Homework Exercise 3 + Tutorial

(Total: 30 points)

In this exercise, you will practice your Pandas skills to examine a dataset, and clean it as well as do some computations to better understand the data (exploratory analysis). In addition, we will also practice adding some new columns/features to the dataset.

As you work through this Tutorial plus exercise, please create a new Python Notebook. For most tasks, you are asked to answer one or more questions based on the code that you have written. These are highlighted in yellow.

Submission:

Please submit your python Jupyter Notebook which contains your code and answers to questions in yellow highlight. Please use Markdown for the following:

- Use a L1 heading for the notebook (Ex3)
- Add your name at the very top under Task 0.
- Label the questions that you are providing answers to using L3 heading with the word **ANSWER**. This should be done immediately preceding your answer in a separate cell.

e.g., **ANSWER 10 ii**.

> your answer as code or comment directly below

Please name your Notebook as **Hw3_LastName_FirstName.ipynb.**

It is very important that you follow the above instructions – otherwise we may not be able to find your answers when we grade. If that happens you will lose points.

Documentation/ API reference:

For Pandas Series and DataFrames, inline documentation is a bit finicky. You start with an object of the `pd.Series` class or `pd.DataFrame` class and then type the dot operator and then <TAB>. This will usually pull up the available member of the class (attributes and methods). But when you have a series of operations, it fails to always work.

For these scenarios, it is best to keep the documentation open in a Browser window.

https://pandas.pydata.org/pandas-docs/stable/reference/series.html

https://pandas.pydata.org/pandas-docs/stable/reference/frame.html

Data: Census data

For this exercise, we will use a small dataset to help understand how to use Pandas for data pre-processing, manipulation and munging. It contains population data of 50 US states and 2 territories –

District of Columbia and Puerto Rico. The initial data for the years 2010-2019 can be found in the file, **census_statespop.csv**. It also contains one additional column titled `Region` to indicate whether the state belongs to one of four census regions – `Northeast, South, Midwest, and West`. Save this file in the same directory as your Notebook at the same level to make it easy to access. We will assume this when we read from files in the code below.

This dataset has a few quirks built-in that you can see by examining the csv file – they will need to be fixed. There are missing values in rows, columns and duplicates. We will learn to detect these quirks and clean the data, create some new vars, and get some descriptive and simple analytical statistics.

Complete the following in a Jupyter Notebook.

Task 0. Housekeeping

Write the exercise number and your name at the top of the document using Markdown.

Task 0.

Completed By: Your name

Make sure you clearly label each task using Markdown. Otherwise, we may miss some of your answers.

Task 1. Import two libraries

We will use NumPy and Pandas, imported as np and pd in our program.

```
import numpy as np
import pandas as pd
```

✓ **Question 1.** Print the version of Pandas that you are using.

Task 2. Creating a DataFrame from reading in a file

For data science purposes, we will often read in data from various file formats. In this exercise, we will read from a csv file and save the data in a DataFrame. To read data from a csv we use `pd.read_csv()` function of Pandas.

> *Function vs method?*
>
> *This function is found in the Pandas module (and not within the Series or DF class), so it's a function rather than a method. The latter is found inside a particular class and operates only on the objects of that particular class).*

It has many optional parameters. If no parameter values are provided, all default values are used. Please take a look at its documentation (inline in the Notebook you can use `?` operator at the end of the function/method name).

```
pd.read_csv?
```

2a. First let us try reading it simply with no optional parameters.

```
statespopDF = pd.read_csv("census_statespop.csv")
```

To view the DataFrame, you call its name. But this would print all the contents of the file, which if too large, becomes unwieldy. A better approach is to ask to view either entries from the head or the tail of the data. `DataFrame.head()` and `DataFrame.tail()` both accept an optional parameter `n =` number of entries to show; by default n=5.

Things to note:

A new column with numbers starting from 0 is added – this is the index - and is found in the left most side of the DataFrame. The column names are kind of unwieldy, and we may prefer to use something simpler. I will show you a few different ways to rename them – one while reading in the file (shown in this Task), and another after we have the data in a DataFrame (in a later task 7).

✓ **Question 2.i.** Write a command to view the top 10 rows of the DataFrame statespopDF?

2b. Now let us read in the file again, and specify some parameters.

The documentation shows many parameters; we usually only need a few of these. Recall that optional (keyword) parameters can be specified as *keyword =value*.

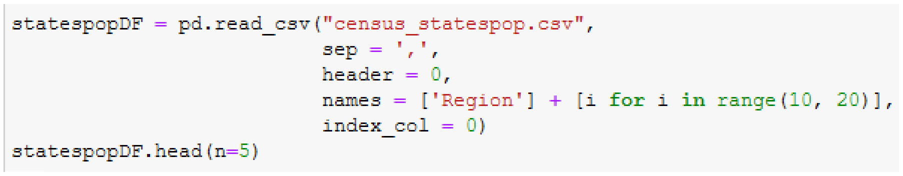
`sep` provides the separator, default = ','

`header` = 0 tells it the row number to use as column names (here row 0), default = infer

`names` = a list of column names to use instead of header. As you noticed above the colnames in the csv file are unwieldy.

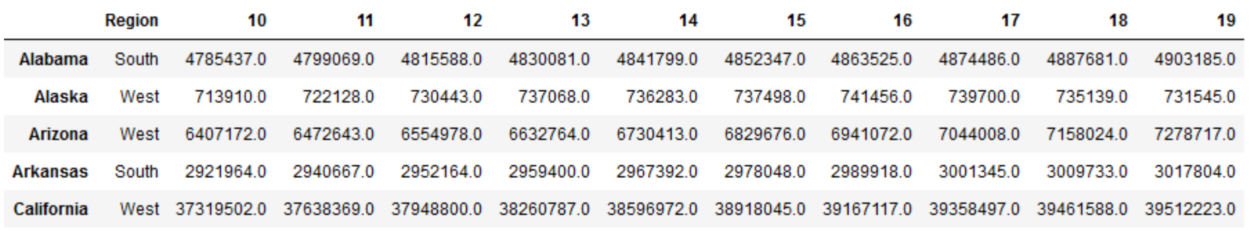
Let us replace the names as we read them in. We will retain the column name Region, but change all the years to two-digit 'intyear' such as 10, 11, … To do this, we can concatenate two lists – the first one only contains one value – 'Region' and the second is a range of ints to represent each year in the data. We can use list comprehension to create the second list of numbers 10 to 19 with the help of the `range()` function. We use standard Python elements – `list` and `range` - since the colnames is provided as a regular list.

