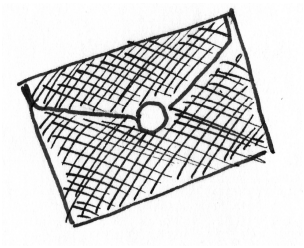
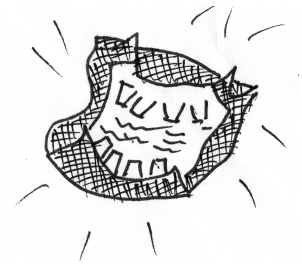
# Chapter 6: Howler: Working with files and STDOUT



In Harry Potter, a "Howler" is a nasty-gram that arrives by owl at Hogwarts. It will tear itself open, shout a blistering message at the recipient, and then combust. In this exercise, we're going to write a program that will transform text into a rather mild-mannered version of a Howler by MAKING ALL THE LETTERS UPPERCASE. The text that we'll process will be given as a single, positional argument.

For instance, if our program is given the input, "How dare you steal that car!", it should scream those back. Remember spaces on the command line delimit arguments, so multiple words need to be enclosed in quotes to be considered one argument:

```
$ ./howler.py 'How dare you steal that car!'
HOW DARE YOU STEAL THAT CAR!
```

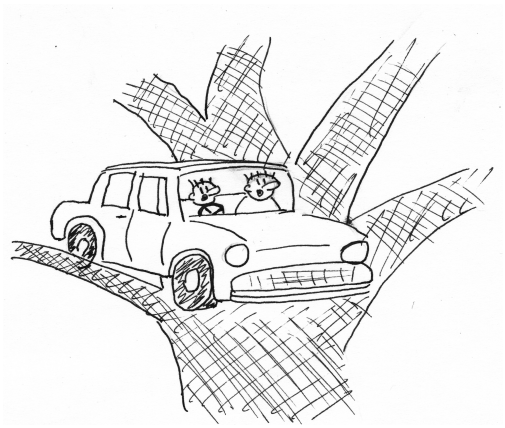


The argument to the program may also name a file, in which case we need to read the file for the input:

```
$ ./howler.py ../inputs/fox.txt
THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.
```

Our program will also accept an `-o` or `--outfile` option that names an output file into which the output text should be written. In that case, *nothing* will be printed on the command line:

```
$ ./howler.py -o out.txt 'How dare you steal that car!'
```



And there should now be a file called `out.txt` that has the output:

```
$ cat out.txt
HOW DARE YOU STEAL THAT CAR!
```

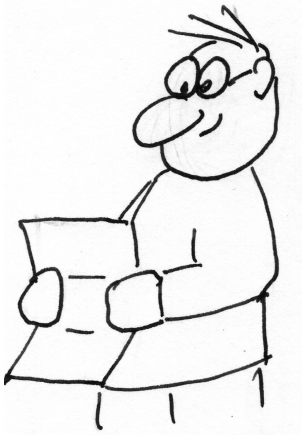
In this exercise, you will learn to:

- Accept text input from the command line or from a file.
- Change strings to uppercase.
- Print output either to the command line or to a file that needs to be created.

## Reading files

This will be our first exercise that will involve reading files. The argument to the program will be some text. That text might name an input file in which case you will open and read the file. If it's not the name of a file, then you'll use the text itself.

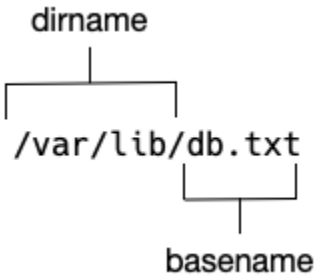
The built-in `os` (operating system) module has a method for detecting if a string is the name of a file. To use it, you must import the `os` module. For instance, there's probably not a file called "blargh" on your system:



```
>>> import os
>>> os.path.isfile('blargh')
False
```

The `os` module contains loads of useful submodules and functions. Consult the documentation at https://docs.python.org/3/library/os.html or use `help(os)` in the REPL. For instance, the `os.path` module has functions like `basename` and `dirname` for getting a file's name or directory from a path, for example:

```
>>> path = '/var/lib/db.txt'
>>> os.path.dirname(path)
'/var/lib'
>>> os.path.basename(path)
'db.txt'
```




