



Practical 05: Data Exploration

Dr. Todd Jones

t.r.jones@reading.ac.uk (<mailto:t.r.jones@reading.ac.uk>)

Department of Computer Science

Follow the instructions to complete each of these tasks.

This set of exercises focuses on writing basic Python code and exploring the use of `pandas` and `numpy` to manipulate and pre-process data in advance of data visualisation and applying machine learning models.

This work is not assessed but will help you gain practical experience for the coursework.

Configuration

```
In [2]: import numpy as np
import pandas as pd

NaN = np.nan

# Configure sample DataFrame
dogs = pd.DataFrame({
    'Name': ['Tom', 'Lupita', 'Lupita', 'Olivia', NaN, 'Marcel', 'Choco'],
    'Age': [5, 4, 4, 1, 10, NaN, NaN],
    'Breed': [NaN, 'Corgi', 'Corgi', 'Husky', 'Poodle', 'Bulldog', NaN],
    'Grade': ['a', 'b', 'b', 'c', 'd', 'e', 'f']})

dogs
```

```
Out[2]:
```

	Name	Age	Breed	Grade
0	Tom	5.0	NaN	a
1	Lupita	4.0	Corgi	b
2	Lupita	4.0	Corgi	b
3	Olivia	1.0	Husky	c
4	NaN	10.0	Poodle	d
5	Marcel	NaN	Bulldog	e
6	Choco	NaN	NaN	f

Detecting Duplicated Rows

Look up the `dogs.duplicated()` method. Use this to **display** an indication of any duplicated rows.

<https://pandas.pydata.org/pandas-docs/version/2.2.2/reference/api/pandas.DataFrame.duplicated.html>
(<https://pandas.pydata.org/pandas-docs/version/2.2.2/reference/api/pandas.DataFrame.duplicated.html>)

```
In [2]: # look up the duplicate() method, which is used to identify duplicate rows in a DataFrame
dogs.duplicated()
```

```
Out[2]: 0    False
        1    False
        2     True
        3    False
        4    False
        5    False
        6    False
        dtype: bool
```

Use the `.sum()` method in conjunction with the previous result to display a count of the number of duplicated rows.

```
In [3]: dogs.duplicated().sum()
```

```
Out[3]: 1
```

Dropping Duplicated Rows

Look up the `drop_duplicates()` method. Use this display a corrected version of the DataFrame

Does this method alter the original DataFrame ?

- If so, continue on.
- If not, save the result before continuing.

```
In [4]: # look up the drop_duplicates() method, which is used to remove duplicate rows from a DataFrame
# Does not modify the original DataFrame
dogs.drop_duplicates()
if dogs.duplicated().sum() > 0:
    dogs = dogs.drop_duplicates()
dogs
```

Out[4]:

	Name	Age	Breed	Grade
0	Tom	5.0	NaN	a
1	Lupita	4.0	Corgi	b
3	Olivia	1.0	Husky	c
4	NaN	10.0	Poodle	d
5	Marcel	NaN	Bulldog	e
6	Choco	NaN	NaN	f

Dropping Irrelevant Columns

It has come to our attention that the data under `Grade` is meaningless.

Look up the `drop()` method. Use this to remove the irrelevant column from the data.

<https://pandas.pydata.org/pandas-docs/version/2.2.2/reference/api/pandas.DataFrame.drop.html>
(<https://pandas.pydata.org/pandas-docs/version/2.2.2/reference/api/pandas.DataFrame.drop.html>).

```
In [5]: # drop() method
dogs_clean = dogs.drop(columns=['Grade'])
dogs_clean
```

Out[5]:

	Name	Age	Breed
0	Tom	5.0	NaN
1	Lupita	4.0	Corgi
3	Olivia	1.0	Husky
4	NaN	10.0	Poodle
5	Marcel	NaN	Bulldog
6	Choco	NaN	NaN

Identifying and handling missing values

`pandas` represents missing numeric values using **NaN** (Not a Number). There are several methods to detect and drop missing values from a dataframe.

