Week 1: Pandas Tutorial for Data Wrangling

Team DSS: Kevin Miao, Youngli Hong, Sandeep Sainath, Amanda Ma

Agenda

1. Introduction & Goals
2. Data Structures
3. Data Selection
4. Applying Functions
5. Reshaping Data
6. Merging Data
7. Data Wrangling in Practice

Introduction & Goals

Introduction & Goals

Why Pandas?

- The Python ecosystem is universally popular in practical machine learning applications
- Pandas is the **most popular choice** for data wrangling

After this module, you will be able to:

- be able to **import and clean raw data** in CSV, Excel, or JSON formats.
- be familiar with the **basic data structures in Pandas** such as DataFrames and Series.
- be familiar with the **basics of data manipulation** in Pandas such as subsetting, selecting, filtering data.
- be able to perform **complex data operations** such as groupbys, pivots, and joins and understand the intuition behind them.
- be able to draw on **common use cases** of Pandas in the context of practical machine learning
- in general, be very comfortable with the **fundamentals** of Pandas for data wrangling.

We assume the following prerequisite knowledge:

- **CS61A and CS61B** or any similar programming experience
    - Object-Oriented Programming
    - Data Structures
    - Proficiency in Python
- Recommended: Introductory machine learning knowledge

Data Structures

Data Structures

<u>DataFrames:</u>

analog of tabular datasets in Pandas

2-dimensional data structure with rows and columns

To create your own DataFrame, use the pd.DataFrame() method to pass in dict-like data

<u>Series:</u>

analog of columns of data in Pandas

array-like operations are element-wise

To create your own Series, use the pd.Series() method

```
In [33]: data
```

Out[33]:

	Area Abbreviation	Area Code	Area	Item Code	Item	Element Code	Element	Unit	latitude	longitude	...	Y2004	Y2005	Y2006	Y2007	Y2008	Y2009
0	AF	2	Afghanistan	2511	Wheat and products	5142	Food	1000 tonnes	33.94	67.71	...	3249.0	3486.0	3704.0	4164.0	4252.0	4538.0
1	AF	2	Afghanistan	2805	Rice (Milled Equivalent)	5142	Food	1000 tonnes	33.94	67.71	...	419.0	445.0	546.0	455.0	490.0	415.0
2	AF	2	Afghanistan	2513	Barley and products	5521	Feed	1000 tonnes	33.94	67.71	...	58.0	236.0	262.0	263.0	230.0	379.0
3	AF	2	Afghanistan	2513	Barley and products	5142	Food	1000 tonnes	33.94	67.71	...	185.0	43.0	44.0	48.0	62.0	55.0
4	AF	2	Afghanistan	2514	Maize and products	5521	Feed	1000 tonnes	33.94	67.71	...	120.0	208.0	233.0	249.0	247.0	195.0
5	AF	2	Afghanistan	2514	Maize and products	5142	Food	1000 tonnes	33.94	67.71	...	231.0	67.0	82.0	67.0	69.0	71.0
6	AF	2	Afghanistan	2517	Millet and products	5142	Food	1000 tonnes	33.94	67.71	...	15.0	21.0	11.0	19.0	21.0	18.0
7	AF	2	Afghanistan	2520	Cereals, Other	5142	Food	1000 tonnes	33.94	67.71	...	2.0	1.0	1.0	0.0	0.0	0.0
8	AF	2	Afghanistan	2531	Potatoes and products	5142	Food	1000 tonnes	33.94	67.71	...	276.0	294.0	294.0	260.0	242.0	250.0
9	AF	2	Afghanistan	2536	Sugar cane	5521	Feed	1000 tonnes	33.94	67.71	...	50.0	29.0	61.0	65.0	54.0	114.0
10	AF	2	Afghanistan	2537	Sugar beet	5521	Feed	1000 tonnes	33.94	67.71	...	0.0	0.0	0.0	0.0	0.0	0.0

<u>Example of a DataFrame with several columns as Series</u>

Data Selection

Data Selection - Indexing, Subsetting

Indexing: .loc() vs. iloc()

.loc() function is used to access a group of rows and columns by **label**

.iloc function is used to access the same by **position**

Object Type	Indexers
Series	`s.loc[indexer]`
DataFrame	`df.loc[row_indexer,column_indexer]`

Example of indexing with .loc

Subsetting Data:

all rows in a DataFrame with a numerical value of a particular column greater than 10 AND string value of a particular column equal to "Hello":

df[(df['column_1'] > 10) & (df['column_2'] == "Hello")]

```
In [27]: s[:5]
Out[27]:
2000-01-01    0.469112
2000-01-02    1.212112
2000-01-03   -0.861849
2000-01-04    0.721555
2000-01-05   -0.424972
Freq: D, Name: A, dtype: float64

In [28]: s[::2]
Out[28]:
2000-01-01    0.469112
2000-01-03   -0.861849
2000-01-05   -0.424972
2000-01-07    0.404705
Freq: 2D, Name: A, dtype: float64

In [29]: s[::-1]
Out[29]:
2000-01-08   -0.370647
2000-01-07    0.404705
2000-01-06   -0.673690
2000-01-05   -0.424972
2000-01-04    0.721555
2000-01-03   -0.861849
2000-01-02    1.212112
2000-01-01    0.469112
Freq: -1D, Name: A, dtype: float64
```

Example of indexing a Series by slicing

Data Selection - Columns, Sorting

Creating/Dropping/Renaming Columns:

df['new_column_name'] = Series_or_list_of_data

Dropping: df.drop('column_name', axis=1)

The *axis=1* parameter indicates that we want to drop a column, not a row (to drop rows, use axis=0).

