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.

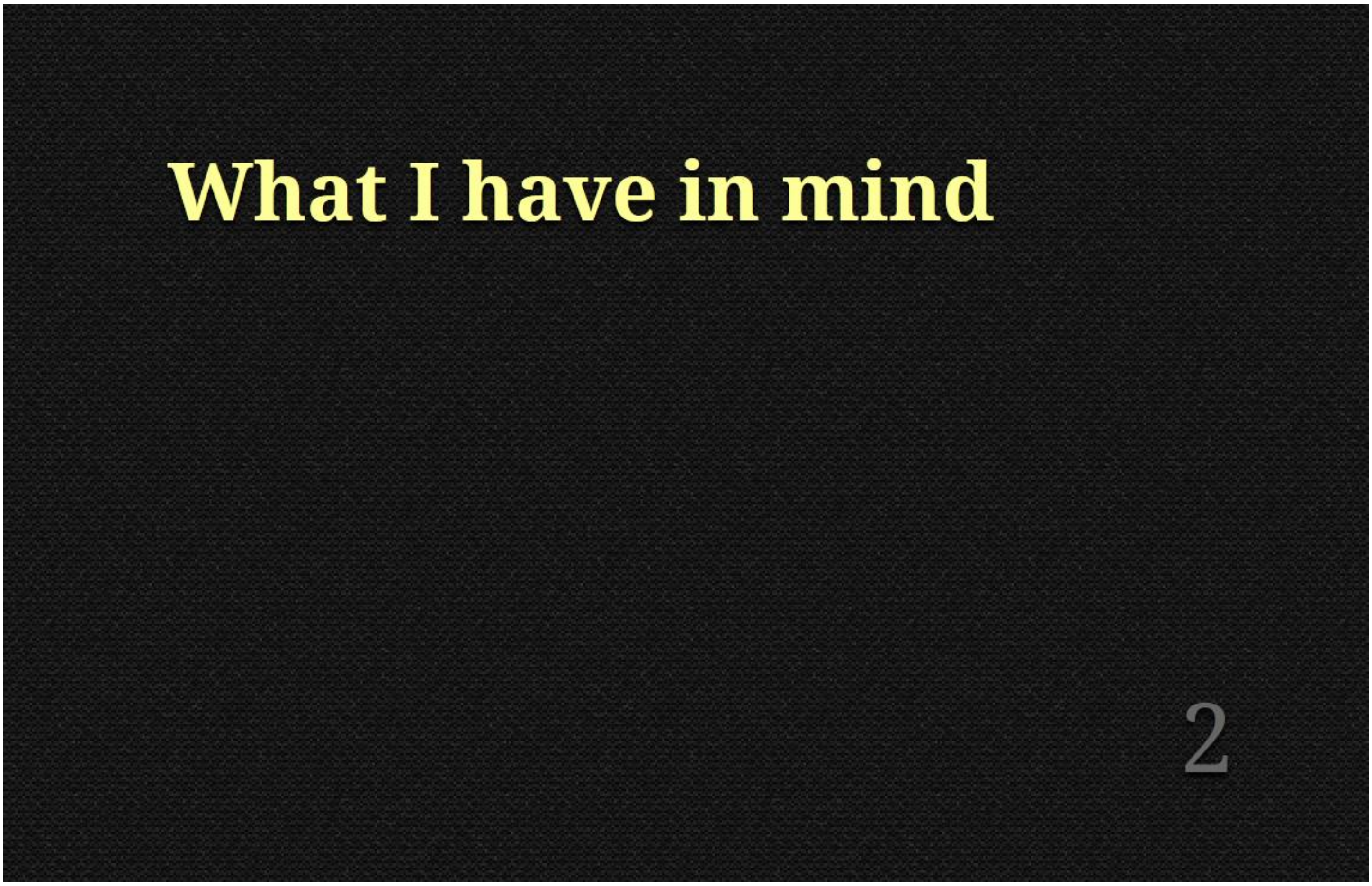
Support

where I work

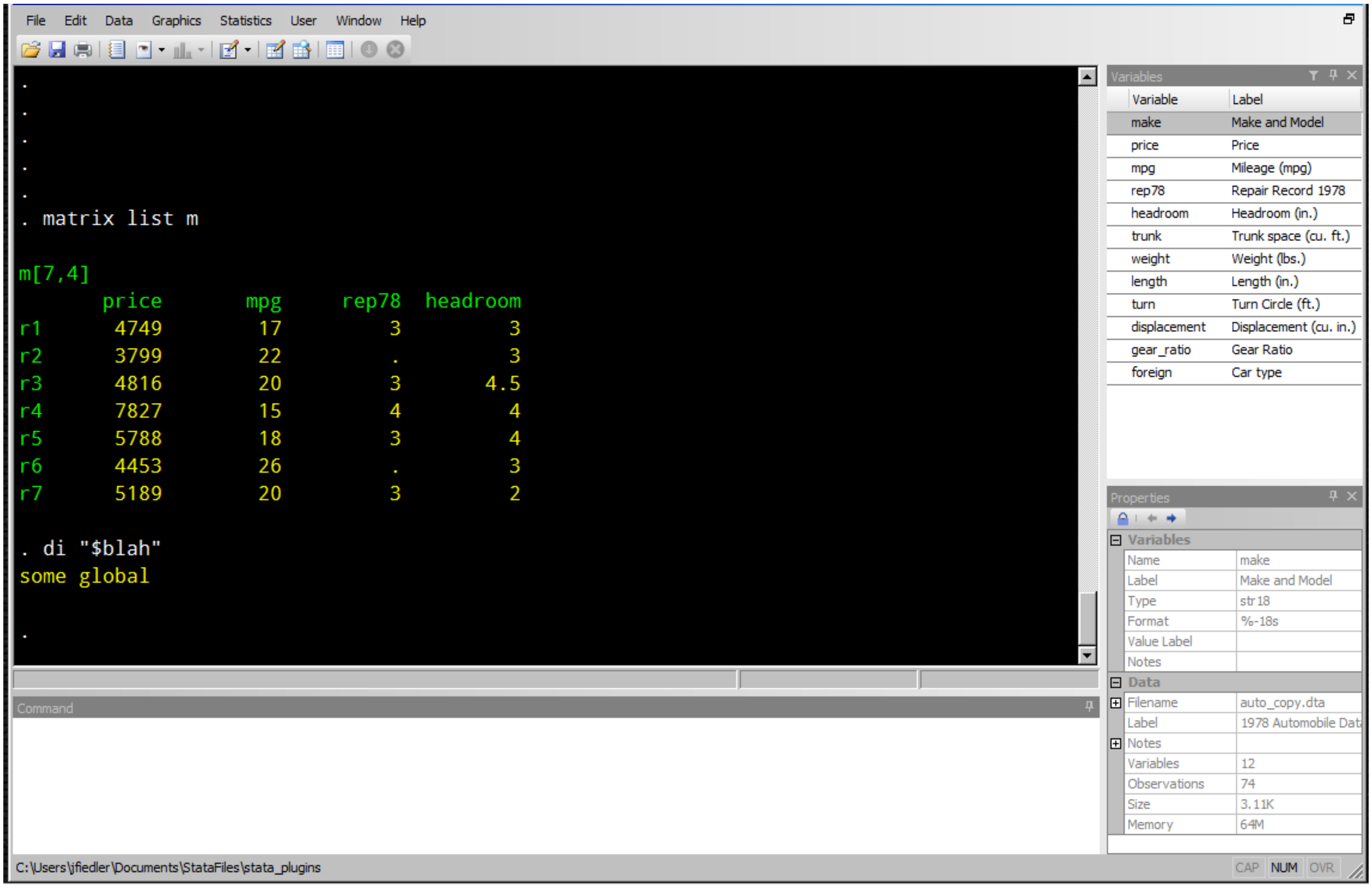
NASA Johnson Space Center

who I work for

Universities Space Research Association

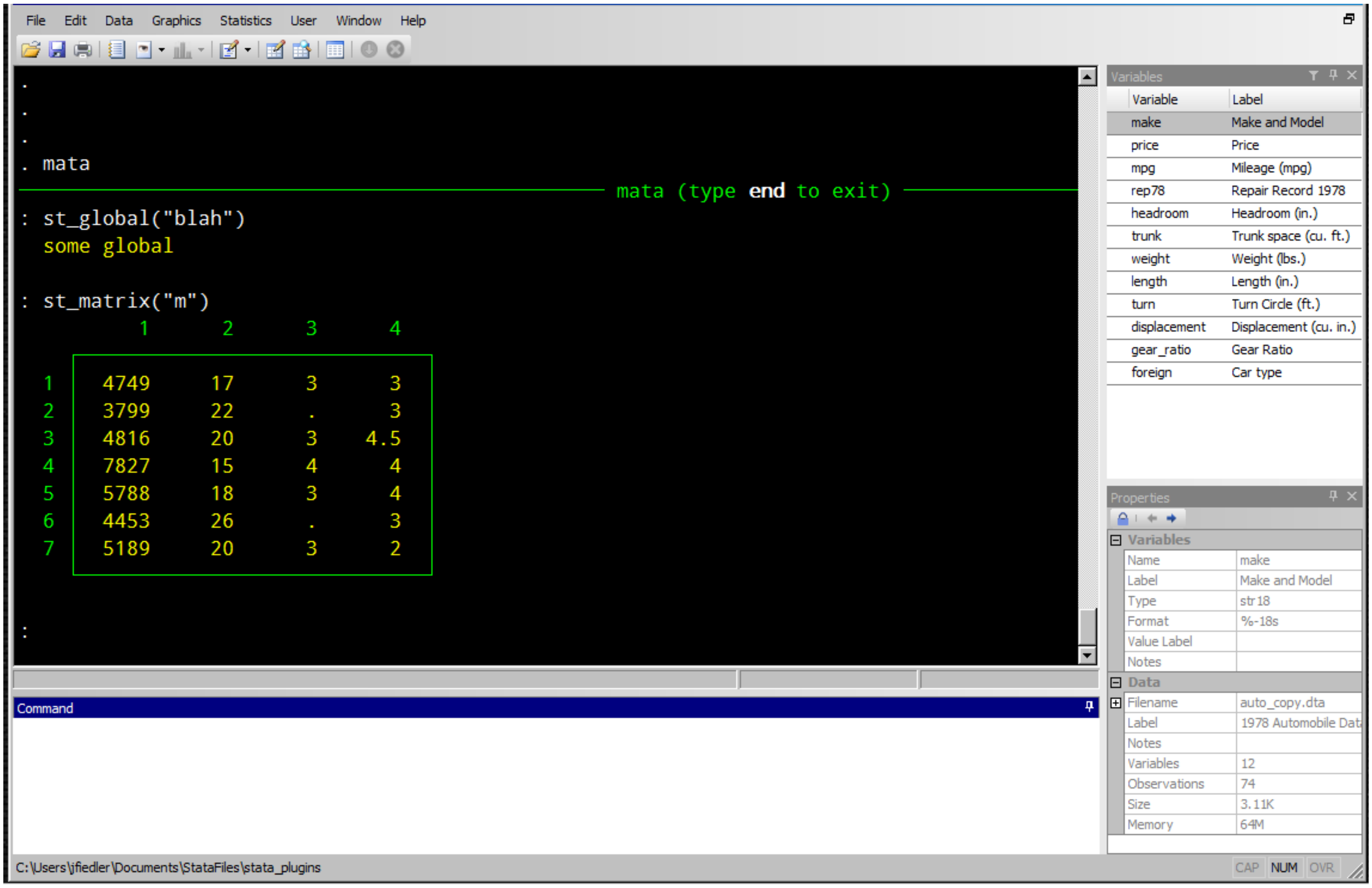


To start off, let me describe the kind of “Stata/Python combination” I have in mind for this talk.

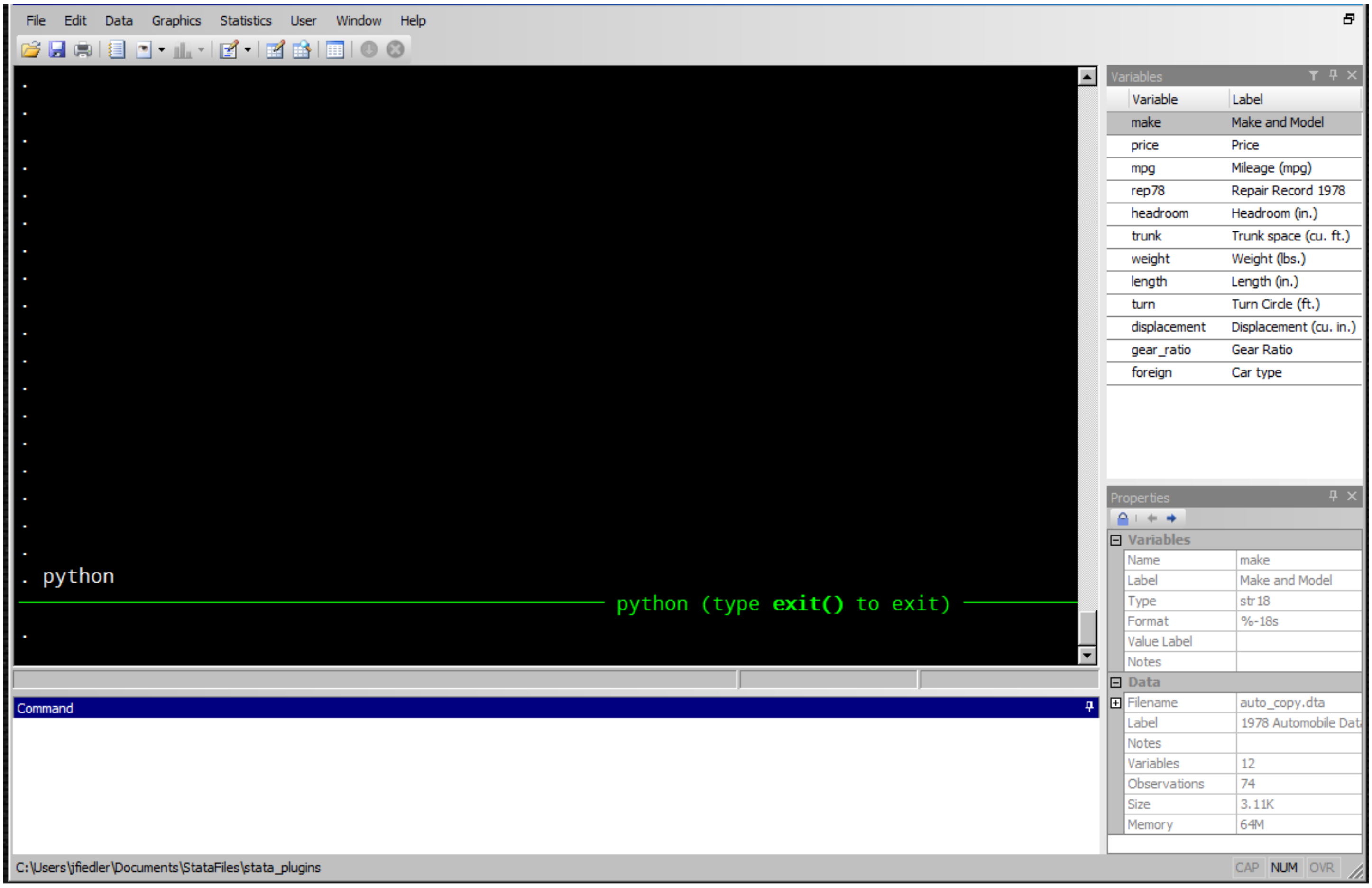


In order to have some values to work with, I have loaded a copy of the auto data set, created a matrix `m` from a subset of the data values, and created a global macro blah.