In the `inputs` directory of the source code repo, there are several files. I'll use a file called `inputs/fox.txt`. Note you will need to be in the main directory of the repo for this to work:

```
>>> file = 'inputs/fox.txt'
>>> os.path.isfile(file)
True
```

Once you've determined that the argument is the name of a file, you must `open` it to `read` it. The return from `open` is a *file handle*, the thing we use to `read` the file. I usually call this variable `fh` to remind me that it's a file handle. If I have more than one open file handle like both input and output handles, I may call them `in_fh` and `out_fh`.

```
>>> fh = open(file)
```

If you try to `open` a file that does not exist, you'll get an exception.

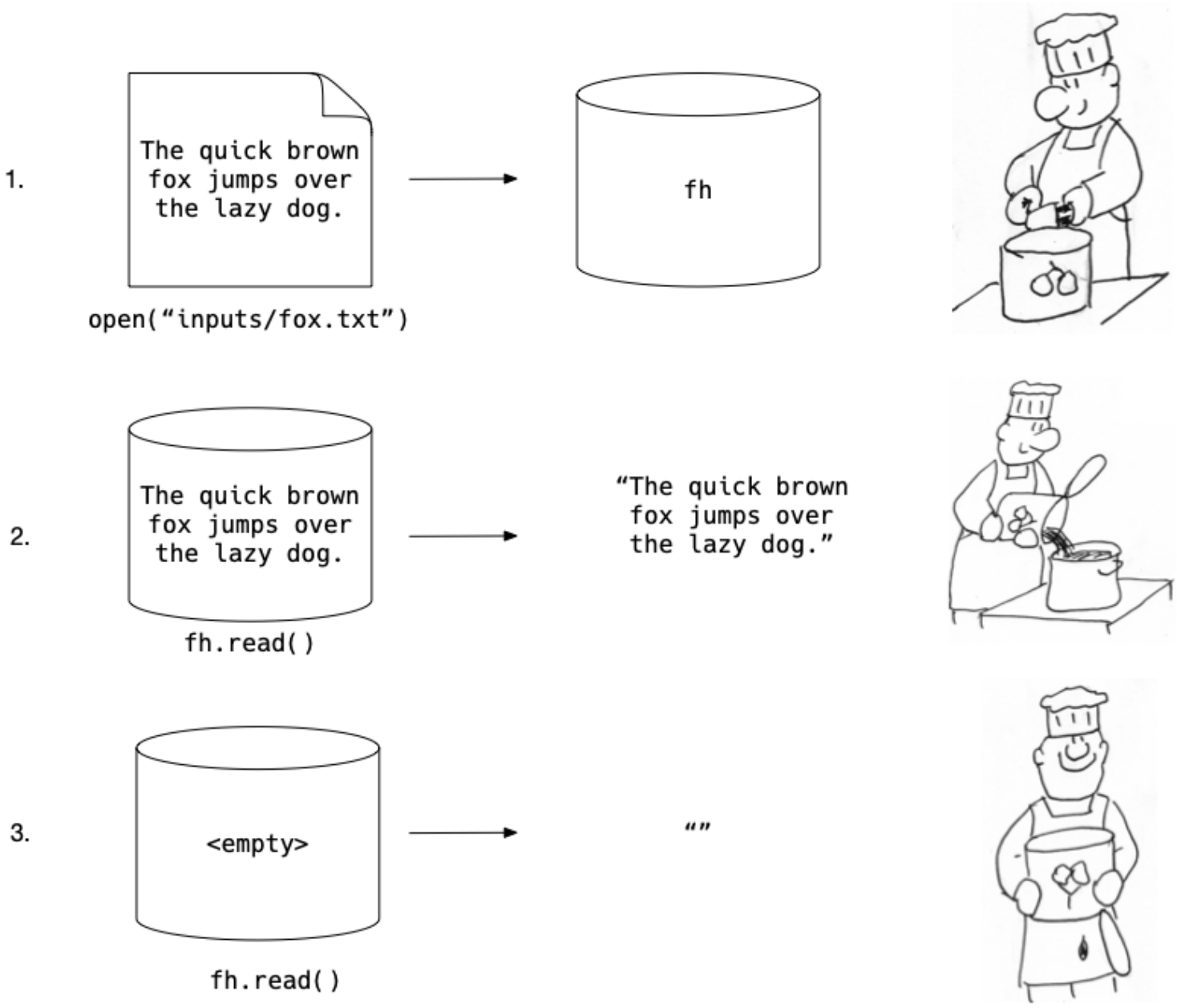
This is unsafe code:

```
>>> file = 'blargh'
>>> open(file)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'blargh'
```

