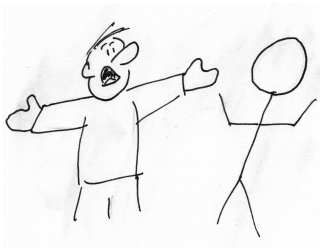
1. Password Strength: Generating a secure and memorable password



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

We're going to write a program called `password.py` that might generate these passwords:

```
$ ./password.py --seed 9
NecrotomicRagefullyIsovalineFlecnode
CitrangeDecentWabenoShamefulness
AcrasialesHyphodromeOutkickNegrotic
```

Well, OK, 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. The passwords above were created from the default word dictionary `/usr/share/dict/words` on my system (which I've included in the GitHub repo as `inputs/words.zip`). This dictionary lists over 235,000 words from the English language. The average speaker, however, tends to use a small fraction of that, somewhere between 20,000 and 40,000 words.

We can generate more memorable words by drawing from some actual piece of English text such as the US Constitution. Note that to use a piece of input text in this way, we will need to remove any punctuation as we have done in several previous exercises. We will also ignore shorter words with fewer than 4 characters:

```
$ ./password.py --seed 2 ../inputs/const.txt
BloodConcurCommencedProduce
ErectedNorthInvasionGold
ArticlesElectorsSeptemberRelating
```

Another strategy for generating memorable words could be to limit the pool of words to more interesting parts of speech like nouns, verbs, and adjectives taken from texts like novels or poetry. I've included a program I wrote called `harvest.py` that uses a Natural Language Processing library in Python called "spaCy" (https://spacy.io) that will extract those parts of speech into files that we can use as input to our program. I ran the `harvest.py` program on some texts and placed the outputs into directories in the GitHub repo.

Here is the output from just the nouns, adjectives, and verbs from the US Consitution:

```
$ ./password.py --seed 2 const/*
CaseConvictConstituteProper
ExcludeNumerousJustHappen
AscertainEqualServiceRegulation
```

Here we have passwords generated from *The Scarlet Letter* by Nathaniel Hawthorne:

```
$ ./password.py --seed 2 scarlet/*
ChurchDrearyDiscernScarlet
HighPeopleMinisterLess
BringHeadTalkSmile
```

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

```
$ ./password.py --seed 2 sonnets/*
BrightCountenanceConsentReek
FlattererOutgoingLeisureHeavenly
BellFaultStraightSend
```

Just in case that is not 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 13
2. Substituting various characters with given table as we did in `jump_the_five.py` from Chapter 5
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/* --l33t
BR1ghTC0UN+en@nc3coNs3NTR33k^
Fl4TT3R3R0U+Go1NGL31sUReHe@venLy'
b3llf4ulT5+r@igh+seND)
```

In this exercise, you will:

- Take an optional list of input files as positional arguments.
- Use a regular expression to remove non-word characters.
- Filter words by some minimum length requirement.
- Use sets to create unique lists.
- Generate some given number of passwords by combining some given number of randomly selected words.
- Optionally encode text using a combination of algorithms we've previously written.

We're getting near the end, so I really wanted to review many of the skills you've used!

1.1. Writing `password.py`

Our program will be called `password.py` and will create some `--num` number of passwords (default `3`) each created by randomly choosing some `--num_words` (default `4`) from a unique set of words from one or more input files (default `/usr/share/dict/words`). As it will use the `random` module, the program will also accept a random `--seed` argument. The words from the input files will need to be a minimum length of some `--min_word_len` (default `4`) after removing any non-characters.

As always, your first priority is to sort out the inputs to your program. Do not move ahead until your program can produce this usage with the `-h` or `--help` flags and can pass the first 7 tests:

```
$ ./password.py -h
usage: password.py [-h] [-n int] [-w int] [-m int] [-s int] [-l]
                   [FILE [FILE ...]]

