Importing data with Python
by datacamp

plain text file > .txt
‘flat’ files > ie .csv

Reading a text file
```
filename = ‘huck_finn.txt’
file = open(filename, mode=‘r’)
text = file. read()
file.close()
```
‘mode’ set to r makes sure that we can only read it
can set it to w to write
*this ensures that we don’t accidentally write to it
now you will be able to print the file to your console to check it out
best practice to always close your connection

a ‘with’ statement allows you to create a context in which you can execute commands with the file open
```
with open(‘huck_finn.txt’, ,’r’) as file:
        print(file.read())
```
this can help you to avoid having to directly close the connection
this with statement is called a ‘context manager’
once out of this clause/context, the file is no longer open
this is called ‘binding’ a variable in the context manager construct

Flat files
text files containing records
unstructured table data
**this is in contrast to a relational database in which columns of distinct tables can be related
a record means a row of fields or attributes, each of which contains at most one item of information
example in the titanic.csv file
a record is a unique passenger onboard and each column is a feature or attribute such as name, gender, and cabin
a flat file can have a header
a header is a row that occurs as the first row and describes the contents of the data columns
or states what th corresponding attributes or features in each column are
**important to know if your data has a header as it may alter your data import

values in flat files can be separated by more than just commas
ie a tab
these separations are called delimiters

Assigning data to a variable
if the data is numerical, we can use the numpy package to import that data as a numpy array
numpy arrays are standard for storing numerical data
fast, efficient, clean
essential for other packages like scikit-learn

using numpy function loadtxt
filename = ‘MNIST.txt’
data = np.loadtxt(filename, delimiter=‘,’, skiprows=1, usecols=[0,2])
data
*a delimiter can be whitespace, always imperative to specify
‘skiprows’ argument here allows you to skip over the first row
in this example allows you to skip over the header and string MNIST
‘usecols’ argument allows you to choose which columns to import
in this example allows us to import the first and third columns of the data
can also use argument ‘dtype’ to set all entries to a specific data type
in the above example we could have done dtype=str which would import all entries as strings
**be aware loadtxt is great for basic cases but tends to break down when we have mixed datatypes

np.genfromtxt()
more versatile than np.loadtxt() can deal with more datatypes
’names’ argument to True tells us there is a header
**because the data are of different types, data is an object called a structured array
**because numpy arrays have to contain elements that are all the same datatype, the structured array solves this by being a 1D array, where each element of the array is a row of the flat file imported

np.recfromcsv() is similar to np.genfromtxt() except that its default dtype is None
also has default delimiter ‘,’ and default names=True

**numpy arrays are limited in their ability to have 2D labeled data structures with columns of potentially different types
this is where pandas DataFrames come in
they allow you to manipulate, slice, reshape, groupby, join, merge, perform statistics, work with time series data

pandas was created by Wes McKinney
**this library and package allow Python to act like R
pandas DataFrame = pythonic analog of R's data frame
"A matrix has rows and columns. A data frame has observations and variables."

Pickeled files
file type native to Python
reason for existence - for datatypes for which it isn't obvious how to store them
pickled files are serialized
meaning able to be imported into Python, converting the object into a sequence of bytes or byte stream
example

```
import pickle
with open('pickled_fruit.pkl', 'rb') as file:
        data = pickle.load(file)
print(data)
```

'rb' argument is r for read-only and b for binary meaning computer-readable and not human-readable

Importing Excel spreadsheets

```
file = 'urbanpop.xlsx'
data = pd.ExcelFile(file)
print(data.sheet_names)
```

Excel file often contains sheets
can parse them out
this example

```
df1 = data.parse('1960-1966') #as a string
df2 = data.parse(0) #same sheet, calling index, setting as a float
```

Importing SAS and Stata files
SAS - Statistical Analysis System - popular in business and biostatistics
Stata - 'statistics' + 'data' - popular in economics and epidemiology

SAS is used for:
advanced analytics, multivariate analysis, business intelligence, data management, predictive analytics, and standard for computational analysis

most commons SAS files extensions are .sas7bdat and .sas7bcat
example

```
import pandas as pd
from sas7bdat import SAS7BDAT
with SAS7BDAT('urban pop.sas7bdat') as file:
        df_sas = file.to_data_frame()
```

in this case binding the variable file to a connection urban pop in a context manager

Stata file example
import pandas as pd
data = pd.read_stata(‘urbanpop.dta’)

Importing HDF5 files
-Hierarchical Data Format version 5
-standard for storing large quantities of numerical data
HDF5 can scale to exabytes

- 1 exabyte is equal to 1,000 petabytes (PB).
- 1 exabyte is equal to 1,000,000 terabytes (TB).
- 1 exabyte is equal to 1,000,000,000 gigabytes (GB).
- 1 exabyte is equal to 1,000,000,000,000 megabytes (MB).
- 1 exabyte is equal to 1,000,000,000,000,000 kilobytes (KB).

```
import h5py
filename = ‘H-H1.LOSC.hdf5’
data = h5py.File(filename, ‘r’) # ‘r’ is for read
print(type(data))
```

HDF5 files have an interesting hierarchical structure
structure is similar to a dictionary
the keys are meta, quality, strain
the file data is appropriately organized off these keys

example using a LIDO HDF5 file

```
for key in data[‘meta’].keys():
        print(key)
```

Output > Description, Detector, Duration, GPSstart, Observatory
could then access any metadata of interest say ‘Description’ and ‘Detector’

```
print(np.array(data[‘meta’][‘Description’]), np.array(data[‘meta’][‘Detector’]))
```

Output > ’Strain data time series from LIGO’ ‘H1’
data in the file and then which detector was used

Importing MATLAB
stands for Matrix Laboratory
-industry standard in engineering and science
data saved as .mat files

