Comp 3005: Data Science Bridge Course: Computer Science Programming Basics

Class 11: File I/O Streams

Dalton Crutchfield
University of Denver

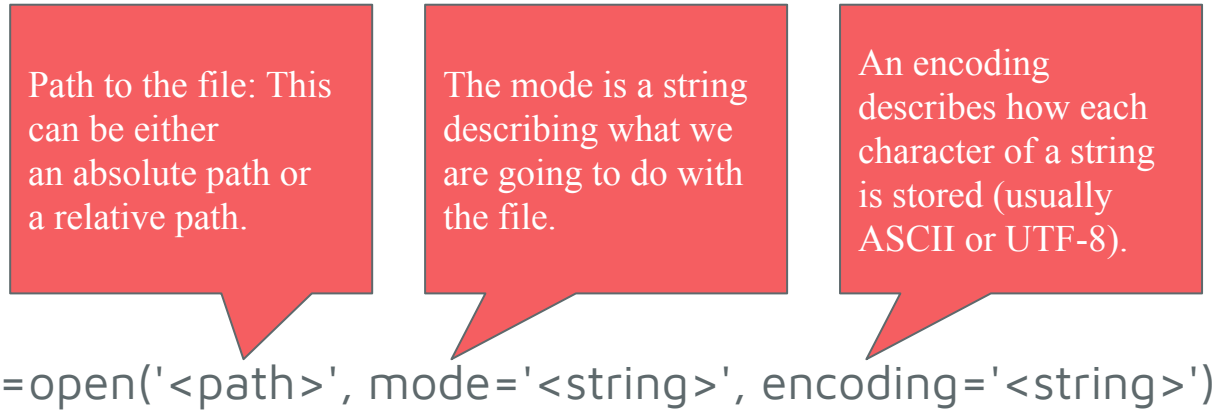
Introduction

Streams

- Streams are used to read and write data to files.
- Input streams are read once from start to finish.
- Output streams are written to file once from start to finish.
- This means that we cannot jump around the file or move backward.
- Python allows random access I/O, but we will not study those.

Opening a File

- The open(…) function returns a file object, which can be used for reading and writing.



Mode

- The modes we are interested in are:

Mode	Name	Meaning
‘r’	Read	Opens the file for reading; fails if it does not exist
‘w’	Write	Opens the file for writing; deletes existing contents if it already exists
‘a’	Append	Opens the file for appending; writes add to the end of the file if it already exists or creates it if it does not exist

- If a mode is not specified, then the default value is ‘r’.

Encoding

• ASCII is great for English characters but not for much else. It only defines 128 characters, which is small enough that each character can fit into a single byte.

• UTF-8 is great for pages that could contain characters from any language. It is currently the dominant encoding for pages on the Internet.

I/O Streams For Text Files

Reading a Text File

- There are a few common ways to read text data.
  - Reading the entire file as a string:

```
giant_string = f.read()
```

  - Reading the file as a list of lines:

```
# each element of 'lines' below is a
# string containing one line of 'f'
lines = f.readlines()
```

Reading a Text File

- There are a few common ways to read text data.
  - Read the lines with a loop:

```
for line in f:
    code to process line
```

  - Read the lines with enumerate:

```
for index, line in enumerate(f):
    code to process line
```

  - Note: When we read the lines of a file, the newline at the end may be included in the string. The last line of a file may or may not have a newline at its end. Blank lines just contain a newline character.

Writing Text Files

• Text files have a method write(…), which writes a given string to the given file and returns the number of characters written:

```
# write(...) takes a string and
# writes it to the given file
f.write('Hello, world!\n')
f.write('This is a sentence.\n')
```

• The trailing newlines are necessary if we want newlines! Each write(...) call only writes the characters we tell it to.

Flushing a File

- Though we may have called the write(...) method, the data may not actually be in the file yet.
- Writing to disk is relatively expensive, so most languages buffer their output by default and write it in larger chunks.
- There are two common approaches:
    1. Only write if there is a large amount of input to write (typically four to eight kilobytes)
    2. Whenever a new line is reached
- We can request that the program sends its data to disk by calling the flush(...) method:

```
# it's unknown whether the data is actually written to disk
f.write('is this written?')
# flushing the file pushes its data to disk
f.flush()
```

Closing a File

• Files are limited resources: A program can only have so many open at a time.
• It's good practice to close a file when we're done with it by calling the close(...) method.

Closing a file flushes all of its changes to disk and lets the operating system know we are done with it:

```
# files can be manually closed with close(...)
f.close()
# after this point, the file cannot be used for
# reading or writing
f.read() # errors
f.write('?') # errors
```

With Statements

• With statements automatically close files when we are finished with them:

```
# opens the file and creates a variable 'f' for it
with open(...) as f:
    all code involving the file 'f' goes here
# the file is closed at the end of the with-block
# (when the indentation changes)
```

• This helps make our code more readable, as it makes us indent the region where we are using a file. Files are closed, even when the code encounters an error. This protection can be quite useful.

