Table of Contents

Lab 6: File Objects and Exceptions

Learning Outcomes

In this lab, we will:

- learn about opening text files
- read and write data to and from files
- learn more about objects and methods
- learn how to handle exceptions gracefully.

Introduction

In this lab we will be opening external files and working with the data that they contain. Working with files brings its own challenges, as a lot of things can go wrong with opening files as we will see. Filenames can be wrong, they can have incorrect permissions and so on. It's important to handle these issues gracefully, and so we will also introduce the concept of exception handling.

6.1 File Objects

By 'files', we are going to be referring (at first) to *plain text* files. In Windows, you often see these files labelled with a `.txt` suffix. Other types of files (such as Word documents) can be opened by Python as well, but have a specific encoding and usually require an external module. Believe it or not, plain text files are widely used to manage system configuration and many types of system data.

Opening and working with files is a three-step process:

1. We need to specify the filename, and create a *file object*.
2. With the *file object*, we have many methods at our disposal for reading and/or writing data.
3. Finally, we need to *close* the file object.

TASK: Create readme.txt

Use a program like notepad to create a file. It should look like this:

```
hello world
this is the second line.
this is the third line.
I would just like to say hello again!
and goodbye.
```

Make sure this file is called `readme.txt` and is in your current directory. You can also download it from here. A reminder: you should probably be creating a directory called `lab6` for all your work.

Creating a File Object

The code to create a file object looks like this:

```
f = open('readme.txt', 'r')
```

The function `open()` takes two arguments: a filename, and a string to tell it what type of file operation we are going to perform.

'r'
Read operation. No change to the contents of the file.

'w'
Write operation. **This will overwrite whatever was in the file**.

'a'
Append operation. This will preserve the contents of the file and add data to the tail.

`open()` returns a special datatype called a file object. The file object contains many useful *methods* for reading and writing data. By 'method', we mean a function that belongs to an object. We will use a method called `.read()` to accomplish our second step…

```
f = open('readme.txt', 'r')
filestring = f.read()
```

…and another method called `.close()` to accomplish our third step.

```
f = open('readme.txt', 'r')
filestring = f.read()
f.close()
```

At this point, our file is closed and its contents are saved in a string.

```
hello world\nthis is the second line.\nthis is the third line.\nI would just
like to say hello again!\nand goodbye.
```

The `\n` are called *newline characters* and define when we will move to the next line. Each `\n` is equivalent to pressing the Enter key. The `print()` function should recognize these characters and print the string in the correct format.

6.2 What Exactly Is An Object?

With that simple example out of the way, perhaps we should define what `f` is in our example.

Objects are ways to organize code. When the age of computing was just getting underway, it was enough to organize our code in the way we have started: write code line-by-line, creating variables as we need them. As programs became more complex, it became necessary to start using functions. Objects are the next step in this progression: objects contain their own private variables and their own functions (called methods).

Object-Oriented Programming (OOP) is a very large topic, one that we won't get into here. Suffice to say, you have already been working with objects – everything in Python is an object!

```
mystring = 'hello world!'
mystring.upper()
```

```
HELLO WORLD!
```

In older programming languages, a string is *just a string*: an array of characters. To convert to capitals, to reverse the letters, to search for a specific character, you would have to write

your own functions to accomplish any of this. This takes a lot more work, but your programs can run faster.

Python takes a different approach from these early languages. We are less concerned with raw speed, and more interested in providing useful features to the programmer. `mystring` is not a string, but a *string object*. Not only does it contain the character array `hello world!`, but it comes with several useful methods such as `.upper()`. We are not forced to create our function to accomplish this, the code has already been written and is already contained inside our string object. The full set of features contained in the string object are hidden from us by default, but we can make use of them if we need to. This is the advantage of OOP.

Note: We use periods (`.`) to indicate that something is *contained inside* something else. We use this for both *methods contained inside of objects* (for example: `mystring.upper()`, and *functions or variables contained inside a module* (for example: `sys.argv`).

