Data Munging Basics

Segment 1 - Filtering and selecting data

****

```
import numpy as np
import pandas as pd
from pandas import Series, DataFrame
```

****

Selecting and retrieving data

```
series_obj = Series(np.arange(8), index=['row 1', 'row 2','row 3','row 4','row 5', 'row 6', 'row 7', 'row 8'])
print(series_obj)
```

****

```
# [integer index]
# ♔ ♔ ♔ ( WHAT THIS DOES ) ♔ ♔ ♔
# When you write square brackets with an integer index inside them, this tells Python to select and
# retrieve all records with the specified integer index.
print(series_obj[[0,7]])
```

****

```
np.random.seed(25)
DF_obj = DataFrame(np.random.rand(36).reshape((6,6)),
            index=['row 1', 'row 2', 'row 3', 'row 4', 'row 5', 'row 6'],
            columns=['column 1', 'column 2', 'column 3', 'column 4', 'column 5', 'column 6'])
print(DF_obj)
```

****

```
# object_name.loc[[row indexes], [column indexes]]
# ♔ ♔ ♔ ( WHAT THIS DOES ) ♔ ♔ ♔
# When you call the .ix[] special indexer, and pass in a set of row and column indexes, this tells Python to
select and retrieve only those specific rows and columns.
DF_obj.loc[['row 2', 'row 5'], ['column 5', 'column 2']]
```

****

Data slicing

```
# ['starting label-index':'ending label-index']
# ♔ ♔ ♔ ( WHAT THIS DOES ) ♔ ♔ ♔
# Data slicing allows you to select and retrieve all records from the starting label-index, to the
# ending label-index, and every record in between.
Print(series_obj['row 3':'row 7'])
```

****

Comparing with scalars

```
# object_name < scalar value
# ♔ ♔ ♔ ( WHAT THIS DOES ) ♔ ♔ ♔
# You can use comparison operators (like greater than or less than) to return True / False values for
```

```
# all records, to indicate how each element compares to a scalar value.
Print(DF_obj < .2)
**********
```

Filtering with scalars

```
# object_name[object_name > scalar value]
# ♔ ♔ ♔ ( WHAT THIS DOES ) ♔ ♔ ♔
# You can also use comparison operators and scalar values for indexing, to return only the records
# that satisfy the comparison expression you write.
Print(series_obj[series_obj > 6])
**********
```

Setting values with scalars

```
# ['label-index', 'label-index', 'label-index'] = scalar value
# ♔ ♔ ♔ ( WHAT THIS DOES ) ♔ ♔ ♔
# Setting is where you select all records associated with the specified label-indexes and set those
# values equal to a scalar.
series_obj['row 1', 'row 5', 'row 8'] = 8
print(series_obj)
**********
```

Segment 2 - Treating missing values

```
import numpy as np
import pandas as pd
from pandas import Series, DataFrame
**********
```

Figuring out what data is missing

```
missing = np.nan
series_obj = Series(['row 1', 'row 2', missing, 'row 4','row 5', 'row 6', missing, 'row 8'])
print(series_obj)
**********
# object_name.isnull()
# ♔ ♔ ♔ ( WHAT THIS DOES ) ♔ ♔ ♔
# The .isnull() method returns a Boolean value that describes (True or False) whether an element in a
# Pandas object is a null value.
print(series_obj.isnull())
**********
```

