Comparison with Stata¶

For potential users coming from Stata this page is meant to demonstrate how different Stata operations would be performed in pandas.

If you’ re new to pandas, you might want to first read through 10 Minutes to pandas to familiarize yourself with the library.

As is customary, we import pandas and NumPy as follows. This means that we can refer to the libraries as pd and np, respectively, for the rest of the document.

```
In [1]: import pandas as pd

In [2]: import numpy as np
```

Note

Throughout this tutorial, the pandas DataFrame will be displayed by calling df.head(), which displays the first N (default 5) rows of the DataFrame. This is often used in interactive work (e.g. Jupyter notebook or terminal) – the equivalent in Stata would be:

```
list in 1/5
```

Data structures

General terminology translation

pandas	Stata
DataFrame	data set
column	variable
row	observation
groupby	bysort
NaN	.

DataFrame / Series

A DataFrame in pandas is analogous to a Stata data set – a two-dimensional data source with labeled columns that can be of different types. As will be shown in this document, almost any operation that can be applied to a data set in Stata can also be accomplished in pandas.

A Series is the data structure that represents one column of a DataFrame. Stata doesn’ t have a separate data structure for a single column, but in general, working with a Series is analogous to referencing a column of a data set in Stata.

Index

Every DataFrame and Series has an Index – labels on the *rows* of the data. Stata does not have an exactly analogous concept. In Stata, a data set’ s rows are essentially unlabeled, other than an implicit integer index that can be accessed with _n.

In pandas, if no index is specified, an integer index is also used by default (first row = 0, second row = 1, and so on). While using a labeled Index or MultiIndex can enable sophisticated analyses and is ultimately an important part of pandas to understand, for this comparison we will essentially ignore the `Index` and just treat the `DataFrame` as a collection of columns. Please see the indexing documentation for much more on how to use an `Index` effectively.

Data input / output

Constructing a DataFrame from values

A Stata data set can be built from specified values by placing the data after an `input` statement and specifying the column names.

```
input x y
1 2
3 4
5 6
end
```

A pandas `DataFrame` can be constructed in many different ways, but for a small number of values, it is often convenient to specify it as a Python dictionary, where the keys are the column names and the values are the data.

```
In [3]: df = pd.DataFrame({'x': [1, 3, 5], 'y': [2, 4, 6]})

In [4]: df
Out[4]:
   x  y
0  1  2
1  3  4
2  5  6
```

Reading external data

Like Stata, pandas provides utilities for reading in data from many formats. The `tips` data set, found within the pandas tests (csv) will be used in many of the following examples.

Stata provides `import delimited` to read csv data into a data set in memory. If the `tips.csv` file is in the current working directory, we can import it as follows.

```
import delimited tips.csv
```

The pandas method is `read_csv()`, which works similarly. Additionally, it will automatically download the data set if presented with a url.

```
In [5]: url = ('https://raw.github.com/pandas-dev'
   ...:        '/pandas/master/pandas/tests/data/tips.csv')
   ...:

In [6]: tips = pd.read_csv(url)

In [7]: tips.head()
Out[7]:
   total_bill   tip     sex smoker  day    time  size
0       16.99  1.01  Female     No  Sun  Dinner     2
1       10.34  1.66    Male     No  Sun  Dinner     3
2       21.01  3.50    Male     No  Sun  Dinner     3
3       23.68  3.31    Male     No  Sun  Dinner     2
4       24.59  3.61  Female     No  Sun  Dinner     4
```

Like `import delimited`, `read_csv()` can take a number of parameters to specify how the data should be parsed. For example, if the data were instead tab delimited, did not have column names, and existed in the current working directory, the pandas command would be:

```
tips = pd.read_csv('tips.csv', sep='\t', header=None)

# alternatively, read_table is an alias to read_csv with tab delimiter
tips = pd.read_table('tips.csv', header=None)
```

Pandas can also read Stata data sets in `.dta` format with the `read_stata()` function.

```
df = pd.read_stata('data.dta')
```

In addition to text/csv and Stata files, pandas supports a variety of other data formats such as Excel, SAS, HDF5, Parquet, and SQL databases. These are all read via a `pd.read_*` function. See the IO documentation for more details.

