Importing, Exploring, and Manipulating Data

Intro to Pandas

A package that helps python become a better and more efficient source for data representation

Built on top of the NumPy package (discussed previously!)

Perks!!

Good for cleaning data because allows for deleting and inserting of columns in DataFrames and higher dimensional objects

Allows for either explicit or automatic data alignment, depending on what the user decides, but either way, the data will be aligned.

Easy handling of missing data!

Intelligent and quick label-based indexing, and subsetting of large data sets

In [ ]:
```
#Import it:
import numpy as np
import pandas as pd
```

Has two primary data structures!

Series

Series are 1 dimensional, and are created using a list:

In [ ]:
```
series = pd.Series([1, 2, 3])
```

DataFrame

DataFrames are 2 dimensional, and are created using a NumPy array

Example 1:

In [ ]:
```
dates = pd.date_range('20130101', periods=6)
# Creates ['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
#                '2013-01-05', '2013-01-06'],

df = pd.DataFrame(np.random.randn(6, 4), index=dates, columns=list('ABCD'))
#  Creates          A         B         C         D
#2013-01-01  0.469112 -0.282863 -1.509059 -1.135632
#2013-01-02  1.212112 -0.173215  0.119209 -1.044236
#2013-01-03 -0.861849 -2.104569 -0.494929  1.071804
#2013-01-04  0.721555 -0.706771 -1.039575  0.271860
#2013-01-05 -0.424972  0.567020  0.276232 -1.087401
#2013-01-06 -0.673690  0.113648 -1.478427  0.524988
```

credit : https://pandas.pydata.org/pandas-docs/stable/getting_started/10min.html

Example 2:

In [ ]:
```
df2 = pd.DataFrame({'A': 1.,
                    'B': pd.Timestamp('20130102'),
                    'C': pd.Series(1, index=list(range(4)), dtype='float32'),
                    'D': np.array([3] * 4, dtype='int32'),
                    'E': pd.Categorical(["test", "train", "test", "train"]),
                    'F': 'foo'})
# Creates:
#     A           B    C  D      E    F
#0  1.0 2013-01-02  1.0  3   test  foo
#1  1.0 2013-01-02  1.0  3  train  foo
#2  1.0 2013-01-02  1.0  3   test  foo
#3  1.0 2013-01-02  1.0  3  train  foo
```

credit : https://pandas.pydata.org/pandas-docs/stable/getting_started/10min.html

Viewing Data

To view the top rows of the data:

In [ ]:
```
df2.head()
```

Out[ ]:

	A	B	C	D	E	F
0	1.0	2013-01-02	1.0	3	test	foo
1	1.0	2013-01-02	1.0	3	train	foo
2	1.0	2013-01-02	1.0	3	test	foo
3	1.0	2013-01-02	1.0	3	train	foo

To view the bottom rows of the data:

In [ ]:
```
df2.tail()
```

Out[ ]:

