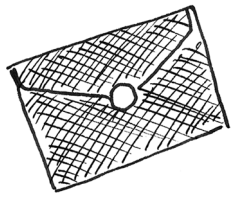
5 Howler: Working with files and STDOUT



In the Harry Potter stories, 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.

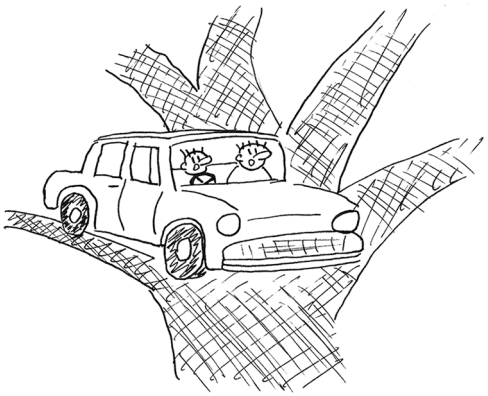
For instance, if our program is given the input, "How dare you steal that car!" it should scream back "HOW DARE YOU STEAL THAT CAR!" Remember spaces on the command line delimit arguments, so multiple words need to be enclosed in quotes to be considered one argument:

```
$ ./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!'
```

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

5.1 Reading files

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

The built-in `os` (operating system) module has a method for detecting whether 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, `os.path.basename()` and `os.path.dirname()` can return a file's name or directory from a path, respectively (see figure 5.1):

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

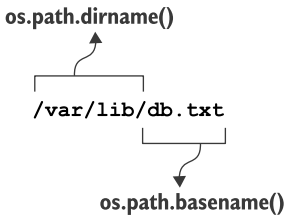


Figure 5.1 The `os` module has handy functions like `os.path.dirname()` and `os.path.basename()` for getting parts of file paths.

In the top level of the GitHub source repository, there 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 that 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*. 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)
```

> **NOTE** Per PEP 8 (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."

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)
```

We will use the `fh.read()` method to get the contents of the file. It might be helpful to think of a file as a can of tomatoes. The file's name, like "inputs/fox.txt," is the label on the can, which is not the same as the *contents*. To get at the text inside (or the "tomatoes"), we need to *open* the can.

