20 Password strength: Generating a secure and memorable password

It's not easy to create passwords that are both difficult to guess and easy to remember. An XKCD comic describes an algorithm that provides both security and recall by suggesting that a password be composed of "four random common words" (https://xkcd.com/936/). For instance, the comic suggests that the password composed of the words "correct," "horse," "battery," and "staple" would provide "~44 bits of entropy" which would require around 550 years for a computer to guess, given 1,000 guesses per second.

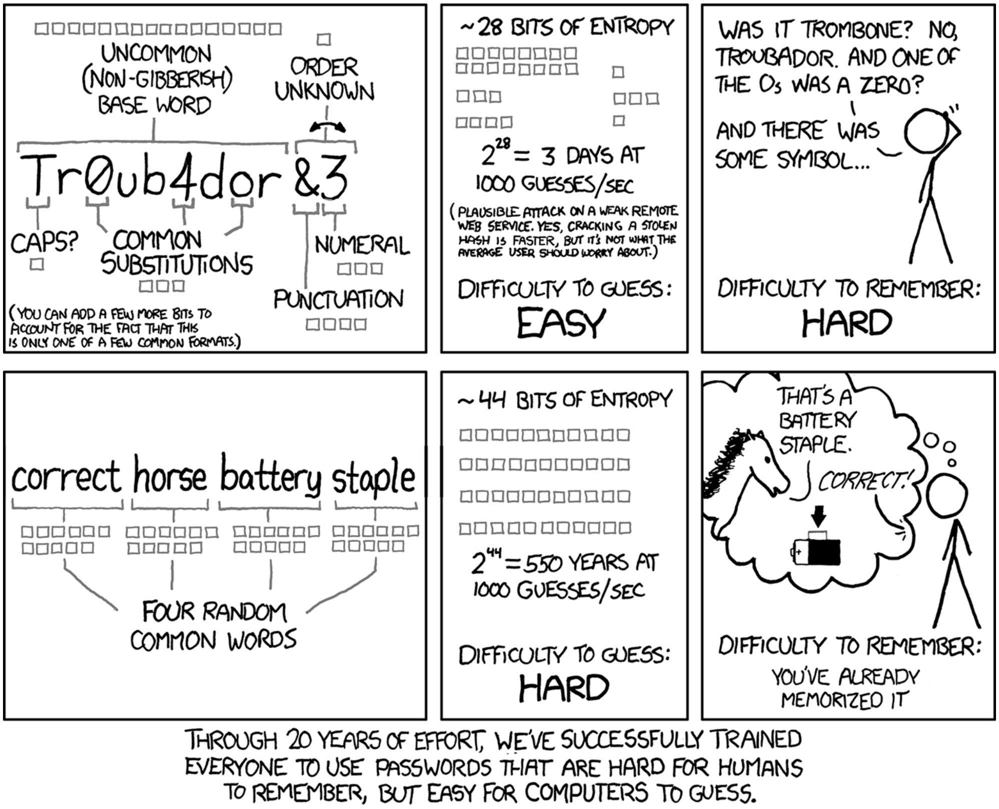
We're going to write a program called password.py that will create passwords by randomly combining words from some input files. Many computers have a file that lists thousands of English words, each on a separate line. On most of my systems, I can find this at /usr/share/dict/words, and it contains over 235,000 words! As the file can vary by system, I've added a version to the repo so that we can use the same file. This file is a little large, so I've compressed to inputs/words.txt.zip. You should unzip it before using it:

```
$ unzip inputs/words.txt.zip
```

Now we should both have the same inputs/words.txt file so that this is reproducible for you:

```
$ ./password.py ../inputs/words.txt --seed 14
CrotalLeavesMeeredLogy
NatalBurrelTizzyOddman
UnbornSignerShodDehort
```

Hmm, maybe those aren't going to be the easiest to remember! Perhaps instead we should be a bit more judicious about the source of our words? We're drawing from


UNCOMMON (NON-GIBBERISH) BASE WORD
ORDER UNKNOWN
Tr0ub4dor&3
CAPS?
COMMON SUBSTITUTIONS
NUMERAL
PUNCTUATION
(YOU CAN ADD A FEW MORE BITS TO ACCOUNT FOR THE FACT THAT THIS IS ONLY ONE OF A FEW COMMON FORMATS.)
~28 BITS OF ENTROPY
2^{28} = 3 DAYS AT 1000 GUESSES/SEC
(PLAUSIBLE ATTACK ON A WEAK REMOTE WEB SERVICE. YES, CRACKING A STOLEN HASH IS FASTER, BUT IT'S NOT WHAT THE AVERAGE USER SHOULD WORRY ABOUT.)
DIFFICULTY TO GUESS: EASY
WAS IT TROMBONE? NO, TROUBADOR. AND ONE OF THE Os WAS A ZERO?
AND THERE WAS SOME SYMBOL...
DIFFICULTY TO REMEMBER: HARD
correct horse battery staple
FOUR RANDOM COMMON WORDS
~44 BITS OF ENTROPY
2^{44}=550 YEARS AT 1000 GUESSES/SEC
DIFFICULTY TO GUESS: HARD
THAT'S A BATTERY STAPLE.
CORRECT!
DIFFICULTY TO REMEMBER: YOU'VE ALREADY MEMORIZED IT
THROUGH 20 YEARS OF EFFORT, WE'VE SUCCESSFULLY TRAINED EVERYONE TO USE PASSWORDS THAT ARE HARD FOR HUMANS TO REMEMBER, BUT EASY FOR COMPUTERS TO GUESS.

