

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`

¹https://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
```


DataFrame

The simplest way to create a DataFrame is with a dictionary of dictionaries:

```
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 column 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':'Database 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

Adding Columns

Add column by broadcasting a single value:

```
In [23]: jobs['CS2316 prepares'] = True
```

```
Out[23]:
```

	openings	salary	2316 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 [24]: jobs['6 Figures'] = jobs['salary'] > 100000
```

```
In [25]: jobs
```

```
Out[25]:
```

	openings	salary	2316 Prepares	6 Figures
Analytics Manager	1958	112000	True	True
Data Engineer	2599	106000	True	True
Data Scientist	4184	110000	True	True
Database Administrator	2877	93000	True	False
DevOps Engineer	2725	110000	True	True
Software Architect	2232	125000	True	True
Software Engineer	17085	101000	True	True
Supply Chain Manager	1270	100000	True	False

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 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. We'll build this up in 3 steps:

1. Find the average of a single student's grades.
2. Create a dictionary mapping all students to their averages.
3. Create a `pd.Series` object from this dictionary.
4. Add this dictionary to our `sgs` `pd.DataFrame`.

Operating on a Row of a DataFrame

We already know how to find the average of a single student's grades:

```
In [5]: np.mean(sgs.loc['Thorny'])  
Out[5]: 90.0
```

Thorny is one of the row (loc) indexes. We get all the row index values with:

```
In [6]: sgs.index  
Out[6]: Index(['Thorny', 'Mac', 'Farva', 'Rabbit', 'Ursula', 'Foster'],  
              dtype='object', name='Student')
```

We can iterate over the index values and create a dictionary mapping all students to their averages.

```
In [7]: {stud: np.mean(sgs.loc[stud]) for stud in sgs.index}  
Out[7]:  
{ 'Farva': 56.0,  
  'Foster': 95.666666666666671,  
  'Mac': 99.333333333333329,  
  'Rabbit': 62.333333333333336,  
  'Thorny': 90.0,  
  'Ursula': 78.333333333333329}
```

Concatenate a Series to a DataFrame

Now that we know how to create a dictionary of student averages, we create a `pd.Series` object from it so we can concatenate the Series to our DataFrame.

```
In [8]: avgs = pd.Series({stud: np.mean(sgs.loc[stud]) for stud in sgs.index})
```

```
In [9]: avgs
```

```
Out[9]:
```

```
Farva      56.000000
Foster     95.666667
Mac        99.333333
Rabbit     62.333333
Thorny     90.000000
Ursula     78.333333
dtype: float64
```

Since the `avgs` Series has the same index as our `sgs` DataFrame, we can simply assign it as a new column:

```
In [11]: sgs['avg'] = avgs
```

```
In [12]: sgs
```

```
Out[12]:
```

	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. Again, we'll build this up in steps.

1. Get the average of a single column.
2. Create a list containing the column averages.
3. Create a `pd.DataFrame` from this dictionary.
4. Append our new `DataFrame` with the averages to the `sgs DataFrame`.

Average of a Single Column

We already know how to get the average of a single column's values:

```
In [14]: sgs['Exam 1']
Out[14]:
Student
Thorny    100
Mac        88
Farva      45
Rabbit     59
Ursula     73
Foster     89
Name: Exam 1, dtype: int64

In [15]: np.mean(sgs['Exam 1'])
Out[15]: 75.666666666666671
```

The column names are stored in an Index:

```
In [16]: sgs.columns
Out[16]: Index(['Exam 1', 'Exam 2', 'Exam 3', 'avg'], dtype='object')
```

Average of all the Columns

We can iterate over the column index to create a dictionary mapping items to dictionaries mapping 'Item Avg' the column average.

```
In [18]: item_avgs = {col: {'Item Avg': np.mean(sgs[col])} for col in sgs.columns}

In [19]: item_avgs
Out[19]:
{'Exam 1': {'Item Avg': 75.66666666666667},
 'Exam 2': {'Item Avg': 80.33333333333333},
 'Exam 3': {'Item Avg': 84.83333333333333},
 'avg': {'Item Avg': 80.27777777777777}}
```

This looks awkward. Remember that:

- ▶ A DataFrame is created from a dictionary of dictionaries.
- ▶ The outer dictionary contains the columns with the keys as the column names and the values of the inner dictionaries as the column values.
- ▶ The inner dictionaries have the same keys, which become row index values with the corresponding values from each inner dictionary under the column headed by the key from the corresponding outer dictionary.

DataFrame from Column Averages

Now that we have this dictionary of dictionaries of column averages, we can create a DataFrame:

```
In [20]: item_avgs_df = pd.DataFrame(item_avgs)
```

```
In [21]: item_avgs_df
```

```
Out[21]:
```

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

... and then append it to sgs

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

```
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.

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](#).