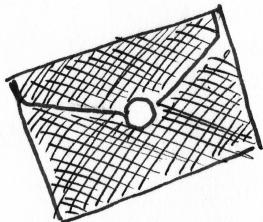


1. Howler: Working with files and STDOUT



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 back "HOW DARE YOU STEAL THAT CAR!" 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.
- Make plain text behave like a file handle.

1.1. 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.

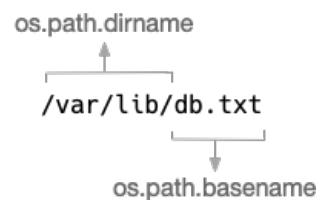


The built-in `os` (operating system) module has a method for detecting if a string is the name of a file. To use it, you must import the `os` module. For instance, there's probably not a file called "blargh" on your system:

```
>>> 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. The `os.path` module has handy functions like `basename` and `dirname` for getting a file's name or directory from a path, for example:

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



In the top level of the repository is a directory called `inputs` that contains several files we'll use for

many of the exercises. Here 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* and is what 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`^[1].

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

WARNING

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

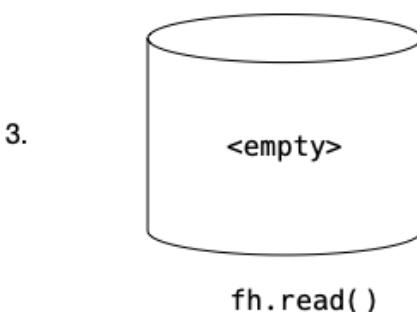
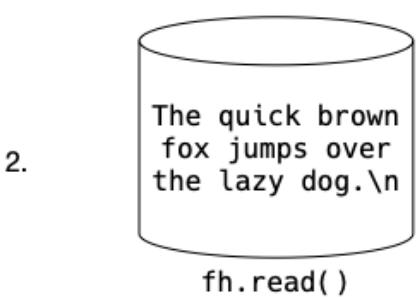
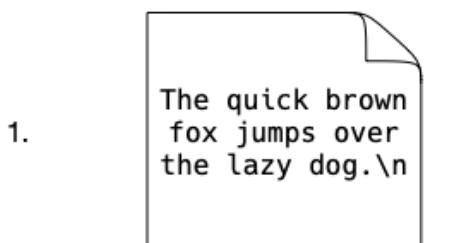
This is unsafe code:

```
>>> 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. ^[2]

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

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

In computer lingo, "io" means "input/output." The `fh` object is something that handles I/O operations. 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 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?

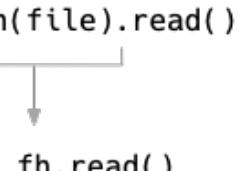
```
>>> fh.read()  
''
```



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`.

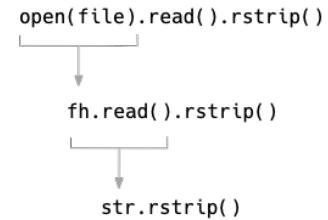
If you want to preserve the contents, you need to copy them into a variable.

```
>>> 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.

1.2. 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 "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 5. 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 5. 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* which means that all the previous data will be lost! If you don't want an existing file to be lost, you can use the `os.path.isfile` function we saw earlier to first check if a file exists and perhaps `open` in the "append" mode instead. For this exercise, we'll use the '`wt`' mode to write text.

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 explicitly add one.

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

```
>>> 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 associated with it. This function returns no value:

