June 7, 2023

1 Lab 1

[ ]:
```
import pandas as pd
import numpy as np
```

1.1 Part 1. Practicing NumPy

1.1.1 a. Define a new Python *list*

Let us practice defining a new list in Python:

[ ]:
```
my_list = [0,0,1,2,3,3,4.5,7.6]
my_list
```

1.1.2 b. (Two ways to) define a *range*

One type of list is a range of (e.g., integer) numbers. Ranges are useful for iterating over in a loop -- that is, to assign some variable to take on each value in this lest sequentially. (For example, `i=1`, then `i=2`, then `i=3`, etc., until `i=100`.)

Let's create an evenly spaced array of integers ranging from 0 to 11:

[ ]:
```
#First, the basic python way:
my_range = range(0,12)
my_range
```

[ ]:
```
#Now the numpy way:
my_range_np = np.arange(12)
my_range_np
```

You can see that the first method returns a special 'range' object, while the latter returns an object of type 'numpy array'. If we convert both to lists, they will be the same:

[ ]:
```
list(my_range)
```

[ ]:
```
list(my_range_np)
```

1.1.3 c. List comprehension

Consider the task of replacing each value in a list with its square. The traditional way of performing te same transformation on every element of a list is via a *for* loop:

[ ]:
```
my_range_np = np.arange(12)

for i in range(0, len(my_range_np)):
    my_range_np[i] = my_range_np[i]**2 #square each element
print(my_range_np)
```

That worked. However, there is a better, more 'Pythonic' way to do it.

List comprehension is one of the most elegant functionalities native to Python, and is beloved by programmers like you. It offers a concise way of applying a particular transformation to every element in a list.

Using list comprehension syntax, we can write a single, easily interpretable line that does the same transformation without using ranges, nor doewe have to introduce an iterating index variable `i`:

[ ]:
```
my_range_np = np.arange(12)

my_range_np = [x**2 for x in my_range_np]
my_range_np
```

1.1.4 d. Creating a one-dimensional NumPy *array*.

Let's explicitly create a one-dimensional numpy array (as opposed to a list):

[ ]:
```
arr = np.array([1, 2, 3, 4])
arr
```

1.1.5 e. Retrieving the dimensions of data structure: len() and np.shape()

How would we go about creating a variable that contains the length of our array `arr` ? We could use the python function `len()`...

[ ]:
```
arr_length = len(arr)
arr_length
```

... or use the numpy method `np.shape`, saving only the first of the two values that it returns:

[ ]:
```
arr_length = np.shape(arr)[0]
arr_length
```

(Tip: try removing the slice indicator `[0]` and see how the printout changes. Notice that there appears to be an empty 'slot' for another number in the returned value pair)

1.1.6 f. Creating a uniform (same value in every position) array: np.ones()

We will now use `np.ones` to create an array of a pre-specified length that contains the value '1' in each position:

[ ]:
```
length = 55
np.ones(length, dtype=int)
```

We can use this method to create an array of any identical values. Let's create an array of length 13, filled with the vlue '7' in every position:

[ ]:
```
7*np.ones(13, dtype=int)
```

1.1.7 g. Creating two-dimensional arrays

Exploring the possibilities of the `np.array` method further, let's move on to creating not one- but two-dimensional arrays (aka matrices):

[ ]:
```
M = np.array([[1,2,3], [4,5,6]])
M
```

[ ]:
```
np.shape(M)
```

Numpy contains useful methods for creating identity matrices of a specified size:

1.1.8 h. Creating an identity matrix: np.eye()

[ ]:
```
np.eye(5)
```

[ ]:
```
#check your intuition: what will be the printout after running this cell?
A = np.eye(3)
B = 4*np.eye(3)
A+B
```

1.1.9 i. A small challenge: matrix transformation; random matrix generation

Using the matrix `M` below and the function `np.triu` (pull up the documentation by typing `np.triu?`), create a matrix `N` which is identical to `M` except in the lower triangle (i.e., all the cells below the diagonal). The lower triangle should be filled with zeros.

[2]:
```
np.triu?
```

[ ]:
```
M = np.round(np.random.rand(5,5),2)
print("M=\n", M)

## your code here:
# N =

## solution:
# N = np.triu(M, k=-1)
#print("N=\n", N)
```

Using the code provided above for generating the matrix `M`, try to figure out how to create a random matrix with 13 rows and 3 columns.

[ ]:
```
## your code here:
# M =

## solution:
M = np.random.rand(13,3)
print("M=\n", M)
```

1.1.10 j. Indexing arrays

Here is how to call an element of a 2D array by its location (i.e., its row index and column index):

[ ]:
```
M[3,2]
```

[ ]:
```
# test your intuition: what would you expect this code to return?
M[3:,2]
```

1.1.11 k. Evaluating a Boolean condition

In real-life data tasks, you will often have to compute the boolean `(True/False)` value of some statement, for all entries in a list, or for a matrix column (essentially, a list), or for the entire matrix. In other words, we may want want to formulate a condition -- think of it as a *test* -- and run a computation that returns `True` or `False` depending on whether the test is passed or failed by a particular value in a data structure.