Exporting data

The inverse of `import delimited` in Stata is `export delimited`

```
export delimited tips2.csv
```

Similarly in pandas, the opposite of `read_csv` is `DataFrame.to_csv()`.

```
tips.to_csv('tips2.csv')
```

Pandas can also export to Stata file format with the `DataFrame.to_stata()` method.

```
tips.to_stata('tips2.dta')
```

Data operations

Operations on columns

In Stata, arbitrary math expressions can be used with the `generate` and `replace` commands on new or existing columns. The `drop` command drops the column from the data set.

```
replace total_bill = total_bill - 2
generate new_bill = total_bill / 2
drop new_bill
```

pandas provides similar vectorized operations by specifying the individual `Series` in the `DataFrame`. New columns can be assigned in the same way. The `DataFrame.drop()` method drops a column from the `DataFrame`.

```
In [8]: tips['total_bill'] = tips['total_bill'] - 2

In [9]: tips['new_bill'] = tips['total_bill'] / 2

In [10]: tips.head()
Out[10]:
   total_bill   tip     sex smoker  day    time  size  new_bill
0       14.99  1.01  Female     No  Sun  Dinner     2     7.495
1        8.34  1.66    Male     No  Sun  Dinner     3     4.170
2       19.01  3.50    Male     No  Sun  Dinner     3     9.505
3       21.68  3.31    Male     No  Sun  Dinner     2    10.840
4       22.59  3.61  Female     No  Sun  Dinner     4    11.295

In [11]: tips = tips.drop('new_bill', axis=1)
```

Filtering

Filtering in Stata is done with an `if` clause on one or more columns.

```
list if total_bill > 10
```

DataFrames can be filtered in multiple ways; the most intuitive of which is using boolean indexing.

```
In [12]: tips[tips['total_bill'] > 10].head()
Out[12]:
   total_bill   tip     sex smoker  day    time  size
0       14.99  1.01  Female     No  Sun  Dinner     2
2       19.01  3.50    Male     No  Sun  Dinner     3
3       21.68  3.31    Male     No  Sun  Dinner     2
4       22.59  3.61  Female     No  Sun  Dinner     4
5       23.29  4.71    Male     No  Sun  Dinner     4
```

If/then logic

In Stata, an `if` clause can also be used to create new columns.

```
generate bucket = "low" if total_bill < 10
replace bucket = "high" if total_bill >= 10
```

The same operation in pandas can be accomplished using the `where` method from `numpy`.

```
In [13]: tips['bucket'] = np.where(tips['total_bill'] < 10, 'low', 'high')

In [14]: tips.head()
Out[14]:
   total_bill   tip     sex smoker  day    time  size bucket
0       14.99  1.01  Female     No  Sun  Dinner     2   high
1        8.34  1.66    Male     No  Sun  Dinner     3    low
2       19.01  3.50    Male     No  Sun  Dinner     3   high
3       21.68  3.31    Male     No  Sun  Dinner     2   high
4       22.59  3.61  Female     No  Sun  Dinner     4   high
```

Date functionality

Stata provides a variety of functions to do operations on date/datetime columns.

```
generate date1 = mdy(1, 15, 2013)
generate date2 = date("Feb152015", "MDY")

generate date1_year = year(date1)
generate date2_month = month(date2)

* shift date to beginning of next month
generate date1_next = mdy(month(date1) + 1, 1, year(date1)) if month(date1) != 12
replace date1_next = mdy(1, 1, year(date1) + 1) if month(date1) == 12
generate months_between = mofd(date2) - mofd(date1)

list date1 date2 date1_year date2_month date1_next months_between
```

The equivalent pandas operations are shown below. In addition to these functions, pandas supports other Time Series features not available in Stata (such as time zone handling and custom offsets) – see the timeseries documentation for more details.

```
In [15]: tips['date1'] = pd.Timestamp('2013-01-15')

In [16]: tips['date2'] = pd.Timestamp('2015-02-15')

In [17]: tips['date1_year'] = tips['date1'].dt.year

In [18]: tips['date2_month'] = tips['date2'].dt.month

In [19]: tips['date1_next'] = tips['date1'] + pd.offsets.MonthBegin()

In [20]: tips['months_between'] = (tips['date2'].dt.to_period('M')
   ....:                           - tips['date1'].dt.to_period('M'))
   ....:

In [21]: tips[['date1', 'date2', 'date1_year', 'date2_month', 'date1_next',
   ....:       'months_between']].head()
   ....:
Out[21]:
       date1      date2  date1_year  date2_month date1_next    months_between
0 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
1 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
2 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
3 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
4 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
```

Selection of columns

Stata provides keywords to select, drop, and rename columns.

```
keep sex total_bill tip

drop sex

rename total_bill total_bill_2
```