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

We can verify that our text made it. Note that the newline appears here as `\n`. You need to `print` the string for it to create an actual newline:

```
>>> 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:

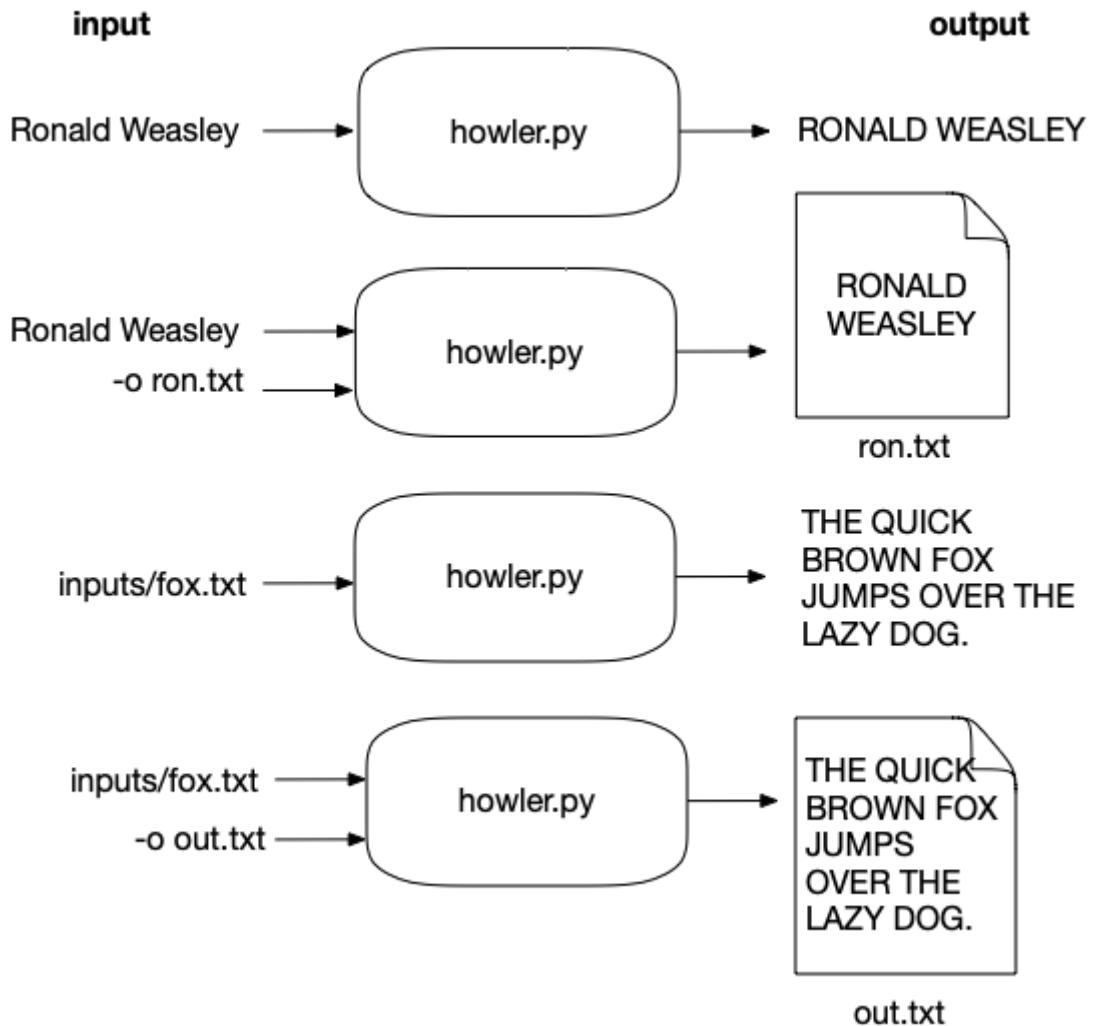
```
>>> open('hagrid.txt').read()
"I am what I am an' I'm not ashamed\n"
```

Just once! Why is that? Remember, each called `open` gives us a new file handle, so calling `open` twice results in new file handles. Each time you run that code, the file is opened anew in *write* mode and the existing data is *overwritten*. So as to avoid confusion, I would recommend instead you write code more along these lines:

```
1 fh = open('hagrid.txt', 'wt')
2 fh.write("I am what I am an' I'm not ashamed.\n")
3 fh.close()
```

1.3. 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 prints the uppercased 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. Don't try to pass all the tests 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.

1.4. Solution

