

Re-imagining a Stata/Python combination

James Fiedler

[Note to NASA reviewers: The code for the bulk of the project described here has been submitted to the Technology Transfer Office. Code described near the end of the presentation will be submitted to the Technology Transfer Office when that code is finished.]

Abstract: At last year's Stata Conference, I presented some ideas for combining Stata and the Python programming language within a single interface. Two methods were presented: in one, Python was used to automate Stata; in the other, Python was used to send simulated keystrokes to the Stata GUI. The first method has the drawback of only working in Windows, and the second can be slow and subject to character input limits. In this presentation, I will demonstrate a method for achieving interaction between Stata and Python that does not suffer these drawbacks, and I will present some examples to show how this interaction can be useful.

Support

where I work

NASA Johnson Space Center

who I work for

Universities Space Research Association

What I have in mind

2

To start off, let me describe the kind of “Stata/Python combination” I have in mind for this talk.

The screenshot shows the Stata software interface. The main command window displays the following text:

```

. matrix list m

m[7,4]
price      mpg      rep78  headroom
r1      4749      17      3      3
r2      3799      22      .      3
r3      4816      20      3      4.5
r4      7827      15      4      4
r5      5788      18      3      4
r6      4453      26      .      3
r7      5189      20      3      2

. di "$blah"
some global

```

Below the command window is a 'Command' input field. On the right side, there are two panels:

- Variables Panel:** Lists variables and their labels:

Variable	Label
make	Make and Model
price	Price
mpg	Mileage (mpg)
rep78	Repair Record 1978
headroom	Headroom (in.)
trunk	Trunk space (cu. ft.)
weight	Weight (lbs.)
length	Length (in.)
turn	Turn Circle (ft.)
displacement	Displacement (cu. in.)
gear_ratio	Gear Ratio
foreign	Car type
- Properties Panel:** Shows details for the current variable 'make':

Variables	
Name	make
Label	Make and Model
Type	str 18
Format	%-18s
Value Label	
Notes	

Below this, the 'Data' section shows:

Data	
Filename	auto_copy.dta
Label	1978 Automobile Data
Notes	
Variables	12
Observations	74
Size	3.11K
Memory	64M

The status bar at the bottom shows the file path: C:\Users\jfriedler\Documents\StataFiles\stata_plugins and the current mode: CAP NUM OVR.

In order to have some values to work with, I have loaded a copy of the auto data set, created a matrix `m` from a subset of the data values, and created a global macro `blah`.

Stata has two modes of interaction. There's (what I call) the regular "Stata mode" where you use `ado` code (as in this slide) ...

The screenshot shows the Stata software interface. The main window is in Mata interactive mode, displaying the following code and output:

```

. mata
----- mata (type end to exit) -----
: st_global("blah")
  some global
: st_matrix("m")
      1      2      3      4
1   4749    17      3      3
2   3799    22      .      3
3   4816    20      3     4.5
4   7827    15      4      4
5   5788    18      3      4
6   4453    26      .      3
7   5189    20      3      2
:

```

The right sidebar shows the 'Variables' list and 'Properties' for the selected variable 'make':

Variable	Label
make	Make and Model
price	Price
mpg	Mileage (mpg)
rep78	Repair Record 1978
headroom	Headroom (in.)
trunk	Trunk space (cu. ft.)
weight	Weight (lbs.)
length	Length (in.)
turn	Turn Circle (ft.)
displacement	Displacement (cu. in.)
gear_ratio	Gear Ratio
foreign	Car type

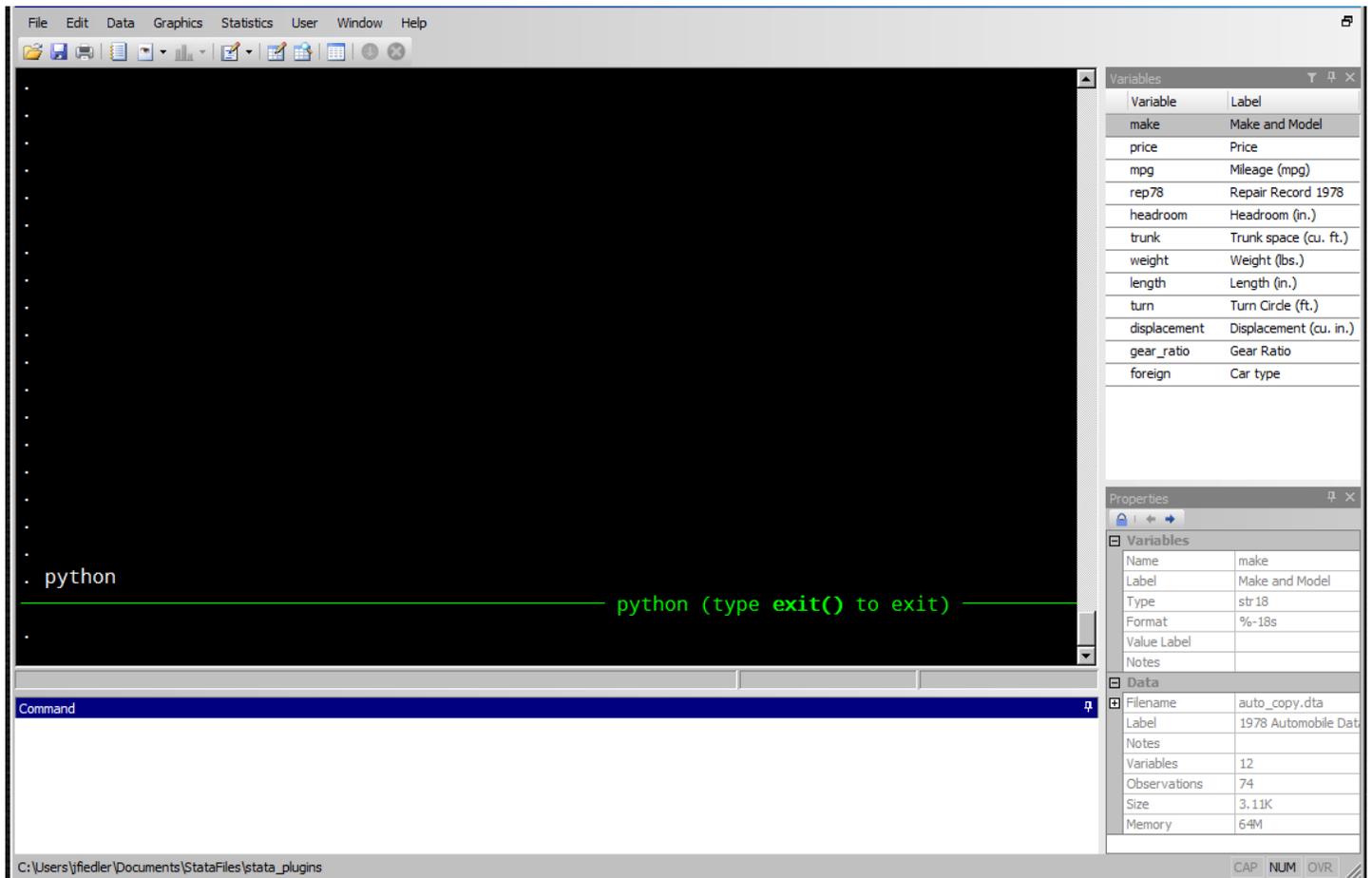
The 'Properties' window for 'make' shows:

Property	Value
Name	make
Label	Make and Model
Type	str18
Format	%-18s
Value Label	
Notes	

The 'Data' window shows:

Property	Value
Filename	auto_copy.dta
Label	1978 Automobile Data
Notes	
Variables	12
Observations	74
Size	3.11K
Memory	64M

... and there's the Mata interactive mode where you use Mata code. The Mata mode is not completely disconnected from the ado mode. There are Mata functions for pulling values from the ado mode into Mata, such as the `st_global()` and `st_matrix()` functions used above.



