Python Dictionaries

Python has a data type called a "dictionary" that allows you to associate some "key" (often a string but it could be a number or even a tuple) to some "value" (which can be anything such as a string, number, tuple, list, set, or another dictionary). The same data structure in other languages is also called a map, hash, and associative array.

You can define the define a dictionary with all the key/value pairs using the `{}` braces:

```
>>> patch = {'species': 'dog', 'age': 4}
>>> patch
{'species': 'dog', 'age': 4}
```

Or you can use the `dict` function and "keyword" arguments (which, in Pythonic style, do not use spaces around the `=` but the whitespace is not actually significant!):

```
>>> patch = dict(species='dog', age=4)
>>> patch
{'species': 'dog', 'age': 4}
```

You might be tempted to use the `{}` curly brackets to access the keys (e.g., if you were coming from Perl or you thought the language might be somehow internally consistent), but Python uses the `[]` square brackets to access dictionary fields just like arrays:

```
>>> patch['species']
'dog'
```

Since a dictionary key may be an integer, it can lead to dictionaries looking like arrays:

```
>>> patch[0] = 'food'
>>> patch[0]
'food'
```

Note that the data types of keys of the dictionary, like lists, may be heterogenous:

```
>>> patch
{'species': 'dog', 'age': 4, 0: 'food'}
>>> list(map(type, patch.keys()))
[<class 'str'>, <class 'str'>, <class 'int'>]
```

As may be the values:

```
>>> type(patch['species'])
<class 'str'>
>>> patch['age']
4
```

```
>>> type(patch['age'])
<class 'int'>
>>> patch['likes'] = ['walking', 'running', 'car trips', 'treats', 'pets']
>>> patch
{'species': 'dog', 'age': 4, 0: 'food', 'likes': ['walking', 'running', 'car trips', 'treats
<class 'list'>
>>> list(map(type, patch.values()))
[<class 'str'>, <class 'int'>, <class 'str'>, <class 'list'>]
```

You can directly use the dictionary values like the data types they are. Here we `join` the `list` that is in the `likes` slot:

```
>>> 'Patch is {} and likes {}'.format(patch['age'], ', '.join(patch['likes']))
'Patch is 4 and likes walking, running, car trips, treats, pets'
```

If you want to know if a key exists, `in` just as we did for list membership:

```
>>> 'likes' in patch
True
>>> 'dislikes' in patch
False
```

Just as you should not request a list position that does not exist in the list, you should not ask for a key that does not exist in a dictionary or you program will asplode at runtime:

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

Better to check first:

```
>>> if 'dislikes' in patch:
...   print(patch['dislikes'])
... else:
...   print('Patch likes everything!')
...
Patch likes everything!
```

Or use the `get` method of the dictionary:

```
>>> patch.get('dislikes')
```

Wait, what did we get?

```
>>> type(patch.get('dislikes'))
<class 'NoneType'>
```

To find all the methods you can call on a dictionary, in the REPL type:

```
>>> help(dict)
```

Type `q` to "quit" the help. Use `/` to initiate a search, e.g., "/pop" to see how you can `pop` similar to the method in the `list` class.

Bridge of Death

Let's write a script to play with a dictionary:

```
$ cat -n bridge_of_death.py
     1  #!/usr/bin/env python3
     2
     3  person = {}
     4  print(person)
     5
     6  print('\n'.join([
     7      'Stop!', 'Who would cross the Bridge of Death',
     8      'Must answer me these questions three,',
     9      '\'ere the other side he see.'
    10  ]))
    11
    12  for field in ['name', 'quest', 'favorite color']:
    13      person[field] = input('What is your {}? '.format(field))
    14      print(person)
    15
    16  if person['favorite color'].lower() == 'blue':
    17      print('Right, off you go.')
    18  else:
    19      print('You have been eaten by a grue.')
```