```
1 #!/usr/bin/env python3
2 """Howler"""
3
4 import argparse
5 import os
6 import sys
7
8
9 # -----
10 def get_args():
11     """get command-line arguments"""
12
13     parser = argparse.ArgumentParser(
14         description='Howler (upper-case input)',
15         formatter_class=argparse.ArgumentDefaultsHelpFormatter)
16
17     parser.add_argument('text', metavar='str', help='Input string or file') ①
18
19     parser.add_argument('-o', ②
20                         '--outfile',
21                         help='Output filename',
22                         metavar='str',
23                         type=str,
24                         default='')
25
26     args = parser.parse_args()      ③
27
28     if os.path.isfile(args.text): ④
29         args.text = open(args.text).read().rstrip() ⑤
30
31     return args ⑥
32
33
34 # -----
35 def main():
36     """Make a jazz noise here"""
37
38     args = get_args()            ⑦
39     out_fh = open(args.outfile, 'wt') if args.outfile else sys.stdout ⑧
40     out_fh.write(args.text.upper() + '\n') ⑨
41     out_fh.close()             ⑩
42
43
44 # -----
45 if __name__ == '__main__':
46     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.

1.5. Discussion

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

1.5.1. 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.

NOTE

The order of positional parameters *relative to each other* is important. The first positional parameter you define will handle the first positional argument provided. It's not important, however, to define positional parameters before or after options and flags. You can declare those in any order you like.

```
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`. Even though the default `type` for all arguments is `str`, I like to state this explicitly. 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.

```
1 parser.add_argument('-o',
2                     '--outfile',
3                     help='Output filename',
4                     metavar='str',
5                     type=str,
6                     default='')
```

1.5.2. 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:

```
1 if os.path.isfile(args.text):
2     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.

WARNING

You should be aware that `fh.read()` will return the *entire file* as a single string. Your computer must have more memory available than the size of the file. For all the programs in this book, that will safe as the files are small. In my day job, I regularly deal with gigabyte-sized files, so calling `fh.read()` would likely crash my program if not my whole system because I would exceed my available memory!

The result of `open(file).read()` is a `str` which has a method called `rstrip` that will return a copy of the string *stripped* of the whitespace off the *right* side. I call this so that the input text will look the same whether is comes from a file or directly from the command line. That is, when you provide the input text directly on the command line, you have to press `<Enter>` to terminate the command. That `<Enter>` is a newline, and the operating system automatically removes it before passing it to the program. I want the text to look the same whether it comes directly from the command line or from a file.

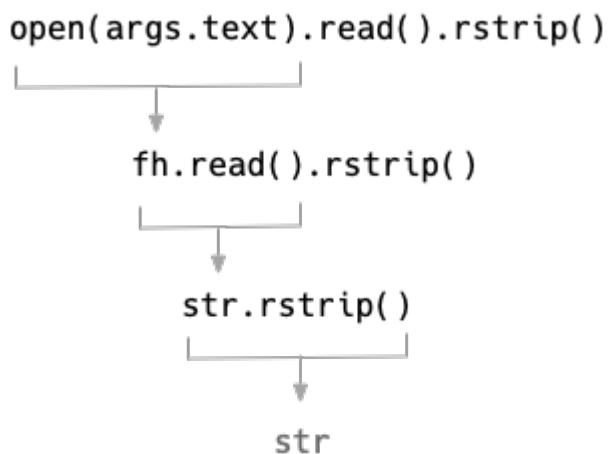


Figure 5. 1. The `open` function returns a file handle (`fh`). The `fh.read()` function returns a `str`. The `str.rstrip()` function returns a new `str` with the whitespace removed from the right side. All these functions can be chained together.

The longer way to write the above would be:

```
1 if os.path.isfile(text):
2     fh = open(text)
3     text = fh.read()
4     text = text.rstrip()
5     fh.close()
```

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.

