Data Wrangling and Manipulation in Pandas

Week 2 – Part 1 – Pandas Library

CS 457 - L1 Data Science

Zeehasham Rasheed

Lessons Intended and Learning Outcomes

On completion of this lesson, students are expected to be able to:

- Identify the Pandas data structures;
- Describe the essential functionality of Pandas;
- Manipulate data using the Pandas library; *and*
- Import and export data in text format.

Outline

- Introduction to Pandas Library and its Data Structures
- Handling Missing Data
- Data Merge and Combination
- Data Transformation
- Reading and Writing Data in Text Format

Pandas Library

- Pandas is a Python package providing fast, flexible, and expressive data structures designed to make working with "relational" or "labeled" data both easy and intuitive.
  - It aims to be the fundamental high-level building block for doing practical, real world data analysis in Python.
  - It has the broader goal of becoming the most powerful and flexible open source *data analysis / manipulation tool* available in any language.

Pandas Library (2)

- Pandas is well suited for many different kinds of data:
  - *Tabular data* with heterogeneously-typed columns, as in an SQL table or Excel spreadsheet
  - Ordered and unordered (not necessarily fixed-frequency) *time series data*.
  - Arbitrary *matrix data* (homogeneously typed or heterogeneous) with row and column labels
  - Any other form of *observational / statistical data sets*. The data actually need not be labeled at all to be placed into a pandas data structure.

Pandas Data Structures

- Series and Index Objects
- DataFrame

Series and DataFrame

- The two primary data structures of Pandas, *Series* (1-dimensional) and *DataFrame* (2-dimensional).
- They handle the vast majority of typical use cases in finance, statistics, social science, and many areas of engineering.
- Import conventions for Pandas:

```
from pandas import Series, DataFrame
```

```
import pandas as pd
```

Series

- A **`Series`** is a one-dimensional array-like object containing:
  - an array of data, and
  - an associated array of data labels.
- The simplest Series is formed from only an array of data:

```
>>> obj = Series([5, 8, -4, 2])
>>> obj
0     5
1     8
2    -4
3     2
dtype: int64
```

Series (2)

- Show `values` and `index` of a Series:

```
>>> obj.values
array([ 5,   8, -4,   2], dtype=int64)
>>> obj.index
RangeIndex(start=0, stop=4, step=1)
```

- Create a Series with an index:

```
>>> popul = Series([13.78, 3.24, 0.65, 1.27,
12.92],index=['China','USA','UK','Japan','India'])
>>> popul
China     13.78
USA        3.24
UK         0.65
Japan      1.27
India     12.92
dtype: float64
```

Series (3)

- Show `values` and `index` of a Series:

```
>>> popul['India']
12.92
>>> popul[['UK','USA']]
UK      0.65
USA     3.24
dtype: float64
>>> popul[popul<10]
USA       3.24
UK        0.65
Japan     1.27
dtype: float64
>>> 'France' in popul
False
```

Series (4)

- Create a Series from a dictionary:

```
>>> popul_dict = popul.to_dict()
>>> popul_dict
{'India': 12.92, 'Japan': 1.27, 'UK': 0.65000000000000002,
'China': 13.779999999999999, 'USA': 3.2400000000000002}
>>> countries =
['China','USA','UK','Japan','India','France']
>>> popul2 = Series(popul_dict, index=countries)
>>> popul2
China     13.78
USA        3.24
UK         0.65
Japan      1.27
India     12.92
France      NaN
dtype: float64
```

Series (5)

- Find missing data:

```
>>> popul2.isnull()
China       False
USA         False
UK          False
Japan       False
India       False
France       True
dtype: bool
```

```
>>> popul2.notnull()
China        True
USA          True
UK           True
Japan        True
India        True
France      False
dtype: bool
```

DataFrame

- A `DataFrame` represents a tabular, spreadsheet-like data structure.
- It contains an ordered collection of columns, each of which can be a different value type (numeric, string , boolean, etc.).
- It has both a row and column index.
- There are numerous ways to construct a `DataFrame`.

One of the most common is from a `dict` of equal- length lists or `NumPy` arrays.

DataFrame (2)

```
>>> data = {'state': ['Ohio', 'Ohio', 'Ohio',
'Nevada', 'Nevada'],'year': [2018, 2019, 2020,
2019, 2020],'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}
>>> frame = DataFrame(data)
>>> frame
   pop   state  year
1  1.5    Ohio  2018
2  1.7    Ohio  2019
3  3.6    Ohio  2020
4  2.4  Nevada  2019
5  2.9  Nevada  2020
```

Create a DataFrame from a dict

DataFrame (3)

```
>>> DataFrame(data, columns=['year', 'state',
'pop'])
   year   state  pop
0  2018    Ohio  1.5
1  2019    Ohio  1.7
2  2020    Ohio  3.6
3  2019  Nevada  2.4
4  2020  Nevada  2.9

>>> frame.columns
Index(['pop', 'state', 'year'], dtype='object')
```

Re-organize the output

DataFrame (4)

```
>>> frame.state
0        Ohio
1        Ohio
2        Ohio
3      Nevada
4      Nevada
Name: state, dtype: object
>>> frame.ix[2]
pop          3.6
state       Ohio
year        2002
Name: 2, dtype: object
```

Display a column of data

Display the third row of data

DataFrame (5)

– Update the data in a DataFrame:

```
>>> frame['year'] = 2020
>>> frame['pop'] = np.arange(1,6)
>>> frame
   pop   state  year
0    1    Ohio  2020
1    2    Ohio  2020
2    3    Ohio  2020
3    4  Nevada  2020
4    5  Nevada  2020
```

DataFrame (6)

- More operations on a DataFrame:

```
>>> frame['eastern'] = frame.state == 'Ohio'
>>> frame
   pop   state  year eastern
0    1    Ohio  2020    True
1    2    Ohio  2020    True
2    3    Ohio  2020    True
3    4  Nevada  2020   False
4    5  Nevada  2020   False

>>> del frame['eastern']
```

See what the output is?