`index_col` = column to use as an index; if not specified, an extra column is added (as in 2a.)



```
statespopDF = pd.read_csv("census_statespop.csv",
                          sep = ',',
                          header = 0,
                          names = ['Region'] + [i for i in range(10, 20)],
                          index_col = 0)
statespopDF.head(n=5)
```

View the head of your output. It should look like this:

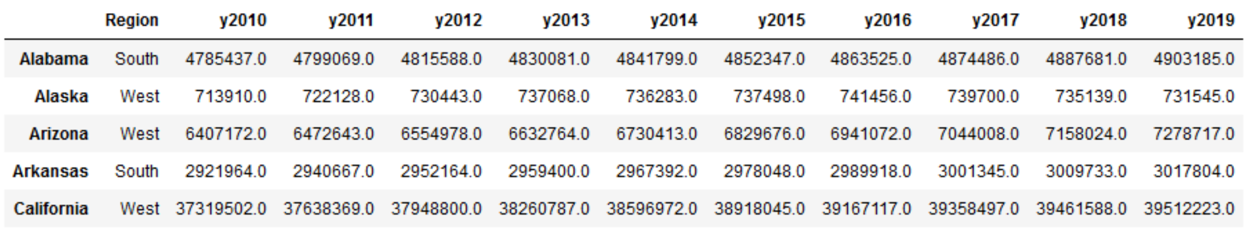


	Region	10	11	12	13	14	15	16	17	18	19
Alabama	South	4785437.0	4799069.0	4815588.0	4830081.0	4841799.0	4852347.0	4863525.0	4874486.0	4887681.0	4903185.0
Alaska	West	713910.0	722128.0	730443.0	737068.0	736283.0	737498.0	741456.0	739700.0	735139.0	731545.0
Arizona	West	6407172.0	6472643.0	6554978.0	6632764.0	6730413.0	6829676.0	6941072.0	7044008.0	7158024.0	7278717.0
Arkansas	South	2921964.0	2940667.0	2952164.0	2959400.0	2967392.0	2978048.0	2989918.0	3001345.0	3009733.0	3017804.0
California	West	37319502.0	37638369.0	37948800.0	38260787.0	38596972.0	38918045.0	39167117.0	39358497.0	39461588.0	39512223.0

✓ **Question 2.ii**. Change the colnames for the years to 'y2010', 'y2011' etc. Leave 'Region' as it is.

To get credit, you should not just name every column in the list, but rather use a loop or list comprehension to automate the generation of new column names. You can do this by following the example above using `pd.read_csv()` function. There is another way to change column names using `DF.rename()` - which we will learn later. For this question, please use `pd.read_csv()`. Your code to create new colnames should use a range() function to generate the numeric part -2010, 2011, …..2019, and add a 'y' to the front of each col name. Keep in mind to add a str and an int, you have cast the int to a str first.

Expected result of viewing top 5 rows.



	Region	y2010	y2011	y2012	y2013	y2014	y2015	y2016	y2017	y2018	y2019
Alabama	South	4785437.0	4799069.0	4815588.0	4830081.0	4841799.0	4852347.0	4863525.0	4874486.0	4887681.0	4903185.0
Alaska	West	713910.0	722128.0	730443.0	737068.0	736283.0	737498.0	741456.0	739700.0	735139.0	731545.0
Arizona	West	6407172.0	6472643.0	6554978.0	6632764.0	6730413.0	6829676.0	6941072.0	7044008.0	7158024.0	7278717.0
Arkansas	South	2921964.0	2940667.0	2952164.0	2959400.0	2967392.0	2978048.0	2989918.0	3001345.0	3009733.0	3017804.0
California	West	37319502.0	37638369.0	37948800.0	38260787.0	38596972.0	38918045.0	39167117.0	39358497.0	39461588.0	39512223.0

We will continue to use this above DataFrame for the rest of the tasks, so make sure you get it right.

Naming convention:

Going forth I will shorten `pd.DataFrame.methodname()` *to just* `DF.methodname()`.

Task 3. Understanding the data

We can use a number of DataFrame attributes to learn about the data. Try calling the following attributes in your Notebook, one in each cell.

-

- `DF.dtypes` (returns the dtype of each column)
- `DF.shape` (returns the row and col dimensions)
- `DF.size` (returns the number of elements)
- `DF.index` (returns an Index object with the row labels)
- `DF.columns` (returns an Index object with the column labels)

Missing values:

NOTE: even though the csv file that we read from has all ints for population, the values in the columns are stored as floats. This happens when there are missing values in the column.

Missing values in Pandas can have one of a few different representations – commonly for numeric data, it is a NumPy object `np.NaN` *(stands for Not a Number), which is treated as a float. So when a numeric column we expect to be ints is read in as floats - this is a good indication that there may be missing values.*

A few other representations for null include: The keyword `None` *for object data, pd.*`NaT` *for datetimelike data, and in more recent versions of Pandas, there is also a* `pd.NA`.

✓ **Question 3.i.** What are the number of rows and columns in statespopDF?

✓ **Question 3.ii.** What is the datatype (not type) of the column labels (not the columns themselves) in statespopDF?

Task 4. Indexing DataFrames: selecting entire columns or row using column name or index label

One of the common tasks we do with DataFrames is to select one or a subset of columns to work with – recall that each column is a Series. We do this by indexing.

Say we want to view only the columns or rows that contain null values. Or we want to apply data transformations to a select few columns or data (take log of population in 2019 and 2018), or we want to perform calculations over subsets of rows where the index satisfies a condition (e.g., all states that belong to West zone). All of these tasks require us to select specific rows and columns. We do this by indexing the DataFrame.