1.5.3. Choosing the output file handle

This line decides where to put the output of our program:

```
1 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 available and open for business.



1.5.4. 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:

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

But you could also do:

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

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

1.5.5. A low-memory version

There is a potentially serious problem waiting to bite us in this program. In the `get_args`, we're reading the entire file into memory with this line:

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

We could, instead, only `open` the file:

```
1 if os.path.isfile(args.text):  
2     args.text = open(args.text)
```

And later read it line-by-line:

```
1 for line in args.text:  
2     out_fh.write(line.upper())
```

The problem, though, is how to handle the times when the `text` argument is actually text and not the name of a file. The `io` (input-output) module in Python has a way to represent text as *stream*:

```
>>> import io  
(1)  
>>> text = io.StringIO('foo\nbar\nbaz\n')  
(2)  
>>> for line in text:  
(3)  
...     print(line, end='')  
(4)  
  
foo  
bar  
baz
```

- ① Import the `io` module.
- ② Use the `io.StringIO` function to turn the given `str` value into something we can treat like an open file handle.
- ③ Use a `for` loop to iterate the "lines" of text by separated by newlines.
- ④ Print the `line` using the `end=''` option to avoid having 2 newlines.

This is the first time we're seeing that we can treat a regular string value as if it were a generator of values similar to a file handle. This is a particularly useful technique for testing any code that needs to read an input file. You can use an `io.StringIO` as a "mock" file handle so that your code doesn't have to read an *actual* file, just a given value that can produce "lines" of text!

To make this work, we can change how we handle the `args.text` like so:

```

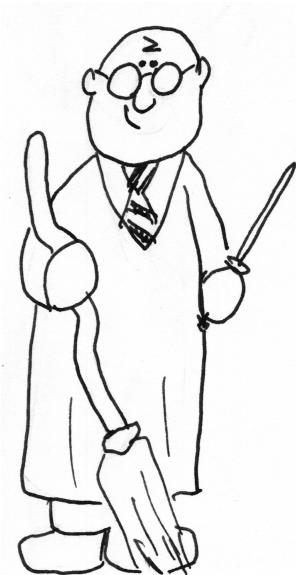
1 #!/usr/bin/env python3
2 """Low-memory Howler"""
3
4 import argparse
5 import os
6 import io
7 import sys
8
9
10 # -----
11 def get_args():
12     """get command-line arguments"""
13
14     parser = argparse.ArgumentParser(
15         description='Howler (upper-cases input)',
16         formatter_class=argparse.ArgumentDefaultsHelpFormatter)
17
18     parser.add_argument('text', metavar='str', help='Input string or file')
19
20     parser.add_argument('-o',
21                         '--outfile',
22                         help='Output filename',
23                         metavar='str',
24                         type=str,
25                         default='')
26
27     args = parser.parse_args()
28
29     if os.path.isfile(args.text):          ①
30         args.text = open(args.text)        ②
31     else:
32         args.text = io.StringIO(args.text + '\n') ③
33
34     return args
35
36
37 # -----
38 def main():
39     """Make a jazz noise here"""
40
41     args = get_args()
42     out_fh = open(args.outfile, 'wt') if args.outfile else sys.stdout
43     for line in args.text:               ④
44         out_fh.write(line.upper())      ⑤
45     out_fh.close()
46
47
48 # -----
49 if __name__ == '__main__':
50     main()

```

- ① Check if `args.text` is a file.
- ② If so, replace `args.text` with the file handle created by opening the file named by it.
- ③ Otherwise, replace `args.text` with an `io.StringIO` value that will act like an open file handle.
Note that we need to add a newline to the text so that it will look like the lines of input coming from an actual file.
- ④ Read the input (whether `io.StringIO` or a file handle) line-by-line.
- ⑤ Handle the line.

1.6. 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`.



1.7. Going Further

- Add a flag that will lowercase the input instead. Maybe call it `--ee` for the poet e e cummings who liked to write poetry devoid of uppercase letters.
- 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.

[1] Per <https://www.python.org/dev/peps/pep-0008/#function-and-variable-names>, function and variable "names should be lowercase, with words separated by underscores as necessary to improve readability."

[2] You can use `fh.seek(0)` to reset the file handle to the beginning if you want to read it again.