(Image used with permission from xkcd.com.)

a pool of over 200,000 words, but the average speaker tends to use somewhere between 20,000 and 40,000 words.

We can generate more memorable passwords by drawing from an actual piece of English text, such as the US Constitution. Note that to use a piece of input text in this way, we will need to remove any punctuation, as we have done in previous exercises:

```
$ ./password.py --seed 8 ../inputs/const.txt
DulyHasHeadsCases
DebtSevenAnswerBest
ChosenEmitTitleMost
```

Another strategy for generating memorable words could be to limit the pool of words to the more interesting parts of speech, like nouns, verbs, and adjectives taken from texts like novels or poetry. I've included a program I wrote called harvest.py that uses a natural language processing library in Python called spaCy (https://spacy.io) that will extract those parts of speech into files that we can use as input to our program. If you want to use this program on your own input files, you'll need to be sure you first install the module:

```
$ python3 -m pip install spacy
```

I ran the harvest.py program on some texts and placed the outputs into directories in the 20_password directory of the source repo. For instance, here is the output drawing from nouns found in the US Constitution:

```
$ ./password.py --seed 5 const/nouns.txt
TaxFourthYearList
TrialYearThingPerson
AidOrdainFifthThing
```

And here we have passwords generated using only verbs found in *The Scarlet Letter* by Nathaniel Hawthorne:

```
$ ./password.py --seed 1 scarlet/verbs.txt
CrySpeakBringHold
CouldSeeReplyRun
WearMeanGazeCast
```

And here are some generated from adjectives extracted from William Shakespeare's sonnets:

```
$ ./password.py --seed 2 sonnets/adjs.txt
BoldCostlyColdPale
FineMaskedKeenGreen
BarrenWiltFemaleSeldom
```

Just in case that does not result in a strong enough password, we will also provide a `--l33t` flag to further obfuscate the text by

1. Passing the generated password through the ransom.py algorithm from chapter 12
2. Substituting various characters with a given table, as we did in jump_the_five.py from chapter 4
3. Adding a randomly selected punctuation character to the end

Here is what the Shakespearean passwords look like with this encoding:

```
$ ./password.py --seed 2 sonnets/adjs.txt --l33t
B0LDco5TLYColdp@l3,
f1n3M45K3dK3eNGR33N[
B4rReNW1LTFeM4l3seldoM/
```

In this exercise, you will

- Take a list of one or more input files as positional arguments
- Use a regular expression to remove non-word characters
- Filter words by some minimum length requirement
- Use sets to create unique lists

- Generate a given number of passwords by combining some given number of randomly selected words
- Optionally encode text using a combination of algorithms we've previously written

20.1 Writing password.py

Our program should be written in the 20_password directory and will be called password.py. It will create some `--num` number of passwords (default, `3`) each by randomly choosing some `--num_words` number of words (default, `4`) from a unique set of words from one or more input files. As it will use the `random` module, the program will also accept a random `--seed` argument, which should be an integer value with a default of `None`. The words from the input files will need to be a `--min_word_len` minimum length (default, `3`) up to a `--max_word_len` maximum length (default, `6`) after removing any non-characters.

As always, our first priority is to sort out the inputs to the program. Do not move ahead until your program can produce this usage with the `-h` or `--help` flags and can pass the first eight tests:

```
$ ./password.py -h
usage: password.py [-h] [-n num_passwords] [-w num_words] [-m minimum]
                   [-x maximum] [-s seed] [-l]
                   FILE [FILE ...]

Password maker

positional arguments:
  FILE                  Input file(s)

optional arguments:
  -h, --help            show this help message and exit
  -n num_passwords, --num num_passwords
                        Number of passwords to generate (default: 3)
  -w num_words, --num_words num_words
                        Number of words to use for password (default: 4)
  -m minimum, --min_word_len minimum
                        Minimum word length (default: 3)
  -x maximum, --max_word_len maximum
                        Maximum word length (default: 6)
  -s seed, --seed seed  Random seed (default: None)
  -l, --l33t            Obfuscate letters (default: False)
```

The words from the input files will be title cased (first letter uppercase, the rest lowercase), which we can achieve using the `str.title()` method. This makes it easier to see and remember the individual words in the output. Note that we can vary the number of words included in each password as well as the number of passwords generated:

```
$ ./password.py --num 2 --num_words 3 --seed 9 sonnets/*
QueenThenceMasked
GullDeemdEven
```