The same operations are expressed in pandas below. Note that in contrast to Stata, these operations do not happen in place. To make these changes persist, assign the operation back to a variable.

```
# keep
In [22]: tips[['sex', 'total_bill', 'tip']].head()
Out[22]:
      sex  total_bill   tip
0  Female       14.99  1.01
1    Male        8.34  1.66
2    Male       19.01  3.50
3    Male       21.68  3.31
4  Female       22.59  3.61

# drop
In [23]: tips.drop('sex', axis=1).head()
Out[23]:
   total_bill   tip smoker  day    time  size
0       14.99  1.01     No  Sun  Dinner     2
1        8.34  1.66     No  Sun  Dinner     3
2       19.01  3.50     No  Sun  Dinner     3
3       21.68  3.31     No  Sun  Dinner     2
4       22.59  3.61     No  Sun  Dinner     4

# rename
In [24]: tips.rename(columns={'total_bill': 'total_bill_2'}).head()
Out[24]:
   total_bill_2   tip     sex smoker  day    time  size
0         14.99  1.01  Female     No  Sun  Dinner     2
1          8.34  1.66    Male     No  Sun  Dinner     3
2         19.01  3.50    Male     No  Sun  Dinner     3
3         21.68  3.31    Male     No  Sun  Dinner     2
4         22.59  3.61  Female     No  Sun  Dinner     4
```

Sorting by values

Sorting in Stata is accomplished via `sort`

```
sort sex total_bill
```

pandas objects have a `DataFrame.sort_values()` method, which takes a list of columns to sort by.

```
In [25]: tips = tips.sort_values(['sex', 'total_bill'])

In [26]: tips.head()
Out[26]:
     total_bill   tip     sex smoker   day    time  size
67         1.07  1.00  Female    Yes   Sat  Dinner     1
92         3.75  1.00  Female    Yes   Fri  Dinner     2
111        5.25  1.00  Female     No   Sat  Dinner     1
145        6.35  1.50  Female     No  Thur   Lunch     2
135        6.51  1.25  Female     No  Thur   Lunch     2
```

String processing

Finding length of string

Stata determines the length of a character string with the **`strlen()`** and **`ustrlen()`** functions for ASCII and Unicode strings, respectively.

```
generate strlen_time = strlen(time)
generate ustrlen_time = ustrlen(time)
```

Python determines the length of a character string with the `len` function. In Python 3, all strings are Unicode strings. `len` includes trailing blanks. Use `len` and `rstrip` to exclude trailing blanks.

```
In [27]: tips['time'].str.len().head()
Out[27]:
67     6
92     6
111    6
145    5
135    5
Name: time, dtype: int64

In [28]: tips['time'].str.rstrip().str.len().head()
Out[28]:
67     6
92     6
111    6
145    5
135    5
Name: time, dtype: int64
```

Finding position of substring

Stata determines the position of a character in a string with the `strpos()` function. This takes the string defined by the first argument and searches for the first position of the substring you supply as the second argument.

```
generate str_position = strpos(sex, "ale")
```

Python determines the position of a character in a string with the `find()` function. `find` searches for the first position of the substring. If the substring is found, the function returns its position. Keep in mind that Python indexes are zero-based and the function will return -1 if it fails to find the substring.

```
In [29]: tips['sex'].str.find("ale").head()
Out[29]:
67     3
92     3
111    3
145    3
135    3
Name: sex, dtype: int64
```

Extracting substring by position

Stata extracts a substring from a string based on its position with the `substr()` function.

```
generate short_sex = substr(sex, 1, 1)
```

With pandas you can use `[]` notation to extract a substring from a string by position locations. Keep in mind that Python indexes are zero-based.

```
In [30]: tips['sex'].str[0:1].head()
Out[30]:
67     F
92     F
111    F
145    F
135    F
Name: sex, dtype: object
```

Extracting nth word

The Stata `word()` function returns the nth word from a string. The first argument is the string you want to parse and the second argument specifies which word you want to extract.

```
clear
input str20 string
"John Smith"
"Jane Cook"
end

generate first_name = word(name, 1)
generate last_name = word(name, -1)
```

Python extracts a substring from a string based on its text by using regular expressions. There are much more powerful approaches, but this just shows a simple approach.

```
In [31]: firstlast = pd.DataFrame({'string': ['John Smith', 'Jane Cook']})

In [32]: firstlast['First_Name'] = firstlast['string'].str.split(" ", expand=True)[0]

In [33]: firstlast['Last_Name'] = firstlast['string'].str.rsplit(" ", expand=True)[0]

In [34]: firstlast
Out[34]:
       string First_Name Last_Name
0  John Smith       John      John
1   Jane Cook       Jane      Jane
```

