Contents

Playful Python

> "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 with a test suite so that you know if your program is working well enough.

I won't necessarily show you beforehand what you need to write a 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.

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 a program in the `bin` directory 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 go into the directory with `cd article` and then do:

```
$ 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. It's best to put `new.py` into your `$PATH` or alter your `$PATH` to include the directory where it's located. I usually create a `$HOME/.local/bin` that I add to my `$PATH` for programs like this.

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.

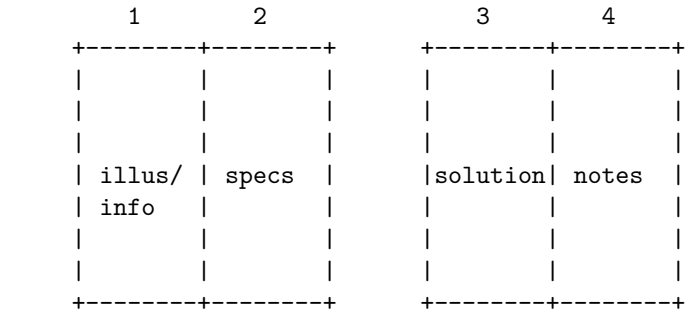
Author

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.

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. 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.

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

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

Cf Appendices: argparse, Truthiness

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* 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'
```

Chapter 2: Jump the Five

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
$ ./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()
```

Chapter 4: 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
```

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. It should complain if the `--vowel` argument isn't a single, lowercase vowel (hint, see `choices` in the `argparse` documentation). If the given text argument is a file, read the text from the file. Replace all vowels with the given vowel, both lower- and uppercase.

```
$ ./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)
$ ./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',
$ ./apples.py foo
faa
$ ./apples.py ../inputs/fox.txt
Tha qaack brawn fax jamps avar tha lazy dag.
```

Solution

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

import argparse
import os
import re
import sys


# --------------------------------------------------
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)
```

```
44      #     elif char in 'AEIOU':
45      #         new_text.append(vowel.upper())
46      #     else:
47      #         new_text.append(char)
48      # text = ''.join(new_text)
49
50      # Method 2: str.replace
51      # for v in 'aeiou':
52      #     text = text.replace(v, vowel).replace(v.upper(), vowel.upper())
53
54      # Method 3: Use a list comprehension
55      # new_text = [
56      #     vowel if c in 'aeiou' else vowel.upper() if c in 'AEIOU' else c
57      #     for c in text
58      # ]
59      # text = ''.join(new_text)
60
61      # Method 4: Define a function, use list comprehension
62      def new_char(c):
63          return vowel if c in 'aeiou' else vowel.upper() if c in 'AEIOU' else c
64
65      # text = ''.join([new_char(c) for c in text])
66
67      # Method 5: Use a `map` to iterate with a `lambda`
68      # text = ''.join(
69      #     map(
70      #         lambda c: vowel if c in 'aeiou' else vowel.upper()
71      #         if c in 'AEIOU' else c, text))
72
73      # Method 6: `map` with the function
74      text = ''.join(map(new_char, text))
75
76      # Method 7: Regular expressions
77      # text = re.sub('[aeiou]', vowel, text)
78      # text = re.sub('[AEIOU]', vowel.upper(), text)
79
80      print(text.rstrip())
81
82
83  # ------------------------------------------------
84  if __name__ == '__main__':
85      main()
```

Chapter 5: Howler

Write a Python program `howler.py` that will uppercase all the text from the command line or from a file.

```
$ ./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.
```

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().strip()

    out_fh = open(out_file, 'wt') if out_file else sys.stdout
    out_fh.write(text.upper() + '\n')


# --------------------------------------------------
if __name__ == '__main__':
```

```
44      main()
```

Chapter 6: Bottles of Beer Song

Write a Python program called `bottles.py` that takes a single option `-n|--num_bottles` which is an positive integer (default 10) and prints the " bottles of beer on the wall song." If the `-n` argument is less than 1, die with "N () must be a positive integer". The program should also respond to `-h|--help` with a usage statement.

I'd encourage you to think about the program as a formal algorithm. Read the introduction to Jeff Erickson's book *Algorithms* available here:

- http://jeffe.cs.illinois.edu/teaching/algorithms/#book
- http://jeffe.cs.illinois.edu/teaching/algorithms/book/00-intro.pdf

You are going to need to count down, so you'll need to consider how to do that. First, let's examine a list and see how it can be sorted and reversed. We've already used the `sorted` *function*, but we haven't really talked about the `list` class's `sort` *method*. Note that the former does not mutate the list itself:

```
>>> a = ['foo', 'bar', 'baz']
>>> sorted(a)
['bar', 'baz', 'foo']
>>> a
['foo', 'bar', 'baz']
```

But the `sort` method does:

```
>>> a.sort()
>>> a
['bar', 'baz', 'foo']
```

Also, note what is returned by `sort`:

```
>>> type(a.sort())
<type 'NoneType'>
```

So if you did this, you'd destroy your data:

```
>>> a = a.sort()
>>> a
```

As with `sort`/`sorted`, so it goes with `reverse`/`reversed`. The past participle version *returns a new copy of the data without affecting the original* and is therefore the safest bet to use:

```
>>> a = ['foo', 'bar', 'baz']
>>> a
['foo', 'bar', 'baz']
>>> reversed(a)
<listreverseiterator object at 0x10f0d61d0>
>>> list(reversed(a))
```

```
['baz', 'bar', 'foo']
>>> a
['foo', 'bar', 'baz']
```

Compare with:

```
>>> a.reverse()
>>> a
['baz', 'bar', 'foo']
```

Given that and your knowledge of how `range` works, can you figure out how to count down, say, from 10 to 1?

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

Bottles of beer song

optional arguments:
  -h, --help            show this help message and exit
  -n INT, --num_bottles INT
                        How many bottles (default: 10)
$ ./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!
```

Solution

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

import argparse
import sys
from dire import die


# --------------------------------------------------
def get_args():
    """get command-line arguments"""
    parser = argparse.ArgumentParser(
        description='Bottles of beer song',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('-n',
                        '--num_bottles',
                        metavar='INT',
                        type=int,
                        default=10,
                        help='How many bottles')

    return parser.parse_args()


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

    if num_bottles < 1:
        die('N ({}) must be a positive integer'.format(num_bottles))

    line1 = '{} bottle{} of beer on the wall'
    line2 = '{} bottle{} of beer'
    line3 = 'Take one down, pass it around'
    tmpl = ',\n'.join([line1, line2, line3, line1 + '!'])

    for n in reversed(range(1, num_bottles + 1)):
        s1 = '' if n == 1 else 's'
        s2 = '' if n - 1 == 1 else 's'
        print(tmpl.format(n, s1, n, s1, n - 1, s2))
        if n > 1: print()
```

```
44
45
46  # ------------------------------------------------
47  if __name__ == '__main__':
48      main()
```

Chapter 7: 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.

```
$ ./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)
$ ./gashlycrumb.py 3
I do not know "3".
$ ./gashlycrumb.py CH
"CH" is not 1 character.
$ ./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 you are not familiar with the work of Edward Gorey, please stop and go read about him immediately, e.g. https://www.brainpickings.org/2011/01/19/edward-gorey-the-gashlycrumb-tinies/!

Write your own version of Gorey's text and pass in your version as the `--file`.

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

```
$ ./gashlycrumb_i.py
Please provide a letter [! to quit]: a
A is for Amy who fell down the stairs.
Please provide a letter [! to quit]: b
B is for Basil assaulted by bears.
Please provide a letter [! to quit]: !
Bye
```

Solution

```
#!/usr/bin/env python3
"""Lookup tables"""

import argparse
import os
from dire import die


# --------------------------------------------------
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=str,
                        default='gashlycrumb.txt')

    return parser.parse_args()


# --------------------------------------------------
def main():
    """Make a jazz noise here"""
    args = get_args()
    letter = args.letter.upper()
    file = args.file

    if not os.path.isfile(file):
        die('--file "{}" is not a file.'.format(file))

    if len(letter) != 1:
        die('"{}" is not 1 character.'.format(letter))

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

```
44
45      if letter in lookup:
46          print(lookup[letter])
47      else:
48          print('I do not know "{}".'.format(letter))
49
50
51  # -----------------------------------------------
52  if __name__ == '__main__':
53      main()
```

Chapter 8: Movie Reader

Write a Python program called `movie_reader.py` that takes a single positional argument that is a bit of text or the name of an input file. The output will be dynamic, so I cannot write a test for how the program should behave, nor can I include a bit of text that shows you how it should work. Your program should print the input text character-by-character and then pause .5 seconds for ending punctuation like `.`, `!` or `?`, .2 seconds for a pause like `,` `:`, or `;`, and .05 seconds for anything else.