- `isnull()`

- generates a data structure of booleans, indicating whether each value is missing
- `dropna()`
 - drops some rows or columns if they contain missing data
- `fillna()`
 - fills in missing values

Display a boolean dataset showing which entries are missing.

```
In [6]: dogs_clean.isnull()
```

```
Out[6]:
```

	Name	Age	Breed
0	False	False	True
1	False	False	False
3	False	False	False
4	True	False	False
5	False	True	False
6	False	True	True

Display a count of the number of missing values in each column.

```
In [7]: dogs_clean.isnull().sum()
```

```
Out[7]: Name      1
Age        2
Breed      2
dtype: int64
```

Display what data would remain if we were to drop any row with a missing value.

```
In [8]: dogs_clean.dropna()
```

```
Out[8]:
```

	Name	Age	Breed
1	Lupita	4.0	Corgi
3	Olivia	1.0	Husky

Fill in the missing values and save the result to a variable.

- Where a name is missing, enter: 'Unknown'
- Where an age is missing, enter the mean of the known ages.
- Where a breed is missing, enter: 'Poodle'

```
In [9]: # Fill the missing values
filled = dogs_clean.fillna({'Name' : 'Unknown', 'Age' : dogs_clean.Age.mean(), 'Breed'
filled
```

Out[9]:

	Name	Age	Breed
0	Tom	5.0	Poodle
1	Lupita	4.0	Corgi
3	Olivia	1.0	Husky
4	Unknown	10.0	Poodle
5	Marcel	5.0	Bulldog
6	Choco	5.0	Poodle

Philadelphia Bike Share Live Data

Complete the code below to load a JSON live feed for a Philadelphia bike share program into a `pandas DataFrame`. It may help to look at the JSON data in a visual inspector. One way of doing this is to open the given URL in a browser.

Note: As this is live data, the results of the following analyses will vary based on when they are performed.

(I've sometimes had to add in some header data to the request, as the server might reject requests that do not include a user agent string.)

You can use the `pandas` function `pd.json_normalize`, but you need to pass it a suitable part of the JSON data. https://pandas.pydata.org/pandas-docs/version/2.2.2/reference/api/pandas.json_normalize.html (https://pandas.pydata.org/pandas-docs/version/2.2.2/reference/api/pandas.json_normalize.html).

The `indegobikes_data` object returned by `requests.get()` can be converted to a Python data structure using the `json()` method.

Once you have loaded the data, look at the `head` of the data frame, and **list** all of the columns.

Have you imported `pandas` already?

What does each row represent?

```
In [28]: import requests
indeg0_bikes_url = ("https://www.rideindeg0.com/stations/json/")
headers = {'User-Agent': "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.13; rv:63.0) Gecko/"
indeg0_bikes_data = requests.get(indeg0_bikes_url)#, headers=headers)
```

```
In [29]: # To display the data you can use the following line to convert the returned
# data to JSON format
indeg0_bikes_data.json()
```

```
Out[29]: {'last_updated': '2025-04-28T08:13:27.908Z',
'features': [{'geometry': {'coordinates': [-75.14403, 39.94733],
'type': 'Point'},
'properties': {'id': 3005,
'name': 'Welcome Park, NPS',
'coordinates': [-75.14403, 39.94733],
'totalDocks': 13,
'docksAvailable': 11,
'bikesAvailable': 1,
'classicBikesAvailable': 1,
'smartBikesAvailable': 0,
'electricBikesAvailable': 0,
'rewardBikesAvailable': 1,
'rewardDocksAvailable': 12,
'kioskStatus': 'FullService',
'kioskPublicStatus': 'Active',
'kioskConnectionStatus': 'Active',
'kioskType': 1,
'addressStreet': '191 S. 2nd St.',
'city': 'Philadelphia',
'zip': '19106',
'lat': 39.94733,
'lon': -75.14403}]}
```

```
In [30]: bikes = pd.json_normalize(indeg0_bikes_data.json()['features'])
```

Run `info()` and `describe()` on the data to see what you can learn.

```
In [31]: # Run info() and describe() on the DataFrame
bikes.info()
bikes.describe()
```

```
<class 'pandas.core.frame.DataFrame'>
```

```
RangeIndex: 265 entries, 0 to 264
```

```
Data columns (total 36 columns):
```