Changing case

The Stata `strupper()`, `strlower()`, `strproper()`, `ustrupper()`, `ustrlower()`, and `ustrtitle()` functions change the case of ASCII and Unicode strings, respectively.

```
clear
input str20 string
"John Smith"
"Jane Cook"
end

generate upper = strupper(string)
generate lower = strlower(string)
generate title = strproper(string)
list
```

The equivalent Python functions are `upper`, `lower`, and `title`.

```
In [35]: firstlast = pd.DataFrame({'string': ['John Smith', 'Jane Cook']})

In [36]: firstlast['upper'] = firstlast['string'].str.upper()

In [37]: firstlast['lower'] = firstlast['string'].str.lower()

In [38]: firstlast['title'] = firstlast['string'].str.title()

In [39]: firstlast
Out[39]:
       string       upper       lower       title
0  John Smith  JOHN SMITH  john smith  John Smith
1   Jane Cook   JANE COOK   jane cook   Jane Cook
```

Merging

The following tables will be used in the merge examples

```
In [40]: df1 = pd.DataFrame({'key': ['A', 'B', 'C', 'D'],
   ....:                      'value': np.random.randn(4)})
   ....:

In [41]: df1
Out[41]:
  key     value
0   A  0.469112
1   B -0.282863
2   C -1.509059
3   D -1.135632

In [42]: df2 = pd.DataFrame({'key': ['B', 'D', 'D', 'E'],
   ....:                      'value': np.random.randn(4)})
   ....:

In [43]: df2
Out[43]:
  key     value
0   B  1.212112
1   D -0.173215
2   D  0.119209
3   E -1.044236
```

In Stata, to perform a merge, one data set must be in memory and the other must be referenced as a file name on disk. In contrast, Python must have both `DataFrames` already in memory.

By default, Stata performs an outer join, where all observations from both data sets are left in memory after the merge. One can keep only observations from the initial data set, the merged data set, or the intersection of the two by using the values created in the `_merge` variable.

```
* First create df2 and save to disk
clear
input str1 key
B
D
D
E
end
generate value = rnormal()
save df2.dta

* Now create df1 in memory
clear
input str1 key
A
B
C
D
end
generate value = rnormal()

preserve

* Left join
merge 1:n key using df2.dta
keep if _merge == 1

* Right join
restore, preserve
merge 1:n key using df2.dta
keep if _merge == 2

* Inner join
restore, preserve
merge 1:n key using df2.dta
keep if _merge == 3

* Outer join
restore
merge 1:n key using df2.dta
```

pandas DataFrames have a `DataFrame.merge()` method, which provides similar functionality. Note that different join types are accomplished via the `how` keyword.

```
In [44]: inner_join = df1.merge(df2, on=['key'], how='inner')

In [45]: inner_join
Out[45]:
  key   value_x   value_y
0   B -0.282863  1.212112
1   D -1.135632 -0.173215
2   D -1.135632  0.119209

In [46]: left_join = df1.merge(df2, on=['key'], how='left')

In [47]: left_join
Out[47]:
  key   value_x   value_y
0   A  0.469112       NaN
1   B -0.282863  1.212112
2   C -1.509059       NaN
3   D -1.135632 -0.173215
4   D -1.135632  0.119209

In [48]: right_join = df1.merge(df2, on=['key'], how='right')

In [49]: right_join
Out[49]:
  key   value_x   value_y
0   B -0.282863  1.212112
1   D -1.135632 -0.173215
2   D -1.135632  0.119209
3   E       NaN -1.044236

In [50]: outer_join = df1.merge(df2, on=['key'], how='outer')

In [51]: outer_join
Out[51]:
  key   value_x   value_y
0   A  0.469112       NaN
1   B -0.282863  1.212112
2   C -1.509059       NaN
3   D -1.135632 -0.173215
4   D -1.135632  0.119209
5   E       NaN -1.044236
```

Missing data

Like Stata, pandas has a representation for missing data – the special float value `NaN` (not a number). Many of the semantics are the same; for example missing data propagates through numeric operations, and is ignored by default for aggregations.

```
In [52]: outer_join
Out[52]:
  key   value_x   value_y
0   A  0.469112       NaN
1   B -0.282863  1.212112
2   C -1.509059       NaN
3   D -1.135632 -0.173215
4   D -1.135632  0.119209
5   E       NaN -1.044236

In [53]: outer_join['value_x'] + outer_join['value_y']
Out[53]:
0         NaN
1    0.929249
2         NaN
3   -1.308847
4   -1.016424
5         NaN
dtype: float64

In [54]: outer_join['value_x'].sum()
Out[54]: -3.5940742896293765
```

