Chapter 8: Gashlycrumb: Looking items up in a dictionary

Every time you log into a website, the code behind it has to look up your username and password to compare to the values you put into the login form. Whenever you give your phone number at the hardware store or scan your library card to checkout a book, a computer program uses one piece of information to find other things like how often you buy compost or if you have any overdue books. Probably all these examples would be using a database to find that information. We're going to use a dictionary that we will fill with information from an input file.

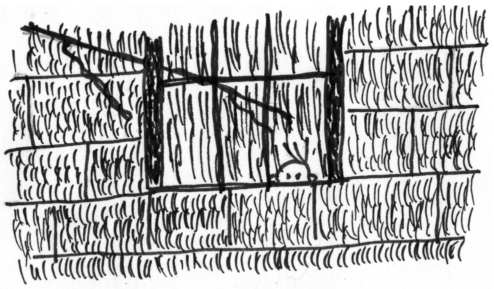


Figure 1. N is for Neville who died of ennui.

In this exercise, we're going to look up a line of text from an input file that starts with the letter provided by the user. The text will come from an input file which will default to Edward Gorey's "The Gashlycrumb Tinies," an abecedarian book that describes various and ghastly ways in which children die. Instead of "A is for apple, B is for bear," we get:

> *A is for Amy who fell down the stairs. B is for Basil assaulted by bears.*

Our `gashlycrumb.py` program will take a letter of the alphabet as a positional argument and will look up the line of text from an *optional* input file that starts with that letter. The input file will have each letter on a separate line:

```
$ head -2 gashlycrumb.txt
A is for Amy who fell down the stairs.
B is for Basil assaulted by bears.
```

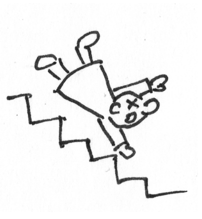
When our unfortunate user runs our program, here is what they would see. Note that we will consider the letter in a *case-insensitive* fashion:

```
$ ./gashlycrumb.py a
A is for Amy who fell down the stairs.
```

Our program will need to:

- Accept a single positional argument we'll call `letter`
- Accept an optional `--file` argument which must be a file. The default value will be `'gashlycrumb.txt'` (provided).
- Read the file, find the first letter of each line, and build a data structure that associates that letter to the line of text.
- Check if the given `letter` is a present in the given `file`.
- Print the line of text for the `letter` if present or an error if it isn't.

You can draw from several previous programs:



- From the "Word Count" program, you know how to take a file input and read it line-by-line.
- From the "Article" program, you know how to get the first letter of a bit of text.
- From the "Jump The Five" program, you know how to build a dictionary and lookup a value.

Now you'll put all those skills together to recite morbid poetry!

Writing `gashlycrumb.py`

Before you begin writing, I would encourage you to run the tests with `make test` or `pytest -xv test.py` in the `gashlycrumb` directory. The first test should fail:

```
test.py::test_exists FAILED
```

This is just a reminder that the first thing to do is to create the file called `gashlycrumb.py`. You can do this however you like, maybe by running `new.py gashlycrumb.py` in the `gashlycrumb` directory or by copying the `template/template.py` file, or by just starting a new file from scratch.

Run your tests again and you should pass the first and possibly the second tests if your program produces a usage statement. Next let's get the arguments straight. Modify your program's parameters in the `get_args` function so that it will produce the following usage when the program is run with *no arguments* or with the `-h` or `--help` flags:

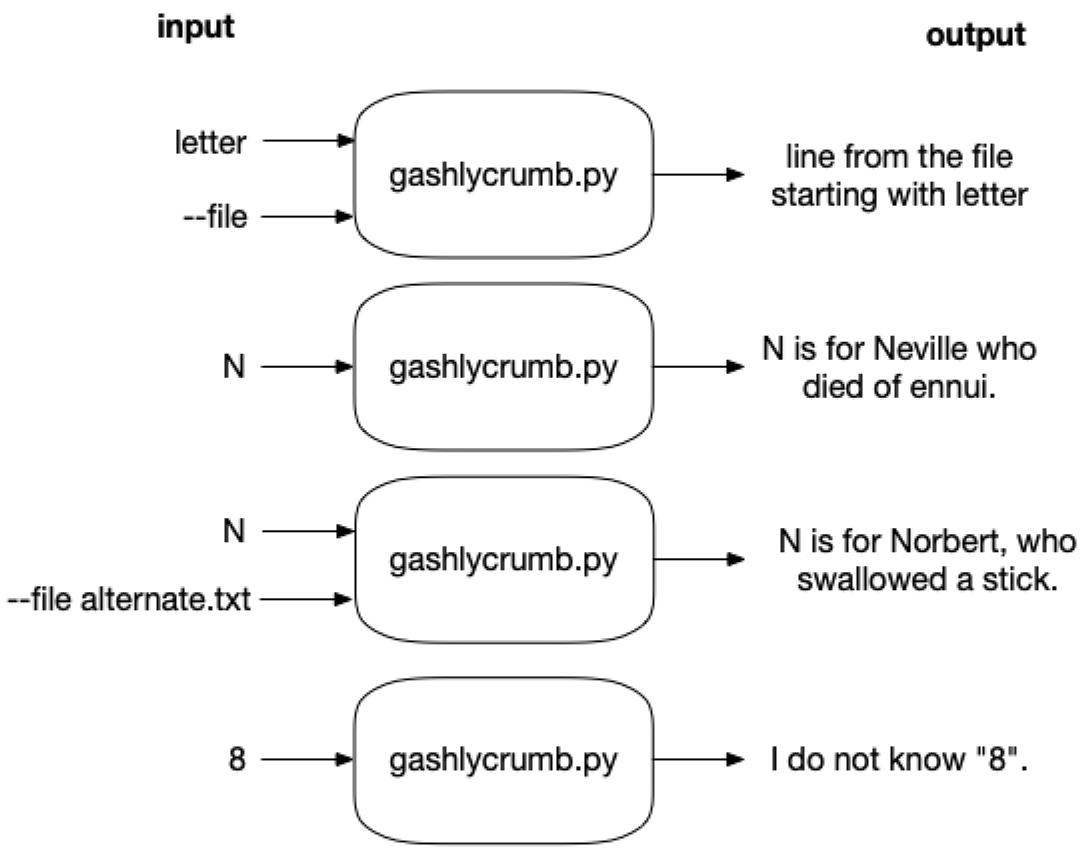
```
$ ./gashlycrumb.py -h
usage: gashlycrumb.py [-h] [-f str] str

Gashlycrumb

positional arguments:
  str                 Letter  1

optional arguments:
  -h, --help          show this help message and exit.      2
  -f str, --file str  Input file (default: gashlycrumb.txt) 3
```

1 The "letter" is a required positional argument.

2 The `-h` and `--help` arguments are created automatically by `argparse`.

3 The `-f` or `--file` argument is an option with a default value of "gashlycrumb.txt"

Here is a string diagram showing how the program will work:



Once you have the correct usage, start off by echoing the `letter`:

```
def main():
    args = get_args()
    print(args.letter)
```

Try running it to make sure it works:

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

Next read the `file` line-by-line using a `for` loop. Note that I'm using `end=''` with the `print` so that it won't print the newline that's already attached to each `line` of the `file`:

```
def main():
    args = get_args()
    print(args.letter)

    for line in args.file:
        print(line, end='')
```

Try running it to ensure you can read the input `file`:

```
$ ./gashlycrumb.py a | head -3
a
A is for Amy who fell down the stairs.
B is for Basil assaulted by bears.
```

Use the "alternate.txt" file, too:

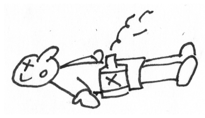
```
$ ./gashlycrumb.py a -f alternate.txt | head -3
a
A is for Alfred, poisoned to death.
B is for Bertrand, consumed by meth.
```

If provided a `--file` argument that does not exist, your program should exit with an error and message. Note that, if you use `type=argparse.FileType('r')`, this error should be produced automatically by `argparse`:

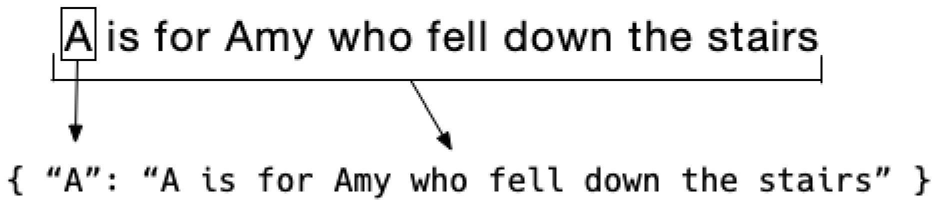
```
$ ./gashlycrumb.py -f blargh b
usage: gashlycrumb.py [-h] [-f str] str
gashlycrumb.py: error: argument -f/--file: can't open 'blargh': [Errno 2] No such file or directory: 'blargh'
```