#	Column	Non-Null Count	Dtype
0	type	265 non-null	object
1	geometry.coordinates	265 non-null	object
2	geometry.type	265 non-null	object
3	properties.id	265 non-null	int64
4	properties.name	265 non-null	object
5	properties.coordinates	265 non-null	object
6	properties.totalDocks	265 non-null	int64
7	properties.docksAvailable	265 non-null	int64
8	properties.bikesAvailable	265 non-null	int64
9	properties.classicBikesAvailable	265 non-null	int64
10	properties.smartBikesAvailable	265 non-null	int64
11	properties.electricBikesAvailable	265 non-null	int64
12	properties.rewardBikesAvailable	265 non-null	int64
13	properties.rewardDocksAvailable	265 non-null	int64
14	properties.kioskStatus	265 non-null	object
15	properties.kioskPublicStatus	265 non-null	object
16	properties.kioskConnectionStatus	265 non-null	object
17	properties.kioskType	265 non-null	int64
18	properties.addressStreet	265 non-null	object
19	properties.addressCity	265 non-null	object
20	properties.addressState	265 non-null	object
21	properties.addressZipCode	265 non-null	object
22	properties.bikes	265 non-null	object
23	properties.closeTime	0 non-null	object
24	properties.eventEnd	0 non-null	object
25	properties.eventStart	0 non-null	object
26	properties.isEventBased	265 non-null	bool
27	properties.isVirtual	265 non-null	bool
28	properties.kioskId	265 non-null	int64
29	properties.notes	0 non-null	object
30	properties.openTime	0 non-null	object
31	properties.publicText	265 non-null	object
32	properties.timeZone	0 non-null	object
33	properties.trikesAvailable	265 non-null	int64
34	properties.latitude	265 non-null	float64
35	properties.longitude	265 non-null	float64

```
dtypes: bool(2), float64(2), int64(12), object(20)
```

```
memory usage: 71.0+ KB
```

Out[31]:

	properties.id	properties.totalDocks	properties.docksAvailable	properties.bikesAvailable	prc
count	265.000000	265.000000	265.000000	265.000000	
mean	3218.400000	19.275472	11.562264	6.890566	
std	126.701445	4.836294	5.842381	4.754514	
min	3005.000000	9.000000	0.000000	0.000000	
25%	3100.000000	16.000000	8.000000	3.000000	
50%	3245.000000	19.000000	12.000000	6.000000	
75%	3329.000000	21.000000	15.000000	9.000000	
max	3418.000000	40.000000	33.000000	24.000000	

Display entries where the number of available docks is greater than 10.

In [32]: `bikes[bikes['properties.docksAvailable'] > 10]`

Out[32]:

	type	geometry.coordinates	geometry.type	properties.id	properties.name	properties.coord
0	Feature	[-75.14403, 39.94733]	Point	3005	Welcome Park, NPS	[-75.14403, 39.
1	Feature	[-75.20311, 39.9522]	Point	3006	40th & Spruce	[-75.20311, 39.
3	Feature	[-75.15067, 39.98081]	Point	3008	Temple University Station	[-75.15067, 39.
4	Feature	[-75.18982, 39.95576]	Point	3009	33rd & Market	[-75.18982, 39.
5	Feature	[-75.16618, 39.94711]	Point	3010	15th & Spruce	[-75.16618, 39.
...
253	Feature	[-75.15474, 39.91551]	Point	3398	4th & Oregon	[-75.15474, 39.
255	Feature	[-75.16081, 39.949]	Point	3400	12th & Walnut	[-75.16081, 39.
260	Feature	[-75.15128, 39.95508]	Point	3409	7th & Race, Franklin Square	[-75.15128, 39.
263	Feature	[-75.14808, 39.99146]	Point	3417	Germantown & Huntingdon	[-75.14808, 39.
264	Feature	[-75.15198, 40.00924]	Point	3418	Broad & Erie	[-75.15198, 40.