Just as there's a mata command, which puts the user into an interactive Mata session, there could be a python command which puts the user into an interactive Python session. And just as there are Mata functions for pulling Stata values into the Mata mode, there could be Python functions for pulling Stata values into the Python mode.

Why add Python?

Functionality: Python has a lot of modules for data-related tasks.

Edge cases: There are things you *can* do in Stata that would be easier in Python.

Python is fun.

3

Why do this? First, doing so adds functionality. There is a large community of people using Python for data analysis, numerical computing, scientific computing, etc. As a consequence, there are a lot of high-quality Python modules for doing the kinds of things that Stata users do. Having access to these modules would allow users to do things they couldn't otherwise do in Stata, or couldn't do without spending an enormous amount of time and effort to develop the necessary code.

Second, Python can help in edge cases. These are cases where methods do exist in Stata for doing a task, but work must be expended to make the existing methods handle the task. Here an *annoying* amount of time and effort must be expended, rather than a prohibitive amount. Sometimes there will be an easier way to do the task in Python.

Problem:

Is anyone aware of a Stata user-written routine or other method of importing data stored in NetCDF files into Stata?

— Statalist, "Reading NetCDF files", February 2012

4

Here's an example from Statalist where adding Python to Stata would add functionality. This person had been given a NetCDF file and was wondering whether there any commands had been written for opening such a file.

Response:

... it's common for researchers in the oceanographic/climate fields to use `python` to work with NetCDF

— Statalist, "Reading NetCDF files", February 2012

5

There weren't any Stata commands for opening the file, so one person suggested using Python.

The file was handled some other way, but the user wrote back with some conclusions, one of which was:

There are quite a number of programs that will extract from or use data in NetCDF files but all involve a minimum of one or two intermediate steps before the data can be imported into Stata. It would be nice to eliminate this, but I don't have the time or (probably) expertise to take it on because, at a minimum, it will involve linking C or Fortran [or, *ahem*, Python] programs to Stata.

With Python added to Stata, and with a Python module to open NetCDF files, it would probably only take a few lines to get this file into Stata.

Problem:

I have a large number of large comma-separated text files that I am trying to import. "insheet" is not working

— Statalist, "importing quirky csv", November 2011

6

Here's an example of an edge case, again from Statalist. This person has a CSV file, which, because of some funky formatting, can't be imported using the `insheet` command.

Response:

I've found Python's csv parser to be quite robust and able to write out the csv files in such a way that Stata will happily read them.

— Statalist, "importing quirky csv", November 2011

7

The user could parse the file with lower level commands in Stata or functions in Mata. Another option is to use Python's csv module, as suggested here. Without more information, we can't be sure that it would be easier to parse this particular CSV file with Python. We can see, though, that the respondent has found that some CSV files can be handled more easily with Python.

How do we add Python?

8

Now, how could we make it happen?

Use a C plugin

Stata provides functions in C for interacting with data and matrices.

Python provides a C API for interacting with Python structures.

9

Stata, through its plugin system, provides C functions for interacting with Stata data, macros, matrices, etc. Python, through its C API (application programming interface) provides C functions for starting a Python interpreter and interacting with Python structures.

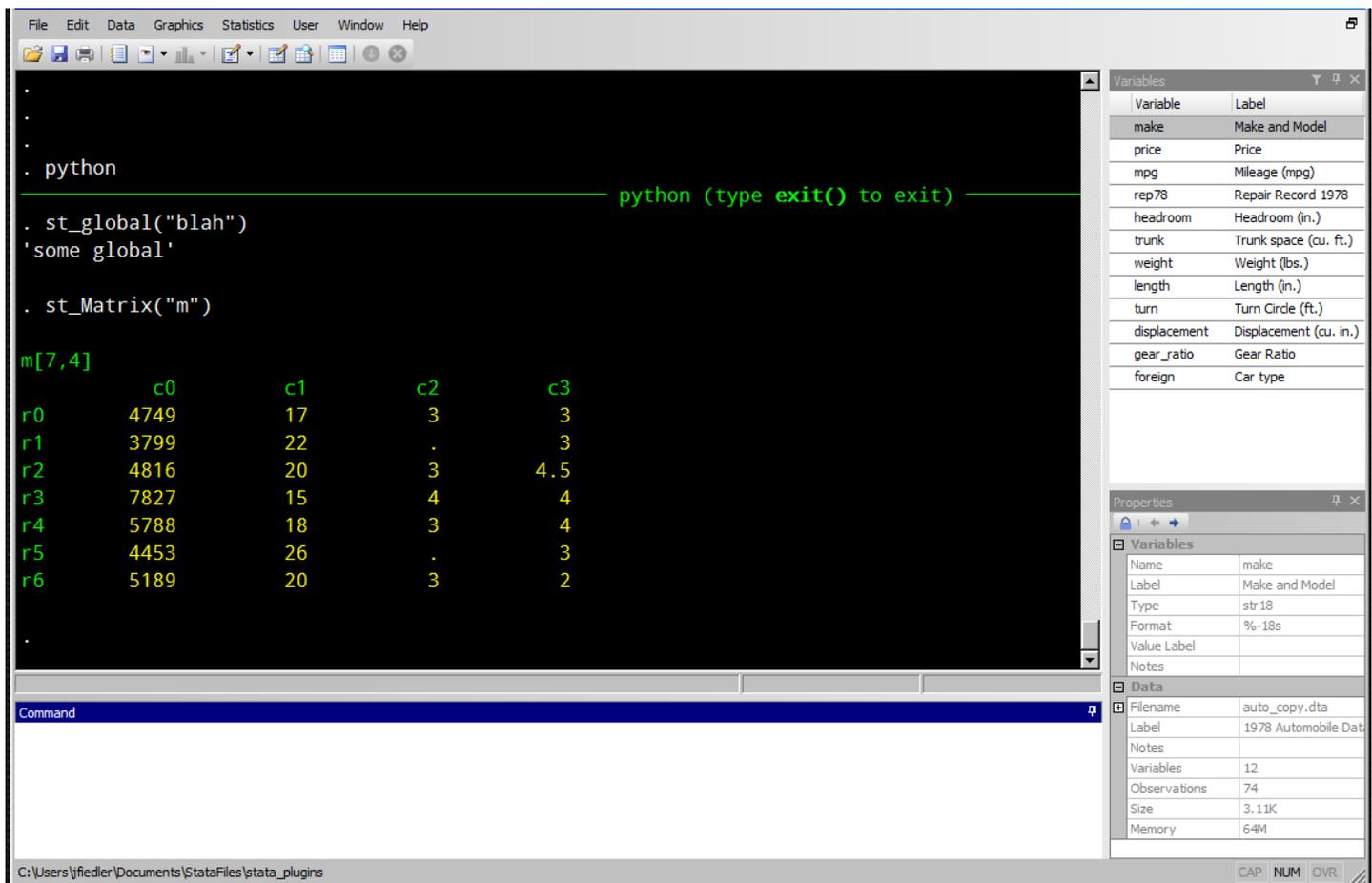
The idea is to match up Stata's C plugin functions with Python's C API functions. In a sense, this would translate the plugin functions from C to Python.

I think this is the way to go, and I wish I could take credit for the idea. In fact, I'm pretty sure the idea came up in a conversation with Stata Corp. employees last year, so credit for the idea should go to them.