Now think about how you can use the first letter of each `line` to create an entry into a `dict`. Use the `print` command to look at your dictionary. Figure out how to check if the given `letter` is *in* (wink, wink, nudge, nudge) your dictionary. If given a value that does not exist in the list of first characters on the lines from the input file (when searched without regard to case), you should print a message:

```
$ ./gashlycrumb.py 3
I do not know "3".
$ ./gashlycrumb.py CH
I do not know "CH".
```



If the given `letter` is in the dictionary, print the value for it:


A is for Amy who fell down the stairs
{ "A": "A is for Amy who fell down the stairs" }

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

Run the test suite to ensure your program meets all the requirements. Read the errors closely and fix your program.

Hints:

- Start with `new.py` and remove everything but the positional and optional `argparse.FileType('r')` parameters.
- A dictionary is a natural data structure that you can use to associate some value like the letter "A" to some phrase like "A is for Amy who fell down the stairs." Create a new, empty `dict`.
- Once you have an open file handle, you can `read` the file line-by-line with a `for` loop.
- Each line of text is a string. How can you get the first character of a string?
- Using that first character, how can you set the value of a `dict` to be the key and the line itself to be the value?
- Once you have constructed the dictionary of letters to lines, how can you check that the user's `letter` argument is `in` the dictionary?
- Can you solve this without a `dict`?

No skipping ahead to the solution until you have written your own version! If you peek, you might be die a horrible death stampeded by kittens.

Solution

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

import argparse


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

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

    parser.add_argument('letter', help='Letter', metavar='str', type=str)  1

    parser.add_argument('-f',  2
                        '--file',
                        help='Input file',
                        metavar='str',
                        type=argparse.FileType('r'),
                        default='gashlycrumb.txt')

    return parser.parse_args()


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

    args = get_args()
    letter = args.letter                    3
    lookup = {line[0].upper(): line.rstrip() for line in args.file}  4

    if letter.upper() in lookup:            5
        print(lookup[letter.upper()])       6
    else:
        print(f'I do not know "{letter}".')  7


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

1 The required argument (a letter).

2 An optional `--file` that must be a readable file, if provided. Defaults to a known value.

3 Copy the `args.letter` value into the variable `letter`.

4 Build the `lookup` using a dictionary comprehension that reads the given file. Use the `upper` function to disregard case.

5 See if the `letter` argument is in the `lookup` dictionary, checking the `upper` value to disregard case.

6 If so, `print` the line of text from the `lookup` for the `letter`.

7 Otherwise, print a message that the `letter` is unknown.

Discussion

Did the frightful paws of the kittens hurt much? Let’s talk about how I solved this problem. Remember, mine is just one of many possible solutions.

Handling the arguments

I prefer to have all the logic for parsing and validating the command-line arguments in the `get_args` function. In particular, `argparse` can do a fine job verifying tedious things such as an argument being an existing, readable `--file` which is why I use `type=argparse.FileType('r')` for that argument. If the user doesn’t supply a valid argument, then `argparse` will throw an error, printing a helpful message along with the short usage and exiting with an error code.

By the time I get to the line `args = get_args()`, I know that I have a valid, open file handle in the `args.file` slot. In the REPL, I can manually do what `argparse` has done by using `open` to get a file handle which I like to usually call `fh`. For copyright purposes, I’ll use my alternate text:

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

Reading the input file

I know that I want to use a dictionary where the keys are the first letters of each line and the values are the lines themselves. That means I need to start by creating a new, empty dictionary either by using the `dict()` function or by setting a variable equal to an empty set of curly braces (`{}`). I’ll call my variable `lookup`:

```
>>> lookup = {}
```