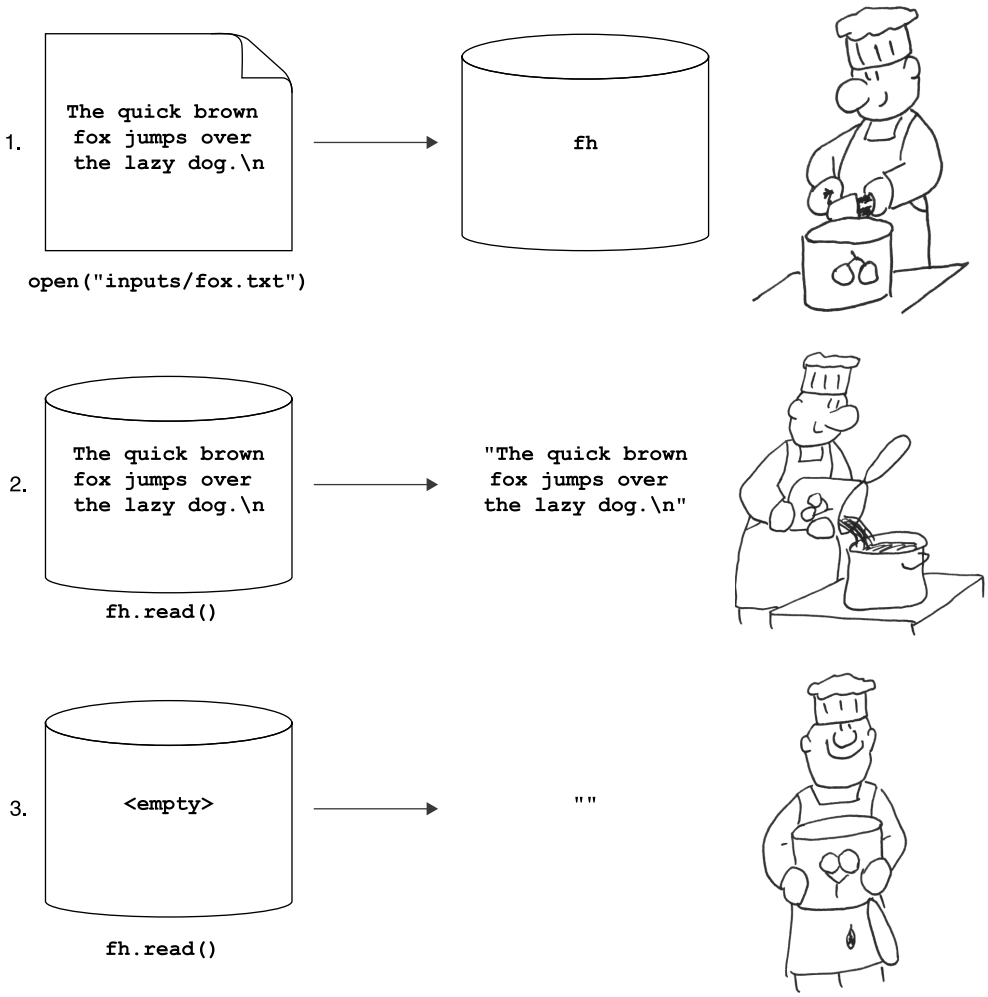


Figure 5.2 A file is a bit like a can of tomatoes. We have to open it first so that we can read it, after which the file handle is exhausted.

Take a look at figure 5.2:

1. The file handle (`fh`) is a mechanism we 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.

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

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

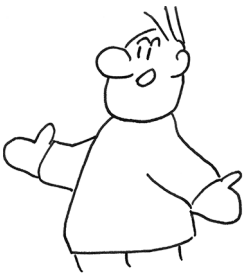
```
>>> type(fh)
<class '_io.TextIOWrapper'>
```

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 that the can is empty, you can't empty it again.

We can actually compress `open()` and `fh.read()` into one line of code by *chaining* those methods together. The `open()` method returns a file handle that can be used for the call to `fh.read()` (see figure 5.3). Run this:

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

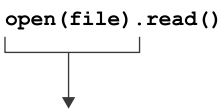


Figure 5.3 The `open()` function returns a file handle, so we can chain it to a call to `read()`.

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 we want to preserve the contents, we'll 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 you want, you can chain any `str` method onto the end of that. For instance, maybe you want to remove the trailing newline. The `str.rstrip()` method will remove any whitespace (which includes newlines) from the *right* end of a string (see figure 5.4).

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

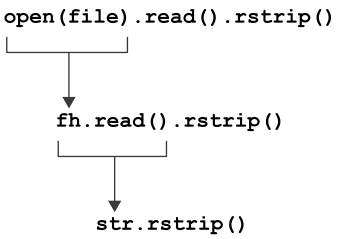


Figure 5.4 The `open()` method returns a file handle, to which we chain `read()`, which returns a string, to which we chain the `str.rstrip()` call.

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()` method is probably what you want.

5.2 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.) Now let's look at how to write the output to a file.

We still need to `open()` a file handle, but we have to use an optional second argument, the string `'w'`, to instruct Python to open it for *writing*. Other modes include `'r'` for *reading* (the default) and `'a'` for *appending*, as listed in table 5.1.

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*, as listed in table 5.2.

Table 5.2 File-content modes

Mode	Meaning
`t`	Text
`b`	Bytes

You can combine the values in 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, 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 you saw earlier to first check if the file exists, and perhaps use `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()` function 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 `out_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 usually bother to capture the results in a variable or check that we got a nonzero return. If `write()` fails, there's usually some much bigger problem with your system.

