuple"""

    return t[0]


# --------------------------------------------------
def snd(t):
    """Return second element of a tuple"""
    return t[1]

```

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

    args = get_args()
    puzzle, words = read_puzzle(args.file)
    combos = all_combos(puzzle)
    found = set()
    reveal = set()
    for word in words:
        for combo in combos:
            test = ''.join(map(fst, combo))
            if word in test:
                start = test.index(word)
                end = start + len(word)
                for cell in map(snd, combo[start:end]):
                    reveal.add(cell)
                found.add(word)
                break

    for row in puzzle:
        cells = [c[0] if c[1] in reveal else '.' for c in row]
        print(''.join(cells))

    missing = [w for w in words if not w in found]
    if missing:
        print('Failed to find:')
        for i, word in enumerate(missing, 1):
            print('{:3}: {}'.format(i, word))


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

Discussion

The only argument to the program is a single positional `file` which I chose to define with `type=argparse.FileType('r')` on line 17 to save me the trouble of testing for a file though you could test yourself and will pass the test as long as your error message includes `No such file or directory: '{}'` for the given file.

Reading the puzzle input

I chose to define a few additional functions while keeping most of the programs logic in the `main`. The first is `read_puzzle` that reads the file given by the user. As noted in the README, this file has the puzzle grid, an empty line, and then the list of words to search, so I define `read_puzzle` to accept the file (`fh`) as an argument and return two lists that represent the `puzzle` and `words` (line 28).

There list of `words` is really most naturally represented as a `list` of `str` elements, but the `puzzle` is a bit more complicated. After working through a couple of solutions, I decided I would number all the characters in the grid in order to know which ones to reveal at the end and which ones to replace with a period, so I define a `cell` variable initialized to `0` to keep count of the characters.

Here is my mental model of the puzzle:

```
Puzzle                    Model
                  Col 0   Col 1   Col 2
A B C       Row 0 (A, 1)  (B, 2)  (C, 3)
D E F       Row 1 (D, 4)  (E, 5)  (F, 6)
G H I       Row 2 (G, 7)  (H, 8)  (I, 9)
```

Lastly, I need to know if I'm reading the first part of the file with the puzzle or the latter part with the words, so I define a `read` variable initialized to `'puzzle'` on line 30.

I start reading with `for line in` the file, but I want to chop off the trailing whitespace so I `map(str.rstrip, fh)`. Remember not to include parens `()` on `str.rstrip` as we want to *reference* the function not *call* it. The first operation in the loop is to check for an empty string (`''`, because we remove the newlines). If we find that, then we note the switch to reading the `'words'` and use `continue` to skip to the next iteration of the loop.

If I'm reading the puzzle part of the file. then I want to read each character (line 38), increment the `cell` counter, then create a new tuple with the character and it's cell number, appending this to the `row`, a list to hold all the new tuples. The `row` then gets appended to the `puzzle` list that will eventually be a `list` of rows, each of which is a `list` of tuples representing `(char, cell)`.

If we get to line 44, we must be reading the latter part of the file, so the `line` is actually a word that I will `append` to the `words` list. Before doing that, however, I will `replace` any space (`' '`) with the empty string (`''`) so as to remove spaces (cf. the `ice_cream.txt` input). Finally I `return puzzle, words` which is actually returning a tuple created by the comma `,` and which I immediately unpack on line 124.

Finding all the strings

I always try to make a function fit into about 50 lines of code. While my `read_puzzle` fits into 22 lines, the other function, `all_combos` is considerable longer. I couldn't find a way to shorten it, so I at least try to keep the idea fully contained to one function that, once it works, I no longer need to consider. The idea of this function is to find all the strings possible by reading each row, column, and diagonal both forward and backward. To do this, I first figure out how many rows and columns are present by checking the length (`len`) of the `puzzle` itself (the number of rows) and the length of the first row (the number of character in the first row). I double-check on line 56 that `all` of the the rows have the same `len` as the first one, using the `die` function from the `dire` module to print a message to STDERR and then `sys.exit(1)` to indicate a failure.

The `all_combos` will return a `list` of the characters and their cells, so I define `combos` on line 59 as an empty list (`[]`). Reading the rows is easiest on lines 61-62 as we just copy each `row` into `combo`. Reading the columns is done by moving from column `0` to the last column using the `range(num_cols)` (remembering the last number is not included which is important because if there are 10 columns then we need to move from column `0` to column `9`). I can then extract each column position from each row in the puzzle by indexing `puzzle[row_num][col_num]` and appending those to the `combos`.

The diagonals are the trickiest. I chose to go up (lower-left to upper-right) first. I start in the top-left corner, row `0` and column `0`. For each row, I'm going to move diagonally upwards (toward the top of the grid) which is actually counting *down* from the row I'm on, so I actually need to move `row_i` *up* and then `row_j` *down*. (I use `i` for "integer" and then `j` because "j" comes after "i". This is a typical naming convention. If I needed a third counter, I'd move to `k`.) I count `row_j` *down* by using `range(row_i, -1, -1)` (where the first `-1` is so I can count all the way to `0` and the second indicates the step should go down by one), I need to move the `col_num` over by `1`. If I successfully read a diagonal, I append that to the `combos`.

The next block starts at the bottommost row of the and moves across the columns and is very similar to how I read the columns. Then moving into reading the diagonals in a downward (upper-left to bottom-right) fashion, I modified the other two blocks to handle the specifics. Finally at the end of the function (line 120), I want to `extend` the `combos` list by adding a `reversed` version of each combo. It's necessary to coerce `list(reversed(c))` otherwise we'd end up with references to `reversed` *objects*.

Solving the puzzle

Once we've read the puzzle and found all the possible strings both forwards and backwards, we can then look for each of the words in each of the strings. In my `main`, I want to use sets to note all the words that are `found` as well as the cell numbers to `reveal`. Because I'll be reading lists of tuples where the character is in the first position and the cell number in the second, I define two functions `fst` and `snd` (stolen from Haskell) that I can use in `map` expressions. I iterate `for word in words` (line 146) and `for combo in combos` to check all combinations. Recall that the `combo` is a list of tuples:

```
>>> combo = [('X', 1), ('F', 2), ('O', 3), ('O', 4)]
```

so I can build a string from the characters in the `fst` position of the tuples by mapping them to `fst`:

```
>>> list(map(fst, combo))
['X', 'F', 'O', 'O']
```

and joining them on an empty string:

```
>>> test = ''.join(map(fst, combo))
>>> test
'XFOO'
```

Then I check if the `word` is in the `test` string:

```
>>> word='FOO'
>>> word in test
True
```

If it is, then I can find where it starts with the `str.index` function:

```
>>> start = test.index(word)
>>> start
1
```

I know then end is:

```
>>> end = start + len(word)
>>> end
4
```

I can use that information to iterate over the elements in the `combo` to extract the cell numbers which are in the `snd` position of the tuple because ultimately what I need to print is the original puzzle grid with the cells showing the hidden words and all the others masked. I can extract a list slice using `combo[start:end]`, `map` those elements through `snd` to get the `cell` and `add` those to the `reveal` set. I can also note that I `found` the `word`.

At line 157, I start the work of printing the revealed puzzle, iterating over the original rows in the puzzle and over each cell in the row. If the cell number is in the `reveal` set, I chose the character (in the first position of the tuple); otherwise I use a period (`.`). Finally I note any missing words by looking to see if any of the original words were not in the `found` set.

Appendix 1: argparse

The `argparse` module will interpret all the command-line arguments to your program. I suggest you use `argparse` for every command-line program you write so that you always have a standard way to get arguments and present help.

Types of arguments

Command-line arguments come in a variety of flavors:

- Positional: 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.
- Named options: Standard Unix format allows for a "short" name like `-f` (one dash and a single character) or a "long" name like `--file` (two dashes and a string of characters) followed by some value like a file name or a number. This allows for arguments to be provided in any order or not provided in which case the program can use a reasonable default value.
- Flag: A "Boolean" value like "yes"/"no" or `True`/`False` usually indicated by something that looks like a named option but without a value, e.g., `-d` or `--debug` to turn on debugging. Typically the presence of the flag indicates a `True` value for the argument; therefore, it's absence would mean `False`, so `--debug` turns *on* debugging while no `--debug` flag means there should not no debugging.

Datatypes of values

The `argparse` module can save you enormous amounts of time by forcing the user to provide arguments of a particular type. If you run `new.py`, all of the above types of arguments are present along with suggestions for how to get string or integer values:

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

    parser = argparse.ArgumentParser(
        description='Argparse Python script',
        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',
                        '--flag',
                        help='A boolean flag',
                        action='store_true')

    return parser.parse_args()
```

