Pandas

Data Manipulation in Python

Pandas

- Built on NumPy
- Adds data structures and data manipulation tools
- Enables easier data cleaning and analysis

```
import pandas as pd
```

Pandas Fundamentals

Three fundamental Pandas data structures:

- `Series` - a one-dimensional array of values indexed by a `pd.Index`
- `Index` - an array-like object used to access elements of a `Series` or `DataFrame`
- `DataFrame` - a two-dimensional array with flexible row indices and column names

Series from List

```
In [4]: data = pd.Series(['a','b','c','d'])

In [5]: data
Out[5]:
0    a
1    b
2    c
3    d
dtype: object
```

The 0..3 in the left column are the `pd.Index` for `data`:

```
In [7]: data.index
Out[7]: RangeIndex(start=0, stop=4, step=1)
```

The elements from the Python list we passed to the `pd.Series` constructor make up the values:

```
In [8]: data.values
Out[8]: array(['a', 'b', 'c', 'd'], dtype=object)
```

Notice that the values are stored in a Numpy array.

Series from Sequence

You can construct a list from any definite sequence:

```
In [24]: pd.Series(np.loadtxt('exam1grades.txt'))
Out[24]:
0       72.0
1       72.0
2       50.0
...
134     87.0
dtype: float64
```

or

```
In [25]: pd.Series(open('exam1grades.txt').readlines())
Out[25]:
0       72\n
1       72\n
2       50\n
...
134     87\n
dtype: object
```

... but not an indefinite sequence:

```
In [26]: pd.Series(open('exam1grades.txt'))
...
TypeError: object of type '_io.TextIOWrapper' has no len()
```

Series from Dictionary

```
salary = {"Data Scientist": 110000,
          "DevOps Engineer": 110000,
          "Data Engineer": 106000,
          "Analytics Manager": 112000,
          "Database Administrator": 93000,
          "Software Architect": 125000,
          "Software Engineer": 101000,
          "Supply Chain Manager": 100000}
```

Create a `pd.Series` from a dict: [1]

```
In [14]: salary_data = pd.Series(salary)

In [15]: salary_data
Out[15]:
Analytics Manager        112000
Data Engineer            106000
Data Scientist           110000
Database Administrator    93000
DevOps Engineer          110000
Software Architect       125000
Software Engineer        101000
Supply Chain Manager     100000
dtype: int64
```

The index is a sorted sequence of the keys of the dictionary passed to `pd.Series`

[1] https://www.glassdoor.com/List/Best-Jobs-in-America-LST_KQ0,20.htm

Series with Custom Index

General form of Series constructor is `pd.Series(data, index=index)`

- Default is integer sequence for sequence data and sorted keys of dictionaries
- Can provide a custom index:

```
In [29]: pd.Series([1,2,3], index=['a', 'b', 'c'])
Out[29]:
a    1
b    2
c    3
dtype: int64
```

The index object itself is an immutable array with set operations.

```
In [30]: i1 = pd.Index([1,2,3,4])

In [31]: i2 = pd.Index([3,4,5,6])

In [32]: i1[1:3]
Out[32]: Int64Index([2, 3], dtype='int64')

In [33]: i1 & i2 # intersection
Out[33]: Int64Index([3, 4], dtype='int64')

In [34]: i1 | i2 # union
Out[34]: Int64Index([1, 2, 3, 4, 5, 6], dtype='int64')

In [35]: i1 ^ i2 # symmetric difference
Out[35]: Int64Index([1, 2, 5, 6], dtype='int64')
```

Series Indexing and Slicing

Indexing feels like dictionary access due to flexible index objects:

```
In [37]: data = pd.Series(['a', 'b', 'c', 'd'])

In [38]: data[0]
Out[38]: 'a'

In [39]: salary_data['Software Engineer']
Out[39]: 101000
```

But you can also slice using these flexible indices:

```
In [40]: salary_data['Data Scientist':'Software Engineer']
Out[40]:
Data Scientist          110000
Database Administrator   93000
DevOps Engineer         110000
Software Architect      125000
Software Engineer       101000
dtype: int64
```