Always check that the file exists!

```
>>> file = 'inputs/fox.txt'
>>> if os.path.isfile(file):
...     fh = open(file)
```

I think of the `file` here ("inputs/fox.txt") as the *name of the file* on disk. It's a bit like a can of tomatoes.




1. The file handle ( `fh` ) is a mechanism I can use to get at the contents of the file. To get at the tomatoes, we need to `open` the can.
2. The `fh.read` method returns what is inside the `file` . With the can opened, we can get at the contents.
3. Once the file handle has been read, there's nothing left.

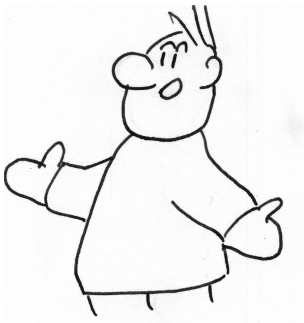
Let's see what `type` the `fh` is:

```
>>> type(fh)
<class '_io.TextIOWrapper'>
```

In computer lingo, "io" means "input/output." This is some sort of object that wraps I/O operations around text. You can use `help(fh)` (using the name of the variable itself) to read the docs on the `class TextIOWrapper` . The two methods you'll use quite often are `read` and `write` . Right now, we just care about `read` . Let's see what that gives us:

```
>>> fh.read()
'The quick brown fox jumps over the lazy dog.\n'
```

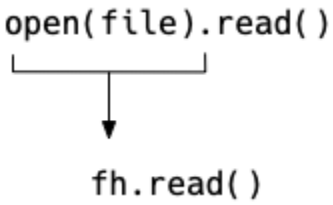
Do me a favor and execute that line one more time. What do you see?



```
>>> fh.read()
''
```

A file handle is different from something like a `str`. Once you `read` a file handle, it's empty! It's like pouring the tomatoes out of the can. Now the can is empty, and you can't empty it again.

We can actually compress the `open` and `read` into one line of code by *chaining* those methods together. The `open` returns a file handle that can be used for the call to `read`. Run this:




```
>>> open(file).read()
'The quick brown fox jumps over the lazy dog.\n'
```

And now run it again:

```
>>> open(file).read()
'The quick brown fox jumps over the lazy dog.\n'
```

Each time you `open` the `file`, you get a fresh file handle to `read`.

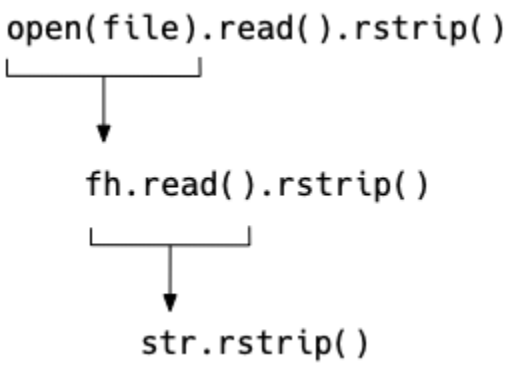
If you want to preserve the contents, you need to copy them into a variable.

```
>>> text = open(file).read()
>>> text
'The quick brown fox jumps over the lazy dog.\n'
```