The `--min_word_len` argument helps to filter out shorter, less interesting words like "a," "I," "an," "of," and so on, while the `--max_word_len` argument prevents the passwords from becoming unbearably long. If you increase these values, the passwords change quite drastically:

```
$ ./password.py -n 2 -w 3 -s 9 -m 10 -x 20 sonnets/*
PerspectiveSuccessionIntelligence
DistillationConscienceCountenance
```

The `--l33t` flag is a nod to "leet"-speak, where `31337 H4X0R` means "ELITE HACKER".[1] When this flag is present, we'll encode each of the passwords in two ways. First, we'll pass the word through the `ransom()` algorithm we wrote in chapter 12:

```
$ ./ransom.py MessengerRevolutionImportune
MesSENGeRReVolUtIonImpoRtune
```

Then, we'll use the following substitution table to substitute characters in the same way we did in chapter 4:

```
a => @
A => 4
O => 0
t => +
E => 3
I => 1
S => 5
```

To cap it off, we'll use `random.choice()` to select one character from `string.punctuation` to add to the end:

```
$ ./password.py --num 2 --num_words 3 --seed 9 --min_word_len 10 --max_word_len
     20 sonnets/* --l33t
p3RsPeC+1Vesucces5i0niN+3lL1Genc3$
D1s+iLl@+ioNconsc1eNc3coun+eN@Nce^
```

Figure 20.1 shows a string diagram that summarizes the inputs.

20.1.1 Creating a unique list of words

Let's start off by making our program print the name of each input file:

```
def main():
    args = get_args()
    random.seed(args.seed)

    for fh in args.file:
        print(fh.name)
```

Always set random.seed() right away as it will globally affect all actions by the random module.

Iterate through the file arguments.

Print the name of the file.

[1] See the "Leet" Wikipedia page (https://en.wikipedia.org/wiki/Leet) or the Cryptii translator https://cryptii.com/.

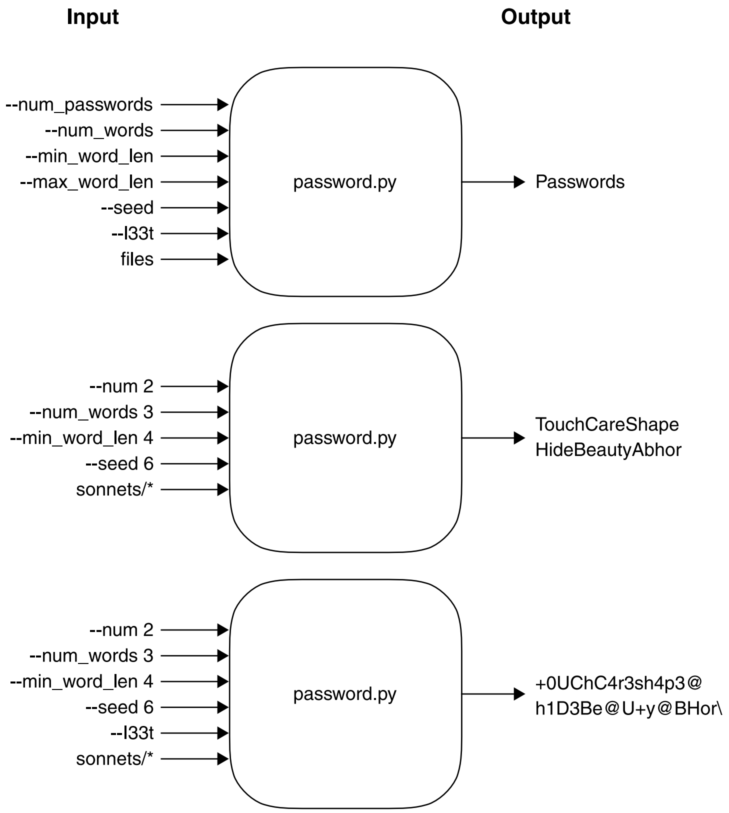


Figure 20.1 Our program has many possible options but requires only one or more input files. The output will be unbreakable passwords.

Let's test it with the words.txt file:

```
$ ./password.py ../inputs/words.txt
../inputs/words.txt
```

Now let's try it with some of the other inputs:

```
$ ./password.py scarlet/*
scarlet/adjs.txt
scarlet/nouns.txt
scarlet/verbs.txt
```

Our first goal is to create a unique list of words we can use for sampling. So far we've used lists to keep ordered collections of things like strings and numbers. The elements in a `list` do not have to be *unique*, though. We've also used dictionaries to create key/value pairs, and the keys of a dictionary *are* unique. Since we don't care about the values, we could set each key of a dictionary equal to some arbitrary value, like `1`:

```
def main():
    args = get_args()
    random.seed(args.seed)
    words = {}

    for fh in args.file:
        for line in fh:
            for word in line.lower().split():
                words[word] = 1

    print(words)
```

Create an empty dict to hold the unique words.

Iterate through the files.

Iterate through the lines of the file.

Lowercase the line and split it on spaces into words.

Set the key words[word] equal to 1 to indicate we saw it. We're only using a dict to get the unique keys. We don't care about the values, so you could use whatever value you like.