Password maker

positional arguments:
  FILE                  Input file(s) (default: [<_io.TextIOWrapper
                        name='/usr/share/dict/words' mode='r'
                        encoding='UTF-8'>])

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

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

```
$ ./password.py --num 2 --num_words 3 --seed 9 sonnets/*
TortureRevenueMaintain
EndlessFuryAdieu
```

The `--min_word_len` argument helps to filter out shorter, less interesting words like "a," "an," and "the." If you increase this value, then the passwords change quite drastically:

```
$ ./password.py --num 2 --num_words 3 --seed 9 --min_word_len 8 sonnets/*
MelancholyPropheticImportune
EntertainConquerdDeathdear
```

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, first by passing the word through the `ransom` algorithm we wrote:

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

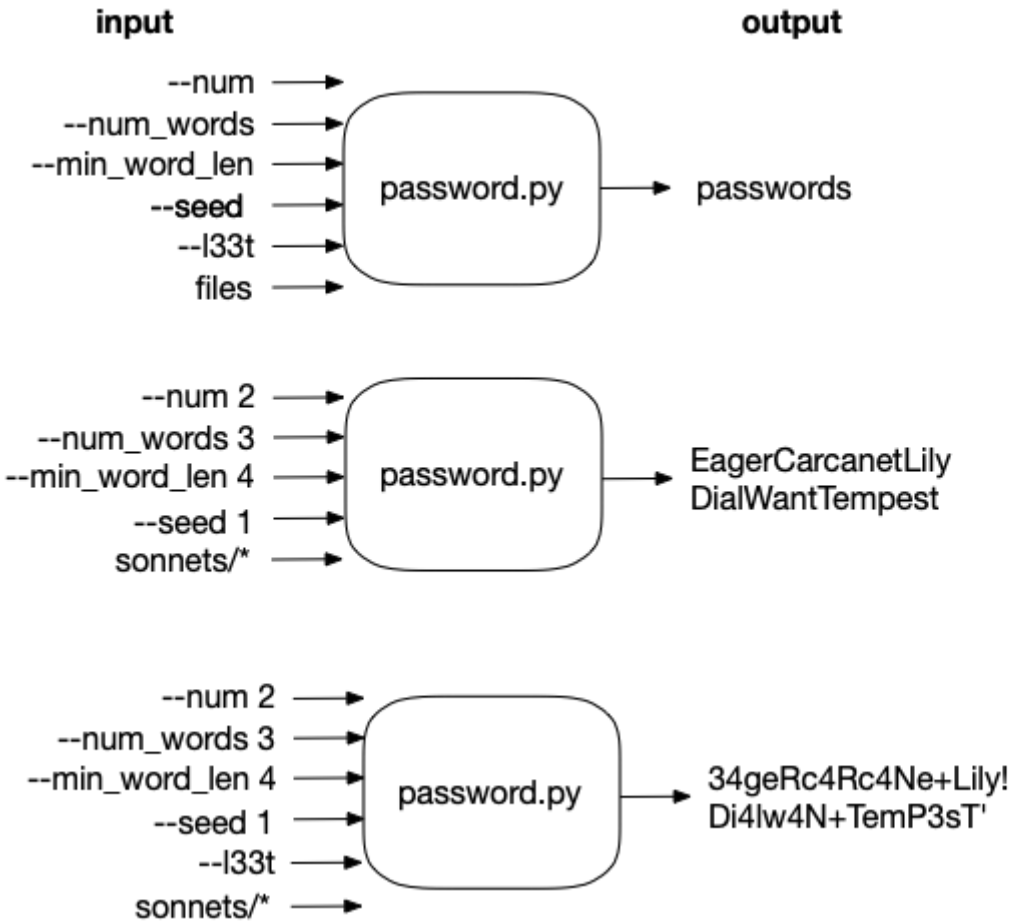
Then we'll use the following substitution table to substitute characters in the same way we did in "Jump the Five":

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

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

```
$ ./password.py --num 2 --num_words 3 --seed 9 --min_word_len 8 --l33t sonnets/*
m3L@NcHoLYprophe+1c1mp0rTuN3@
en+3rT4inc0nquerDde@+hDe4r^
```

Here is the string diagram to summarize the inputs:



1.1.1. Creating a unique list of words

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

```
1 def main():
2     args = get_args()
3     random.seed(args.seed) ①
4
5     for fh in args.file:   ②
6         print(fh.name)     ③
```

① Always set `random.seed` right away as it will globally affect all actions by the `random` module.