Data types in python and pandas

- There are many data types in pandas
- Objects : “A”, “Hello”..
- Int64 : 1,3,5
- Float64 : 2.123, 632.31,0.12

Incorrect data types

- Sometimes the wrong data type is assigned to a feature.

```
df["price"].tail(5)

200     16845
201     19045
202     21485
203     22470
204     22625
Name: price, dtype: object
```

Correcting data types

To *identify* data types:

- Use **`dataframe.dtypes()`** to identify data type.

To *convert* data types:

- Use **`dataframe.astype()`** to convert data type.

Example: convert data type to integer in column “price”

```
df["price"] = df["price"].astype("int")
```

End of Part 1

Data Wrangling and Manipulation in Pandas

Week 2 – Part 2 – Essential Operations in Pandas

CS 457 - L1 Data Science

Zeehasham Rasheed

Some Operations

- Reindexing
- Dropping Entries
- Selecting Entries
- Data Alignment
- Rank and Sort

Reindexing

- How can we *update* the index object?

```
>>> ser1 = Series([1,2,3,4],index=['A','B','C','D'])
>>> ser1
A     1
B     2
C     3
D     4
dtype: int64
>>> ser2 = ser1.reindex(['A','B','C','D','E','F'])
>>> ser2
A     1.0
B     2.0
C     3.0
D     4.0
E     NaN
F     NaN
dtype: float64
```

Not a Number

Reindexing (2)

- Fill in values for *new indexes*:

```
>>> ser2.reindex(['A','B','C','D','E','F','G'],
fill_value=0)
A     1.0
B     2.0
C     3.0
D     4.0
E     NaN
F     NaN
G     0.0
dtype: float64
```

Reindexing (3)

```
>>> ser3 = Series(['HK','Tokyo','Seoul'],index=[0,3,6])
>>> ser3
0        HK
3     Tokyo
6     Seoul
dtype: object
>>> ser3.reindex(range(9),method='ffill')
0        HK
1        HK
2        HK
3     Tokyo
4     Tokyo
5     Tokyo
6     Seoul
7     Seoul
8     Seoul
dtype: object
```

Fill values forward, OR fill values backward ('bfill')

Reindexing (4)

- Reindexing row, columns or both:

```
>>> dframe = DataFrame(np.arange(9).reshape((3,3)),
index=['A','B','C'], columns=['Apple', 'Orange', 'Mango'])
>>> dframe
   Apple  Orange  Mango
A      0       1      2
B      3       4      5
C      6       7      8
>>> dframe2 = dframe.reindex(['A','B','C','D','E'])
>>> dframe2
   Apple  Orange  Mango
A    0.0     1.0    2.0
B    3.0     4.0    5.0
C    6.0     7.0    8.0
D    NaN     NaN    NaN
E    NaN     NaN    NaN
```

Reindexing (5)

```
>>> new_columns = ['Apple', 'Orange', 'Mango',
'Grape']
>>> dframe2.reindex(columns=new_columns)
   Apple  Orange  Mango  Grape
A    0.0     1.0    2.0    NaN
B    3.0     4.0    5.0    NaN
C    6.0     7.0    8.0    NaN
D    NaN     NaN    NaN    NaN
E    NaN     NaN    NaN    NaN
```

Dropping Entries

- Dropping a row from a Series:

```
>>> ser1 =
Series(np.arange(3),index=['a','b','c'])
>>> ser1
a     0
b     1
c     2
dtype: int32
>>> ser1.drop('b')
a     0
c     2
dtype: int32
```

Dropping Entries (2)

- With a DataFrame we can drop values from either axis:

```
>>> dframe1 =
DataFrame(np.arange(9).reshape((3,3)),index=['SF','LA','NY'],columns=['pop',
'size','year'])
>>> dframe1
    pop  size  year
SF    0     1     2
LA    3     4     5
NY    6     7     8
>>> dframe1.drop('LA')
    pop  size  year
SF    0     1     2
NY    6     7     8
>>> dframe1.drop('year',axis=1)
    pop  size
SF    0     1
LA    3     4
NY    6     7
```

Selecting Entries

- Let's try selection in a *Series*:

```
>>> ser1 = Series(np.arange(3),index=['A','B','C'])
>>> ser1 = ser1 * 2
>>> ser1
A     0
B     2
C     4
dtype: int32
>>> ser1['B']
2
>>> ser1[1]
2
>>> ser1[0:4]
A     0
B     2
C     4
dtype: int32
```

Selecting Entries (2)

```
>>> ser1[['A','C']]
A     0
C     4
dtype: int32
>>> ser1[ser1>2]
C     4
dtype: int32
>>> ser1[ser1>3] = 7
>>> ser1
A     0
B     2
C     7
dtype: int32
```

Selecting Entries (3)

– Let's try selection in a *DataFrame*:

```
>>> dframe =
DataFrame(np.arange(16).reshape(4,4),index=['NJ','LA','SF','DC
'], columns=['A','B','C','D'])
>>> dframe
      A    B    C    D
NJ    0    1    2    3
LA    4    5    6    7
SF    8    9   10   11
DC   12   13   14   15
>>> dframe['C']
NJ     2
LA     6
SF    10
DC    14
Name: C, dtype: int32
```

Selecting Entries (4)

```
>>> dframe[['A','D']]
     A   D
NJ   0   3
LA   4   7
SF   8  11
DC  12  15
>>> dframe[dframe['B']>5]
     A   B   C   D
SF   8   9  10  11
DC  12  13  14  15
```