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

Movie Reader

positional arguments:
  str         Input text or file

optional arguments:
  -h, --help  show this help message and exit
$ ./movie_reader.py 'Foo, bar!'
Foo, bar!
$ ./movie_reader.py ../inputs/fox.txt
The quick brown fox jumps over the lazy dog.
```

Solution

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

import argparse
import os
import sys
import time


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

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

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

    return parser.parse_args()


# --------------------------------------------------
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():
        for char in line:
            print(char, end='')
            time.sleep(.5 if char in '.!?\n' else .2 if char in ',:;' else .05)
            sys.stdout.flush()

        print()


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

Chapter 9: Palindromes

Write a Python program called `palindromic.py` that will find words that are palindromes in positional argument which is either a string or a file name.

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

Find palindromes in text

positional arguments:
  str                 Input text or file

optional arguments:
  -h, --help          show this help message and exit
  -m int, --min int   Minimum word length (default: 3)
$ ./palindromic.py '"Wow!" said Mom.'
wow
mom
$ ./palindromic.py input.txt
anna
civic
kayak
madam
mom
wow
level
noon
racecar
radar
redder
refer
rotator
rotor
solos
stats
tenet
```

Solution

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

import argparse
import os
import re


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

    parser = argparse.ArgumentParser(
        description='Find palindromes in text',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

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

    parser.add_argument('-m',
                        '--min',
                        metavar='int',
                        type=int,
                        help='Minimum word length',
                        default=3)

    return parser.parse_args()


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

    args = get_args()
    text = args.text
    min_length = args.min

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

    for line in text.splitlines():
        for word in re.split(r'(\W+)', line.lower()):
            if len(word) >= min_length:
                rev = ''.join(reversed(word))
                if rev == word:
```

```
44                      print(word)
45
46
47  # ------------------------------------------------
48  if __name__ == '__main__':
49      main()
```

Chapter 10: Ransom

Create a Python program called `ransom.py` that will randomly capitalize the letters in a given word or phrase. The input text may also name a file in which case the text should come from the file. The program should take a `-s|--seed` argument for the `random.seed` to control randomness for the test suite. It should also respond to `-h|--help` for usage.

```
$ ./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)
$ cat fox.txt
The quick brown fox jumps over the lazy dog.
$ ./ransom.py fox.txt
the quiCK bROWn fOx JUMps OveR tHe LAzy Dog.
$ ./ransom.py -s 2 'The quick brown fox jumps over the lazy dog.'
the qUIck BROWN fOX JUmps ovEr ThE LAZY DOg.
```

Solution

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

import argparse
import os
import random
import sys


# --------------------------------------------------
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)

    return parser.parse_args()


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

    random.seed(args.seed)

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

    #ransom = []
    #for char in text:
    #    ransom.append(char.upper() if random.choice([0, 1]) else char.lower())

    #ransom = [c.upper() if random.choice([0, 1]) else c.lower() for c in text]
```

```
44
45      #ransom = map(lambda c: c.upper() if random.choice([0, 1]) else c.lower(),
46      #             text)
47
48      f = lambda c: c.upper() if random.choice([0, 1]) else c.lower()
49      ransom = map(f, text)
50
51      print(''.join(ransom))
52
53
54  # ------------------------------------------------
55  if __name__ == '__main__':
56      main()
```

Chapter 11: Simple Rhymer

Write a Python program called `rhymer.py` that will create new words by removing the consonant(s) from the beginning of the word and then creating 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
```

```
$ ./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
$ ./rhymer.py apple
Word "apple" must start with consonants
$ ./rhymer.py take | head
bake
cake
dake
fake
gake
hake
jake
kake
lake
make
```

Solution

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

import argparse
import re
import string
import sys
from dire import die


# --------------------------------------------------
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')

    return parser.parse_args()


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

    vowels = 'aeiou'
    if word[0] in vowels:
        die('Word "{}" must start with consonants'.format(word))

    consonants = [c for c in string.ascii_lowercase if c not in 'aeiou']
    match = re.match('^([' + ''.join(consonants) + ']+)(.+)', word)

    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').split()

    if match:
        start, rest = match.group(1), match.group(2)
        for c in filter(lambda c: c != start, consonants + clusters):
            print(c + rest)
```

```
44
45
46  # ------------------------------------------------
47  if __name__ == '__main__':
48      main()
```

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()))


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

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

Chapter 13: Abuse

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 (defualt `None`) to pass to `random.seed`.

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

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

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

Solution

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

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='Argparse Python script',
        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)

```

```
44      parser.add_argument('-s',
45                          '--seed',
46                          help='Random seed',
47                          metavar='int',
48                          type=int,
49                          default=None)
50
51      return parser.parse_args()
52
53
54  # --------------------------------------------------
55  def main():
56      """Make a jazz noise here"""
57      args = get_args()
58      num_adj = args.adjectives
59      num_insults = args.number
60
61      random.seed(args.seed)
62
63      for _ in range(num_insults):
64          adjs = random.sample(adjectives, k=num_adj)
65          noun = random.choice(nouns)
66          print('You {} {}!'.format(', '.join(adjs), noun))
67
68
69  # --------------------------------------------------
70  if __name__ == '__main__':
71      main()
```

Chapter 14: Bacronym

Write a Python program called `bacronym.py` that takes a string like "FBI" and retrofits some `-n|--number` (default `5`) of acronyms by reading a `-w|--wordlist` argument (defualt `/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.

```
$ ./bacronym.py
usage: bacronym.py [-h] [-n NUM] [-w STR] [-x 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] [-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, --exclude STR
                        List of words to exclude (default: a,an,the)
  -s INT, --seed INT    Random seed (default: None)
$ ./bacronym.py FBI -s 1
FBI =
 - Fecundity Brokage Imitant
 - Figureless Basketmaking Ismailite
 - Frumpery Bonedog Irregardless
 - Foxily Blastomyces Inedited
 - Fastland Bouncingly Idiospasm
```

Solution

```
#!/usr/bin/env python3
"""Make guesses about acronyms"""

import argparse
import sys
import os
import random
import re
from collections import defaultdict


# --------------------------------------------------
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=str,
                        metavar='STR',
                        default='/usr/share/dict/words')

    parser.add_argument('-x',
                        '--exclude',
                        help='List of words to exclude',
                        type=str,
                        metavar='STR',
                        default='a,an,the')

    parser.add_argument('-s',
                        '--seed',
```

```
44                          help='Random seed',
45                          type=int,
46                          metavar='INT',
47                          default=None)
48
49      return parser.parse_args()
50
51
52  # --------------------------------------------------
53  def main():
54      """main"""
55
56      args = get_args()
57      acronym = args.acronym
58      wordlist = args.wordlist
59      limit = args.num
60      goodword = r'^[a-z]{2,}$'
61      badwords = set(re.split(r'\s*,\s*', args.exclude.lower()))
62
63      random.seed(args.seed)
64
65      if not re.match(goodword, acronym.lower()):
66          print('"{}" must be >1 in length, only use letters'.format(acronym))
67          sys.exit(1)
68
69      if not os.path.isfile(wordlist):
70          print('"{}" is not a file.'.format(wordlist))
71          sys.exit(1)
72
73      seen = set()
74      words_by_letter = defaultdict(list)
75      for word in open(wordlist).read().lower().split():
76          clean = re.sub('[^a-z]', '', word)
77          if not clean:  # nothing left?
78              continue
79
80          if re.match(goodword,
81                      clean) and clean not in seen and clean not in badwords:
82              seen.add(clean)
83              words_by_letter[clean[0]].append(clean)
84
85      len_acronym = len(acronym)
86      definitions = []
87      for i in range(0, limit):
88          definition = []
89          for letter in acronym.lower():
```

```
90              possible = words_by_letter.get(letter, [])
91              if len(possible) > 0:
92                  definition.append(
93                      random.choice(possible).title() if possible else '?')
94
95          if len(definition) == len_acronym:
96              definitions.append(' '.join(definition))
97
98      if len(definitions) > 0:
99          print(acronym.upper() + ' =')
100         for definition in definitions:
101             print(' - ' + definition)
102     else:
103         print('Sorry I could not find any good definitions')
104
105
106 # ------------------------------------------------
107 if __name__ == '__main__':
108     main()
```

Chapter 15: 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',
```

```
44                          action='store_true')
45
46      return parser.parse_args()
47
48
49  # --------------------------------------------------
50  def read_csv(fh):
51      """Read the CSV input"""
52
53      exercises = []
54
55      for row in csv.DictReader(fh, delimiter=','):
56          name = row['exercise']
57          low, high = row['reps'].split('-')
58          exercises.append((name, int(low), int(high)))
59
60      return exercises
61
62
63  # --------------------------------------------------
64  def main():
65      """Make a jazz noise here"""
66
67      args = get_args()
68      random.seed(args.seed)
69      exercises = read_csv(args.file)
70      table = []
71
72      for name, low, high in random.sample(exercises, k=args.num_exercises):
73          if args.easy:
74              low = int(low / 2)
75              high = int(high / 2)
76
77          table.append((name, random.randint(low, high)))
78
79      print(tabulate(table, headers=('Exercise', 'Reps')))
80
81
82  # --------------------------------------------------
83  if __name__ == '__main__':
84      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 16: Blackjack

Write a Python program called `blackjack.py` that plays an abbreviated game of Blackjack. You will need to `import random` to get random cards from a deck you will construct, and so your program will need to accept a `-s|--seed` that will set `random.seed()` with the value that is passed in so that the test suite will work. The other arguments you will accept are two flags (Boolean values) of `-p|--player_hits` and `-d|--dealer_hits`. As usual, you will also have a `-h|--help` option for usage statement.

To play the game, the user will run the program and will see a display of what cards the dealer has (noted "D") and what cards the player has (noted "P") 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).

To create your deck of cards, you will need to use the Unicode symbols for the suites ( ) [which won't display in the PDF, so consult the Markdown file].

Combine these with the numbers 2-10 and the letters "A", "J", "Q," and "K" (hint: look at `itertools.product`). Because your game will use randomness, you will need to sort your deck and then use the `random.shuffle` method so that your cards will be in the correct order to pass the tests.

When you make the initial 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 `-p|--player_htis` flag is present, deal an additional card to the player; likewise with the `-d|--dealer_hits` flag.

After displaying the hands, the code should:

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."

NB: Look at the Markdown format to see the actual output as the suites won't display in the PDF version!