If you run this on the US Constitution, you should see a fairly large list of words (some output elided here):

```
$ ./password.py ../inputs/const.txt
{'we': 1, 'the': 1, 'people': 1, 'of': 1, 'united': 1, 'states,': 1, ...}
```

I can spot one problem, in that the word `'states,'` has a comma attached to it. If we try in the REPL with the first bit of text from the Constitution, we can see the problem:

```
>>> 'We the People of the United States,'.lower().split()
['we', 'the', 'people', 'of', 'the', 'united', 'states,']
```

How can we get rid of the punctuation?

20.1.2 Cleaning the text

We've seen several times that splitting on spaces leaves punctuation, but splitting on non-word characters can break contracted words like "Don't" in two. We'd like a function that will `clean()` a word.

First let's imagine the test for it. Note that in this exercise, I'll put all my unit tests into a file called unit.py, which I can run with `pytest -xv unit.py`.

Here is the test for our `clean()` function:

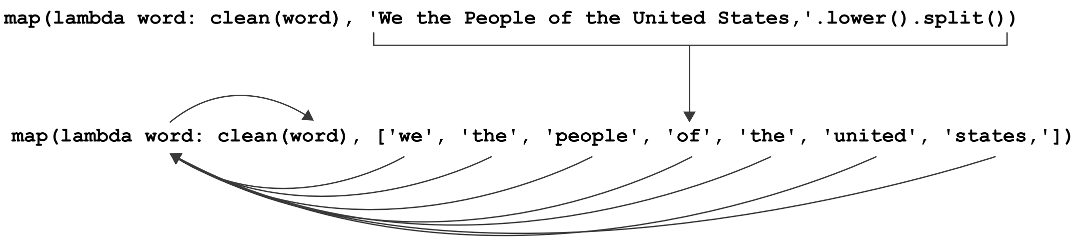
```
def test_clean():
    assert clean('') == ''
    assert clean("states,") == 'states'
    assert clean("Don't") == 'Dont'
```

It's always good to test your functions on nothing, just to make sure it does something sane.

The function should remove punctuation at the end of a string.

The function should not split a contracted word in two.

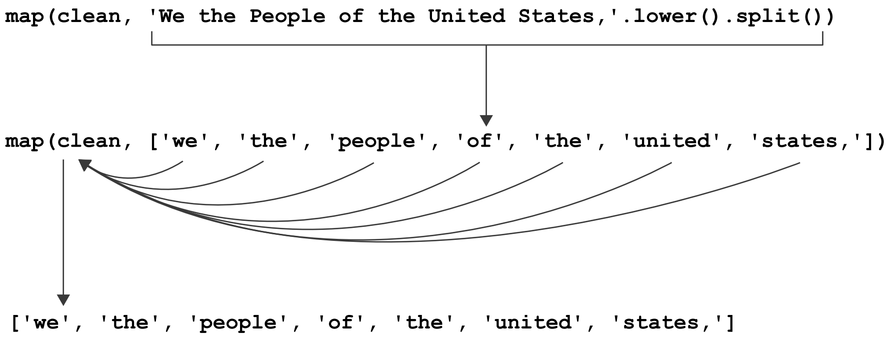
I would like to apply this to all the elements returned by splitting each line into words, and `map()` is a fine way to do that. We often use a `lambda` when writing `map()`, as in figure 20.2.



```
map(lambda word: clean(word), 'We the People of the United States,'.lower().split())

map(lambda word: clean(word), ['we', 'the', 'people', 'of', 'the', 'united', 'states,'])
```

Figure 20.2 Writing `map()` using a `lambda` to accept each word from splitting a string

We don't actually need to write a `lambda` for `map()` here because the `clean()` function expects a single argument, as shown in figure 20.3.



```
map(clean, 'We the People of the United States,'.lower().split())

map(clean, ['we', 'the', 'people', 'of', 'the', 'united', 'states,'])

['we', 'the', 'people', 'of', 'the', 'united', 'states,']
```

Figure 20.3 Writing the `map()` without the `lambda` because the function expects a single value

See how it integrates with the code:

```
def main():
    args = get_args()
    random.seed(args.seed)
    words = {}

    for fh in args.file:
        for line in fh:
            for word in map(clean, line.lower().split()):
                words[word] = 1

    print(words)
```

Use map() to apply the clean() function to the results of splitting the line on spaces. No lambda is required because clean() expects a single argument.

If we run that on the US Constitution again, we can see that `'states'` has been fixed:

```
$ ./password.py ../inputs/const.txt
{'we': 1, 'the': 1, 'people': 1, 'of': 1, 'united': 1, 'states': 1, ...}
```

I'll leave it to you to write a `clean()` function that will satisfy that test. You might use a list comprehension, a `filter()`, or maybe a regular expression. The choice is yours, so long as it passes the test.