Filling in for missing values

```
np.random.seed(25)
DF_obj = DataFrame(np.random.randn(36).reshape(6,6))
print(DF_obj)
**********
```

```
DF_obj.iloc[3:5, 0] = missing
DF_obj.iloc[1:4, 5] = missing
print(DF_obj)
********************************************************************************
# object_name.fillna(numeric value)
# ♔ ♔ ♔ ( WHAT THIS DOES ) ♔ ♔ ♔
# The .fillna method() finds each missing value from within a Pandas object and fills it with the
# numeric value that you've passed in.
filled_DF = DF_obj.fillna(0)
print(filled_DF)
********************************************************************************
# object_name.fillna(dict)
# ♔ ♔ ♔ ( WHAT THIS DOES ) ♔ ♔ ♔
# You can pass a dictionary into the .fillna() method. The method will then fill in missing values
# from each column Series (as designated by the dictionary key) with its own unique value
# (as specified in the corresponding dictionary value).
filled_DF = DF_obj.fillna({0: 0.1, 5: 1.25})
print(filled_DF)
********************************************************************************
# ♔ ♔ ♔ ( WHAT THIS DOES ) ♔ ♔ ♔
# You can also pass in the method='ffill' arguement, and the .fillna() method will fill-forward any
# missing values with values from the last non-null element in the column Series.
fill_DF = DF_obj.fillna(method='ffill')
print(fill_DF)
********************************************************************************
```

Counting missing values

```
np.random.seed(25)
DF_obj = DataFrame(np.random.randn(36).reshape(6,6))
DF_obj.iloc[3:5, 0] = missing
DF_obj.iloc[1:4, 5] = missing
print(DF_obj)
********************************************************************************
# object_name.isnull().sum()
# ♔ ♔ ♔ ( WHAT THIS DOES ) ♔ ♔ ♔
# To generate a count of how many missing values a DataFrame has per column, just call the .isnull()
# method off of the object, and then call the .sum() method off of the matrix of Boolean values it
# returns.
print(DF_obj.isnull().sum())
********************************************************************************
```

Filtering out missing values

```
# object_name.dropna()
```

```
# ♔ ♔ ♔ ( WHAT THIS DOES ) ♔ ♔ ♔
# To identify and drop all rows from a DataFrame that contain ANY missing values, simply call the
# .dropna() method off of the DataFrame object. NOTE: If you wanted to drop columns that contain
# any missing values, you'd just pass in the axis=1 argument to select and search the DataFrame
# by columns, instead of by row.
DF_no_NaN = DF_obj.dropna(axis=1)
print(DF_no_NaN)
**********
# object_name.dropna(how='all')
# ♔ ♔ ♔ ( WHAT THIS DOES ) ♔ ♔ ♔
# To identify and drop only the rows from a DataFrame that contain ALL missing values, simply
# call the .dropna() method off of the DataFrame object, and pass in the how='all' argument.
print(DF_obj.dropna(how='all'))
**********
```

Segment 3 - Removing duplicates

```
import numpy as np
import pandas as pd
from pandas import Series, DataFrame
**********
```

Removing duplicates

```
DF_obj = DataFrame({'column 1': [1, 1, 2, 2, 3, 3, 3],
                    'column 2': ['a', 'a', 'b', 'b', 'c', 'c', 'c'],
                    'column 3': ['A', 'A', 'B', 'B', 'C', 'C', 'C']})
print(DF_obj)
**********
# object_name.duplicated()
# ♔ ♔ ♔ ( WHAT THIS DOES ) ♔ ♔ ♔
# The .duplicated() method searches each row in the DataFrame, and returns a True or False value to
#indicate whether it is a duplicate of another row found earlier in the DataFrame.
print(DF_obj.duplicated())
**********
# object_name.drop_duplicates()
# ♔ ♔ ♔ ( WHAT THIS DOES ) ♔ ♔ ♔
# To drop all duplicate rows, just call the drop_duplicates() method off of the DataFrame.
print(DF_obj.drop_duplicates())
**********
DF_obj = DataFrame({'column 1': [1, 1, 2, 2, 3, 3, 3],
                    'column 2': ['a', 'a', 'b', 'b', 'c', 'c', 'c'],
                    'column 3': ['A', 'A', 'B', 'B', 'C', 'D', 'C']})
print(DF_obj)
```

```
**********************************************************************************
# object_name.drop_duplicates(['column_name'])
# ♔ ♔ ♔ ( WHAT THIS DOES ) ♔ ♔ ♔
# To drop the rows that have duplicates in only one column Series, just call the drop_duplicates()
# method of the DataFrame, and pass in the label-index of the column you want the de-duplication
# to be based on. This method will drops all rows that have duplicates in the column you specify.
print(DF_obj.drop_duplicates(['column 3']))
**********************************************************************************
```