② Iterate through the file arguments.

③ Print the name of the file.

We can run it with the default:

```
$ ./password.py
/usr/share/dict/words
```

Or with some of the other inputs:

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

Our first goal is to create a unique list of words we can use for sampling. So far we've used lists to keep ordered collections and dictionaries to create key/value structures. The elements in a `list` do not have to be unique, so we can't use that. The keys of a dictionary *are* unique, however, so that's a possibility:

```
1 def main():
2     args = get_args()
3     random.seed(args.seed)
4     words = {}           ①
5
6     for fh in args.file: ②
7         for line in fh:  ③
8             for word in line.lower().split(): ④
9                 words[word] = 1               ⑤
```

① Create an empty `dict` to hold the words.

② Iterate through the files.

③ Iterate through the lines of the file.

④ Lowercase the line and split it on spaces into words.

⑤ Set the key `words[word]` equal to `1` to indicate we saw it. We're only using a `dict` to get the unique keys. We don't care about the values, so you could use whatever value you like.

If you run this on the US Constitution, you should 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:

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

How can we get rid of punctuation?

1.1.2. Cleaning the text

We've seen several times that splitting on spaces leaves punctuation, but splitting on non-word

characters can break contracted words like "Don't" in two. I'd like to create a function that will `clean` a word. First I'll imagine the test for it. Note that in this exercise, I'll put all my unit tests into a file called `unit.py` which I can run with `pytest -xv unit.py`.

Here is the test for our `clean` function:

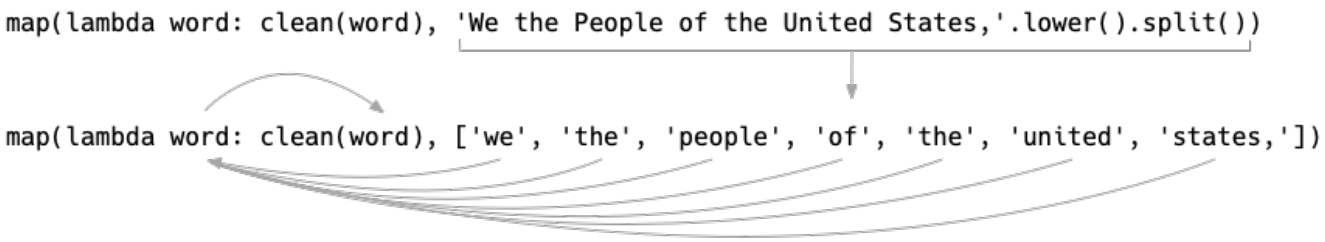
```
1 def test_clean():
2     assert clean('') == ''              ①
3     assert clean("states,") == 'states' ②
4     assert clean("Don't") == 'Dont'     ③
```

① It's always good to test your functions on nothing just to make sure it does something sane.

