# 1. Gashlycrumb: Looking items up in a dictionary

Every time you log into a website, the code behind it has to look up your *N is for Neville who died* username and password to compare to the values you put into the login *of ennui.* 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.
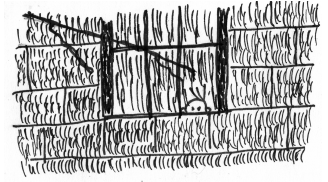
In this exercise, we're going to look up a lines of text from an input file that start with the letters provided by the user. The text will come from an input file which will default to Edward Gorey's "The Gashlycrumb Tinies," an abecedarian book that describes various and ghastly ways in which children die. Instead of "A is for artichoke, B is for blackberry," we get:

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

Our `gashlycrumb.py` program will take one or more letters of the alphabet as positional arguments and will look up the lines of text from an *optional* input file that start 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 e f
E is for Ernest who choked on a peach.
F is for Fanny sucked dry by a leech.
```

In this exercise, you will:

- Accept one or more positional arguments 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. (We'll only be using files where each line starts with a single, unique letter. This program would fail with any other format of text.)
- For each `letter` provided by the user, either print the line of text for the `letter` if present or a

message if it isn't.

- Learn how to "pretty print" a data structure.

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 "Crow's Nest" 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.
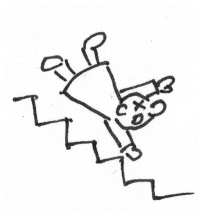
Now you'll put all those skills together to recite morbid poetry!

1.1. 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 FILE] letter [letter ...]

Gashlycrumb

positional arguments:
  letter                Letter(s)                         ①

optional arguments:
  -h, --help            show this help message and exit   ②
  -f FILE, --file FILE  Input file (default: gashlycrumb.txt) ③
```
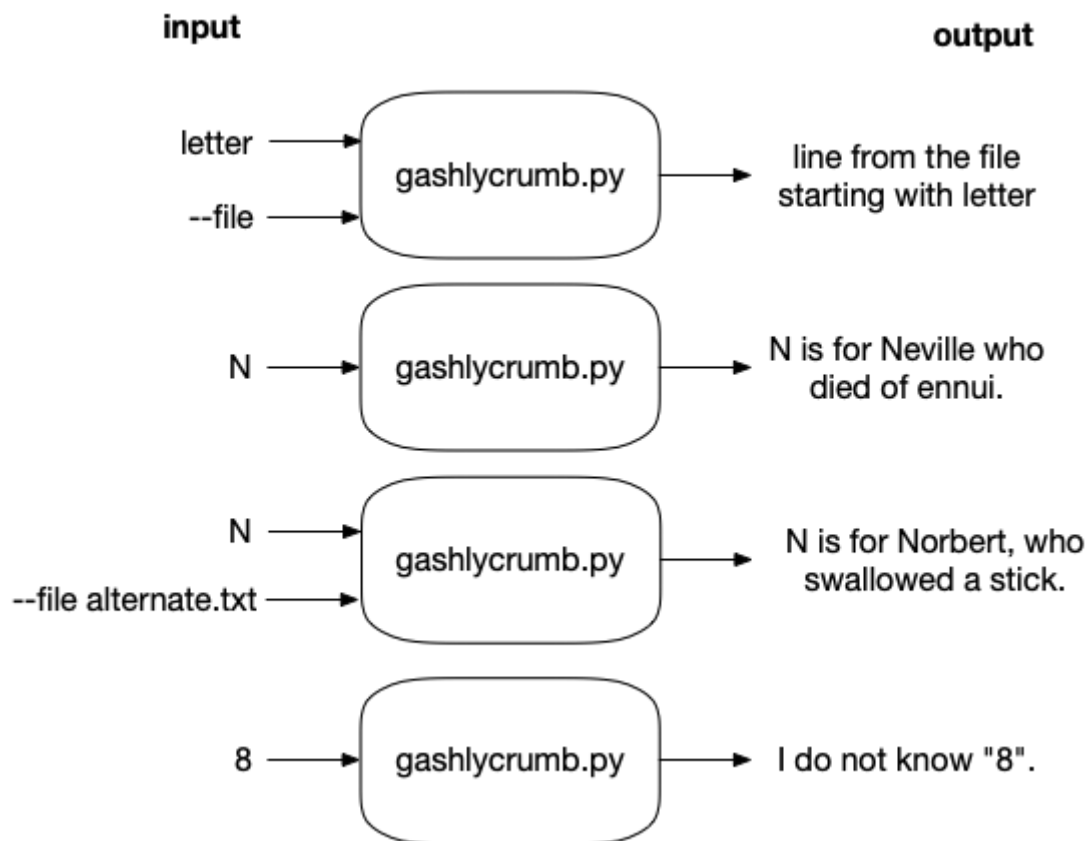
① The `letter` is a required positional argument that accepts one or more values (notice `[str ···]` to indicate this).