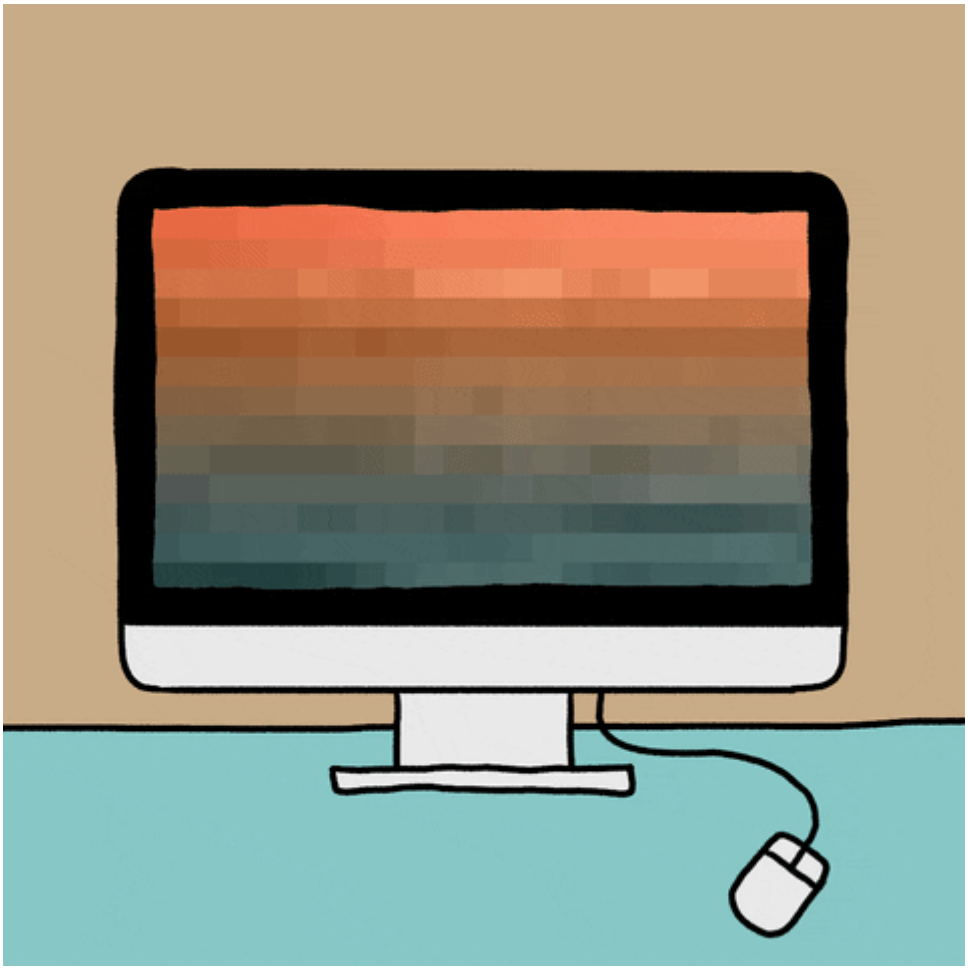
	A	B	C	D	E	F
0	1.0	2013-01-02	1.0	3	test	foo
1	1.0	2013-01-02	1.0	3	train	foo
2	1.0	2013-01-02	1.0	3	test	foo
3	1.0	2013-01-02	1.0	3	train	foo

For both of these, the parameter would be the number of rows you want to view!

Importing and Exploring Data

Import data from a spreadsheet (.csv format) using `read_csv()`. We're going to be using a data set that has information about the Kansas City housing market. It contains info about

various attributes of a house (number of bedrooms, number of floors, squarefoot size of living space, year built, etc.) and the price of each house.



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

data = pd.read_csv("../housing_data.csv")
```

As covered earlier, `.head()` and `.tail()` can be used to view the top and bottom rows of much bigger datasets stored in a dataframe, too:

In [ ]:
```
data.head()
```

Out[ ]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floor
0	7129300520	20141013T000000	221900.0	3	1.00	1180	5650	1.
1	6414100192	20141209T000000	538000.0	3	2.25	2570	7242	2.
2	5631500400	20150225T000000	180000.0	2	1.00	770	10000	1.
3	2487200875	20141209T000000	604000.0	4	3.00	1960	5000	1.
4	1954400510	20150218T000000	510000.0	3	2.00	1680	8080	1.

5 rows × 21 columns

In [ ]:
```
data.tail()
```

Out[ ]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot
21608	263000018	20140521T000000	360000.0	3	2.50	1530	1131
21609	6600060120	20150223T000000	400000.0	4	2.50	2310	5813
21610	1523300141	20140623T000000	402101.0	2	0.75	1020	1350
21611	291310100	20150116T000000	400000.0	3	2.50	1600	2388
21612	1523300157	20141015T000000	325000.0	2	0.75	1020	1076

5 rows × 21 columns

For a summary of the entire dataframe, use `.info()`. This tells you what the column names are, how many values are in each column, whether all the values in a column are filled (`non-null`), and what the data type of the values each column are.

In [ ]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21613 entries, 0 to 21612
Data columns (total 21 columns):
id               21613 non-null int64
date             21613 non-null object
price            21613 non-null float64
bedrooms         21613 non-null int64
bathrooms        21613 non-null float64
sqft_living      21613 non-null int64
sqft_lot         21613 non-null int64
floors           21613 non-null float64
waterfront       21613 non-null int64
view             21613 non-null int64
condition        21613 non-null int64
grade            21613 non-null int64
sqft_above       21613 non-null int64
sqft_basement    21613 non-null int64
yr_built         21613 non-null int64
yr_renovated     21613 non-null int64
zipcode          21613 non-null int64
lat              21613 non-null float64
long             21613 non-null float64
sqft_living15    21613 non-null int64
sqft_lot15       21613 non-null int64
dtypes: float64(5), int64(15), object(1)
memory usage: 3.5+ MB
```

