Importing Data from a .csv File

Brandon Krakowsky

Penn Engineering

Loading Data – Open a File

- Python has lots of built-in functionality for *reading* and *writing* files
- Before *reading* a file, you have to *open* a file
  - You can use the built-in *open()* function
- Format: open(*file_name*, *access_mode*)
  - *file_name* is the name of the file (including the file path)
  - *access_mode* is an optional parameter indicating the *mode* in which the file is opened, i.e., read, write, append, etc. The default value is read (*r*)
- Example:

```
my_file = open('my_file_path/my_file.csv', 'r')
```

  - This would create a *file object* for the 'my_file_path/my_file.csv' file in *read (r)* mode

For reference: https://docs.python.org/3/library/functions.html#open

Loading Data – Access Modes

- Access *modes* for the *open()* function

  r opens a file for reading only – the default mode

  w opens a file for writing only – overwrites the file if it exists and creates a new file if it does not

  a opens a file for appending with the cursor at the end of the file -- creates a new file if it does not exist

  r+ opens a file for both reading and writing

For reference: https://docs.python.org/3/library/functions.html#open

Loading Data – csv Module

- The *csv* module implements classes to read and write .csv (comma-separated values) files
- It provides a useful *DictReader* class which reads a .csv file and maps the information into a *dict* object, behind the scenes

```
import csv
my_file = open('my_file_path/my_file.csv', 'r')
reader = csv.DictReader(my_file)
```

For reference: https://docs.python.org/3/library/csv.html

Loading Data – Close a File

- After *reading* a file, you should *close* the file
  - You can use the built-in *close()* function
- This flushes any unwritten information and closes the file object
  - After it's closed, no more reading/writing can be done!
- It is good practice to use the *close()* function to close a file
- Example:

```
import csv
my_file = open('my_file_path/my_file.csv', 'r') #open the file
reader = csv.DictReader(my_file) #read the file
#do some stuff here with reader
my_file.close() #close the file
```

For reference: https://docs.python.org/3/library/csv.html

Loading Data – Using *with*

- Or, you can use the *with* statement to open a file and close it all at once

```
import csv
with open('my_file_path/my_file.csv', 'r') as csvfile:
    reader = csv.DictReader(csvfile)
    #do some stuff here with reader
```

- *with* will take care of closing the file for you!

For reference: https://docs.python.org/3/library/csv.html

Analyzing The 500 Greatest Albums of All Time

Our Data – “The 500 Greatest Albums of All Time”

- Rolling Stone magazine's 2012 list of 500 greatest albums of all time with genres
- Courtesy @notgibs (https://data.world/notgibs)
  - Read more about “The 500 Greatest Albums of All Time”: https://en.wikipedia.org/wiki/Rolling_Stone's_500_Greatest_Albums_of_All_Time
- Data columns:
  - Number: Position on the list
  - Year: Year of release
  - Album: Album name
  - Artist: Artist name
  - Genre: Genre name
  - Subgenre: Comma-separated list of subgenre names

Loading Data – “The 500 Greatest Albums of All Time”

- Let’s load and read the ‘albumlist.csv’ file

```
import csv
with open(’albumlist.csv', 'r') as csvfile:
    reader = csv.DictReader(csvfile)
```

  - Note: ‘albumlist.csv’ is in the same directory as the Python script
  - This loads the file into *reader*

- *reader* is a DictReader

```
print(type(reader))
```

- You can look at the column headers by accessing the *fieldnames* attribute

```
print(reader.fieldnames)
```

Querying Data – Using *for loop*

- Let's print each row using a *for loop*

```
with open('albumlist.csv', 'r') as csvfile:
    reader = csv.DictReader(csvfile)
    for row in reader:
        print(row)
```

  - This loads the file into *reader* and *iterates* over (steps through) it to print each row
- Important Note: By default, a DictReader only allows you to *iterate* over it once!
  - This is because a DictReader reads the file directly (does not store the data in memory)
  - There are workarounds to this, but don't try them (for now)
  - The best solution is to reload the file every time you need to use the DictReader

Querying Data – Limiting Using *for loop*

- Let's print the first 100 rows of the data using a *for loop*

```
with open('albumlist.csv', 'r') as csvfile:
    reader = csv.DictReader(csvfile)
    rows = 101
    for row in reader:
        rows -= 1
        if (rows > 0):
            print(row)
        else:
            break
```

  - This loads the file into *reader* and *iterates* over (steps through) it, decrementing a *rows* count
  - If *rows* is greater than 0, it prints the row, otherwise it breaks out of the *for loop*

Querying Data – Using *for loop*

- Using a *for loop*, let's add each row to an empty list and get the number of records in that list

```
with open('albumlist.csv', 'r') as csvfile:
    reader = csv.DictReader(csvfile)
    albums = []
    for row in reader:
        albums.append(row)

    print("Number albums:", len(albums))
```

  - For each record (*row*) in *reader*, add to a new list named *albums*
  - Get the length of the list and print it (should be 500!)
- Note: we can now use the *albums* list for the remainder of our code
  - No need to re-open the file and read into a DictReader

Querying Data – Using List Comprehension

- Another way to add records to a list is by using *list comprehension*, an elegant and concise way to create a new list from a data structure
  - Consists of an *expression* followed by a *for loop* inside square brackets []
  - Meaning: Do something for each iteration of the loop and add the result to a new list
- Using *list comprehension*, let's get the number of albums released in 1974

```
albums_1974 = [row for row in albums if row[“Year”] == "1974”]
print("Number albums in 1974:", len(albums_1974))
```

