Stata to Python Equivalents

Contents

Special thanks to John Coglianese for feedback and for supplying the list of "vital" Stata commands. Feedback and requests for additions to the list are always welcome!

The official Pandas documentation includes a "Comparison with Stata" (https://pandas.pydata.org/pandas-docs/stable/comparison_with_stata.html) page which is another great resource.

1 Intro/Note on Notation

Coding in Python is a little different than coding in Stata.

In Stata, you have one dataset in memory. The dataset is a matrix where each column is a "variable" with a unique name and each row has a number (the special variable `_n`). Everything in Stata is built around this paradigm.

Python is a general purpose programming language where a "variable" is not a column of data. Variables can be anything, a single number, a matrix, a list, a string, etc. The Pandas package implements a kind of variable called a DataFrame that acts a lot like the single dataset in Stata. It is a matrix where each column and each row has a name. The key distinction in Python is that a DataFrame is itself a variable and you can work with any number of DataFrames at one time. You can think of each column in a DataFrame as a variable just like in Stata, except that when you reference a column, you also have to specify the DataFrame.

The Stata-to-Python translations below are written assuming that you have a single DataFrame called `df`. Placeholders like `<varname>` and `<dtafile>` show where user-specified values go in each language. Note that in many cases, `<varname>` will be simple text in Stata (e.g., `avg_income`) while in Python it will be a string (`'avg_income'`). If you were to write `df[avg_income]` without quotes, Python would go looking for a variable--a list, a number, a string--that's been defined somewhere else. Because of this, `<varlist>` in Python represents a list of variable names: `['educ', 'income', 2017]`.

2 Input/Output

Stata	Python
`log using <file>`	Python doesn't display results automatically like Stata. You have to explicitly call the `print` function. Using a Jupyter notebook is the closest equivalent.
`help <command>`	`help(<command>)` OR `<command>?` in IPython (as in `pd.read_stata?`)

`cd <directory>`	`import os`
	`os.chdir('<directory>')`
	but this is bad practice. Better practice is to use full pathnames whenever possible.

`use <dtafile>`	`import pandas as pd` `df = pd.read_stata('<dtafile>')`
`use <varlist> using <dtafile>`	`df = pd.read_stata('<dtafile>', columns=<varlist>)`
`import excel using <excelfile>`	`df = pd.read_excel('<excelfile>')`
`import delimited using <csvfile>`	`df = pd.read_csv('<csvfile>')`
`save <filename>, replace`	`df.to_stata('<filename>')` OR `df.to_pickle('<filename>')` for Python-native file type.
`outsheet using <csv_name>, comma`	`df.to_csv('<csv_name>')`
`export excel using <excel_name>`	`df.to_excel('<excel_name>')`

3 Sample Selection

Stata	Python
`keep if <condition>`	`df = df[<condition>]`
`drop if <condition>`	`df = df[~(<condition>)]`
`keep <var>`	`df = df[<var>]`
`keep varstem*`	`df = df.filter(like='varstem*')`
`drop <var>`	`del df[<var>]` OR `df = df.drop(<var>, axis=1)`
`drop varstem*`	`df = df.drop(df.filter(like='varstem*').columns, axis=1)`

4 Data Info and Summary Statistics

Stata	Python
`describe`	`df.info()` OR `df.dtypes` just to get data types. Note that Python does not have value labels like Stata does.
`describe <var>`	`df[<var>].dtype`
`count`	`df.shape[0]` OR `len(df)`. Here `df.shape` returns a tuple with the length and width of the DataFrame.
`count if <condition>`	`df[<condition>].shape[0]` OR `(<condition>).sum()` if the condition involves a DataFrame, e.g., `(df['age'] > 2).sum()`
`summ <var>`	`df['<var>'].describe()`
`summ <var> if <condition>`	`df[<condition>][<var>].describe()` OR `df.loc[<condition>, <var>].describe()`
`summ <var> [aw = <weight>]`	Right now you have to calculate weighted summary stats manually. There are also some tools available in the Statsmodels package.
`summ <var>, d`	`df[<var>].describe()` plus `df[<var>].quantile([.1, .25, .5, .75, .9])` or whatever other statistics you want.

