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
pd.set_option("display.width", 120)
```

That last line allows you to display DataFrames with many columns without wrapping.

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 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`:

```
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

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
```

Series Indexing and Slicing

Indexing feels like dictionary access due to flexible index objects (download hotjobs.py to play along):

```
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
```

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
```

Basic DataFrame Structure

```
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
```

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
```

Try `jobs['salary'] > 100000` by itself. What's happening in `In[17]` above?

loc and iloc Indexing

The previous examples 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
Software Architect      2232  125000
Software Engineer      17085  101000
```

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 Column by Applying Ufuncs

```
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 the Python REPL):

```
pd.read_csv?
```

Read a CSV File into a DataFrame

Download credit.csv:

```
In [34]: credit = pd.read_csv(credit.csv)

In [35]: credit
Out[35]:
    age  income  approve
0    64      90        1
1    78      92        1
2    38      80        1
3    29      66       -1
4    94      79        1
5    95      94        1
6    61      40       -1
7    21      38       -1
8    33      54       -1
9    96      50        1
10   83      75        1
11   32      44       -1
12   49      37       -1
13   49      83        1
14   79      56        1
15   90      67        1
16   40      30       -1
17   61      71        1
18   21      53       -1
```