Selecting Entries (5)

```
>>> dframe > 6
        A       B       C       D
NJ  False   False   False   False
LA  False   False   False    True
SF   True    True    True    True
DC   True    True    True    True
>>> dframe.ix['DC']
A    12
B    13
C    14
D    15
Name: DC, dtype: int32
```

Data Alignment

- Let's learn about arithmetic between *Series*:

```
>>> ser1 = Series([0,1,2],index=['A','B','C'])
>>> ser1
A     0
B     1
C     2
dtype: int64
>>> ser2 = Series([3,4,5,6],index=['A','B','C','D'])
>>> ser2
A     3
B     4
C     5
D     6
dtype: int64
```

```
>>> ser1 + ser2
A     3.0
B     5.0
C     7.0
D     NaN
dtype: float64
```

Data Alignment (2)

- Arithmetic between *DataFrames*:

```
>>> dframe1 =
DataFrame(np.arange(4).reshape(2,2),columns=list('AB'),index=[
'NJ','LA'])
>>> dframe1
    A   B
NJ  0   1
LA  2   3
>>> dframe2 = DataFrame(np.arange(9).reshape(3,3),
columns=list('ADC'),
index=['NJ','SF','LA'])
>>> dframe2
    A   D   C
NJ  0   1   2
SF  3   4   5
LA  6   7   8
```

```
>>> dframe1 + dframe2
      A    B    C    D
LA  8.0  NaN  NaN  NaN
NJ  0.0  NaN  NaN  NaN
SF  NaN  NaN  NaN  NaN
```

Data Alignment (3)

- Use `.add()`:

```
>>> dframe1.add(dframe2,fill_value=0)
      A    B    C    D
LA   8.0  3.0  8.0  7.0
NJ   0.0  1.0  2.0  1.0
SF   3.0  NaN  5.0  4.0
```

Other arithmetic methods include: `sub`, `div`, `mul`

– More operations:

```
>>> dframe2
    A  D  C
NJ  0  1  2
SF  3  4  5
LA  6  7  8
```

Data Alignment (4)

```
>>> ser3 = dframe2.ix[0]
>>> ser3
A     0
D     1
C     2
Name: NJ, dtype: int32
>>> dframe2 - ser3
    A  D  C
NJ  0  0  0
SF  3  3  3
LA  6  6  6
```

Rank and Sort

- Re-order data using `sort_index()/sort_values()`:

```
>>> ser1 = Series(range(3),index=['C','A','B'])
>>> ser1
C      0
A      1
B      2
dtype: int32
>>> ser1.sort_index()
A      1
B      2
C      0
dtype: int32
```

```
>>>
ser1.sort_values()
C      0
A      1
B      2
dtype: int32
```

Rank and Sort (2)

- Let’s see how ranking works:

```
>>> from numpy.random import randn
>>> ser2 = Series(randn(5))
>>> ser2
0    1.029665
1    0.705042
2   -0.761126
3   -1.767447
4    1.175974
dtype: float64
>>> ser2.rank()
0    4.0
1    3.0
2    2.0
3    1.0
4    5.0
dtype: float64
```

Rank and Sort (3)

```
>>> ser2.sort_values()
3   -1.767447
2   -0.761126
1    0.705042
0    1.029665
4    1.175974
dtype: float64
>>> ser2.rank()
3    1.0
2    2.0
1    3.0
0    4.0
4    5.0
dtype: float64
```

End of Part 2

Data Wrangling and Manipulation in Pandas

Week 2 – Part 3 – Handling Missing Data

CS 457 - L1 Data Science

Zeehasham Rasheed

Missing Data Operations

- Missing Data
- Filtering Out Missing Data
- Filling In Missing Data

Missing Data

- Missing data is common in most data analysis applications.
- One of the goals in designing pandas was to make working with missing data as painless as possible.
- Note all of the descriptive statistics on pandas exclude missing data.
- pandas uses the floating point value **`NaN`** *(Not a Number) to represent missing data* in both floating as well as non-floating point arrays.
- The built-in Python **`None`** value is also treated as **`NA`** *(Not Available)* in object arrays.

Missing Data (2)

- Find the missing values:

```
>>> data = Series(['one','two',np.nan,'four'])
>>> data
0      one
1     two
2     NaN
3     four
dtype: object
>>> data.isnull()
0     False
1     False
2     True
3     False
dtype: bool
```

Filtering Out Missing Data

- There are a number of options for filtering out missing data.

```
>>> from numpy import nan as NA
>>> data = Series([1, NA, 3.5, NA, 7])
>>> data
0     1.0
1     NaN
2     3.5
3     NaN
4     7.0
dtype: float64
>>> data.dropna()
0     1.0
2     3.5
4     7.0
dtype: float64
```

```
>>>
data[data.notnull()]
0     1.0
2     3.5
4     7.0
dtype: float64
```

Filtering Out Missing Data (2)

- Filtering out missing data in a DataFrame:

```
>>> dframe = DataFrame([[1,2,3],[NA,5,6],[7,NA,9],
[NA,NA,NA]])
>>> dframe
     0    1    2
0  1.0  2.0  3.0
1  NaN  5.0  6.0
2  7.0  NaN  9.0
3  NaN  NaN  NaN
>>> clean_dframe = dframe.dropna()
>>> clean_dframe
     0    1    2
0  1.0  2.0  3.0
```

Filtering Out Missing Data (3)

```
>>> dframe.dropna(how='all')
     0    1    2
0  1.0  2.0  3.0
1  NaN  5.0  6.0
2  7.0  NaN  9.0
```

~~3 NaN NaN NaN~~

Drop rows that are complete missing all data.

```
>>> dframe.dropna(axis=1)
Empty DataFrame
Columns: []
Index: [0, 1, 2, 3]
```

Drop columns with a missing data.