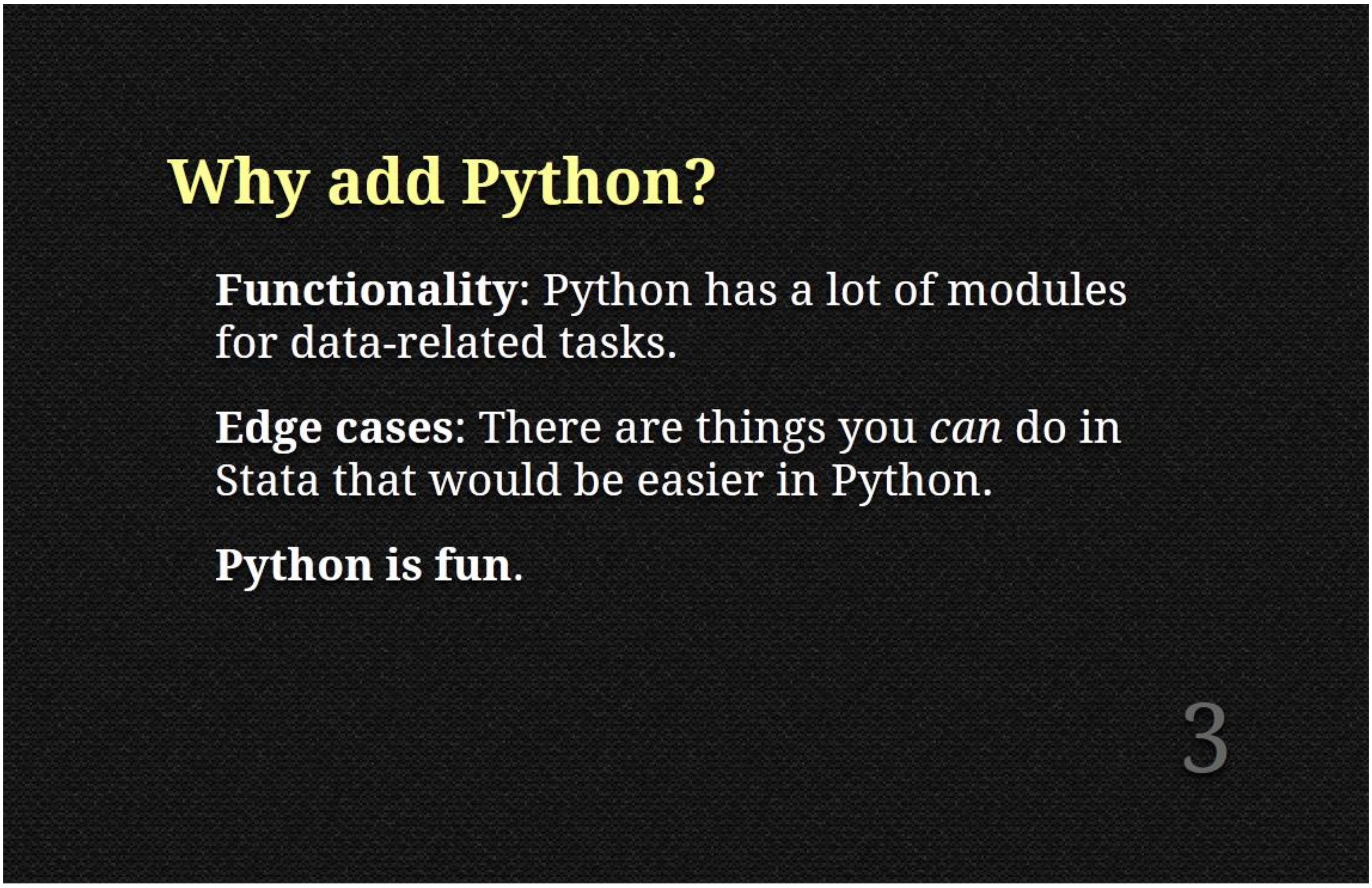
Stata has two modes of interaction. There's (what I call) the regular "Stata mode" where you use ado code (as in this slide) …



… 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.

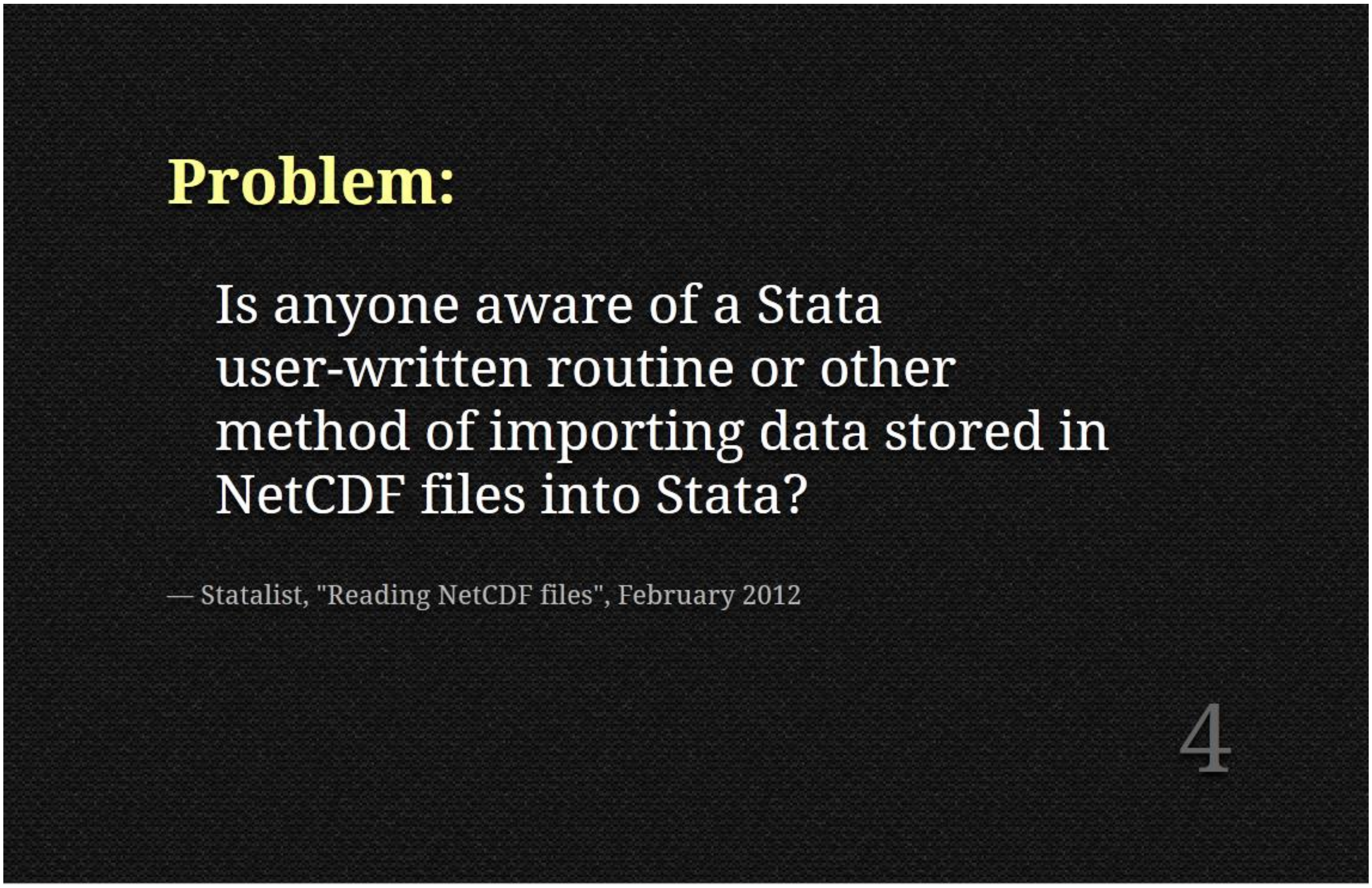


Just as there's a `mata` command, which puts the user into an interactive Mata session, there could be a `python` command which puts the user into an interactive Python session. And just as there are Mata functions for pulling Stata values into the Mata mode, there could be Python functions for pulling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



Why 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.



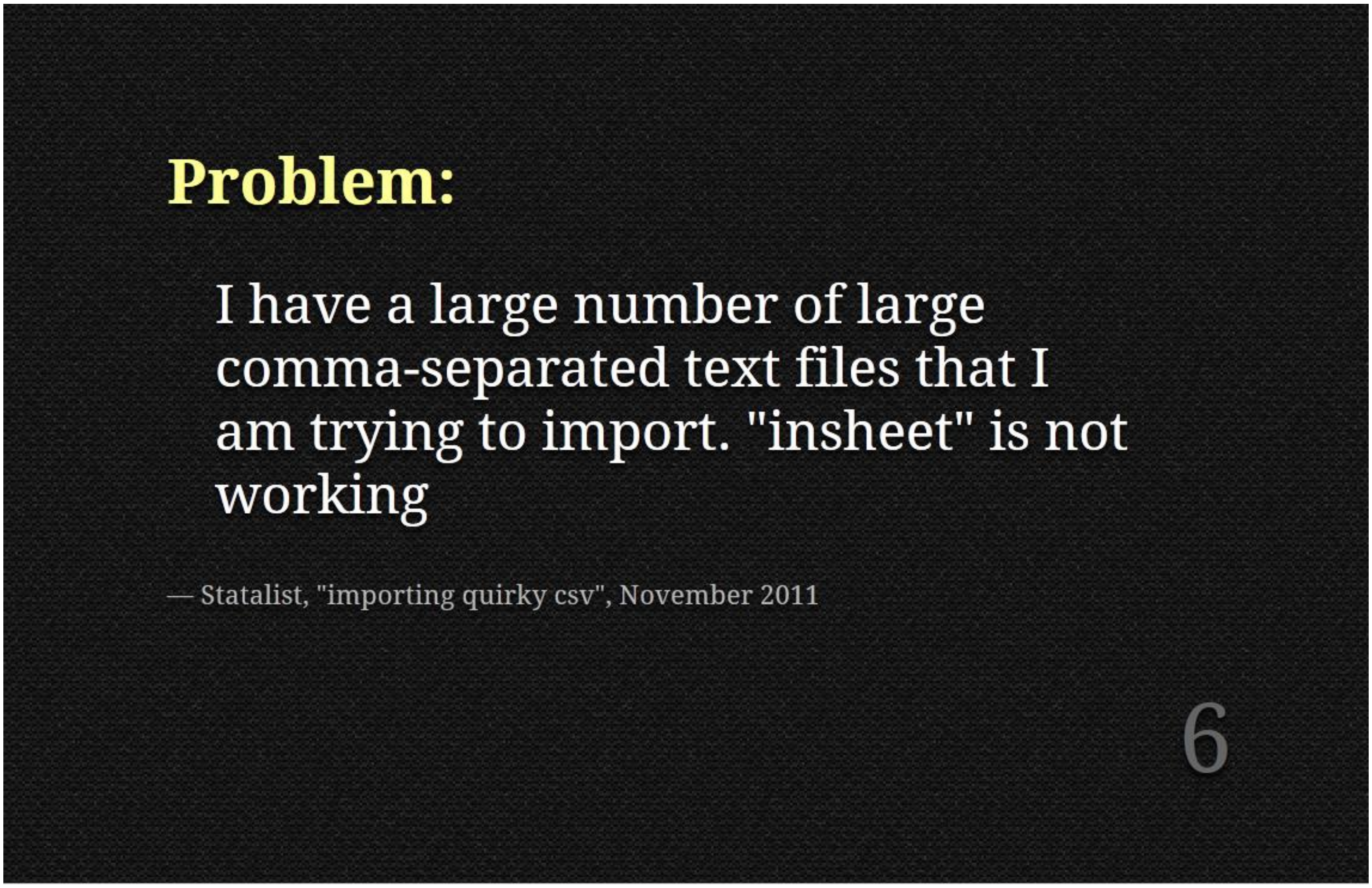
Here’s an example from Statalist where adding Python to Stata would add functionality. This person had been given a NetCDF file and was wondering whether there any commands had been written for opening such a file.