```
$ ./blackjack.py
D [11]:  J  A
```

```
P [18]:  8  10
Dealer should hit.
$ ./blackjack.py
D [13]:  3  J
P [16]:  6  10
Dealer should hit.
Player should hit.
$ ./blackjack.py -s 5
D [ 5]:  4  A
P [19]:  10  9
Dealer should hit.
$ ./blackjack.py -s 3 -p
D [19]:  K  9
P [22]:  3  9  J
Player busts! You lose, loser!
$ ./blackjack.py -s 15 -p
D [19]:  10  9
P [21]:  10  8  3
Player wins. You probably cheated.
```

Solution

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

import argparse
import random
import sys
from itertools import product
from dire import die


# --------------------------------------------------
def get_args():
    """get command-line arguments"""
    parser = argparse.ArgumentParser(
        description='Argparse Python script',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

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

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

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

    return parser.parse_args()


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

```

```
44
45  # --------------------------------------------------
46  def card_value(card):
47      """card to numeric value"""
48      val = card[1:]
49      faces = {'A': 1, 'J': 10, 'Q': 10, 'K': 10}
50      if val.isdigit():
51          return int(val)
52      elif val in faces:
53          return faces[val]
54      else:
55          die('Unknown card value for "{}"'.format(card))
56
57
58  # --------------------------------------------------
59  def main():
60      """Make a jazz noise here"""
61      args = get_args()
62      random.seed(args.seed)
63      suites = list('♥♦♣♠')
64      values = list(range(2, 11)) + list('AJQK')
65      cards = sorted(map(lambda t: '{}{}'.format(*t), product(suites, values)))
66      random.shuffle(cards)
67
68      p1, d1, p2, d2 = cards.pop(), cards.pop(), cards.pop(), cards.pop()
69      player = [p1, p2]
70      dealer = [d1, d2]
71
72      if args.player_hits:
73          player.append(cards.pop())
74      if args.dealer_hits:
75          dealer.append(cards.pop())
76
77      player_hand = sum(map(card_value, player))
78      dealer_hand = sum(map(card_value, dealer))
79
80      print('D [{:2}]: {}'.format(dealer_hand, ' '.join(dealer)))
81      print('P [{:2}]: {}'.format(player_hand, ' '.join(player)))
82
83      if player_hand > 21:
84          bail('Player busts! You lose, loser!')
85      elif dealer_hand > 21:
86          bail('Dealer busts.')
87      elif player_hand == 21:
88          bail('Player wins. You probably cheated.')
89      elif dealer_hand == 21:
```

```
90          bail('Dealer wins!')
91
92      if dealer_hand < 18: print('Dealer should hit.')
93      if player_hand < 18: print('Player should hit.')
94
95
96  # -----------------------------------------------
97  if __name__ == '__main__':
98      main()
```

Chapter 17: Family Tree

Write a program called `tree.py` that will take an input file as a single positional argument and produce a graph of the family tree described therein. The file can have only three kinds of statements:

1. `INITIALS = Full Name`
2. `person1 married person2`
3. `person1 and person2 begat child1[, child2...]`

Use the `graphviz` module to generate a graph like the `kyc.gv.pdf` included here that was generated 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
$ ./tree.py tudor.txt
Done, see output in "tudor.txt.gv".
```

Solution

```
#!/usr/bin/env python3
"""
Author : kyclark
Date   : 2019-05-24
Purpose: Display a family tree
"""

import argparse
import os
import re
import sys
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='')

    return parser.parse_args()


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

    args = get_args()
    fh = args.file
```

```
44      out_file = args.outfile or os.path.basename(fh.name) + '.gv'
45
46      nodes, edges = parse_tree(fh)
47      dot = Digraph(comment='Tree')
48      for initials, name in nodes.items():
49          dot.node(name)
50
51      for n1, n2 in edges:
52          if n1 in nodes:
53              n1 = nodes[n1]
54          if n2 in nodes:
55              n2 = nodes[n2]
56
57          dot.edge(n1, n2)
58
59      dot.render(out_file, view=True)
60
61      print('Done, see output in "{}".'.format(out_file))
62
63  # --------------------------------------------------
64  def parse_tree(fh):
65      """parse input file"""
66
67      ini_patt = '([A-Za-z0-9]+)'
68      name_patt = ini_patt + '\s*=\s*(.+)'
69      begat_patt = ini_patt + '\s+and\s+' + ini_patt + '\s+begat\s+(.+)'
70      married_patt = ini_patt + '\s+married\s+' + ini_patt
71      edges = set()
72      nodes = {}
73
74      for line in fh:
75          name_match = re.match(name_patt, line)
76          begat_match = re.match(begat_patt, line)
77          married_match = re.match(married_patt, line)
78
79          if name_match:
80              initials, name = name_match.groups()
81              nodes[initials] = name
82          elif married_match:
83              p1, p2 = married_match.groups()
84              edges.add((p1, p2))
85          elif begat_match:
86              p1, p2, begat = begat_match.groups()
87              children = re.split('\s*,\s*', begat)
88              for parent in p1, p2:
89                  for child in children:
```

```
90                      edges.add((parent, child))
91
92      return nodes, edges
93
94
95  # ------------------------------------------------
96  if __name__ == '__main__':
97      main()
```

Chapter 18: Gematria

Write a Python program called `gematria.py`

Gematria is 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). It is not necessary, however, to encode this table in our program as Python provides 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, 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 members of a list (in order):

```
>>> for char in ['p', 'y', 't', 'h', 'o', 'n']:
...     print(ord(char))
...
112
121
116
104
111
110
```

Now you just need to `sum` those up for each word!

```
$ ./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
$ ./gematria.py 'foo bar baz'
324 309 317
$ ./gematria.py ../inputs/fox.txt
289 541 552 333 559 444 321 448 314
```

Solution

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

import argparse
import os
import re
import sys


# --------------------------------------------------
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')

    return parser.parse_args()


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

    args = get_args()
    text = args.text

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

    def clean(word):
        return re.sub('[^a-zA-Z0-9]', '', word)

    for line in text.splitlines():
        words = line.rstrip().split()
        nums = map(lambda word: str(sum(map(ord, clean(word)))), words)
        print(' '.join(nums))


# --------------------------------------------------
if __name__ == '__main__':
```

```
44      main()
```

Chapter 19: Histogram

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 `-c|--character` 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.

```
$ ./histy.py
usage: histy.py [-h] [-c str] [-m int] [-w int] [-i] [-f] str
histy.py: error: the following arguments are required: str
$ ./histy.py -h
usage: histy.py [-h] [-c 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
  -c str, --character str
                        Character 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)
$ ./histy.py ../inputs/fox.txt
T      1 |
a      1 |
b      1 |
c      1 |
d      1 |
e      3 |||
f      1 |
g      1 |
h      2 ||
i      1 |
j      1 |
k      1 |
```

```
l      1 |
m      1 |
n      1 |
o      4 ||||
p      1 |
q      1 |
r      2 ||
s      1 |
t      1 |
u      2 ||
v      1 |
w      1 |
x      1 |
y      1 |
z      1 |
$ ./histy.py ../inputs/const.txt -fim 100 -w 50 -c '#'
E   5107 ##################################################
T   3751 ####################################
O   2729 ##########################
S   2676 ##########################
A   2675 ##########################
N   2630 #########################
I   2433 #######################
R   2206 #####################
H   2029 ###################
L   1490 ##############
D   1230 ############
C   1164 ###########
F   1021 #########
U    848 ########
P    767 #######
M    730 #######
B    612 #####
Y    504 ####
V    460 ####
G    444 ####
W    375 ###
```

Solution

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

import argparse
import os
import re
from collections import Counter
from dire import die


# --------------------------------------------------
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('-c',
                        '--character',
                        help='Character 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')

    return parser.parse_args()


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

    args = get_args()
    text = args.text
    char = args.character
    width = args.width
    min_val = args.minimum

    if len(char) != 1:
        die('--character "{}" must be one character'.format(char))

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

    freqs = Counter(filter(lambda c: re.match(r'\w', c), list(text)))
    high = max(freqs.values())
    scale = high / width if high > width else 1
    items = map(lambda t: (t[1], t[0]),
                sorted([(v, k) for k, v in freqs.items()],
                       reverse=True)) if args.frequency_sort else sorted(
                           freqs.items())

    for c, num in items:
        if num < min_val:
            continue
        print('{} {:6} {}'.format(c, num, char * int(num / scale)))


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

Chapter 20: Guessing Game

Write a Python program called `guess.py` that plays a guessing game for a number between a `-m|--min` and `-x|--max` value (default 1 and 50, respectively) with a limited number of `-g|--guesses` (default 5). Complain if either `--min` or `--guesses` is less than 1. Accept a `-s|--seed` for `random.seed`. If the user guesses something that is not a number, complain about it.

