Data wrangling in Python - Data wrangling with Pandas - 2

One should look for what is and not what he thinks should be. (Albert Einstein)

Module completion checklist

Objective	Complete
Define the use and properties of DataFrames, and apply basic operations	
Define the use and properties of Index in DataFrames, and apply basic operations	

DataFrames

- Now that we have reviewed `Series`, let's look at what a `DataFrame` is
- A **`DataFrame`** is the single most important object in Pandas
  - It is a collection of `Series` of equal lengths
  - Just like `Series`, `DataFrames` come with many useful methods
- Review complete documentation of the `DataFrame` function ***here***
- We're now going to build a sample `DataFrame` using two different Series to show the average number of days people were out of office in a given month

Series to DataFrame

- To create a `DataFrame`, we are going to need to create two `Series` first
  - The first `Series` will consist of a range of dates at monthly intervals
  - The second `Series` will consist of the average number of days people were out of office, in the form of a list of numerical values

```
# Series 1 - times:
times = pd.date_range(start = '20170101', end = '20170630', freq = 'M')

# Series 2 - days out of the office:
days = pd.Series([2, 2, 6, 6, 2, 3])
```

- Now that we have our `Series`, we can create a `DataFrame` object with the `pd.DataFrame` function, specifying the two `Series` we want to include

Generate DataFrame from Series

- Create a `DataFrame` using dictionary-like syntax:
  - Dictionary `keys` become column names of the DataFrame, and
  - Dictionary `values` become column values
- Inspect the `DataFrame` by looking at the first few rows, using `.head()`

```
# Create a DataFrame from the two Series we just created, as a dictionary.
average_ooo = pd.DataFrame({'Timestamp': times, 'OOO': days})

# View the first few rows of the DataFrame, using the Pandas function `.head()`.
print(average_ooo.head())
```

```
   Timestamp  OOO
0 2017-01-31    2
1 2017-02-28    2
2 2017-03-31    6
3 2017-04-30    6
4 2017-05-31    2
```

Look up DataFrame information

- As with arrays and lists, we can look up the `type` of the created object as well as its shape

```
# Look up the type of object.
print(type(average_ooo))
```

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

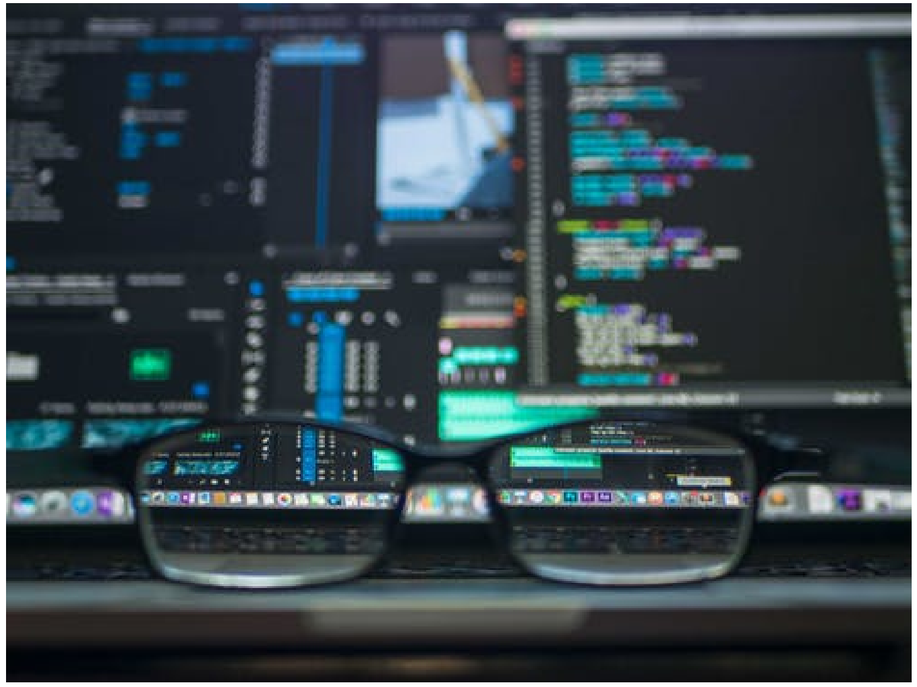
```
# Look up its shape.
print(average_ooo.shape)
```

```
(6, 2)
```