Demonstration please

10

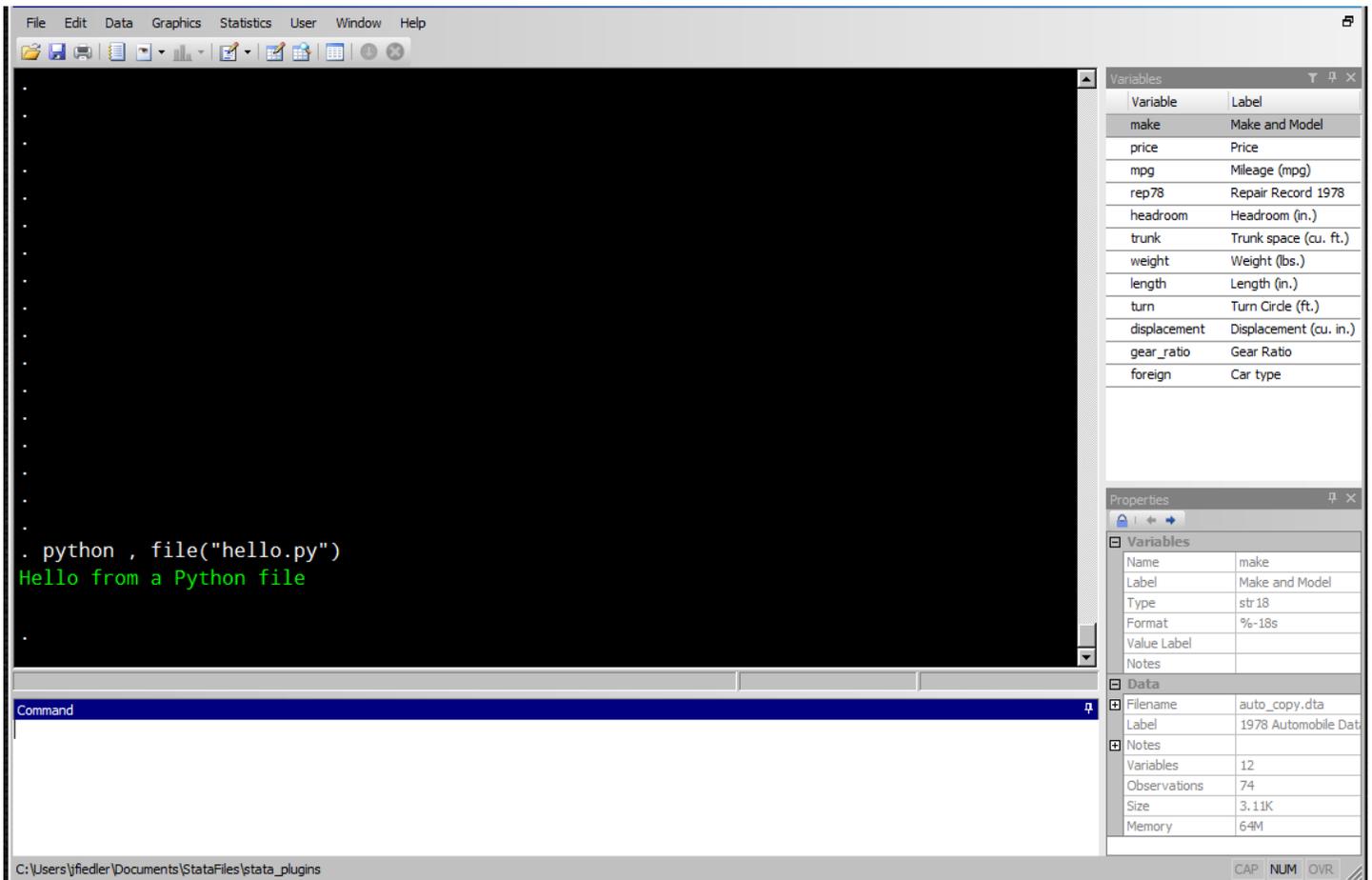
Before I begin a demonstration, I should probably make it clear that this isn't a simulation. Last year I used an imitation of the Stata GUI to demonstrate some ideas for changes to the interface. This year there's no imitation Stata, and no imitation Python. This is a real instance of Stata, and I am really using Python within Stata.



Switching to Stata, the idea, as I said, is to have a python command which puts the user in an interactive Python session. The analogy, again, is with using the command mata to enter an interactive Mata session.

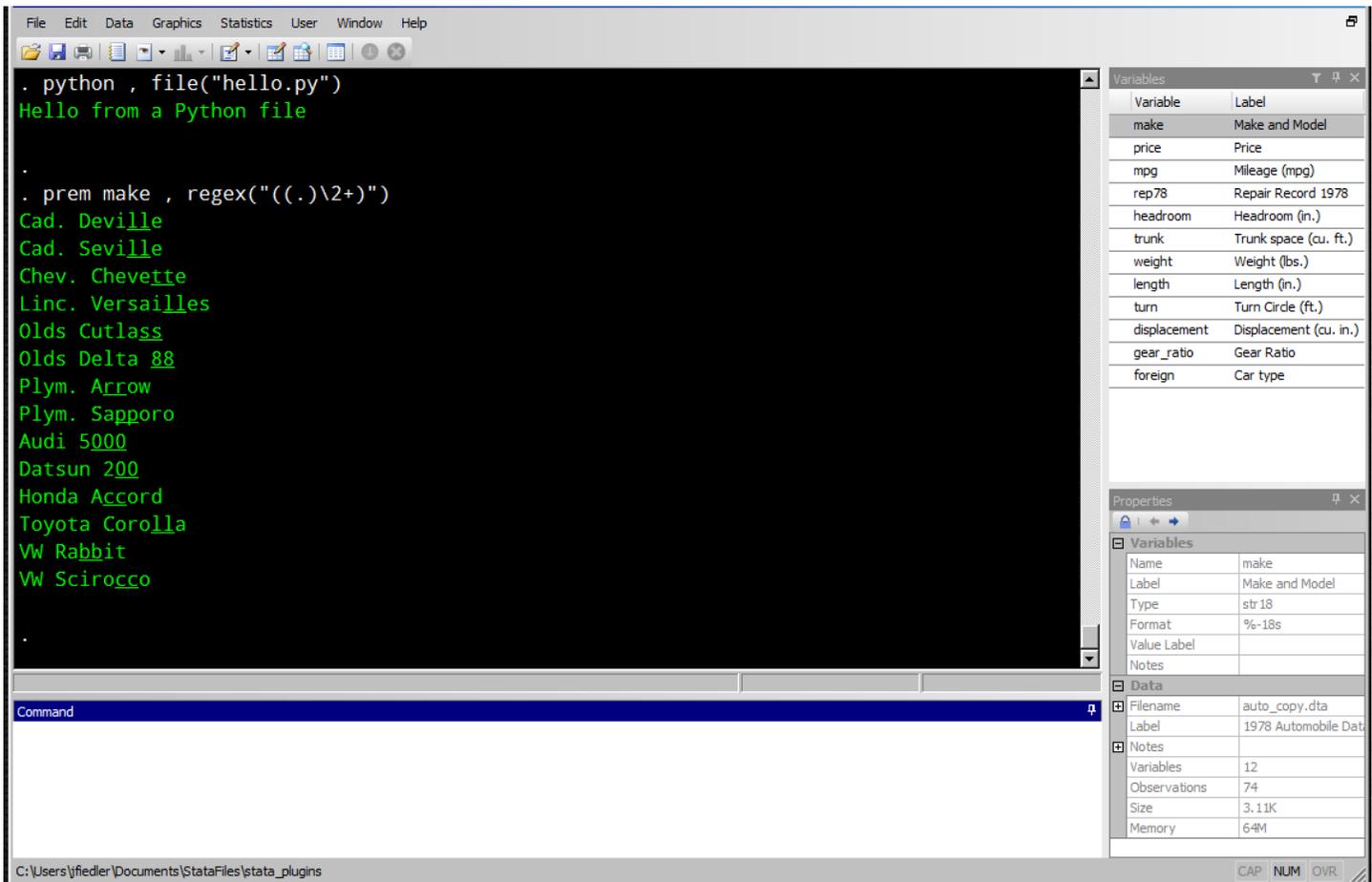
In Mata, there are functions for pulling Stata values in as Mata values. Stata's plugin functions provide some of the same functionality as those Mata functions, so I can also make Python functions that pull Stata values into Python.