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

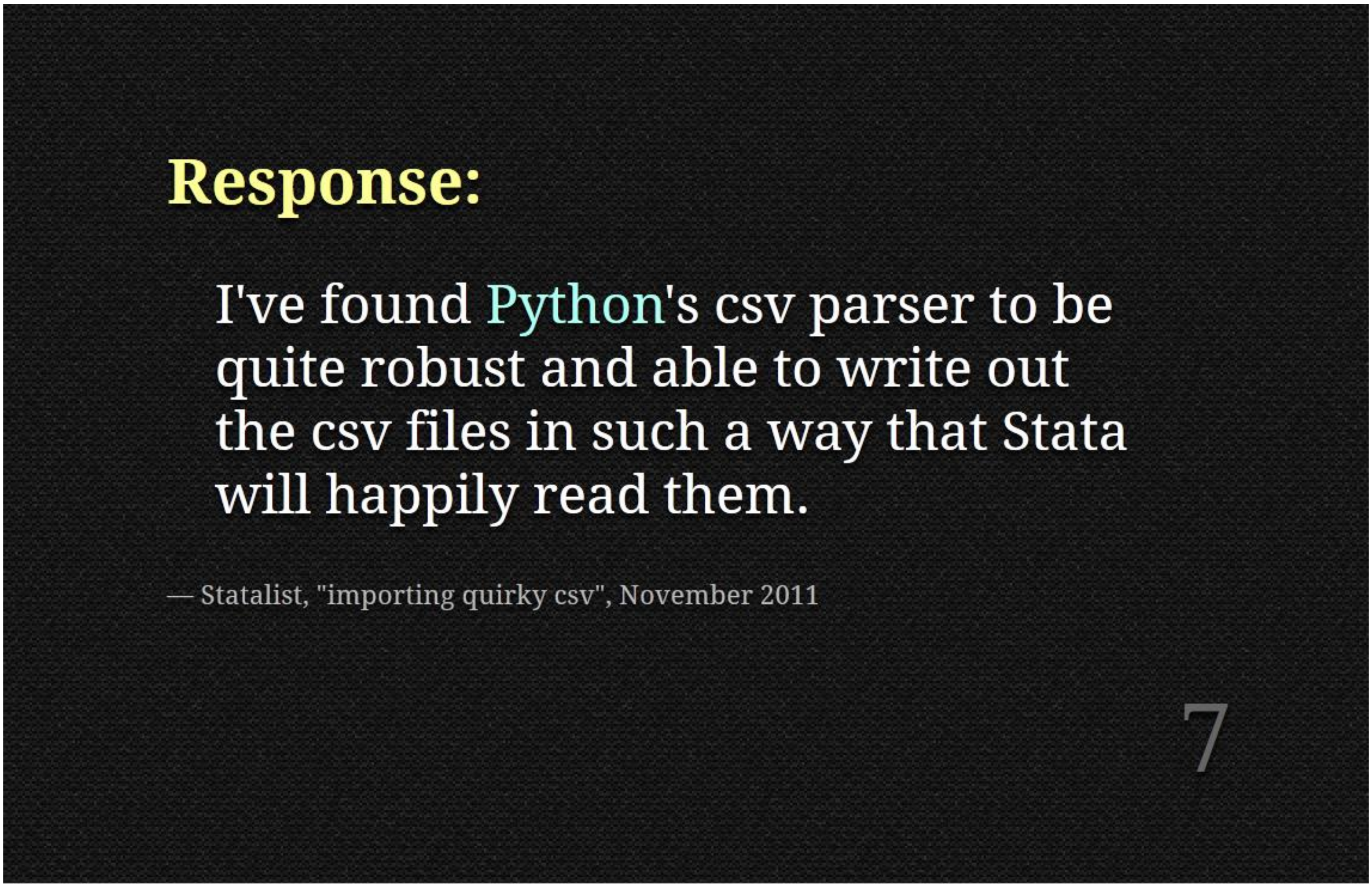
The file was handled some other way, but the user wrote back with some conclusions, one of which was:

> 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.

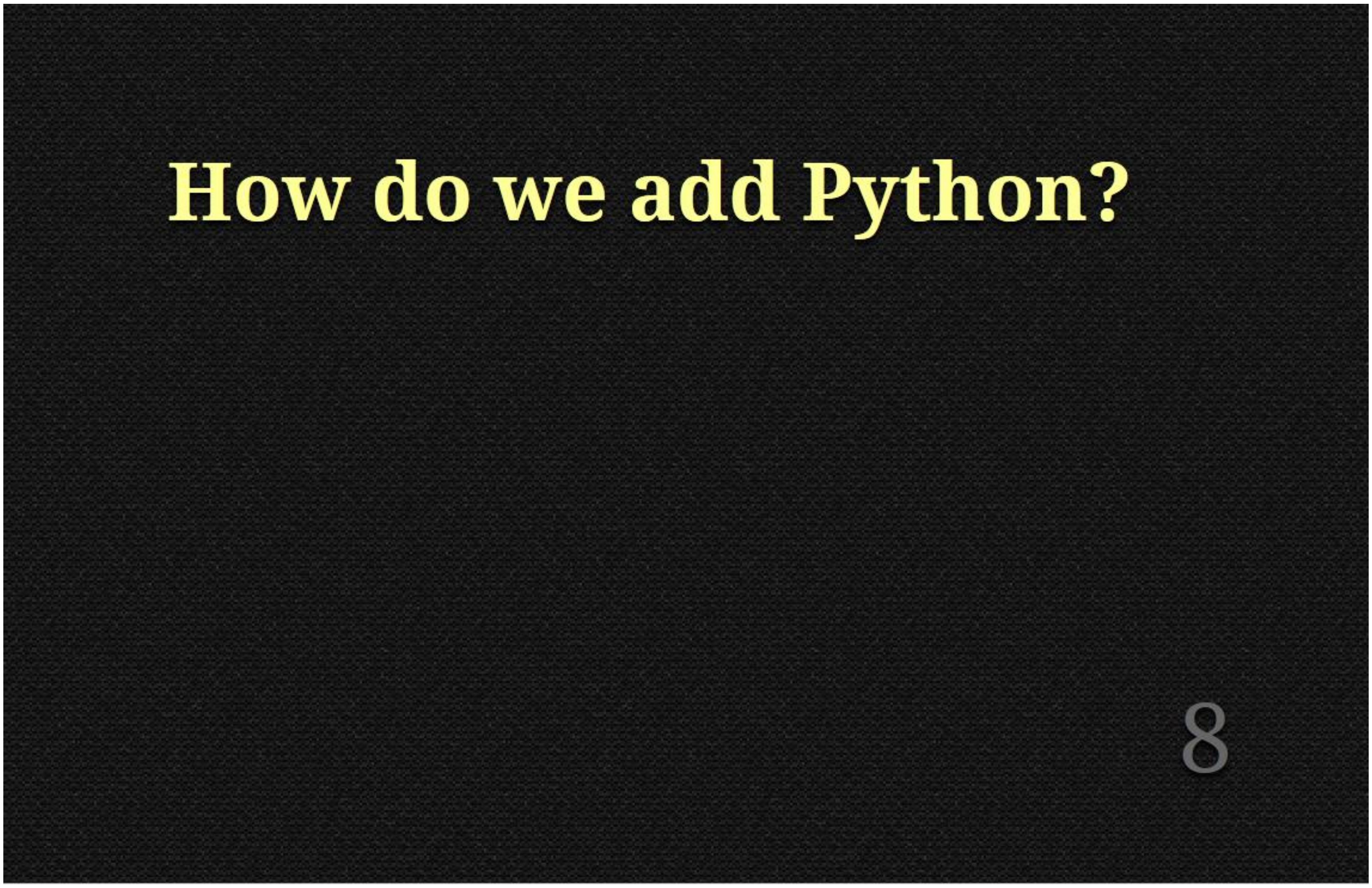
With Python added to Stata, and with a Python module to open NetCDF files, it would probably only take a few lines to get this file into Stata.



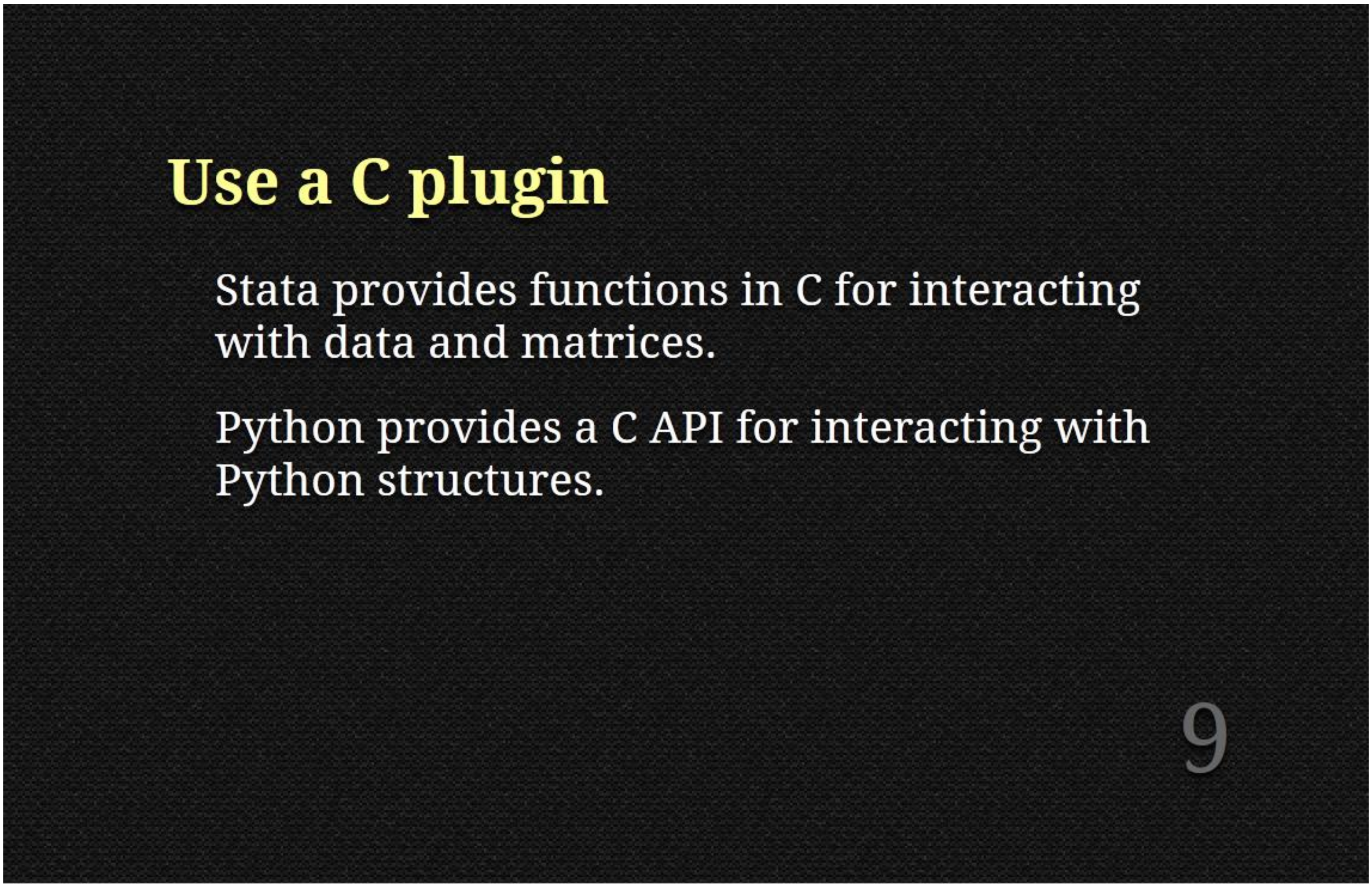
Here’s an example of an edge case, again from Statalist. This person has a CSV file, which, because of some funky formatting, can’t be imported using the `insheet` command.



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.



Now, how could we make it happen?

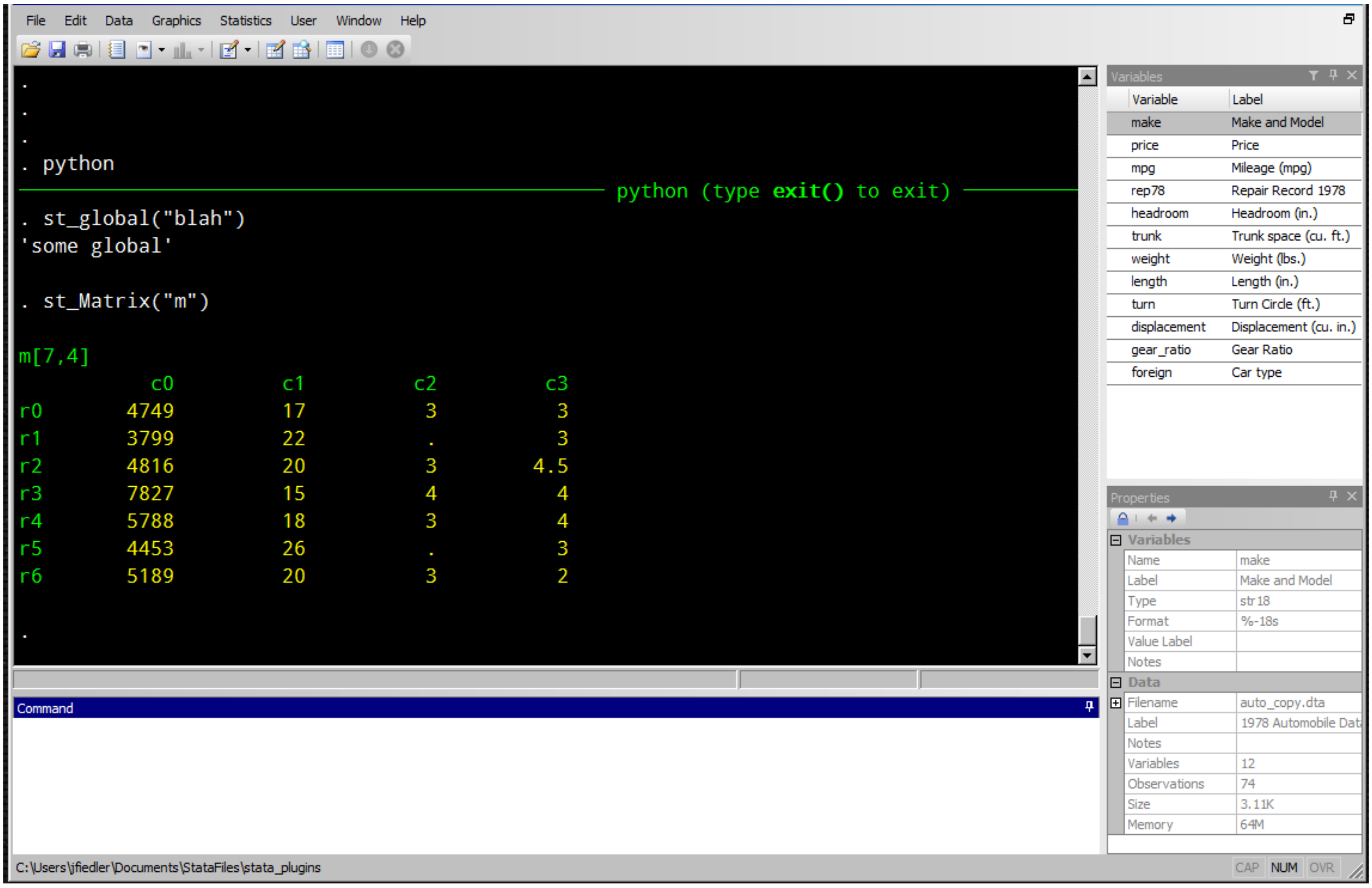


Stata, through its plugin system, provides C functions for interacting with Stata data, macros, matrices, etc. Python, through its C API (application programming interface) provides C functions for starting a Python interpreter and interacting with Python structures.

