Introduction to Pandas

Pandas is a Python module that contains structures and functions useful for data exploration and analysis. The two main data structures Pandas introduces are **Series** and **DataFrames**.

Pandas Series

- 1-D data structure (similar to Python lists, or an Excel column)
- Can contain multiple data types, but usually should contain data of one type
- Create a Pandas Series by passing in a **list** to **pd.Series()**
- By default, a Pandas Series will have an index that starts at 0; can access specific values using this index
- Learn more: https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.html

[1]
```
# Import Pandas module

import pandas as pd
```

[2]
```
# Creating a Pandas Series

my_list = [100, 200, 400, 600, 900]
my_series = pd.Series(my_list)
my_series
```

```
0    100
1    200
2    400
3    600
4    900
dtype: int64
```

[3]
```
# Accessing specific values within Series

print(my_series[1]) # will print 200
print(my_series[3]) # will print 600
```

```
200
600
```

Pandas DataFrames

- 2-D data structure with labeled rows and columns (similar to tables in Excel)
    - For example: if we were looking at traffic violations data for NYC, each row could represent a violation instance, and each column could represent a specific attribution of a violation (date, amount of fine, location, etc.)
- Create a Pandas Dataframe by using **pd.DataFrame()**, and passing in either a **list of dictionaries**, or a **dictionary with lists**
- A lot of data in the real world will be provided in tabular format which can be easily translated into DataFrames
- Learn more: https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html

[5]
```
# Creating a Pandas DataFrame by passing in a LIST OF DICTIONARIES
# Each value in the list is a dictionary
# Imagine that each dictionary represents a row of data in our eventual d
# Each dictionary should have the same keys, since these keys dictate the

my_list = [{"id": 1, "name": "Bob", "account_balance": 500.14},
           {"id": 2, "name": "Amanda", "account_balance": 300.42},
           {"id": 3, "name": "Jill", "account_balance": 943.54},
           {"id": 4, "name": "Dylan", "account_balance": 112.53},
           {"id": 5, "name": "Alex", "account_balance": 895.51}]

my_df_1 = pd.DataFrame(my_list)
my_df_1
```

	account_balance	id	name
0	500.14	1	Bob
1	300.42	2	Amanda
2	943.54	3	Jill
3	112.53	4	Dylan
4	895.51	5	Alex

[6]
```
# Re-create the previous Pandas DataFrame, passing in a DICTIONARY WITH L
# The keys of the dictionary represent the column headers of our eventual
# The lists contain the data for each column

my_dict = {"id": [1, 2, 3, 4, 5],
           "name": ["Bob", "Amanda", "Jill", "Dylan", "Alex"],
           "account_balance": [500.14, 300.42, 943.54, 112.53, 895.51]}

my_df_2 = pd.DataFrame(my_dict)
my_df_2
```

	account_balance	id	name
0	500.14	1	Bob

	account_balance	id	name
1	300.42	2	Amanda
2	943.54	3	Jill
3	112.53	4	Dylan
4	895.51	5	Alex

[7]
```
# Select a single column from a dataframe by passing in the column's name

my_df_2["account_balance"]
```

```
0    500.14
1    300.42
2    943.54
3    112.53
4    895.51
Name: account_balance, dtype: float64
```

[8]
```
# You can also select a single column and assign it to another variable

names_col = my_df_2["name"]
names_col
```

```
0       Bob
1    Amanda
2      Jill
3     Dylan
4      Alex
Name: name, dtype: object
```

[9]
```
# Now names_col contains only the "names" column

print(names_col[1])
```

```
Amanda
```

[10]
```
# Select multiple columns from a dataframe by passing in a list of the na

my_df_2[["name", "account_balance"]]
```

	name	account_balance
0	Bob	500.14
1	Amanda	300.42
2	Jill	943.54

	name	account_balance
3	Dylan	112.53
4	Alex	895.51

Important DataFrame Functions

.head() returns the first 5 rows of data

[11]
```
# Create a new DataFrame that represents purchase data from an online ret

my_dict_2 = {"order_id": [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13],
             "price": [13.50, 9.99, 12.00, 29.99,
                       14.99, 7.99, 3.49, 10.00,
                       9.99, 17.99, 20.00, 21.00, 14.99],
             "purchase_category": ["Apparel", "Sports", "Toys",
                                   "Apparel", "Apparel", "Household",
                                   "Household", "Toys", "Sports",
                                   "Sports", "Apparel", "Household", "App
             "clicked_ad": [True, True, False, True, False,
                            True, True, False, False, True,
                            True, True, False]}
purchase_df = pd.DataFrame(my_dict_2)
```

[12]
```
# Show the first 5 rows of data using .head()
# .head() is great for getting a taste of the data you're dealing with

purchase_df.head()
```

	clicked_ad	order_id	price	purchase_category
0	True	1	13.50	Apparel
1	True	2	9.99	Sports
2	False	3	12.00	Toys
3	True	4	29.99	Apparel
4	False	5	14.99	Apparel

.describe() returns a table of summary statistics on numeric columns in a dataframe

[13]
```
# Note that .describe() will only return summary statistics for your nume
# In this case, statistics for order_id and price columns are returned
```

```
purchase_df.describe()
```

	order_id	price
count	13.00000	13.000000
mean	7.00000	14.301538
std	3.89444	6.809116
min	1.00000	3.490000
25%	4.00000	9.990000
50%	7.00000	13.500000
75%	10.00000	17.990000
max	13.00000	29.990000