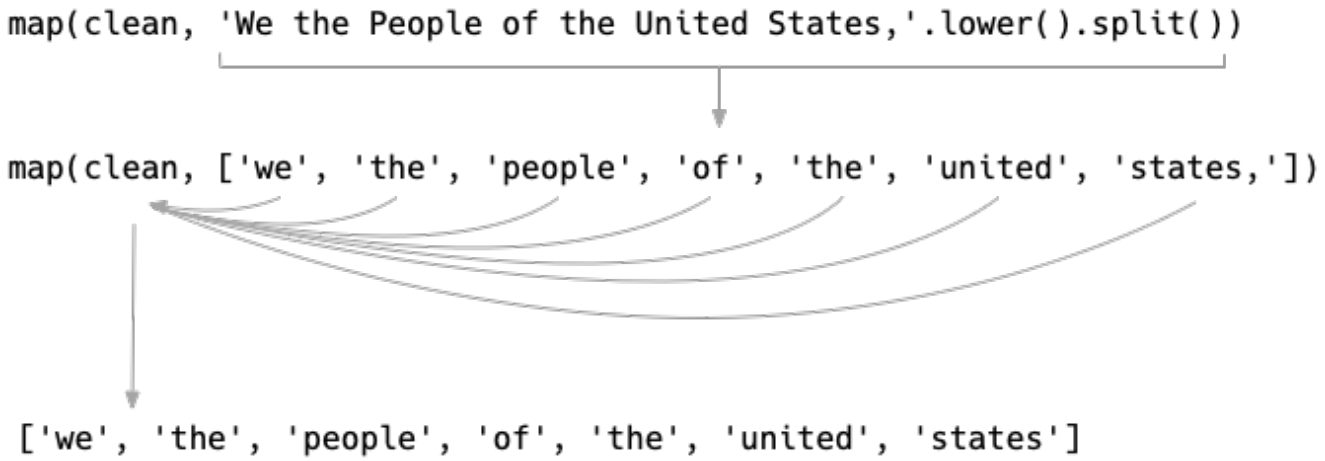
② The function should remove punctuation at the end of a string.

③ The function should not split a contracted word in two.

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



Notice that I do not need to write a `lambda` for the `map` because the `clean` function expects a single argument:



See how it integrates with the code:

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

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

    print(words)
```

① Use `map` to apply the `clean` function to the results of splitting the `line` on spaces. No `lambda` is required because `clean` expects a single argument.

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

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

I'll leave it to you to write the `clean` function that will satisfy that test.

1.1.3. Using a `set`

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

```
def main():
    args = get_args()
    random.seed(args.seed)
    words = set() ①

    for fh in args.file:
        for line in fh:
            for word in map(clean, line.lower().split()):
                words.add(word) ②

    print(words)
```

① Use the `set` function to create an empty set.

② Use `set.add` to add a value to a set.

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

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

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

```
1 >>> nums1 = set(range(1, 10))
2 >>> nums2 = set(range(5, 15))
3 >>> nums1.intersection(nums2)
4 {5, 6, 7, 8, 9}
```

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

1.1.4. Filtering the words

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

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

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

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

In our case, we want to accept only words that have a length greater or equal to the `--min_word_len` argument. In the REPL, I can use a `lambda` to create an anonymous function that accepts a `word` and then compares that word's length to a `min_word_len`. The result of that comparison is either `True` or `False`. Only words with a length of `4` or greater are allowed through, so this has the effect of removing words like them empty string or the English articles. Remember that `filter` is lazy, so I have to coerce it using the `list` function in the REPL to see the output:

```
1 >>> min_word_len = 4
2 >>> list(filter(lambda word: len(word) >= min_word_len, ['', 'a', 'an', 'the',
'that']))
3 ['that']
```

Here is one way we could incorporate that function:

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

    for fh in args.file:
        for line in fh:
            for word in filter(lambda w: len(w) >= args.min_word_len, ①
                               map(clean,
                                   line.lower().split())):
                words.add(word)

    print(words)
```

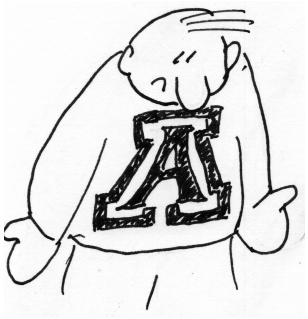
① Only allow words with a length greater or equal to the minimum word length.