How to import these files into Python
SciPy package to the rescue

scipy.io.loadmat() - read .mat files
scipy.io.savemat() - write .mat files
import scipy.io
filename = ‘workspace.mat’
mat = scipy.io.loadmat(filename)
print(type(mat))
output > class ‘dict’

How this dictionary relates to MATLAB
keys are MATLAB variable names
values are MATLAB objects that are assigned to the variables

Introduction to relational databases
database that is based upon the Relational model of data

The relational model of data is a conceptual framework for representing and organizing data in a relational database.
It was introduced by E.F. Codd in 1970 and has since become the most widely used data model in the field of database management systems.
The relational model represents data as a collection of tables, also known as relations.
Each table consists of rows (tuples) and columns (attributes).
The rows represent individual records or instances, while the columns represent the attributes or properties of those records.
Key principles of the relational model include:

- Tabular Structure: Data is organized into tables with rows and columns. Each table has a unique name and consists of a set of attributes with defined data types.
- Primary Keys: Each table has a primary key that uniquely identifies each row within the table. The primary key ensures the uniqueness and integrity of the data.
- Foreign Keys: Relationships between tables are established using foreign keys. A foreign key in one table refers to the primary key in another table, creating associations or dependencies between the tables.
- Data Integrity: The relational model enforces integrity constraints, such as referential integrity, entity integrity, and data type constraints, to ensure the accuracy and consistency of the data.
- Relational Algebra: The relational model provides a set of algebraic operations, such as selection, projection, join, and union, to manipulate and retrieve data from the tables.
- Normalization: The relational model promotes data normalization, which involves breaking down tables into smaller, well-structured tables to eliminate redundancy and improve data integrity.

The relational model offers a flexible and intuitive way to organize and manage data.
It enables efficient storage, retrieval, and manipulation of data through the use of standardized operations and constraints.
Relational databases, such as MySQL, PostgreSQL, Oracle, and SQL Server, are built based on the relational model and provide powerful tools for managing structured data.

Codd’s 12 Commandments

- Rule 0: The Foundation Rule
  - A relational database management system must be based on a solid foundation, providing a sound theoretical basis for data management.
- Rule 1: The Information Rule
  - All information in a relational database is represented explicitly as values in tables.
- Rule 2: Guaranteed Access Rule
  - Every single value in a relational database is accessible logically by using a combination of the table name, primary key, and column name.
- Rule 3: Systematic Treatment of Null Values
  - Null values are supported and treated systematically to indicate missing or unknown information.
- Rule 4: Active Online Catalog
  - The structure and metadata of the database, including table definitions, relationships, and integrity constraints, are stored in the system catalog and can be queried like other data.
- Rule 5: Comprehensive Data Sublanguage Rule
  - The database system should support a complete, non-procedural language that enables users to define, query, and manipulate the data in the database.
- Rule 6: View Updating Rule
  - All views that are theoretically updatable should also be updatable through the system.
- Rule 7: High-Level Insert, Update, and Delete
  - The database system should support high-level insert, update, and delete operations, allowing users to modify data in a simple and intuitive manner.
- Rule 8: Physical Data Independence
  - The application programs and activities of users should remain unaffected by changes in the physical storage structure or access methods.
- Rule 9: Logical Data Independence
  - Changes in the logical structure (e.g., table definitions, relationships) should not affect the existing application programs.
- Rule 10: Integrity Independence
  - Integrity constraints, such as primary key, foreign key, and other data

validation rules, should be specified and enforced separately from application programs.

- Rule 11: Distribution Independence
    - The distribution of the data across different locations or sites should be invisible to users and applications.

PostgreSQL, MySQL, SQLite are all relational database management systems that use SQL query language
SQL stands for structured query language

There are many packages we could use to access an SQLite database
example sqlite3 or SQLAlchemy
SQLAlchemy works with many other relational database management systems
Let’s connect
example
from sqlalchemy import create_engine
engine = create_engine(’sqlite:///Northwind.sqlite')
the create_engine function fires up an SQL engine that will communicate our queries to the database
before we connect, we would like to know the names of the tables
table_names = engine.table_names()
print(table_names)

Basic SQL query
SELECT * From Table_Name
-returns all columns of all rows of the table

Workflow of SQL querying
import packages and functions
create the database engine
connect to the engine
query the database
save query results to a DataFrame
close the connection
example
from sqlalchemy import create_engine
import pandas as pd
engine = create_engine(‘sqlite:///Northwind.sqlite')
con = engine.connect()
rs = con.execute(“SELECT * FROM Orders”) #rs stands for ‘relational SQL query’
df = pd.DataFrame(rs.fetchall()) #fetchall fetches all rows
df.columns = rs.keys() #setting the dataframe’s column names
con.close()

check to ensure all went through as desired
```
print(df.head())
```

**can also use the context manager construct which will save you the trouble of potentially forgetting to close the connection
```
from sqlalchemy import create_engine
import pandas as pd
engine = create_engine(‘sqlite:///Northwind.sqlite')
with engine.connect() as con:
        rs = con.execute(“SELECT OrderID, OrderDate, ShipName FROM Orders”)
        df = pd.DataFrame(rs.fetchmany(size=5))
        df.columns = rs.keys()
```

**can do this all in one line of code
```
df = pd.read_sql_query(“SELECT * FROM Orders”, engine)
#first argument is query that you want
#second argument is engine that you want to connect to
```

**frequently will need to join tables
example on inner join
```
from sqlalchemy import create_engine
import pandas as pd
engine = create_engine(’sqlite:///Northwind.sqlite')
df = pd.read_sql_query(“SELECT OrderID, CompanyName FROM Orders INNER
JOIN Customers on Orders.CustomerID = Customers.CustomerID”, engine)
print(df.head())
```