.mean() returns the average of all values in a given column or dataframe

[14]
```
# Return the mean of the price column

purchase_df["price"].mean()
```

```
14.30153846153846
```

.sum() returns the sum of all values in a given column or dataframe

[15]
```
# Return the sum of all values in the order_id column

purchase_df["order_id"].sum()
```

```
91
```

[16]
```
# These values can also be assigned to variables

mean_price = purchase_df["price"].mean()
sum_order_id = purchase_df["order_id"].sum()

mean_price + sum_order_id
```

```
105.30153846153846
```

.unique() returns an array of all of the unique values within a given column

```
[17]  # Returns unique values in the purchase_category column

      unique_pcat = purchase_df["purchase_category"].unique()
      print(unique_pcat)
      print(unique_pcat[2])
```

```
['Apparel' 'Sports' 'Toys' 'Household']
Toys
```

.value_counts() returns an array containing the # of times each unique value occurs in a given column

```
[18]  # Returns the value counts of each unique value in the purchase_category

      print(purchase_df["purchase_category"].value_counts())
```

```
Apparel      5
Sports       3
Household    3
Toys         2
Name: purchase_category, dtype: int64
```

Exploring Pandas DataFrames

Two functions exist to make life easier when trying to slice and dice any DataFrame: **.iloc** and **.loc**

.iloc uses the *numeric* indexes of a dataframe's rows and columns to return specific values

```
[19]  # Use the purchase_df dataframe created above

      purchase_df
```

	clicked_ad	order_id	price	purchase_category
0	True	1	13.50	Apparel
1	True	2	9.99	Sports

	clicked_ad	order_id	price	purchase_category
2	False	3	12.00	Toys

3	True	4	29.99	Apparel
4	False	5	14.99	Apparel
5	True	6	7.99	Household
6	True	7	3.49	Household
7	False	8	10.00	Toys
8	False	9	9.99	Sports
9	True	10	17.99	Sports
10	True	11	20.00	Apparel
11	True	12	21.00	Household
12	False	13	14.99	Apparel

There are several possible ways to use **.iloc**
The general structure is: **.iloc*rows you want, columns you want***

- Use single values to just get one row/column
- Use a colon (:) to get all rows/columns
- Use a list to get specific rows/columns
- Use a range(x:y) to get a range of rows/columns

[20]
```
# To return ALL ROWS and COLUMN 2 (order_id)

purchase_df.iloc[ : , 1]

# The colon before the comma in .iloc[] means we want ALL rows
# The 1 after the comma means we want the column at index 1
```

```
0      1
1      2
2      3
3      4
4      5
5      6
6      7
7      8
8      9
9     10
10    11
11    12
12    13
Name: order_id, dtype: int64
```

[21]
```
# To return ROWS 1 THROUGH 4 (including 4), and ALL COLUMNS

purchase_df.iloc[0:4, : ]
```

	clicked_ad	order_id	price	purchase_category
0	True	1	13.50	Apparel
1	True	2	9.99	Sports
2	False	3	12.00	Toys
3	True	4	29.99	Apparel

[22]
```
# To returns ROWS 2, 3, AND 5, and COLUMNS 2 THROUGH 4 (including 4)

purchase_df.iloc[[1, 2, 4], 1:4]
```

	order_id	price	purchase_category
1	2	9.99	Sports
2	3	12.00	Toys
4	5	14.99	Apparel

.loc uses the *named* indexes of a dataframe's rows and columns to return specific values.

The general structure for .loc[] is the same as that for .iloc[], except named indexes are used instead of numeric indexes
A column's named index is simply its **column name**
By default, when we create a dataframe, a row's index is numeric and starts at 0. You can set a named index for a dataframe's rows by using **.set_index()**

[23]
```
# Create a new dataframe
example_dict = {"first_name": ["Bill", "James", "Tyler", "Matt", "Jon"],
                "last_name": ["Smith", "Alvarez", "Dant", "May", "Livings
                "age": [25, 34, 52, 26, 43],
                "credit_score": [721, 683, 761, 641, 602]}
credit_df = pd.DataFrame(example_dict)
credit_df
```

	age	credit_score	first_name	last_name
0	25	721	Bill	Smith
1	34	683	James	Alvarez
2	52	761	Tyler	Dant
3	26	641	Matt	May
4	43	602	Jon	Livingston

[24]
```
# Set the row index to be the first_name

credit_df = credit_df.set_index("first_name")
credit_df
```

	age	credit_score	last_name
first_name			
Bill	25	721	Smith
James	34	683	Alvarez
Tyler	52	761	Dant
Matt	26	641	May
Jon	43	602	Livingston

[25]
```
# Now, we can filter this dataframe using .loc[]

# Return data for James' and Tyler's rows, ALL COLUMNS included

credit_df.loc[["James", "Tyler"], : ]
```

	age	credit_score	last_name
first_name			
James	34	683	Alvarez
Tyler	52	761	Dant

[26]
```
# Return rows from Bill to Matt (including Matt), and only the age and cr

credit_df.loc["Bill":"Matt", ["age", "credit_score"]]
```

	age	credit_score
first_name		
Bill	25	721
James	34	683
Tyler	52	761
Matt	26	641

[27]
```
# Return all rows, only the credit_score column
```

```
credit_df.loc[ : , "credit_score"]
```

```
first_name
Bill      721
James     683
Tyler     761
Matt      641
Jon       602
Name: credit_score, dtype: int64
```

Remember, you can get the same data you want using either .loc or .iloc
The two functions essentially perform the same task, but with different methods of operation

```
[ ]
```