4a. Selecting only on column(s)

To select one column, we use `DF['colname']` notation. [] only works with a column name label not integer column index, and [] cannot be used with row labels. Obtain some values from the Region column. This will be returned as a Series.

```
statespopDF['Region'].head()
```

To obtain only one column and retain it as a DataFrame, use

```
statespopDF[['Region']].head()
```

We cannot use `DF[intindex]` such as `DF[0]` – this will cause an error. To select more than one column, provide a list of column labels/names, use `popDF[[collab1, collab2, collab3]]`

```
statespopDF[['Region','y2010', 'y2011']].head()
```

4b. Label indexing: Selecting specific rows and columns using labels

Let's start with label indexing, where we specify the row labels and /or column labels. The labels can be selected using fancy indexing (`[label1, label2, ….]`) or slicing (`startlabel : endlabel : stride`). Fancy indexing and slicing can be mixed across rows and columns.

In the case of slicing with labels, the endlabel is ***inclusive***.

To select all values from a dimension, we can specify the index for that dimension as : or ::

For DataFrames with two axes, you can specify indices for both axes.

```
DF.loc[rowslice, colslice]
DF.loc[[rowlabel1, … ,rowlabeln],[collabel1, … ,collabeln])
DF.loc[[rowlabel1, … ,rowlabeln], colslice]
DF.loc[rowslice, [collabel1, … ,collabeln]]
DF.loc[rowslice, :]  OR DF.loc[[rowlabel1, … ,rowlabeln], :]
DF.loc[: , colslice] OR DF.loc[:, [collabel1, … ,collabeln]]
```

To select all the columns for the row 'Georgia', use either one of the following (when : is in the col dimension, it is optional, leaving it out means the same)

```
statespopDF.loc['Georgia']
statespopDF.loc['Georgia', :]
```

To select the rows from Georgia to Minnesota, inclusive, specify a slice for row labels

```
popDF.loc['Georgia' : 'Minnesota']
```

To select all rows but only two particular columns, we can use fancy indexing for columns

```
statespopDF.loc[:,['Region','y2015']].head()
```

✓ **Question 4.i**. Obtain the population of all states only for the odd years. Print just the first 5 rows. Use label indexing with slicing.

✓ **Question 4.ii**. Obtain the population for states starting at Georgia and ending with Indiana, for the years 2012 to 2015 inclusive. Use label indexing with slicing in both dimensions.

✓ **Question 4.iii**. Obtain the population for the following 3 states in order: Georgia, Utah and Indiana, for the even years from 2012 -2016 inclusive. Use fancy indexing for rows and slicing for columns. Use label indexing in both dimensions.

4c. Integer indexing: Selecting specific rows and columns using int index

Here instead of labels, we specify the row index as an integer (0,1,…) and columns are also accessed by their position integer. We can use fancy indexing (`[label1, label2, ….]`) or slicing (`startlabel : endlabel : stride`). Fancy indexing and slicing can be mixed across rows and columns.

In the case of slicing with int index, the endlabel is ***not inclusive***.

To select all values from a dimension, we can specify the index for that dimension as : or ::

For DataFrames with two axes, you can specify indices for both axes.

```
DF.iloc[rowslice, colslice]

DF.iloc[[rindex1, … ,rindexn],[cindex1, … ,cindexn])

DF.iloc[[rindex1, … ,rindexn], colslice]

DF.iloc[rowslice, [cindex1, … ,cindexn]]

DF.iloc[rowindices, :]

DF.iloc[: , colindices]
```

To select the entire first row, we can write either of the following: when we want an entire row (all columns), we can use : for column dimension, or we can leave it out and it means the same. By default, `startindex = 0` and `endindex = lastcolindex`, `step = 1`. To request the first row, we write

```
popDF.iloc[0]
popDF.iloc[0, :]
```

To select the entire 'Region' column, we can write the following. 'Region' is the first column at index 0. I only print the first 5 rows.

```
statespopDF.iloc[:, 0].head()
```

To select only the even years from 2010 to 2019 inclusive, and all rows, but only print first 5 rows, we can write:

```
statespopDF.iloc[:,1::2].head()
```

To select rows at index 0 and 2, and columns at index 3 and 5, we can write:

```
statespopDF.iloc[[0,2], [3, 5]]
```

✓ **Question 4.iv**. Obtain only the Region for every other state starting with the first (in the default order it is in the DataFrame), and print all records. Use integer indexing with slicing for rows.

Bottom line with indexing:

DF.loc[] is the most versatile and important of the group – we can also use that with Boolean indexing, which we will learn later… So learn that well.

Task 5. Cleaning and pre-processing the data

Before doing analyses, we should check to see what the data contains - counts, frequencies, data types. We should check if the data is good - whether there are missing values/ nulls and duplicates etc. There are many DataFrame methods / functions that can help us. Most of these use a default axis = 0, which means perform the operations within columns (which is one variable in a table).

Indexing a DataFrame

Series is like a vector, one dimensional, whereas DataFrame is 2D, the axis that runs vertically down is referred to as axis=0 and the axis that runs horizontally across is referred to as axis=1.

So many DataFrame methods accept a parameter called ***axis*** to tell it over which axis to perform the operation on. *In many cases,* ***the default is axis = 0****, because we usually care about performing operations to a column of data (within the column).*

5a. Determine which columns (axis = 0) have null values.

The `DF.isnull()` method will produce a DataFrame of Boolean values to check whether each DF cell is null. There is also `DF.isna()` – which is equivalent, an alias.

Getting a DF of True/False values is not helpful for large data. But we can obtain a summary for each column (or row) by calling on the output of `DF.isnull()` – which itself is a DF.

- `DF.any()` - returns True if any value in the DF along an axis is True.
- `DF.all()` - returns True if all values in the DF along an axis is True.

Both of these methods have an optional parameter `axis`. `axis = 0` is the default, and will compute whether there are nulls within columns, and `axis = 1` will do so within rows.

What we want is to determine whether there are nulls – `DF.isnull()` will return booleans for each cell, then we want to summarize for each column, whether *all* values are null, or *any* value is null. To do this, we will want to chain together the above commands as follows.

```
statespopDF.isnull().any()
```

```
statespopDF.isnull().any(axis = 0)
```

The above command should you tell you that all 11 columns have one or more null values.

Chaining of methods:

Chaining involves attaching a second method to the end of the first method call, and this can be repeated many times.

DF.method1().method2().method3()….

The only caveat is that each method should be a method of the class/object on the left of the dot operator preceding its name. This means method2 should be a method of the type of object that DF.method1() returns, and method3 should be a method of the type of object that DF.method2() returns.