I/O Streams For CSV Files

Reading a CSV File

• You may also read in spreadsheets in the same manner, the only difference would be how you handle the lines. First Open the File in the same way

Path to the file: This can be either an absolute path or a relative path.

The mode is a string describing what we are going to do with the file.

An encoding describes how each character of a string is stored (usually ASCII or UTF-8).

```
f=open('<path>.csv', mode='<string>', encoding='<string>')
```

Reading a CSV File

• As seen before, often dictionaries are good ways to store tables, we can read the file into this format:

```
f=open('FileIO.csv', mode='r', encoding='ASCII')
dict = {} #create empty dictionary
lines = f.readlines() # gets all 'rows' in a list
headers = lines[0][:-1].split(',')# column headers
for i in range(len(headers)): # loop through each header
    dataToStore = []
    for line in lines[1:]: #loop through row
        data = line[:-1].split(',')#split line on comma
        dataToStore.append(data[i]) # add in data for appropriate column
    dict.update({headers[i]:dataToStore})#add column to dictionary
```

Exception Handling

Exceptions or Errors

- We will use the terms exceptions and errors to refer to unexpected errors in a Python program.
- Here is a list of the errors we have seen so far:

Exception	Causes
SyntaxError	Python doesn't recognize what we've typed as valid code.
IndentationError	Python doesn't recognize our indentation and therefore can't read our code.
NameError	Python doesn't have a definition for the variable we're trying to use.
TypeError	The wrong types are being used, like adding a string to an int.
ValueError	The wrong value was given to a function, like running int('hello').
IndexError	An invalid index was used on a list, like accessing a list beyond its length.
KeyError	An invalid key was used on a dictionary, like one that is not yet mapped.
FileNotFoundError	A file of a given name could not be opened because it could not be found.

Catching Exceptions

• Exceptions can be caught by placing our code in a try-block. Once any exception occurs, Python jumps to the except-block. For example:

```
try:
    lst = [1, 2, 3]
    for i in range(0, 100): # clearly, this is the wrong range...
        print(lst[i])
Except:
    print('something bad happened!')
```

• Instead of crashing, the IndexError was caught and handled by the except-block.
• Catching an exception just means that it is identified by the tryexcept code. Handling the exception means that we have actually reacted to that exception.

Catching Exceptions

• A try-block may have multiple except-blocks, each of which only catches a certain type of error. For example:

```
try:
    # some offensive code
except ValueError:
    # only handles ValueErrors
except TypeError:
    # only handles TypeErrors
except:
     # handles everything else...
```

• Using except with no exception name following will catch any exception (not recommended).

Final Thoughts

• Knowing when an exception could be thrown can help make our code more reactive.
• We should only use them to catch exceptional circumstances. They are not for hiding our own errors.
• Exceptions are only caught if they occur within a try-block. This includes exceptions that are thrown inside of functions:

```
def div(a, b):
    return a / b
try:
    div(10, 0)
except ZeroDivisionError:
    print('cannot divide by zero!')
```

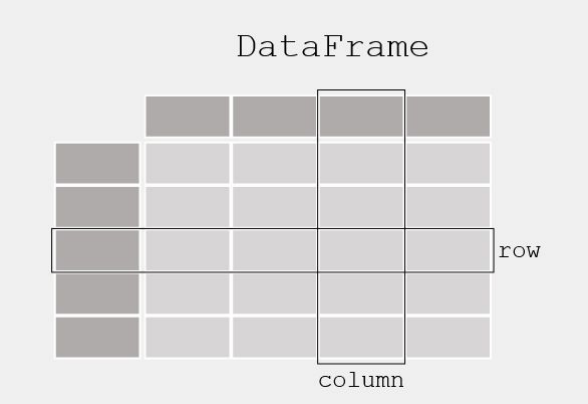
• Any exception that we don’t catch is called an unhandled exception and causes the program to crash.

Pandas
(Not Generally Covered in This Class, but a Useful Tool)

Overview

• When working with tabular data, such as data stored in spreadsheets or databases, pandas is often the right tool for

• In Pandas, data is stored in a data table called a DataFrame:



Reading Data

• Pandas supports the integration with many file formats or data sources out of the box (csv, excel, sql, json, parquet,…).

• Importing data from each of these data sources is provided by function with the prefix read_*

• For Example:
    • read_csv
    • read_json
    • read_html

What Else?

• From there, you can access, update, slice, etc. In a similar way to other data types (just with different syntax)

• You can easily add, remove, and iterate over rows and columns

• Pandas also has built in functions to calculate basic statistics like mean, median, min, max, counts, etc.

• You can also reshape data and easily plot the data.

• Any many more!

Conclusion

• Pandas is great and is commonly used in industry. It can easily handle tabular data and do everything you would need to do to the data.

• Unfortunately, do to the nature of this class, and what I am supposed to prepare you for in future courses, for assignments you should use the 4 data types we covered in class and **NOT** use pandas

    • For your Final Project, you **may** use pandas however

Today in Tech History

October 19, 1979

According to Dan Bricklin, the first “real” release of VisiCalc was completed and packaged for shipment. VisiCalc was the first commercially available spreadsheet software and quickly became the first “killer app” of the personal computer market.