There's some leeway in how the Python functions are made. I've decided, ultimately, to mimic the Mata functions as much as possible. To bring a Stata global macro in as a Python string, use the function `st_global()`. To bring a Stata matrix in, use the function `st_Matrix()`. (Notice the capital "M". Python has no inherent notion of a matrix, so I've made a class `st_Matrix`, which is mostly just a view onto a Stata matrix.)



We can also use the Python plugin to run Python files. Here I've exited Python mode (not shown), and used the `python` function with the `file` option. The `hello.py` file is just a single line:

```
print("Hello from a Python file")
```



Here’s an example using regular expressions. Many Stata users rarely if ever use regular expressions, and for those that do, Stata’s regular expressions are probably sufficient for most uses. Occasionally, though, some Stata users want to do something more complicated than what Stata’s regular expressions can handle. In these edge cases, Python can help.

Here I’ve created a Stata command called `prem` (for **p**rint **r**egular **e**xpression **m**atches). It takes a single Stata variable as its argument, and takes a regular expression string as an option. In the example above, the regular expression uses a *back reference* to find repeated characters. For example, it matches the “bb” in “Rabbit” and the “000” in “5000”.

This example also demonstrates how Python files and `.ado` files can work together seamlessly. Thus, users could benefit from having Python without having to know any Python themselves.

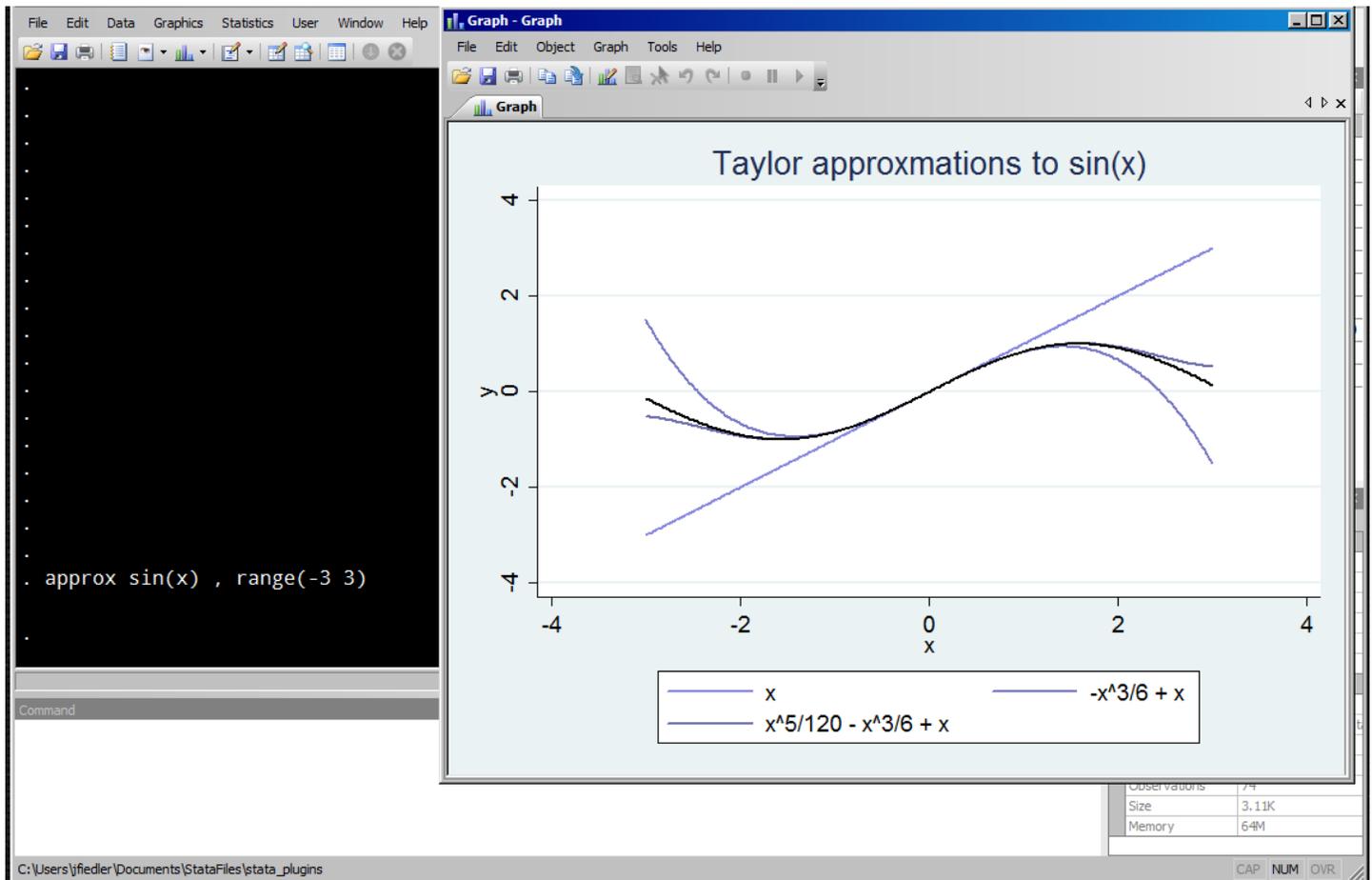
Sympy module

Sympy is a Python library for symbolic mathematics. It aims to become a full-featured computer algebra system.