Renaming: df.rename(columns={'old1':'new1', 'old2':'new2'})

Sorting:

Values in a DataFrame and Series can be sorted in any order using the **sort_values()** method.

Sort the *"order_number"* column in descending order:

df["number"].sort_values(axis=0, ascending=False)

Applying Functions

Applying Functions

The **.apply()** function allows you to pass in any function, built-in or custom, to apply along any axis of a DataFrame

- the *"axis"* parameter in df.apply() with value 0 corresponds to applying a function over its rows, while value 1 corresponds to columns.

- using the NumPy function **"np.sum"** aggregation function to calculate the arithmetic sum in .apply() will return one value, aka the sum of the specified Series.

- using a custom function such as **"lambda x: x + 2"**, which adds 2 to every row, will return a DataFrame/Series with the column modified appropriately.

Reshaping Data

Reshaping Data - Groupbys

Groupby operations involve some combination of splitting the object, applying a function, and combining the results

- **df.groupby()** function enables this functionality. The function returns a *DataFrameGroupBy* object, a special class in Pandas.

Aggregation functions can take the form of both built-in functions and custom functions.

- Common **built-in functions** include .mean(), .max(), .min(), .sum(), and .count().

For custom functions, the **.agg()** function takes in any lambda or pre-defined function and applies to a DataFrameGroupBy object.

Reshaping Data - Pivots, Melts

Pivots:

- a rotational transformation of a DataFrame

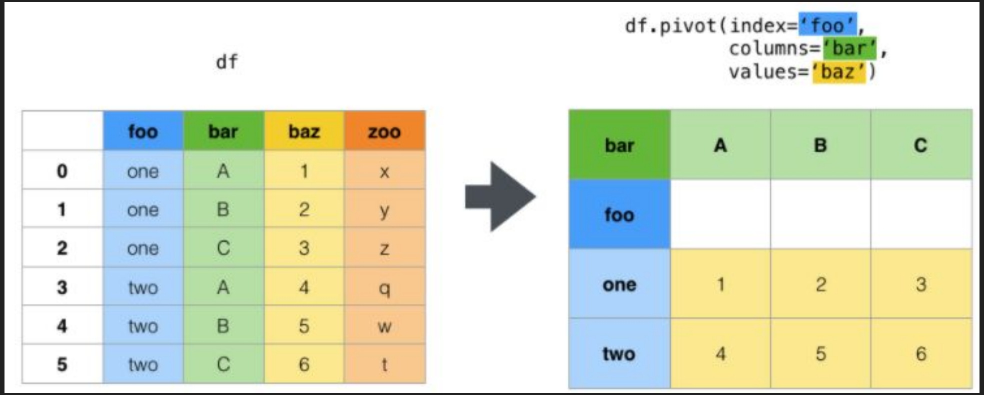
- reorganizes a DataFrame such that the table is indexed by one or more of its columns

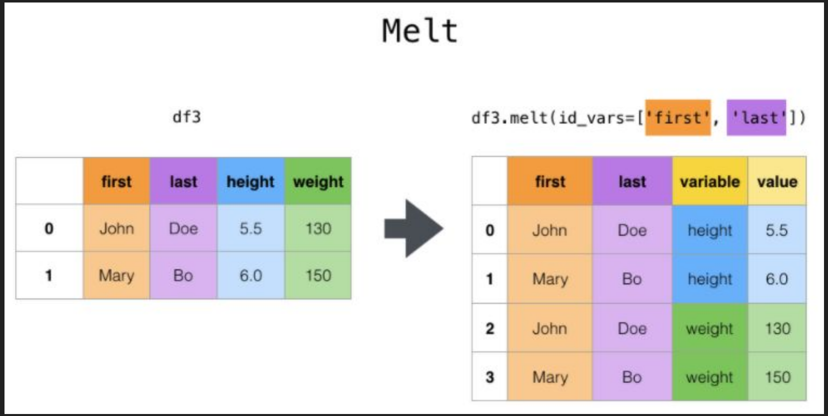
- df.pivot() method takes in three main parameters: "index", "columns", and "values”

Melts:

- involves "unpivoting" columns over columns that are identifier variables.