DF methods usually return DataFrames, but if you index into only one column, then sometimes, you may get back a Series object instead of DataFrame object. Series and DataFrames may have some methods in common, and others that are different. So when learning new classes and methods, always look up documentation to see what type of object is returned. Or use the type() function of standard Python.

✓ **Question 5.i**. Write code to output whether each column has 'all' null values, and provide an answer: how many columns have all null values? Your command should include the DF.all() method.

5b. Count of null and non-null values in each column

We can also determine the count of nulls in our data using another method. The `DF.count()` method obtains counts of non-missing or non-null values. An optional parameter `axis` is allowed. By default `axis = 0`

```
statespopDF.count()
```

Recall that the DF.shape attribute showed us that there are 54 rows and 11 columns.

✓ **Question 5.ii**. Print the number of non-missing values for each row. Write the command using DF.count() method and provide the answer.

5c. Drop rows (or columns) that are null throughout

In our data, there is one row of all null values. You can verify this by writing the following command to view the contents of such a row.

```
statespopDF.loc[statespopDF.isnull().all(axis=1)]
```

To do the same for column, you have to write the below command (there are no empty columns however).

```
statespopDF.loc[:,statespopDF.isnull().all(axis=0)]
```

Boolean indexing with loc[]

`loc[]` is primarily label based, but may also be used with a boolean array. In DataFrames, we can create an index of boolean values (has to be of the same length as the DataFrame) and use that to select elements along an axis. We can then pass this boolean index to loc[].

To create a boolean index, we must use a condition that is relevant to our needs. Conditions return either a True or False. We use `statespopDF.isnull().all(axis=1)`, which will return a Series of True/False values the same length as the number of rows, since axis = 1. We then give this boolean index to `loc[]` as the index for rows, which will then only print the value of a row where the boolean index is True. The boolean index will be True only if all values in that row are null.

To drop this row, we will call the `DF.dropna()` method, which can drop rows or columns from the DF. It accepts many parameters, three of which are interesting and useful for our purpose now. The parameter `axis` specifies the dimension to look for nulls: `axis = 0` (default) means drop rows (or index that runs vertically down) that contain missing values and `axis = 1` means drop columns that contain missing values.

The parameter `how` specifies how to drop, we can set `how = 'all'` so it will only drop when all values in an axis are null. We can set `how = 'any'` to drop if any value is null.

The parameter `inplace` is by default `False`, which will create a copy of the DF after dropping data. By changing it to True will replace the values in the original DF. We will save the resulting data in a new DF.

```
statespopDF_dropna = statespopDF.dropna(axis = 0, how = 'all')
```

Check the shape of the new DF to ensure that a row has been dropped.

5d. Frequency counts of unique values in rows

If we want to obtain the unique values in a Series (column), we can use the `Series.unique()` method. It includes missing values, you can exclude that by setting an optional parameter `dropna = True`

```
statespopDF_dropna['Region'].unique()
```

When we want to count the frequency of each type of value found in a Series or column in a DataFrame, we can use `DF.value_counts()`. This method returns a Series containing frequency counts of unique rows in the DataFrame. We can specify as a list the rows over which we want to count unique values; the default is all rows. The result is by default sorted in descending order of the frequency count.

Perform the count for the Region column,

```
statespopDF_dropna['Region'].value_counts()
```

✓ **Question 5.iii**. Write code to obtain a proportion of each of the unique values in the Region column (there are four) instead of counts, (after dropping the missing value). Hint: change a parameter in the value_counts() method

5e. Determine and remove duplicate rows

A lot of time, datasets may contain duplicate rows. Removing duplicate rows is another important data pre-processing step.

A method that tells us which row(s) are duplicated is `DF.duplicated()`. It produces a Series with Boolean True or False value for each row index. We can optionally use parameter `subset = [colnames]` to give `DF.duplicated()` a list of colnames to use to for defining duplicates. The default is all columns.

We can chain the result with a `Series.value_counts()` – a Series method to obtain a frequency count of True or False values. The below chained command will tell us how many rows are duplicated and how many rows are unique.

```
statespopDF_dropna.duplicated().value_counts()
```

Can you verify that there is one duplicated row? There are 53 total rows in total and only 52 are unique.

We can then drop those rows using the `DF.drop_duplicates()` method. This method has two useful parameters, `keep = 'first'` will keep the first of the duplicate values, and drop the rest. The optional parameter called `inplace = False`, by default. If you wish to change the underlying DF itself without creating a copy, set `inplace = True`. Let us create a new DF.

```
statespopDF_dropnadup = statespopDF_dropna.drop_duplicates(
                                                    keep='first', inplace=False)
```

You can verify that there are no more duplicates by calling DF`.duplicated()` and `Series.value_counts()`, chained together as before.

✓ **Question 5.iv**. What is the shape of statespopDF_dropnadup after dropping rows with all nulls and rows with duplicate values?

Hint: at this point, you should have shape (52,11).

Task 6. Create a deep copy and save to a file

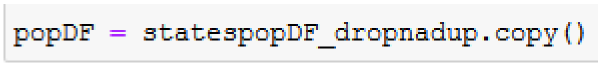
We have just finished processing the dataset - we have a DataFrame with no all-empty rows and no duplicate rows. We still have some missing values in the column Regions for territories as they dont belong to the 4 regions. Let us save it for future use. So first we will make a deep copy of it and then write out to a file.

6a. Create a deep copy of the DF

There are two ways to make a copy of a DF, a ***shallow copy*** and a ***deep copy.*** `DF.copy()` can be used for both. It has one optional parameter, `deep = True`, by default. When `deep=True`, a new object will be created with a copy of the original object's data and indices. Modifications to the data or indices of the copy will not be reflected in the original object. (i.e. the deep copy is truly a separate object in a separate memory location).

When `deep=False`, a new object will be created without copying the original object's data or index (only references to the data and index are copied). Any changes to the data of the original DF will be reflected in the shallow copy (and vice versa). To make a shallow copy, you can also use DF1shallowcopy = DF1.

Create a deep copy of `statespopDF_dropnadup` and call it `popDF` (to shorten the name).



```
popDF = statespopDF_dropnadup.copy()
```

✓ **Question 6.i.** If we set `DF_b = DF_a`, any changes made to `DF_b` will also be propagated to `DF_a`. True or False?