The `type` of the result is a `str`:

```
>>> type(text)
<class 'str'>
```

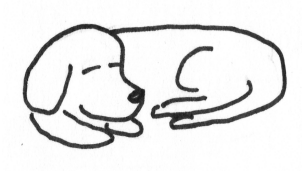
If I want, I can chain a `str` method onto the end of that. For instance, maybe I want to remove the trailing newline. The `str.rstrip` method will remove any whitespace (which includes newlines) from the *right* end of a string.




```
>>> text = open(file).read().rstrip()
>>> text
'The quick brown fox jumps over the lazy dog.'
```

Once you have your input text, whether it is from the command line or from a file, you need to UPPERCASE it. The `str.upper()` is probably what you want.

At this point, you have read your input, whether it is from the command-line or from a file.



## Writing files

The output of the program should either appear on the command line or be written to a file. Command-line output is also called called "standard out" or `STDOUT`. It's the *standard* or normal place for *output* to occur.

There's also an option to the program to write the output to a file, so let's look at how to do that. You still need to `open` a file handle, but we have to use an optional second argument, the string `'w'`, that instructs Python to open it for *writing*. Other modes include `'r'` for *reading* (the default) and `'a'` for _appending.

*Table 1. File writing modes*

| Mode | Meaning |
|---|---|
| w | write |
| r | read |
| a | append |

You can additionally describe the kind of content, whether `'t'` for *text* (the default) or `'b'` for _binary:

*Table 2. File content modes*

| Mode | Meaning |
|---|---|
| t | text |
| b | bytes |

You can combine these two tables like `'rb'` to read a binary file or `'at'` to append to a text file. Here we will use `'wt'` to write a text file. I'll call my variable `out_fh` to remind me that this is the "output file handle":

```
>>> out_fh = open('out.txt', 'wt')
```

If the file does not exist, it will be created. If the file does exist, then it will be *overwritten*. It's possible to `open` in a mode that will append text to an existing file. For this exercise, `'wt'` will suffice.

You can use the `write` method of the file handle to put text into the file. Whereas the `print` method will append a newline (`\n`) unless you instruct it not to, the `write` method will *not* add a newline, so you have to explictly add one.

If you use the `fh.write` method in the REPL, you will see that it return the number of bytes written. Here each character is a byte, and remember that the newline (`\n`) is included:

```
>>> out_fh.write('this is some text\n')
18
```

You can check that this is correct:

```
>>> len('this is some text\n')
18
```

Most code tends to ignore this return value; that is, we don't bother to capture the results into a variable or check that we got a non-zero return. If `write` fails, there's usually some much bigger problem with your system. It's also possible to use the `print` function with the optional `file` argument. Notice that I don't include a newline with `print` because it will add one:

```
>>> print('this is some more text', file=out_fh)
```

When you are done writing to a file handle, you should `close` it so that Python can clean up the file and release the memory associate with it. This function returns no value:

```
>>> out_fh.close()
```

We can verify that our text made it:

```
>>> open('out.txt').read()
'this is some text\nthis is some more text\n'
```

When you `print` on an open file handle, the text will be appended to any previously written data. Look at this code, though:

```
>>> print("I am what I am an' I'm not ashamed", file=open('hagrid.txt', 'wt'))
```