153 rows × 36 columns

Display the average number of available bikes for stations north of 30.96 degrees latitude and east of -75.16 degrees longitude. **Round** the result to the nearest whole number.

```
In [33]: #Display the average number of available bikes for stations north of 30.96, east of -75
filtered_bikes_df = bikes[
    (bikes['properties.latitude'] > 30.96) &
    (bikes['properties.longitude'] > -75.16)
]

average_bikes_available = round(filtered_bikes_df['properties.bikesAvailable'].mean())

print(average_bikes_available)
```

8

Display the Zip Code (American version of post code) that has the greatest number of stations.

How can you count stations in each Zip Code?

```
In [34]: bikes['properties.addressZipCode'].value_counts().index[0]
```

Out[34]: '19104'

Use `pandas` to **count** the total number of **available docks** in each Zip Code, producing a `Series` of Zip Codes and available dock counts.

You can use the `pandas` method `sum()` on a `grouby` object to add the all values in a particular group.

```
In [36]: bikes.groupby('properties.addressZipCode')['properties.docksAvailable'].sum()
```

```
Out[36]: properties.addressZipCode
19107      10
19102      55
19103     267
19104     401
19106     128
19107     204
19112      56
19121     128
19122     202
19123     164
19125      50
19127      50
19129      48
19130     243
19131      48
19132      89
19133      31
19139     102
19140      38
19143     119
19145      59
19146     267
19147     150
19148     134
19149      21
Name: properties.docksAvailable, dtype: int64
```

Using `pandas` , find the minimum and maximum number of **available bikes** and the **range between these two values** within each Zip Code.

- Write your own function to calculate the **range**.

Use the `.agg()` method and **include your own function** as one of the arguments.

Save the resulting `DataFrame` , with all three values for each Zip Code, to a CSV file.

```
In [37]: # Find the min and max number of available bikes and the range between these two values
def range_func(x):
    return x.max() - x.min()

bikes.groupby('properties.addressZipCode')['properties.bikesAvailable'].agg(['min', 'ma
```

Out[37]:

	properties.addressZipCode	min	max	range_func
0	19107	1	1	0
1	19102	1	11	10
2	19103	2	18	16
3	19104	0	12	12
4	19106	1	16	15
5	19107	0	12	12
6	19112	1	14	13
7	19121	0	13	13
8	19122	0	17	17
9	19123	3	16	13
10	19125	9	18	9
11	19127	1	13	12
12	19129	2	9	7
13	19130	1	19	18
14	19131	4	8	4
15	19132	0	6	6
16	19133	0	15	15
17	19139	2	11	9
18	19140	0	1	1
19	19143	0	13	13
20	19145	5	15	10
21	19146	0	17	17
22	19147	2	23	21
23	19148	0	24	24
24	19149	5	5	0

Write Python code using `pandas` to determine the Zip Code with the **highest median** of *docks available*.

Print the solution in an f-string to say something like:

Zip Code 19149 has the largest median: 24.

Help: https://pandas.pydata.org/pandas-docs/version/2.2.2/reference/api/pandas.DataFrame.sort_values.html

```
In [38]: median_docks = bikes.groupby('properties.addressZipCode')['properties.docksAvailable'].
max_zip = median_docks.idxmax()
print(f"Zip code {max_zip} has the largest median: {median_docks.max()}")
```

Zip code 19149 has the largest median: 21.0

Iris Flower Data

Load the `./data/iris.csv` flower data from the practical archive, and **display** the first 5 rows.

```
In [44]: iris = pd.read_csv('./data/iris.csv')
iris.head()
```

Out[44]:

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5.0	3.6	1.4	0.2	setosa

Add two extra columns to the `DataFrame` giving the ratios of sepal length to width and petal length to width.

```
In [47]: iris['Sepal.Ratio'] = iris['Sepal.Length'] / iris['Sepal.Width']
iris['Petal.Ratio'] = iris['Petal.Length'] / iris['Petal.Width']
iris.head()
```

Out[47]:

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species	Sepal.Ratio	Petal.Ratio
1	5.1	3.5	1.4	0.2	setosa	1.457143	7.0
2	4.9	3.0	1.4	0.2	setosa	1.633333	7.0
3	4.7	3.2	1.3	0.2	setosa	1.468750	6.5
4	4.6	3.1	1.5	0.2	setosa	1.483871	7.5
5	5.0	3.6	1.4	0.2	setosa	1.388889	7.0

Calculate and **display** the means of the ratios of sepal length over width and petal length over width.

```
In [48]: iris['Sepal.Ratio'].mean(), iris['Petal.Ratio'].mean()
```

```
Out[48]: (1.953529355383419, 4.31117608552762)
```

Perform some exploration on the dataset.