And here it is in action:

```
$ ./bridge_of_death.py
{}
Stop!
Who would cross the Bridge of Death
must answer me these questions three,
ere the other side he see.
What is your name? Sir Lancelot of Camelot
{'name': 'Sir Lancelot of Camelot'}
What is your quest? To seek the Holy Grail
{'name': 'Sir Lancelot of Camelot', 'quest': 'To seek the Holy Grail'}
What is your favorite color? Blue
{'name': 'Sir Lancelot of Camelot', 'quest': 'To seek the Holy Grail', 'favorite color': 'Bl
Right, off you go.
$ ./bridge_of_death.py
{}
```

```
Stop!
Who would cross the Bridge of Death
must answer me these questions three,
ere the other side he see.
What is your name? Sir Galahad of Camelot
{'name': 'Sir Galahad of Camelot'}
What is your quest? I seek the Holy Grail
{'name': 'Sir Galahad of Camelot', 'quest': 'I seek the Holy Grail'}
What is your favorite color? Blue. No yello--
{'name': 'Sir Galahad of Camelot', 'quest': 'I seek the Holy Grail', 'favorite color': 'Blue
You have been eaten by a grue.
```

Gashlycrumb

Dictionaries are perfect for looking up some bit of information by some value:

```
$ ./gashlycrumb.py c
C is for Clara who wasted away.
$ ./gashlycrumb.py t
T is for Titus who flew into bits.
$ cat -n gashlycrumb.py
     1    #!/usr/bin/env python3
     2    """dictionary lookup"""
     3
     4    import os
     5    import sys
     6
     7    args = sys.argv[1:]
     8
     9    if len(args) != 1:
    10        print('Usage: {} LETTER'.format(os.path.basename(sys.argv[0])))
    11        sys.exit(1)
    12
    13    letter = args[0].upper()
    14
    15    text = """
    16    A is for Amy who fell down the stairs.
    17    B is for Basil assaulted by bears.
    18    C is for Clara who wasted away.
    19    D is for Desmond thrown out of a sleigh.
    20    E is for Ernest who choked on a peach.
    21    F is for Fanny sucked dry by a leech.
    22    G is for George smothered under a rug.
    23    H is for Hector done in by a thug.
```

```
    I is for Ida who drowned in a lake.
    J is for James who took lye by mistake.
    K is for Kate who was struck with an axe.
    L is for Leo who choked on some tacks.
    M is for Maud who was swept out to sea.
    N is for Neville who died of ennui.
    O is for Olive run through with an awl.
    P is for Prue trampled flat in a brawl.
    Q is for Quentin who sank on a mire.
    R is for Rhoda consumed by a fire.
    S is for Susan who perished of fits.
    T is for Titus who flew into bits.
    U is for Una who slipped down a drain.
    V is for Victor squashed under a train.
    W is for Winnie embedded in ice.
    X is for Xerxes devoured by mice.
    Y is for Yorick whose head was bashed in.
    Z is for Zillah who drank too much gin.
    """

    lookup = {}
    for line in text.splitlines():
        if line:
            lookup[line[0]] = line

    if letter in lookup:
        print(lookup[letter])
    else:
        print('I do not know "{}"'.format(letter))
$ ./gashlycrumb.py
Usage: gashlycrumb.py LETTER
$ ./gashlycrumb.py a
A is for Amy who fell down the stairs.
$ ./gashlycrumb.py b
B is for Basil assaulted by bears.
$ ./gashlycrumb.py 8
I do not know "8"
```

On line 47, we create the `lookup` using the first character of the line (`line[0]`). On line 49, we look to see if we have that letter in the `lookup`, printing the line of text if we do or complaining if we don't.

If we return to our previous chapter's DNA base counter, we can use dictionaries for this:

```
$ cat -n dna3.py
     1  #!/usr/bin/env python3
```

```
 2  """Tetra-nucleotide counter"""
 3
 4  import sys
 5  import os
 6
 7  args = sys.argv[1:]
 8
 9  if len(args) != 1:
10      print('Usage: {} DNA'.format(os.path.basename(sys.argv[0])))
11      sys.exit(1)
12
13  dna = args[0]
14
15  count = {}
16
17  for base in dna.lower():
18      if not base in count:
19          count[base] = 0
20
21      count[base] += 1
22
23  counts = []
24  for base in "acgt":
25      num = count[base] if base in count else 0
26      counts.append(str(num))
27
28  print(' '.join(counts))
```

```
$ cat dna.txt
AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGTCTGATAGCAGC
$ ./dna3.py `cat dna.txt`
20 12 17 21
```

But why? Well, this has the great advantage of not having to declare four variables to count the four bases. True, we're only checking (in line 24) for those four, but we can now count all the letters in any string.

