File & Database I/O

Introduction to Data Science with Python

Read/Write Files

- A very common operation in an programming language is reading and writing files
- Open a file as read-only or read/write

```
# Read a text file
countries = open(“countries.txt“, “r”)
# Print out the results
for line in countries:
    print(line)


# Close when done
countries.close()
```

Write Files

- The converse to reading is writing

```
# Writing text to a file
test_file = open(“./test_file.txt“, “w”)
# Write something
test_file.write(“Hello World! \nGood-by World!“)
# Close when done
test_file.close()
```

Using with

- Python has the with command to perform garbage collection
- Use with to ensure that open handles/connections are closed and destroyed automatically

  Note that objects declared using with exist for the code block only

```
## Use with to open files to ensure closure

with open(r"..\data\green_eggs_ham.txt") as f1:

    dat = f1.read()

print(dat)
```

Iterate through a file

- You don't have to read a file all at once. You can read one line at a time using the readline method
- Or you can read the file in as a list of strings using the readlines method

```
## Process files one line at a time
with open(r"..\data\green_eggs_ham.txt") as f1:
    dat = f1.readline()
print(dat)
## read in lines as a list of strings
with open(r"..\data\green_eggs_ham.txt") as f1:
    dat1 = f1.readlines()
dat1
```

Test Your Knowledge – Ex 8.1

- Set your working directory in code
- Open the states file
- Read the file and print out each individual state
- Close the file

Creating Directories

- Use the **os** module to perform file system operations
  - Check if a directory or file exists
  - Create a directory
  - Remove a directory (or file)

```
log_path = "../Datasets/Logs"
if not os.path.exists(log_path):
    os.mkdir(log_path)
```

Saving Python Objects

- There are a number of ways to save an object to a file
  - One of the most popular is the JSON format, used as a common exchange between different systems
  - Python provides the json module for this

```
import json
obj = [1,"the",4]

## text representation of file
json.dumps(obj)
```

Writing JSON to a file

- The json module provides the dump method for outputting an object to a json file

```
import json
out_list = [1,"the",4]

# Write JSON out to a file
with open(r"..\output\test.json") as output_file:
    json.dump(out_list, output_file)

## Load the results back from JSON
with open("..\output\test.json","r") as input_file:
    in_list = json.load(input_file)
```

Serializing Data - Pickle

- JSON can be a little slow and hefty for some applications
- The pickle module allows Python to save and load the binary object directly

```
import pickle
test_obj = {"x":[1,2],"y":[3,4]}

pickle.dump(test_obj,open("temp/test_obj.p", "wb"))
with open("temp/test_obj.p", "wb") as output_file:
    pickle.dump(test_obj, output_file)

# Load the object back
del test_obj
with open("temp/test_obj.p","rb") as input_file:
    test_obj = pickle.load(input_file)
```

Working with CSV files

- CSV files are among the most common transfer formats, especially for tabular data
- Python also provides a module for working directly with CSV files

```
# Read/Write CSV files
import csv
with open('temp/age.csv', 'w’, newline='\n') as output_file:
    writ = csv.writer(output_file)
    writ.writerow(['id','age'])
    writ.writerow(['01', '22'])
    writ.writerow(['02', '26'])
```

CSV Delimiters

- By default, CSV files use the comma as a delimiter, but not always

```
## Change delimiter
import csv
with open('temp/age.csv', 'w’, newline='\n') as output_file:
    writ = csv.writer(output_file,delimiter='\t',
                                quotechar = '"', quoting=csv.QUOTE_MINIMAL)
    writ.writerow(['id','age'])
    writ.writerow(['01', '22'])
    writ.writerow(['02', '26'])
```

Iterating Through an Object

- With CSVs you get a series of rows you can iterate through like any other collection

```
# Iterating through an object to write
import csv
import numpy as np
out_res = np.arange(10).reshape(5,2)
with open('temp/sim1.csv', 'w',newline='\n’) as output_file:
    writ = csv.writer(output_file)
    writ.writerow(['v1','v2'])
    for i in out_res:
        writ.writerow(i)

# Reading in csv
with open('temp/sim1.csv', 'r') as input_file:
    writ = csv.reader(input_file)
    for i in writ:
        print(i)
```

Enumerating Rows

- You don’t have to read the whole thing either

```
# Only read in the first two rows: enumerate
with open('temp/sim1.csv', 'r’) as input_file:
    writ = csv.reader(input_file)
    for i,row in enumerate(writ):
        if i < 3:
            print(row)
        if i == 3:
            break
```

Connecting to a Database

- Another common operation is connecting to a database
  - Use the pyodbc driver to connect to any ODBC-compliant database (which is most of them)

```
import pyodbc
import pandas as pd

# ODBC log in to database - untrusted
cnxn = pyodbc.connect('DRIVER={ODBC Driver 13 for SQL Server};SERVER='
\

        + server + ';DATABASE=' + database + ';UID=' + username \

        + ';PWD='+ password)

cursor = cnxn.cursor()

cursor.close()
cnxn.close()
```

Connecting cont.

- You are not stuck with using ODBC either

  There are many alternatives depending upon how much research you want to do

```
# OLEDB - Log in to SQL Server database - untrusted
cnxn = pyodbc.connect('Driver={SQL Server Native Client 11.0};SERVER=' \

        + server + ';DATABASE=' + database + ';UID=' + username \

        + ';PWD='+ password)

cursor = cnxn.cursor()

cursor.close()
cnxn.close()
```

Get Data

- Pulling data from a database requires running a select statement and assigning the result to a Pandas data frame
    - Use the read.sql function

```
# get some data
sql = "SELECT * FROM Person.Person"

data = pd.read_sql(sql, cnxn)

cursor.close()
```

Insert/Update/Delete

- You can add, delete and update rows with the execute method
  - You can also call stored procedures

```
# execute a command
sql = "insert into TestTable VALUES('This is a test')"

cursor = cnxn.cursor()

number_of_rows = cursor.execute(sql)

# you need to call commit() method to save
# your changes to the database
cursor.commit()
```

Using SQLLite

- Sometimes accessing a relational database is not that easy (like in a class)
- We can build our own using a module call sqllite3
- With this module we can create a local relational database engine which uses a bare-bones ANSI SQL
- We first need to create the database engine

```
# create engine
import sqlite3
conn = sqlite3.connect('example.db')

c = conn.cursor()
```

Running Commands

- We can use the same execution process with the local database

```
c.execute(''' CREATE TABLE person
        (id INTEGER PRIMARY KEY ASC, name varchar(250) NOT NULL) ''')
c.execute(''' CREATE TABLE address
        (id INTEGER PRIMARY KEY ASC, street_name varchar(250), street_number varchar(250),
         post_code varchar(250) NOT NULL, person_id INTEGER NOT NULL,
         FOREIGN KEY(person_id) REFERENCES person(id)) ''')

c.execute("INSERT INTO person VALUES(1, 'pythoncentral')")
c.execute("INSERT INTO address VALUES(1, 'python road', '1', '00000', 1)")

conn.commit()
conn.close()
```

Lab 8 – File & Database IO

- Do the File IO and Database exercises in Lab 8 (45 min)