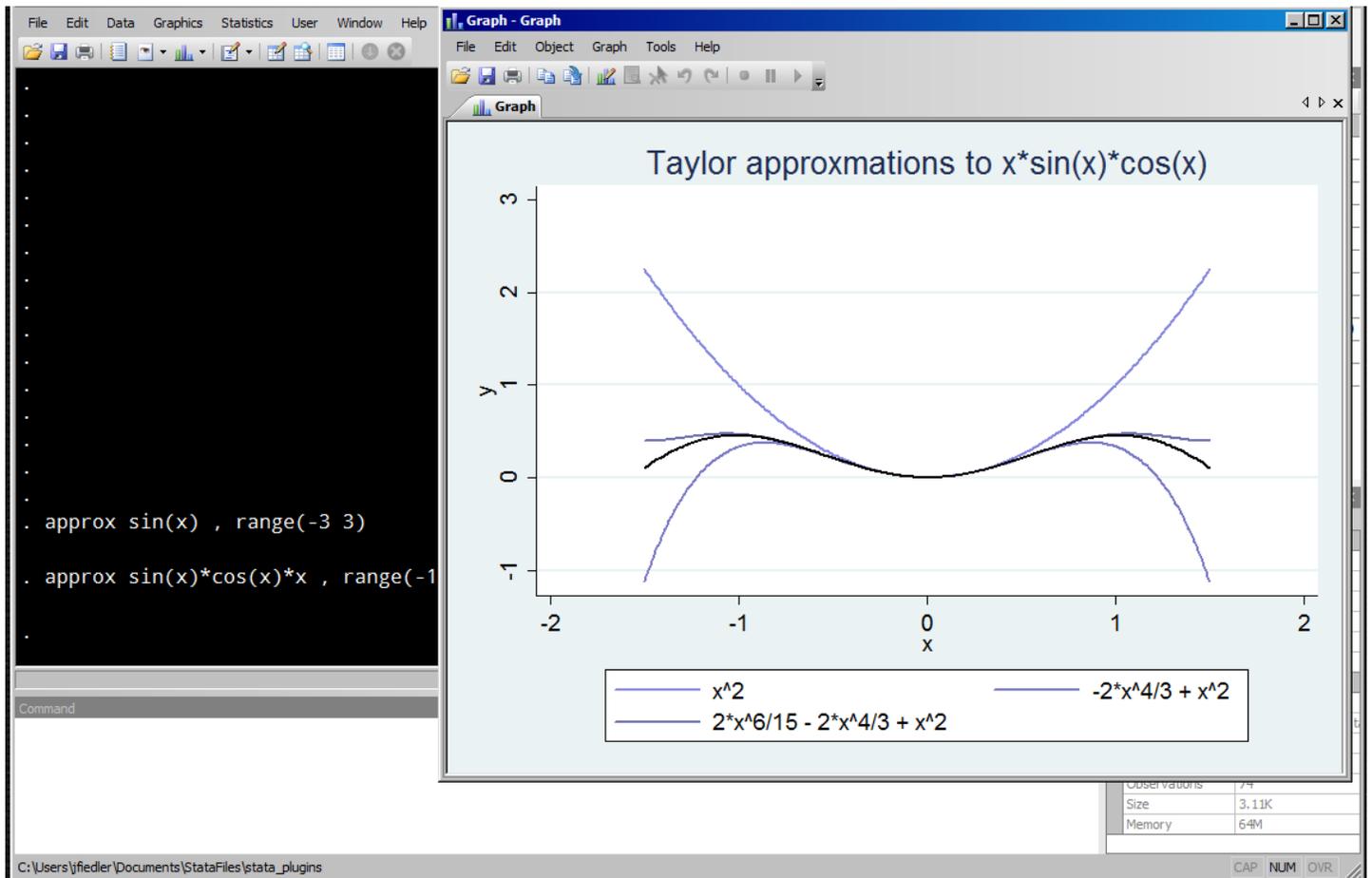
— sympy.org

11

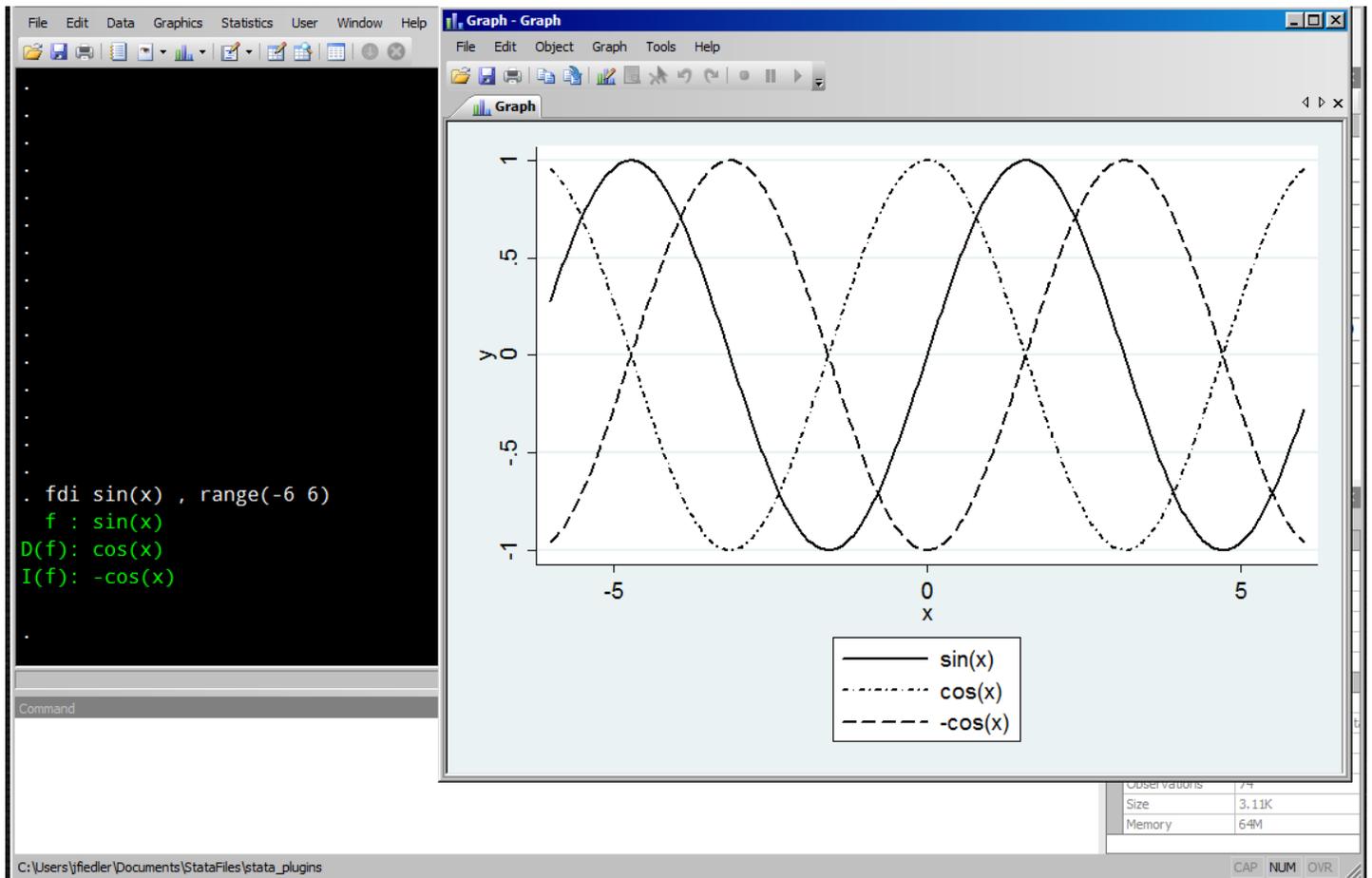
Next I'll show a couple examples with graphs. For these I'll be using a symbolic math module called Sympy. With Sympy you can do things like take the limit of a function, factorize polynomials, and get the Taylor series expansion of a function.