The game is intended to actually be interactive, which makes it difficult to test. Here is how it should look in interactive mode:

```
$ ./guess.py -s 1
Guess a number between 1 and 50 (q to quit): 25
"25" is too high.
Guess a number between 1 and 50 (q to quit): foo
"foo" is not a number.
Guess a number between 1 and 50 (q to quit): 12
"12" is too high.
Guess a number between 1 and 50 (q to quit): 6
"6" is too low.
Guess a number between 1 and 50 (q to quit): 9
"9" is correct. You win!
```

Because I want to be able to write a test for this, I also want the program to accept an `-i|--inputs` option so that the game can also be played exactly the same but without the prompts for input:

```
$ ./guess.py -s 1 -i 25 foo 12 6 9
"25" is too high.
"foo" is not a number.
"12" is too high.
"6" is too low.
"9" is correct. You win!
```

You should be able to handle this in your inifinite game loop.

Solution

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

import argparse
import random
import re
import sys
from dire import die


# --------------------------------------------------
def get_args():
    """get args"""
    parser = argparse.ArgumentParser(
        description='Guessing game',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('-m',
                        '--min',
                        help='Minimum value',
                        metavar='int',
                        type=int,
                        default=1)

    parser.add_argument('-x',
                        '--max',
                        help='Maximum value',
                        metavar='int',
                        type=int,
                        default=50)

    parser.add_argument('-g',
                        '--guesses',
                        help='Number of guesses',
                        metavar='int',
                        type=int,
                        default=5)

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

```
44
45      parser.add_argument('-i',
46                          '--inputs',
47                          help='Inputs',
48                          metavar='str',
49                          type=str,
50                          nargs='+',
51                          default=[])
52
53      return parser.parse_args()
54
55
56  # --------------------------------------------------
57  def main():
58      """main"""
59      args = get_args()
60      low = args.min
61      high = args.max
62      guesses_allowed = args.guesses
63      inputs = args.inputs
64      random.seed(args.seed)
65
66      if low < 1:
67          die('--min "{}" cannot be lower than 1'.format(low))
68
69      if guesses_allowed < 1:
70          die('--guesses "{}" cannot be lower than 1'.format(guesses_allowed))
71
72      if low > high:
73          die('--min "{}" is higher than --max "{}"'.format(low, high))
74
75      secret = random.randint(low, high)
76      prompt = 'Guess a number between {} and {} (q to quit): '.format(low, high)
77      num_guesses = 0
78
79      while True:
80          guess = inputs.pop(0) if inputs else input(prompt)
81          num_guesses += 1
82
83          if re.match('q(uit)?', guess.lower()):
84              print('Now you will never know the answer.')
85              sys.exit()
86
87          # Method 1: test if the guess is a digit
88          if not guess.isdigit():
89              print('"{}" is not a number.'.format(guess))
```

```
90              continue
91          num = int(guess)
92
93          # Method 2: try/except
94          num = 0
95          try:
96              num = int(guess)
97          except:
98              warn('"{}" is not an integer'.format(guess))
99              continue
100
101         if not low <= num <= high:
102             print('Number "{}" is not in the allowed range'.format(num))
103         elif num == secret:
104             print('"{}" is correct. You win!'.format(num))
105             break
106         else:
107             print('"{}" is too {}.'.format(num,
108                                            'low' if num < secret else 'high'))
109
110         if num_guesses >= guesses_allowed:
111             print(
112                 'Too many guesses, loser! The number was "{}."'.format(secret))
113             sys.exit(1)
114
115
116 # --------------------------------------------------
117 if __name__ == '__main__':
118     main()
```

Chapter 21: 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"'.

```
$ ./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'" + (ing_word.group(2) or '')
    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()
```

Chapter 22: Mad Libs

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 reuqested, 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, please add a `-i|--inputs` option so the test suite can pass in all the answers and bypass the `input` calls.

```
$ ./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)
$ cat help.txt
<exclamation>! I need <noun>!
<exclamation>! Not just <noun>!
<exclamation>! You know I need <noun>!
<exclamation>!
$ ./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!
$ ./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.

Solution

```
#!/usr/bin/env python3
"""Mad Libs"""

import argparse
import os
import re
import sys
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))
```

```
44
45      if not blanks: die('File "{}" has no placeholders'.format(args.file.name))
46
47      for blank in blanks:
48          name = blank.group(1)
49          answer = inputs.pop(0) if inputs else input('{}: '.format(
50              name.replace('<', '').replace('>', '')))
51          text = re.sub(name, answer, text, count=1)
52
53      print(text)
54
55
56  # ----------------------------------------------------
57  if __name__ == '__main__':
58      main()
```

Chapter 23: 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
- 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'),
              ('0', '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))

```

```
44      for possible in sorted(product(*chars)):
45          s = ''.join(possible)
46          print(s, 'OK' if re.search(regex, s) else 'NO')
47
48
49  # -----------------------------------------------
50  if __name__ == '__main__':
51      main()
```

Chapter 24: Markov Chains for Words

Write a Python program called `markov.py` that uses the Markov chain algorithm to generate new words from 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.

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

```
$ ./markov.py
usage: markov.py [-h] [-n int] [-k int] [-m int] [-s int] [-d] FILE [FILE ...]
markov.py: error: the following arguments are required: FILE
$ ./markov.py -h
usage: markov.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)
$ ./markov.py /usr/share/dict/words -s 1
  1: oveli
  2: uming
  3: uylatiteda
  4: owsh
  5: uuse
  6: ismandl
  7: efortai
  8: eyhopy
  9: auretrab
 10: ozogralach
$ ./markov.py ../inputs/const.txt -s 2 -k 3
  1: romot
  2: leasonsusp
  3: gdoned
  4: bunablished
  5: neithere
  6: achmen
  7: reason
  8: nmentyone
  9: effereof
 10: eipts
```

Solution

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

import argparse
import logging
import os
import random
import re
import sys
from collections import defaultdict


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

    parser = argparse.ArgumentParser(
        description='Markov chain for characters/words',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file',
                        metavar='FILE',
                        nargs='+',
                        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,
```

```
44                          default=12)
45
46      parser.add_argument('-s',
47                          '--seed',
48                          help='Random seed',
49                          metavar='int',
50                          type=int,
51                          default=None)
52
53      parser.add_argument('-d',
54                          '--debug',
55                          help='Debug to ".log"',
56                          action='store_true')
57
58      return parser.parse_args()
59
60
61  # --------------------------------------------------
62  def main():
63      """Make a jazz noise here"""
64
65      args = get_args()
66      k = args.kmer_size
67      random.seed(args.seed)
68
69      logging.basicConfig(
70          filename='.log',
71          filemode='w',
72          level=logging.DEBUG if args.debug else logging.CRITICAL)
73
74      # debate use of set/list in terms of letter frequencies
75      chains = defaultdict(list)
76      for file in args.file:
77          for line in open(file):
78              for word in line.lower().split():
79                  word = re.sub('[^a-z]', '', word)
80                  for i in range(0, len(word) - k):
81                      kmer = word[i:i + k + 1]
82                      chains[kmer[:-1]].append(kmer[-1])
83
84      logging.debug(chains)
85
86      kmers = list(chains.keys())
87      starts = set()
88
89      for i in range(1, args.num_words + 1):
```

```
 90          word = ''
 91          while not word:
 92              kmer = random.choice(kmers)
 93              if not kmer in starts and chains[kmer] and re.search(
 94                      '[aeiou]', kmer):
 95                  starts.add(kmer)
 96                  word = kmer
 97
 98          length = random.choice(range(k + 2, args.max_word))
 99          logging.debug('Make a word {} long starting with "{}"'.format(
100              length, word))
101          while len(word) < length:
102              if not chains[kmer]: break
103              char = random.choice(list(chains[kmer]))
104              logging.debug('char = "{}"'.format(char))
105              word += char
106              kmer = kmer[1:] + char
107
108          logging.debug('word = "{}"'.format(word))
109          print('{:3}: {}'.format(i, word))
110
111
112  # ----------------------------------------------------
113  if __name__ == '__main__':
114      main()
```

Chapter 25: Pig Latin

Write a Python program named `piggie.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 a file 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."

```
$ ./piggie.py
usage: piggie.py [-h] [-o str] FILE [FILE ...]
piggie.py: error: the following arguments are required: FILE
$ ./piggie.py -h
usage: piggie.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)
[cholla@~/work/python/playful_python/piggie]$ ./piggie.py
usage: piggie.py [-h] [-o str] FILE [FILE ...]
piggie.py: error: the following arguments are required: FILE
[cholla@~/work/python/playful_python/piggie]$ ./piggie.py -h
usage: piggie.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)
```

```
$ ./piggie.py ../inputs/sonnet-29.txt
  1: sonnet-29.txt
Done, wrote 1 file to "out-yay".
$ head out-yay/sonnet-29.txt
onnet-Say 29-yay
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 y-may outcast-yay ate-stay,
And-yay ouble-tray eaf-day eaven-hay ith-way y-may ootless-bay ies-cray,
And-yay ook-lay upon-yay elf-mysay and-yay urse-cay y-may ate-fay,
ishing-Way e-may ike-lay o-tay one-yay ore-may ich-ray in-yay ope-hay,
eatured-Fay ike-lay im-hay, ike-lay im-hay ith-way iends-fray ossessed-pay,
esiring-Day is-thay an-may's-yay art-yay and-yay at-thay an-may's-yay ope-scay,
```

Solution

```
#!/usr/bin/env python3
"""Convert text to Pig Latin"""

import argparse
import os
import re
import string
from dire import warn


# --------------------------------------------------
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='+',
                        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)

```