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.

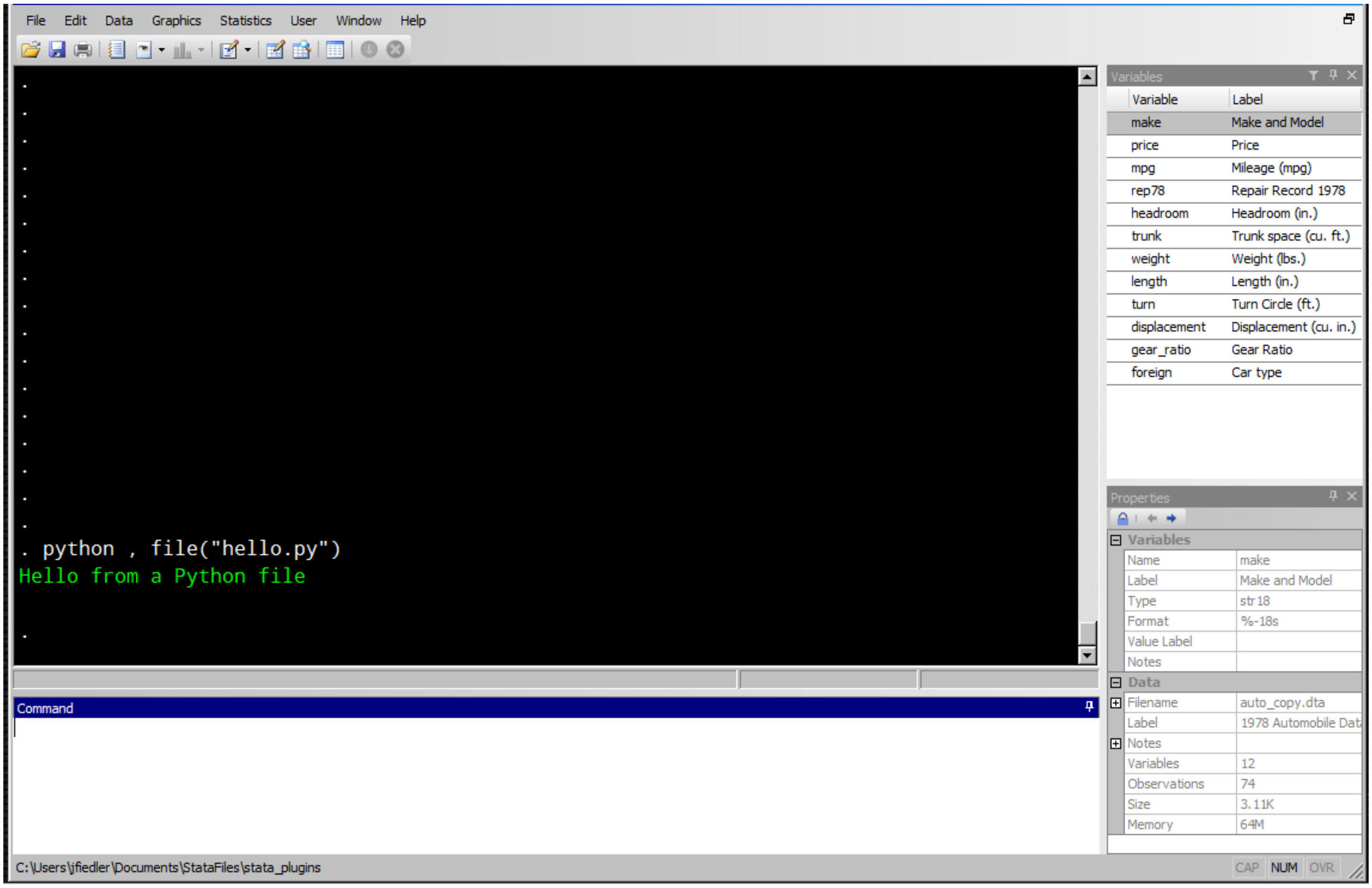
Before I begin a demonstration, I should probably make it clear that this isn't a simulation. Last year I used an imitation of the Stata GUI to demonstrate some ideas for changes to the interface. This year there's no imitation Stata, and no imitation Python. This is a real instance of Stata, and I am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

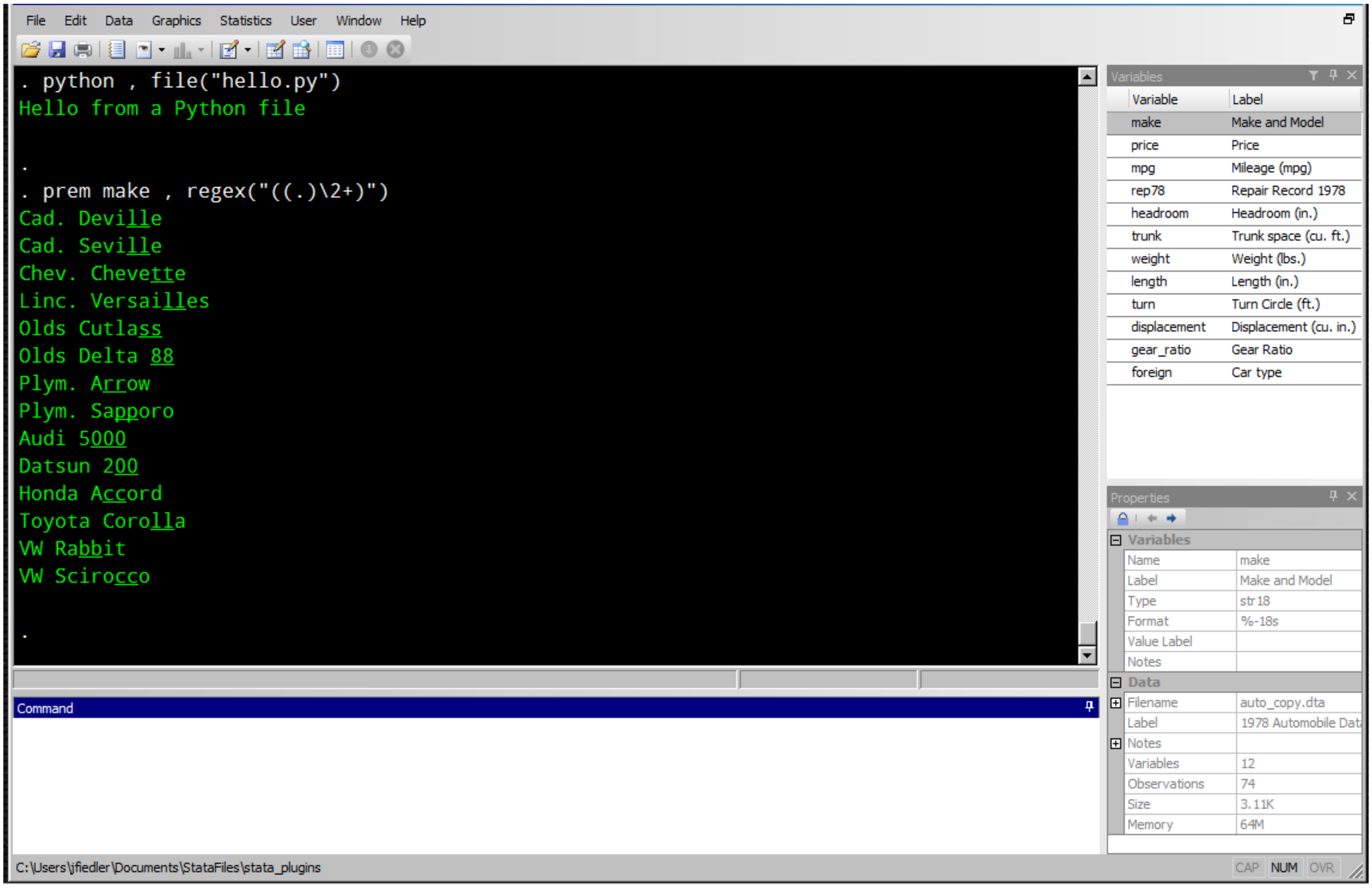
In Mata, there are functions for pulling Stata values in as Mata values. Stata's plugin functions provide some of the same functionality as those Mata functions, so I can also make Python functions that pull Stata values into Python.

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:

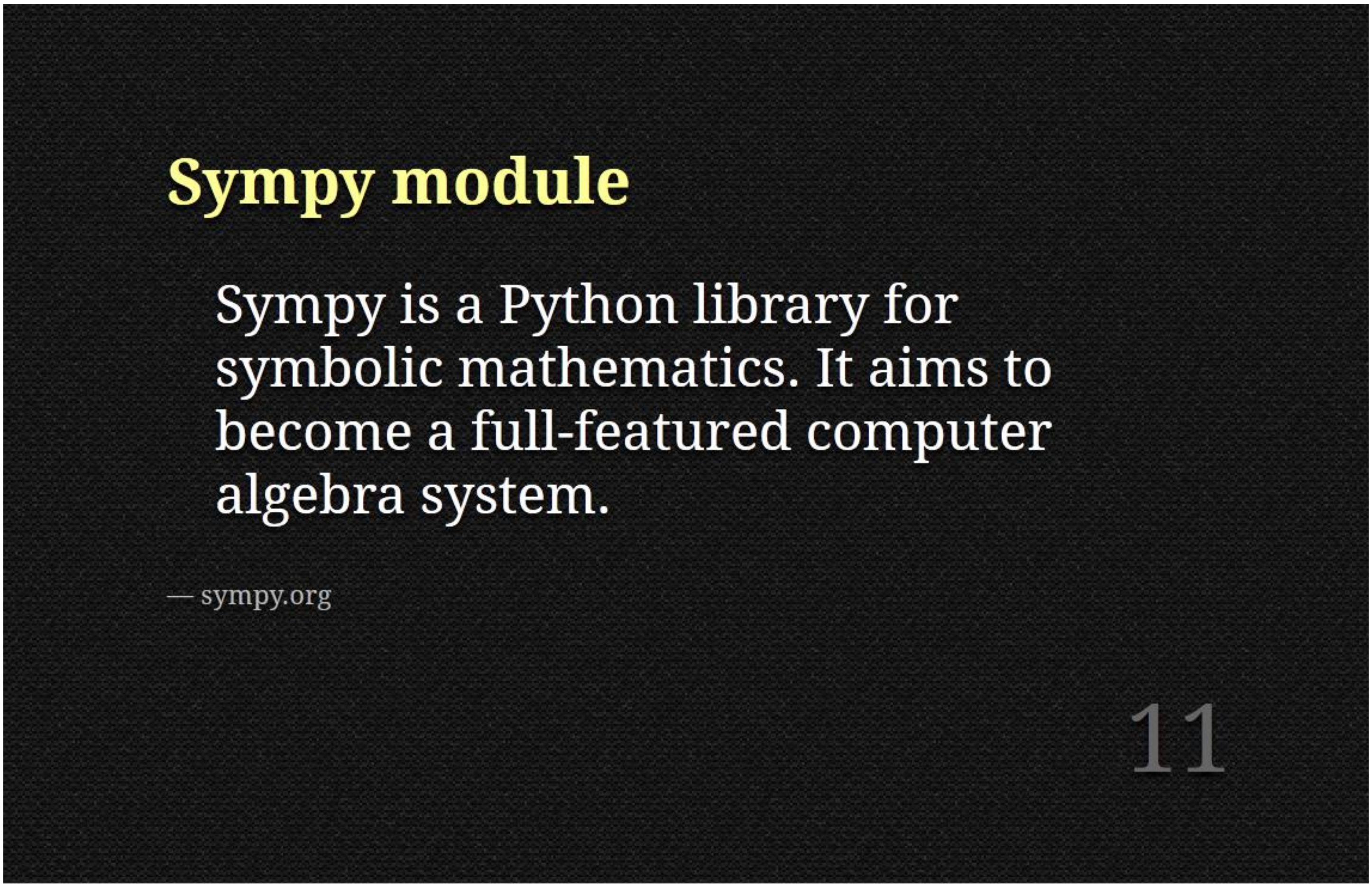
```
print("Hello from a Python file")
```



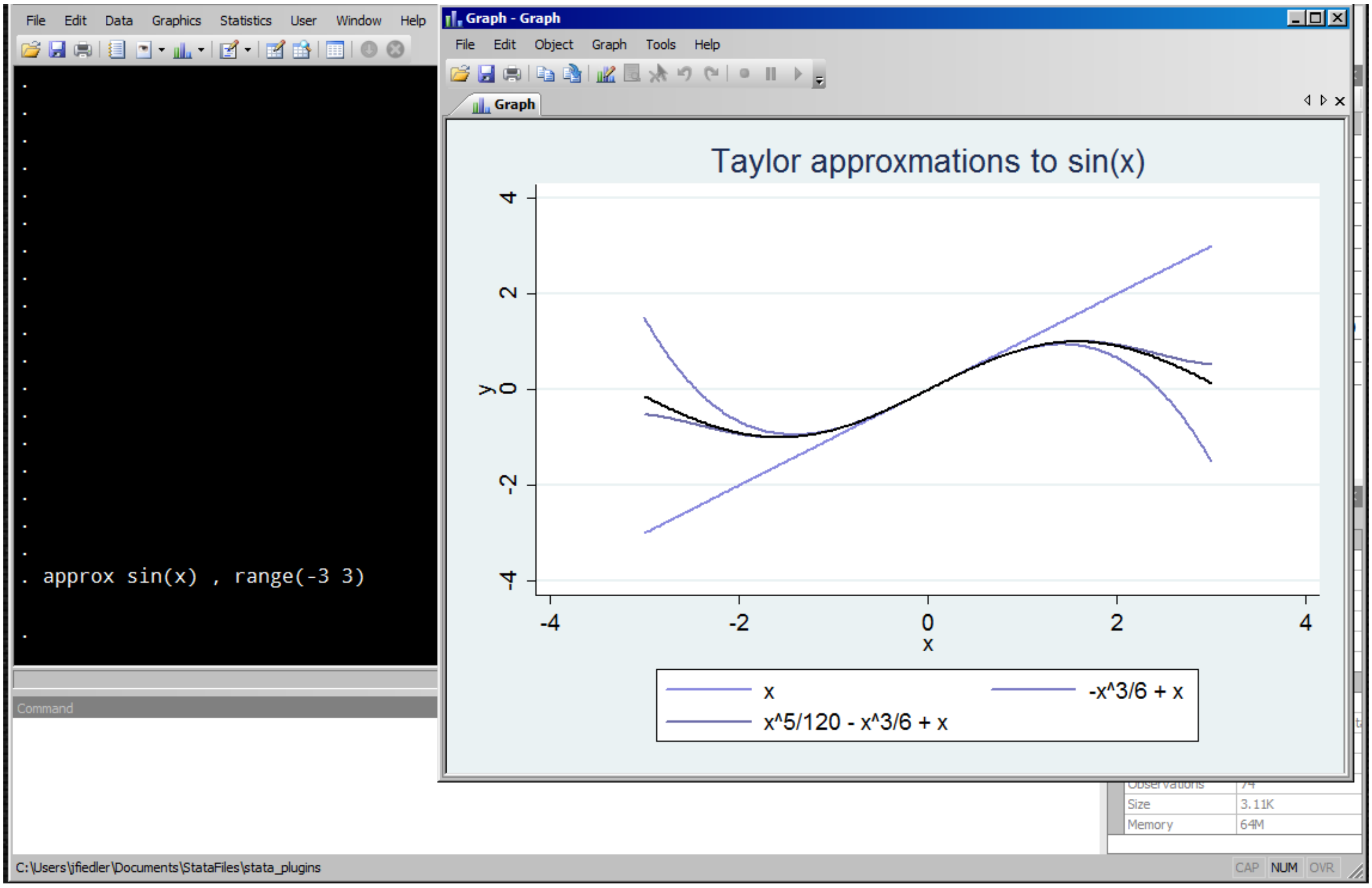
Here's an example using regular expressions. Many Stata users rarely if ever use regular expressions, and for those that do, Stata's regular expressions are probably sufficient for most uses. Occasionally, though, some Stata users want to do something more complicated than what Stata's regular expressions 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".