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/*
{'professional', 'letter', 'feel', 'gaze', ...}
```



1.1.5. Titlecasing the words

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

```
$ ./password.py scarlet/*
{'Find', 'Evil', 'Professional', 'Person', ...}
```

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.

1.1.6. Sampling and making a password

We're going to use the `random.sample` function to randomly choose some `--num` number of words from our `set` to create an unbreakable yet memorable password. We've talked before about the

importance of using a random seed to test that our "random" selections are reproducible. It's also quite important that the items from which we sample always be ordered in the same way so that the same selections are made. If we use the `sorted` function on a `set`, we get back a sorted `list` which is just perfect for using with `random.sample`:

```
>>> import random
>>> words = sorted(words)
>>> random.sample(words, 4)
['Could', 'Conscious', 'First', 'Prison']
```

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

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

You will need to create `args.num` of passwords. How will you do that?

1.2. l33t-ify

The last piece of our program is to create a `l33t` function that will obfuscate the password. The first step is to convert it with the same algorithm we wrote for `ransom.py`. I'm going to create a `ransom` function for this, and here is the test that is in `unit.py`. I'll leave it to you to create the function that satisfies this test [2]:

```
def test_ransom():
    random.seed(1)                      ①
    assert(ransom('Money') == 'moNeY')
    assert(ransom('Dollars') == 'DOLlaRs')
    random.seed(None)                   ②
```

① Set the `random.seed` to a known value for the test.

② Unset the seed by using the value `None`.

Next I will substitute some of the characters according to the following table. I would recommend you revisit "Jump The Five" to see how you did that:

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

I wrote a `l33t` function that combines the `ransom` with the substitution above and finally adds a punctuation character by appending `random.choice(string.punctuation)`. Here is the `test_l33t` function you can use to write your function:

```
def test_l33t():
    random.seed(1)
    assert (l33t('Money') == 'm0N3Y{')
    assert (l33t('Dollars') == 'D0ll4r5`')
    random.seed(None)
```

1.2.1. Putting it all together

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

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

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

```
$ ./password.py -s 1 -w 2 sonnets/* --l33t
3@G3RC@rC@N3+{
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 implementaion, I was modifying this state after choosing the first password by immediately modifying the new password with the `l33t` function. Because the `l33t` function also uses `random` functions, the state was altered for the next password. The solution was to first generate *all* the passwords and then to `l33t` them, 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. This is the last program, so give it your best shot before looking at the solution!

1.3. Solution

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

import argparse
import random
import re
import string


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

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

    parser.add_argument('file',
                        metavar='FILE',
                        type=argparse.FileType('r'),
                        nargs='*',
                        help='Input file(s)',
                        default=[open('/usr/share/dict/words')])

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

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

    parser.add_argument('-m',
                        '--min_word_len',
                        metavar='int',
                        type=int,
                        default=4,
                        help='Minimum word length')

    parser.add_argument('-s',
                        '--seed',
                        metavar='int',
```

```
49                          type=int,
50                          help='Random seed')
51
52      parser.add_argument('-l',
53                          '--l33t',
54                          action='store_true',
55                          help='Obsfuscate letters')
56
57      return parser.parse_args()
58
59
60  # --------------------------------------------------
61  def main():
62      """Make a jazz noise here"""
63
64      args = get_args()
65      random.seed(args.seed)                                  ①
66      words = set()                                           ②
67
68      for fh in args.file:                                    ③
69          for line in fh:                                     ④
70              for word in filter(lambda w: len(w) >= args.min_word_len, ⑤
71                                 map(clean,
72                                     line.lower().split())):
73                  words.add(word.title())                     ⑥
74
75      words = sorted(words)                                   ⑦
76      passwords = []                                          ⑧
77      for _ in range(args.num):                               ⑨
78          passwords.append(''.join(random.sample(words, args.num_words))) ⑩
79
80      for password in passwords:                              ⑪
81          print(l33t(password) if args.l33t else password)    ⑫
82
83
84  # --------------------------------------------------
85  def clean(word):                                            ⑬
86      """Remove non-word characters from word"""
87
88      return re.sub('[^a-zA-Z]', '', word)                    ⑭
89
90
91  # --------------------------------------------------
92  def l33t(text):                                             ⑮
93      """l33t"""
94
95      text = ransom(text)                                     ⑯
96      xform = str.maketrans({                                 ⑰
97          'a': '@', 'A': '4', 'O': '0', 't': '+', 'E': '3', 'I': '1', 'S': '5'
98      })
99      return text.translate(xform) + random.choice(string.punctuation) ⑱
```