Back To File Objects

In the interpreter, create a new file object using `readme.txt`, and then use Python's built-in `help()` function:

```
f = open('readme.txt', 'r')
help(f)
```

You will get a help document related to `TextIOWrapper` objects. Again, you can scroll past the technical description to the methods contained. Check the following:

- Close
- Read
- Write
- Readlines
- Writelines

Before moving on to the next section, try and predict how these methods work.

6.3 TASK: File Writing

Create a file called `lab6a.py`. We will use this to test file write operations.

Writing to a file is slightly different. Instead of calling `.read()` and getting a return value, we call `write()` with an argument. A number of characters is returned, but this can be ignored.

```
f = open('testing.txt', 'w')
f.write('This is a test sentence.')
f.close()
```

Run your script and verify that it works.

Now modify your script so that it looks like this:

```
f = open('testing.txt', 'w')
f.write('This is a new test sentence!')
f.write('This is the second line(?)')
f.close()
```

Run this new script again. What do you notice?

Add this line to the top of your script:

```
data_to_write = ['First Line!', 'Second Line!!', 'Third Line!!!', '...and so on!']
```

Remember that using `w` will **overwrite the contents of a file.** Each time that you run your script, the previous data is lost.

To complete this task, use what you've learned about file operations, newline characters, and for loops to put each element of `data_to_write` on a new line in the 'testing.txt' file. The contents should look like this:

```
First Line!
Second Line!!
Third Line!!!
...and so on!
```

Include `lab6a.py` in your lab submission.

6.4 String Functions

Hopefully you've figured out a way to *add* newline characters to your output. Let's explore ways of *removing* them. Open the interpreter and search through `help` for a string again. In particular, look at the following methods:

- replace
- split
- strip

By default, `.strip()` with no arguments will remove newline characters from the beginning and end of a string. So a string like `'hello\n'.strip()` will become `'hello'`.

The method `.split('\n')` will act on all newline characters inside a string, and will return a list:

```
newlist = 'One\nTwo\nThree'.split('\n')
print(newlist)
```

```
['One', 'Two', 'Three']
```

The argument for `.split()` is the character that will define the *delimiter*, which is another way of defining where one element ends and another begins. Notice how each newline character defines the boundary between elements in the list.

`.replace()` isn't related to working with newline characters, but is incredibly useful since finding and replacing strings is such a common task. The optional argument `1` here defines the number of substitutions to perform; by default it will make every substitution possible.

```
print('What is old is new'.replace('is', 'was', 1))
```

```
What was old is new.
```

If you merely want to check if a string exists inside a bigger string without replacing it, this can be accomplished without a method:

```
teststring = 'Alphabet Soup'

if 'Soup' in testring:
    print('Soup found')
else:
    print('No soup for you!')
```

Remember that `help()` is a powerful tool when trying to solve programming problems!

6.5 TASK: Reverse Reading

- Create `lab6b.py`.
- This program should open a file that is defined by a *command line argument*. Make sure you import the `sys` module in order to do this.
- If the user doesn't specify a file, use `readme.txt`.
- Read the contents of the filename specified.
- Print the contents of the file, but each line should be in *reverse order*. So the last line of the file should be printed first, then the second-last, and so on.

Sample Output

```
python3 lab6b.py testing.txt
```

```
...and so on!
Third Line!!!
Second Line!!
First Line!
```

```
python3 lab6b.py
```

```
and goodbye.
I would just like to say hello again!
This is the third line.
This is the second line.
Hello World!
```