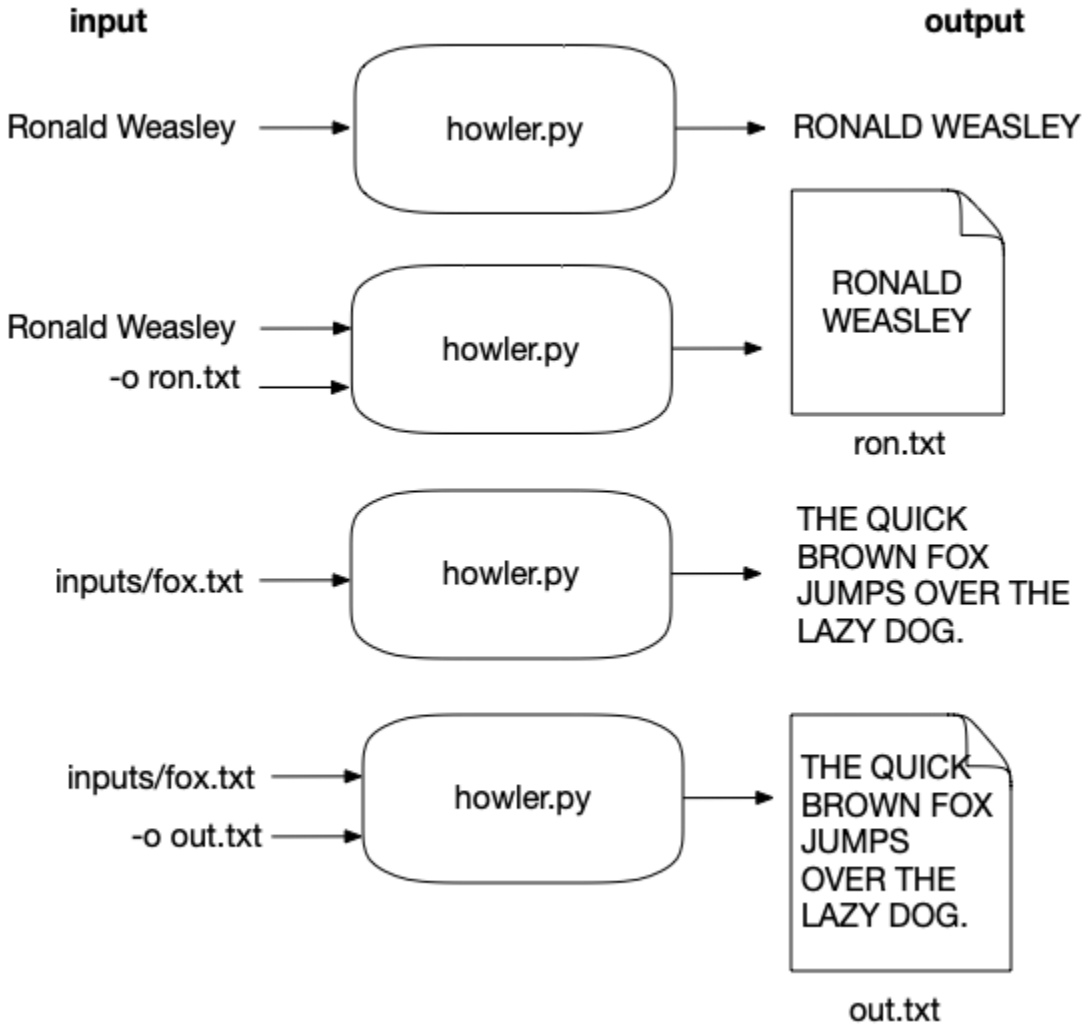
If you run that line twice, will the file called "hagrid.txt" have the line once or twice? Let's find out:

```
>>> open('hagrid.txt').read()
"I am what I am an' I'm not ashamed\n"
```

Just once! Why is that? Remember, each called `open` gives us a new file handle, so calling `open` twice results in new file handles. Both are opened in *write* mode, so that existing data is *overwritten*. It's important to understand how to `open` files properly or you may end up erasing important data files!

## Writing `howler.py`

Here is a string diagram showing the overview of the program and some example inputs and outputs:




When run with no arguments, it should print a short usage:

```
$ ./howler.py
usage: howler.py [-h] [-o str] STR
howler.py: error: the following arguments are required: STR
```

When run with `-h` or `--help`, the program should print a longer usage statement:

```
$ ./howler.py -h
usage: howler.py [-h] [-o str] str

Howler (upper-cases input)

positional arguments:
  str                   Input string or file

optional arguments:
  -h, --help            show this help message and exit
  -o str, --outfile str
                        Output filename (default: )
```

If the argument is a regular string, it should uppercase that:

```
$ ./howler.py 'How dare you steal that car!'
HOW DARE YOU STEAL THAT CAR!
```

If the argument is the name of a file, it should uppercase the *contents of the file*:

```
$ ./howler.py ../inputs/fox.txt
THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.
```