- **How many** classes (species) are there, and what are the species names?
- What is the distribution of the classes? (How many in each?)
- What are the characteristics of the data in general and per class?

You can use methods like `unique()` , `value_counts()` , and `describe()` .

```
In [49]: # How many classes (species) are there,
# and what are the species names?
iris['Species'].unique()
```

```
Out[49]: array(['setosa', 'versicolor', 'virginica'], dtype=object)
```

```
In [51]: # What is the distribution of the classes?
iris['Species'].value_counts()
```

```
Out[51]: setosa      50
versicolor  50
virginica   50
Name: Species, dtype: int64
```

```
In [50]: # What are the characteristics of the data in general?
iris.describe()
```

```
Out[50]:
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Sepal.Ratio	Petal.Ratio
count	150.000000	150.000000	150.000000	150.000000	150.000000	150.000000
mean	5.842667	3.057333	3.758000	1.198667	1.953529	4.311176
std	0.828223	0.435866	1.765298	0.761224	0.400735	2.489121
min	4.300000	2.000000	1.000000	0.100000	1.268293	2.125000
25%	5.100000	2.800000	1.600000	0.300000	1.546188	2.802381
50%	5.800000	3.000000	4.350000	1.300000	2.032292	3.300000
75%	6.400000	3.300000	5.100000	1.800000	2.224910	4.666667
max	7.900000	4.400000	6.900000	2.500000	2.961538	15.000000

```
In [54]: # What are the characteristics of the data per class?
iris[iris['Species'] == 'setosa'].describe()
```

```
Out[54]:
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Sepal.Ratio	Petal.Ratio
count	50.000000	50.000000	50.000000	50.000000	50.000000	50.000000
mean	5.004000	3.428000	1.462000	0.246000	1.469733	6.908000
std	0.348735	0.379064	0.173664	0.105386	0.119473	2.854545
min	4.300000	2.300000	1.000000	0.100000	1.268293	2.666667
25%	4.800000	3.200000	1.400000	0.200000	1.385684	4.687500
50%	5.000000	3.400000	1.500000	0.200000	1.463063	7.000000
75%	5.200000	3.675000	1.575000	0.300000	1.541444	7.500000
max	5.800000	4.400000	1.900000	0.600000	1.956522	15.000000

```
In [55]: # What are the characteristics of the data per class? (again)
iris[iris['Species'] == 'versicolor'].describe()
```

```
Out[55]:
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Sepal.Ratio	Petal.Ratio
count	50.000000	50.000000	50.000000	50.000000	50.000000	50.000000
mean	5.936000	2.770000	4.260000	1.326000	2.160402	3.242837
std	0.516171	0.313798	0.469911	0.197753	0.228658	0.312456
min	4.900000	2.000000	3.000000	1.000000	1.764706	2.666667
25%	5.600000	2.525000	4.000000	1.200000	2.033929	3.016667
50%	5.900000	2.800000	4.350000	1.300000	2.161290	3.240385
75%	6.300000	3.000000	4.600000	1.500000	2.232692	3.417582
max	7.000000	3.400000	5.100000	1.800000	2.818182	4.100000

```
In [56]: # What are the characteristics of the data per class? (again)
iris[iris['Species'] == 'virginica'].describe()
```