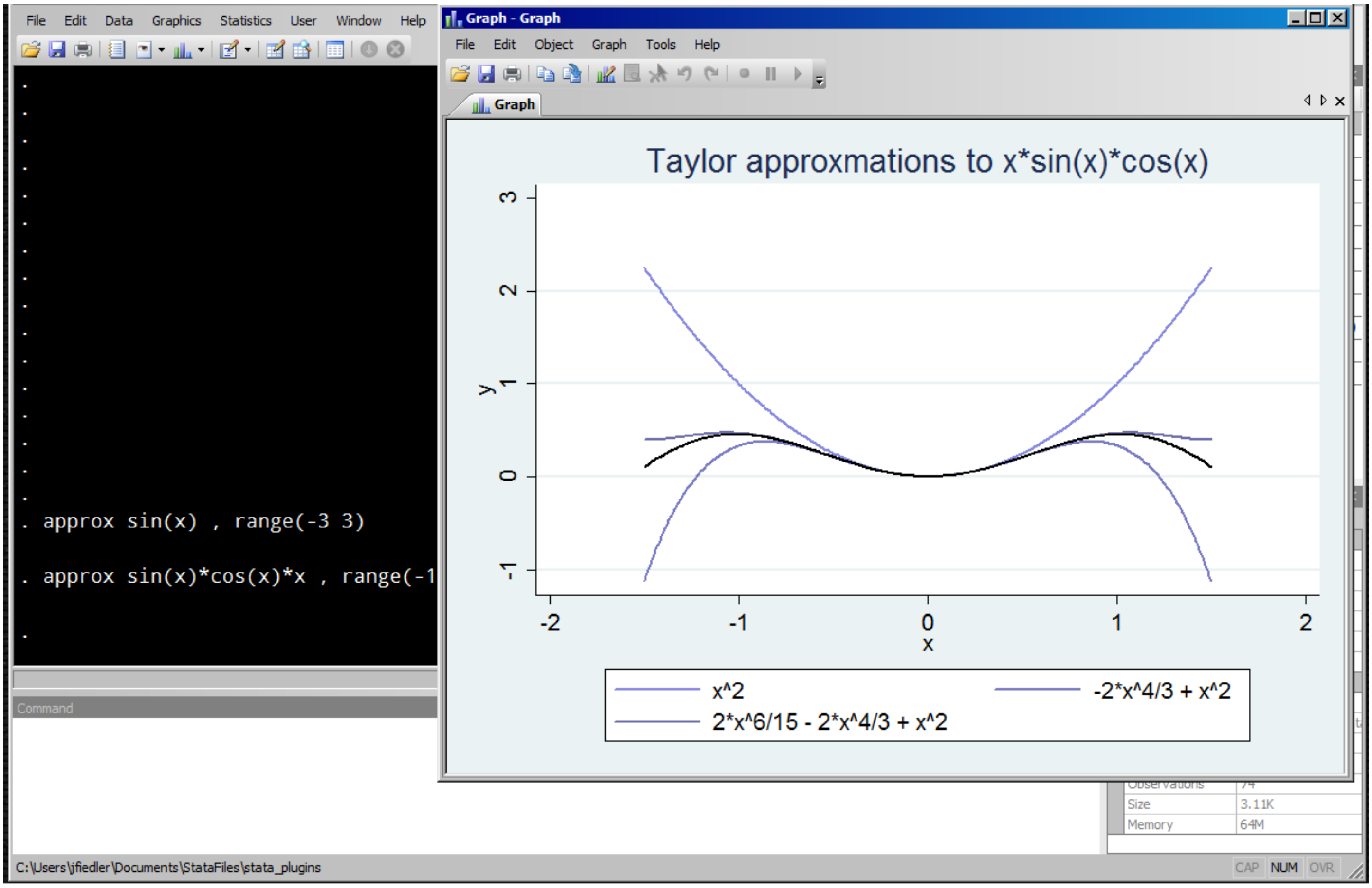
This example also demonstrates how Python files and .ado files can work together seamlessly. Thus, users could benefit from having Python without having to know any Python themselves.



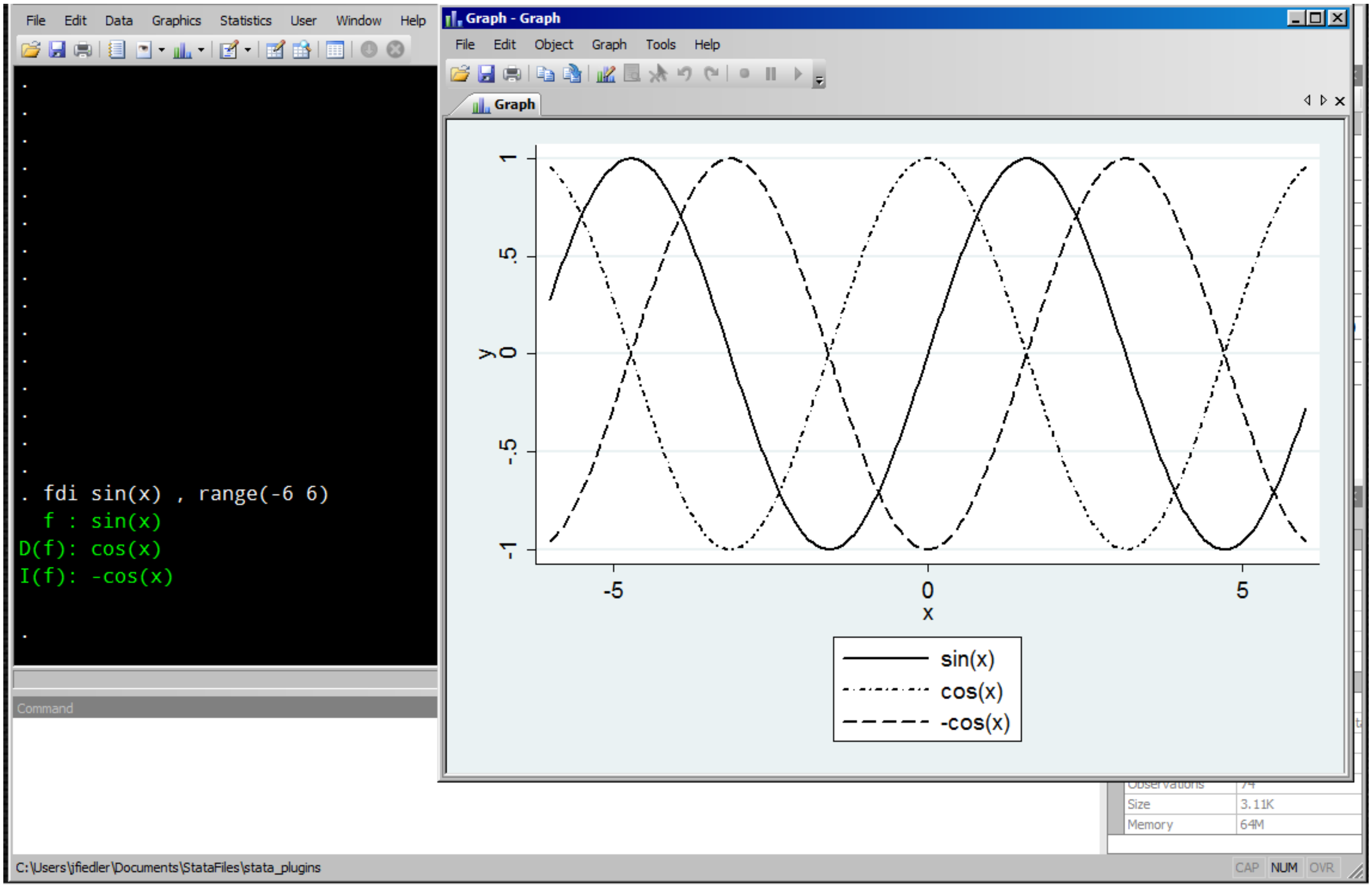
Next I'll show a couple examples with graphs. For these I'll be using a symbolic math module called Sympy. With Sympy you can do things like take the limit of a function, factorize polynomials, and get the Taylor series expansion of a function.



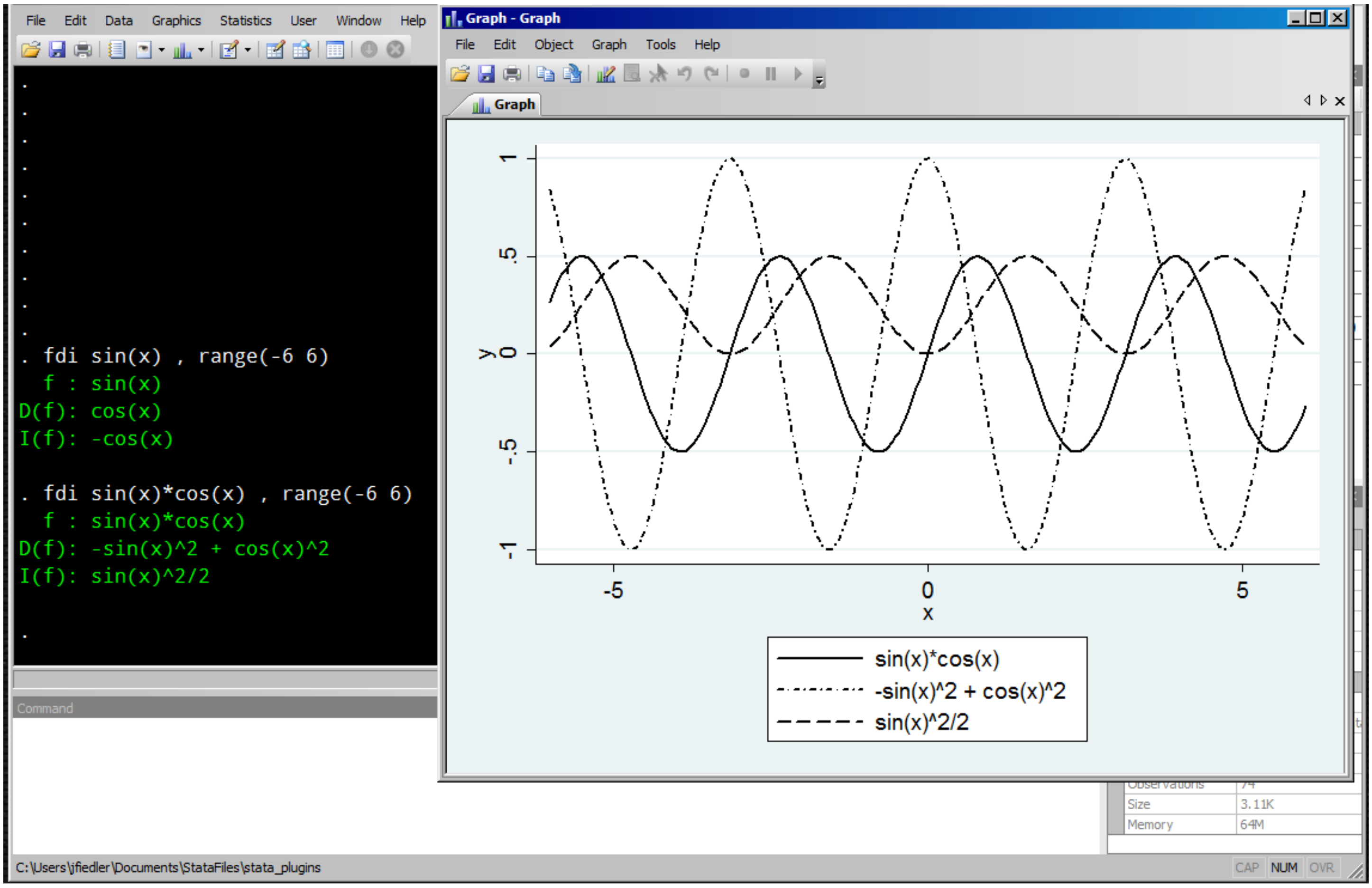
Here I've written a command called `approx`, which takes a function of *x* as its argument, and graphs the function together with Taylor approximations. In the graph, the thick black line represents sin(*x*) and the blue lines show the first three Taylor approximations.