  - For each album (*row*) in our *albums* list where the “Year” is 1974, append to a new list named *albums_1974*

Querying Data – Using List Comprehension

- Now let's print the albums released in 1974, along with the artists

```
for album in albums_1974:
    print(album["Album"], "by", album["Artist"])
```

  - For each album in *albums_1974*, print the album title ("Album") and artist name ("Artist")

Querying Data - Limiting Using List Comprehension

- What if we only want to list 10 of the albums released in 1974?

```
print([row for row in albums if row["Year"] == "1974"][:10])
```

  - For each album (*row*) in our *albums* list where the “Year” is 1974, append to a new list named *albums_1974* and take a 10 record *slice*

Querying Data - Exercise

- Print the Album, Artist, Genre, and Subgenre for albums in the “Rock” genre and in the “Pop Rock” or “Fusion” subgenre
  - Remember, the “Subgenre” column contains a *list* of values!

```
rock_albums = [row for row in albums
    if (row["Genre"] == "Rock" and
        ("Pop Rock" in row["Subgenre"] or "Fusion" in
row["Subgenre"]))]

for album in rock_albums:
    print(album["Album"], album["Artist"], album["Genre"],
album["Subgenre”])
```

Querying Data – Catching Data Errors

- What's the earliest release year for any album in our dataset?
- First, let's get a list of release years, casting each to an integer
  `release_years = [int(row['Year']) for row in albums if row['Year']]`
  - For each object (*row*) in our *albums* list where "Year" exists, cast "Year" to an integer and add to a new list named *release_years*
- Did you get a ValueError?
  - That's because there are "Year" values that are not valid numeric years and can't be casted to an integer
  - e.g. "Year" for one of the songs is "nineteen seventy three"
- We need to account for this by ensuring we are dealing with integers

Querying Data – Catching Data Errors

- Let's define a function that checks if a value can be casted to an integer by catching a ValueError

```
def is_valid_year(string):
    try:
        year = int(string) #try casting string to an integer named year
    except ValueError:
        return False #return False if a ValueError is generated
    else:
        return year #otherwise, return the year value
```

Querying Data – Catching Data Errors

- Then use *is_valid_year* to eliminate values that do not have a valid year

```
release_years = [int(row['Year']) for row in albums if is_valid_year(row['Year'])]
print(release_years)
```

  - For each object (*row*) in our *albums* list where "Year" exists and is valid, cast "Year" to an integer and add to a new list named *release_years*
  - Print the *release_years* list!

Querying Data – Catching Data Errors

- Get the minimum value from the *release_years* list and print it

```
min_release_year = min(release_years)
print(min_release_year)
```

- Did you get a year that is before 1400? That can't be right …
- Let's update our *is_valid_year* function to account for those years

```
def is_valid_year(string):
    try:
        year = int(string) #try casting string to an integer named year
    except ValueError:
        return False #return False if a ValueError is generated
    else:
        return year > 1400 #otherwise, return the year, if it's after 1400
```

Querying Data – Catching Data Errors

- Then use *is_valid_year* to eliminate values that do not have a valid year or are before 1400

```
release_years = [int(row['Year']) for row in albums if
is_valid_year(row['Year'])]
```

- Get the minimum value from the *release_years* list and print it

```
min_release_year = min(release_years)
print(min_release_year)
```

- Much better!

Lambda Functions

- A *lambda* function is a one-line mini-function defined on the fly
  - They can be used anywhere a function is required
- For example, here's a simple function *double* that doubles an argument *x* by 2

```
def double(x):
    return x * 2

double(4)
```

- We can also define a *lambda* function *double_l* that does the same thing

```
double_l = lambda x: x * 2 #given parameter x, return x * 2
double_l(4)
```

Lambda Functions – With Sorting

- *Lambda* functions are especially useful as arguments to other functions
- For example, in the *sorted* function:
  sorted(*list*, *key*) – sorts the *list* based on the optional *key* (*lambda* function)
  - *key* optionally specifies a function that tells *sorted* what to sort by
  - The default value (None) compares the elements directly

For reference: https://docs.python.org/3.5/library/functions.html#sorted

Lambda Functions – With Sorting

- What if we want to sort our albums by artist (“Artist”)?

```
albums_sorted = sorted(albums, key = lambda x: x["Artist"])
print(albums_sorted)
```

  - When sorting *albums*, use *key* to compare them
  - *key* is a lambda function -- given parameter x (row), use the “Artist” column to compare

For reference: https://docs.python.org/3.5/library/functions.html#sorted

Lambda Functions

- Some other functions that accept *lambda* functions as arguments:

  max(*list*, *key*) – returns the maximum value from the *list* based on the optional *key* (*lambda* function)
  - *key* optionally specifies a function that tells *max* what to sort by
  - The default value (None) compares the elements directly

  min(*list*, *key*) – returns the minimum value from the *list* based on the optional *key* (*lambda* function)
  - *key* optionally specifies a function that tells *min* what to sort by
  - The default value (None) compares the elements directly

For reference: http://www.diveintopython.net/power_of_introspection/lambda_functions.html

Lambda Functions – Exercise

- What's the Album, Artist, and release Year for the most recent album in the data?
  - Hint: Ignore albums that do not have a valid year!

```
valid_albums = [row for row in albums if is_valid_year(row['Year'])]
album_max = max(valid_albums, key = lambda x: x["Year"])
print(album_max["Album"], album_max["Artist"], album_max["Year"])
```