```
Out[56]:
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Sepal.Ratio	Petal.Ratio
count	50.000000	50.000000	50.000000	50.000000	50.000000	50.000000
mean	6.588000	2.974000	5.552000	2.024000	2.230453	2.782691
std	0.63588	0.322497	0.551895	0.272224	0.246992	0.405341
min	4.900000	2.200000	4.500000	1.400000	1.823529	2.125000
25%	6.22500	2.800000	5.100000	1.800000	2.031771	2.511364
50%	6.500000	3.000000	5.550000	2.000000	2.169540	2.666667
75%	6.900000	3.175000	5.875000	2.300000	2.342949	3.055556
max	7.900000	3.800000	6.900000	2.500000	2.961538	4.000000

For flowers with a petal ratio greater than 3, **report** the mean sepal ratio for each species.

```
In [60]: iris[iris['Petal.Ratio'] > 3].groupby('Species')['Sepal.Ratio'].mean()
```

```
Out[60]: Species
setosa      1.470573
versicolor  2.212210
virginica    2.410454
Name: Sepal.Ratio, dtype: float64
```

Working with NumPy

Have you imported `numpy` ?

Helpful `numpy` routines for this section:

- `np.all()`
- `np.isfinite()`
- `np.identity()`
- `np.random.normal()` (slightly different from `np.random.randn()`)

Write code to test whether these `numpy` arrays contain 0.

```
np.array([1, 2, 3, 4])
np.array([1, 2, 3, 4, 0])
```

```
In [ ]: arr1 = np.array([1, 2, 3, 4])
arr2 = np.array([1, 2, 3, 4, 0])
print(np.all(arr1 > 0))
print(np.all(arr2 > 0))
```

```
True
False
```

Test whether the array below contains any NaNs or infite numbers.

```
np.array([1, 0, np.nan, np.inf])
```

```
In [10]: arr = np.array([1, 0, np.nan, np.inf])
print(np.isfinite(arr))
```

```
[ True  True False False]
```

Create and display a 3x3 identity matrix (i.e., the diagonal elements are 1, the rest are 0).

```
In [11]: arr = np.identity(3)
         print(arr)
```

```
[[1.  0.  0.]
 [0.  1.  0.]
 [0.  0.  1.]]
```

Write code to generate an array of 10 random numbers from a normal distribution.

```
In [12]: random_nums = np.random.normal(0, 1, 10)
         print(random_nums)
```

```
[-0.51312921  1.07138898  0.82318244 -0.02963296 -1.19552067  0.06783692
  1.47566159 -1.25403759  1.40814635 -0.33700389]
```

Write code to compute the coordinates for points on a cosine curve from zero to 3π . Calculate a value every $0.2rad$.

Store the inputs to the function in `x` and the result in `y`. Then, create a `pandas DataFrame` to hold both of these. Display the contents of the `DataFrame`.

```
In [14]: x = np.arange(0, 3 * np.pi, 0.2)
y = np.cos(x)
df = pd.DataFrame({
    'x': x,
    'y': y
})

print(df)
```

	x	y
0	0.0	1.000000
1	0.2	0.980067
2	0.4	0.921061
3	0.6	0.825336
4	0.8	0.696707
5	1.0	0.540302
6	1.2	0.362358
7	1.4	0.169967
8	1.6	-0.029200
9	1.8	-0.227202
10	2.0	-0.416147
11	2.2	-0.588501
12	2.4	-0.737394
13	2.6	-0.856889
14	2.8	-0.942222
15	3.0	-0.989992
16	3.2	-0.998295
17	3.4	-0.966798
18	3.6	-0.896758
19	3.8	-0.790968
20	4.0	-0.653644
21	4.2	-0.490261
22	4.4	-0.307333
23	4.6	-0.112153
24	4.8	0.087499
25	5.0	0.283662
26	5.2	0.468517
27	5.4	0.634693
28	5.6	0.775566
29	5.8	0.885520
30	6.0	0.960170
31	6.2	0.996542
32	6.4	0.993185
33	6.6	0.950233
34	6.8	0.869397
35	7.0	0.753902
36	7.2	0.608351
37	7.4	0.438547
38	7.6	0.251260
39	7.8	0.053955
40	8.0	-0.145500
41	8.2	-0.339155
42	8.4	-0.519289
43	8.6	-0.678720
44	8.8	-0.811093
45	9.0	-0.911130
46	9.2	-0.974844
47	9.4	-0.999693