Manipulating Data

Once you have data in Pandas, there are various Pandas commands that allow you to explore and manipulate your data even further. For example, you can filter your data for certain conditions using `.loc()`. To filter for houses with 2 bedrooms and 2 bathrooms and see their prices, use:

In [ ]:
```
data.loc[(data["bedrooms"]==2) & (data["bathrooms"]==2), ["bedrooms","bathrooms
```

Out[ ]:

	bedrooms	bathrooms	price
226	2	2.0	479950.0
255	2	2.0	592500.0
438	2	2.0	438000.0
470	2	2.0	290900.0
487	2	2.0	207950.0
525	2	2.0	727500.0
537	2	2.0	595000.0
547	2	2.0	259950.0
640	2	2.0	378000.0
773	2	2.0	450000.0
817	2	2.0	250000.0
1032	2	2.0	554000.0
1138	2	2.0	219000.0
1187	2	2.0	545000.0
1275	2	2.0	363000.0
1305	2	2.0	428000.0
1399	2	2.0	408000.0
1605	2	2.0	512000.0
1717	2	2.0	360000.0
1753	2	2.0	368000.0
1825	2	2.0	501000.0
2032	2	2.0	210000.0
2049	2	2.0	575000.0
2058	2	2.0	439900.0
2303	2	2.0	210000.0
2352	2	2.0	172500.0
2436	2	2.0	360000.0
2487	2	2.0	335000.0
2530	2	2.0	509000.0
2758	2	2.0	400000.0
...	...	...	...
18842	2	2.0	330000.0
18960	2	2.0	480000.0
19025	2	2.0	400000.0
19131	2	2.0	163000.0

	bedrooms	bathrooms	price
19287	2	2.0	316000.0
19321	2	2.0	478000.0
19412	2	2.0	445000.0
19480	2	2.0	885000.0
19591	2	2.0	564000.0
19600	2	2.0	1410000.0
19787	2	2.0	439950.0
19922	2	2.0	455000.0
20134	2	2.0	548000.0
20446	2	2.0	405000.0
20492	2	2.0	670000.0
20495	2	2.0	399000.0
20523	2	2.0	459000.0
20545	2	2.0	455000.0
20567	2	2.0	700000.0
20604	2	2.0	256950.0
20614	2	2.0	308625.0
20618	2	2.0	610000.0
20688	2	2.0	485000.0
20786	2	2.0	411000.0
20792	2	2.0	699999.0
21002	2	2.0	529000.0
21106	2	2.0	380950.0
21512	2	2.0	406000.0
21547	2	2.0	327000.0
21556	2	2.0	553000.0

216 rows × 3 columns

You can also apply a function to the data to do some computation using the `apply()` function. For example, you can apply the `numpy` `sum` function to each column in the data set by using the `apply` function, passing in the `np.sum` function as a parameter, and setting the `axis` parameter to 0 to let it know to apply the function along columns (to apply along rows, you could also do `axis = 1`):