Filling In Missing Data

- Rather than filtering out missing data, you may want to fill in the "holes" in any number of ways.
- We may use **`fillna`** method to perform this task:

```
>>> dframe
     0    1    2
0  1.0  2.0  3.0
1  NaN  5.0  6.0
2  7.0  NaN  9.0
3  NaN  NaN  NaN
```

```
>>> dframe.fillna(0)
     0    1    2
0  1.0  2.0  3.0
1  0.0  5.0  6.0
2  7.0  0.0  9.0
3  0.0  0.0  0.0
```

Filling In Missing Data (2)

- We may use fillna method to perform this task:

```
>>> dframe.mean()
0     4.0
1     3.5
2     6.0
dtype: float64
>>> dframe.fillna(dframe.mean())
     0    1    2
0  1.0  2.0  3.0
1  4.0  5.0  6.0
2  7.0  3.5  9.0
3  4.0  3.5  6.0
```

Find the mean of each column.

End of Part 3

Data Wrangling and Manipulation in Pandas

Week 2 – Part 4 – Data Merging

CS 457 - L1 Data Science

Zeehasham Rasheed

Introduction to Data Wrangling

- Much of the programming work in data analysis and modeling is spent on data preparation.
- ***Data preparation*** includes loading, cleaning, transforming, and rearranging of data.
- ***Data wrangling*** (or *data munging*) is loosely the process of manually converting or mapping data *from one "raw" form into another format* that allows for more convenient consumption of the data with the help of semi-automated tools.
- *pandas* along with the Python standard library provide us with a high-level, flexible, and high-performance set of core manipulations and algorithms to enable you to wrangle data into the right form without much trouble.

Introduction to Data Wrangling (2)

- You need to include the following import statements before running the Python scripts presented on this lesson:

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

Merging Data Sets

– Data contained in pandas objects can be combined together in a number of built-in ways:

1. `pandas.merge` connects rows in DataFrames based on or more keys. It implements database *join* operations.
2. `pandas.concat` glues or stacks together objects along an axis.
3. `combine_first` instance method enables splicing together overlapping data to fill in missing values in one object with values from another.

– Merge or join operations combine data sets by linking rows using one or more keys.

– The merge function in pandas is the main entry point for using these algorithms on your data.

Merging Data Sets (2)

- Let’s start with a simple example:

```
>>> df1 = DataFrame({'key': ['Y', 'Y', 'X', 'Z', 'X', 'X',
'Y'], 'data1': range(7)})
>>> df2 = DataFrame({'key': ['X', 'Y', 'W'],
'data2': range(3)})
>>> df1
   data1 key
0      0   Y
1      1   Y
2      2   X
3      3   Z
4      4   X
5      5   X
6      6   Y
```

```
>>> df2
   data2 key
0      0   X
1      1   Y
2      2   W
```

Merging Data Sets (3)

– Here is an example:

```
>>> pd.merge(df1, df2)
     data1  key  data2
0        0    Y      1
1        1    Y      1
2        6    Y      1
3        2    X      0
4        4    X      0
5        5    X      0
```

***merge**, by default, uses the overlapping column names as the keys.*

Merging Data Sets (4)

- We may specify explicitly the key:

```
>>> pd.merge(df1, df2, on='key')
     data1 key  data2
0        0   Y      1
1        1   Y      1
2        6   Y      1
3        2   X      0
4        4   X      0
5        5   X      0
```

Merging Data Sets (5)

- We can choose which DataFrame's keys to use:

```
>>> pd.merge(df1, df2, how='left')
   data1 key  data2
0      1   Y    1.0
1      2   Y    1.0
2      3   X    0.0
3      4   Z    NaN
4      5   X    0.0
5      6   X    0.0
6      7   Y    1.0
>>> pd.merge(df1, df2, how='right')
   data1 key  data2
0    0.0   Y      1
1    1.0   Y      1
2    6.0   Y      1
3    2.0   X      0
4    4.0   X      0
5    5.0   X      0
6    NaN   W      2
```

Merging Data Sets (6)

– Choosing the `outer` method selects the ***union*** of both keys:

```
>>> pd.merge(df1, df2, on='key', how='outer')
    data1  key   data2
0     0.0    Y     1.0
1     1.0    Y     1.0
2     6.0    Y     1.0
3     2.0    X     0.0
4     4.0    X     0.0
5     5.0    X     0.0
6     3.0    Z     NaN
7     NaN    W     2.0
```

'inner' is the default value

Merging Data Sets (7)

- We can specify the column names separately if they are different in each object:

```
>>> df3 = DataFrame({'lkey': ['Y', 'Y', 'X', 'Z', 'X',
'X', 'Y'], 'data1': range(7)})
>>> df4 = DataFrame({'rkey': ['X', 'Y', 'W'],
'data2': range(3)})
>>> pd.merge(df3, df4, left_on='lkey', right_on='rkey')
   data1 lkey  data2 rkey
0      0    Y      1    Y
1      1    Y      1    Y
2      6    Y      1    Y
3      2    X      0    X
4      4    X      0    X
5      5    X      0    X
```

Merging Data Sets (9)

– To merge with ***multiple keys***, pass a list of column names:

```
>>> left = DataFrame({'key1': ['foo', 'foo', 'bar'],
'key2': ['one', 'two', 'one'], 'ldata': [1, 2, 3]})
>>> right = DataFrame({'key1': ['foo', 'foo', 'bar',
'bar'], 'key2': ['one', 'one', 'one', 'two'],
'rdata': [4, 5, 6, 7]})
>>> pd.merge(left, right, on=['key1', 'key2'], how='outer')
  key1 key2  ldata  rdata
1  foo  one    1.0    4.0
2  foo  one    1.0    5.0
3  foo  two    2.0    NaN
4  bar  one    3.0    6.0
5  bar  two    NaN    7.0
```

