> *Brave Sir Robin ran away, bravely ran away away. When danger reared his ugly head, he bravely turned his tail and fled. Yes, brave Sir Robin turned about, he turned his tail, he chickened out. Bravely taking to his feet, he beat a very brave retreat. A brave retreat by Sir Robin.*

I/O

Files

It is very common in programming that input is received from a file instead of, or as well as, user input. It is therefore necessary to be able to open, read and save to files. Python has a **file object** that enables us to perform these tasks.

Reading a File

To show how to use the file object in Python here are a few examples using the text.txt file.

```
>>> f = open('text.txt', 'r')
>>> type(f)
<class '_io.TextIOWrapper'>
>>> f.read()
'Python is fun,\nit lets me play with files.\nI like playing with files,\nI can
do some really fun stuff.\n\nI like Python!\n'
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    f.read()
ValueError: I/O operation on closed file
>>>
```

The first line shows opening a file using the open function and assigns it to the variable `f`. The open statement takes two string parameters; one is the name of the file to be opened. (The file name may include the path to the file if it is in a different directory to the Python program.) The other is the open mode, we will discuss this soon. The second example shows that `f` is a file object. The third example shows one of the multiple methods for getting the data from the file. The fourth example shows closing the file, **it is important to close a file after the program is finished with it**. The last example shows that it is not possible to perform operations on a closed file.

> **open Syntax**
>
> open(*filename*, *mode*)
>
> Where `filename` is a string which is the name of the file and `mode` is a string indicating the opening mode.
>
> **Semantics**
>
> `mode` is usually 'r' or 'w' for read or write respectively. `filename` can either be relative to the current working directory (i.e the directory or folder from which the program is executed) or can contain the full path to the file.

More Reading

As before, dir(f), will list all the methods available for use with files. The more useful ones for us are the ones that allow us to read from and write to the file. Below are some examples of the read methods we have not seen yet.

```
>>> f = open('text.txt', 'r')
>>> dir(f)
['_CHUNK_SIZE', '__class__', '__del__', '__delattr__', '__dict__', '__dir__',
'__doc__', '__enter__', '__eq__', '__exit__', '__format__', '__ge__',
'__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__', '__iter__',
'__le__', '__lt__', '__ne__', '__new__', '__next__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '_checkClosed', '_checkReadable', '_checkSeekable',
'_checkWritable', '_finalizing', 'buffer', 'close', 'closed', 'detach',
'encoding', 'errors', 'fileno', 'flush', 'isatty', 'line_buffering', 'mode',
'name', 'newlines', 'read', 'readable', 'readline', 'readlines', 'seek',
'seekable', 'tell', 'truncate', 'writable', 'write', 'writelines']
>>> f.readline()
'Python is fun,\n'
>>> f.readline()
'it lets me play with files.\n'
>>> f.readline()
'I like playing with files,\n'
>>> f.readline()
'I can do some really fun stuff.\n'
>>> f.readline()
'\n'
>>> f.readline()
'I like Python!\n'
>>> f.readline()
''
>>> f.close()
>>> f = open('text.txt', 'r')
>>> f.readlines()
['Python is fun,\n', 'it lets me play with files.\n', 'I like playing with
files,\n', 'I can do some really fun stuff.\n', '\n', 'I like Python!\n']
>>> f.close()
>>> f = open('text.txt', 'r')
>>> for line in f :
        print(line)

Python is fun,

it lets me play with files.

I like playing with files,

I can do some really fun stuff.



I like Python!

>>> f.close()
>>>
```

The first few examples are the use of `readline`, it returns one line of the file each time it is called. When there are no lines left `readline` returns an empty string. The next example is of `readlines` (notice the extra *s*). `readlines` returns a list with each element in the list being a string of each line in the file. Notice how each line of the file ends in a `\n`. This is the new line character; it is the character that is, invisibly, inserted when we hit the Enter (return) key. The last example is the use of a `for` loop to directly iterate over the file one line at a time. Notice how there is an extra line in between each line that we print. This is caused by the new line character, `\n`. print interprets the new line and inserts it, but print also inserts its own new line, therefore we end up with two new lines printed.

Writing

Now let's have a look at a method for writing to a file:

```
>>> f = open("lets_write.txt", "w")
>>> f.write("I'm writing in the file\n")
25
>>> text = ['look', 'more', 'words']
>>> for word in text :
        f.write(word)

>>> f.close()
>>>
```

Now if we look at the file it should look like lets_write1.txt. Notice how we need to put in newlines manually when required, they are not automatically inserted. The write method returns the number of bytes or characters that it wrote to the file, in our example the string "I'm writing in the file\n" is 25 characters long. Also, notice how we did not have to first create the destination file. Opening the file for writing creates the file if it does not already exist.