Notice that we create a new dict on line 15 with empty curlies `{}`. In line 18, we have to check if the base exists in the dict; if it doesn't, we initialize it to 0, and then we increment it by one. In line 25, we have to be careful when asking for a key that doesn't exist:

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

If we were counting a string of DNA like "AAAAAA," then there would be no C, G or T to report, so we have to use an `if/then` expression:

```
>>> seq = 'AAAAAA'
>>> counts = {}
>>> for base in seq:
...   if not base in counts:
...     counts[base] = 0
...   counts[base] += 1
...
>>> counts
{'A': 6}
>>> counts['G']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'G'
>>> g = counts['G'] if 'G' in counts else 0
```

Or we can use the `get` method of a dictionary to safely get a value by a key even if the key doesn't exist:

```
>>> counts.get('G')
>>> type(counts.get('G'))
<class 'NoneType'>
```

If you look at "dna4.py," you'll see it's exactly the same as "dna3.py" with this exception:

```
23  counts = []
24  for base in "acgt":
25      num = count.get(base, 0)
26      counts.append(str(num))
```

The `get` method will not blow up your program, and it accepts an optional second argument for the default value when nothing is present:

```
>>> cat.get('likes')
>>> type(cat.get('likes'))
<class 'NoneType'>
>>> cat.get('likes', 'Cats like nothing')
'Cats like nothing'
```

Sidebar: Truthiness

Note that you might be tempted to write:

```
>>> cat.get('likes') or 'Cats like nothing'
'Cats like nothing'
```

Which appears to do the same thing, but compare with this:

```
>>> d = {'x': 0, 'y': '', 'z': None}
```

```
>>> for k in sorted(d.keys()):
...   print('{} = "{}"'.format(k, d.get(k) or 'NA'))
...
x = "NA"
y = "NA"
z = "NA"
>>> for k in sorted(d.keys()):
...   print('{} = "{}"'.format(k, d.get(k, 'NA')))
...
x = "0"
y = ""
z = "None"
```

This is a minor but potentially pernicious error due to Python's idea of Truthiness (tm):

```
>>> 1 == True
True
>>> 0 == False
True
```

The integer `1` is not actually the same thing as the boolean value `True`, but Python will treat it as such. Vice verse for `0` and `False`. The only true way to get around this is to explicitly check for `None`:

```
>>> for k in sorted(d.keys()):
...   val = d.get(k)
...   print('{} = "{}"'.format(k, 'NA' if val is None else val))
...
x = "0"
y = ""
z = "NA"
```

To get around the check, we could initialize the dict:

```
$ cat -n dna5.py
     1    #!/usr/bin/env python3
     2    """Tetra-nucleotide counter"""
     3
     4    import sys
     5    import os
     6
     7    args = sys.argv[1:]
     8
     9    if len(args) != 1:
    10        print('Usage: {} DNA'.format(os.path.basename(sys.argv[0])))
    11        sys.exit(1)
    12
    13    dna = args[0]
```

```
14
15    count = {'a': 0, 'c': 0, 'g': 0, 't': 0}
16
17    for base in dna.lower():
18        if base in count:
19            count[base] += 1
20
21    counts = []
22    for base in "acgt":
23        counts.append(str(count[base]))
24
25    print(' '.join(counts))
```

Back To Our Program

Now when we check on line 18, we're only going to count bases that we initialized; further, we can then just use the `keys` method to get the bases:

```
$ cat -n dna5.py
     1    #!/usr/bin/env python3
     2    """Tetra-nucleotide counter"""
     3
     4    import sys
     5    import os
     6
     7    args = sys.argv[1:]
     8
     9    if len(args) != 1:
    10        print('Usage: {} DNA'.format(os.path.basename(sys.argv[0])))
    11        sys.exit(1)
    12
    13    dna = args[0]
    14
    15    count = {'a': 0, 'c': 0, 'g': 0, 't': 0}
    16
    17    for base in dna.lower():
    18        if base in count:
    19            count[base] += 1
    20
    21    counts = []
    22    for base in sorted(count.keys()):
    23        counts.append(str(count[base]))
    24
    25    print(' '.join(counts))
```