Merging on Index

- The merge key or keys in a DataFrame may be found in its index.
- In this case, we can pass `left_index=True` or `right_index=True` (*or both*) to indicate that the index should be used as the merge key:

```
>>> left1 = DataFrame({'key': ['X', 'Y', 'X', 'X', 'Y', 'Z'], 'value':
range(6)})
>>> right1 = DataFrame({'group_val': [3, 6]}, index=
['X', 'Y'])
>>> pd.merge(left1, right1, left_on='key', right_index=True)
  key  value  group_val
0  X     0        3
2  X     2        3
3  X     3        3
1  Y     1        6
4  Y     4        6
```

Merging on Index (2)

- We indicate multiple columns to merge on as a list:

```
>>> pd.merge(left_df, right_df, left_on=['key1', 'key2'],
right_index=True)
```

```
    data  key1  key2   event1  event2
0    0.0   TST  2014        4       5
0    0.0   TST  2014        6       7
1    1.0   TST  2015        8       9
2    2.0   TST  2016       10      11
```

Merging on Index (3)

- We can also use the indices of both sides of the merge:

```
>>> left2 = DataFrame([[1, 2], [3, 4], [5, 6]],
index=['A', 'C', 'E'], columns=['TST', 'MK'])
>>> right2 = DataFrame([[7, 8], [9, 10], [11, 12], [13,
14]], index=['B', 'C', 'D', 'E'], columns=['TW', 'KT'])
>>> pd.merge(left2, right2, how='outer',
left_index=True, right_index=True)
   TST   MK    TW    KT
A  1.0  2.0   NaN   NaN
B  NaN  NaN   7.0   8.0
C  3.0  4.0   9.0  10.0
D  NaN  NaN  11.0  12.0
E  5.0  6.0  13.0  14.0
```

Merging on Index (4)

- We can use `join` for merging by index:

```
>>> left2.join(right2, how='outer')
   TST   MK    TW    KT
A. 1.0   2.0   NaN   NaN
B. NaN   NaN   7.0   8.0
C  3.0   4.0   9.0   10.0
D  NaN   NaN   11.0  12.0
E  5.0   6.0   13.0  14.0
>>> left1.join(right1, on='key')
  key  value  group_val
 0  X      0        3.0
 1  Y      1        6.0
 2  X      2        3.0
 3  X      3        3.0
 4  Y      4        6.0
5   Z     5         NaN
```

Concatenating Along an Axis

- *Concatenation* (or *binding*, *stacking*) is another kind of data combination.
- *NumPy* has a concatenate function for doing this with raw *NumPy* arrays:

```
>>> arr = np.arange(12).reshape((3, 4))
>>> arr
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> np.concatenate([arr, arr])
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

Concatenating Along an Axis (2)

– Let's start with a simple example:

```
>>> np.concatenate([arr, arr], axis=1)
array([[ 0,  1,  2,  3,  0,  1,  2,  3],
       [ 4,  5,  6,  7,  4,  5,  6,  7],
       [ 8,  9, 10, 11,  8,  9, 10, 11]])
```

By default, *axis = 0*

– Concatenate two Series with no overlap:

```
>>> ser1 = Series([0,1,2], index=['T','U','V'])
>>> ser2 = Series([3,4], index=['X','Y'])
>>> pd.concat([ser1,ser2])
T    0
U    1
V    2
X    3
Y    4
dtype: int64
```

Concatenating Along an Axis (3)

- Passing along another axis will produce a *DataFrame*:

```
>>> pd.concat([ser1,ser2], axis=1)
     0    1
T  0.0  NaN
U  1.0  NaN
V  2.0  NaN
X  NaN  3.0
Y  NaN  4.0
```

- We can specify which specific axes to be used:

```
>>> pd.concat([ser1,ser2], axis=1, join_axes=[['U','V','Y']])
     0    1
U  1.0  NaN
V  2.0  NaN
Y  NaN  4.0
```

Concatenating Along an Axis (4)

– Concatenation works similarly in *DataFrames*:

```
>>> dframe1 = DataFrame(np.random.randn(4, 3),
columns=['X', 'Y', 'Z'])
>>> dframe2 = DataFrame(np.random.randn(3, 3),
columns=['Y', 'Q', 'X'])
>>> pd.concat([dframe1,dframe2])
          Q         X         Y         Z
0       NaN -0.950978  1.729998  0.721512
1       NaN -0.203453 -0.834730 -0.877719
2       NaN  0.226450  1.515619 -1.278597
3       NaN  1.460541 -0.179448 -0.728863
0 -0.975134 -1.309284 -0.644482       NaN
1  1.346980  1.458585 -0.497242       NaN
2  0.126452  1.501683  0.285019       NaN
```

End of Part 4

Data Wrangling and Manipulation in Pandas

Week 2 – Part 5 – Data Transformation

CS 457 - L1 Data Science

Zeehasham Rasheed

- Removing Duplicates
- Data Cleaning
- Data Mapping
- Replacing Values

Removing Duplicates

- So far in this lesson we’ve concerned with rearranging data.
- Filtering, cleaning, and other transformations are another class of important operations for data wrangling.
- *Duplicate rows* may be found in a DataFrame for any number of reasons.
- Consider the example:

```
>>> df = DataFrame({'key1': ['A'] * 2 + ['B'] * 3,
'key2': [2, 2, 2, 3, 3]})
>>> df
  key1  key2
0    A     2
1    A     2
2    B     2
3    B     3
4    B     3
```

Removing Duplicates (2)

- Method **`duplicated`** returns a boolean Series indicating whether each row is a duplicate or not:

```
>>> df.duplicated()
0     False
1      True
2     False
3     False
4      True
dtype: bool
```

- We can drop duplicates like this:

```
>>> df.drop_duplicates()
  key1  key2
0    A     2
2    B     2
3    B     3
```

Data Mapping

- For many data sets, we may want to perform some transformation based on the values in an array, Series, or column in a DataFrame.
- Consider the DataFrame:

```
>>> df = DataFrame({'city':['Hong Kong',
'London','New York'],'population':
[7.25,8.54,8.55]})
>>> df
        city  population
1  Hong Kong        7.25
2     London        8.54
3   New York        8.55
```

Data Mapping (2)

- Suppose we want to add a column indicate the belonging country of each city:

```
>>> country_map = {'Hong Kong':'China','New York':'USA','London':'England'}
>>> df['country'] = df['city'].map(country_map)
>>> df
        city  population  country
1  Hong Kong        7.25    China
2     London        8.54  England
3   New York        8.55      USA
```

- We could also ***pass a function*** to the `map` method.

Replacing Values

- *Filling in missing data* with the `fillna` method nay be thought of as a special case of more general value replacement.

```
>>> ser1 = Series([11,22,33,44,11,22,33,44])
>>> ser1.replace(11, np.nan)
0     NaN
1    22.0
2    33.0
3    44.0
4     NaN
5    22.0
6    33.0
7    44.0
```

Replacing Values (2)

- Some variations of `replace` operations:

```
>>> ser1.replace([11,44],[100,400])
0     100
1      22
2      33
3     400
4     100
5      22
6      33
7     400
dtype: int64
```

```
>>> ser1.replace({22:200,
33:300})
0      11
1     200
2     300
3      44
4      11
5     200
6     300
7      44
dtype: int64
```

Data Inconsistency

- Ideally we would like to see all the sales for Hy-Vee, Costco, Sam's, etc grouped together.
- Lets assume this data is stored in pandas dataframe variable **df**

	Store Name	Sale (Dollars)	percent
0	Central City 2	11,877,164	3.40%
1	Hy-Vee #3 / BDI / Des Moines	11,275,152	3.23%
2	Hy-Vee Wine and Spirits / Iowa City	5,001,156	1.43%
3	Wilkie Liquors	3,639,515	1.04%
4	Lot-A-Spirits	3,504,665	1.00%
5	Costco Wholesale #788 / WDM	3,178,079	0.91%
6	Sam's Club 8162 / Cedar Rapids	3,147,579	0.90%
7	Benz Distributing	3,082,936	0.88%
8	Hy-Vee Food Store / Urbandale	3,073,798	0.88%
9	Sam's Club 6344 / Windsor Heights	2,963,108	0.85%

Data Inconsistency (2)

- This code will search for the string ‘Hy-Vee’ using a case insensitive search and store the value “Hy-Vee” in a new column called Store_Group_1 .
- This code will effectively convert names like “Hy-Vee #3 / BDI / Des Moines” or “Hy-Vee Food Store / Urbandale” into a common “Hy-Vee”.

```
df.loc[df['Store Name'].str.contains('Hy-Vee', case=False), 'Store Name'] = 'Hy-Vee'
```

Replacing Value

- Need to replace 'ABC' and 'AB' in column BrandName by A

BrandName	Specialty
A	H
B	I
ABC	J
D	K
AB	L

Replacing Value (2)

The easiest way is to use the replace method on the column. The arguments are a list of the things you want to replace (here ['ABC', 'AB']) and what you want to replace them with (the string 'A' in this case):

```
>>> df['BrandName'].replace(['ABC', 'AB'], 'A')
0    A
1    B
2    A
3    D
4    A
```

Cleaning Strings in Pandas

```
# Loading a Sample Pandas DataFrame
import pandas as pd
df = pd.DataFrame.from_dict({
'Name': ['Tranter, Melvyn', 'Lana, Courtney', 'Abel, Shakti', 'Vasu, Imogene', 'Aravind,
Shelly'],
'Region': ['Region A', 'Region A', 'Region B', 'Region C', 'Region D'], 'Location':
['TORONTO', 'LONDON', 'New york', 'ATLANTA', 'toronto'], 'Favorite Color': [' green ', 'red',
' yellow', 'blue', 'purple ']
})
print(df)
# Returns:
# Name Region Location Favorite Color
# 0 Tranter, Melvyn    Region A       TORONTO  green
# 1 Lana, Courtney     Region A       LONDON    red
# 2 Abel, Shakti       Region B       New york yellow
# 3 Vasu, Imogene      Region C       ATLANTA    blue
# 4 Aravind, Shelly    Region D       toronto   purple
```

Trimming Whitespaces

```
# Trimming Whitespace from a Pandas Column

df['Favorite Color'] = df['Favorite Color'].str.strip()

print(df)

# Returns:
# Name Region Location Favorite Color
# 0 Tranter, Melvyn     Region A TORONTO    green
# 1 Lana, Courtney      Region A LONDON       red
# 2 Abel, Shakti        Region B New york  yellow
# 3 Vasu, Imogene       Region C ATLANTA     blue
# 4 Aravind, Shelly     Region D toronto   purple
```

Splitting Column

```
# Splitting a Column into Two Columns

df[['Last Name', 'First Name']] = df['Name'].str.split(',', expand=True)
print(df)