- df.melt() function specifies this with the "id_vars" parameter through a list

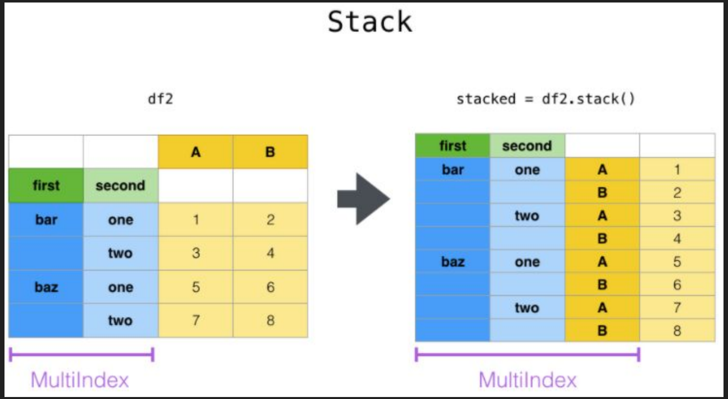




Reshaping Data - Stacking, Unstacking

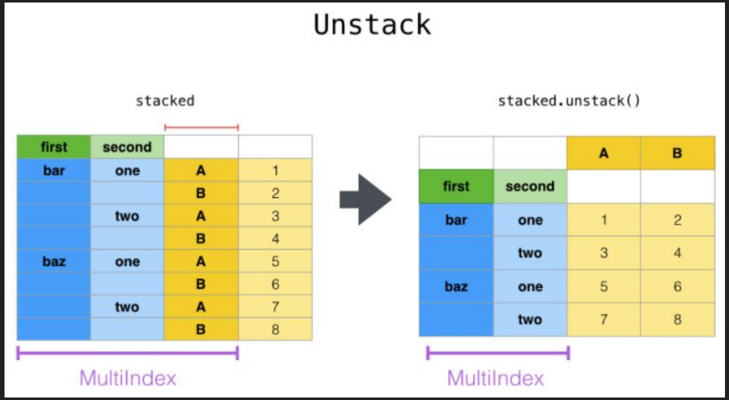
Stacking:

involves adding a column as the next additional level to a DataFrame



Unstacking:

removing a column from the index of a DataFrame and reverting it to a column



Merging Data

```
pd.merge(left, right, how='inner', on=None, left_on=None, right_on=None,
         left_index=False, right_index=False, sort=True,
         suffixes=('_x', '_y'), copy=True, indicator=False,
         validate=None)
```

Syntax for pd.merge()

Pandas has extensive functionality to perform traditional database-style joins. These include inner joins, left joins, right joins, and outer joins.

- **pd.merge()** function enables this functionality.

```
In [39]: left = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
   ....:                      'A': ['A0', 'A1', 'A2', 'A3'],
   ....:                      'B': ['B0', 'B1', 'B2', 'B3']})
   ....:

In [40]: right = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
   ....:                       'C': ['C0', 'C1', 'C2', 'C3'],
   ....:                       'D': ['D0', 'D1', 'D2', 'D3']})
   ....:

In [41]: result = pd.merge(left, right, on='key')
```

left

	key	A	B
0	K0	A0	B0
1	K1	A1	B1
2	K2	A2	B2
3	K3	A3	B3

right

	key	C	D
0	K0	C0	D0
1	K1	C1	D1
2	K2	C2	D2
3	K3	C3	D3

Result

	key	A	B	C	D
0	K0	A0	B0	C0	D0
1	K1	A1	B1	C1	D1
2	K2	A2	B2	C2	D2
3	K3	A3	B3	C3	D3

Visual example of a basic inner join

```
In [54]: left = pd.DataFrame({'A': [1, 2], 'B': [2, 2]})

In [55]: right = pd.DataFrame({'A': [4, 5, 6], 'B': [2, 2, 2]})

In [56]: result = pd.merge(left, right, on='B', how='outer')
```

left

	A	B
0	1	2
1	2	2

right

	A	B
0	4	2
1	5	2
2	6	2

Result

	A_x	B	A_y
0	1	2	4
1	1	2	5
2	1	2	6
3	2	2	4
4	2	2	5
5	2	2	6

<- Visual example of an outer join

Data Wrangling in Practice

Data Wrangling in Practice

Data wrangling is messy and comes in many different forms!

Missing value imputation:

- Detecting missing values: **is.na()**, **not.na()** methods
- Method of imputation: Context is everything!
- Filling in missing values: **.fillna()** method

Iteration: .apply() is extremely computationally efficient. Use over for-loops and .iterrows() unless absolutely necessary.

Time series data:

Concept	Scalar Class	Array Class	pandas Data Type	Primary Creation Method
Date times	`Timestamp`	`DatetimeIndex`	`datetime64[ns]` or `datetime64[ns, tz]`	`to_datetime` or `date_range`
Time deltas	`Timedelta`	`TimedeltaIndex`	`timedelta64[ns]`	`to_timedelta` or `timedelta_range`