```
44      num_files = 0
45      for i, file in enumerate(args.file, start=1):
46          basename = os.path.basename(file)
47          out_file = os.path.join(out_dir, basename)
48          out_fh = open(out_file, 'wt')
49          print('{:3}: {}'.format(i, basename))
50
51          if not os.path.isfile(file):
52              warn('"{}" is not a file.'.format(file))
53              continue
54
55          num_files += 1
56          for line in open(file):
57              for bit in re.split(r"([\w']+)", line):
58                  out_fh.write(pig(bit))
59
60          out_fh.close()
61
62      print('Done, wrote {} file{} to "{}".'.format(
63          num_files, '' if num_files == 1 else 's', out_dir))
64
65
66  # --------------------------------------------------
67  def pig(word):
68      """Create Pig Latin version of a word"""
69
70      if re.match(r"^[\w']+$", word):
71          consonants = re.sub('[aeiouAEIOU]', '', string.ascii_letters)
72          match = re.match('^([' + consonants + ']+)(.+)', word)
73          if match:
74              word = '-'.join([match.group(2), match.group(1) + 'ay'])
75          else:
76              word = word + '-yay'
77
78      return word
79
80
81  # --------------------------------------------------
82  if __name__ == '__main__':
83      main()
```

Chapter 26: 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, it would be best to discount any leading consonants, e.g., the words "listen" and "glisten" rhyme but only if you compare the "isten" part. The program should take an optional `-w|--wordlist` argument (default `/usr/share/dict/words`) for the comparisons.

See also:

- https://en.wikipedia.org/wiki/Soundex
- https://pypi.org/project/soundex/)

```
$ ./rhymer.py
usage: rhymer.py [-h] [-w str] str
rhymer.py: error: the following arguments are required: str
[cholla@~/work/python/playful_python/soundex-rhymer]$ ./rhymer.py -h
usage: rhymer.py [-h] [-w str] str

Use Soundex to find rhyming words

positional arguments:
  str                   Word

optional arguments:
  -h, --help            show this help message and exit
  -w str, --wordlist str
                        Wordlist (default: /usr/share/dict/words)
$ ./rhymer.py orange | head
boring
borning
boronic
borrowing
chloranemic
chlorinize
chlorinous
chorionic
choromanic
clowring
```

Solution

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

import argparse
import re
import soundex
import string
import sys


# --------------------------------------------------
def get_args():
    """get command-line arguments"""
    parser = argparse.ArgumentParser(
        description='Use Soundex to find rhyming words',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

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

    parser.add_argument('-w',
                        '--wordlist',
                        metavar='str',
                        help='Wordlist',
                        default='/usr/share/dict/words')

    return parser.parse_args()


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

    stem = word
    consonants = [c for c in string.ascii_lowercase if c not in 'aeiou']
    regex = re.compile('^[' + ''.join(consonants) + ']+(.+)')

    def stemmer(word):
        match = regex.search(word)
        return match.group(1) if match else word

    sndx = soundex.Soundex()
```

```
44      cmp = sndx.soundex(stemmer(word))
45
46      for line in open(wordlist):
47          for w in line.split():
48              if w != word and sndx.soundex(stemmer(w)) == cmp:
49                  print(w)
50
51
52  # ------------------------------------------------
53  if __name__ == '__main__':
54      main()
```

Chapter 27: Substring Guessing Game

Write a Python program called `sub.py` that plays a guessing game where you read a `-f|--file` input (default `/usr/share/dict/words`) and use a given `-k|--ksize` to find all the words grouped by their shared kmers. Remove any kmers where the number of words is fewer than `-m|--min_words`. Also accept a `-s|--seed` for `random.seed` for testing purposes. Prompt the user to guess a word for a randomly chosen kmer. If their guess is not present in the shared list, taunt them mercilessly. If their guess is present, affirm their worth and prompt to guess again. Allow them to use `!` to quit and `?` to be provided a hint (a word from the list). For both successful guesses and hints, remove the word from the shared list. When they have quit or exhausted the list, quit play. At the end of the game, report the number of found words.

```
$ ./sub.py -h
usage: sub.py [-h] [-f str] [-s int] [-m int] [-k int]

Find words sharing a substring

optional arguments:
  -h, --help            show this help message and exit
  -f str, --file str    Input file (default: /usr/share/dict/words)
  -s int, --seed int    Random seed (default: None)
  -m int, --min_words int
                        Minimum number of words for a given kmer (default: 3)
  -k int, --ksize int   Size of k (default: 4)
$ ./sub.py
Name a word that contains "slak" [!=quit, ?=hint] (10 left) slake
Totes! "slake" is found!
Name a word that contains "slak" [!=quit, ?=hint] (9 left) ?
For instance, "breislakite"...
Name a word that contains "slak" [!=quit, ?=hint] (8 left) unslakable
Totes! "unslakable" is found!
Name a word that contains "slak" [!=quit, ?=hint] (7 left) q
What is wrong with you?
Name a word that contains "slak" [!=quit, ?=hint] (7 left) !
Quitter!
Hey, you found 2 words! Not bad.
```

Solution

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

import argparse
import os
import random
import re
import sys
from collections import defaultdict
from dire import die


# --------------------------------------------------
def get_args():
    """get command-line arguments"""
    parser = argparse.ArgumentParser(
        description='Find words sharing a substring',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('-f',
                        '--file',
                        metavar='str',
                        help='Input file',
                        type=str,
                        default='/usr/share/dict/words')

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

    parser.add_argument('-m',
                        '--min_words',
                        help='Minimum number of words for a given kmer',
                        metavar='int',
                        type=int,
                        default=3)

    parser.add_argument('-k',
                        '--ksize',
                        help='Size of k',
                        metavar='int',
```

```
44                          type=int,
45                          default=4)
46
47      return parser.parse_args()
48
49
50  # --------------------------------------------------
51  def get_words(file):
52      """Get words from input file"""
53
54      if not os.path.isfile(file):
55          die('"{}" is not a file')
56
57      words = set()
58      for line in open(file):
59          for word in line.split():
60              words.add(re.sub('[^a-zA-Z0-9]', '', word.lower()))
61
62      if not words:
63          die('No usable words in "{}"'.format(file))
64
65      return words
66
67
68  # --------------------------------------------------
69  def get_kmers(words, k, min_words):
70      """ Find all words sharing kmers"""
71
72      if k <= 1:
73          die('-k "{}" must be greater than 1'.format(k))
74
75      shared = defaultdict(list)
76      for word in words:
77          for kmer in [word[i:i + k] for i in range(len(word) - k + 1)]:
78              shared[kmer].append(word)
79
80      # Select kmers having enough words (can't use `pop`!)
81
82      # Method 1: for loop
83      ok = dict()
84      for kmer in shared:
85          if len(shared[kmer]) >= min_words:
86              ok[kmer] = shared[kmer]
87
88      # Method 2: list comprehension
89      # ok = dict([(kmer, shared[kmer]) for kmer in shared
```

```
90      #             if len(shared[kmer]) >= min_words])
91
92      # Method 3: map/filter
93      # ok = dict(
94      #     map(lambda kmer: (kmer, shared[kmer]),
95      #         filter(lambda kmer: len(shared[kmer]) >= min_words,
96      #                shared.keys())))
97
98      return ok
99
100
101 # --------------------------------------------------
102 def main():
103     """Make a jazz noise here"""
104
105     args = get_args()
106
107     random.seed(args.seed)
108
109     shared = get_kmers(get_words(args.file), args.ksize, args.min_words)
110
111     # Choose a kmer, setup game state
112     kmer = random.choice(list(shared.keys()))
113     guessed = set()
114     found = []
115     prompt = 'Name a word that contains "{}" [!=quit, ?=hint] '.format(kmer)
116     compliments = ['Nice', 'Rock on', 'Totes', 'Fantastic', 'Excellent']
117     taunts = [
118         'Surely you jest!', 'Are you kidding me?',
119         'You must have rocks for brains.', 'What is wrong with you?'
120     ]
121
122     #print(kmer, shared[kmer])
123
124     while True:
125         num_left = len(shared[kmer])
126         if num_left == 0:
127             print('No more words!')
128             break
129
130         guess = input(prompt + '({} left) '.format(num_left)).lower()
131
132         if guess == '?':
133             # Provide a hint
134             pos = random.choice(range(len(shared[kmer])))
135             word = shared[kmer].pop(pos)
```

```
136             print('For instance, "{}"...'.format(word))
137
138         elif guess == '!':
139             # Bail
140             print('Quitter!')
141             break
142
143         elif guess in guessed:
144             # Chastise
145             print('You have already guessed "{}"'.format(guess))
146
147         elif guess in shared[kmer]:
148             # Remove the word, feedback with compliment
149             pos = shared[kmer].index(guess)
150             word = shared[kmer].pop(pos)
151             print('{}! "{}" is found!'.format(random.choice(compliments),
152                                               word))
153             found.append(word)
154             guessed.add(guess)
155
156         else:
157             # Taunt
158             print(random.choice(taunts))
159
160     # Game over, man!
161     if found:
162         n = len(found)
163         print('Hey, you found {} word{}! Not bad.'.format(
164             n, '' if n == 1 else 's'))
165     else:
166         print('Wow, you found no words. You suck!')
167
168
169 # ------------------------------------------------
170 if __name__ == '__main__':
171     main()
```

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
```

```
44
45      # for player in ['X', 'O']:
46      #     for i, j, k in winning:
47      #         chars = ''.join([state[i], state[j], state[k]])
48      #         if ''.join(chars) == '{}{}{}'.format(player, player, player):
49      #             winner = player
50      #             break
51
52      for player in ['X', 'O']:
53          for i, j, k in winning:
54              combo = [state[i], state[j], state[k]]
55              if combo == [player, player, player]:
56                  winner = '{} has won'.format(player)
57                  break
58
59      # for combo in winning:
60      #     group = list(map(lambda i: state[i], combo))
61      #     for player in ['X', 'O']:
62      #         if all(x == player for x in group):
63      #             winner = player
64      #             break
65
66      print(winner)
67
68
69  # -----------------------------------------------
70  if __name__ == '__main__':
71      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: War

> The generation of random numbers is too important to be left to chance. – Robert R. Coveyou

Create a Python program called `war.py` that plays the card game "War." The program will use the `random` module to shuffle a deck of cards, so your program will need to accept a `-s|--seed` argument (default: `None`) which you will use to call `random.seed`, if present.

First you program will need to create a deck of cards. You will need to use the Unicode symbols for the suites ( ) [which won't display in the PDF, so consult the Markdown file] and combine those with the numbers 2-10 and the letters "J", "Q," "K," and "A." (hint: look at `itertools.product`).