You can also use the `print()` function with the optional `file` argument. Notice that I don't include a newline with `print()` because it will add one. This method returns `None`:

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

When you are done writing to a file handle, you should `out_fh.close()` it so that Python can clean up the file and release the memory associated with it. This method also returns `None`:

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

Let's check if the lines of text we printed to our out.txt file made it by opening the file and reading it. Note that the newline appears here as `\n`. We 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 we `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 call to `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*. To avoid confusion, I recommend you write code more along these lines:

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

5.3 Writing howler.py

You'll need to create a program called howler.py in the 05_howler directory. You can use the new.py program for this, copy template.py, or start however you prefer. Figure 5.5 is a string diagram showing an overview of the program and some example inputs and outputs.

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

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

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

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

Howler (upper-cases input)

positional arguments:
  text                  Input string or file

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

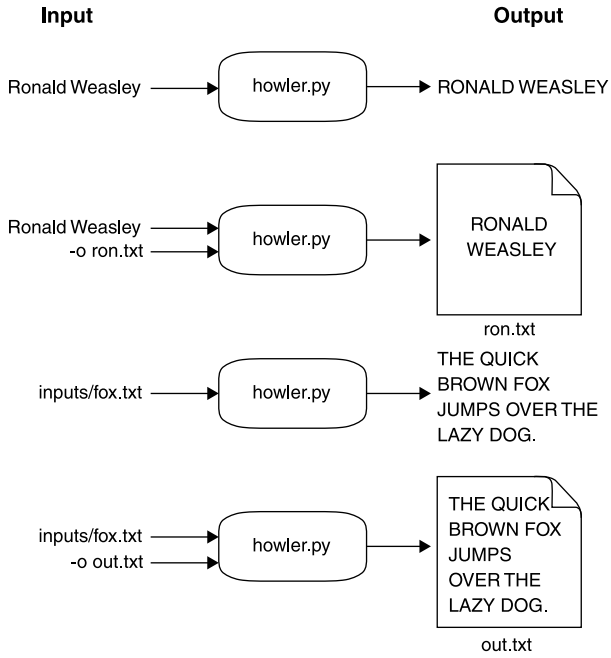


Figure 5.5 A string diagram showing that our howler.py program will accept strings or files as inputs and possibly an output filename.

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.
```

Here are a few 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, 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.

5.4 Solution

Here is a solution that will pass the tests. It's rather short because Python allows us to express some really powerful ideas very concisely.

```
#!/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='text',
                        type=str,
                        help='Input string or file')

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

    args = parser.parse_args()

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

    return args
```

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

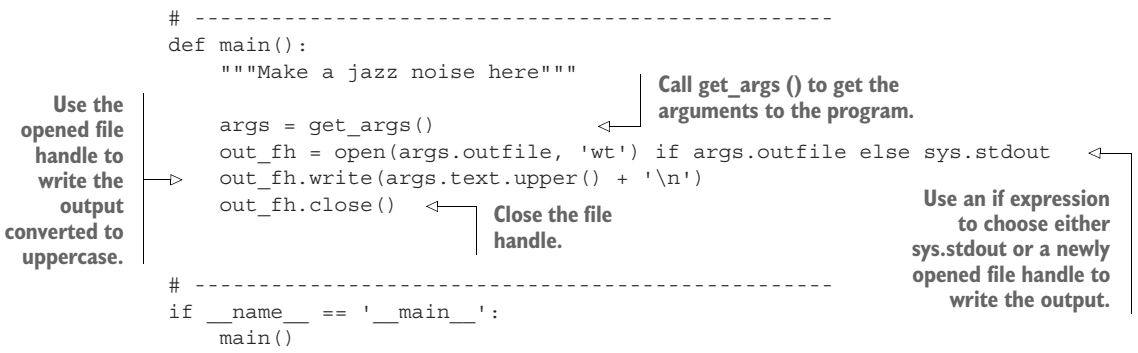
The --outfile option is also a string that names a file.

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

Check if args.text is the name of an existing file.

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

Return the arguments to the caller.




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

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

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

5.5 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.

5.5.1 Defining the arguments

The `get_args()` function, as always, comes 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='text',
                    type=str,
                    help='Input string or file')
```

> **NOTE** If you define multiple positional parameters, their order *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.

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.