I will use a `for` loop to read each `line` of text. From Crow’s Nest, I know I can use `line[0].upper()` to get the first letter of `line` and uppercase it. I can use that as the key into `lookup`. Each `line` of text ends with a newline that I’d like to remove. I can use the `str.rstrip` method to strip whitespace from the right side of the `line` (`rstrip` = *right strip*). The result of that will be the value for my `lookup`:

```
for line in fh:
    lookup[line[0].upper()] = line.rstrip()
```

I’d like to look at the resulting `lookup` dictionary. I can `print` it from the program or type `lookup` in the REPL, but it’s going to be hard to read. I encourage you to try it. Luckily there is a lovely module called `pprint` to "pretty print" data structures. Here is how I can import the `pprint.pprint` function as the shortcut `pp`:

```
>>> from pprint import pprint as pp
```

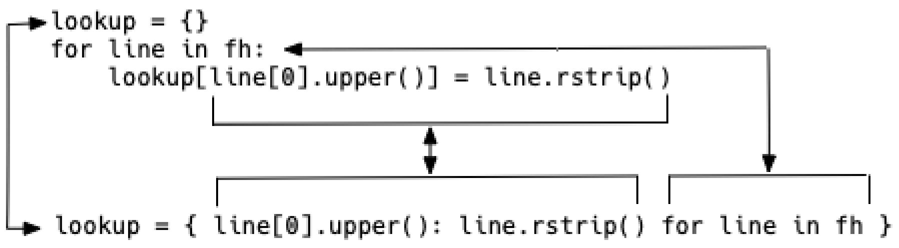
Now let’s take a peek at `lookup`:

```
>>> pp(lookup)
{'A': 'A is for Alfred, poisoned to death.',
 'B': 'B is for Bertrand, consumed by meth.',
 'C': 'C is for Cornell, who ate some glass.',
 'D': 'D is for Donald, who died from gas.',
 'E': 'E is for Edward, hanged by the neck.',
 'F': 'F is for Freddy, crushed in a wreck.',
 'G': 'G is for Geoffrey, who slit his wrist.',
 'H': "H is for Henry, who's neck got a twist.",
 'I': 'I is for Ingrid, who tripped down a stair.',
 'J': 'J is for Jered, who fell off a chair,',
 'K': 'K is for Kevin, bit by a snake.',
 'L': 'L is for Lauryl, impaled on a stake.',
 'M': 'M is for Moira, hit by a brick.',
 'N': 'N is for Norbert, who swallowed a stick.',
 'O': 'O is for Orville, who fell in a canyon,',
 'P': 'P is for Paul, strangled by his banyan,',
 'Q': 'Q is for Quintanna, flayed in the night,',
 'R': 'R is for Robert, who died of spite,',
 'S': 'S is for Susan, stung by a jelly,',
 'T': 'T is for Terrange, kicked in the belly,',
 'U': "U is for Uma, who's life was vanquished,",
 'V': 'V is for Victor, consumed by anguish,',
 'W': "W is for Walter, who's socks were too long,",
 'X': 'X is for Xavier, stuck through with a prong,',
 'Y': 'Y is for Yoeman, too fat by a piece,',
 'Z': 'Z is for Zora, smothered by a fleece.'}
```

Hey, that looks like a handy data structure. Hooray for us!

Looping with `for` versus a list comprehensions

We used three lines of code to build our `lookup` dictionary. We can actually accomplish that in *one* line by using a "dictionary comprehension." We've created list comprehensions by sticking a `for` inside brackets `[]`. A dictionary comprehension is the same but the `for` loop is inside curlies `{}`.


lookup = {}
for line in fh:
    lookup[line[0].upper()] = line.rstrip()
lookup = { line[0].upper(): line.rstrip() for line in fh }

If you are following along by pasting code into the REPL, note that we have exhausted the file handle `fh` just above by reading it. (Refer back to the Howler to read about file handles.) I need to `open` it again for this next bit:

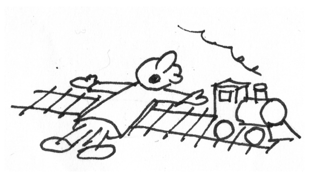
```
>>> fh = open('gashlycrumb.txt')
>>> lookup = { line[0].upper(): line.rstrip() for line in fh }
```

If you `pprint` it again, you should see the same output as above. It may seem like showing off to write one line of code instead of three, but it really does make a good deal of sense to write compact, idiomatic code. More code always means more chances for bugs, so I usually try to write code that is as simple as possible (but no simpler).