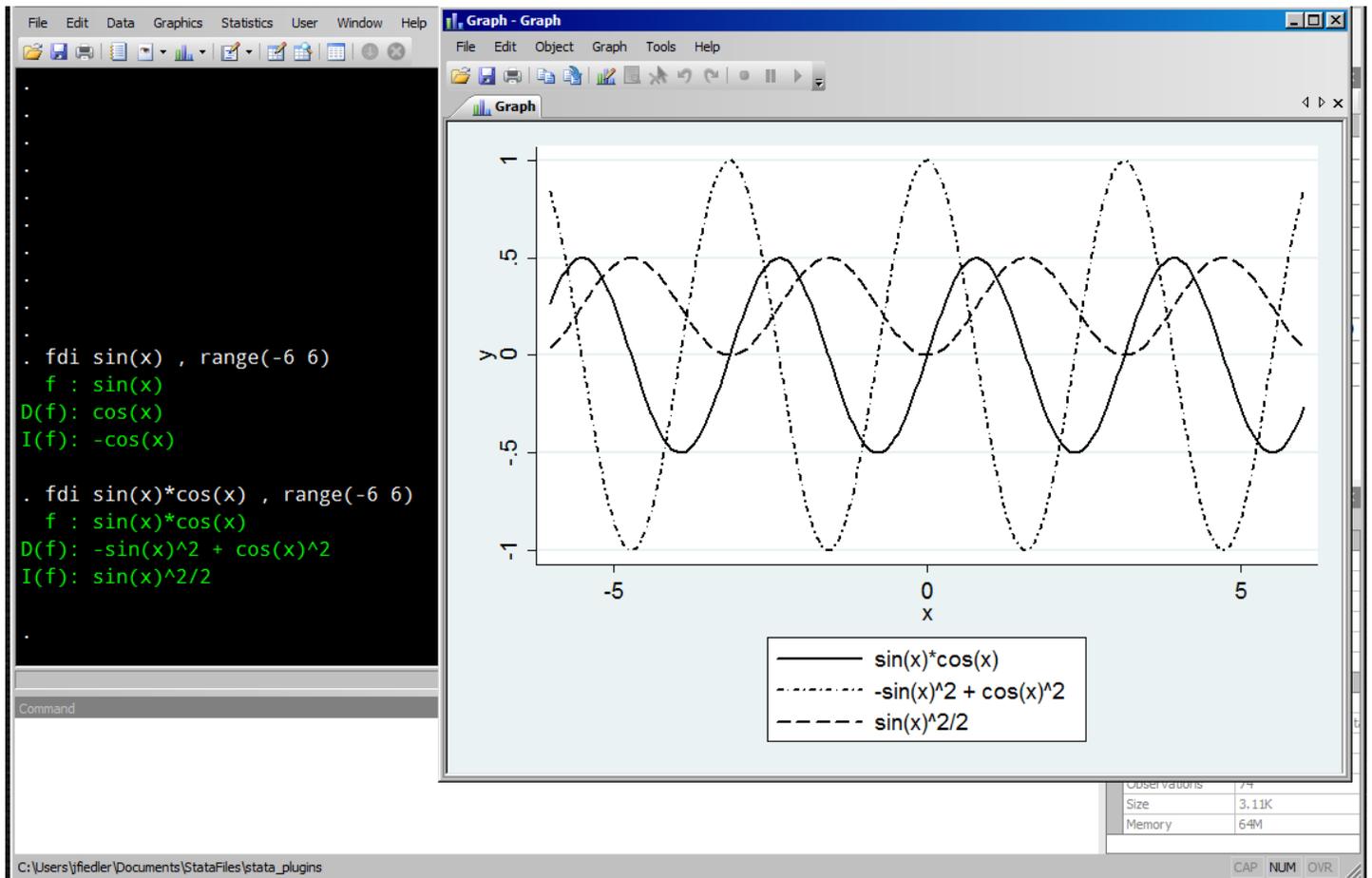
Here I've written a command called `approx`, which takes a function of x as its argument, and graphs the function together with Taylor approximations. In the graph, the thick black line represents $\sin(x)$ and the blue lines show the first three Taylor approximations.



Here's another example for with a more complicated function.



Sympy can also be used to find the functional form of derivatives and integrals. Here is a command for graphing a function with its derivative and integral.



Here's another example, with a more complicated function.

Next: The dta module

12

Looking ahead and imagining using the Python plugin, I would guess that one of common things people would do is it to build or modify datasets. They might use Python to build their datasets from scratch or by importing, e.g., CSV files, or they might want to open and modify existing Stata .dta files. Most likely users would also want to be able to save the Python datasets as .dta files so they could be loaded in Stata.

I'm working on a module for doing these kinds of tasks.

The dta module provides

A Python version of the Stata data structure, the Dta class

Ability to create a Dta object from Python iterables (lists, tuples, etc.)

13

The module includes a complete Python version of the Stata data structure, which includes everything that a .dta file would include.

The module also includes the ability to create a Python dataset from Python array types (“iterable” types such as lists, tuples, generators, etc.).

The dta module provides

Methods for converting
Dta object ↔ .dta file

14

And the module also provides a way to create a Python dataset from a saved .dta file, and to save to a .dta file.

Demonstration please

15

```
. python  
python (type exit() to exit)  
. from stata_dta import Dta  
. v = [[10*i + j for j in range(5)] for i in range(6)]  
. v  
[[0, 1, 2, 3, 4], [10, 11, 12, 13, 14], [20, 21, 22, 23, 24], [30, 31, 32, 33, 34], [40,  
> 51, 52, 53, 54]]  
. for row in v: print(row)  
[0, 1, 2, 3, 4]  
[10, 11, 12, 13, 14]  
[20, 21, 22, 23, 24]  
[30, 31, 32, 33, 34]  
[40, 41, 42, 43, 44]  
[50, 51, 52, 53, 54]  
. 
```

Variable	Label
make	Make and Model
price	Price
mpg	Mileage (mpg)
rep78	Repair Record 1978
headroom	Headroom (in.)
trunk	Trunk space (cu. ft.)
weight	Weight (lbs.)
length	Length (in.)
turn	Turn Circle (ft.)
displacement	Displacement (cu. in.)
gear_ratio	Gear Ratio
foreign	Car type

Name	make
Label	Make and Model
Type	str18
Format	%-18s
Value Label	
Notes	

Filename	auto_copy.dta
Label	1978 Automobile Data
Notes	
Variables	12
Observations	74
Size	3.11K
Memory	64M

The next slide shows an example. The current slide shows some preparatory work.

Here is purpose of each of the above inputs (notice the labels 0-4 to the left of the slide):

0. Start the Python interactive mode
1. The “dta module” is actually called `stata_dta`. This line bring the `Dta` class constructor (i.e., dataset constructor) into the interactive session.
2. This line creates an array of arrays of values called `v`.
3. This is just to show the value of `v`. Think of the outer array of `v` as the dataset, and think of each inner array as a row within the dataset.
4. Another way of looking at what’s in `v`. This stacks the inner arrays to make it easier to visualize `v` as rows within a dataset.

The screenshot shows the Stata software interface. The command window contains the following code:

```
. data = Dta(v)
. data.list()
```

The output of the `list()` command is a table with 6 rows and 5 columns:

	var0	var1	var2	var3	var4
0.	0	1	2	3	4
1.	10	11	12	13	14
2.	20	21	22	23	24
3.	30	31	32	33	34
4.	40	41	42	43	44
5.	50	51	52	53	54

The right-hand side of the window shows the **Variables** panel with the following list:

Variable	Label
make	Make and Model
price	Price
mpg	Mileage (mpg)
rep78	Repair Record 1978
headroom	Headroom (in.)
trunk	Trunk space (cu. ft.)
weight	Weight (lbs.)
length	Length (in.)
turn	Turn Circle (ft.)
displacement	Displacement (cu. in.)
gear_ratio	Gear Ratio
foreign	Car type

The **Properties** panel shows the following information for the current dataset:

Variables	
Name	make
Label	Make and Model
Type	str 18
Format	%-18s
Value Label	
Notes	

The **Data** panel shows the following information:

Filename	auto_copy.dta
Label	1978 Automobile Data
Notes	
Variables	12
Observations	74
Size	3.11K
Memory	64M

The command window shows the path `C:\Users\jfriedler\Documents\StataFiles\stata_plugins` and the status bar displays `CAP NUM OVR`.

A dataset is created from `v` just by calling `Dta` with `v` as argument. To see the values in a dataset, use the `list()` method.