```
>>> from itertools import product
>>> a = list('AB')
>>> b = range(2)
>>> list(product(a, b))
[('A', 0), ('A', 1), ('B', 0), ('B', 1)]
```

NB: You must sort your deck and then use the `random.shuffle` method so that your cards will be in the correct order to pass the tests!

In the real game of War, the cards are shuffled and then dealt one card each first to the non-dealer, then to the dealer, until all cards are dealt and each player has 26 cards. We will not be modeling this behavior. When writing your version of the game, simply `pop` two cards off the deck as the cards for player 1 and player 2, respectively. Compare the two cards by ignoring the suite and evaluating the value where 2 is the lowest and Aces are the highest. When two cards have the same values (e.g., two 5s or two Jacks), print "WAR!" In the real game, this initiates a sub-game of War which is a "recursive" algorithm which we will not bother modeling. Keep track of which player wins each round where no points are awarded in a tie. At the end, report the points for each player and state the winner. In the event of a tie, print "DRAW."

```
$ ./war.py -h
usage: war.py [-h] [-s int]

"War" cardgame

optional arguments:
  -h, --help          show this help message and exit
  -s int, --seed int  Random seed (default: None)
$ ./war.py -s 1
 9  J P2
 A  5 P1
 4  8 P2
```

```
 6  3 P1
 5  3 P1
 K 10 P1
 7  7 WAR!
 2  4 P2
 2 10 P2
 6  5 P1
 2  6 P2
 4  8 P2
 J  9 P1
10  Q P2
 8  7 P1
 K  Q P1
10  2 P1
 9  9 WAR!
 8  J P2
 3  5 P2
 Q  4 P1
 6  A P2
 K  7 P1
 Q  3 P1
 A  K P1
 A  J P1
P1 14 P2 10: Player 1 wins
$ ./war.py -s 2
 4  6 P2
 K  J P1
 J  4 P1
 7  4 P1
 Q 10 P1
 5  3 P1
 K  9 P1
 2  Q P2
 7  A P2
 3  A P2
 5  8 P2
 2 10 P2
10  K P2
 2  3 P2
 Q  8 P1
 6  J P2
 6  8 P2
 8  7 P1
 5  2 P1
 6  J P2
 9  9 WAR!
```

```
  K   A P2
 10   Q P2
  7   5 P1
  9   A P2
  4   3 P1
P1 11 P2 14: Player 2 wins
$ ./war.py -s 10
  J   3 P1
  2   5 P2
  Q  10 P1
 10   4 P1
  6   5 P1
  3   J P2
  K   8 P1
  5   8 P2
  5   3 P1
  J  10 P1
 10   J P2
  A   7 P1
  K   Q P1
  7   A P2
  9   9 WAR!
  2   6 P2
  K   A P2
  6   Q P2
  8   9 P2
  3   7 P2
  8   Q P2
  6   4 P1
  7   2 P1
  4   4 WAR!
  9   2 P1
  K   A P2
P1 12 P2 12: DRAW
```

Solution

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

import argparse
import random
import sys
from itertools import product


# --------------------------------------------------
def get_args():
    """get command-line arguments"""
    parser = argparse.ArgumentParser(
        description='"War" cardgame',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

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

    return parser.parse_args()


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

    if seed is not None:
        random.seed(seed)

    suits = list('♠♥♦♣')
    values = list(map(str, range(2, 11))) + list('JQKA')
    cards = sorted(map(lambda t: '{}{}'.format(*t), product(suits, values)))
    random.shuffle(cards)

    p1_wins = 0
    p2_wins = 0

    card_value = dict(
```

```
44          list(map(lambda t: list(reversed(t)), enumerate(list(values)))))
45
46      while cards:
47          p1, p2 = cards.pop(), cards.pop()
48          v1, v2 = card_value[p1[1:]], card_value[p2[1:]]
49          res = ''
50
51          if v1 > v2:
52              p1_wins += 1
53              res = 'P1'
54          elif v2 > v1:
55              p2_wins += 1
56              res = 'P2'
57          else:
58              res = 'WAR!'
59
60          print('{:>3} {:>3} {}'.format(p1, p2, res))
61
62      print('P1 {} P2 {}: {}'.format(
63          p1_wins, p2_wins, 'Player 1 wins' if p1_wins > p2_wins else
64          'Player 2 wins' if p2_wins > p1_wins else 'DRAW'))
65
66
67  # -----------------------------------------------
68  if __name__ == '__main__':
69      main()
```

Chapter 31: 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.

```
$ ./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)
$ ./presto.py presto
presto =
   1. poster
   2. repost
   3. respot
   4. stoper
$ ./presto.py listen
listen =
   1. enlist
   2. silent
   3. tinsel
$ ./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

Solution

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

import argparse
import logging
import os
import re
import sys
from collections import defaultdict, Counter
from itertools import combinations, permutations, product, chain
from dire import warn, die


# --------------------------------------------------
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=str,
                        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
    word_list = args.wordlist

    if not os.path.isfile(word_list):
        die('--wordlist "{}" is not a file'.format(word_list))

    logging.basicConfig(
        filename='.log',
        filemode='w',
        level=logging.DEBUG if args.debug else logging.CRITICAL)

    words = defaultdict(set)
    for line in open(word_list):
        for word in line.split():
            clean = re.sub('[^a-z0-9]', '', 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 i 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), i)))))

        for keys in key_combos:
            logging.debug('Searching keys {}'.format(keys))
            word_combos = list(product(*list(map(lambda k: words[k], keys))))

            for t in word_combos:
                if Counter(''.join(t)) == counts:
                    for p in filter(
                            lambda x: x != text,
                            map(lambda x: ' '.join(x), permutations(t))):
                        anagrams.add(p)

            logging.debug('# anagrams = {}'.format(len(anagrams)))

    logging.debug('Finished searching')
```

```
 90
 91      if anagrams:
 92          print('{} ='.format(text))
 93          for i, t in enumerate(sorted(anagrams), 1):
 94              print('{:4}. {}'.format(i, t))
 95      else:
 96          print('No anagrams for "{}".'.format(text))
 97
 98
 99  # -----------------------------------------------
100  if __name__ == '__main__':
101      main()
```

Chapter 32: 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.

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, I cannot write an test suite, so you can play my version and then try to write one like it. 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 -h
usage: hangman.py [-h] [-l MAXLEN] [-n MINLEN] [-m MISSES] [-s SEED]
                  [-w WORDLIST]

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)
$ ./hangman.py
_ _ _ _ _ _ _ _ (Misses: 0)
Your guess? ("?" for hint, "!" to quit) a
_ _ _ _ _ _ _ _ (Misses: 1)
Your guess? ("?" for hint, "!" to quit) i
_ _ _ _ _ _ i _ (Misses: 1)
Your guess? ("?" for hint, "!" to quit) e
```

```
_ _ _ _ _ _ i _ (Misses: 2)
Your guess? ("?" for hint, "!" to quit) o
_ o _ _ _ _ i _ (Misses: 2)
Your guess? ("?" for hint, "!" to quit) u
_ o _ _ _ _ i _ (Misses: 3)
Your guess? ("?" for hint, "!" to quit) y
_ o _ _ _ _ i _ (Misses: 4)
Your guess? ("?" for hint, "!" to quit) c
_ o _ _ _ _ i _ (Misses: 5)
Your guess? ("?" for hint, "!" to quit) d
_ o _ _ _ _ i _ (Misses: 6)
Your guess? ("?" for hint, "!" to quit) p
_ o _ _ _ _ i p (Misses: 6)
Your guess? ("?" for hint, "!" to quit) m
_ o _ _ _ _ i p (Misses: 7)
Your guess? ("?" for hint, "!" to quit) n
_ o _ _ _ _ i p (Misses: 8)
Your guess? ("?" for hint, "!" to quit) s
_ o s _ s _ i p (Misses: 8)
Your guess? ("?" for hint, "!" to quit) t
_ o s t s _ i p (Misses: 8)
Your guess? ("?" for hint, "!" to quit) h
You win. You guessed "hostship" with "8" misses!
$ ./hangman.py -m 2
_ _ _ _ _ _ _ _ _ _ (Misses: 0)
Your guess? ("?" for hint, "!" to quit) a
_ _ _ _ _ _ a _ _ a (Misses: 0)
Your guess? ("?" for hint, "!" to quit) b
_ _ _ _ _ _ a _ _ a (Misses: 1)
Your guess? ("?" for hint, "!" to quit) c
You lose, loser!  The word was "metromania."
```

Solution

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

import argparse
import os
import random
import re
import sys
from dire import die


# --------------------------------------------------
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=str,
                        default='/usr/share/dict/words')

    return parser.parse_args()


# --------------------------------------------------
def bail(msg):
    """Print a message to STDOUT and quit with no error"""
    print(msg)
    sys.exit(0)


# --------------------------------------------------
def main():
    """main"""
    args = get_args()
    max_len = args.maxlen
    min_len = args.minlen
    max_misses = args.misses
    wordlist = args.wordlist

    random.seed(args.seed)

    if not os.path.isfile(wordlist):
        die('--wordlist "{}" is not a file.'.format(wordlist))

    if min_len < 1:
        die('--minlen must be positive')

    if not 3 <= max_len <= 20:
        die('--maxlen should be between 3 and 20')

    if min_len > max_len:
        die('--minlen ({}) is greater than --maxlen ({})'.format(
            min_len, max_len))

    good_word = re.compile('^[a-z]{' + str(min_len) + ',' + str(max_len) +
                           '}$')
    words = [w for w in open(wordlist).read().split() if good_word.match(w)]

    word = random.choice(words)
    play({'word': word, 'max_misses': max_misses})