5 Variable Manipulation

Stata	Python
`gen <newvar> = <expression>`	`df[<newvar>] = <expression>`
`gen <newvar> = <expression> if <condition>`	`df.loc[<condition>, <newvar>] = <expression>`. As with Stata, the rows of `df` that don't meet the condition will be missing (`numpy.nan`).
`replace <var> = <expression> if <condition>`	`df.loc[<condition>, <var>] = <expression>`
`rename <var> <newvar>`	`df = df.rename(columns={<var>: <newvar>})`. You can also directly manipulate `df.columns` like a list: `df.columns = ['a', 'b', 'c']`.
`inlist(<var>, <val1>, <val2>)`	`df[<var>].isin((<val1>, <val2>))`

Stata	Python
`inrange(<var>, <val1>, <val2>)`	`df[<var>].between((<val1>, <val2>))`
`subinstr(<str>, " ", "_", .)`	`df[<var>].str.replace(' ', '_')`
`egen <newvar> = count(<var>)`	`<newvar> = df[<var>].notnull().sum()`. NOTE: For these `egen` commands, `<newvar>` is a full (constant) column in Stata, while it is a scalar in Python.
`egen <newvar> = group(<varlist>)`	`<newvar> = econtools.group_id(df, cols=<varlist>)`
`egen <newvar> = max(<var>)`	`<newvar> = df[<var>].max()`
`egen <newvar> = mean(<var>)`	`<newvar> = df[<var>].mean()`
`egen <newvar> = total(<var>)`	`<newvar> = df[<var>].sum()`
`egen <newvar> = <stat>(<var>), by(<groupvars>)`	`df[<newvar>] = df.groupby(<groupvars>)[<var>].transform('<stat>')`.
`collapse (sd) <var> (median) <var> /// (max) <var> (min) <var>, /// by(<groupvars>)`	`df.groupby(<groupvars>)[<var>].agg(['std', 'median', 'min', 'max', 'sum'])`
`collapse (<stat>) <var> [iw = <weight>]`	Manually or maybe through Statsmodels tool.
`collapse (<stat>) <stat_vars>, by(<groupvars>)`	`df.groupby(<groupvars>)[<stat_vars>].<stat>()`
`recode <var> (1/5 = 1)`	N/A, see note below.
`recode <var> (1/5 = 1), gen(<newvar>)`	N/A.
`label var <var> <label>`	N/A.
`label define <labelname> 1 <valuelabel>`	N/A.
`label values <var> <labelname>`	N/A.
`label list <labelname>`	N/A.

Python doesn't have "labels" built into DataFrames like Stata does. However, you can use a dictionary to map data values to labels when necessary.

```
variable_labels = {
    1: "First Category",
    2: "Second Category",
    3: "Last Category",
}
```

6 Panel Data

There is no general equivalent to `tsset` in Python. However, you can accomplish most if not all of the same tasks using a DataFrame's index (the row's equivalent of columns.) In Stata, the "DataFrame" in memory always has the observation row number, denoted by the Stata built-in variable `_n`. In Python and Pandas, a DataFrame index can be anything (though you can also refer to rows by the row number; see `.loc` vs `iloc`). It can also be hierarchical with mutiple levels. It is a much more general tool than `tsset`.

Stata	Python
`tsset <panelvar> <timevar>`	`df = df.set_index([<panelvar>, <timevar>])`
`L.<var>`	`df.shift()` NOTE: The index must be correctly sorted for `shift` to work the way you want it to.
`L2.<var>`	`df.shift(2)`
`F.<var>`	`df.shift(-1)`