In [ ]:
```
data.apply(np.sum, axis = 0)
```

Out[ ]:
```
id                                                    98994056770455
date          20141013T000000201412097000000201502257000000...
price                                                 1.16729e+10
bedrooms                                                    72854
bathrooms                                                 45706.2
sqft_living                                              44952873
sqft_lot                                                326506890
floors                                                    32296.5
waterfront                                                    163
view                                                         5064
condition                                                   73688
grade                                                      165488
sqft_above                                               38652488
sqft_basement                                             6300385
yr_built                                                 42599334
yr_renovated                                              1824186
zipcode                                                2119758513
lat                                                   1.02792e+06
long                                                 -2.64141e+06
sqft_living15                                            42935359
sqft_lot15                                              275964632
dtype: object
```

`.apply()` returns another DataFrame, which you can query just like any other DataFrame in Pandas, so if you wanted to get values in a particular row, you could still:

In [ ]:
```
data.apply(np.sum, axis = 0)['waterfront']
```

Out[ ]:
```
163
```

`.apply` also works with functions you write yourself!

In [ ]:
```
def count_missing_vals(x):
  return sum(x.isnull())

data.apply(count_missing_vals, axis=0)
```

Out[ ]:
```
id               0
date             0
price            0
bedrooms         0
bathrooms        0
sqft_living      0
sqft_lot         0
floors           0
waterfront       0
view             0
condition        0
grade            0
sqft_above       0
sqft_basement    0
yr_built         0
yr_renovated     0
zipcode          0
lat              0
long             0
sqft_living15    0
sqft_lot15       0
dtype: int64
```

You can also sort a DataFrame by any values you want. For example, you can sort by the year a home was renovated, specifically sorting by decending value by setting `ascending=False`. Sorting the DataFrame still returns a DataFrame, so you can query it for certain columns as usual:

In [ ]:
```
sorted_reno_date = data.sort_values(['yr_renovated'], ascending=False)
sorted_reno_date[['price','yr_built','yr_renovated']]
```

Out[ ]:

	price	yr_built	yr_renovated
8692	1485000.0	1964	2015
18575	476000.0	1945	2015
4240	815000.0	1962	2015
7958	203000.0	1952	2015
2295	585000.0	1922	2015
19444	872500.0	1956	2015
16683	420000.0	1961	2015
13216	579000.0	1962	2015
3156	830000.0	1968	2015
7097	285000.0	1940	2015
5683	335000.0	1954	2015
11599	850000.0	1923	2015
11633	717000.0	1959	2015
7417	459000.0	1954	2015
15687	825000.0	1955	2015
14859	805000.0	1956	2015
3933	515000.0	1944	2014
5827	1697000.0	1970	2014
379	435000.0	1904	2014
8044	420000.0	1980	2014
12379	800000.0	1926	2014
18554	1115500.0	1919	2014
4919	620000.0	1962	2014
2097	688000.0	1913	2014
15046	905000.0	1900	2014
6136	543000.0	1962	2014
11132	324950.0	1959	2014
19174	209000.0	1912	2014
2850	399950.0	1930	2014
11727	1280000.0	1930	2014
...	...	...	...
7356	849000.0	1928	0
7355	315000.0	2005	0
7354	279900.0	1962	0
7353	304000.0	2002	0

	price	yr_built	yr_renovated
7352	560000.0	1985	0
7351	340000.0	1979	0
7350	775000.0	1990	0
7349	500000.0	1987	0
7348	554000.0	2004	0
7365	325000.0	2001	0
7366	1150000.0	1979	0
7367	406000.0	1962	0
7380	1070000.0	1939	0
7387	1775000.0	1985	0
7386	450000.0	1989	0
7385	450000.0	1990	0
7384	249950.0	1959	0
7383	585000.0	1947	0
7382	880000.0	1920	0
7381	460000.0	1978	0
7379	713000.0	2011	0
7368	650000.0	1990	0
7377	740000.0	1927	0
7376	295000.0	1924	0
7374	190000.0	1978	0
7373	510000.0	1984	0
7372	609000.0	1982	0
7371	395000.0	1907	0
7370	375000.0	1955	0
21612	325000.0	2008	0

21613 rows × 3 columns


SPICY
DATA