Common Issues with Files

by Sophia

WHAT'S COVERED

In this lesson, we'll work through common issues when it comes to working with files. Specifically, this lesson covers:

1. Working With Files

We really do not want to have to edit our Python code every time we want to process a different file. It would be ideal to ask the user to enter the file name string each time the program runs so they can use our program on different files without changing the Python code.

This is quite simple to do by reading the filename from the user using input as follows:

EXAMPLE

```
fname = input('Enter the file name: ')
fhand = open(fname)
count = 0
for line in fhand:
    count = count + 1
print('There were', count, 'lines in', fname)
```

We read the filename from the user and place it in a variable named `fname` (filename) and have the variable `fhand` (file handler) use the `open()` function on that file. Now we can run the program repeatedly on different files, and for this example we count the number of lines in the file.

KEY CONCEPT

Is there any importance to knowing the number of lines in a file?

Knowing the number of lines in a file can be useful to quickly determine how we can make use of data in the file. The number of lines indicates how large a file is and certain operations may be easier to work with when it is a smaller file vs. extremely large files that may require alternative approaches to data processing. In knowing the number of lines in a file, we can also determine if the file is empty or not. Although a file exists, if it is empty, we can change the processing accordingly.

---

2. Filename Errors

Based on the program above, what if our user types something that is not a filename?

Let's try opening a file called "missing.txt".

Enter the file name: missing.text

```
missing.txt
Traceback (most recent call last):
  File "/home/main.py", line 2, in <module>
    fhand = open(fname)
FileNotFoundError: [Errno 2] No such file or directory: ''
```

Note: You may need to select your "Enter" key to continue with the execution if only "missing.txt" is shown in the output.

Or a file called "na na boo boo".

```
na na boo boo.txt
Traceback (most recent call last):
  File "/home/main.py", line 2, in <module>
    fhand = open(fname)
FileNotFoundError: [Errno 2] No such file or directory: ''
```

Directions: Try using the code in the section above and open files that you know do not exist in your file directory of the IDE. Receiving the same error message?

Users will eventually do every possible thing they can do to break your programs, either by mistake, on purpose, or with malicious intent. As a matter of fact, an important part of any software development team is a person or group called Quality Assurance (or QA for short) whose very job it is to do the craziest things possible to break the software that the programmer has created.

The QA (Quality Assurance) team is responsible for finding the flaws in a program before it is delivered to end-users who may be purchasing the software or paying our salary to write the software. So, the QA team is the

programmer's best friend.

Now that we see the flaw in the program above, we can elegantly fix it using the `try` and `except` statements. We first discussed the `try` and `except` statements in Unit 1 while covering exceptions.

So, we need to assume that the `open()` function call might fail and thus add recovery code when it does.

EXAMPLE

```
fname = input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print('File cannot be opened:', fname)
    exit()
count = 0
for line in fhand:
    count = count + 1
print('There were', count, 'lines in', fname)
```

exit() function

You should notice that we used the `exit()` function in the example above to terminate the program if the `except` statement catches a filename that cannot be opened. The **exit() function** is a built-in function that we can call that will quit and leave the program. The `exit()` function may not always be available and it does not accept any arguments. Since the `exit()` function lacks some key functionality, it is not recommended to be used for production. However, in small programs, using the `exit()` function works well for just testing purposes.

sys.exit() function

A better use for production code is to use the `sys.exit()` function of the `sys` module. The **sys.exit() function**, which is part of the `sys` module, allows the termination of a program and can optionally take in an argument such as a number. If a number is passed, it means that the program exited normally without issues. Conventionally 0 (zero) is passed if there are no issues with the program and the program is exiting successfully. 1 (one) is passed if there is an issue or error with the program. However, any number can be passed to indicate an error back to the operating system. This number is something that is used at the operating system level rather than from Python.

KEY CONCEPT

The `sys` module is used to do anything in the operating system. It allows you to work with files and directories/folders. This is different depending on the operating system which the sys module handles for you directly.