You should change the `description` to a short sentence describing your program. The `formatter_class` argument tells `argparse` to show the default values in the the standard help documentation.

The `positional` argument's definition indicates we expect exactly one positional argument. The `-a` argument's `type` must be a `str` while the `-i` option must be something that Python can convert to the `int` type (you can also use `float`). Both of these arguments have `default` values which means the user is not required to provide them. You could instead define them with `required=True` to force the user to provide values themselves.

The `-f` flag notes that the `action` is to `store_true` which means the value's default with be `True` if the argument is present and `False` otherwise.

The `type` of the argument can be something much richer than simple Python types like strings or numbers. You can indicate that an argument must be a existing, readable file. Here is a simple implementation in Python of `cat -n`:

```
#!/usr/bin/env python3
"""Python version of `cat -n`"""

import argparse


# --------------------------------------------------
def get_args():
```

```
    """Get command-line arguments"""

    parser = argparse.ArgumentParser(
        description='Argparse Python script',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file',
                        metavar='FILE',
                        type=argparse.FileType('r'),
                        help='Input file')

    return parser.parse_args()


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

    args = get_args()
    fh = args.file

    print('Reading "{}"'.format(fh.name))
    for i, line in enumerate(fh):
        print(i, line, end='')


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

The `type` of the input `file` argument is an *open file handle* which we can directly read line-by-line with a `for` loop! Because it's a file *handle* and not a file *name*, I chose to call the variable `fh` to help me remember what it is. You can access the file's name via `fh.name`.

```
$ ./cat_n.py ../../inputs/the-bustle.txt
Reading "../../inputs/the-bustle.txt"
0 The bustle in a house
1 The morning after death
2 Is solemnest of industries
3 Enacted upon earth,--
4
5 The sweeping up the heart,
6 And putting love away
7 We shall not want to use again
8 Until eternity.
```

Number of arguments

If you want one positional argument, you can define them like so:

```
#!/usr/bin/env python3
"""One positional argument"""

import argparse

parser = argparse.ArgumentParser(
    description='One positional argument',
    formatter_class=argparse.ArgumentDefaultsHelpFormatter)

parser.add_argument('first', metavar='str', help='First argument')
args = parser.parse_args()
print('first =', args.first)
```

If the user provides anything other exactly one argument, they get a help message:

```
$ ./one_arg.py
usage: one_arg.py [-h] str
one_arg.py: error: the following arguments are required: str
$ ./one_arg.py foo bar
usage: one_arg.py [-h] str
one_arg.py: error: unrecognized arguments: bar
$ ./one_arg.py foo
first = foo
```

If you want two different positional arguments:

```
#!/usr/bin/env python3
"""Two positional arguments"""

import argparse

parser = argparse.ArgumentParser(
    description='Two positional arguments',
    formatter_class=argparse.ArgumentDefaultsHelpFormatter)

parser.add_argument('first', metavar='str', help='First argument')

parser.add_argument('second', metavar='int', help='Second argument')

return parser.parse_args()

print('first =', args.first)
print('second =', args.second)
```

Again, the user must provide exactly this number of positional arguments:

```
$ ./two_args.py
usage: two_args.py [-h] str str
two_args.py: error: the following arguments are required: str, str
$ ./two_args.py foo
usage: two_args.py [-h] str str
two_args.py: error: the following arguments are required: str
$ ./two_args.py foo bar
first = foo
second = bar
```

You can also use the `nargs=N` option to specify some number of arguments. It only makes sense if the arguments are the same thing like two files:

```
#!/usr/bin/env python3
"""nargs=2"""

import argparse

parser = argparse.ArgumentParser(
    description='nargs=2',
    formatter_class=argparse.ArgumentDefaultsHelpFormatter)

parser.add_argument('files', metavar='FILE', nargs=2, help='Two files')

args = parser.parse_args()

file1, file2 = args.files
print('file1 =', file1)
print('file2 =', file2)
```

The help indicates we want two files:

```
$ ./nargs2.py foo
usage: nargs2.py [-h] FILE FILE
nargs2.py: error: the following arguments are required: FILE
```

And we can unpack the two file arguments and use them:

```
$ ./nargs2.py foo bar
file1 = foo
file2 = bar
```

If you want one or more of some argument, you can use `nargs='+'`:

```
$ cat nargs+.py
#!/usr/bin/env python3
"""nargs=+"""
```

```
import argparse

parser = argparse.ArgumentParser(
    description='nargs=+',
    formatter_class=argparse.ArgumentDefaultsHelpFormatter)

parser.add_argument('files', metavar='FILE', nargs='+', help='Some files')

args = parser.parse_args()
files = args.files

print('number = {}'.format(len(files)))
print('files  = {}'.format(', '.join(files)))
```

Note that this will return a `list` – even a single argument will become a `list` of one value:

```
$ ./nargs+.py
usage: nargs+.py [-h] FILE [FILE ...]
nargs+.py: error: the following arguments are required: FILE
$ ./nargs+.py foo
number = 1
files  = foo
$ ./nargs+.py foo bar
number = 2
files  = foo, bar
```

Choices

Sometimes you want to limit the values of an argument. You can pass in a `list` of valid values to the `choices` option.

```
$ cat appendix/argparse/choices.py
#!/usr/bin/env python3
"""Choices"""

import argparse

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

parser.add_argument('color', metavar='str', help='Color', choices=['red', 'yellow', 'blue'])

args = parser.parse_args()
```

```
print('color =', args.color)
```

Any value not present in the list will be rejected and the user will be shown the valid choices:

```
$ ./choices.py
usage: choices.py [-h] str
choices.py: error: the following arguments are required: str
$ ./choices.py purple
usage: choices.py [-h] str
choices.py: error: argument str: invalid choice: 'purple' (choose from 'red', 'yellow', 'blu
```

Automatic help

The `argparse` module reserves the `-h` and `--help` flags for generating help documentation. You do not need to add these nor are you allowed to use these flags for other purposes. Using the above definition, this is the help that `argparse` will generate:

```
$ ./foo.py
usage: foo.py [-h] [-a str] [-i int] [-f] str
foo.py: error: the following arguments are required: str
[cholla@~/work/python/playful_python/article]$ ./foo.py -h
usage: foo.py [-h] [-a str] [-i int] [-f] str

Argparse Python script

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, --flag         A boolean flag (default: False)
```

Notice how unhelpful a name like `positional` is?

Getting the argument values

The values for the arguments will be accessible through the "long" name you define and will have been coerced to the Python data type you indicated. If I change `main` to this:

```
# --------------------------------------------------
def main():
```

```
    """Make a jazz noise here"""

    args = get_args()
    str_arg = args.arg
    int_arg = args.int
    flag_arg = args.flag
    pos_arg = args.positional

    print('str_arg = "{}" ({})'.format(str_arg, type(str_arg)))
    print('int_arg = "{}" ({})'.format(int_arg, type(int_arg)))
    print('flag_arg = "{}" ({})'.format(flag_arg, type(flag_arg)))
    print('positional = "{}" ({})'.format(pos_arg, type(pos_arg)))
```

And then run it:

```
$ ./foo.py -a foo -i 4 -f bar
str_arg = "foo" (<class 'str'>)
int_arg = "4" (<class 'int'>)
flag_arg = "True" (<class 'bool'>)
positional = "bar" (<class 'str'>)
```

Notice how we might think that `-f` takes the argument `bar`, but it is defined as a `flag` and the `argparse` knows that the program take