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

5.5.2 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:

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

The `os.path.isfile()` function will tell me if there is a file with the specified name in `text`. If that returns `True`, I can safely `open(file)` to get a file handle, which has a method called `read` and 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, you will be 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 `str.rstrip()` that will return a copy of the string *stripped* of any whitespace on the *right* side (see figure 5.6). I call this so that the input text will look the same whether it comes from a file or directly from the command line. 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.

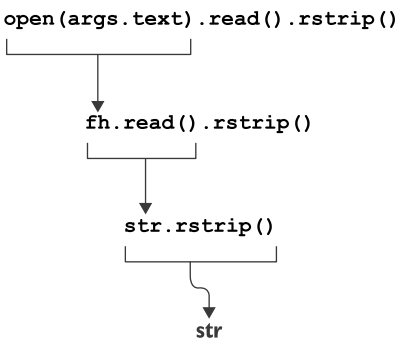


Figure 5.6 The `open()` function returns a file handle (`fh`). The `fh.read()` function returns a `str`. The `str.rstrip()` function returns a new `str` with the whitespace removed from the right side. All these functions can be chained together.

The longer way to write the preceding statement would be

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

In my version, I chose to handle this inside the `get_args()` function. This is the first time I've shown you that you can intercept and alter arguments before passing them on to `main()`. We'll use this idea quite a bit in later 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.

5.5.3 Choosing the output file handle

The following line decides where to put the output of the 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, it will use `sys.stdout`, which is a file handle to `STDOUT`. Note that I don't have to call `open()` on `sys.stdout` because it is always available and open for business (figure 5.7).

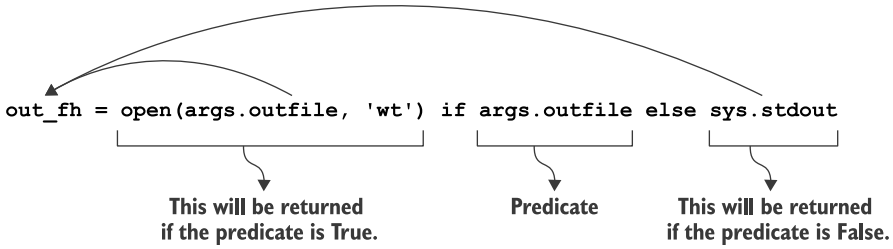


Figure 5.7 An `if` expression succinctly handles a binary choice. Here we want the output file handle to be the result of opening the outfile argument if present; otherwise, it should be `sys.stdout`.

5.5.4 Printing the output

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

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

Alternatively, you could do this:

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

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

5.5.5 A low-memory version

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

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

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

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

Later we could read it line by line:

```
for line in args.text:
    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 a *stream*:

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

Import the io module.

Use the io.StringIO() function to turn the given str value into something we can treat like an open file handle.

Use a for loop to iterate through the "lines" of text separated by newlines.

Print the line using the end=" option to avoid having two newlines.

This is the first time you're seeing that you 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 the return from `io.StreamIO()` 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 `args.text`, like so:

```
#!/usr/bin/env python3
"""Low-memory Howler"""

import argparse
import os
import io
import sys


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

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

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

    parser.add_argument('-o',
                        '--outfile',
                        help='Output filename',
```

```
                        metavar='str',
                        type=str,
                        default='')

    args = parser.parse_args()

    if os.path.isfile(args.text):
        args.text = open(args.text)
    else:
        args.text = io.StringIO(args.text + '\n')

    return args


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

    args = get_args()
    out_fh = open(args.outfile, 'wt') if args.outfile else sys.stdout
    for line in args.text:
        out_fh.write(line.upper())
    out_fh.close()


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

Check if args.text is a file.

If it is, replace args.text with the file handle created by opening the file.

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.

Read the input (whether io.StringIO() or a file handle) line by line.

Process the line as before.

5.6 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 to handle multiple input files. Change `--outfile` to `--outdir`, and write each input file to the same filename in the output directory.

Summary

- To read or write files, you must first `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()`.
- Text is the default type of data that you `write()` 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 `os.path.isfile()`, that will tell you if a file exists with a 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 specifying where to put the output. That argument must be an open file handle, such as `sys.stdout` (the default) or the result of `open()`.