Let's look at another method of writing to a file:

```
>>> f = open("lets_write.txt", "w")
>>> text = ['many, many \n', 'lines\n', 'are\n', 'easily\n', 'inserted\n',
        'this way!']
>>> f.writelines(text)
>>> f.close()
```

Now if we look at the file again it should look like lets_write2.txt. First, notice that the data we wrote before to the file is no longer there, this is because 'w' writes from the start of the file and **writes over anything that is already there**. Notice again that the newlines had to be inserted manually.

Read and Write

Let's look at an example that uses both reading and writing, with some functionality to process the data. We are going to write a function that will take two filenames as arguments and turn all the characters in one file into uppercase and write them into the other file. Here is the code:

```
def make_all_caps(in_filename, out_filename) :
    """Convert all characters in 'in_filename' to caps and save to 'out_filename'.

    Parameters:
        in_filename (string): Name of the file from which to read the data.
        out_filename (string): Name of the file to which the data is to be saved.

    Preconditions:
        The files can be opened for reading and writing.
    """
    fin = open(in_filename, 'r')
    fout = open(out_filename, 'w')
    for line in fin :
        fout.write(line.upper())
    fin.close()
    fout.close()
```

The first thing we do is open the `in_filename` in Universal read mode for portability. We then open the `out_filename` in write mode. Next, we use a `for` loop to iterate over the lines in `fin`. The body of the `for` loop writes the uppercase version (using strings `upper` method) of each line of the input file (`fin`) to the output file (`fout`). Both files are then closed so that the file pointers are not left open.

Saving as make_all_caps.py we can run an example using our text.txt file from earlier.

```
>>> make_all_caps('text.txt', 'text_caps.txt')
```

If we now have a look at text_caps.txt. it should have the contents of `text.txt` all in uppercase.

Files often have blank lines in them in one form or another. When dealing with files it is generally easier to ignore blank lines than to attempt to process them. However, blank lines are not really blank, they contain the new line character '\n'.

Let's modify our code above to remove blank lines from the file as well as make it all upper case.

```
def all_caps_no_blanks(in_filename, out_filename) :
    """Changes every character in 'in_filename' to all caps,
       removes blank lines and saves to 'out_filename'.

    Parameters:
        in_filename (string): Name of the file from which to read the data.
        out_filename (string): Name of the file to which the data is to be saved.

    Preconditions:
        The files can be opened for reading and writing.
    """
    fin = open(in_filename, 'r')
    fout = open(out_filename, 'w')
    for line in fin :
        if line != '\n' :
            fout.write(line.upper())
    fin.close()
    fout.close()
```

Notice the change to the function is the addition of the `if` statement to check if the current `line` is equal to a new line character. If it is not then we write the upper case version of the line to the `fout` file.

Now if we run all_caps_no_blanks1.py on our text.txt file like below:

```
>>> all_caps_no_blanks('text.txt', 'text_caps_nb.txt')
```

We should now have a file that looks like text_caps_nb.txt.

Blank lines might not just include new line characters, they may also include other forms of whitespace, such as spaces and tabs. If we run the previous function on words.txt we will notice that there are still blank lines as some of them have spaces.

We can modify the example again to work in these cases using the string method `strip`. `strip` returns a copy of the string with all whitespace removed from the beginning and end of the string. If the string incorporates only whitespace then an empty string is returned.

```
def all_caps_no_blanks(in_filename, out_filename) :
    """Changes every character in 'in_filename' to all caps,
       removes blank lines and saves to 'out_filename'.

    Parameters:
        in_filename (string): Name of the file from which to read the data.
        out_filename (string): Name of the file to which the data is to be saved.

    Preconditions:
        The files can be opened for reading and writing.
    """
    fin = open(in_filename, 'r')
    fout = open(out_filename, 'w')
    for line in fin :
        if line.strip() :
            fout.write(line.upper())
    fin.close()
    fout.close()
```

Now if we run all_caps_no_blanks2.py on our words.txt file and words_allcaps_nb.txt then it should be capitalised with no blank lines.

Notice how the `if` statement is simply `if line.strip()`. This is because the boolean representation of an empty string is `False` while a string with anything in it is `True`. The same can be said for objects such as lists, tuples and more.

```
>>> bool('')
False
>>> bool(' ')
True
>>> bool('sfg')
True
>>> bool([])
False
>>> bool([0])
True
>>> bool([2,5,6,3])
True
```

```
>>> bool(())
False
>>> bool((2,3,4))
True
```