This kind of checking and initializing is so common that there is a standard

module to define a dictionary with a default value. Unsurprisingly, it is called "defaultdict":

```
$ cat -n dna6.py
     1    #!/usr/bin/env python3
     2    """Tetra-nucleotide counter"""
     3
     4    import sys
     5    import os
     6    from collections import defaultdict
     7
     8    args = sys.argv[1:]
     9
    10    if len(args) != 1:
    11        print('Usage: {} DNA'.format(os.path.basename(sys.argv[0])))
    12        sys.exit(1)
    13
    14    dna = args[0]
    15
    16    count = defaultdict(int)
    17
    18    for base in dna.lower():
    19        count[base] += 1
    20
    21    counts = []
    22    for base in "acgt":
    23        counts.append(str(count[base]))
    24
    25    print(' '.join(counts))
```

On line 16, we create a `defaultdict` with the `int` type (not in quotes) for which the default value will be zero:

```
>>> from collections import defaultdict
>>> counts = defaultdict(int)
>>> counts['a']
0
```

Finally, I will show you the `Counter` that will do all the base-counting for you, returning a `defaultdict`:

```
>>> from collections import Counter
>>> c = Counter('AACTAC')
>>> c['A']
3
>>> c['G']
0
```

And here is it in the script:

```
$ cat -n dna7.py
     1    #!/usr/bin/env python3
     2    """Tetra-nucleotide counter"""
     3
     4    import sys
     5    import os
     6    from collections import Counter
     7
     8    args = sys.argv[1:]
     9
    10    if len(args) != 1:
    11        print('Usage: {} DNA'.format(os.path.basename(sys.argv[0])))
    12        sys.exit(1)
    13
    14    dna = args[0]
    15
    16    count = Counter(dna.lower())
    17
    18    counts = []
    19    for base in "acgt":
    20        counts.append(str(count[base]))
    21
    22    print(' '.join(counts))
```

So we can take that and create a program that counts all characters either from the command line or a file:

```
$ cat -n char_count1.py
     1    #!/usr/bin/env python3
     2    """Character counter"""
     3
     4    import sys
     5    import os
     6    from collections import Counter
     7
     8    args = sys.argv
     9
    10    if len(args) != 2:
    11        print('Usage: {} INPUT'.format(os.path.basename(args[0])))
    12        sys.exit(1)
    13
    14    arg = args[1]
    15    text = ''
    16    if os.path.isfile(arg):
    17        text = ''.join(open(arg).read().splitlines())
    18    else:
    19        text = arg
```

```
    20
    21    count = Counter(text.lower())
    22
    23    for letter, num in count.items():
    24        print('{} {:5}'.format(letter, num))
$ ./char_count1.py input.txt
a    20
g    17
c    12
t    21
```

Methods

The `keys` from a dict are in no particular order:

```
>>> c = Counter('AAACTAGGGACTGA')
>>> c
Counter({'A': 6, 'G': 4, 'C': 2, 'T': 2})
>>> c.keys()
dict_keys(['A', 'C', 'T', 'G'])
```

If you want them sorted, you must be explicit:

```
>>> sorted(c.keys())
['A', 'C', 'G', 'T']
```

Note that, unlike a list, you cannot call `sort` which makes sense as that will try to sort a list in-place:

```
>>> c.keys().sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'dict_keys' object has no attribute 'sort'
```

You can also just call `values` to get those:

```
>>> c.values()
dict_values([6, 2, 2, 4])
```

Often you will want to go through the `items` in a dict and do something with the key and value:

```
>>> for base, count in c.items():
...   print('{} = {}'.format(base, count))
...
A = 6
C = 2
T = 2
```

```
G = 4
```

But if you want to have the `keys` in a particular order, you can do this:

```
>>> for base in sorted(c.keys()):
...   print('{} = {}'.format(base, c[base]))
...
A = 6
C = 2
G = 4
T = 2
```

Or you can notice that `items` returns a list of tuples:

```
>>> c.items()
dict_items([('A', 6), ('C', 2), ('T', 2), ('G', 4)])
```

And you can call `sorted` on that:

```
>>> sorted(c.items())
[('A', 6), ('C', 2), ('G', 4), ('T', 2)]
```

Which means this will work:

```
>>> for base, count in sorted(c.items()):
...   print('{} = {}'.format(base, count))
...
A = 6
C = 2
G = 4
T = 2
```

Note that `sorted` will sort by the first elements of all the tuples, then by the second, and so forth:

```
>>> genes = [('Indy', 4), ('Boss', 2), ('Lush', 10), ('Boss', 4), ('Lush', 1)]
>>> sorted(genes)
[('Boss', 2), ('Boss', 4), ('Indy', 4), ('Lush', 1), ('Lush', 10)]
```