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 each of the `letter` arguments:

```
1 def main():
2     args = get_args()
3     for letter in args.letter:
4         print(letter)
```

Try running it to make sure it works:

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

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`:

```
1 def main():
2     args = get_args()
3     for letter in args.letter:
4         print(letter)
5
6     for line in args.file:
7         print(line, end='')
```

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

```
$ ./gashlycrumb.py a b | head -4
a
b
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 b --file alternate.txt | head -4
a
b
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 FILE] letter [letter ...]
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:
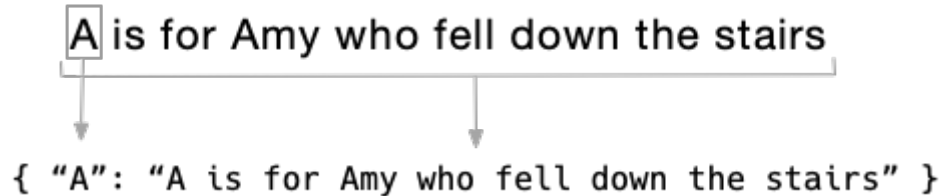
```
$ ./gashlycrumb.py 3
I do not know "3".
$ ./gashlycrumb.py CH
I do not know "CH".
```

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

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

A is for Amy who fell down the stairs

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

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 `letter` and optional `--file` parameters.

- Use `type=argparse.FileType('r')` to validate the `--file` argument.

- Use `nargs='+'` to define the positional argument `letter` so it will require one or more values.

- 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?

- Iterate through each `letter` argument. How can you check that a given value is `in` the dictionary?

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

1.2. Solution

```python
1  #!/usr/bin/env python3
2  """Lookup tables"""
3
4  import argparse
5
6
7  # --------------------------------------------------
8  def get_args():
9      """get command-line arguments"""
10
11     parser = argparse.ArgumentParser(
12         description='Gashlycrumb',
13         formatter_class=argparse.ArgumentDefaultsHelpFormatter)
14
15     parser.add_argument('letter',
16                         help='Letter(s)',
17                         metavar='letter',
18                         nargs='+',                    ①
19                         type=str)
20
21     parser.add_argument('-f',
22                         '--file',
23                         help='Input file',
24                         metavar='FILE',
25                         type=argparse.FileType('r'), ②
26                         default='gashlycrumb.txt')
27
28     return parser.parse_args()
29
30
31 # --------------------------------------------------
32 def main():
33     """Make a jazz noise here"""
34
35     args = get_args()
36
37     lookup = {}                 ③
38     for line in args.file:  ④
39         lookup[line[0].upper()] = line.rstrip()      ⑤
40
41     for letter in args.letter:                       ⑥
42         if letter.upper() in lookup:                 ⑦
43             print(lookup[letter.upper()])            ⑧
44         else:
45             print(f'I do not know "{letter}".')      ⑨
46
47
48 # --------------------------------------------------
49 if __name__ == '__main__':
50     main()
```

① A positional argument we'll call `letter` that uses `nargs='+'` to indicate one or more values are required.

② The optional `--file` argument must be a readable file because of `type=argparse.FileType('r')`. The `default` value is the `gashlycrumb.txt` file which we know to exist.

③ Create an empty dictionary to hold our `lookup` table.

④ Iterate through each line of the `args.file` which will be an open file handle.

⑤ Uppercase the first character of the line to use as the key into the `lookup` and set the value to be the line stripped of whitespace on the right side.

⑥ Use a `for` loop to iterate over each `letter` in `args.letter`.

⑦ See if the `letter` is in the `lookup` dictionary, checking the `letter.upper()` value to disregard case.

⑧ If so, `print` the line of text from the `lookup` for the `letter`.

⑨ Otherwise, print a message that the `letter` is unknown.

1.3. 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.

1.3.1. 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 arguments, 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 one or more "letter" arguments and a valid, open file handle in the `args.file` slot. In the REPL, I can use `open` to get a file handle which I like to usually like to call `fh`. For copyright purposes, I'll use my alternate text:

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

1.3.2. Reading the input file

We know that we want to use a dictionary where the keys are the first letters of each line and the values are the lines themselves. That means we 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 = {}
```