6b. Write DF to a file

We can use the `DF.to_csv('filename')` method. This has optional parameters, `header = True`, which means we want to write out the headers from row 0 and `index = True`, which will write out the index labels or row names.

```
popDF.to_csv('popDF.csv', header = True, index = True )
```

✓ **Question 6**.ii. When we save or write a DataFrame out to file, what parameter and value is used to indicate that column labels are included in the first row? (not the whole command, just the optional parameter and its value)

6c. Read the data back from the file and change dtype of some columns

Try reading the file back in and ensure that the data looks as expected.

Now that our yearly population columns do not contain missing values, we can set their `dtype` to `int` using the `DF.astype()` method. They are read in as float because they were float when we write out to csv. This method `DF.astype()` accepts a dict of colname and dtype that you want to set it to.

```
popDF = popDF.astype(
    {'y2019': int,
     'y2018': int,
     'y2017': int,
     'y2016': int,
     'y2015': int,
     'y2014': int,
     'y2013': int,
     'y2012': int,
     'y2011': int,
     'y2010': int
    })
popDF.head()
```

Afterwards, you can use the `DF.dtypes` attribute to verify the change.

Task 7. Indexing based on conditions

In Task 4, we learned about two types of indexing, where we selected values based on indices or labels. Another useful approach is to select rows or columns that satisfy some condition. We do this by Boolean indexing. There are a few different ways to do this including the use of two methods `Series.isin()` and `DF.filter()`

7a. Boolean Indexing with comparison operators

We saw a little `Boolean indexing` in Task 5. Let us explore that more here.

What is boolean indexing?

We create a boolean mask, which is a Series of booleans that we can use for indexing the DataFrame. We create a mask using a condition that produces True or False values. We can then use `DF.loc(mask)` *to obtain only the rows where mask is True*

Let us select all rows where the 2019 population is greater than 10 million. The first line below will output all columns, since we didn't specify any columns – so all are selected. To only output some columns, use `.loc[]` indexing to specify a list or slice of columns to select.

```
popDF.loc[popDF['y2019']> 10000000]
```

✓ **Question 7.i**. Use Boolean indexing to select states where the Region is equal to 'South' and only select the columns – Region and years 2018 and 2019.

✓ **Question 7.ii**. Use Boolean indexing to select states where the 2019 population lies between 10M and 20M, and show only the columns Region and y2019.

HINT: to use more than one condition, you must surround each in parentheses.

Also use the Boolean operator version `&` and `|` (instead of `and` and `or`). No select where the condition is not true, use ~ with .isin(), e.g., DF.loc[~(DF['column].isin(['va1', …. 'valn']))]

7b. Using Series.isin() method

We can use `Series.isin([list of values])` to select rows where the content is found in the values provided as a list. (think of `.isin` as the Series equivalent of `in` operator)

Select rows where 'Region' matches 'Midwest' and only display the first 5 rows of some selected columns.

```
popDF.loc[popDF['Region'].isin(['Midwest']), ['Region', 'y2010', 'y2019']].head()
```

We can also select values from the index that match 'Connecticut' by first obtaining the index Series using DF.index.

```
popDF.loc[popDF.index.isin(["Connecticut"])]
```

✓ **Question 7.iii**. Use Series.isin() to select values for states in the Northeast region that also have population greater than 10M in the year 2019. Show the values only for the column Region and y2019.

7c. Using DF.filter() method

`DF.filter()` works on ***labels only*** – both row and columns labels. We specify which axis using `axis = 0` (rows) or `axis = 1` (columns). It tries to find the specified substring in the row or column labels. This method returns a DataFrame. It can accept three types of parameters that it uses to perform searches:

- parameter `like` accepts substrings
- parameter `items` accepts exact strings to find
- parameter `regex` accepts a str pattern to match

To select rows that contain the substring 'New'.

```
popDF.filter(like = "New" , axis=0)
```

To select rows where the index is exactly equal to a specified list

```
popDF.filter(items = ["Utah", 'Colorado', 'Arizona'] , axis=0)
```

You can also use `Series.str` to perform substring and regular expression matches on string data in any Series/column, not just the labels.

For example, to select the rows that contain the word 'New' in the row index, we can write

```
popDF.loc[popDF.index.str.startswith("New")]
```

✓ **Question 7.iv**. Write code to select all rows from popDF where the row label starts with the letters G or H or I only.

Task 8. Manipulating row and column labels

In Task 2, we renamed the columns at read time. Sometimes, after we read the file, we may wish to change some col and row labels. The steps in this Task will create new DF in order to not change popDF.

8a. Rename (some) columns

We can use the `DF.rename()` method to rename selected columns or rows. To rename columns, we provide a parameter `columns = {}`, where {} is a mapper or dict such as `{oldvalue1 : newvalue1, oldvalue2 : newvalue2, …}`. It is common to use dict comprehension to create the new values if there are too many values to write down.

Recall that pd.Index is immutable- so a new index object is created in both cases when you modify some values of column or row labels. Renaming does not change the labels on the original DF, by default. Can change that by setting optional parameter inplace = True.

Here is an example of changing the year column names to remove the y, and keep just the number, e.g., '2010', '2011', etc. We use ***dict comprehension*** (very much like list comprehension but used to create a dict) to rename only some columns by creating a dict of {oldvalue : newvalue}

The key of the dict is just the oldvalue (here x = ‘y2010’,…’y2019’), and we need to give a formula to create the newvalue (here we obtain just the numeric part of the string from index -4 onwards). And we only want to do this for column 1, 2, ,… and not column 0 (which is Region, and we do not change it).

```
popDF.rename(columns =
                {x : x[-4:] for x in statespopDF.columns[1:]}).head()
```

Here are the first few rows

	Region	2010	2011	2012	2013	2014	2015	2016	2017	2018	2019
Alabama	South	4785437.0	4799069.0	4815588.0	4830081.0	4841799.0	4852347.0	4863525.0	4874486.0	4887681.0	4903185.0
Alaska	West	713910.0	722128.0	730443.0	737068.0	736283.0	737498.0	741456.0	739700.0	735139.0	731545.0
Arizona	West	6407172.0	6472643.0	6554978.0	6632764.0	6730413.0	6829676.0	6941072.0	7044008.0	7158024.0	7278717.0
Arkansas	South	2921964.0	2940667.0	2952164.0	2959400.0	2967392.0	2978048.0	2989918.0	3001345.0	3009733.0	3017804.0
California	West	37319502.0	37638369.0	37948800.0	38260787.0	38596972.0	38918045.0	39167117.0	39358497.0	39461588.0	39512223.0

We don’t want to save this change, so I did not use `inplace = True` as a parameter. Instead, we will continue to use column names ‘y2010’,’y2011’… as before.