If we want to sort the bases instead by their frequency, we have to use some trickery like a list comprehension to first reverse the tuples:

```
>>> [(x[1], x[0]) for x in c.items()]
[(6, 'A'), (2, 'C'), (2, 'T'), (4, 'G')]
>>> sorted([(x[1], x[0]) for x in c.items()])
[(2, 'C'), (2, 'T'), (4, 'G'), (6, 'A')]
```

But what is particularly nifty about Counters is that they have built-in methods to help you with such actions:

```
>>> c.most_common(2)
[('A', 6), ('G', 4)]
>>> c.most_common()
```

```
[('A', 6), ('G', 4), ('C', 2), ('T', 2)]
```

You should read the documentation to learn more ([https://docs.python.org/3/library/collections.html](https://d

Character Counter with the works

Finally, I'll show you a version of the character counter that takes some other arguments to control how to show the results:

```
$ cat -n char_count2.py
     1  #!/usr/bin/env python3
     2  """
     3  Author : Ken Youens-Clark <kyclark@email.arizona.edu>
     4  Date   : 2019-02-06
     5  Purpose: Character Counter
     6  """
     7
     8  import argparse
     9  import os
    10  import sys
    11  from collections import Counter
    12
    13
    14  # --------------------------------------------------
    15  def get_args():
    16      """get command-line arguments"""
    17      parser = argparse.ArgumentParser(
    18          description='Character counter',
    19          formatter_class=argparse.ArgumentDefaultsHelpFormatter)
    20
    21      parser.add_argument('input', help='Filename or string to count', type=str)
    22
    23      parser.add_argument(
    24          '-c',
    25          '--charsort',
    26          help='Sort by character',
    27          dest='charsort',
    28          action='store_true')
    29      parser.add_argument(
    30          '-n',
    31          '--numsort',
    32          help='Sort by number',
    33          dest='numsort',
    34          action='store_true')
    35      parser.add_argument(
```

```
36          '-r',
37          '--reverse',
38          help='Sort in reverse order',
39          dest='reverse',
40          action='store_true')
41
42      return parser.parse_args()
43
44
45  # --------------------------------------------------
46  def warn(msg):
47      """Print a message to STDERR"""
48      print(msg, file=sys.stderr)
49
50
51  # --------------------------------------------------
52  def die(msg='Something bad happened'):
53      """warn() and exit with error"""
54      warn(msg)
55      sys.exit(1)
56
57
58  # --------------------------------------------------
59  def main():
60      """Make a jazz noise here"""
61      args = get_args()
62      input_arg = args.input
63      charsort = args.charsort
64      numsort = args.numsort
65      revsort = args.reverse
66
67      if charsort and numsort:
68          die('Please choose one of --charsort or --numsort')
69
70      if not charsort and not numsort:
71          charsort = True
72
73      text = ''
74      if os.path.isfile(input_arg):
75          text = ''.join(open(input_arg).read().splitlines())
76      else:
77          text = input_arg
78
79      count = Counter(text.lower())
80
81      if charsort:
```

```
    82          letters = sorted(count.keys())
    83          if revsort:
    84              letters.reverse()
    85
    86          for letter in letters:
    87              print('{} {:5}'.format(letter, count[letter]))
    88      else:
    89          pairs = sorted([(x[1], x[0]) for x in count.items()])
    90          if revsort:
    91              pairs.reverse()
    92
    93          for n, char in pairs:
    94              print('{} {:5}'.format(char, n))
    95
    96
    97  # ------------------------------------------------
    98  if __name__ == '__main__':
    99      main()
```

Acronym Finder

Similar to the `gashlycrumb.py` program that looked up a line of text for a given letter, we could randomly create meanings for a given acronym:

```
$ ./bacryonym.py NSF
NSF =
 - Nonrepresentationalism Staunchness Forever
 - Naturing Significantly Fontal
 - Nonclinical Solecistical Folkmoter
 - Nonhumanist Scaledrake Fellani
 - Naumk Sulpha Fause
$ ./bacryonym.py FBI
FBI =
 - Folksiness Boxmaker Interviewer
 - Flavorless Bumbler Incorruption
 - Flusterate Bakuninism Isopilocarpine
 - Freshen Bondsman Indigene
 - Fluotantalate Bornyl Interligamentous
```

