File Input and Ouput

CSE/IT 107L

NMT Department of Computer Science and Engineering

"The danger that computers will become like humans is not as big as the danger that humans will become like computers." ("Die Gefahr, dass der Computer so wird wie der Mensch ist nicht so groß, wie die Gefahr, dass der Mensch so wird wie der Computer.")

— Konrad Zuse

"First, solve the problem. Then, write the code."

— John Johnson

"I don't need to waste my time with a computer just because I am a computer scientist."

— Edsger W. Dijkstra

Introduction

In previous labs, we taught you how to use functions, math, lists, strings and other features. We will combine all of these concepts in this lab and teach you how to interact with files. You will also learn how error handling works in Python. Some basic concepts of object oriented programming will be introduced using turtles as an example.

Contents

1 File I/O

Knowing how to work with files is important, since it lets us store and retrieve data beyond the scope of the single execution of a program. To open a file for reading or writing we will use the `open()` function. The following example opens a file named "hello.txt" and writes "Hello World" to it using the `print()` function.

```
output_file = open("hello.txt", "w")

print("Hello World", file=output_file)
output_file.close()
```

The `print()` function can have an additional "file" parameter passed to it to allow writing to a file. This causes it to send its output to the file rather than the screen, though otherwise it performs identically.

The `.close()` method is used to tell Python that you are done with a file so it can close your connection to it. If you don't call it the file can be corrupted or other programs will be unable to access that file.

Files can also be written to by using `.write(contents)`. This method will write only the characters given to it, so a newline `\n` must be included for a newline.

```
output_file = open("hello.txt", "w")

output_file.write("Hello World\nGoodbye!\n")
output_file.close()
```

The arguments to the `open()` function are, in order, the name of the file to open and the mode in which to open the file. "w" means that the file is to be opened in write mode. If the file does not exist, this will create the file. If it does exist, then the contents of the file will be cleared in preparation for the new ones.

Other options include "a", which is similar to "w" but will not clear the contents of an existing file and will instead append the new data to the end, and "r" which will read the file instead. If "r" is used and the file does not exist, then an error will occur. The following code takes a filename as user input, then prints out the entire contents of that file.

Mode	What it does
a	Create file if does not exist, open file, append contents to the end
w	Create or **delete the contents** of the file, open file, write contents to the beginning of file
r	Open file, permit reading only

```
filename = input("What file should be read? ")

input_file = open(filename, "r")
for line in input_file:
```

```
    print(line, end="")

input_file.close()
```

The `print()` function has an additional optional "end" parameter. This allows you to specify what should be printed after the main string given to it. This is important because it defaults to `"\n"`, which causes a newline after every print statement. By changing "end" to `""` we prevent a newline from being added after every line of the file is printed. This is because each line in the file already has a newline at the end of it, so we don't need `print()` to add its own.

When reading from a file, Python can use a `for` loop to go through each line in sequence. This works identically to looping through a list of strings. Think of the file as a list with every line being a different element of the list. The entirety of the file can also be read into a single string using the `.read()` function.

This code prints the contents of a Python script.

```
>>> input_file = open("test.py", "r")
>>> contents = input_file.read()
>>> print(contents)
filename = input("What file should be read? ")

input_file = open(filename, "r")
for line in input_file:
    print(line, end="")

input_file.close()
>>> input_file.close()
```

The `.readlines()` function can be used to read all of a file at once, though it splits the file into a list. Each element of the list will be one line of the file being read.

1.1 Opening Files Using `with`

What if the function returns or the program exits or an error is raised before the file is closed? There's an easy way to make sure every open file is closed in many situations. The `with` ensures that your file is closed when the code in the `with` block finished executing.

```
filename = input("Enter filename: ")

with open(filename, "r") as input_file:
    for line in input_file:
        print(line, end="")
```

`input_file.close()` is not necessary because the file was closed automatically.

1.2 Opening Multiple Files Using `with`

If you need to open multiple files, you may do so using a single with-statement. The following code opens two files and copies the contents of one file into the other file.

```
with open("input.txt", "r") as input_file, \
     open("output.txt", "w") as output_file:
    for line in input_file:  # read every line in input.txt
        output_file.write(line)  # and write it to output.txt
```

The backslash is used to break the long line into multiple lines.

Using `with`

Always use the `with` statement to deal with file I/O in Python.

2 Exceptions

An *exception* is an error message in Python. When a certain operation encounters an error, it can *raise* an exception that is then passed on to the user. If your script is being executed as a program, exceptions cause it to print the exception and exit.

```
>>> 43 / 0
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ZeroDivisionError: division by zero
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ZeroDivisionError: division by zero
```

However, exceptions can be handled using the `try` and `except` statements. The `except` block is only executed if an exception is caught in the `try` block. Additionally, when an error is caught in the `try` block we stop executing commands in the `try` block and immediately jump to the `except` block.

The following example throws a division by zero error and prints "division by zero":

```
prime = 7

try:
    result = prime / 0
    result = 7 * 42
except ZeroDivisionError as err:
    print(err)
```

Since the error is thrown on line 5, line 6 is never executed. Also, we are only catching `ZeroDivisionError` exceptions, any other exceptions will remain uncaught.

If you are experimenting with code and want to know the name of an exception that is thrown, take a look at the error message:

```
>>> float('obviously not a float)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: could not convert string to float: 'obviously not a float'
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: could not convert string to float: 'obviously not a float'
```

The part highlighted in red here is the name of the error message, `ValueErrror`.