20.1.3 Using a set

There is a better data structure than a `dict` to use for our purposes here. It's called a `set`, and you can think of it as being like a unique `list` or just the keys of a `dict`. Here is how we could change our code to use a `set` to keep track of *unique* words:

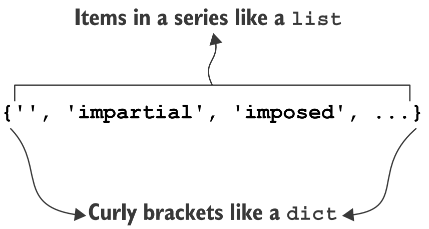
```
def main():
    args = get_args()
    random.seed(args.seed)
    words = set()

    for fh in args.file:
        for line in fh:
            for word in map(clean, line.lower().split()):
                words.add(word)

    print(words)
```

Use the set() function to create an empty set.

Use set.add() to add a value to a set.

If you run this code now, you will see slightly different output, where Python shows you a data structure in curly brackets (`{}`) that will make you think of a `dict`, but you'll notice that the contents look more like a `list` (as pointed out in figure 20.4):

```
$ ./password.py ../inputs/const.txt
{'', 'impartial', 'imposed', 'jared', 'levying', ...}
```



```
{'', 'impartial', 'imposed', ...}
```

Figure 20.4 A set looks like a cross between a dictionary and a list.

We're using sets here because they so easily allow us to keep a unique list of words, but sets are much more powerful than this. For instance, you can find the shared values between two lists by using `set.intersection()`:

```
>>> nums1 = set(range(1, 10))
>>> nums2 = set(range(5, 15))
>>> nums1.intersection(nums2)
{5, 6, 7, 8, 9}
```

You can read `help(set)` in the REPL or in the documentation online to learn about all the amazing things you can do with sets.

20.1.4 Filtering the words

If we look again at the output we have, we'll see that the empty string is the first element:

```
$ ./password.py ../inputs/const.txt
{'', 'impartial', 'imposed', 'jared', 'levying', ...}
```

We need a way to filter out unwanted values like strings that are too short. In chapter 14 we looked at the `filter()` function, which is a higher-order function that takes two arguments:

- A function that accepts one element and returns `True` if the element should be kept or `False` if the element should be excluded
- Some "iterable" (like a `list` or `map()`) that produces a sequence of elements to be filtered

In our case, we want to accept only words that have a length greater than or equal to the `--min_word_len` argument, and less than or equal to `--max_word_len`. In the REPL, we can use a `lambda` to create an anonymous function that accepts a `word` and makes these comparisons. The result of that comparison is either `True` or `False`. Only words with a length from 3 to 6 are allowed, so this has the effect of removing short, uninteresting words. Remember that `filter()` is lazy, so I have to coerce it using the `list` function in the REPL to see the output:

```
>>> shorter = ['', 'a', 'an', 'the', 'this']
>>> min_word_len = 3
>>> max_word_len = 6
>>> list(filter(lambda word: min_word_len <= len(word) <= max_word_len, shorter))
['the', 'this']
```

This `filter()` will also remove longer words that would make our passwords cumbersome:

```
>>> longer = ['that', 'other', 'egalitarian', 'disequilibrium']
>>> list(filter(lambda word: min_word_len <= len(word) <= max_word_len, longer))
['that', 'other']
```

One way we could incorporate the `filter()` is to create a `word_len()` function that encapsulates the preceding `lambda`. Note that I defined it inside `main()` in order to create a *closure*, because I want to reference the values of `args.min_word_len` and `args.max_word_len`:

```
def main():
    args = get_args()
    random.seed(args.seed)
    words = set()

    def word_len(word):
        return args.min_word_len <= len(word) <= args.max_word_len
```

This function will return True if the length of the given word is in the allowed range.

```
    for fh in args.file:
        for line in fh:
            for word in filter(word_len, map(clean, line.lower().split())):
                words.add(word)

    print(words)
```

We can use word_len (without the parentheses!) as the function argument to filter().

We can again try our program to see what it produces:

```
$ ./password.py ../inputs/const.txt
{'measures', 'richard', 'deprived', 'equal', ...}
```

Try it on multiple inputs, such as all the nouns, adjectives, and verbs from *The Scarlet Letter*:

```
$ ./password.py scarlet/*
{'walk', 'lose', 'could', 'law', ...}
```

20.1.5 Titlecasing the words

We used the `line.lower()` function to lowercase all the input, but the passwords we generate will need each word to be in "Title Case," where the first letter is uppercase and the rest of the word is lowercase. Can you figure out how to change the program to produce this output?

```
$ ./password.py scarlet/*
{'Dark', 'Sinful', 'Life', 'Native', ...}
```

Now we have a way to process any number of files to produce a unique list of title-cased words that have non-word characters removed and have been filtered to remove the ones that are too short or long. That's quite a lot of power packed into a few lines of code!

20.1.6 Sampling and making a password

We're going to use the `random.sample()` function to randomly choose `--num` number of words from our `set` to create an unbreakable, yet memorable, password. We've talked before about the importance of using a random seed to test that our "random" selections are reproducible. It's also quite important that the items from which we sample always be ordered in the same way so that the same selections are made. If we use the `sorted()` function on a `set`, we get back a sorted `list`, which is perfect for using with `random.sample()`.