One difference is that missing data cannot be compared to its sentinel value. For example, in Stata you could do this to filter missing values.

```
* Keep missing values
list if value_x == .
* Keep non-missing values
list if value_x != .
```

This doesn’ t work in pandas. Instead, the `pd.isna()` or `pd.notna()` functions should be used for comparisons.

```
In [55]: outer_join[pd.isna(outer_join['value_x'])]
Out[55]:
  key  value_x   value_y
5   E      NaN -1.044236

In [56]: outer_join[pd.notna(outer_join['value_x'])]
Out[56]:
  key   value_x   value_y
0   A  0.469112       NaN
1   B -0.282863  1.212112
2   C -1.509059       NaN
3   D -1.135632 -0.173215
4   D -1.135632  0.119209
```

Pandas also provides a variety of methods to work with missing data – some of which would be challenging to express in Stata. For example, there are methods to drop all rows with any missing values, replacing missing values with a specified value, like the mean, or forward filling from previous rows. See the missing data documentation for more.

```
# Drop rows with any missing value
In [57]: outer_join.dropna()
Out[57]:
  key   value_x   value_y
1   B -0.282863  1.212112
3   D -1.135632 -0.173215
4   D -1.135632  0.119209

# Fill forwards
In [58]: outer_join.fillna(method='ffill')
Out[58]:
  key   value_x   value_y
0   A  0.469112       NaN
1   B -0.282863  1.212112
2   C -1.509059  1.212112
3   D -1.135632 -0.173215
4   D -1.135632  0.119209
5   E -1.135632 -1.044236

# Impute missing values with the mean
In [59]: outer_join['value_x'].fillna(outer_join['value_x'].mean())
Out[59]:
0    0.469112
1   -0.282863
2   -1.509059
3   -1.135632
4   -1.135632
5   -0.718815
Name: value_x, dtype: float64
```

GroupBy

Aggregation

Stata’s `collapse` can be used to group by one or more key variables and compute aggregations on numeric columns.

```
collapse (sum) total_bill tip, by(sex smoker)
```

pandas provides a flexible `groupby` mechanism that allows similar aggregations. See the groupby documentation for more details and examples.

```
In [60]: tips_summed = tips.groupby(['sex', 'smoker'])[['total_bill', 'tip']].sum()

In [61]: tips_summed.head()
Out[61]:
               total_bill     tip
sex    smoker
Female No          869.68  149.77
       Yes         527.27   96.74
Male   No         1725.75  302.00
       Yes        1217.07  183.07
```

Transformation

In Stata, if the group aggregations need to be used with the original data set, one would usually use `bysort` with `egen()`. For example, to subtract the mean for each observation by smoker group.

```
bysort sex smoker: egen group_bill = mean(total_bill)
generate adj_total_bill = total_bill - group_bill
```

pandas `groupby` provides a `transform` mechanism that allows these type of operations to be succinctly expressed in one operation.

```
In [62]: gb = tips.groupby('smoker')['total_bill']

In [63]: tips['adj_total_bill'] = tips['total_bill'] - gb.transform('mean')

In [64]: tips.head()
Out[64]:
     total_bill   tip     sex smoker   day    time  size  adj_total_bill
67         1.07  1.00  Female    Yes   Sat  Dinner     1      -17.686344
92         3.75  1.00  Female    Yes   Fri  Dinner     2      -15.006344
111        5.25  1.00  Female     No   Sat  Dinner     1      -11.938278
145        6.35  1.50  Female     No  Thur   Lunch     2      -10.838278
135        6.51  1.25  Female     No  Thur   Lunch     2      -10.678278
```

By group processing

In addition to aggregation, pandas `groupby` can be used to replicate most other `bysort` processing from Stata. For example, the following example lists the first observation in the current sort order by sex/smoker group.

```
bysort sex smoker: list if _n == 1
```

In pandas this would be written as:

```
In [65]: tips.groupby(['sex', 'smoker']).first()
Out[65]: 
               total_bill   tip   day    time  size  adj_total_bill
sex    smoker                                                      
Female No            5.25  1.00   Sat  Dinner     1      -11.938278
       Yes           1.07  1.00   Sat  Dinner     1      -17.686344
Male   No            5.51  2.00  Thur   Lunch     2      -11.678278
       Yes           5.25  5.15   Sun  Dinner     2      -13.506344
```

Other considerations

Disk vs memory

Pandas and Stata both operate exclusively in memory. This means that the size of data able to be loaded in pandas is limited by your machine’ s memory. If out of core processing is needed, one possibility is the dask.dataframe library, which provides a subset of pandas functionality for an on-disk `DataFrame`.

<< Comparison with SAS Tutorials >>