- A dataframe is a rectangular object, with rows and columns like a matrix:
  - The first number in parentheses gives us the number of rows
  - The second number is the number of columns

DataFrame description metrics

- There are many metrics you can pull from a `DataFrame` object
- We will now review some key metrics that will help us understand our data
  - `.columns` returns column names
  - `.info()` gives us some extra info about each column like its data type, and how many null values it has
  - `.describe()` computes summary statistics on any numeric column



DataFrame description metrics (cont'd)

- Now, let's preview these metrics on the `average_ooo` dataset

```
print(average_ooo.columns)
```

```
Index(['Timestamp', 'OOO'], dtype='object')
```

```
print(average_ooo.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6 entries, 0 to 5
Data columns (total 2 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   Timestamp  6 non-null
datetime64[ns]
 1   OOO        6 non-null      int64
dtypes: datetime64[ns](1), int64(1)
memory usage: 224.0 bytes
None
```

```
print(average_ooo.describe())
```

```
              OOO
count   6.000000
mean    3.500000
std     1.974842
min     2.000000
25%     2.000000
50%     2.500000
75%     5.250000
max     6.000000
```

Extracting a single column

- To extract a column, just put its name in quotation marks into square brackets like this: `data_frame['column_name']`

```
print(average_ooo['Timestamp'])
```

```
0   2017-01-31
1   2017-02-28
2   2017-03-31
3   2017-04-30
4   2017-05-31
5   2017-06-30
Name: Timestamp, dtype: datetime64[ns]
```

- The resulting object is a `Series` type
- If you would like to get a `DataFrame` object with a single column, then pass the `list` with a single column name into the square brackets like this: `data_frame[['column_name']]`

Extracting multiple columns

- To extract multiple columns, just pass a list of columns

```
print(average_ooo[['Timestamp', 'OOO']])
```

```
   Timestamp  OOO
0 2017-01-31    2
1 2017-02-28    2
2 2017-03-31    6
3 2017-04-30    6
4 2017-05-31    2
5 2017-06-30    3
```

Extracting a single row

- To extract a particular row from a dataframe, we can use a syntax similar to what we used for `ndarray`, but with one small change: **we must use the `iloc` method!**

```
june_ooo = average_ooo.iloc[5, :]
print(june_ooo)
```

```
Timestamp    2017-06-30 00:00:00
OOO                            3
Name: 5, dtype: object
```

```
june_ooo = average_ooo.iloc[5] #<- equivalent without the colon
print(june_ooo)
```

```
Timestamp    2017-06-30 00:00:00
OOO                            3
Name: 5, dtype: object
```

- `iloc` can be used when the index label of a dataframe is numeric (*integer* in *i* loc) or if you aren't sure of the index label
- Further, we're going to talk more about the importance of the index in Pandas

Module completion checklist

Objective	Complete
Define the use and properties of DataFrames, and apply basic operations	✔
Define the use and properties of Index in dataframes, and apply basic operations	

Working with DataFrame indices

- `Dataframes` in Pandas have a property called the `index`
- The `index` serves many purposes and is an important concept to understand within Pandas

- Some main purposes are:
  - **identifying** data using known indicators, important for analysis, visualization, and interactive console display
  - **enabling** automatic and explicit data alignment
  - **facilitating** intuitive getting and setting of subsets of the dataset

Index for our dataset

- The **index** of the `average_ooo` dataframe is an **unlabeled column** consisting of the numbers `0` to `5`
- By default, the `index` is simply the row number (starting with `0`), but it can sometimes make sense to use something more descriptive for the index
- We are going to use `set_index` to set our index in `average_ooo`

```
# Let's use the `Timestamp` column as our new index.
average_ooo = average_ooo.set_index('Timestamp')
print(average_ooo)
```

```
                OOO
Timestamp
2017-01-31      2
2017-02-28      2
2017-03-31      6
2017-04-30      6
2017-05-31      2
2017-06-30      3
```