We can add this line to the code from before:

```
words = sorted(words)
print(random.sample(words, args.num_words))
```

Now when I run the program with *The Scarlet Letter* input, I will get a list of words that might make an interesting password:

```
$ ./password.py scarlet/*
['Lose', 'Figure', 'Heart', 'Bad']
```

The result of `random.sample()` is a `list` that you can join on the empty string in order to make a new password:

```
>>> ''.join(random.sample(words, num_words))
'TokenBeholdMarketBegin'
```

You will need to create the number of passwords indicated by the user, similar to how we created some number of insults in chapter 9. How will you do that?

20.1.7 l33t-ify

The last piece of our program involves creating an `l33t()` function that will obfuscate the password. The first step is to convert the password with the same algorithm we wrote for ransom.py. I'm going to create a `ransom()` function for this, and here is the test that is in unit.py:

```
def test_ransom():
    state = random.getstate()
    random.seed(1)
    assert ransom('Money') == 'moNeY'
    assert ransom('Dollars') == 'DOLlaRs'
    random.setstate(state)
```

Save the current global state.

Set random.seed() to a known value for the test.

Restore the state.

I'll leave it to you to create the function that satisfies this test.

> **NOTE** You can run `pytest -xv unit.py` to run the unit tests. The program will import the various functions from your password.py file to test. Open unit.py and inspect it to understand how this happens.

Next I will replace some of the characters according to the following table. I recommend you revisit chapter 4 to see how you did that:

```
a => @
A => 4
O => 0
t => +
E => 3
I => 1
S => 5
```

I wrote an `l33t()` function that combines `ransom()` with the preceding substitution and then adds a punctuation character by appending `random.choice(string.punctuation)`.

Here is the `test_l33t()` function you can use to write your function. It works almost identically to the previous test, so I shall eschew commentary:

```
def test_l33t():
    state = random.getstate()
    random.seed(1)
    assert l33t('Money') == 'moNeY{'
    assert l33t('Dollars') == 'D0ll4r5`'
    random.setstate(state)
```

20.1.8 Putting it all together

Without giving away the ending, I'd like to say that you need to be *really careful* about the order of operations that include the `random` module. My first implementation would print different passwords given the same seed when I used the `--l33t` flag. Here was the output for plain passwords:

```
$ ./password.py -s 1 -w 2 sonnets/*
EagerCarcanet
LilyDial
WantTempest
```

I would have expected the *exact same passwords*, only encoded. Here is what my program produced instead:

```
$ ./password.py -s 1 -w 2 sonnets/* --l33t
3@G3RC@rC@N3+{
m4dnes5iNcoN5+4n+|
MouTh45s15T4nCe^
```

The first password looks OK, but what are those other two? I modified my code to print both the original password and the l33ted one:

```
$ ./password.py -s 1 -w 2 sonnets/* --l33t
3@G3RC@rC@N3+{ (EagerCarcanet)
m4dnes5iNcoN5+4n+| (MadnessInconstant)
MouTh45s15T4nCe^ (MouthAssistance)
```

The `random` module uses a global state to make each of its "random" choices. In my first implementation, I was modifying this state after choosing the first password by immediately modifying the new password with the `l33t()` function. Because the `l33t()` function also uses `random` functions, the state was altered for the next password. My solution was to first generate *all* the passwords and then alter them using the `l33t()` function, if necessary.

Those are all the pieces you should need to write your program. You have the unit tests to help you verify the functions, and you have the integration tests to ensure your program works as a whole.

20.2 Solution

I hope you will use your program to generate your passwords. Be sure to share them with your author, especially the ones to your bank account and favorite shopping sites!

```
#!/usr/bin/env python3
"""Password maker, https://xkcd.com/936/"""

import argparse
import random
import re
import string
```

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

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

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

    parser.add_argument('-n',
                        '--num',
                        metavar='num_passwords',
                        type=int,
                        default=3,
                        help='Number of passwords to generate')

    parser.add_argument('-w',
                        '--num_words',
                        metavar='num_words',
                        type=int,
                        default=4,
                        help='Number of words to use for password')

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

    parser.add_argument('-x',
                        '--max_word_len',
                        metavar='maximum',
                        type=int,
                        default=6,
                        help='Maximum word length')

    parser.add_argument('-s',
                        '--seed',
                        metavar='seed',
                        type=int,
                        help='Random seed')

    parser.add_argument('-l',
                        '--l33t',
                        action='store_true',
                        help='Obfuscate letters')

    return parser.parse_args()
```

Set the random.seed() to the given value or the default None, which is the same as not setting the seed.

Iterate through each word generated by splitting the lowercased line on spaces, removing non-word characters with the clean() function, and filtering for words of an acceptable length.

Create an empty set to hold all the unique words we'll extract from the texts.

Create a word_len() function for filter() that returns True if the word's length is in the allowed range and False otherwise.

Iterate through each open file handle.

Iterate through each line of text in the file handle.

Title-case the word before adding it to the set.