```
100
101
102 # --------------------------------------------------
103 def ransom(text):                                          ⑲
104     """Randomly choose an upper or lowercase letter to return"""
105
106     return ''.join(                                        ⑳
107         map(lambda c: c.upper() if random.choice([0, 1]) else c.lower(), text))
108
109
110 # --------------------------------------------------
111 if __name__ == '__main__':
112     main()
```

① Set the `random.seed` to the given value or the default `None` which is the same as not setting the seed.

② Create an empty `set` to hold all the unique of words we'll extract from the texts.

③ Iterate through each open file handle.

④ Iterate through each line of text in the file handle.

⑤ Iterate through each word generated by splitting the line on spaces, removing non-word characters with the `clean` function, and filtering for words greater or equal in length to the given minimum.

⑥ Titlecase the word before adding it to the set.

⑦ Use the `sorted` function to order `words` into a new `list`.

⑧ Initialize an empty `list` to hold the `passwords` we will create.

⑨ Use a `for` loop with a `range` to create the correct number of passwords. Since I don't need the actual value from `range`, I can use the `_` to ignore the value.

⑩ Make a new password by joining a random sampling of words on the empty string.

⑪ Now that all the passwords have been created, it's safe to call the `l33t` function if required. If we had used it in the above loop, it would have altered the global state of the `random` module and we would have gotten different passwords.

⑫ If the `l33t` flag is present, obfuscate the password; otherwise, print it as-is.

⑬ Define a function to "clean" a word.

⑭ Use a regular expression to substitute the empty string for anything that is not an English alphabet character.

⑮ Define a function to `l33t` a word.

⑯ First use the `ransom` function to randomly capitalize letters.

⑰ Make a translation table/`dict` for character substitutions.

⑱ Use the `str.translate` function to perform the substitutions, append a random piece of punctuation.

⑲ Define a function for the `ransom` algorithm we wrote in chapter 5.

⑳ Return a new string created by randomly upper- or lowercasing each letter in a word.

1.4. Discussion

Well, that was it. The last exercise! I hope you found it challenging and fun. Let's break it down a bit. There wasn't anything new in `get_args`, so let's start with the auxiliary functions:

1.4.1. Cleaning the text

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

```
1 def clean(word):
2     """Remove non-word characters from word"""
3
4     return re.sub('[^a-zA-Z]', '', word) ①
```

① The `re.sub` function will substitute any text matching the pattern (the first argument) found in the given text (the third argument) with the value given by the second argument.

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

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

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

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

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

It honestly seems like more effort to me. Besides, if the function needed to be changed to allow, say, numbers and a few specific pieces of punctuation, then the regular expression version becomes significantly easier to write and maintain.

1.4.2. A king's ransom

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

```
1 def ransom(text):
2     """Randomly choose an upper or lowercase letter to return"""
3
4     return ''.join( ②
5         map(lambda c: c.upper() if random.choice([0, 1]) else c.lower(), text)) ①
```

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

② Join the resulting `list` from the `map` on the empty string to create a new `str`.

1.4.3. How to `l33t`

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

```
1 def l33t(text):
2     """l33t"""
3
4     text = ransom(text)     ①
5     xform = str.maketrans({ ②
6         'a': '@', 'A': '4', 'O': '0', 't': '+', 'E': '3', 'I': '1', 'S': '5'
7     })
8     return text.translate(xform) + random.choice(string.punctuation) ③
```

① First randomly capitalize the given `text`.

② Make a translation table from the given `dict` which describes how to modify one character to another. Any characters not listed in the keys of this `dict` will be ignored.

③ Use the `str.translate` method to make all the character substitutions. Use `random.choice` to select one additional character from `string.punctuation` to append to the end.

1.4.4. Processing the files

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