For example, our test may be something like "the value is greater than 0.5", and we would like to know if this is true or false for each of the values in a list. Here's how to compute a list of values that the condition test takes for each value of a given list:

[ ]:
```
g = np.random.rand(1, 20) #first, create the list
print(g)
```

[ ]:
```
is_greater = g>0.5
print(is_greater)
```

[ ]:
```
# Let's print the matrix M again so we can glance at it for our next exercise
print(M)
```

[ ]:
```
# What would you expect to see once you run the code below?
c_is_greater = M[:,1]>0.5
c_is_greater
```

1.1.12 L . Nympy functions `np.any`, `np.unique`

We can use `np.any` to determine if there is any entry in column 1 that is smaller than 0.1:

[ ]:
```
c_is_smaller = M[:,1]<0.1
np.any(c_is_smaller)
```

A small challenge: You have birthday data for a cohort of 100 people all born in 1990. Given the one-dimensional array of birthdays `random_bdays` generated below, figure out if there exists a pair of people who share a birthday. (Tip: you may find the function `np.unique()` useful. Feel free to read up on it by printing `np.unique?` in a new cell)

[ ]:
```
# do not edit this code:
random_nums = np.random.choice(365, size = 100)
random_bdays = np.datetime64('1990-01-01') + random_nums

## your code here:
# duplicates_exist =
```

```
## solution:
# duplicates_exist = len(random_bdays)!=len(np.unique(random_bdays))
# duplicates_exist
```

1.2 Part 2. Pandas DataFrames

1.2.1 a. Creating a DataFrame: two (of the many) ways

We will overview two ways of creating Pandas dataframes from scratch: from a *list of lists*, and from a *dictionary*.

[ ]:
```
my_list = [['+1', '(929)-000-0000'], ['+34', '(917)-000-0000'], ['+7',␣
↪'(470)-000-0000']]
df = pd.DataFrame(my_list, columns = ['country_code', 'phone'])
df
```

[ ]:
```
my_dict = {'country_code': ['+1', '+34', '+7'], 'phone':['(929)-000-0000',␣
↪'(917)-000-0000', '(470)-000-0000']}
df = pd.DataFrame(my_dict)
df
```

1.2.2 b. Adding a column to a DataFrame object

Below, we add a new column of values to the dataframe:

[ ]:
```
df['grade']= ['A','B','A']
df
```

1.2.3 c. Sorting the dataframe by values in a specific column: `Pandas.df.sort_values`

[ ]:
```
df.sort_values(['grade'], axis = 0)
```

In real life settings, you will often need to combine separate sets of related data. To illustrate, let's create a second dataframe:

1.2.4 d. Combining multiple dataframes: `Pandas.concat` and `Pandas.merge`. Renaming the columns: `df.rename()`

[ ]:
```
my_dict2 = {'country': ['+32', '+81', '+11'], 'grade':['B', 'B+', 'A'], 'phone':
↪['(874)-444-0000', '(313)-003-1000', '(990)-006-0660']}
df2 = pd.DataFrame(my_dict2)
df2
```

Use the Pandas `pd.concat` method to append the second dataframe to the first one:

[ ]:
```
pd.concat([df,df2])
```

One immediate problem with this result is that the index has repeated values. This defeats the purpose of an index, and ought to be fixed. Let's try the concatenation again, this time adding `reset_index()` method to produce correct results:

[ ]:
```
pd.concat([df,df2]).reset_index()
```

Much better! This result is valid, but it contains redundancy caused by different spelling of the country code column in the two dataframes. This can be easily fixed:

[ ]:
```
df2 = df2.rename(columns={'country':'country_code'})

pd.concat([df,df2]) # aha!
```

What if our task was to merge `df2` with yet another dataset -- one that contains additional unique columns?

[ ]:
```
df2
```

[ ]:
```
my_dict3 = {'country_code': ['+32', '+44', '+11'], 'phone':['(874)-444-0000',␣
↪'(575)-755-1000', '(990)-006-0660'], 'grade':['B', 'B+', 'A'], 'n_credits':␣
↪[12, 3, 9]}
df3 = pd.DataFrame(my_dict3)
df3
```

Feel free to consult the definition of the Pandas `merge` method to better undertsand the next cell:

[ ]:
```
df2.merge(df3, on = 'phone')
```

1.2.5 e. Loading a dataset: `Pandas.read_csv`

We are now well equipped to deal with a real dataset!

For the next few exercises, our dataset will contain information about New York City listings on the Airbnb platform.