6.6 Exceptions

By now you have no doubt encountered exceptions as you write programs. Exceptions can be triggered by error in *Syntax*:

```
if condition == True:
print("True")
```

```
File "<stdin>", line 2
    print('true')
    ^
IndentationError: expected an indented block
```

or when you are attempting to do something that will seem to trigger undefined behaviour.

```
mylist = ['A', 'B', 'C']

print(mylist[4])
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

As we have mentioned, exceptions are actually the best outcome in this sorts of situations. Exceptions identify code that could trigger ambiguous outcomes. In programming, ambiguity is our enemy. We want to specify our preferred behaviour as much as possible. The hardest types of problems to debug and solve are ones that occur quietly or inconsistently. Exceptions are *loud*, because they prevent us from proceeding until we have solved their root cause.

However, not all exceptions can be prevented in our code. Sometimes, believe or not, we are not the ones at fault! Since we are working with files in this lab, we should be aware of all the ways that reading files can fail:

- The filename can be incorrect, or the file could be missing.
- Our user might not have permission to read the file contents, or it could be a read-only file.
- There might be a hardware issue that otherwise prevents us from reading the file.

In each of these cases, the user's 'plan A' will not succeed, and we will need to execute a 'plan B'. This alternative set of code should *only* run if an exception is encountered. For the moment, our 'plan B' should only be to print simple message to the user, and halt the program.

For this example we will put our file operations into a function. The function argument will be the filename, and the return value will be the file contents as a string.

```
filename = 'missing.txt'

def read_file(filename):
```

```
    "read a file, halt if missing"
    try:
        f = open(filename, 'r')
        string = f.read()
        f.close()
    except:
        print("ERROR!")
        sys.exit(1)
    return string
```

The code inside the `try:` block is 'risky' code. We think that an exception might possibly occur here. If an exception does occur, we run the code defined in the `except:` code block.

Notice that the error message here is not very useful. That's because we have no idea what *type* of exception has occurred, and so our error message is hopelessly vague. This code breaks the first principle of exception handling, which is to **specify the type of exception to handle.**

```
filename = 'missing.txt'
line_count = 0

def read_file(filename):
    "read a file, halt if missing"
    try:
        f = open(filename, 'r')
        string = f.read()
        result = max_lines / count
        f.close()
    except FileNotFoundError:
        print(f"ERROR: {filename} was not found.")
        sys.exit(1)
    string = f.read()
    f.close()
    return string
```

We have now specified the `FileNotFoundError` as a specific exception to handle, and we can specify a more useful message for the user. However, notice now that we have another issue: inside our `try` block we are going to cause a divide by zero error. This brings us another guideline: **make your try blocks as short as possible**.

```
filename = 'missing.txt'
line_count = 0

def read_file(filename):
    "read a file, halt if missing"
    try:
        f = open(filename, 'r')
    except FileNotFoundError:
        print(f"ERROR: {filename} was not found.")
        sys.exit(1)
```

```
    except PermissionError:
        print(f"You do not have permission to open {filename}.")
        sys.exit(1)
    string = f.read()
    f.close()
    return string

def divide_max_by_count():
    try:
        result = max_lines / count
    except ZeroDivisionError:
        print("Can't divide by zero.")
```

It didn't make sense to perform the max_lines by count calculation inside that function (remember, functions should do only one thing!) so we have put that into a new function with its own exception handling. This will make the life of the user much easier. In addition, we are handling another possible issue: a permission error.

Nevertheless, we can't predict every possible thing that can go wrong. In this case, we can introduce another guideline: **use a default except block to catch any other error you haven't already handled.**

```
filename = 'missing.txt'
line_count = 0

def read_file(filename):
    "read a file, halt if missing"
    try:
        f = open(filename, 'r')
    except FileNotFoundError:
        print(f"ERROR: {filename} was not found.")
        sys.exit(1)
    except PermissionError:
        print(f"You do not have permission to open {filename}.")
        sys.exit(1)
    except:
        print("Something has gone wrong, but we know it isn't a permission
error or file not found error.")
        sys.exit(2)
    string = f.read()
    f.close()
    return string
```

Remember that exception handling will work like the `if/elif/else` pattern: if the first condition is false, the interpreter will test each successive condition in turn. If all conditions are false, the `else` code block runs as a last resort. Similarly, here we want to start with specific exceptions and then define the most broad exceptions at the end. The error in `except:` is vague, but with luck we should almost never encounter this error.

A full list of exception types can be found in the official documentation. However, you will become very familiar with many of the common types of exception as you progress.

6.7 TASK: A Simple Grep

`grep` is a tool installed on most *Nix systems. It will search for a particular keyword inside of a file.

Inside the file `readme.txt`:

```
hello world
this is the second line.
this is the third line.
I would just like to say hello again!
and goodbye.
```

```
eric@Archie ~ $ grep hello readme.txt
hello world
I would just like to say hello again!
```

Your task will be to re-create this basic functionality.

1. Name this file `lab6c.py` and include it in your lab submission.
2. Your program should handle command line arguments. **Do not use input()**. The first argument is the keyword to search, and the second is the filename.
3. If the user doesn't have two arguments, print a sensible usage message.
4. Open the filename. Your script should handle `FileNotFoundError` exceptions at the very least. If the file is missing, print a sensible error message and halt.
5. Save the contents of the file as a *list*.
6. For each line of the list, search for the keyword. If the keyword is found, print the entire line **including a line number**. If it is not, do nothing.
7. Make sure to close the file before your program exits.

Use the command prompt to test your code with both valid files and with bad filenames.

Sample Output

Lines with >> are executed by the user. All other lines indicate program output.

```
>> python lab6c.py
Usage: lab6c.py keyword filename
>> python lab6c.py hello readme.txt
1: hello world
4: I would just like to say hello again!
>> python lab6c.py line readme.txt
2: this is the second line.
3: this is the third line.
>> python lab6c.py toaster readme.txt
>> python lab6c.py line missing.txt
ERROR: missing.txt not found.
```

6.8 TASK: Text Parser

For this task, you can use a file called `letters-and-numbers.txt`, which can be downloaded from here. The contents of the file look something like the following:

```
a e 12 o 3.5 i
g 4 c 7 12 mn
```

Your task will be to open this file, remove all the numbers, and save the contents of the file. Your script should also **print** the sum of all numbers on each line. So for the example above, the script should print:

```
15.5
23.0
```

and the contents of `letters-and-numbers.txt` will be:

```
a e o i
g c mn
```

1. Name this file `lab6d.py`.
2. As before, this script should command line arguments to specify the file to open.
3. If the user doesn't provide an argument, print a sensible usage message. It should start with the word `Usage:`.
4. Open the filename. Your script should handle `FileNotFoundError` exceptions at the very least. If the file is missing, print a sensible error message and halt. It should start with `ERROR:` and include the invalid filename.
5. When you have read the file, you should have a *list of strings* that looks like this: `['a e 12 o 3.5 i', 'g 4 c 7 12 mn']`.
6. In order to separate numbers from non-letters, use a `.split()` function.
7. Notice that you will need to handle both *int* and *float* values. We have learned an easier way to do this. Using `float('3.5')` will succeed, but `float('a')` will cause an exception. You can use exception handling to deal with non-numeric strings.
8. Save the strings (with numbers removed) into a new list of strings. **Be sure to remove any trailing spaces.**
9. In order to update the file, you can simply rewrite to the target file with the new list of strings. You may wish to back up the `letters-and-numbers.txt` file as you are testing.
10. Submit `lab6d.py` with the rest of your lab files.

CHALLENGE TASK: Last Letter

- Call this file `challenge6.py`.
- Open a file using a command line argument, with exception handling for FileNotFoundError.
- For each line of the file contents:

- iterate through each character of the line. Find the alphabetical character (a-z) that comes closest to the *end of the alphabet.*
- In other words, you want to find the letter that is closest to 'z' on each line. Print that character only.
- Also be sure to *ignore case*. So capital letters should work just like lower-case letters. You can simply convert all capitals to lower case to do this.

Sample Output

```
python3 challenge6.py readme.txt

w
t
t
y
y

python3 challenge6.py testing.txt

t
s
t
s
```

Deliverables

The check script can downloaded from here.

- ☐ lab6a.py
- ☐ lab6b.py
- ☐ lab6c.py
- ☐ lab6d.py
- ☐ challenge6.py
- ☐ lab6-check-output.txt

Glossary

- Object
- Method
- Newline character
- Delimiter