# Returns:
# Name Region Location Favorite Color Last Name First Name
# 0 Tranter, Melvyn     Region A TORONTO  green        Tranter      Melvyn
# 1 Lana, Courtney      Region A LONDON   red          Lana         Courtney
# 2 Abel, Shakti        Region B New york yellow       Abel         Shakti
# 3 Vasu, Imogene       Region C ATLANTA  blue         Vasu         Imogene
# 4 Aravind, Shelly     Region D toronto  purple       Aravind      Shelly
```

Replace Text in Column

```
# Replacing a Substring in Pandas

df['Region'] = df['Region'].str.replace('Region ', '')
print(df)

# Returns:
# Name Region Location Favorite Color
# 0 Tranter, Melvyn      A TORONTO   green
# 1 Lana, Courtney       A LONDON    red
# 2 Abel, Shakti         B New york  yellow
# 3 Vasu, Imogene        C ATLANTA   blue
# 4 Aravind, Shelly      D toronto   purple
```

Changing String Case

Pandas provides access to a number of methods that allow us to change cases of strings:

- `.upper()` will convert a string to all upper case
- `.lower()` will convert a string to all lower case
- `.title()` will convert a string to title case

we want our locations to be in title case, so we can apply to `.str.title()` method to the string:

```
# Changing Text to Title Case in Pandas
df['Location'] = df['Location'].str.title()
print(df)
# Returns:
# Name Region Location Favorite Color
# 0 Tranter, Melvyn    Region A Toronto      green
# 1 Lana, Courtney     Region A London       red
# 2 Abel, Shakti       Region B New York     yellow
# 3 Vasu, Imogene      Region C Atlanta      blue
# 4 Aravind, Shelly    Region D Toronto      purple
```

Create Conditional Column

- There are many times when you may need to set a Pandas column value based on the condition of another column.

```
import pandas as pd

df = pd.DataFrame.from_dict( { 'Name': ['Jane', 'Melissa', 'John',
'Matt'], 'Age': [23, 45, 35, 64], 'Birth City': ['London', 'Paris',
'Toronto', 'Atlanta'], 'Gender': ['F', 'F', 'M', 'M'] } )

print(df)
```

```
        Name Age Birth City Gender
 0      Jane 23 London F
 1      Melissa 45 Paris F
 2      John 35 Toronto M
 3      Matt 64 Atlanta M
```

Create Conditional Column (2)

- Using Pandas loc to Set Pandas Conditional Column
  - We assigned the string 'Over 30' to every record in the new column name “Age Category”
  - We then use .loc to create a boolean mask on the Age column to filter down to rows where the age is less than 30. When this condition is met, the Age Category column is assigned the new value 'Under 30'

```
df['Age Category'] = ‘Over 30’
df.loc[df['Age'] < 30, 'Age Category'] = 'Under 30'
```

```
df['Age Category'] = 'Over 30'
print(df)
```

```
      Name  Age Birth City Gender Age Category
0     Jane   23     London      F      Over 30
1  Melissa   45      Paris      F      Over 30
2     John   35    Toronto      M      Over 30
3     Matt   64    Atlanta      M      Over 30
```

```
df.loc[df['Age'] < 30, 'Age Category'] = 'Under 30'
print(df)
```

```
      Name  Age Birth City Gender Age Category
0     Jane   23     London      F     Under 30
1  Melissa   45      Paris      F      Over 30
2     John   35    Toronto      M      Over 30
3     Matt   64    Atlanta      M      Over 30
```

Applying Python Built-in Functions

- We can easily apply a built-in function using the **.apply()** method.
- Let's see how we can use the **len()** function to count how long a string of a given column.

```
df['Name Length'] = df['Name'].apply(len)
print(df)
```

```
        Name    Age Birth City Gender Name Length
0       Jane    23     London        F        4
1       Melissa 45     Paris         F        7
2       John    35     Toronto       M        4
3       Matt    64     Atlanta       M        4
```

End of Part 5

Data Wrangling and Manipulation in Pandas

Week 2 – Part 6 – Import Export Data

CS 457 - L1 Data Science

Zeehasham Rasheed

Reading and Writing Data

- Importing and Exporting Data
- Parsing Functions in Pandas
- Reading Text in Pieces
- Writing Data Out to Text Format

Parsing Functions in Pandas

- Python has become a beloved language for text and file munging.
- This is due to Python's simple syntax for interacting with files, intuitive data structures, and convenient features like tuple packing and unpacking.
- `Pandas` features a number of functions for reading tabular data as a `DataFrame` object.
- Next slide will show a summary of these functions.

Parsing Functions in Pandas (2)

Function	Description
`read_csv`	Load delimited data from a file, URL, or file-like object. Use comma as default delimiter.
`read_table`	Load delimited data from a file, URL, or file-like object. Use tab (**`'\t'`**) as default delimiter.
`read_fwf`	Read data in fixed-width column format (that is, no delimiters).
`read_clipboard`	Version of `read_table` that reads data from the clipboard. Useful for converting tables from webpages.

Here is a summary of parsing functions:

Parsing Functions in Pandas (3)

- Let's start with a small comma-separated (CSV) text file:

```
>>> import numpy as np
>>> from pandas import Series, DataFrame
>>> import pandas as pd
>>> dframe = pd.read_csv('lec0601.csv')
>>> dframe
   q  r  s  t   apple
0  2  3  4  5    pear
1  a  s  d  f  rabbit
2  5  2  5  7     dog
```

A	B	C	D	E
q	r	s	t	apple
2	3	4	4	pear
a	s	d	f	rabbit
5	2	5	7	dog

Parsing Functions in Pandas (4)

- We can also use **`read_table`** with ',' as a delimiter:

```
>>> dframe = pd.read_table('lec0601.csv', sep=',')
>>> dframe
   q  r  s  t   apple
0  2  3  4  5    pear
1  a  s  d  f  rabbit
2  5  2  5  7     dog
>>> dframe = pd.read_table('lec0601.csv')
>>> dframe
     q,r,s,t,apple
0     2,3,4,5,pear
1   a,s,d,f,rabbit
2      5,2,5,7,dog
```

Parsing Functions in Pandas (5)

- More examples on **`read_csv`**:

```
>>> dframe = pd.read_csv('lec0601.csv', header=None)
>>> dframe
   0  1  2  3       4
0  q  r  s  t   apple
1  2  3  4  5    pear
2  a  s  d  f  rabbit
3  5  2  5  7     dog
>>> dframe = pd.read_csv('lec0601.csv',
names=['A','B','C','D','Message'])
>>> dframe
   A  B  C  D Message
0  q  r  s  t   apple
1  2  3  4  5    pear
2  a  s  d  f  rabbit
3  5  2  5  7     dog
```

Reading Text in Pieces

- When processing very large files, we may only want to read in a small piece of a file:

```
>>> result = pd.read_csv('lec0602.csv')
>>> result
           one       two     three      four key
0     0.467976 -0.038649 -0.295344 -1.824726   L
1    -0.358893  1.404453  0.704965 -0.200638   B
2    -0.501840  0.659254 -0.421691 -0.057688   G
3     0.204886  1.074134  1.388361 -0.982404   R
...        ...       ...       ...       ...  ..
9997  0.523331  0.787112  0.486066  1.093156   K
9998 -0.362559  0.598894 -1.843201  0.887292   G
9999 -0.096376 -1.012999 -0.657431 -0.573315   0

[10000 rows x 5 columns]
```

Reading Text in Pieces (2)

– We may want to only read out a small number of rows:

```
>>> pd.read_csv('lec0602.csv', nrows=5)

        one       two     three      four key
0  0.467976 -0.038649 -0.295344 -1.824726   L
1 -0.358893  1.404453  0.704965 -0.200638   B
2 -0.501840  0.659254 -0.421691 -0.057688   G
3  0.204886  1.074134  1.388361 -0.982404   R
4  0.354628 -0.133116  0.283763 -0.837063   Q
```

Writing Data Out to Text Format

- Data can also be exported to delimiter format.
- Let's consider one of the CSV files:

```
>>> data = pd.read_csv('lec0603.csv')
>>> data
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       two  5   6   NaN   8   world
2     three  9  10  11.0  12     foo
```

- We use **`DataFrame`**'s **`to_csv`** method to write the data out to a comma-separated file:

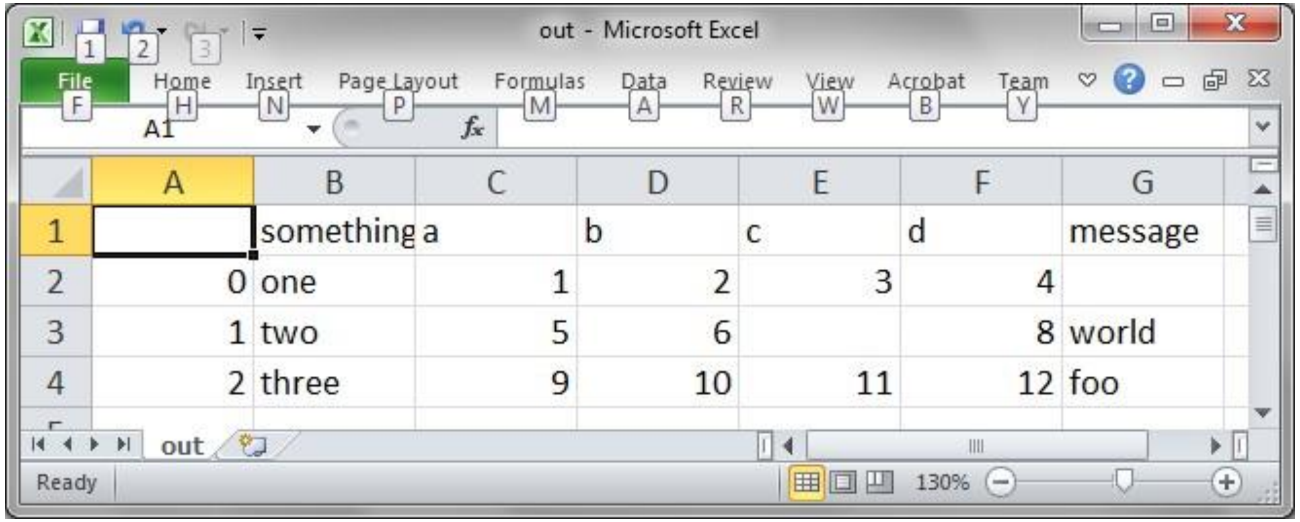
```
>>> data.to_csv('out.csv')
```

Check the content of the file

Writing Data Out to Text Format (2)

– Other delimiter can be used:

```
>>> import sys
>>> data.to_csv(sys.stdout, sep='|')
|something|a|b|c|d|message
0|one|1|2|3.0|4|
1|two|5|6||8|world
2|three|9|10|11.0|12|foo
```



Writing Data Out to Text Format (3)

```
>>> data.to_csv(sys.stdout, na_rep='NULL')
,something,a,b,c,d,message
0,one,1,2,3.0,4,NULL
1,two,5,6,NULL,8,world
2,three,9,10,11.0,12,foo
>>> data.to_csv(sys.stdout, index=False,
header=False)
one,1,2,3.0,4,
two,5,6,,8,world
three,9,10,11.0,12,foo
```

Self Study Guide

Online Resources

- http://pandas.pydata.org/pandas-docs/stable/api. html
- http://docs.scipy.org/doc/numpy/reference/index. html

End of Part 6