Hence, if you are getting user input and want to check whether it is the correct type, use a try-except block around the conversion:

```
try:
    x = int(input('Enter an integer number: '))
except ValueError:
    print('You did not enter an integer!')
else:
    print('You entered {}'.format(x))
```

Notice that you can use an `else` block to execute code if no exception was thrown.

Any `except` block that does not list built-in exceptions will catch all exceptions not listed in previous `except` blocks. For example, the following code will throw an error if the user enters anything but an integer:

```
try:
    x = int(input("Enter a number: "))
except:
    print("Unknown error.")
    raise
else:
    print("You entered: " + str(x))
```

The `raise` keyword causes a detailed trace and prints out additional information if an exception is encountered. The `else` block is optional and will be executed if no exceptions are thrown in the `try` block. There are many more built-in exceptions such as `IOError` that can be found here:

`https://docs.python.org/3/library/exceptions.html`

2.1 File I/O Exceptions

When working with files it is important to check for exceptions. These are only some of the exceptions that can be raised when doing file I/O.

- `FileNotFoundError` is raised when we try to open a file that. It should be handled so we can exit the program safely or recover rather than observe unexpected results.

```
>>> open("not_a_file.txt", "r")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'not_a_file.txt'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'not_a_file.txt'
```

- `IsADirectoryError` is raised when trying open an directory as a file.

```
>>> open("directory", "w")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IsADirectoryError: [Errno 21] Is a directory: 'directory'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IsADirectoryError: [Errno 21] Is a directory: 'directory'
```

The `finally` block is where clean-up actions are performed and is always executed after leaving the `try` block. The following is a good example of using `with` and exception handling together to open and read a file.

```
filename = input('Enter a filename >>> ')

try:
    with open(filename, 'r') as f:
        sum = 0
        for line in f:
            sum += float(line)
except FileNotFoundError:
    print('File "{}" not found'.format(filename))
except IsADirectoryError:
    print('File "{}" is a directory'.format(filename))
except ValueError:
    print('File "{}" contains a line that'.format(filename)
        + 'cannot be converted to a float')
else:
    print('Sum of all numbers in file is {}'.format(sum))
finally:
    print('Goodbye.')
```

3 Exercises

Using `with`

If you manually close the file by calling `.close()`, the file may not be closed in exceptional circumstances. **Always use the `with` statement when opening files** in Python. See Section 1.1

for more detail.

Exercise 3.1 (word_count.py).
Write a program that takes in a filename and string as input. Then print how many times that string appears inside the chosen file. If the file does not exist, continue asking for a filename until one is given that exists. Use your source code file as test input.

Make sure to test files with that contain the same word multiple times.

```
$ python3 word_count.py
Please enter a filename: word_count.py
Please enter a string to search for: print
The string 'print' appears 102 times in the file 'word_count.py'
```

Exercise 3.2 (simplediff.py).
Write a "diff" program that prints out the differences, line by line, of two files. Your program should ask the user for the names of two files, then print the differences between them. Follow the format output as shown below. Make sure to use proper error handling techniques for file I/O.

Assume all files have the same number of lines. The following output shows the output of the files `file1.txt` and `file2.txt`.

```
$ cat file1.txt
John goes to work.
Keith and Kyle went to the Ensiferum concert.
Alice ate an apple pie.
Joe cut down a tree.
The dog jumped over the wall.
$ cat file2.txt
John goes to work.
Coral went to a Kesha concert.
Alice ate an apple pie.
Joe planted a tree.
The dog jumped over the wall.
```

This is the result of running the script `simplediff.py` on the two files:

```
$ python simplediff.py
Enter file name 1 >>> file1.txt
Enter file name 2 >>> file2.txt

2c2
< Keith and Kyle went to the Ensiferum concert.
---
> Coral went to a Kesha concert.
4c4
< Joe cut down a tree.
---
> Joe planted a tree.
```

Compare the output of your script to that of the `diff` program by typing `diff file1.txt file2.txt` in your shell.

Exercise 3.3 (readscores.py).
Download the file `actsat.txt` provided on Canvas. It contains the following columns of whitespace-separated data:

Column 1 2-letter state/territory code (includes DC)

Column 2 % of graduates in that state taking the ACT

Column 3 Average composite ACT score

Column 4 % of graduates in that state taking the SAT

Column 5 Average SAT Math score

Column 6 Average SAT Reading score

Column 7 Average SAT Writing score

You must open this file and generate a list of dictionaries containing each row of data. Please use these keys for the dictionaries:

- `"state"`
- `"act_percent_taking"`
- `"act_average_score"`
- `"sat_percent_taking"`
- `"sat_average_math"`
- `"sat_average_reading"`
- `"sat_average_writing"`

For example, your code should process this two line file to form the following list of dictionaries.

```
AK         27 21.2    48      517         519                491
AL      81     20.3   9   556          563     554
```

```
[{"state": "AK",
  "act_percent_taking":  27
  "act_average_score":   21.2
  "sat_percent_taking":  48
  "sat_average_math":    517
  "sat_average_reading": 519
  "sat_average_writing": 491
 },
 {"state": "AK",
  "act_percent_taking":  81
  "act_average_score":   20.3
  "sat_percent_taking":  9
  "sat_average_math":    556
```

```
14    "sat_average_reading": 563
15    "sat_average_writing": 554
16  }]
```

Submitting

You should submit your code as a tarball that contains all the exercise files for this lab. The submitted file should be named

`cse107_firstname_lastname_lab9.tar.gz`

Upload your tarball to Canvas.

List of Files to Submit

Exercises start on page 5.