That is just using the standard dictionary to look up words, so we could make it more interesting by using the works of Shakespeare:

```
$ ./bacryonym.py -w shakespeare.txt FBI
FBI =
 - Furthermore Burnet Instigation
```

```
- Favor Bursting Insisting
- Flower Beart Immanity
- Fearfully Borne Itmy
- Fooleries Blunts Intoxicates
```

Here is the Python for that:

```
$ cat -n bacryonym.py
     1    #!/usr/bin/env python3
     2    """Make guesses about acronyms"""
     3
     4    import argparse
     5    import sys
     6    import os
     7    import random
     8    import re
     9    from collections import defaultdict
    10
    11    # --------------------------------------------------
    12    def main():
    13        """main"""
    14        args = get_args()
    15        acronym = args.acronym
    16        wordlist = args.wordlist
    17        limit = args.num
    18        goodword = r'^[a-z]{2,}$'
    19        badwords = set(re.split(r'\s*,\s*', args.exclude.lower()))
    20
    21        if not re.match(goodword, acronym.lower()):
    22            print('"{}" must be >1 in length, only use letters'.format(acronym))
    23            sys.exit(1)
    24
    25        if not os.path.isfile(wordlist):
    26            print('"{}" is not a file.'.format(wordlist))
    27            sys.exit(1)
    28
    29        seen = {}
    30        words_by_letter = defaultdict(list)
    31        for word in open(wordlist).read().lower().split():
    32            clean = re.sub('[^a-z]', '', word)
    33            if re.match(goodword, clean) and clean not in seen and clean not in badwo
    34                seen[clean] = 1
    35                words_by_letter[clean[0]].append(clean)
    36
    37        len_acronym = len(acronym)
    38        definitions = []
    39        for i in range(0, limit):
```

```
40            definition = []
41            for letter in acronym.lower():
42                possible = words_by_letter[letter]
43                if len(possible) > 0:
44                    definition.append(random.choice(possible).title())
45
46            if len(definition) == len_acronym:
47                definitions.append(' '.join(definition))
48
49        if len(definitions) > 0:
50            print(acronym.upper() + ' =')
51            for definition in definitions:
52                print(' - ' + definition)
53        else:
54            print('Sorry I could not find any good definitions')
55
56    # --------------------------------------------------
57    def get_args():
58        """get arguments"""
59        parser = argparse.ArgumentParser(description='Explain acronyms')
60        parser.add_argument('acronym', help='Acronym', type=str, metavar='STR')
61        parser.add_argument('-n', '--num', help='Maximum number of definitions',
62                            type=int, metavar='NUM', default=5)
63        parser.add_argument('-w', '--wordlist', help='Dictionary/word file',
64                            type=str, metavar='STR',
65                            default='/usr/share/dict/words')
66        parser.add_argument('-x', '--exclude', help='List of words to exclude',
67                            type=str, metavar='STR', default='a,an,the')
68        return parser.parse_args()
69
70    # --------------------------------------------------
71    if __name__ == '__main__':
72        main()
```

Sequence Similarity

We can use dictionaries to count how many words are in common between any two texts. Since I'm only trying to see if a word is present, I can use a `set` which is like a `dict` where the values are just "1." Here is the code:

```
$ cat -n common_words.py
     1    #!/usr/bin/env python3
     2    """Count words in common between two files"""
     3
```

```
 4  import os
 5  import re
 6  import sys
 7  import string
 8
 9  # --------------------------------------------------
10  def main():
11      files = sys.argv[1:]
12
13      if len(files) != 2:
14          msg = 'Usage: {} FILE1 FILE2'
15          print(msg.format(os.path.basename(sys.argv[0])))
16          sys.exit(1)
17
18      for file in files:
19          if not os.path.isfile(file):
20              print('"{}" is not a file'.format(file))
21              sys.exit(1)
22
23      file1, file2 = files[0], files[1]
24      words1 = uniq_words(file1)
25      words2 = uniq_words(file2)
26      common = words1.intersection(words2)
27      num_common = len(common)
28      msg = 'There {} {} word{} in common between "{}" and "{}."'
29      print(msg.format('is' if num_common == 1 else 'are',
30                       num_common,
31                       '' if num_common == 1 else 's',
32                       os.path.basename(file1),
33                       os.path.basename(file2)))
34
35      for i, word in enumerate(sorted(common)):
36          print('{:3}: {}'.format(i + 1, word))
37
38  # --------------------------------------------------
39  def uniq_words(file):
40      regex = re.compile('[' + string.punctuation + ']')
41      words = set()
42      for line in open(file):
43          for word in [regex.sub('', w) for w in line.lower().split()]:
44              words.add(word)
45
46      return words
47
48  # --------------------------------------------------
49  if __name__ == '__main__':
```

```
    50        main()
```