Looking up by the new index

- Now the rows of our dataframe are indexed by the time stamp and the `Timestamp` column has been removed
- This makes it really easy to look up values corresponding to a particular time stamp, rather than to an arbitrary row number
- To do so, we now use the `.loc()` method

```
print(average_ooo.index)
```

```
DatetimeIndex(['2017-01-31', '2017-02-28', '2017-03-31', '2017-04-30',
               '2017-05-31', '2017-06-30'],
              dtype='datetime64[ns]', name='Timestamp', freq=None)
```

```
# Look up a specific row by index.
print(average_ooo.loc['2017-02-28'])
```

```
OOO    2
Name: 2017-02-28 00:00:00, dtype: int64
```

Loc vs. iloc

- Notice that after implementing our new index, we used `loc` rather than `iloc`
- The "`i`" in `iloc` stands for integer, so we can only pass it numerical values
- `iloc` is helpful if
  - Your index still consists of the default integer values, i.e. the row numbers
  - You need to reference your rows by their numeric order and not by the index value you've set (e.g. by using a counter in a loop)
- Since the row we wanted was in position `1`, we could also use `iloc` to look it up:

```
print(average_ooo.iloc[1])
```

```
OOO    2
Name: 2017-02-28 00:00:00, dtype: int64
```

Reset the index

- We can change the index back to the default integer values by using `.reset_index()`
- Note that the original unlabeled index column is back and that `Timestamp` is back in the `DataFrame` as a labeled column

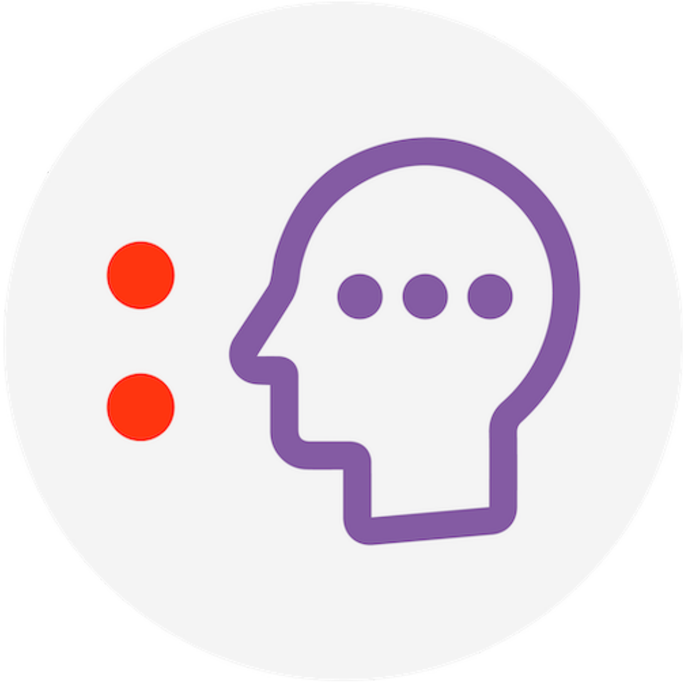
```
average_ooo = average_ooo.reset_index()
print(average_ooo.index)
```

```
RangeIndex(start=0, stop=6, step=1)
```

```
# You can see that now `Timestamp` is once again a column vs. what it looked like when it was an
index.
print(average_ooo)
```

```
   Timestamp  OOO
0 2017-01-31    2
1 2017-02-28    2
2 2017-03-31    6
3 2017-04-30    6
4 2017-05-31    2
5 2017-06-30    3
```

Knowledge check



Link: ***https://forms.gle/rLRZXU8MDaTUwxSc6***

Module completion checklist

Objective	Complete
Define the use and properties of DataFrames, and apply basic operations	✔
Define the use and properties of Index in DataFrames, and apply basic operations	✔

Congratulations on completing this module!

You are now ready to try Tasks 5-8 in the Exercise for this topic