The screenshot shows the Stata software interface. The main window displays the following commands and their output:

```

. data.summ()

      Variable |      Obs   Mean   Std. Dev.   Min   Max
-----+-----+-----+-----+-----+-----
      var0     |         6    25    18.7083     0    50
      var1     |         6    26    18.7083     1    51
      var2     |         6    27    18.7083     2    52
      var3     |         6    28    18.7083     3    53
      var4     |         6    29    18.7083     4    54

. data.save("temp.dta", replace=True)

. exit()

```

The Properties window on the right shows the following information:

Variable	Label
make	Make and Model
price	Price
mpg	Mileage (mpg)
rep78	Repair Record 1978
headroom	Headroom (in.)
trunk	Trunk space (cu. ft.)
weight	Weight (lbs.)
length	Length (in.)
turn	Turn Circle (ft.)
displacement	Displacement (cu. in.)
gear_ratio	Gear Ratio
foreign	Car type

The Properties window also shows the following information:

Variables	
Name	make
Label	Make and Model
Type	str 18
Format	%-18s
Value Label	
Notes	

The Properties window also shows the following information:

Data	
Filename	auto_copy.dta
Label	1978 Automobile Data
Notes	
Variables	12
Observations	74
Size	3.11K
Memory	64M

The Command window at the bottom is empty. The status bar at the bottom shows the file path: C:\Users\jfriedler\Documents\StataFiles\stata_plugins and the status: CAP NUM OVR.

We can also summarize the data, and once we're satisfied with it, we can save the dataset as a dta file. Above, the optional `replace` argument is used because there's already a `temp.dta` file in this directory. In the last line above, `exit()` is used to return to Stata's default mode.

The screenshot displays the Stata software interface. The main window shows the following commands and their output:

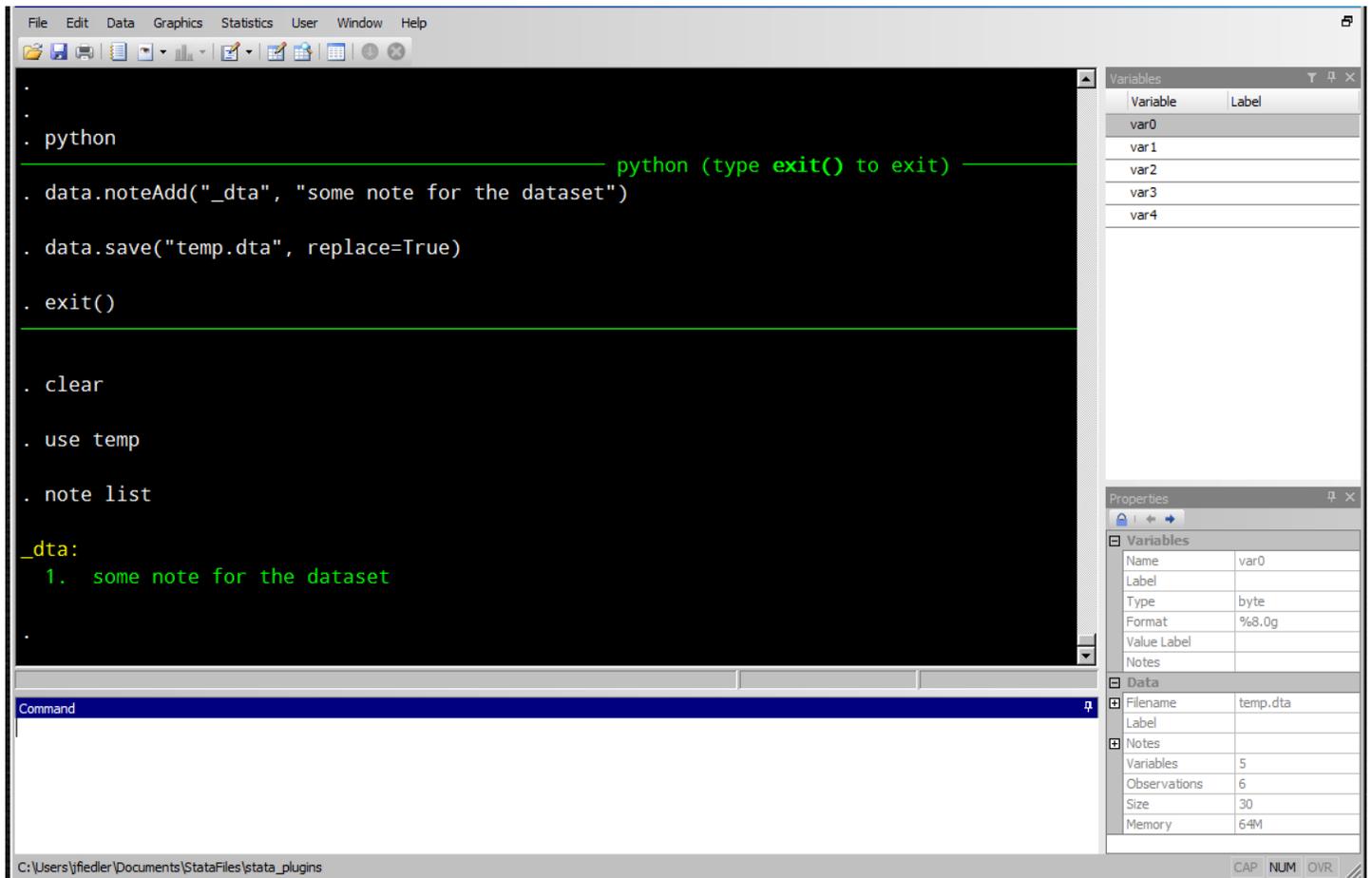
```
. clear  
. use temp  
. list
```

	var0	var1	var2	var3	var4
1.	0	1	2	3	4
2.	10	11	12	13	14
3.	20	21	22	23	24
4.	30	31	32	33	34
5.	40	41	42	43	44
6.	50	51	52	53	54

The Command window at the bottom is empty. The right-hand side of the interface features two panels: 'Variables' and 'Properties'. The 'Variables' panel lists the variables var0 through var4. The 'Properties' panel shows details for the selected variable 'var0', including its name, label, type (byte), format (%8.0g), and value label. Below this, the 'Data' panel provides summary statistics for the dataset: filename (temp.dta), label, notes, 5 variables, 6 observations, 30 size, and 64M memory.

C:\Users\jriedler\Documents\StataFiles\stata_plugins CAP NUM OVR

Now in Stata mode, the inputs above clear the existing dataset (the auto dataset was loaded), loads temp.dta, and lists the values.



The dataset in Python can contain anything that a .dta file would. Here I demonstrate that notes can be added in Python.

What else could we do?

16

What else could we do with Stata datasets in Python? Here's an idea: how about adding functionality for recording the data units? Obviously, knowing the data units is necessary for understanding the contents of a dataset. There already are places available for recording this information, such as in variable labels or notes, but these aren't specifically for recording units. It might be useful to have a dedicated place for units and dedicated functions for setting, removing, replacing, etc.