✓ **Question 8.i**. Write code to rename the columns of popDF as ‘y10’, ‘y11’, ‘y12’…’y19’ using the DF.rename() method and a new dict comprehension.

To get full credit you must generate the column names using a formula and dict comprehension. Just writing out the column labels using a dict will not get credit.

8b. Create a new surrogate index (row labels) with value 0,1,2,…..

pd.Index is immutable, so we cannot change or modify selected row indices as follows.

`popDF.index[0] = ‘Newstate’` # not allowed

But we can change the entire index of a DataFrame by re-assigning a new index object

We can use `DF.reset_index()` and save in a new DF.

```
popDF_newindex = popDF.reset_index()
```

Here is the output of viewing the top 5 rows. Note that the original index becomes a column named ‘Index’ and there is a new row index object (0,1,….)

	index	Region	y2010	y2011	y2012	y2013	y2014	y2015	y2016	y2017	y2018	y2019
0	Alabama	South	4785437.0	4799069.0	4815588.0	4830081.0	4841799.0	4852347.0	4863525.0	4874486.0	4887681.0	4903185.0
1	Alaska	West	713910.0	722128.0	730443.0	737068.0	736283.0	737498.0	741456.0	739700.0	735139.0	731545.0
2	Arizona	West	6407172.0	6472643.0	6554978.0	6632764.0	6730413.0	6829676.0	6941072.0	7044008.0	7158024.0	7278717.0
3	Arkansas	South	2921964.0	2940667.0	2952164.0	2959400.0	2967392.0	2978048.0	2989918.0	3001345.0	3009733.0	3017804.0
4	California	West	37319502.0	37638369.0	37948800.0	38260787.0	38596972.0	38918045.0	39167117.0	39358497.0	39461588.0	39512223.0

If we don’t really need this surrogate row index, we can set the index back to the ‘index’ column using the `DF.set_index()` method

```
popDF_newindex = popDF_newindex.set_index('index')
```

After this, the original state names will become the row labels again.

8c. Drop columns

We can drop columns that we don't need using `DF.drop()`. You can give it a list of colnames to drop. It has a few parameters, `axis = 1` specifies that the labels are column labels, `inplace = True` will change the original DF, and finally `errors = 'ignore'` won't produce errors if the column labels are not found.

```
popDF_newindex.drop(['y2010'], axis = 1, inplace = True, errors = 'ignore')
```

✓ **Question 8.ii**. What does the parameter errors = 'raise' of method DF.drop() do? Look up the documentation and explain in your own words

Task 9. Columnar operations on the DF data

Let us return to work with **popDF**.

When we perform operation on a DF, we do not have to save the results, if we don't want to access it again in the future. As with regular objects, we can perform operations (by calling methods or extracting subsets of data using indexing) and view the results immediately. We will usually just do this to avoid creating unnecessary new objects.

```
myDF.operation
```

If we want to save the results, we can write

`newDF = myDF.operation` *if the method returns a new DF*

`myDF[newcol] = myDF.operation` *if we want to add the returned Series to the DF*

9a. Create a new column copied from Region column and change some of its values

popDF has two missing values in the 'Region' column. Instead of changing the original column, create a new column called 'RegionAll' and change the null values there

Let us create a new column labeled RegionAll using `DF[newcol]` and set it equal to an existing column.

```
popDF['RegionAll'] = popDF['Region']
```

Use *Boolean indexing* to select only the values where the RegionAll column has missing values. We can use `Series.isna()` or `Series.isnull()` on this new RegionAll column to create a mask for popDF, such that it will only return to us rows where the mask is True (RegionAll is null). Save this in a Series called ***bmask***.

```
bmask = popDF['RegionAll'].isna()
popDF.loc[bmask, ['Region', 'RegionAll']]
```

This should identify two entries for index = 'District of Columbia' and 'Puerto Rico'. Both columns Region and RegionAll should have NA or missing values.

✓ **Question 9.i**. What is the value of the Series bmask for the row with label 'District of Columbia'?
Note: this is not asking for what the value of the popDF DataFrame is for 'District of Columbia'

9b. Change specific values in a column

Now, we can set or replace the null value at these locations in column RegionAll to the following two values. Notice, we use `DF.loc[]` to choose one cell each time.

```
popDF.loc['District of Columbia','RegionAll'] = "DC"
popDF.loc['Puerto Rico','RegionAll'] = "PR"
```

After changing it, you can view the changes. RegionAll values are no longer NA.

```
popDF.loc[bmask, ['Region', 'RegionAll']]
```

9c. Sort the DataFrame

We can sort the DataFrame on (data in one or more) columns or along its (row) index - using `DF.sort_values()` or `DF.sort_index()`, respectively.

`DF.sort_values()` accepts a list of column labels to sort in order. The default sort order is ascending, but you can change that using the optional parameter `ascending = []`, and give it a list of Booleans to match the number of column labels used. By default `ascending = True.`

Sort in ascending order of column Region and then in ascending order of column y2010

```
statespopDF.sort_values(['Region', 'y2010']).head(5)
```

Sort in descending order of column Region and then ascending order of column y2010

```
statespopDF.sort_values(['Region', 'y2010'], ascending=[False, True]).head()
```

Missing values in the sort columns are added to the end.

✓ **Question 9.ii**. Write code to select from popDF only the Region and y2019 columns, and sort the DF first on Region (ascending) and then on y2019 (descending).

Show the top 5 rows of the resulting DataFrame.

Task 10. Computations on DF data: Math and stat operations

10a. Obtain summary information

To obtain some summary information on each column, we can call `DF.info()` to obtain.

```
popDF.info()
```

Verify that : There are 10 int32 columns and 2 object columns – Region and RegionAll.