[ ]:
```
import os
filepath = os.getcwd()+"/labs_data"
filename = "/airbnb_lab1.csv.gz"
data = pd.read_csv(filepath+filename)
```

[ ]:
```
data.shape
```

First, get a peek at the data:

[ ]:
```
data.head()
```

That's a lot of columns, and the layout is a little difficult to read. Let us retrieve just the list of column names, so we can read it and get a feeling for what kind of information is presented in the dataset.

1.2.6 f. Get column names: `df.columns`

[ ]:
```
list(data.columns)
```

What do the column names mean? Some of them are less intitively interpretable than others. Careful data documentation is indispensable for business analytics. Make sure to consult the documentation that accompanies this open source dataset for a detailed description of the key variable names, what they represent, and how they were generated: https://docs.google.com/spreadsheets/d/1iWCNJcSutYqpULSQHlNyGInUvHg2BoUGoNRIGa6Szc4/edit#gic

1.2.7 g. Summary statistics of the dataset: `df.describe()`

Let's print some general statistics for each one of the data columns:

[ ]:
```
data.describe(include='all')
```

Consider the following business question: What is the average availability (out of 365 days in a year) for the listings in Brooklyn? The answer can be obtained by the use of **filters** on the dataset.

1.2.8 h. Filtering the data: `df[ < condition > ]`

We need to filter the entries that are in Brooklyn. To do this, we need to know what is the exact way that Manhattan listings are spelled and entered in the data. Let's print all of th eunique values of the 'neighborhood' column:

[ ]:
```
data['neighbourhood'].unique()
```

You may have noticed that there is a lot of heterogeneity in the way 'neighborhood' values are specified. The values are not standardized. There are overlaps, redundancies, and inconsistencies (e.g., some entries specify `'Greenpoint, Brooklyn, New York, United States'`, some other ones list `'BROOKLYN, New York, United States',`, yet other ones say `'Williamsburg, Brooklyn, New York, United States'`, etc. In real life, you would have to clean this data and replace these values with standard, identically formated, consistent values.

For this data file, however, we are lucky to already have a 'cleansed' version of the neighborhood information based on the latitude and the longitude of every listing location.

We will list the unique values of the columns titled 'neighbourhood_cleansed' and 'neighbourhood_group_cleansed']:

[ ]:
```
data['neighbourhood_cleansed'].unique()
```

[ ]:
```
data['neighbourhood_group_cleansed'].unique()
```

Great, this last one is what we want! Let's filter out all data entries that pertain to Brooklyn listings:

[ ]:
```
bk = data[data['neighbourhood_group_cleansed'] == 'Brooklyn']
bk.shape
```

(Tip: to better understand what happenned above, you are incouraged to insert a new code cell below and copy *just the condition* of the filter that we used on the `data` object above -- that is, everything that we specified inside the brackets for the outermost data[...]. Run the new cell and see what that condition alone evalueates to ---- you should see a series of True/False values. When we pass that series to `data` as a Boolean filter by writing `data[ < our Boolean series > ]`, we tell Pandas to keep the values of `data` only with those indices for which the condition evaluated to `True`. Don't worry! You'll get the hang of this!)

1.2.9 i. Combining values in a column: `np.mean()`

Now that we isolated only the relevant entries, it remains to average the value of a particular column that we care about:

[ ]:
```
np.mean(bk['availability_365'])
```

1.2.10 j. Group data by (categorical) column values: `df.groupby`

The next question of interest could be: What are the top 5 most reviewed neighborhoods in New York? (By sheer number of reviews, regardless of their quality). We will use the `.groupby` method from the Pandas package:

[ ]:
```
nbhd_reviews = data.groupby('neighbourhood_cleansed')['number_of_reviews'].sum()
nbhd_reviews.head()
```

Perform a (descending order) sorting on this series:

[ ]:
```
nbhd_reviews = nbhd_reviews.sort_values(ascending = False)
nbhd_reviews.head(5)
```

Success! While we re at it, what are the least reviewed neighborhoods?

[ ]:
```
nbhd_reviews.tail(5)
```

This result makes it apparent that our dataset is somewhat messy!

Notice we could have chained the transformations above into a single command, as in:

[ ]:
```
data.groupby('neighbourhood_cleansed')['number_of_reviews'].sum().
↪sort_values(ascending = False).head(5)
```

This way we don't store objects that we won't need.

1.2.11 Bonus: easy histogram plotting with Matplotlib: `plt.hist`

As a final touch, run the cell below to instantly visualize the density of (average!) values of review numbers across all neighbourhoods:

[ ]:
```
%matplotlib inline
nbhd_reviews.hist()
```

This plot suggests that the vast majority of neighborhoods have only very few reviews, with just a handful of outliers (those ranked at the top in our previous computed cell) having the number of reviews upward of 40000.