```
$ ./foo.py foo -a bar -i 4 -f
str_arg = "bar" (<class 'str'>)
int_arg = "4" (<class 'int'>)
flag_arg = "True" (<class 'bool'>)
positional = "foo" (<class 'str'>)
```

Appendix 2: Truthiness

While it would seem Python has an actual Boolean (Yes/No, True/False) type, this idea can be seriously abused in many odd and confusing ways. First off, there are actual `True` and `False` values:

```
>>> True == True
True
>>> False == False
True
```

But they are equivalent to integers:

```
>>> True == 1
True
>>> False == 0
True
```

Which means, oddly, that you can add them:

```
>>> True + True
2
>>> True + True + False
2
```

Lots of things are `False`-ey when they are evaluated in a Boolean context. The `int 0`, the `float 0.0`, the empty string, an empty list, and the special value `None` are all considered `False`-ey:

```
>>> 'Hooray!' if 0 else 'Shucks!'
'Shucks!'
>>> 'Hooray!' if 0. else 'Shucks!'
'Shucks!'
>>> 'Hooray!' if [] else 'Shucks!'
'Shucks!'
>>> 'Hooray!' if '' else 'Shucks!'
'Shucks!'
>>> 'Hooray!' if None else 'Shucks!'
'Shucks!'
```

But note:

```
>>> 'Hooray!' if 'None' else 'Shucks!'
'Hooray!'
```

There are quotes around `'None'` so it's the literal string "None" and not the special value `None`, and, since this is not an empty string, it evaluates *in a Boolean context* to not-`False` which is basically `True`.

This behavior can introduce extremely subtle logical bugs into your programs that the Python compiler and linters cannot uncover. Consider the `dict.get` method that will safely return the value for a given key in a dictionary, returning `None` if the key does not exist. Given this dictionary:

```
>>> d = {'foo': 0, 'bar': None}
```

If we access a key that doesn't exist, Python generates an exception that, if not caught in our code, would immediately crash the program:

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

But we can use `d.get()` to do this safely:

```
>>> d.get('baz')
```

Hmm, that seems unhelpful! What did we get back?

```
>>> type(d.get('baz'))
<class 'NoneType'>
```

Ah, we got `None`!

We could use an `or` to define a default value:

```
>>> d.get('baz') or 'NA'
'NA'
```

It turns out the `get` method accepts a second, optional argument of the default value to return:

```
>>> d.get('baz', 'NA')
'NA'
```

Great! So let's use that on the other values:

```
>>> d.get('foo', 'NA')
0
>>> d.get('bar', 'NA')
```

The call for `bar` returned nothing because we put an actual `None` as the value:

```
>>> type(d.get('bar', 'NA'))
<class 'NoneType'>
```

The key `bar` didn't fail because that key exists in the dictionary. The `dict.get` method only returns the second, default argument *if the key does not exist in the dictionary* which is entirely different from checking the *value* of the key in the dictionary. OK, so we go back to this:

```
>>> d.get('bar') or 'NA'
'NA'
```

Which seems to work, but notice this:

```
>>> d.get('foo') or 'NA'
'NA'
```

The value for `foo` is actually `0` which evaluates to `False` given the Boolean evaluation of the `or`. If this were a measurement of some value like the amount of sodium in water, then the string `NA` would indicate that no value was recorded whereas `0` indicates that sodium was measured and none detected. If some sort of important analysis rested on our interpretation of the strings in a spreadsheet, we might inadvertently introduce missing values because of the way Python coerces various non-Boolean values into Boolean values.

Perhaps a safer way to access these values would be:

```
>>> for key in ['foo', 'bar', 'baz']:
...     val = d[key] if key in d else 'NA'
...     val = 'NA' if val is None else val
...     print(key, val)
...
foo 0
bar NA
baz NA
```

Appendix 3: File Handles

A file's name is a string like `'nobody.txt'`. To read or write the contents of the file, you need a *file handle* which you can get from `open`. Think of a file name as the address of your house. It's where your house can be found, but I can't know what's in your house unless I go there and open the door. That's what `open` does – it finds the file's bits on disk and opens the door to read or write the file.

File Modes

By default, a file is opened in *read* mode which means that it can't be altered. Also, the default is to open for reading *text*. The only required argument to `open` is the file name, but a second optional argument is a combination of characters to explain how to open the file. From the documentation for `open`:

```
========= ===============================================================
Character Meaning
--------- ---------------------------------------------------------------
'r'       open for reading (default)
'w'       open for writing, truncating the file first
'x'       create a new file and open it for writing
'a'       open for writing, appending to the end of the file if it exists
'b'       binary mode
't'       text mode (default)
'+'       open a disk file for updating (reading and writing)
'U'       universal newline mode (deprecated)
========= ===============================================================
```

So if you do:

```
fh = open('out.txt')
```

It's the same as doing:

```
fh = open('out.txt', 'wt')
```

Where the combination of `wt` means `write text`. We can also read and write raw bits in `binary`, e.g., if you wanted to read the bit values of the pixels in an image.

I always make a distinction in the variable names for the `file` or `filename` and the *file handle* which I usually call `fh` if there's just one or maybe `in_fh` and `out_fh` if there is one for reading and one for writing, etc.

STDIN, STDOUT, STDERR

Unix has three standard files or channels called *standard in*, *standard out*, and *standard error* which are normally written as STDIN, STDOUT, and STDERR. When you `print`, the default is that the text goes to STDOUT which you see in your terminal or REPL.

The `print` function takes some optional keyword arguments, one of which is `file` which has the default value of `sys.stdout`. If you wish to `print` to *standard error* (STDERR), you can use the `sys.stderr` file:

```
print('This is an error!', file=sys.stderr)
```

Note that you *do not* have to `open` these two special file handles. They are always available to you.

If you wish to write to a file on disc, you can `open` a file for writing and pass that:

```
print('This is an error!', file=open('error.txt', 'wt'))
```

Note that if each time you `open` a file for writing, you overwrite any existing data. If you wanted to `print` repeatedly in a program, you would either need to `open` in append mode:

```
print('This is an error!', file=open('error.txt', 'at'))
print('This is an also error!', file=open('error.txt', 'at'))
```

Or, better yet, `open` the file at the beginning of the program, `print` as often as you like, and then `close` the file:

```
fh = open('out.txt', 'wt')
print('Writing some text.', file=fh)
print('Adding more text.', file=fh)
fh.close()
```

Or use the `write` method of the file handle:

```
fh = open('out.txt', 'wt')
fh.write('Writing some text.\n')
fh.write('Adding more text.\n')
fh.close()
```

Note that `print` automatically adds a newline to the end of the text whereas `write` does not so you need to add it yourself.

You can only *read* from STDIN. Again, you do not need to `open` it as it is always available. Treat it exactly like a file handle you've opened for reading, e.g., to read lines from STDIN until you recieve EOF (end of file):

```
for line in sys.stdin:
```

Appendix 4: N-grams, K-mers, and Markov Chains

Read about Markov chains:

- Claude Shannon's 1948 MS thesis, "A Mathematical Theory of Communication" (https://onlinelibrary.wiley.com/doi/abs/10.1002/j.1538-7305.1948.tb01338.x)
- https://en.wikipedia.org/wiki/Markov_chain
- Chapter 3 of *The Practice of Programming* by Brian Kernighan and Rob Pike where they discuss implementations in C, C++, Java, awk, and Perl
- "Computer Recreations", A. K. Dewdney, Scientific American, 1989 (https://archive.org/details/ComputerRecreationsMarkovChainer)

I'd like you to consider how a Markov chain creates a graph structure. Consult the three PDFs (generated by the `mk-graphs.sh` program) that visualize the graphs created by k-mer sizes of 1, 2, 3, and 4 when given this input:

```
$ cat words.txt
maamselle
mabi
mabolo
mac
macaasim
macabre
```

Notice that sometimes the branches terminate and sometimes you can find multiple paths through the graphs. As `k` grows, there are fewer options.