To obtain some pre-defined statistics like count, mean, std, min, max and several quantiles of the data within each column, call `DF.describe()` – which will ignore non-numeric columns by default.

```
popDF.describe()
```

10b. Performing math /scientific operations on the data

Many mathematical operations can be found in the NumPy library as it is the math and science library. These NumPy universal functions (*ufuncs*) operate on ndarrays, `Series` and `DataFrames`. Because of *vectorization* (from NumPy slides) of NumPy ufuncs, we only need to specify the operation like a scalar and it will be applied to all rows.

Unary operations (on one value)

Below, we call a NumPy method to calculate log of a Pandas column (a Series) and we call `DF.sample()` to view a random sample of 5 values.

```
np.log(popDF['y2019']).sample(5)
```

Binary operations (on two values)

To add a scalar to a column, we can use a NumPy binary ufunc and we use `DF.tail()` to view the last 5 values.

```
(np.add(popDF['y2018'], 1000)).tail()
```

Pandas also has some basic mathematical methods. These are found in the Series or DataFrame classes, and are called using a`Series.method()` or `DF.method()`.

Below is Pandas version of the same addition.

```
(popDF['y2018'].add(1000)).tail()
```

#Note: the `DF.add()` method used above is called a wrapper, as it wraps around the underlying arithmetic operator +. But in addition to performing the math operation, `add()` method gives us a few additional features, such as filling in missing values if desired (cant do that when we use +). See parameter `fill_value` in `DF.add()`

```
(popDF['y2018'] + 1000).tail()
```

Similar Series/DataFrame wrapper methods exist for -, *, /, //, %, ** known as `sub, mul, div, mod, pow`

To calculate the percentage change in population from year 2018 to 2019 for every state. We can use another either NumPy or Pandas methods. I will use Pandas. I call `DF.nsmallest(n)` to show the n smallest values in the new Series.

```
((popDF['y2019'] - popDF['y2018']) / popDF['y2018']).nsmallest(10)
```

✓ **Question 10.i**. Write code to compute, for each state, the ratio of its population in 2019 divided by the maximum population of all states in 2019. Then display the state names and ratios for the 5 largest values using the `DF.nlargest()` method.

10c. Performing stats (aggregation) operations on the data

You can obtain statistics such as mean, variance, standard dev, etc for the data in the DataFrame using methods in `DataFrame`. Examples include `DF.mean()`, `DF.var()`, `DF.kurt()` etc. You can specify the axis =0 to calculate within columns, and axis = 0 to calculate within rows. By default, it will perform these calculations on all numeric columns only.

To obtain the mean within numeric columns

```
popDF.mean(axis=0)
```

Or better select the numeric columns first (to avoid a warning).

```
popDF.loc[:, 'y2010' : 'y2019'].mean(axis=0)
```

To find the index (row label, here state) with the lowest value within each specified column, we use `DF.idxmin()`. This will give us the index of the row with the lowest value in each of the specified columns in the DF.

```
popDF.loc[:,['y2017', 'y2018', 'y2019']].idxmin()
```

To find the index (row label, here state) with the highest value within each specified column, we use `DF.idxmax()`. This will give us the row index for with the highest value in each specified column in the DF.

```
popDF[['y2015', 'y2019']].idxmax()
```

Note when indexing only columns, we can use either `DF.loc[:, [columns]]` or `DF[[columns]]` as we learned above in an earlier task.

```
popDF.corr()
```

✓ **Question 10.ii.** Write code to compute the row means of the population across years 2015-2019 inclusive, and print the means for the states with the 5 largest values.

HINT: To compute the mean on all the year population columns, we first select just the columns we want from popDF and then call a Pandas DataFrame method, `DF.mean().` This method allows us to specify what axis we want the mean over (as usual `axis = 0`, is within columns and `axis = 1` means within rows).

After, use the `Series.nlargest()` method to do obtain the n largest values from this new column (which is a Series).

Task 11. Create new computed columns

11a. Create a new column with the % change in population between 2018 and 2019

Adding a new column is as easy as the following: `DF['newcolname'] = calculation`

```
popDF['2019_pctg'] = (popDF['y2019'] - popDF['y2018'])/popDF['y2018']
```

✓ **Question 11.i** Write code to compute a new column in popDF that contains the compound annual growth rate of population between 2014 and 2019. Call this new column '2019_5cagr'.

$$Pop_{Future} = Pop_{Present}\, x(1+i)^n$$

Where:

Pop_{Future} = Future Population
$Pop_{Present}$ = Present Population
i = Growth Rate (unknown)
n = Number of Years

Which can be solved for i (the compund annual growth rate.

i = ((ending value /starting value) ^ (1/num years)) -1

num years = 2019-2014 = 5

Round the column values to 4 decimals, using `np.round().`

Recall that you can pass an entire Series to a numpy function.

Finally, print a random sample of 5 rows from popDF after you complete this task.

11b. More complex operations on data for which built-in functions may not exist – using DF.apply()

`DF.apply()` applies a function along an axis of the DataFrame, ***row-wise*** or ***column-wise***. It has access to all the values within an axis (column or row) each time, and applies the same operation to every element along that axis.

This method requires a function as the first parameter, and optionally we can specify the `axis = 0` (to compute within columns) or `axis = 1` (to compute within rows).

When do we use DF.apply()?

When we want to perform complex operation for which no readymade built-in optimized universal functions exist in either NumPy or Pandas, then we can write our ow custom functions and apply it to the DF using DF.apply()

Say we want to compute the exponential of the standardized value (subtract mean and divide by std.dev) of each value in the column y2019. Assume no built-in function exists. We can use `DF.apply()` and specify a function to perform the computation. `DF.apply()` is commonly combined with a lambda function. `x.mean()`is a Series method that will produce the mean of the Series x and `x.std()` is a Series method that calculates the std dev of the values in Series x.