```
# --------------------------------------------------
def main():
    args = get_args()
    random.seed(args.seed)
    words = set()

    def word_len(word):
        return args.min_word_len <= len(word) <= args.max_word_len

    for fh in args.file:
        for line in fh:
            for word in filter(word_len, map(clean, line.lower().split())):
                words.add(word.title())

    words = sorted(words)
    passwords = [
        ''.join(random.sample(words, args.num_words)) for _ in range(args.num)
    ]

    if args.l33t:
        passwords = map(l33t, passwords)

    print('\n'.join(passwords))
```

See if the args.l33t flag is True.

Use map() to run all the passwords through the l33t() function to produce a new list of passwords. It's safe to call the l33t() function here. If we had used the function in the list comprehension, it would have altered the global state of the random module, thereby altering the following passwords.

Use a list comprehension with a range to create the correct number of passwords. Since I don't need the actual value from range, I can use _ to ignore the value.

Print the passwords joined on newlines.

Use the sorted() function to order words into a new list.

Define a function to clean() a word.

Use a regular expression to substitute the empty string for anything that is not an English alphabet character.

Define a function to l33t() a word.

```
# --------------------------------------------------
def clean(word):
    """Remove non-word characters from word"""

    return re.sub('[^a-zA-Z]', '', word)


# --------------------------------------------------
def l33t(text):
    """l33t"""

    text = ransom(text)
    xform = str.maketrans({
        'a': '@', 'A': '4', 'O': '0', 't': '+', 'E': '3', 'I': '1', 'S': '5'
    })
    return text.translate(xform) + random.choice(string.punctuation)
```

Use the ransom() function to randomly capitalize letters.

Make a translation table/dict for character substitutions.

Use the str.translate() function to perform the substitutions and append a random piece of punctuation.

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

    return ''.join(
        map(lambda c: c.upper() if random.choice([0, 1]) else c.lower(), text))


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

Define a function for the ransom() algorithm from chapter 12.

Return a new string created by randomly upper- or lowercasing each letter in a word.

20.3 Discussion

I hope you found this program challenging and interesting. There wasn't anything new in `get_args()`, but, again, about half the lines of code are found just in this function. I feel this is indicative of just how important it is to correctly define and validate the inputs to a program!

Now, let's get on with talking about the auxiliary functions.

20.3.1 Cleaning the text

I chose to use a regular expression to remove any characters that are outside the set of lower- and uppercase English characters:

```
def clean(word):
    """Remove non-word characters from word"""
    return re.sub('[^a-zA-Z]', '', word)
```

The re.sub() function will substitute any text matching the pattern (the first argument) found in the given text (the third argument) with the value given by the second argument.

Recall from chapter 18 that we can write the character class `[a-zA-Z]` to define the characters in the ASCII table bounded by those two ranges. We can then *negate* or complement that class by placing a caret (^) as the *first character* inside that class, so `[^a-zA-Z]` can be read as "any character not matching a to z or A to Z."

It's perhaps easier to see it in action in the REPL. In the following example, only the letters "AbCd" will be left from the text "A1b*C!d4":

```
>>> import re
>>> re.sub('[^a-zA-Z]', '', 'A1b*C!d4')
'AbCd'
```

If the only goal were to match ASCII letters, it would be possible to solve it by looking for membership in `string.ascii_letters`:

```
>>> import string
>>> text = 'A1b*C!d4'
>>> [c for c in text if c in string.ascii_letters]
['A', 'b', 'C', 'd']
```

A list comprehension with a guard can also be written using `filter()`:

```
>>> list(filter(lambda c: c in string.ascii_letters, text))
['A', 'b', 'C', 'd']
```

Both of the non-regex versions seem like more effort to me. Additionally, if the function ever needed to be changed to allow, say, numbers and a few specific pieces of punctuation, the regular expression version becomes significantly easier to write and maintain.

20.3.2 A king's ransom

The `ransom()` function was taken straight from the ransom.py program in chapter 12, so there isn't too much to say about it except, hey, look how far we've come! What was the idea for an entire chapter is now a single line in a much longer and more complicated program:

```
def ransom(text):
    """Randomly choose an upper or lowercase letter to return"""
    return ''.join(
        map(lambda c: c.upper() if random.choice([0, 1]) else c.lower(), text))
```

Join the resulting list from the map() on the empty string to create a new string.

Use map() to iterate through each character in the text and select either the upper- or lowercase version of the character based on a "coin toss," using random.choice() to select between a "truthy" value (1) or a "falsey" value (0).

20.3.3 How to l33t()

The `l33t()` function builds on `ransom()` and then adds a text substitution that is straight out of chapter 4. I like the `str.translate()` version of that program, so I used it again here:

```
def l33t(text):
    """l33t"""
    text = ransom(text)
    xform = str.maketrans({
        'a': '@', 'A': '4', 'O': '0', 't': '+', 'E': '3', 'I': '1', 'S': '5'
    })
    return text.translate(xform) + random.choice(string.punctuation)
```

Randomly capitalize the given text.

Make a translation table from the given dict that describes how to modify one character to another. Any characters not listed in the keys of this dict will be ignored.

Use the str.translate() method to make all the character substitutions. Use random.choice() to select one additional character from string.punctuation to append to the end.

20.3.4 Processing the files

To use these functions, we need to create a unique set of all the words in our input files. I wrote this bit of code with an eye both on performance and on style:

```
words = set()
for fh in args.file:
    for line in fh:
```

Iterate through each open file handle.

Read the file handle line by line with a for loop, not with a method like fh.read(), which will read the entire contents of the file at once.

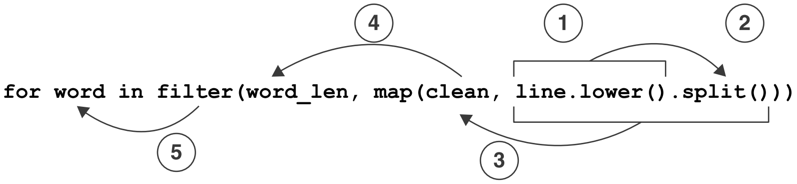
```
        for word in filter(word_len, map(clean, line.lower().split())):
            words.add(word.title())
```

Title-case the word before adding it to the set.

Reading this code requires starting at the end where I split line.lower() on spaces. Each word from str.split() goes into clean(), which then must pass through the filter() function.

Figure 20.5 shows a diagram of that `for` line.

1. `line.lower()` will return a lowercase version of `line`.
2. The `str.split()` method will break the text on whitespace to return words.
3. Each word is fed into the `clean()` function to remove any character that is not in the English alphabet.
4. The cleaned words are filtered by the `word_len()` function.
5. The resulting `word` has been transformed, cleaned, and filtered.



4
1
2
for word in filter(word_len, map(clean, line.lower().split()))
5
3

Figure 20.5 A visualization of the order of operations for the various functions

If you don't like the `map()` and `filter()` functions, you might rewrite the code like so:

```
words = set()
for fh in args.file:
    for line in fh:
        for word in line.lower().split():
            word = map(clean)
            if args.min_word_len <= len(word) <= args.max_word_len:
                words.add(word.title()
```

Iterate through each open file handle.

Iterate through each line of the file handle.

Iterate through each "word" from splitting the lowercased line on spaces.

Remove unwanted characters.

Check if the word is an acceptable length.

Add the title-cased word to the set.

However you choose to process the files, at this point you should have a complete `set` of all the unique, title-cased words from the input files.

20.3.5 Sampling and creating the passwords

As noted earlier, it's vital to sort the `words` for our tests so that we can verify that we are making consistent choices. If you only wanted random choices and didn't care about testing, you would not need to worry about sorting—but then you'd also be a morally deficient person for not testing, so perish the thought! I chose to use the `sorted()` function, as there is no other way to sort a `set`:

```
words = sorted(words)
```

There is no set.sort() function. Sets are ordered internally by Python. Calling sorted() on a set will create a new, sorted list.

We need to create a given number of passwords, and I thought it might be easiest to use a `for` loop with a `range()`. In my code, I used `for _ in range(…)` just as in chapter 9 because I don't need to know the value each time through the loop. The underscore (`_`) is a way to indicate that you are ignoring the value. It's fine to say `for i in range(…)` if you want, but some linters might complain if they see that your code declares the variable `i` but never uses it. That could legitimately be a bug, so it's best to use the `_` to show that you mean to ignore this value.

Here is the first way I wrote the code that led to the bug I mentioned earlier, where different passwords would be chosen even when I used the same random seed. Can you spot the bug?

Iterate through the args.num of passwords to create.

Each password will be based on a random sampling from words, and I will choose the value given in args.num_words. The random.sample() function returns a list of words that I str.join() on the empty string to create a new string.

```
for _ in range(args.num):
    password = ''.join(random.sample(words, args.num_words))
    print(l33t(password) if args.l33t else password)
```

If the args.l33t flag is True, we'll print the l33t version of the password; otherwise, I'll print the password as is. This is the bug! Calling l33t() here modifies the global state used by the random module, so the next time I call random.sample(), I get a different sample.

The solution is to separate the concerns of generating the passwords and possibly modifying them:

Use a list comprehension to iterate through range(args.num) to generate the correct number of passwords.

```
passwords = [
    ''.join(random.sample(words, args.num_words)) for _ in range(args.num)
]

if args.l33t:
    passwords = map(l33t, passwords)

print('\n'.join(passwords))
```

If the args.leet flag is True, use the l33t() function to modify the passwords.

Print the passwords joined on newlines.

20.4 Going further

- The substitution part of the `l33t()` function changes every available character, which perhaps makes the password too difficult to remember. It would be better to modify only maybe 10% of the password, much like how we changed the input strings in chapter 10's Telephone exercise.

- Create programs that combine other skills you've learned. Like maybe a lyrics generator that randomly selects lines from files of songs by your favorite bands, then encodes the text as in chapter 15, then changes all the vowels to one vowel as in chapter 8, and then SHOUTS IT OUT as in chapter 5?

Summary

- A `set` is a unique collection of values. Sets can interact with other sets to create differences, intersections, unions, and more.
- Changing the order of operations using the `random` module can change the output of a program because the global state of the `random` module may be affected.
- Short, tested functions can be composed to create more complicated, tested programs. Here we combined many ideas from previous exercises in concise, powerful expressions.