```

```
 90  # -----------------------------------------------
 91  def play(state):
 92      """Loop to play the game"""
 93      word = state.get('word') or ''
 94
 95      if not word: die('No word!')
 96
 97      guessed = state.get('guessed') or list('_' * len(word))
 98      prev_guesses = state.get('prev_guesses') or set()
 99      num_misses = state.get('num_misses') or 0
100      max_misses = state.get('max_misses') or 0
101
102      if ''.join(guessed) == word:
103          msg = 'You win. You guessed "{}" with "{}" miss{}!'
104          bail(msg.format(word, num_misses, '' if num_misses == 1 else 'es'))
105
106      if num_misses >= max_misses:
107          bail('You lose, loser!  The word was "{}."'.format(word))
108
109      print('{} (Misses: {})'.format(' '.join(guessed), num_misses))
110      new_guess = input('Your guess? ("?" for hint, "!" to quit) ').lower()
111
112      if new_guess == '!':
113          bail('Better luck next time, loser.')
114      elif new_guess == '?':
115          new_guess = random.choice([x for x in word if x not in guessed])
116          num_misses += 1
117
118      if not re.match('^[a-z]$', new_guess):
119          print('"{}" is not a letter'.format(new_guess))
120          num_misses += 1
121      elif new_guess in prev_guesses:
122          print('You already guessed that')
123      elif new_guess in word:
124          prev_guesses.add(new_guess)
125          last_pos = 0
126          while True:
127              pos = word.find(new_guess, last_pos)
128              if pos < 0:
129                  break
130              elif pos >= 0:
131                  guessed[pos] = new_guess
132                  last_pos = pos + 1
133      else:
134          num_misses += 1
135
```

```
136     play({
137         'word': word,
138         'guessed': guessed,
139         'num_misses': num_misses,
140         'prev_guesses': prev_guesses,
141         'max_misses': max_misses
142     })
143 
144 
145 # -----------------------------------------------
146 if __name__ == '__main__':
147     main()
```

Chapter 33: 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.

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
Discoveries; To constitute Tribunals inferior to the seat of the
Senate and House of Representatives shall have been committed, which
district shall have the Qualifications requisite for Electors of the
sixth Year, so that one third may be imposed on such Importation, not
exceeding three on the Journal. Neither House, during the Time of
Adjournment, he may require it. No Bill of Attainder or ex post facto
Law shall be established by Law: but the Party convicted shall
nevertheless be liable and subject to their Consideration such
Measures as he shall nominate, and by and with the Advice and Consent
of the government of the United States under this Constitution, or,
on the List the said Office, the same State claiming Lands under
Grants of different States; between Citizens of each shall constitute
a Quorum to do Business; but a smaller number may adjourn from day to
day, and may be included within this Union, according to their
Consideration such Measures as he shall nominate, and by and with the
Advice and Consent of the United States.
```

Solution

```
#!/usr/bin/env python3
"""Markov Chain"""

import argparse
import logging
import os
import random
import string
import sys
from pprint import pprint as pp
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',
```

```
44                          help='Random seed',
45                          metavar='int',
46                          type=int,
47                          default=None)
48
49      parser.add_argument('-w',
50                          '--text_width',
51                          help='Max number of characters per line',
52                          metavar='int',
53                          type=int,
54                          default=70)
55
56      parser.add_argument('-d',
57                          '--debug',
58                          help='Debug to ".log"',
59                          action='store_true')
60
61      return parser.parse_args()
62
63
64  # --------------------------------------------------
65  def main():
66      """Make a jazz noise here"""
67
68      args = get_args()
69      num_words = args.num_words
70      char_max = args.length
71      text_width = args.text_width
72
73      random.seed(args.seed)
74
75      logging.basicConfig(
76          filename='.log',
77          filemode='w',
78          level=logging.DEBUG if args.debug else logging.CRITICAL)
79
80      all_words = defaultdict(list)
81      for fh in args.training:
82          words = fh.read().split()
83
84          for i in range(0, len(words) - num_words):
85              l = words[i:i + num_words + 1]
86              all_words[tuple(l[:-1])].append(l[-1])
87
88      logging.debug('all words = {}'.format(all_words))
89
```

```
90      prev = ''
91      while not prev:
92          start = random.choice(
93              list(
94                  filter(lambda w: w[0][0] in string.ascii_uppercase,
95                         all_words.keys())))
96          if all_words[start]:
97              prev = start
98
99      logging.debug('Starting with "{}"'.format(prev))
100
101     p = ' '.join(prev)
102     char_count = len(p)
103     print(p, end=' ')
104     line_width = char_count
105
106     while True:
107         if not prev in all_words: break
108
109         new_word = random.choice(all_words[prev])
110         new_len = len(new_word) + 1
111         logging.debug('chose = "{}" from {}'.format(new_word, all_words[prev]))
112
113         if line_width + new_len > text_width:
114             print()
115             line_width = new_len
116         else:
117             line_width += new_len
118
119         char_count += new_len
120         print(new_word, end=' ')
121         if char_count >= char_max and new_word[-1] in '.!?': break
122         prev = prev[1:] + (new_word, )
123
124     logging.debug('Finished')
125     print()
126
127
128 # -----------------------------------------------
129 if __name__ == '__main__':
130     main()
```

Chapter 34: 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 35: 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')

```

```
44      parser.add_argument('-D', '--debug', help='Debug', action='store_true')
45
46      return parser.parse_args()
47
48
49  # -------------------------------------------------
50  def encode_word(word, table):
51      """Encode word using given table"""
52
53      coded = []
54      for char in word.upper():
55          logging.debug(char)
56          if char != ' ' and char in table:
57              coded.append(table[char])
58
59      encoded = ' '.join(coded)
60      logging.debug('endoding "{}" to "{}"'.format(word, encoded))
61
62      return encoded
63
64
65  # -------------------------------------------------
66  def decode_word(encoded, table):
67      """Decode word using given table"""
68
69      decoded = []
70      for code in encoded.split(' '):
71          if code in table:
72              decoded.append(table[code])
73
74      word = ''.join(decoded)
75      logging.debug('dedoding "{}" to "{}"'.format(encoded, word))
76
77      return word
78
79
80  # -------------------------------------------------
81  def test_encode_word():
82      """Test Encoding"""
83
84      assert encode_word('sos', ENCODE_ITU) == '... --- ...'
85      assert encode_word('sos', ENCODE_MORSE) == '... .,. ...'
86
87
88  # -------------------------------------------------
89  def test_decode_word():
```

```
90      """Test Decoding"""
91
92      assert decode_word('... --- ...', DECODE_ITU) == 'SOS'
93      assert decode_word('... .,. ...', DECODE_MORSE) == 'SOS'
94
95
96  # -------------------------------------------------
97  def test_roundtrip():
98      """Test En/decoding"""
99
100     random_str = lambda: ''.join(random.sample(string.ascii_lowercase, k=10))
101     for _ in range(10):
102         word = random_str()
103         for encode_tbl, decode_tbl in [(ENCODE_ITU, DECODE_ITU),
104                                        (ENCODE_MORSE, DECODE_MORSE)]:
105
106             assert word.upper() == decode_word(encode_word(word, encode_tbl),
107                                                decode_tbl)
108
109
110 # -------------------------------------------------
111 def main():
112     """Make a jazz noise here"""
113     args = get_args()
114     action = 'decode' if args.decode else 'encode'
115     output = open(args.outfile, 'wt') if args.outfile else sys.stdout
116     source = sys.stdin if args.input == '-' else open(args.input)
117
118     coding_table = ''
119     if args.coding == 'itu':
120         coding_table = ENCODE_ITU if action == 'encode' else DECODE_ITU
121     else:
122         coding_table = ENCODE_MORSE if action == 'encode' else DECODE_MORSE
123
124     logging.basicConfig(
125         filename='.log',
126         filemode='w',
127         level=logging.DEBUG if args.debug else logging.CRITICAL)
128
129     word_split = r'\s+' if action == 'encode' else r'\s{2}'
130
131     for line in source:
132         for word in re.split(word_split, line):
133             if action == 'encode':
134                 print(encode_word(word, coding_table), end='  ')
135             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 36: 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 37: Tranpose ABC Notation

Write a Python program called `transpose.py` that will read a file in ABC notation (https://en.wikipedia.org/wiki/ABC_notation) and transpose the melody line up or down by a given `-s|--shift` argument. Like the `rot13` exercise, it might be helpful to think of the space of notes (`ABCDEFG`) as a list which you can roll through. For instance, if you have the note `c` and want to transpose up a (minor) third (`-s 3`), you would make the new note `e`; similarly if you have the note `F` and you go up a (major) third, you get `A`. You will not need to worry about the actual number of semitones that you are being asked to shift, as the previous example showed that we might be shifting by a major/minor/augmented/diminished/pure interval. The purpose of the exercise is simply to practice with lists.

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

Tranpose ABC notation

positional arguments:
  FILE                  Input file

optional arguments:
  -h, --help            show this help message and exit
  -s int, --shift int   Interval to shift (default: 2)
$ ./transpose.py foo
"foo" is not a file
$ ./transpose.py songs/legacy.abc -s 1
--shift "1" must be between 2 and 8
$ ./transpose.py songs/legacy.abc
<score lang="ABC">
X:1
T:The Legacy Jig
M:6/8
L:1/8
R:jig
K:A
AGA CBC | aga abc | AGA CBC | e2B BGE |
AGA CBC | aga abc | baf feC |1 eCB BGE :|2 eCB BCe |:
fgf feC | eCB BCe | fgf feC | aeC BCe |
fgf feC | e2e efg | agf feC |1 eCB BCe :|2 eCB BGE |]
</score>
```