Start by first selecting the desired column from the dataframe –select column 'y2019'. Note the [[ ]] double brackets in order to get the result as a DataFrame. If you only wrote popDF['y2019']- it would return just a Series.

```
popDF[['y2019']].apply(lambda x : np.exp((x - x.mean())/ x.std())).head()
```

Instead of a lambda function, sometimes we may want to write a named custom function, so it can be available for future use in the program. Here is how you can change the above code to use a custom function called *expstandardize*. The results are the same.

```
def expstandardize(x):
    return np.exp((x - x.mean() )/ x.std())

popDF[['y2019']].apply(expstandardize).head()
```

If we wanted to perform the same calculation on more than one column, we just select more columns from the DF to begin with. Each column of the DF is a Series and the same operation will be applied to each.

```
popDF[['y2018','y2019']].apply(expstandardize).head()
```

Notice that the above steps did not actually add new columns to our data, but performed some computations and gave us results. In the next task, you are asked to perform calculations using `DF.apply()` and add a new column to popDF.

✓ **Question 11.ii**. Write code to compute a new column in popDF that contains the coefficient of variation for each state for the population in years 2010-2019. Use `DF.apply()` for this task.

Coefficient of variation = std / mean, and gives us the variability around the mean.

Before calling apply() on popDF, first select only the columns that contain the data over which you want to perform calculations, i. e. the population from 2010-2019. If you don't do this, you will get an error since there are some non-numeric columns in popDF and you can't use them in this calculation.

There are a few different ways to do this. I'll give you one. DF.columns will give you a Index of all columns, which you can then use indexing on to select only the years columns. You can then use that as an index on popDF to get a new DF with only the 10 year-columns. Now this is the DF on which to call apply().

In apply(), think about whether this calculation is row-based or column-based and set the axis appropriately.

Finally, because we want to add a new column with the answers, don't forget to create a new column in popDF and set its value equal to the results of `DF.apply()`.

```
___________ = popDF[__________].apply(___________________, axis = ___).head()
```

11c. Aggregation operations using DF.agg()

When we want to obtain many statistics (and different statistics) for diff columns in one go, we can call the `DF.agg()` method. This method aggregates (reduces) the data using one or more operations over the specified axis. The first parameter of this method – **func** - is the function(s) to use for aggregating the data. **func** has to be a *reducing* function, i.e. many – to 1 operation. It can be expressed as :

- A single function
- Multiple functions specified as a list
- A dict of axis labels as keys and the function to use for each axis label as values

If using a built-in Pandas aggregation function (or method) – which is common - we call it as a string, e.g., `'mean'`. If we use NumPy function, we can write it as `np.ufunc such as np.mean`. Note when calling numpy ufuncs inside `.agg()`, we omit the (). We can also call custom functions by using just their name.

It is best to learn this through an example. Say, we want to calculate a bunch of different statistics for a few numeric columns – y2019 and y2018. First select the columns that you want from popDF, and then call .agg() and give it a list of funcs (here we will use built-in Pandas functions).

```
popDF[['y2019', 'y2018']].agg(['count','sum', 'min', 'max',
                                'mean', 'var', 'skew','kurt', 'median', 'idxmax', 'idxmin'])
```

If we want to perform different aggregation operation for different columns in a DF, we can provide a dict with axis labels as keys and function(s) to compute for that axis.

```
popDF[['y2019', 'y2018']].agg({'y2019': ['min','max'], 'y2018' : ['var', 'std']})
```

Or , you can add a few NumPy functions to the mix as well, as below. Notice that even the answers are numerically the same, np.min and 'min' call different functions – the first is a Numpy function, and the second is a Pandas function.

```
popDF[['y2019', 'y2018']].agg({'y2019': ['min','max'],
                               'y2018' : ['var', 'std', np.min, np.max]})
```

<u>11d. Performing grouping operations using DF.groupby()</u>

Notice that so far, all of our columnar computations were performed over all the rows in the DF. Sometimes, we may wish to perform these analyses and computation within sub-groups of data (i.e. subset of rows). To specify the groups, we use `DF.groupby()`, which will return a GroupByDataFrame object, which has its own methods (different from DataFrame object).

See GroupByDataFrame's documentation here: https://pandas.pydata.org/pandas-docs/stable/reference/groupby.html.

Then, within groups, we can specify the operations we want to perform. This is frequently referred to as a ***split-apply-combine*** operation

- first split the data into groups within our columns based on some criteria,
- then apply a function to each group independently
    - which can be built-in aggregation functions (such as mean(), max() etc) using `DF.groupby().aggfunc()`
    - or custom functions using `DF.groupby().apply(lambda func)`
- then combine all the answers together to obtain the final DF (this is done automatically)

Say we want to compute the mean population for two years (2010 and 2019) within each Region. This is a grouping operation. We can first select the columns of the DF we want to work with, then call DF.groupby to split the groups. Next, we apply a function – such as mean() here. .groupby() will combine the results of the groups – and we can additionally specify if we want the results to be sorted in any way when combined.

```
popDF[['Region','y2010','y2019']].groupby(['Region']).mean().sort_values(['Region'])
```

Another example: Say, we want to compute several aggregation statistics for the column y2010 and y2019 – such as min, id of min value, max, id of max value, and mean for each of the years. Once again, we can groupby and then select the columns (it is important again to use [[ ]] to obtain a DF on which to call groupby()), and then apply a function – here agg(). The results of subgroup calculations will be automatically combined at the end.

```
popDF.groupby(['Region'])[['y2019', 'y2010']].agg(['min', 'idxmin', 'max', 'idxmax', 'mean'])
```

Observe that the result now contains a multi-index in the columns – this will happen when you use groupby() and compute multiple statistics.

✓ **Question 11.iii**. Write code to compute a new column in popDF that contains the mean of the population in year 2019 grouped by region (Midwest, Northeast, South, West). Call it 'mean2019popbyReg'

> `DF.groupby()` will produce a GroupByDataFrame, which will only contain 4 rows, one for each region. But what we want is to add a new column of group means with as many rows as there are observations in popDF.
>
> To do this, instead of calling an aggregation function (aggfunc) right after `.groupby()`, instead call a transforming function. Transform functions perform 1:1 calculations (instead of M:1 as aggregate functions did).
>
> The command is: `GroupByDataFrame.transform(aggfunc).`Inside transform() the aggfunc is specified as a string or np.ufunc (as we did for .agg()).
>
> Since we want to create a new column, don't forget to save the results back in a newly created column with the name given above.

Print the first 5 rows of popDF.

Hopefully, this exercise provided you with a flavor of what Pandas can do – for data manipulation, munging, cleaning, and processing. Of course, you can also create plots to see trends in the data.