6.1 Examples

```
In [1]: import numpy as np

In [2]: import pandas as pd

In [3]: df0 = pd.DataFrame({'var1': np.arange(6),
   ...:                     'id': [1, 1, 2, 2, 3, 3],
   ...:                     'period': [0, 1] * 3})

In [4]: print(df0)
   var1  id  period
0     0   1       0
1     1   1       1
2     2   2       0
3     3   2       1
4     4   3       0
5     5   3       1

In [5]: df = df0.set_index(['id', 'period'])

In [6]: print(df)
           var1
id period
1  0          0
   1          1
2  0          2
   1          3
3  0          4
   1          5

In [7]: df['var1_lag'] = df.groupby(level='id')['var1'].shift()

In [8]: print(df)
           var1  var1_lag
id period
1  0          0       NaN
   1          1       0.0
2  0          2       NaN
   1          3       2.0
3  0          4       NaN
   1          5       4.0

In [9]: df['var1_for'] = df.groupby(level='id')['var1'].shift(-1)

In [10]: print(df)
           var1  var1_lag  var1_for
id period
1  0          0       NaN       1.0
   1          1       0.0       NaN
2  0          2       NaN       3.0
   1          3       2.0       NaN
```

```
3  0          4       NaN       5.0
   1          5       4.0       NaN
```

7 Merging and Joining

Stata	Python
`append using <filename>`	`df_joint = df1.append(df2)`
`merge 1:1 <vars> using <filename>`	`df_joint = df1.join(df2)` if `<vars>` are the DataFrames' indexes, or `df_joint = pd.merge(df1, df2, on=<vars>)` otherwise. Beware that `pd.merge` will not keep the index of either DataFrame. NOTE: Merging in Python is like R, SQL, etc. Needs more robust explanation.

Merging with Pandas DataFrames does not require you to specify "many-to-one" or "one-to-many". Pandas will figure that out based on whether the variables you're merging on are unique or not. However, you can specify what sub-sample of the merge to keep using the keyword argument `how`, e.g., `df_joint = df1.join(df2, how='left')` is the default for `join` while `how='inner'` is the default for `pd.merge`.

Pandas `how`	Stata `, keep()`	Intuition
`how='left'`	`keep(1, 3)`	Keeps all observations in the "left" DataFrame.
`how='right'`	`keep(2, 3)`	Keeps all observations in the "right" DataFrame.
`how='inner'`	`keep(3)`	Keeps observations that are in both DataFrames.
`how='outer'`	`keep(1 2 3)`	Keeps all observations.

8 Reshape

Like with merging, reshaping a DataFrame in Python is a bit different because of the paradigm shift from the "only data table in memory" model of Stata to "a data table is just another object/variable" of Python. But this difference also makes reshaping a little easier in Python.

The most fundamental reshape commands in Python/Pandas are `stack` and `unstack`:

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

In [2]: import numpy as np

In [3]: long = pd.DataFrame(np.arange(8),
   ...:                     columns=['some_variable'],
   ...:                     index=pd.MultiIndex.from_tuples(
   ...:                         [('a', 1), ('a', 2),
   ...:                          ('b', 1), ('b', 2),
   ...:                          ('c', 1), ('c', 2),
   ...:                          ('d', 1), ('d', 2)]))

In [4]: long.index.names=['unit_id', 'time']

In [5]: long.columns.name = 'varname'

In [6]: long
Out[6]:
varname       some_variable
unit_id time
a       1                 0
        2                 1
b       1                 2
        2                 3
c       1                 4
        2                 5
d       1                 6
        2                 7

In [7]: wide = long.unstack('time')

In [8]: wide
Out[8]:
varname some_variable
time                1  2
unit_id
a                   0  1
b                   2  3
c                   4  5
d                   6  7

In [9]: long2 = wide.stack('time')

In [10]: long2
Out[10]:
varname       some_variable
unit_id time
a       1                 0
        2                 1
b       1                 2
        2                 3
```

```
c       1               4
        2               5
d       1               6
        2               7
```

Here Input 3 creates a DataFrame, Input 4 gives each of the index columns a name, and Input 5 names the columns. Coming from Stata, it's a little weird to think of the column names themselves having a "name", but the columns names are just an index like the row names are. It starts to make more sense when you realize columns don't have to be strings. They can be integers, like years or FIPS codes. In those cases, it makes a lot of sense to give the columns a name so you know what you're dealing with.

Input 6 does the reshaping using `unstack('time')`, which takes the index `'time'` and creates a new column for every unique value it has. Notice that the columns now have multiple levels, just like the index previously did. This is another good reason to label your index and columns. If you want to access either of those columns, you can do so as usual, using a tuple to differentiate between the two levels:

```
In [11]: wide[('some_variable', 1)]
Out[11]:
unit_id
a    0
b    2
c    4
d    6
Name: (some_variable, 1), dtype: int32
```

If you want to combine the two levels (like Stata defaults to), you can simply rename the columns:

```
In [13]: wide_single_level_column = wide.copy()

In [14]: wide_single_level_column.columns = [
    ...:        '{}_{}'.format(var, time)
    ...:        for var, time in wide_single_level_column.columns]

In [15]: wide_single_level_column
Out[15]:
                  some_variable_1  some_variable_2
         unit_id
         a                       0               1
         b                       2               3
         c                       4               5
         d                       6               7
```

The `pivot` command can also be useful, but it's a bit more complicated than `stack` and `unstack` and is better to revisit `pivot` after you are comfortable working with DataFrame indexes and columns.

Stata	Python
`reshape <wide/long> <stubs>, i(<vars>) j(<var>)`	wide: `df.unstack(<level>)` long: `df.stack(<column_level>)` see also `df.pivot`

9 Econometrics

Stata	Python
`ttest <var>, by(<var>)`	`from scipy.stats import ttest_ind` `ttest_ind(<array1>, <array2>)`
`xi: i.<var>`	`pd.get_dummies(df[<var>])`
`i.<var2>#c.<var1>`	`pd.get_dummies(df[<var2>]).multiply(df[<var1>])`
`reg <yvar> <xvar> if <condition>, r`	`import econtools.metrics as mt` `results = mt.reg(df[<condition>], <yvar>, <xvar>, robust=True)`
`reg <yvar> <xvar> if <condition>, vce(cluster <clustervar>)`	`results = mt.reg(df[<condition>], <yvar>, <xvar>, cluster=<clustervar>)`
`areg <yvar> <xvar>, absorb(<fe_var>)`	`results = mt.reg(df, <yvar>, <xvar>, a_name=<fe_var>)`
`predict <newvar>, resid`	`<newvar> = results.resid`
`predict <newvar>, xb`	`<newvar> = results.yhat`
`_b[<var>]`, `_se[<var>]`	`results.beta[<var>]`, `results.se[<var>]`
`test <varlist>`	`results.Ftest(<varlist>)`
`test <varlist>, equal`	`results.Ftest(<varlist>, equal=True)`
`lincom <var1> + <var2>`	`econtools.metrics.f_test` with appropriate parameters.
`ivreg2`	`econtools.metrics.ivreg`
`outreg2`	`econtools.outreg`
`reghdfe`	None (hoping to add it to Econtools soon).

10 Plotting

Visualizations are best handled by the packages Matplotlib and Seaborn.

Stata	Python
`binscatter`	`econtools.binscatter`
`maptile`	No quick tool, but easy to do with Cartopy.
`coefplot`	`ax.scatter(results.beta.index, results.beta)` often works. Depends on context.

Stata	Python
`twoway scatter <var1> <var2>`	`df.scatter(<var2>, <var1>)`
`twoway scatter <var1> <var2> if <condition>`	`df[<condition>].scatter(<var2>, <var1>)`
`twoway <connected/line/area/bar/rarea>`	As above, though `ax.plot(<var1>, <var2>)` is better. Like merge, it's a different paradigm, needs more explanation.

11 Other differences

11.1 Missing values

In Python, missing values are represented by a NumPy "not a number" object, `np.nan`. In Stata, missing (`.`) is larger than every number, so `10 < .` yields True. In Python, `np.nan` is never equal to anything. Any comparison involving `np.nan` is always False, even `np.nan == np.nan`.

To look for missing values in DataFrame columns, use any of the following.

- `df[<varname>].isnull()` returns a vector of True and False values for each row of `df[<varname>]`.
- `df[<varname>].notnull()` is the complement of `.isnull()`.
- The function `np.isnan(<arraylike>)` takes an array and returns True or False for each element of the array (a DataFrame is a special type of array).

Another important difference is that `np.nan` is a floating point data type, so any column of a DataFrame that contains missing numbers will be floats. If a column of integers gets changed so that even one row is `np.nan`, the whole column will be converted to floats.

11.2 Floating point equality

In Stata, decimal numbers are never equal to anything, e.g., `3.0 == 3` is False. This is not a problem in Python, the above equality check returns True.