Georgia Tech

Basic DataFrame Structure

A DataFrame is a series Serieses with the same keys. For example, consider the following dictionary of dictionaries meant to leverage your experience with spreadsheets (in spreadsheet.py):

```
In [5]: import spreadsheet; spreadsheet.cells

Out[5]:
{'A': {1: 'A1', 2: 'A2', 3: 'A3'},
 'B': {1: 'B1', 2: 'B2', 3: 'B3'},
 'C': {1: 'C1', 2: 'C2', 3: 'C3'},
 'D': {1: 'D1', 2: 'D2', 3: 'D3'}}
```

All of these dictionaries have the same keys, so we can pass this dictionary of dictionaries to the DataFrame constructor:

```
In [7]: ss = pd.DataFrame(spreadsheet.cells); ss

Out[7]:
    A   B   C   D
1  A1  B1  C1  D1
2  A2  B2  C2  D2
3  A3  B3  C3  D3
```

- Each column is a Series whose keys (index) are the values printed to the left (1, 2 and 3).
- Each row is a Series whose keys (index) are the column headers.

Try evaluating `ss.columns` and `ss.index`.

DataFrame Example

Download hotjobs.py and do a `%load hotjobs.py` (to evaluate the code in the top-level namespace instead of importing it).

```
In [42]: jobs = pd.DataFrame({'salary': salary, 'openings': openings})

In [43]: jobs
Out[43]:
                          openings  salary
Analytics Manager             1958  112000
Data Engineer                 2599  106000
Data Scientist                4184  110000
Database Administrator        2877   93000
DevOps Engineer               2725  110000
Software Architect            2232  125000
Software Engineer            17085  101000
Supply Chain Manager          1270  100000
UX Designer                   1691   92500
```

```
In [46]: jobs.index
Out[46]:
Index(['Analytics Manager', 'Data Engineer', 'Data Scientist',
       'Database Administrator', 'DevOps Engineer', 'Software Architect',
       'Software Engineer', 'Supply Chain Manager', 'UX Designer'],
      dtype='object')

In [47]: jobs.columns
Out[47]: Index(['openings', 'salary'], dtype='object')
```

Simple DataFrame Indexing

Simplest indexing of DataFrame is by column name.

```
In [48]: jobs['salary']
Out[48]:
Analytics Manager        112000
Data Engineer            106000
Data Scientist           110000
Database Administrator    93000
DevOps Engineer          110000
Software Architect       125000
Software Engineer        101000
Supply Chain Manager     100000
UX Designer               92500
Name: salary, dtype: int64
```

Each colum is a Series:

```
In [49]: type(jobs['salary'])
Out[49]: pandas.core.series.Series
```

General Row Indexing

The `loc` indexer indexes by row name:

```
In [13]: jobs.loc['Software Engineer']
Out[13]:
openings    17085
salary     101000
Name: Software Engineer, dtype: int64

In [14]: jobs.loc['Data Engineer':'Databse Administrator']
Out[14]:
                      openings  salary
Data Engineer             2599  106000
Data Scientist            4184  110000
Database Administrator    2877   93000
```

Note that slice ending is inclusive when indexing by name.

The `iloc` indexer indexes rows by position:

```
In [15]: jobs.iloc[1:3]
Out[15]:
              openings  salary
Data Engineer     2599  106000
Data Scientist    4184  110000
```

Note that slice ending is exclusive when indexing by integer position.

Special Case Row Indexing

```
In [16]: jobs[:2]
Out[16]:
                  openings  salary
Analytics Manager     1958  112000
Data Engineer         2599  106000

In [17]: jobs[jobs['salary'] > 100000]
Out[17]:
                    openings  salary
Analytics Manager       1958  112000
Data Engineer           2599  106000
Data Scientist          4184  110000
DevOps Engineer         2725  110000
Software Architect      2232  125000
Software Engineer      17085  101000
```

These are shortcuts for `loc` and `iloc` indexing:

```
In [20]: jobs.iloc[:2]
Out[20]:
                  openings  salary
Analytics Manager     1958  112000
Data Engineer         2599  106000

In [21]: jobs.loc[jobs['salary'] > 100000]
Out[21]:
                    openings  salary
Analytics Manager       1958  112000
Data Engineer           2599  106000
Data Scientist          4184  110000
DevOps Engineer         2725  110000
```

Aggregate Functions

The values in a series is a `numpy.ndarray`, so you can use NumPy functions, broadcasting, etc.