If given an `--outfile` filename, the uppercased text should be written to the indicated file and nothing should be printed to STDOUT:

```
$ ./howler.py -o out.txt ../inputs/fox.txt
$ cat out.txt
THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.
```

Hints:

- Start with `new.py` and alter the `get_args` section until your usage statements match the ones above.
- Run the test suite and try to pass just the first test that handles text on the command line and output to `STDOUT`.
- The next test is to see if you can write the output to a given file. Figure out how to do that.
- The next test is for reading input from a file. Then work on that. Don't try to do all the things at once!
- There is a special file handle that always exists called "standard out" (often `STDOUT`). If you `print` without a `file` argument, then it defaults to `sys.stdout`. You will need to `import sys` in order to use it.

Be sure you really try to write the program and pass all the tests before moving on to read the solution! If you get stuck, maybe whip up a batch of Polyjuice Potion and freak out your friends.

## Solution

```
#!/usr/bin/env python3
"""Howler"""

import argparse
import os
import sys


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

    parser = argparse.ArgumentParser(
        description='Howler (upper-case input)',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('text', metavar='str', help='Input string or file')  1

    parser.add_argument('-o',  2
                        '--outfile',
                        help='Output filename',
                        metavar='str',
                        type=str,
                        default='')

    args = parser.parse_args()    3

    if os.path.isfile(args.text):  4
        args.text = open(args.text).read().rstrip()  5

    return args  6


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

    args = get_args()                       7
    out_fh = open(args.outfile, 'wt') if args.outfile else sys.stdout  8
    out_fh.write(args.text.upper() + '\n')  9
    out_fh.close()                          10


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

1 The `text` argument is a string that may be the name of a file.

2 The `--outfile` option is also a string that names a file.

3 Parse the command-line arguments into the variable `args` so that we can manually check the `text` argument.

4 Check if `args.text` names an existing file.

5 If so, overwrite the value of `args.text` with the results of reading the file.

6 Now that we've fixed up the `args`, we can `return` them to the caller.

7 Call `get_args` to get the arguments to the program.

8 Use an `if` expression to choose either `sys.stdout` or a newly opened file handle to write the output.

9 Use the opened file handle to write the output converted to `upper`.

10 Close the file handle.

## Discussion

How did it go for you this time? I hope you avoided Snape's office. You really don't want more Saturday detentions.

## Defining the arguments

The `get_args` function, as always, is the first. Here I define two arguments. The first is a positional `text` argument. Since it may or may not name a file, all I can know is that it will be a string.

```
parser.add_argument('text', metavar='str', help='Input string or file')
```

The other argument is an option, so I give it a short name of `-o` and a long name of `--outfile`. The default `type` for all arguments is `str`, but I go ahead and state that explictly. The `default` value is the empty string. I could just as easily use the special `None` type which is also the default value, but I prefer to use a defined argument like the empty string.

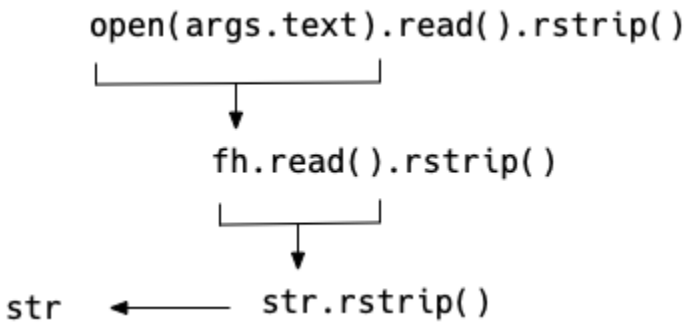
```
parser.add_argument('-o',
                    '--outfile',
                    help='Output filename',
                    metavar='str',
                    type=str,
                    default='')
```

### Reading input from a file or the command line

This is a deceptively simple program that demonstrates a couple of very important elements of file input and output. The `text` input might be a plain string or it might be the name of a file. This pattern will come up repeatedly in this book:

```
if os.path.isfile(args.text):
    args.text = open(args.text).read().rstrip()
```

The function `os.path.isfile` will tell us if there is a file with the name in `text`. If that returns `True`, then we can safely `open(file)` to get a *file handle* which has a method called `read` which will return *all* the contents of the file. This is usually safe, but be careful if you write a program that could potentially read gigantic files. For instance, in my day job we regularly deal with files with sizes in the 10s to 100s of gigabytes, so I would need to ensure my system has more memory than the size of the file!




The result of `open(file).read()` is a `str` which itself has a method called `rstrip` that will return a copy of the string *stripped* of the whitespace off the *right* side of the string (hence the name `rstrip`). The longer way to write the above would be:

```
if os.path.isfile(text):
    fh = open(text)
    text = fh.read()
    text = text.rstrip()
```

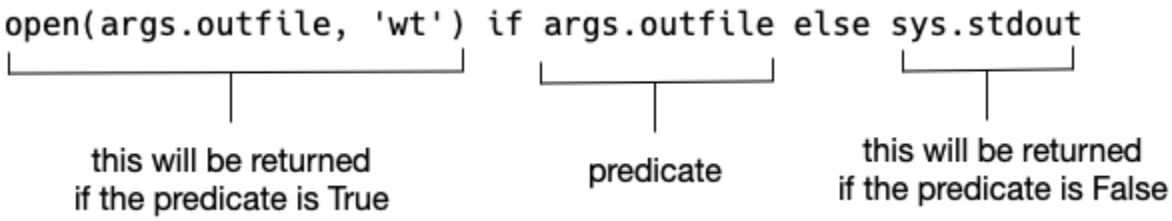
In my version, I choose to handle this inside the `get_args` function. This is the first time I'm showing you that you can intercept and alter the arguments before passing them on to `main`. We'll use this idea quite a bit in more exercises. I like to do all the work to validate the user's arguments inside `get_args`. I could just as easily do this in `main` after the call to `get_args`, so this is entirely a style issue.

## Choosing the output file handle

The line decides where to put the output of our program

```
out_fh = open(args.outfile, 'wt') if args.outfile else sys.stdout
```

The `if` expression will open `args.outfile` for writing text (`wt`) if the user provided that argument; otherwise, we will use `sys.stdout` which is a file handle to STDOUT. Note that we don't have to call `open` on `sys.stdout` because it is always there and always open for business.




## Printing the output

To get uppercase, we can use the `text.upper` method, then we need to find a way to print it to the output file handle. I chose to do:

```
out_fh.write(text.upper())
```

But you could also do:

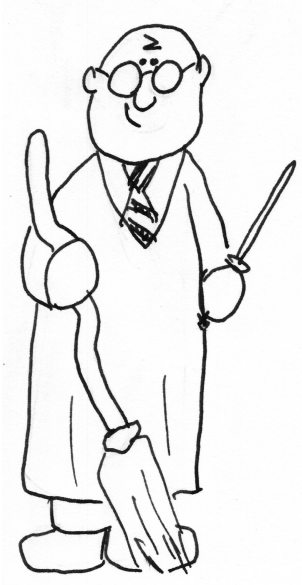
```
print(text.upper(), file=out_fh)
```

Finally I need to close the file handle with `out_fh.close()`.

# Review

- To `read` or `write` to files, you must `open` them.
- The default mode for `open` is for reading a file.
- To write a text file, you must use `'wt'` as the second argument to `open`.
- By default, you `write` text to a file handle. You must use the `'b'` flag to indicate that you want to write binary data.

- The `os.path` module contains many useful functions such as `isfile` that will tell you if a file exists by the given name.
- `STDOUT` (standard output) is always available via the special `sys.stdout` file handle which is always open.
- The `print` function takes an optional `file` argument of where to put the output. That argument must be an open file handle such as `sys.stdout` (the default) or the result of `open`.



## Going Further

- Add a flag that will lowercase the input instead.
- Alter the program handle multiple input files
- Change `--outfile` to `--outdir` and write each input file to the same file name in the output directory.

Last updated 2020-01-14 20:37:36 -0700