Let's see it in action using a common nursery rhyme and a poem by William Blake (1757-1827):

```
$ cat mary-had-a-little-lamb.txt
Mary had a little lamb,
It's fleece was white as snow,
And everywhere that Mary went,
The lamb was sure to go.
$ cat little-lamb.txt
Little Lamb, who made thee?
Dost thou know who made thee?
Gave thee life, & bid thee feed
By the stream & o'er the mead;
Gave thee clothing of delight,
Softest clothing, wooly, bright;
Gave thee such a tender voice,
Making all the vales rejoice?
Little Lamb, who made thee?
Dost thou know who made thee?
Little Lamb, I'll tell thee,
Little Lamb, I'll tell thee,
He is called by thy name,
For he calls himself a Lamb.
He is meek, & he is mild;
He became a little child.
I a child, & thou a lamb,
We are called by his name.
Little Lamb, God bless thee!
Little Lamb, God bless thee!
$ ./common_words.py mary-had-a-little-lamb.txt little-lamb.txt
There are 4 words in common between "mary-had-a-little-lamb.txt" and "little-lamb.txt."
  1: a
  2: lamb
  3: little
  4: the
```

Well, that's pretty uninformative. Sure "a" and "the" are shared, but we don't much care about those. And while "little" and "lamb" are present, it hardly tells us about how prevalent they are. In the nursery rhyme, they occur a total of 3 times, but they make up a significant portion of the Blake poem. Let's try to work in word frequency:

```
$ cat -n common_words2.py
     1    #!/usr/bin/env python3
     2    """Count words/frequencies in two files"""
     3
```

```
 4  import os
 5  import re
 6  import sys
 7  import string
 8  from collections import defaultdict
 9
10  # --------------------------------------------------
11  def word_counts(file):
12      """Return a dictionary of words/counts"""
13      words = defaultdict(int)
14      regex = re.compile('[' + string.punctuation + ']')
15      for line in open(file):
16          for word in [regex.sub('', w) for w in line.lower().split()]:
17              words[word] += 1
18
19      return words
20
21  # --------------------------------------------------
22  def main():
23      """Start here"""
24      args = sys.argv[1:]
25
26      if len(args) != 2:
27          msg = 'Usage: {} FILE1 FILE2'
28          print(msg.format(os.path.basename(sys.argv[0])))
29          sys.exit(1)
30
31      for file in args[0:2]:
32          if not os.path.isfile(file):
33              print('"{}" is not a file'.format(file))
34              sys.exit(1)
35
36      file1 = args[0]
37      file2 = args[1]
38      words1 = word_counts(file1)
39      words2 = word_counts(file2)
40      common = set(words1.keys()).intersection(set(words2.keys()))
41      num_common = len(common)
42      verb = 'is' if num_common == 1 else 'are'
43      plural = '' if num_common == 1 else 's'
44      msg = 'There {} {} word{} in common between "{}" ({}) and "{}" ({}).'
45      tot1 = sum(words1.values())
46      tot2 = sum(words2.values())
47      print(msg.format(verb, num_common, plural, file1, tot1, file2, tot2))
48
49      if num_common > 0:
```

```
    50            fmt = '{:>3} {:20} {:>5} {:>5}'
    51            print(fmt.format('#', 'word', '1', '2'))
    52            print('-' * 36)
    53            shared1, shared2 = 0, 0
    54            for i, word in enumerate(sorted(common)):
    55                c1 = words1[word]
    56                c2 = words2[word]
    57                shared1 += c1
    58                shared2 += c2
    59                print(fmt.format(i + 1, word, c1, c2))
    60
    61            print(fmt.format('', '-----', '--', '--'))
    62            print(fmt.format('', 'total', shared1, shared2))
    63            print(fmt.format('', 'pct',
    64                             int(shared1/tot1 * 100), int(shared2/tot2 * 100)))
    65
    66    # ------------------------------------------------
    67    if __name__ == '__main__':
    68        main()
```