- Average salary for all these jobs:

```
In [14]: np.average(jobs['salary'])
Out[14]: 107125.0
```

- Total number of openings:

```
In [15]: np.sum(jobs['openings'])
Out[15]: 34930
```

And so on.

Adding Columns

Add column by broadcasting a constant value:

```
In [16]: jobs['DM Prepares'] = True

In [17]: jobs
Out[17]:
                          openings  salary  DM Prepares
Analytics Manager             1958  112000         True
Data Engineer                 2599  106000         True
Data Scientist                4184  110000         True
Database Administrator        2877   93000         True
DevOps Engineer               2725  110000         True
Software Architect            2232  125000         True
Software Engineer            17085  101000         True
Supply Chain Manager          1270  100000         True
```

Add column by computing value based on row's data:

```
In [25]: jobs['Percent Openings'] = jobs['openings'] / np.sum(jobs['openings'])

In [26]: jobs
Out[26]:
                          openings  salary  DM Prepares  Percent Openings
Analytics Manager             1958  112000         True          0.056055
Data Engineer                 2599  106000         True          0.074406
Data Scientist                4184  110000         True          0.119782
Database Administrator        2877   93000         True          0.082365
DevOps Engineer               2725  110000         True          0.078013
Software Architect            2232  125000         True          0.063899
Software Engineer            17085  101000         True          0.489121
Supply Chain Manager          1270  100000         True          0.036358
```

CSV Files

Pandas has a very powerful CSV reader. Do this in iPython (or `help(pd.read_csv)` in Python):

```
pd.read_csv?
```

Now let's read the `super-grades.csv` file and re-do Calc Grades exercise using Pandas.

Read a CSV File into a DataFrame

`super-grades.csv` contains:

```
Student,Exam 1,Exam 2,Exam 3
Thorny,100,90,80
Mac,88,99,111
Farva,45,56,67
Rabbit,59,61,67
Ursula,73,79,83
Foster,89,97,101
```

The first line is a header, which Pandas will infer, and we want to use the first column for index values:

```
sgs = pd.read_csv('super-grades.csv', index_col=0)
```

Now we have the DataFrame we want:

```
In [3]: sgs = pd.read_csv('super-grades.csv', index_col=0)

In [4]: sgs
Out[4]:
        Exam 1  Exam 2  Exam 3
Student
Thorny     100      90      80
Mac         88      99     111
Farva       45      56      67
Rabbit      59      61      67
Ursula      73      79      83
Foster      89      97     101
```

Adding a Calculated Column to a DataFrame

We've seen how to add a column broadcast from a scalar value or a simple calculation from another column. Now let's add a column with the average grades for each student.

If we apply this to the DataFrame we get a Series with averages. Notice that we're "collapsing" columns (axis=1), that is, calculating values from a row like we did in NumPy:

```
In [33]: sgs.apply(course_avg, axis=1)
Out[33]:
Student
Thorny    90.000000
Mac       99.333333
Farva     56.000000
Rabbit    62.333333
Ursula    78.333333
Foster    95.666667
dtype: float64
```

So we just add this series to the DataFrame:

```
In [35]: sgs["avg"] = sgs.apply(course_avg, axis=1); sgs
Out[35]:
        Exam 1  Exam 2  Exam 3        avg
Student
Thorny     100      90      80  90.000000
Mac         88      99     111  99.333333
Farva       45      56      67  56.000000
Rabbit      59      61      67  62.333333
Ursula      73      79      83  78.333333
```

Appending DataFrames

Now let's add a new row containing the averages for each exam.