The `sys` module is always available so the `sys.exit()` function that this module contains can be used once this module is imported.

Here is an example that is importing the `sys` module and using the preferred `sys.exit()` function to exit from the program.

EXAMPLE

```
import sys

fname = input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print('File cannot be opened:', fname)
    sys.exit()
count = 0
for line in fhand:
    count = count + 1
print('There were', count, 'lines in', fname)
```

Now when our user (or QA team) types in silliness or bad file names, we “catch” them and recover gracefully.

TRY IT

Directions: Go ahead and add the `try` and `except` statements and run the program again. Try a file that you have in your local IDE directory and one that is not.

Here is the expected output to a file that was found. Here we see that `myfile.txt` was found when we tried the updated program and it returned 4 lines.

```
Enter the file name: myfile.txt
There were 4 lines in myfile.txt
```

And here is the expected output to a file that is not found.

```
Enter the file name: na na boo boo
File cannot be opened: na na boo boo
```

Note, you may see a separate error in the IDE; this will not show up in other systems that we run the code in.

Protecting the `open()` function call is a good example of the proper use of `try` and `except` statements in a Python program. We use the term “Pythonic” when we are doing something the “Python way”. We might say that the above example is the Pythonic way to open a file.

Once we become more skilled in Python, we can engage in scenarios with other Python programmers to decide which of two equivalent solutions to a problem is better. We are not always interested in just making something work; we also want our solution to be elegant and to be appreciated as elegant by our peers. Similar

to the `exit()` function, although this works to exit the program, it would generally be better to import the `sys` module and use the `sys.exit()` function to keep the code consistent.

3. Whitespace Errors

When you are reading and writing files, you might run into problems with whitespace. Whitespace are characters such as tabs, spaces, and newlines. These errors can be hard to debug because spaces, tabs, and newlines are normally invisible.

EXAMPLE

```
s = '1 2\t 3\n 4'
print(s)
```

TRY IT

Directions: Try running this short snippet of code.

```
1 2    3
 4
```

A little difficult to read if you did not know or forgot there were whitespace characters in there.

The built-in function `repr()` can help. The **repr() function** takes any object as an argument and returns a string representation of the object. For strings, it represents whitespace characters with backslash sequences:

EXAMPLE

```
s = '1 2\t 3\n 4'
print(repr(s))
```

TRY IT

Directions: Try the sample code above and run it.

```
'1 2\t 3\n 4'
```

This can be helpful for debugging.

One other problem you might run into is that different systems use different characters to indicate the end of a line. Some systems use a newline, represented by \n. Others use a return character, represented by \r. Some use both. If you move files between different systems, these inconsistencies might cause problems.

TERMS TO KNOW

exit()
The `exit()` function is a built-in function that we can call that will quit and leave the program.

sys.exit()
The `sys.exit()` function, which is part of the `sys` module, allows the termination of a program and can optionally take in an argument such as a number.

repr()
The `repr()` function takes any object as an argument and returns a string representation of the object.

Whitespace
Whitespace are characters such as tabs, spaces, and newlines.

SUMMARY

In this lesson, we continued **working with files** and built a small program that allows for a user to input a filename to open. Then, we identified **filename errors** that would appear if the filename entered did not exist. We updated the code to include `try` and `except` statements to capture exceptions to the filename inputs and handle it accordingly. Finally, we looked at some **whitespace errors** that can appear in file usage and how the use of the `repr()` function can help see the whitespace characters more clearly.

Best of luck in your learning!

TERMS TO KNOW

Whitespace
Whitespace are characters such as tabs, spaces, and newlines.

exit()
The `exit()` function is a built-in function that we can call that will quit and leave the program.

repr()
The `repr()` function takes any object as an argument and returns a string representation of the object.

sys.exit()
The `sys.exit()` function, which is part of the `sys` module, allows the termination of a program and can optionally take in an argument such as a number.