The screenshot shows a Stata window with a Python interactive interpreter. The code executed is:

```

. python
. from units_dta import UDta
. data = UDta("auto_copy.dta")
(1978 Automobile Data)
. data.summ()

```

The output is a summary table:

Variable	Obs	Mean	Std. Dev.	Min	Max
make	0				
price	74	6165.26	2949.5	3291	15906
mpg	74	21.2973	5.7855	12	41
rep78	69	3.4058	0.989932	1	5
headroom	74	2.99324	0.845995	1.5	5
trunk	74	13.7568	4.2774	5	23
weight	74	3019.46	777.194	1760	4840

The Stata interface also shows a 'Variables' panel on the right with a table:

Variable	Label
var0	
var1	
var2	
var3	
var4	

Below the main window, the 'Properties' panel shows details for the current dataset:

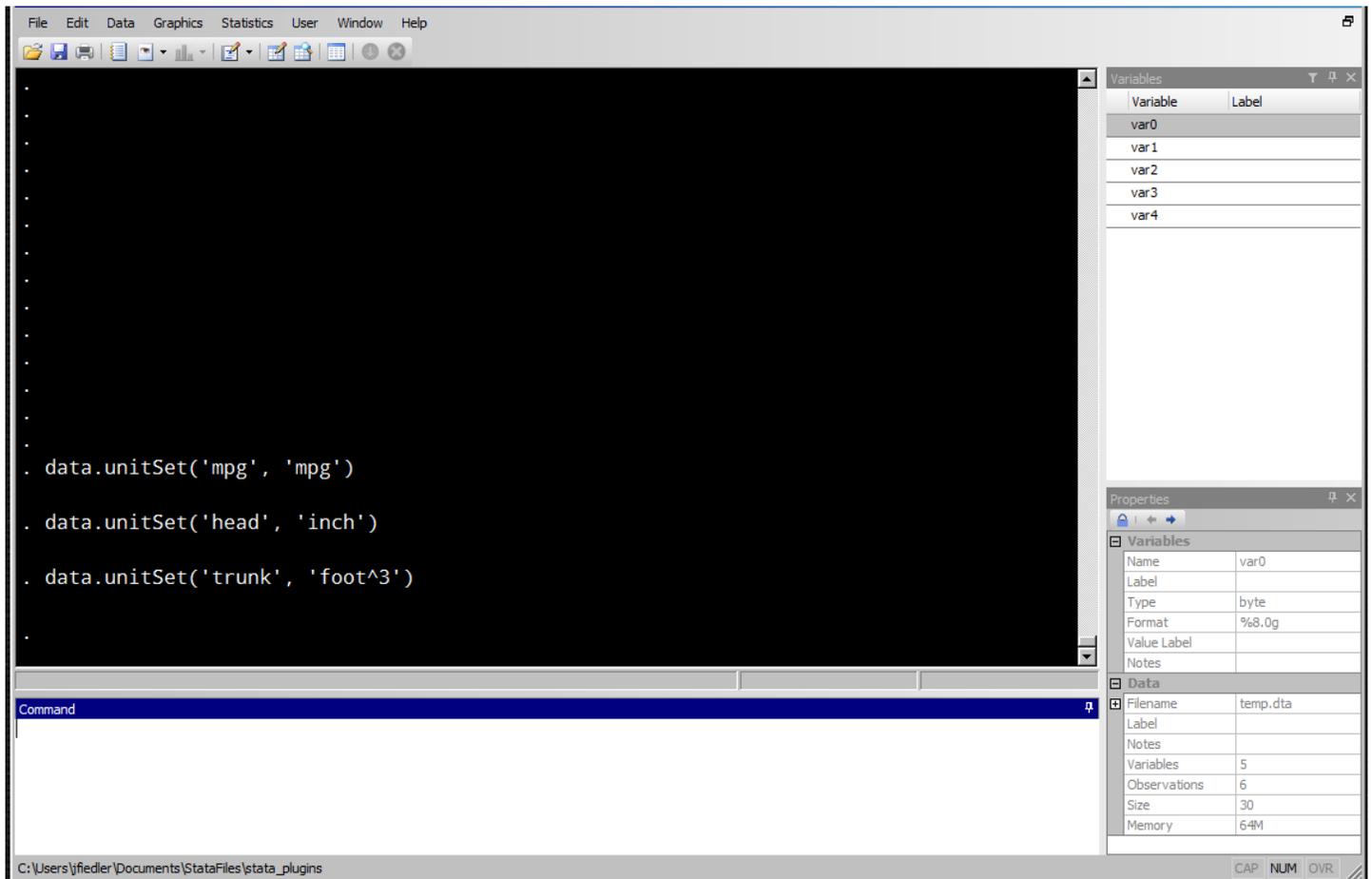
Variables	
Name	var0
Label	
Type	byte
Format	%8.0g
Value Label	
Notes	

Below that, the 'Data' panel shows:

Data	
Filename	temp.dta
Label	
Notes	
Variables	5
Observations	6
Size	30
Memory	64M

The status bar at the bottom indicates 'CAP NUM OVR'.

Here I've entered the Python interactive interpreter, then imported UDta from the `units_dta` module. UDta is subclass of Dta. It has all of the functionality that Dta has, but I've also added some functions for working with units. Above, I've used UDta to open a saved dataset and then summarize it, just as I would with Dta.



Part of the added functionality in UDta is a method `unitSet()` for setting the units. Here I've set the units for `mpg`, `headroom`, and `trunk`. In normal usage I would set the rest of the units likewise.

The screenshot shows the Stata software interface. The main window displays the following commands and results:

```
. data.summ('head')  
  
Variable | Obs   Mean   Std. Dev.   Min   Max  
-----|-----|-----|-----|-----|-----  
headroom |   74  2.99324  0.845995   1.5   5  
  
. data.unitConvert('head', 'cm')  
  
. data.summ('head')
```

The second summary table shows the results after conversion to centimeters:

Variable	Obs	Mean	Std. Dev.	Min	Max
headroom	74	7.60284	2.14883	3.81	12.7

The right-hand side of the interface shows the 'Variables' and 'Properties' panels. The 'Variables' panel lists variables var0 through var4. The 'Properties' panel shows details for the selected variable 'var0', including Name, Label, Type (byte), Format (%8.0g), Value Label, and Notes. The 'Data' panel shows details for the current dataset, including Filename (temp.dta), Label, Notes, Variables (5), Observations (6), Size (30), and Memory (64M).

Now, suppose you're in the US (so these units probably seem natural to you), and you give this dataset to someone in Europe, who would probably prefer metric units. The `UDta` class has a method `unitConvert()` for making conversions. This method doesn't just replace the old unit with the new unit, it also makes the necessary conversion of data values. Above I've summarized `headroom` while the units are inch, then converted to cm, and then summarized again to show that the values have been multiplied by 2.54.

The screenshot shows the Stata command window with the following commands and output:

```

. data.rename('mpg', 'efficiency')
. data.summ('eff')

```

Variable	Obs	Mean	Std. Dev.	Min	Max
efficiency	74	21.2973	5.7855	12	41

```

. data.unitConvert('eff', 'lp100km')
. data.summ('eff')

```

Variable	Obs	Mean	Std. Dev.	Min	Max
efficiency	74	11.8061	3.01041	5.73694	19.6012

The Command window is empty. The Properties window shows the following information:

Variables	
Variable	Label
var0	
var1	
var2	
var3	
var4	

Properties	
Variables	
Name	var0
Label	
Type	byte
Format	%8.0g
Value Label	
Notes	

Data	
Filename	temp.dta
Label	
Notes	
Variables	5
Observations	6
Size	30
Memory	64M

Command window: Command

Path: C:\Users\jfriedler\Documents\StataFiles\stata_plugins

Bottom right: CAP NUM OVR

Here's an example of unit conversion for the mpg variable. First, I've renamed the variable to efficiency, since it would be odd to have the mpg variable in units other than mpg. After renaming, the example proceeds as the last example does: first summarize, then convert, then summarize again to see that the values have been converted. Here the conversion is to liters per 100km, which I have abbreviated as "lp100km".

Resources

Stata plugins www.stata.com/plugins/

Python www.python.org

Python C API docs.python.org/3/extending/
docs.python.org/3/c-api/

Sympy sympy.org/en/index.html

Contact jrfiedler@gmail.com

17

The end.