A sample ABC song is given:

```
$ cat songs/legacy.abc
<score lang="ABC">
X:1
T:The Legacy Jig
M:6/8
L:1/8
R:jig
K:G
GFG BAB | gfg gab | GFG BAB | d2A AFD |
GFG BAB | gfg gab | age edB |1 dBA AFD :|2 dBA ABd |:
efe edB | dBA ABd | efe edB | gdB ABd |
efe edB | d2d def | gfe edB |1 dBA ABd :|2 dBA AFD |]
</score>
```

If you use `new_py.py` to create your new program with the `file` as a single positional argument, you can use this code to get the input file and check that it is, indeed, a file:

```
args = get_args()
file = args.file

if not os.path.isfile(file):
    die('"{}" is not a file'.formate(file))
```

Now that you have a file, you can use a `for` loop to read it. Each line will still have a newline attached to the end, so you can use `rstrip()` to remove it:

```
for line in open(file):
    line = line.rstrip()
```

If a line starts with `<` and ends with `>` (cf. `str.startswith` and `str.endswith`), you can just print the line as-is. If the line starts with `K:`, then you have the key signature and should transpose it, e.g., if you have `K:A` and you are shifting a fifth, you should print `K:E`. If you have a line that starts with any other single uppercase letter and a colon, just print the line as-is. Finally, if you have a line that doesn't match any of the above conditions, you have a line of melody that needs to be transposed.

If you are unfamiliar with musical transposition, you may be a bit confused by the notion of a interval. A "second" equals a `--shift` of one note; that is, the distance from `A` to `B` is one note, but we call that a "second." Therefore, assume that the `--shift` argument is the name of the interval, e.g., `4` (a "fourth") is actually a move of three notes. That means the argument provided by the user should be in the range 2 to 8, inclusive, so complain if it is not.

Note that the transposition of a tune up a fourth is the same as down a fifth:

```
$ ./transpose.py songs/legacy.abc -s 4
```

```
<score lang="ABC">
X:1
T:The Legacy Jig
M:6/8
L:1/8
R:jig
K:C
CBC EDE | cbc cde | CBC EDE | g2D DBG |
CBC EDE | cbc cde | dca agE |1 gED DBG :|2 gED DEg |:
aba agE | gED DEg | aba agE | cgE DEg |
aba agE | g2g gab | cba agE |1 gED DEg :|2 gED DBG |]
</score>
$ ./transpose.py songs/legacy.abc -s -5
<score lang="ABC">
X:1
T:The Legacy Jig
M:6/8
L:1/8
R:jig
K:C
CBC EDE | cbc cde | CBC EDE | g2D DBG |
CBC EDE | cbc cde | dca agE |1 gED DBG :|2 gED DEg |:
aba agE | gED DEg | aba agE | cgE DEg |
aba agE | g2g gab | cba agE |1 gED DEg :|2 gED DBG |]
</score>
```

Solution

```
#!/usr/bin/env python3
"""Tranpose ABC notation"""

import argparse
import os
import re
import sys
from dire import die


# --------------------------------------------------
def get_args():
    """get command-line arguments"""
    parser = argparse.ArgumentParser(
        description='Tranpose ABC notation',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file', metavar='FILE', help='Input file')

    parser.add_argument('-s',
                        '--shift',
                        help='Interval to shift',
                        metavar='int',
                        type=int,
                        default=2)

    return parser.parse_args()


# --------------------------------------------------
def main():
    """Make a jazz noise here"""
    args = get_args()
    file = args.file
    shift = args.shift
    ucase = 'ABCDEFG'
    lcase = 'abcdefg'
    num_notes = 7

    if not 1 < abs(shift) <= 8:
        die('--shift "{}" must be between 2 and 8'.format(shift))

    if not os.path.isfile(file):
```

```
44          die('"{}" is not a file'.format(file))
45
46      # account for interval where a 2nd (-s 2) is a move of one note
47      shift = shift - 1 if shift > 0 else shift + 1
48
49      def transpose(note):
50          if note in lcase:
51              pos = lcase.index(note)
52              tran = (pos + shift) % num_notes
53              return lcase[tran]
54          elif note in ucase:
55              pos = ucase.index(note)
56              tran = (pos + shift) % num_notes
57              return ucase[tran]
58          else:
59              return note
60
61      for line in open(file):
62          line = line.rstrip()
63
64          if line.startswith('K:'):
65              key = line[2]
66              print('K:' + transpose(key))
67          elif (line.startswith('<') and line.endswith('>')) or re.match(
68                  '[A-Z]:\s?', line):
69              print(line)
70          else:
71              for char in line.rstrip():
72                  print(transpose(char), end='')
73
74              print()
75
76
77  # -----------------------------------------------
78  if __name__ == '__main__':
79      main()
```

Chapter 38: 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:
```

```
44              words.append(line.replace(' ', ''))
45
46      return puzzle, words
47
48
49  # -----------------------------------------------
50  def all_combos(puzzle):
51      """Find all combos in puzzle"""
52
53      num_rows = len(puzzle)
54      num_cols = len(puzzle[0])
55
56      if not all([len(row) == num_cols for row in puzzle]):
57          die('Uneven number of columns')
58
59      combos = []
60
61      # Horizontal
62      for row in puzzle:
63          combos.append(row)
64
65      # Vertical
66      for col_num in range(num_cols):
67          col = [puzzle[row_num][col_num] for row_num in range(num_rows)]
68          combos.append(col)
69
70      # Diagonals Up
71      for row_i in range(0, num_rows):
72          diag = []
73          col_num = 0
74          for row_j in range(row_i, -1, -1):
75              diag.append(puzzle[row_j][col_num])
76              col_num += 1
77
78          if diag:
79              combos.append(diag)
80
81      for col_i in range(1, num_cols):
82          diag = []
83
84          col_num = col_i
85          for row_num in range(num_rows - 1, -1, -1):
86              diag.append(puzzle[row_num][col_num])
87              col_num += 1
88              if col_num == num_cols:
89                  break
```

```
 90
 91          if diag:
 92              combos.append(diag)
 93
 94      # Diagonals Down
 95      for row_i in range(0, num_rows):
 96          diag = []
 97          col_num = 0
 98          for row_j in range(row_i, num_rows):
 99              diag.append(puzzle[row_j][col_num])
100              col_num += 1
101              if col_num == num_cols:
102                  break
103
104          if diag:
105              combos.append(diag)
106
107      for col_i in range(0, num_cols):
108          diag = []
109
110          col_num = col_i
111          for row_num in range(0, num_rows):
112              diag.append(puzzle[row_num][col_num])
113              col_num += 1
114              if col_num == num_cols:
115                  break
116
117          if diag:
118              combos.append(diag)
119
120      combos.extend([list(reversed(c)) for c in combos])
121      return combos
122
123
124  # ------------------------------------------------
125  def fst(t):
126      """Return first element of a tuple"""
127
128      return t[0]
129
130
131  # ------------------------------------------------
132  def snd(t):
133      """Return second element of a tuple"""
134      return t[1]
135
```

```
136
137 # --------------------------------------------------
138 def main():
139     """Make a jazz noise here"""
140
141     args = get_args()
142     puzzle, words = read_puzzle(args.file)
143     combos = all_combos(puzzle)
144     found = set()
145     reveal = set()
146     for word in words:
147         for combo in combos:
148             test = ''.join(map(fst, combo))
149             if word in test:
150                 start = test.index(word)
151                 end = start + len(word)
152                 for cell in map(snd, combo[start:end]):
153                     reveal.add(cell)
154                 found.add(word)
155                 break
156
157     for row in puzzle:
158         cells = [c[0] if c[1] in reveal else '.' for c in row]
159         print(''.join(cells))
160
161     missing = [w for w in words if not w in found]
162     if missing:
163         print('Failed to find:')
164         for i, word in enumerate(missing, 1):
165             print('{:3}: {}'.format(i, word))
166
167
168 # --------------------------------------------------
169 if __name__ == '__main__':
170     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
```

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: 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.