```
words = set()
for fh in args.file: ①
    for line in fh:  ②
        for word in filter(lambda w: len(w) >= args.min_word_len, ⑤
                           map(clean,                             ④
                               line.lower().split())):            ③
            words.add(word.title())                               ⑥
```

① Use each open file handle.

② Read the file handle line-by-line with a `for` loop, *not* with a method like `fh.read()` which will read the entire contents of the file at once.

③ Reading this code actually requires starting at the end where we split the `line` on spaces.

④ Each word from `split` goes into `clean`. Notice that we do not need a `lambda` because `clean` expects a single argument.

⑤ Filter out words that are too short. The result from the pipeline goes into the `for` loop.

⑥ Titlecase the word before adding it to the set.

If you don't like the `map` and `filter` functions, then you can rewrite the code in a more traditional way like so:

```
words = set()
for fh in args.file: ①
    for line in fh:  ②
        for word in line.lower().split():      ③
            word = map(clean)                  ④
            if len(word) >= args.min_word_len: ⑤
                words.add(word.title()         ⑥
```

① Iterate through each open file handle.

② Iterate through each line of the file handle.

③ Iterate through each "word" from splitting the lowercased line on spaces.

④ Clean the word up.

⑤ If the word is long enough,

⑥ Then add the titlecased word to the set.

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

1.4.5. Sampling and creating the passwords

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

```
1 words = sorted(words) ①
```

① There is no `set.sort` function. Sets are ordered internally by Python. Calling `sorted` on a `set` will create a new, sorted `list`.

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

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

```
1 for _ in range(args.num): ①
2     password = ''.join(random.sample(words, args.num_words)) ②
3     print(l33t(password) if args.l33t else password)         ③
```

① Iterate through the `args.num` of passwords to create.

② Each password will be based on a random sampling from our `words`, and we will choose the value given in `args.num_words`. The `random.sample` function returns a `list` of words that we `join` on the empty string to create a new string.

③ If the `args.l33t` flag is `True`, then we'll print the l33t version of the password; otherwise, we'll print the password as-is. **This is the bug!** Calling `l33t` here modifies the global state used by the `random` module, so the next time we call `random.sample` we get *a different sample*.

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

```
1 passwords = []                                                  ①
2 for _ in range(args.num):                                       ②
3     passwords.append(''.join(random.sample(words, args.num_words))) ③
4
5 for password in passwords:                                      ④
6     print(l33t(password) if args.l33t else password)            ⑤
```

① Create an empty `list` to hold our new `passwords`.

② Iterate through `args.num`.

③ Create all and store all the `passwords`.

④ Iterate through the `passwords`.

⑤ Print either the `l33t` or plain password.

An alternate way to write this could use list comprehensions and `map`:

```
passwords = [
    ''.join(random.sample(words, args.num_words)) for _ in range(args.num)
]
print('\n'.join(map(l33t, passwords) if args.l33t else passwords))
```

You should write whatever version will still make sense to you when you come back to the code at some point in the future. Remember:

> Any code of your own that you haven't looked at for six or more months might as well have been written by someone else. — Eagleson's Law

1.5. Review

This exercise kind of has it all. Validating user input, reading files, using a new data structure in the `set`, higher-order functions with `map` and `filter`, random values, and lots of functions and tests! I hope you enjoyed programming it, and maybe you'll even use the program to generate your new passwords. Be sure to share those passwords with your author, especially the ones to your bank account and favorite shopping sites!

1.6. Going Further

- The substitution part of the `l33t` function changes every available character which perhaps makes the password too difficult to remember. It would be better to modify only maybe 10% of the password similar to how we changed the input strings in the "Telephone" exercise.
- Create programs that combine other skills you've learned. Like maybe a lyrics generator that randomly selects lines from a files of songs by your favorite bands, then encodes the text with the "Kentucky Friar," then changes all the vowels to one vowel with "Apples and Bananas," and then SHOUTS IT OUT with "The Howler"?

Congratulations, you are now 733+ HAX0R!

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

[2] 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!