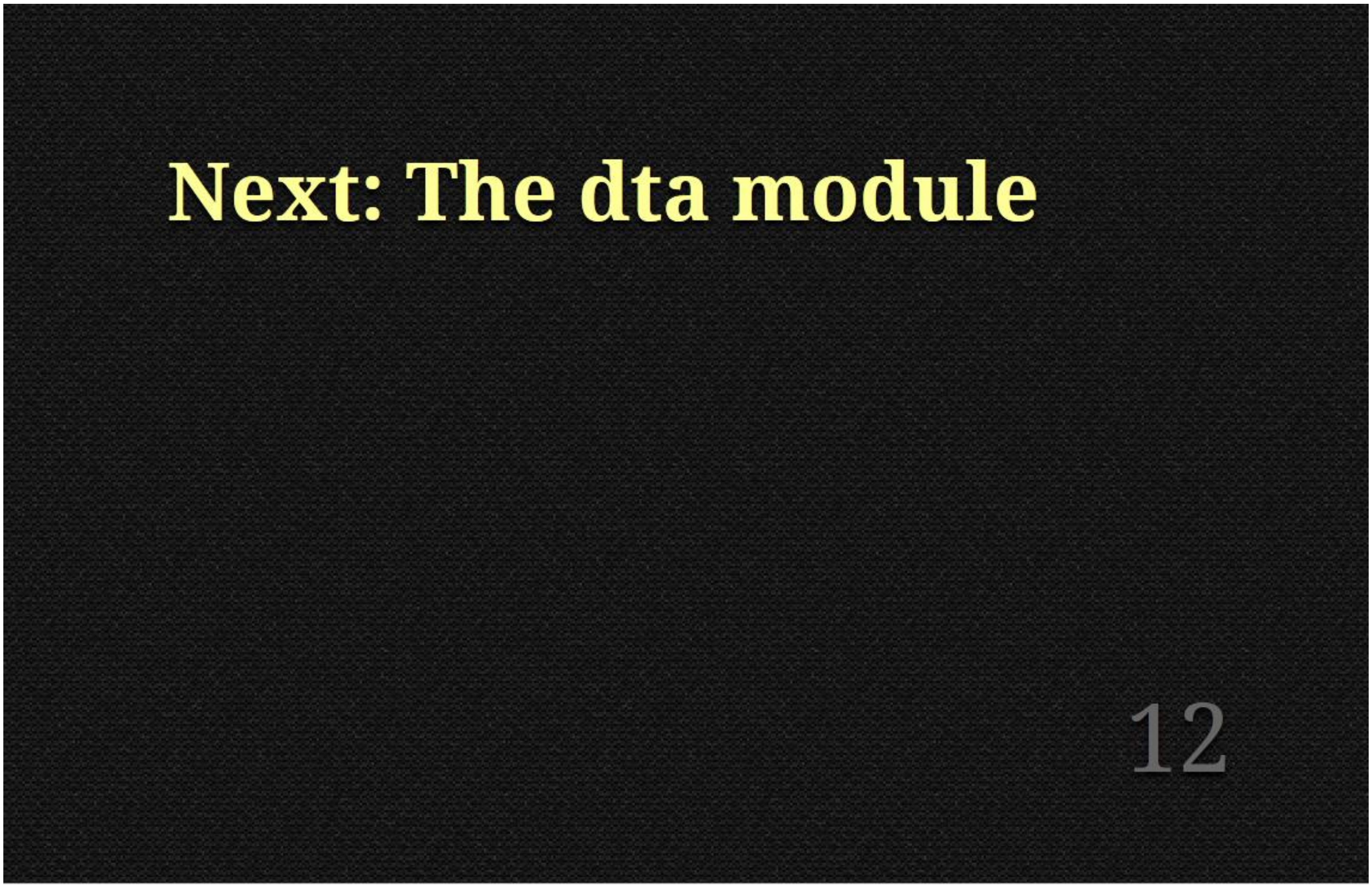
Here’s another example for with a more complicated function.



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.

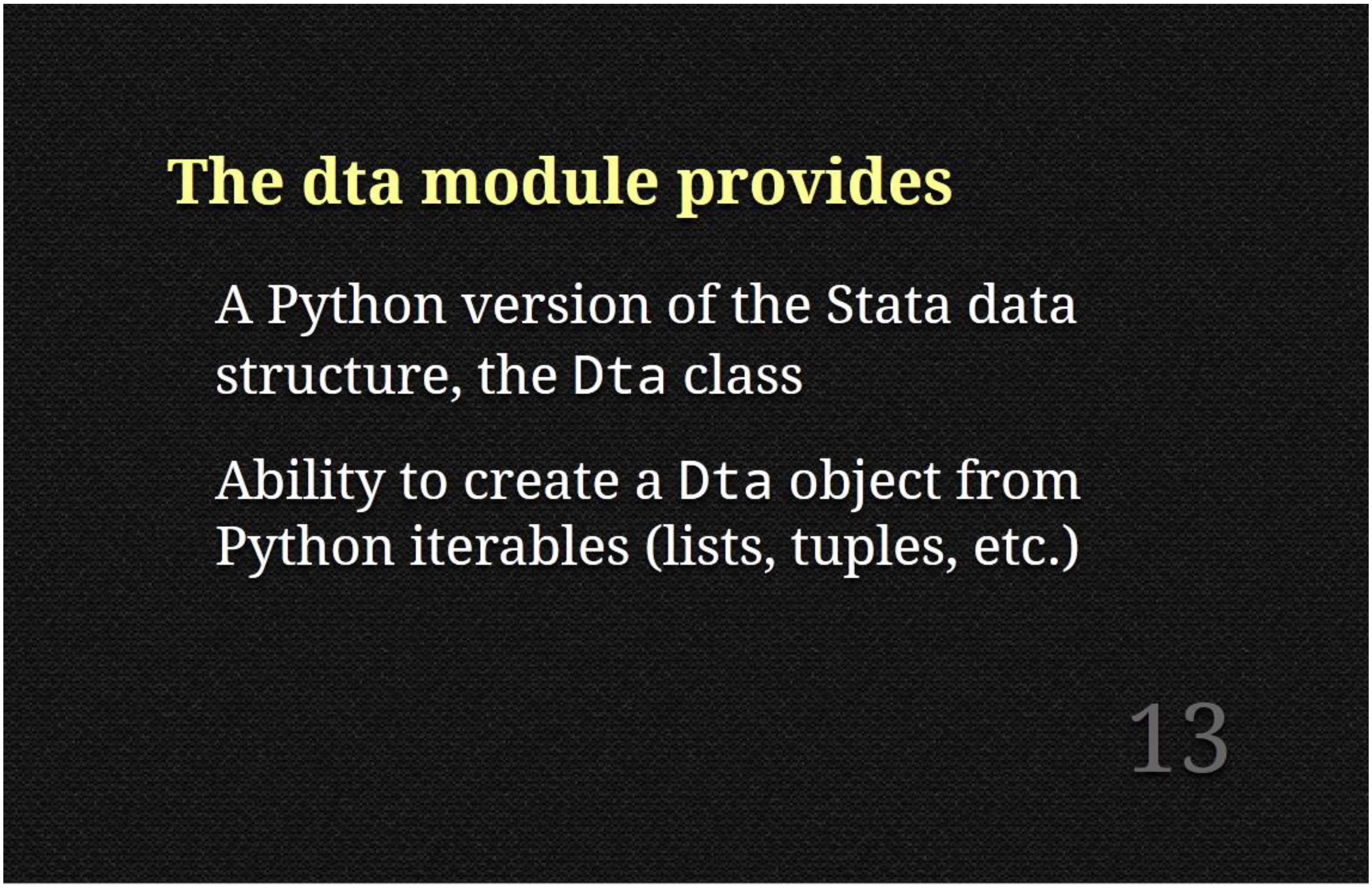


Here's another example, with a more complicated function.



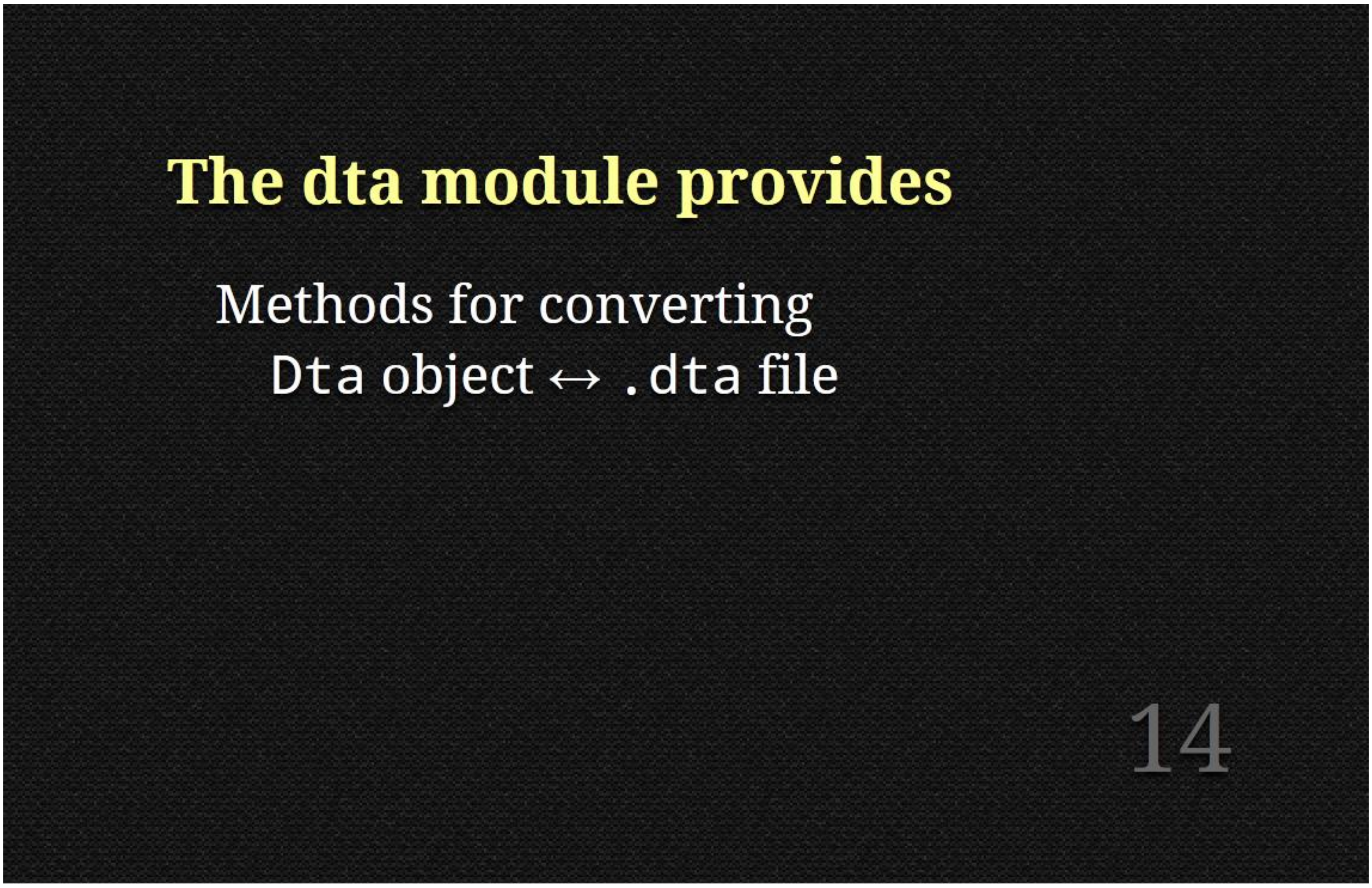
Looking ahead and imagining using the Python plugin, I would guess that one of common things people would do is it to build or modify datasets. They might use Python to build their datasets from scratch or by importing, e.g., CSV files, or they might want to open and modify existing Stata .dta files. 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.



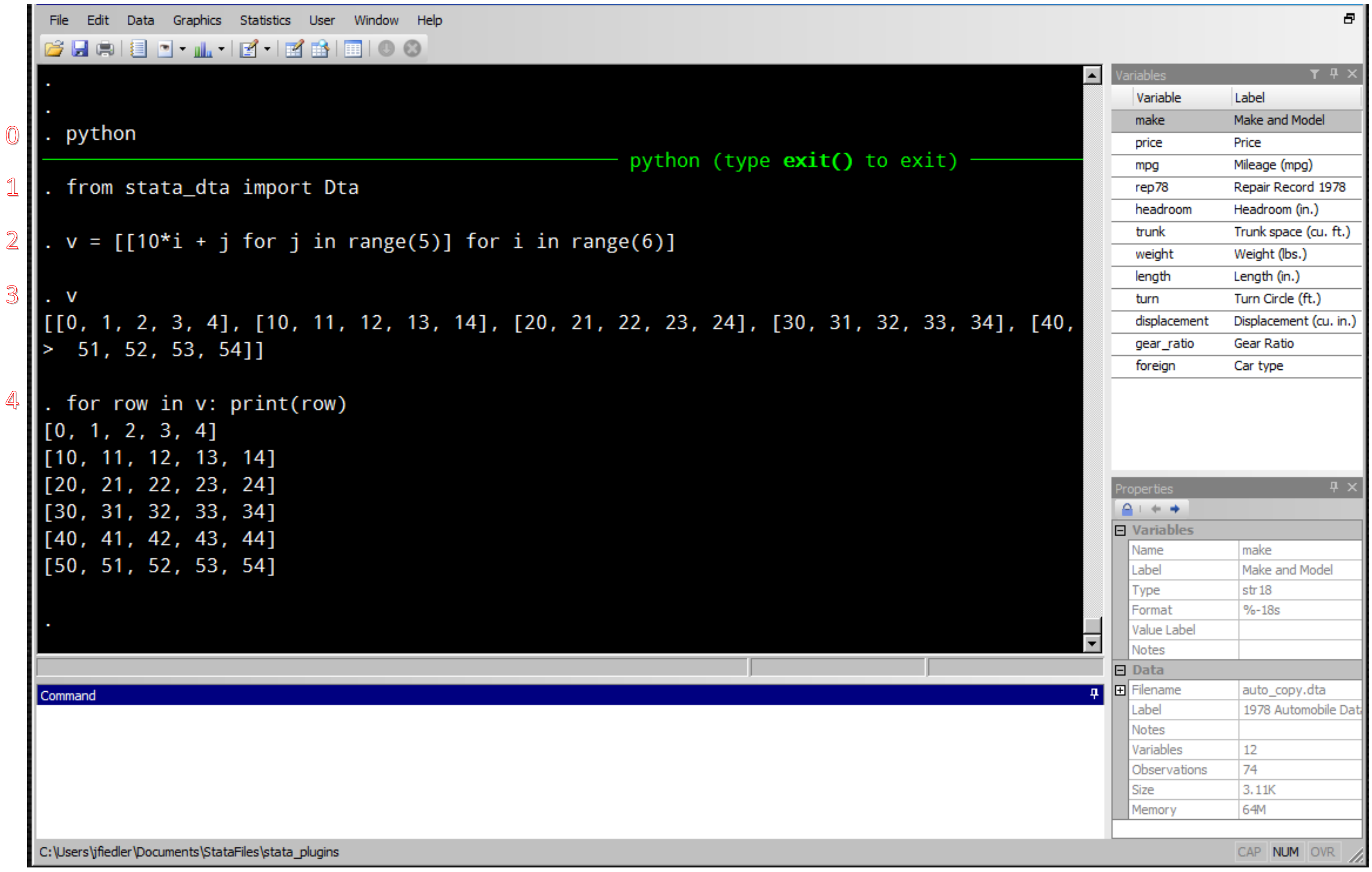
The module includes a complete Python version of the Stata data structure, which includes everything that a .dta file would include.