- We can get the item averages by applying `np.mean` to the columns (axis=0 – "collapsing" rows):

```
In [35]: sgs.apply(np.mean, axis=0)
Out[35]:
Exam 1    75.666667
Exam 2    80.333333
Exam 3    84.833333
avg       80.277778
dtype: float64
```

- We can turn this Series into a DaraFrame with the label we want:

```
In [38]: pd.DataFrame({"ItemAverage": sgs.apply(np.mean, axis=0)})
Out[38]:
        ItemAverage
Exam 1    75.666667
Exam 2    80.333333
Exam 3    84.833333
avg       80.277778
```

Georgia Tech

DataFrame Transpose

- But we need to give this DataFrame the same shape as our grades DataFrame:

```
In [41]: item_avgs = pd.DataFrame({"Item Avg": sgs.apply(np.mean,
     axis=0)}).transpose()

In [43]: item_avgs
Out[43]:
            Exam 1    Exam 2    Exam 3       avg
Item Avg 75.666667 80.333333 84.833333 80.277778
```

Then we can simply append the DataFrame because it has the same columns:

```
In [24]: sgs = sgs.append(item_avgs)

In [25]: sgs
Out[25]:
             Exam 1    Exam 2     Exam 3       avg
Thorny   100.000000 90.000000  80.000000 90.000000
Mac       88.000000 99.000000 111.000000 99.333333
Farva     45.000000 56.000000  67.000000 56.000000
Rabbit    59.000000 61.000000  67.000000 62.333333
Ursula    73.000000 79.000000  83.000000 78.333333
Foster    89.000000 97.000000 101.000000 95.666667
Item Avg  75.666667 80.333333  84.833333 80.277778
```

Note that append is non-destructive, so we have to reassign its returned DataFrame to sgs.

Adding a Letter Grades Column

Adding a column with letter grades is easier than adding a column with a more complex calculation.

```
In [40]: sgs['Grade'] = \
    ...:     np.where(sgs['avg'] >= 90, 'A',
    ...:             np.where(sgs['avg'] >= 80, 'B',
    ...:                     np.where(sgs['avg'] >= 70, 'C',
    ...:                             np.where(sgs['avg'] >= 60, 'D',
    ...:                                     'D'))))
    ...:

In [41]: sgs
Out[41]:
             Exam 1     Exam 2      Exam 3        avg Grade
Thorny   100.000000 90.000000  80.000000 90.000000    A
Mac       88.000000 99.000000 111.000000 99.333333    A
Farva     45.000000 56.000000  67.000000 56.000000    D
Rabbit    59.000000 61.000000  67.000000 62.333333    D
Ursula    73.000000 79.000000  83.000000 78.333333    C
Foster    89.000000 97.000000 101.000000 95.666667    A
Item Avg  75.666667 80.333333  84.833333 80.277778    B
```

Grouping and Aggregation

Grouping and aggregation can be conceptualized as a split, apply, combine pipeline.

- Split by Grade

	Exam 1	Exam 2	Exam 3	avg	Grade
Thorny	100.000000	90.000000	80.000000	90.000000	A
Mac	88.000000	99.000000	111.000000	99.333333	A
Foster	89.000000	97.000000	101.000000	95.666667	A

	Exam 1	Exam 2	Exam 3	avg	Grade
Item Avg	75.666667	80.333333	84.833333	80.277778	B

	Exam 1	Exam 2	Exam 3	avg	Grade
Ursula	73.000000	79.000000	83.000000	78.333333	C

	Exam 1	Exam 2	Exam 3	avg	Grade
Farva	45.000000	56.000000	67.000000	56.000000	D
Rabbit	59.000000	61.000000	67.000000	62.333333	D

- Apply some aggregation function to each group, such as sum, mean, count.
- Combine results of function applications to get final results for each group.

Letter Grades Counts

Here's how to find the counts of letter grades for our super troopers:

```
In [58]: sgs['Grade'].groupby(sgs['Grade']).count()
Out[58]:
Grade
A    3
B    1
C    1
D    2
Name: Grade, dtype: int64
```

Messy CSV Files

Remember the Tides Exercise? Pandas's `read_csv` can handle most of the data pre-processing:

```
pd.read_csv('wpb-tides-2017.txt', sep='\t', skiprows=14, header=None,
            usecols=[0,1,2,3,5,7],
            names=['Date', 'Day', 'Time', 'Pred(ft)', 'Pred(cm)', 'High/Low'],
            parse_dates=['Date','Time'])
```

Let's use the indexing and data selection techniques we've learned to re-do the Tides Exercise as a Jupyter Notebook. For convenience, `wpb-tides-2017.txt` is in the code/analytics directory, or you can download it.