And here it is in action:

```
$ ./common_words2.py mary-had-a-little-lamb.txt little-lamb.txt
There are 4 words in common between "mary-had-a-little-lamb.txt" (22) and "little-lamb.txt"
  # word                      1     2
------------------------------------
  1 a                         1     5
  2 lamb                      2     8
  3 little                    1     7
  4 the                       1     3
    -----                    --    --
    total                     5    23
    pct                      22    20
```

It is interesting (to me, at least) that the shared content actually works out to about the same proportion no matter the direction. Imagine comparing a large genome to a smaller one – what is a significant portion of shared sequence space from the smaller genome might be only a small fraction of the larger one. Here we see that just those few words make up an equivalent proportion of both texts because of how repeated the words are in the Blake poem.

This is all pretty good as long as the words are spelled the same, but take the two texts here that show variations between British and American English:

```
$ cat british.txt
I went to the theatre last night with my neighbour and had a litre of
beer, the colour and flavour of which put us into such a good humour
that we forgot our labours.  We set about to analyse our behaviour,
```

```
organise our thoughts, recognise our faults, catalogue our merits, and
generally have a dialogue without pretence as a licence to improve
ourselves.
$ cat american.txt
I went to the theater last night with my neighbor and had a liter of
beer, the color and flavor of which put us into such a good humor that
we forgot our labors.  We set about to analyze our behavior, organize
our thoughts, recognize our faults, catalog our merits, and generally
have a dialog without pretense as a license to improve ourselves.
$ ./common_words2.py british.txt american.txt
There are 34 words in common between "british.txt" (63) and "american.txt" (63).
  # word                      1     2
-----------------------------------
  1 a                         4     4
  2 about                     1     1
  3 and                       3     3
  4 as                        1     1
  5 beer                      1     1
  6 faults                    1     1
  7 forgot                    1     1
  8 generally                 1     1
  9 good                      1     1
 10 had                       1     1
 11 have                      1     1
 12 i                         1     1
 13 improve                   1     1
 14 into                      1     1
 15 last                      1     1
 16 merits                    1     1
 17 my                        1     1
 18 night                     1     1
 19 of                        2     2
 20 our                       5     5
 21 ourselves                 1     1
 22 put                       1     1
 23 set                       1     1
 24 such                      1     1
 25 that                      1     1
 26 the                       2     2
 27 thoughts                  1     1
 28 to                        3     3
 29 us                        1     1
 30 we                        2     2
 31 went                      1     1
 32 which                     1     1
 33 with                      1     1
```

```
 34 without                  1     1
    -----                   --    --
    total                   48    48
    pct                     76    76
```

Obviously we will miss all those words because the are not spelled exactly the same. Neither are genomes. So we need a way to decide if two words or sequences are similar enough. One way is through sequence alignment:

```
l a b o u r      c a t a l o g u e      p r e t e n c e     l i t r e
| | | |   |      | | | | | | |          | | | | | |   |     | | |
l a b o   r      c a t a l o g          p r e t e n s e     l i t e r
```

Try writing a sequence alignment program (no, really!), and you'll find it's really quite difficult. Decades of research have gone into Smith-Waterman and BLAST and BLAT and LAST and more. Alignment works very well, but it's computationally expensive. We need a faster approximation of similarity. Enter k-mers!

A k-mer is a `k` length of "mers" or contiguous sequence (think "polymers"). Here are the 3/4-mers in my last name:

```
$ ./kmer_tiler.py youens
There are 4 3-mers in "youens."
youens
you
 oue
  uen
   ens
$ ./kmer_tiler.py youens 4
There are 3 4-mers in "youens."
youens
youe
 ouen
  uens
```

If instead looking for shared "words" we search for k-mers, we will find very different results, and the length of the k-mer matters. For instance, the first 3-mer in my name, "you" can be found 81 times in my local dictionary, but the 4-mer "youe" not at all. The longer the k-mer, the greater the specificity. Let's try our English variations with a k-mer counter:

```
$ ./common_kmers.py british.txt american.txt
There are 112 kmers in common between "british.txt" (127) and "american.txt" (127).
  # kmer                      1     2
-----------------------------------
  1 abo                       2     2
  2 all                       1     1
...
```

```
111 whi                     1     1
112 wit                     2     2
    -----                  --    --
    total                 142   133
    pct                    86    86
```

Our word counting program thought these two texts only 76% similar, but our kmer counter thinks they are 86% similar.