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



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



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):

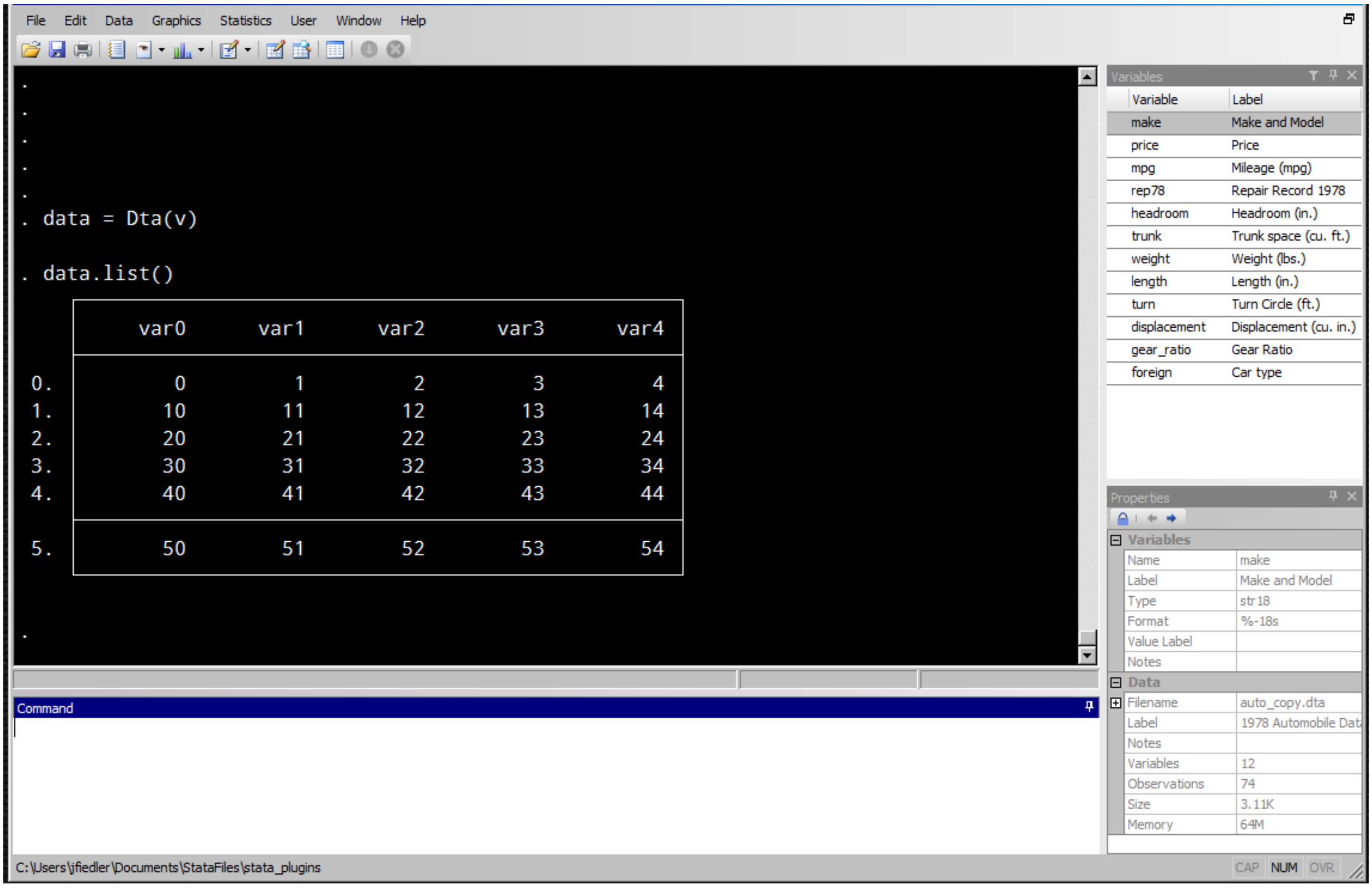
0. Start the Python interactive mode

1. The “dta module” is actually called `stata_dta`. This line bring the `Dta` class constructor (i.e., dataset constructor) into the interactive session.

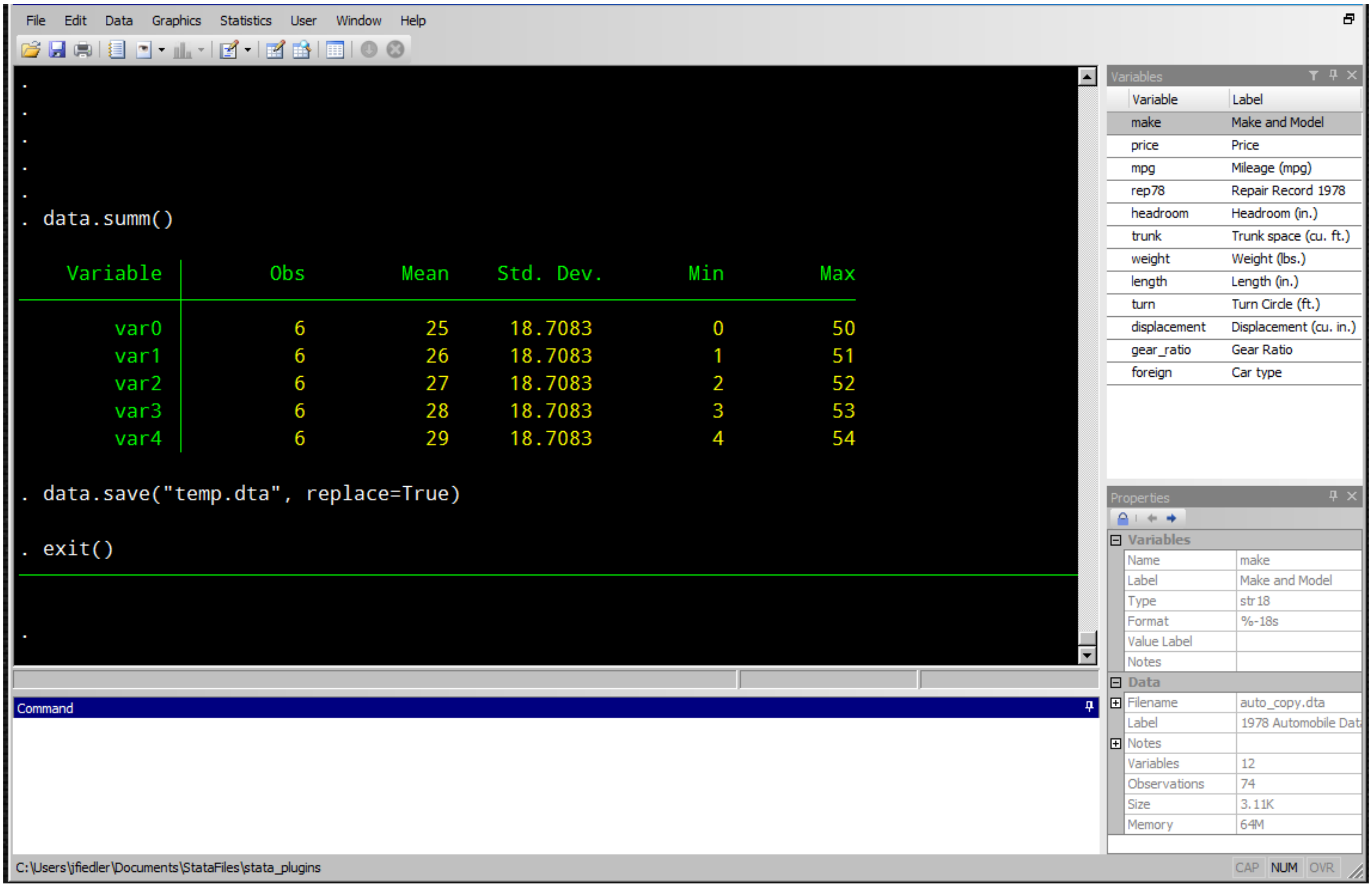
2. This line creates an array of arrays of values called `v`.

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.

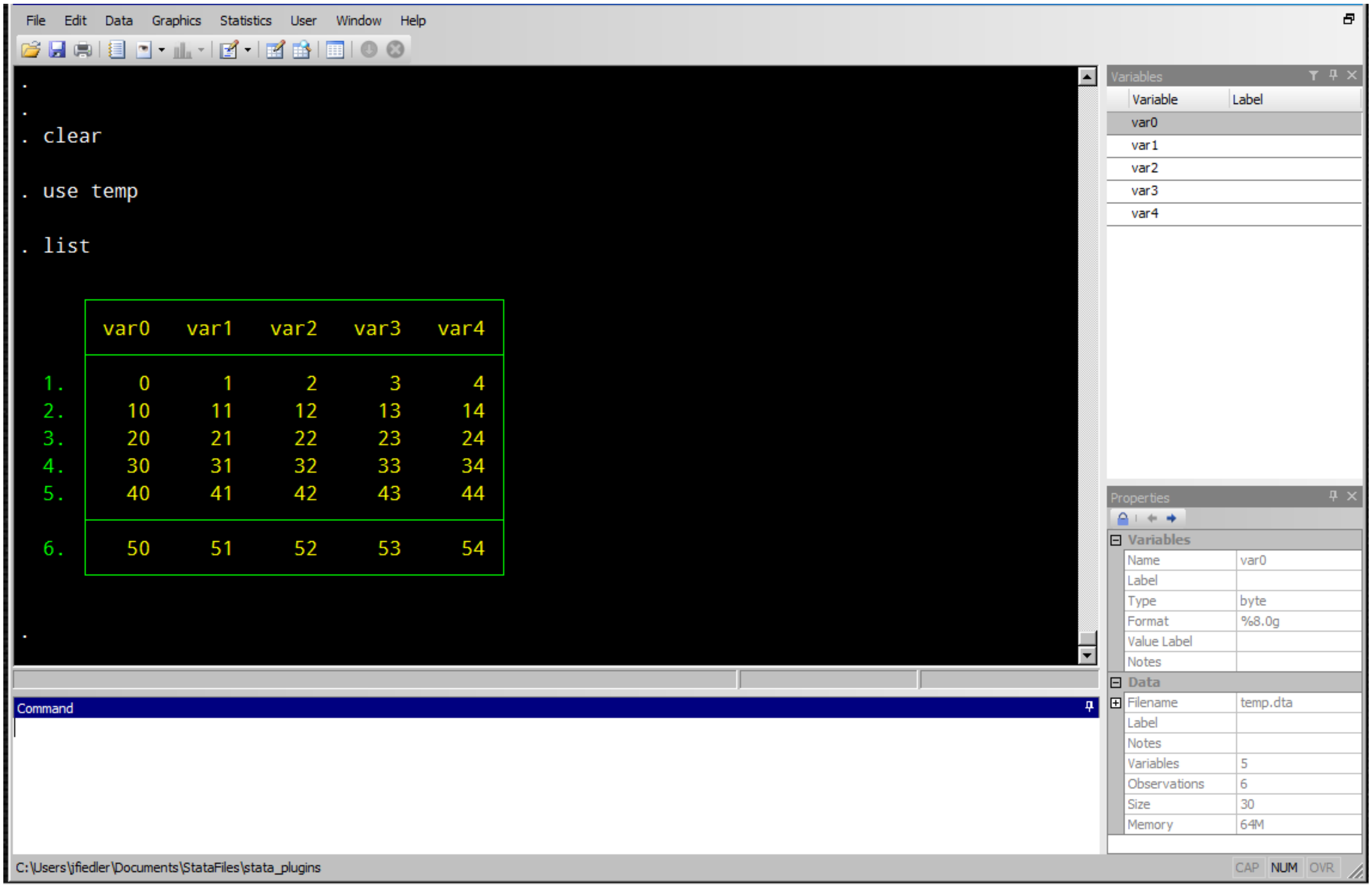
4. Another way of looking at what’s in `v`. This stacks the inner arrays to make it easier to visualize `v` as rows within a dataset.