We can use a `for` loop to read each `line` of text. From the "Crow's Nest" program, we know we can use `line[0].upper()` to get the first letter of `line` and uppercase it. We can use that as the key into

lookup. Each `line` of text ends with a newline that I'd like to remove. The `str.rstrip()` method will strip whitespace from the right side of the `line` (`rstrip` = *right strip*). The result of that will be the value for my `lookup`:

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

We'd like to look at the resulting `lookup` dictionary. We 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 we can import the `pprint()` function from the `pprint` module with the alias `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!

1.3.3. Using a dictionary comprehension

In "Jump the Five," we saw that a list comprehension is a way to build a list by putting a `for` loop inside `[]`. If we change the brackets to curlies (`{}`), then we create a dictionary comprehension. This code is equivalent to the `for` loop above:

```
>>> fh = open('gashlycrumb.txt')
>>> lookup = { line[0].upper(): line.rstrip() for line in fh }
```
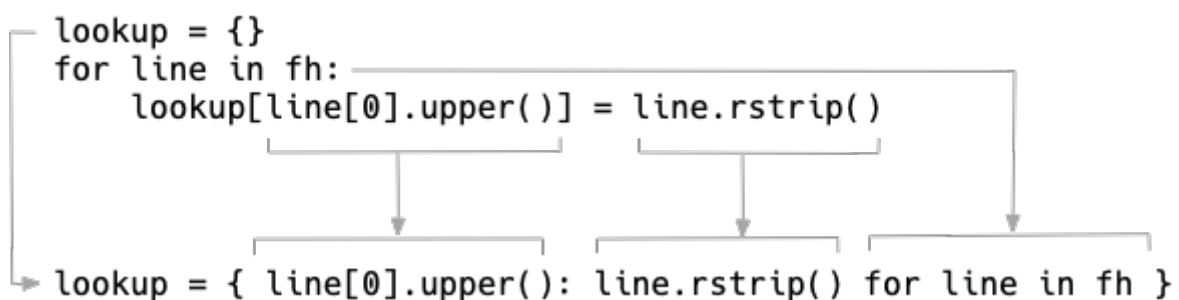
If you print the `lookup` again, you should see the same output as above. It may seem like showing off to write one line of code instead of three, but it really does make a good deal of sense to write compact, idiomatic code. More code always means more chances for bugs, so I usually try to write code that is as simple as possible (but no simpler).

1.3.4. 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:
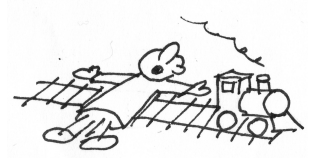
```
>>> letter = 'a'
>>> letter.upper() in lookup
True
>>> lookup[letter.upper()]
'A is for Amy who fell down the stairs.'
```

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".
```

An even shorter way to write that would use the `dict.get()` method:

```
1 def main():
2     args = get_args()
3     lookup = {line[0].upper(): line.rstrip() for line in args.file}
4
5     for letter in args.letter:
6         print(lookup.get(letter.upper(), f'I do not know "{letter}".')) ①
```

① If `lookup.get()` will return the value for `letter.upper()` or the phrase "I don't know..."

1.4. 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.

- Python's `pprint` module is used for "pretty printing" complex data structures.

1.5. Going Further

- Write a phonebook that reads a file and creates a dictionary from the names of your friends to their email or phone numbers.

- Create a program that uses a dictionary to count the number of times you see each word in a document.

- 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
```