Dictionary lookups

Now that I have a `lookup`, I can ask if some value is `in` the keys. Note that I know the letters are in uppercase and I assume the user could give me lower, so I use `letter.upper()` to only compare that case:

```
>>> letter = 'a'
>>> letter.upper() in lookup
True
>>> lookup[letter.upper()]
'A is for Amy who fell down the stairs.'
```



If the letter is found, I can print the line of text for that letter; otherwise, I can print the message that I don't know that letter:

```
>>> letter = '4'
>>> if letter.upper() in lookup:
...     print(lookup[letter.upper()])
... else:
...     print('I do not know "{}".'.format(letter))
...
I do not know "4".
```

`dict` vs `list` of `tuple`

I don't have to use a `dict` to solve this problem. I could, for example, use a `list` of `tuple` values. We haven't talked about the `tuple` yet. It's an immutable `list`, and you can create a tuple by separating items with commas:

```
>>> x = 'foo', 'bar'
>>> type(x)
<class 'tuple'>
```

It's common to put parentheses around the values if only to make your intent clearer. For instance, this is valid syntax for a tuple with only one value:

```
>>> x = 'foo',
```

To me, that looks like an incomplete statement. I think this is easier to understand:

```
>>> x = ('foo',)
```

Note that the trailing comma is required or you end up with a single value:

```
>>> x = ('foo')
>>> type(x)
<class 'str'>
```

Let's compare and contrast a `tuple` and a `list`:

```
>>> x = ('foo', 'bar')
>>> y = ['foo', 'bar']
```

We can use the `len` function to find their lengths:

```
>>> len(x)
2
>>> len(y)
2
```

And we can use index values to get specific elements:

```
>>> x[0]
'foo'
>>> y[1]
'bar'
```

We can alter the `list` called `y` like maybe adding a new value:

```
>>> y.append('baz')
>>> y
['foo', 'bar', 'baz']
```

But we cannot do this to the `tuple` called `x`:

```
>>> x.append('baz')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
```

We can also change any element inside a `list`:

```
>>> y[0] = 'quux'
>>> y
['quux', 'bar', 'baz']
```

But, again, a `tuple` is immutable:

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

The `dict.items` method will return a `list` of `tuple` values. Our `lookup` is rather large, so here's a way to look at the first three values of that using `pp` (pretty print) function. Note the outermost square brackets `[]` show that this is a `list`. Each element of the list is enclosed in parentheses to indicate they are `tuple` values:

```
>>> pp(list(lookup.items())[:3])
[('A', 'A is for Amy who fell down the stairs.'),
 ('B', 'B is for Basil assaulted by bears.'),
 ('C', 'C is for Clara who wasted away.')]
```

So a `dict` can be thought of as a list of tuples which I could make directly using a list comprehension:

```
>>> fh = open('gashlycrumb.txt')
>>> lookup = [ (line[0].upper(), line.rstrip()) for line in fh ]
```

I can unpack `lookup` into `char` and `line` to extract, say, just the first 10 `char` elements:

```
>>> [char for char, line in lookup][:10]
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J']
```

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

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

And get the value like so:

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

The problem is that the cost of the search is proportional to the number of values. That is, if we were searching a million keys in a list, then Python starts searching at the beginning of the list and goes until it finds the value. When you store items in a `dict`, the search time for a key can be much shorter, often nearly instantaneous. The method `dict.items()` will return a `list` of `tuple` values containing `(key, value)`, so the line between these two is already blurry.

Review

- A dictionary comprehension is a way to build a dictionary in a one-line `for` loop.
- Defining file input arguments using `argparse.FileType` saves you time and code.
- Looking up a key in a `dict` is faster than searching for an element in a `list`. The time to search a `list` is proportional to its length while dictionary keys are stored in a way optimized for finding them.
- Python's `pprint` module is used for "pretty printing" complex data structures.

Going Further

- Write an interactive version that takes input directly from the user. Use `while True` to set up an infinite loop and keep using `input` to get the user's next `letter`:

  ```
  $ ./gashlycrumb_interactive.py
  Please provide a letter [! to quit]: t
  T is for Titus who flew into bits.
  Please provide a letter [! to quit]: 7
  I do not know "7".
  Please provide a letter [! to quit]: !
  Bye
  ```

Last updated 2020-01-14 20:37:44 -0700