Segment 4 - Concatenating and transforming data

```
import numpy as np
import pandas as pd
from pandas import Series, DataFrame
**********************************************************************************
DF_obj = pd.DataFrame(np.arange(36).reshape(6,6))
print(DF_obj)
**********************************************************************************
DF_obj_2 = pd.DataFrame(np.arange(15).reshape(5,3))
print(DF_obj_2)
**********************************************************************************
```

Concatenating data

```
# pd.concat([left_object, right_object], axis=1)
# ♔ ♔ ♔ ( WHAT THIS DOES ) ♔ ♔ ♔
# The concat() method joins data from seperate sources into one combined data table. If you want to
# join objects based on their row index values, just call the pd.concat() method on the objects you
# want joined, and then pass in the axis=1 argument. The axis=1 argument tells Python to concatenate
# the DataFrames by adding columns (in other words, joining on the row index values).
print(pd.concat([DF_obj, DF_obj_2], axis =1))
**********************************************************************************
print(pd.concat([DF_obj, DF_obj_2]))
**********************************************************************************
```

Transforming data

Dropping data

```
# object_name.drop([row indexes])
# ♔ ♔ ♔ ( WHAT THIS DOES ) ♔ ♔ ♔
# You can easily drop rows from a DataFrame by calling the .drop() method and passing in the index
# values for the rows you want dropped.
print(DF_obj.drop([0,2]))
**********************************************************************************
```

```
print(DF_obj.drop([0,2], axis=1))
********************************************************************************
```

Adding data

```
series_obj = Series(np.arange(6))
series_obj.name = "added_variable"
print(DF_obj.drop([0,2], axis=1))
********************************************************************************
# DataFrame.join(left_object, right_object)
# ♔ ♔ ♔ ( WHAT THIS DOES ) ♔ ♔ ♔
# You can use .join() method two join two data sources into one. The .join() method works by joining
# the two sources on their row index values.
variable_added = DataFrame.join(DF_obj, series_obj)
print(variable_added)
********************************************************************************
added_datatable = variable_added.append(variable_added, ignore_index=False)
print(added_datatable)
********************************************************************************
added_datatable = variable_added.append(variable_added, ignore_index=True)
print(added_datatable)
********************************************************************************
```

Sorting data

```
# object_name.sort_values(by=[index value], ascending=[False])
# ♔ ♔ ♔ ( WHAT THIS DOES ) ♔ ♔ ♔
# To sort rows in a DataFrame, either in ascending or descending order, call the .sort_values()
# method off of the DataFrame, and pass in the by argument to specify the column index upon which
# the DataFrame should be sorted.
DF_sorted = DF_obj.sort_values(by=[5], ascending=[False])
print(DF_sorted)
********************************************************************************
```

Segment 5 - Grouping and data aggregation

```
import numpy as np
import pandas as pd
from pandas import Series, DataFrame
********************************************************************************
```

Grouping data by column index

```
filepath = 'mtcars.csv'
cars = pd.read_csv(filepath)
cars.columns = ['car_names','mpg','cyl','disp', 'hp', 'drat', 'wt', 'qsec', 'vs', 'am', 'gear', 'carb']
print(cars.head())
********************************************************************************
# object_name.groupby('Series_name')
```

# ♔ ♔ ♔ (WHAT THIS DOES) ♔ ♔ ♔
# To group a DataFrame by its values in a particular column, call the .groupby() method off of the DataFrame, and then pass
# in the column Series you want the DataFrame to be grouped by.
cars_groups = cars.groupby(cars['cyl'])
print(cars_groups.mean())
****