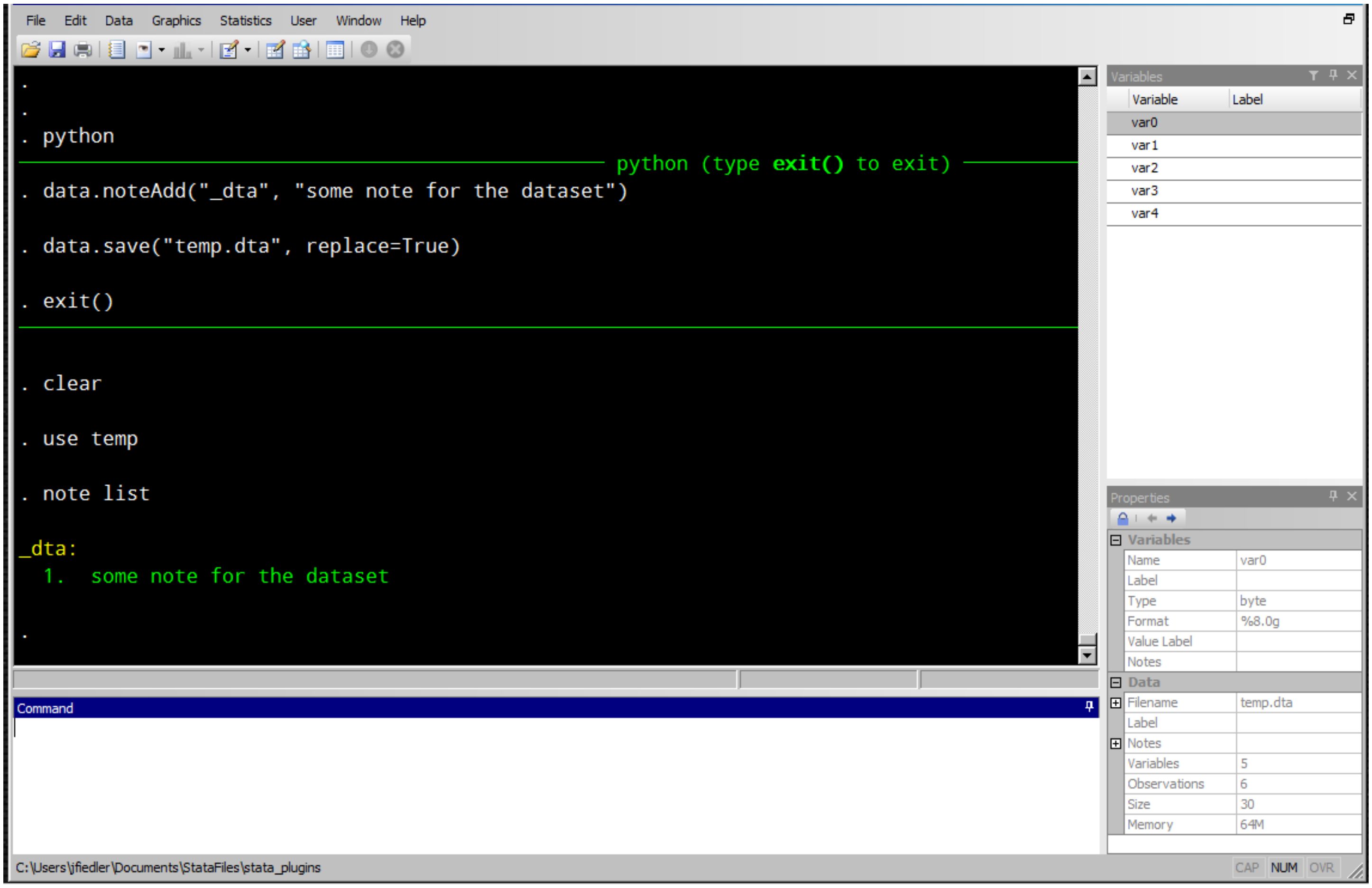
A dataset is created from `v` just by calling `Dta` with `v` as argument. To see the values in a dataset, use the `list()` method.



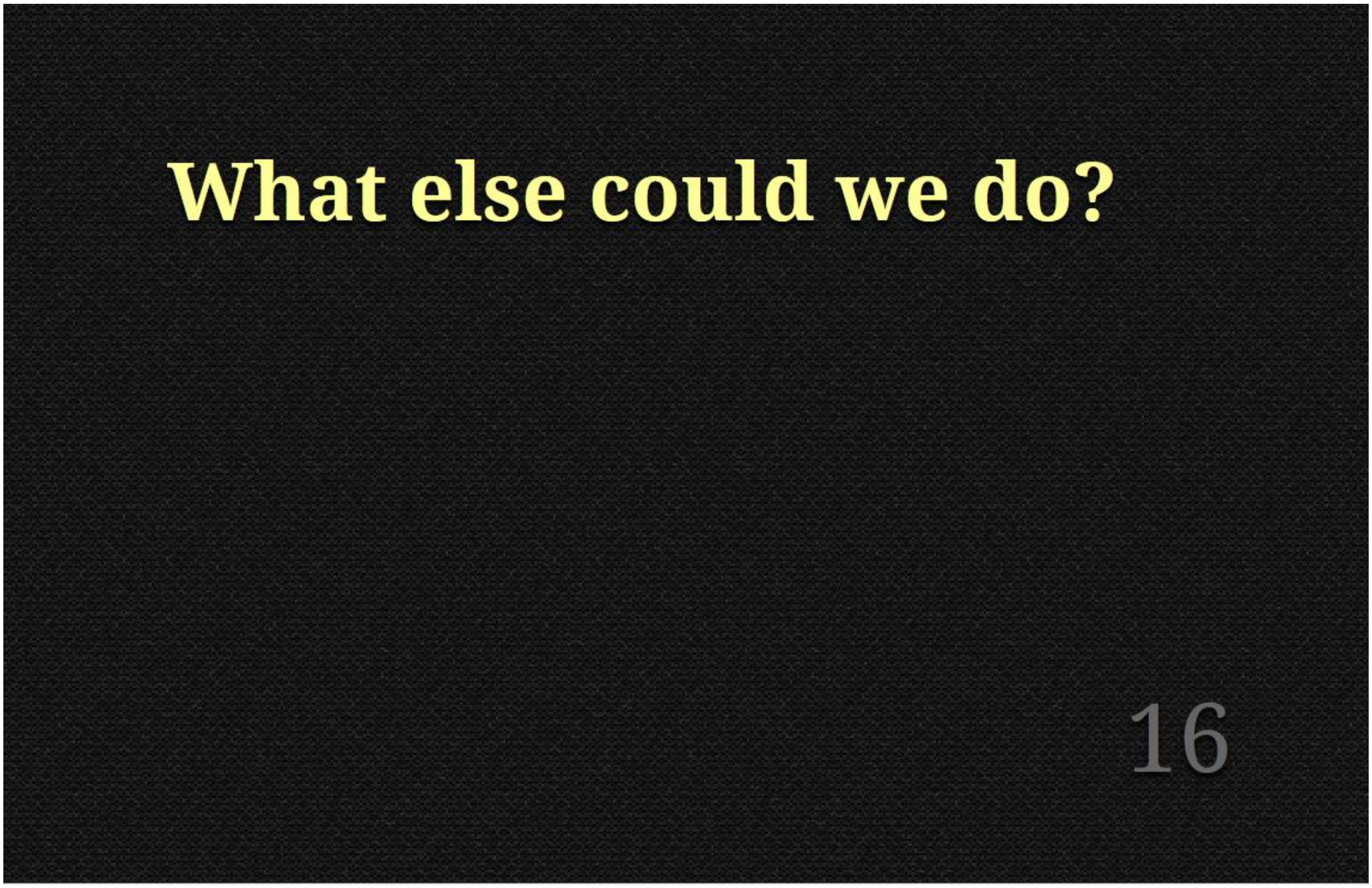
We can also summarize the data, and once we're satisfied with it, we can save the dataset as a `dta` file. Above, the optional `replace` argument is used because there's already a `temp.dta` file in this directory. In the last line above, `exit()` is used to return to Stata's default mode.



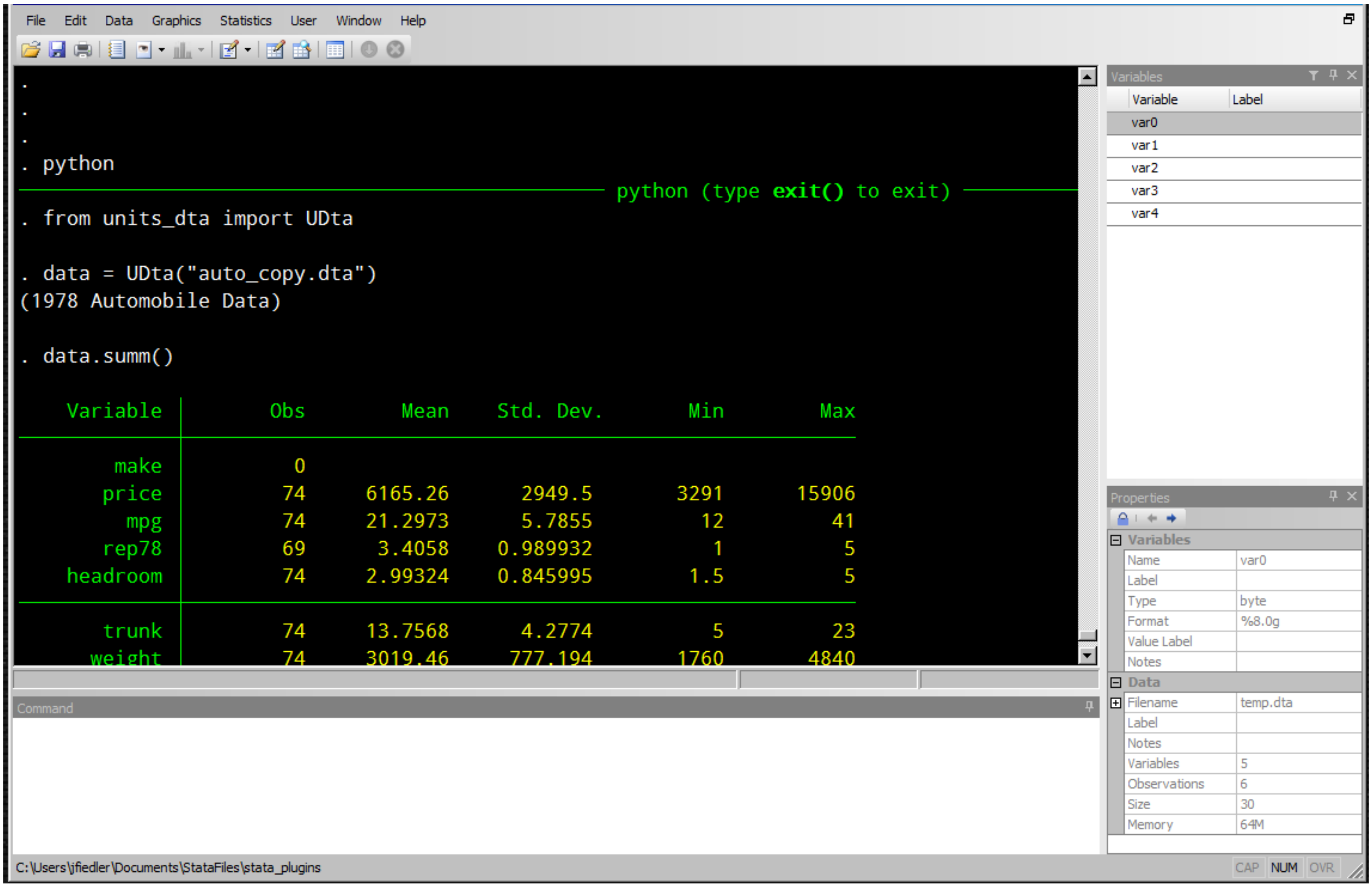
Now in Stata mode, the inputs above clear the existing dataset (the auto dataset was loaded), loads `temp.dta`, and lists the values.



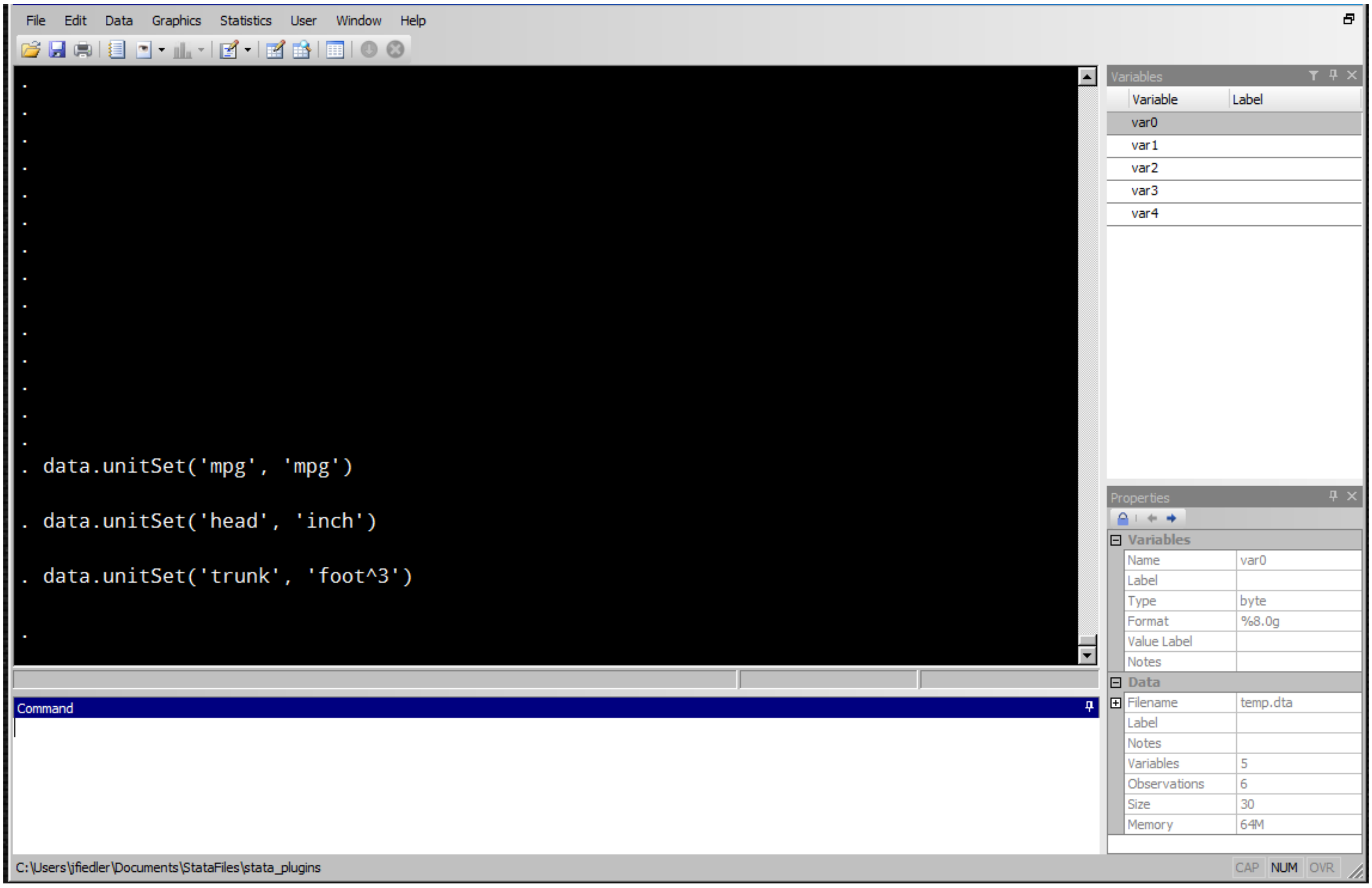
The dataset in Python can contain anything that a .dta file would. Here I demonstrate that notes can be added in Python.



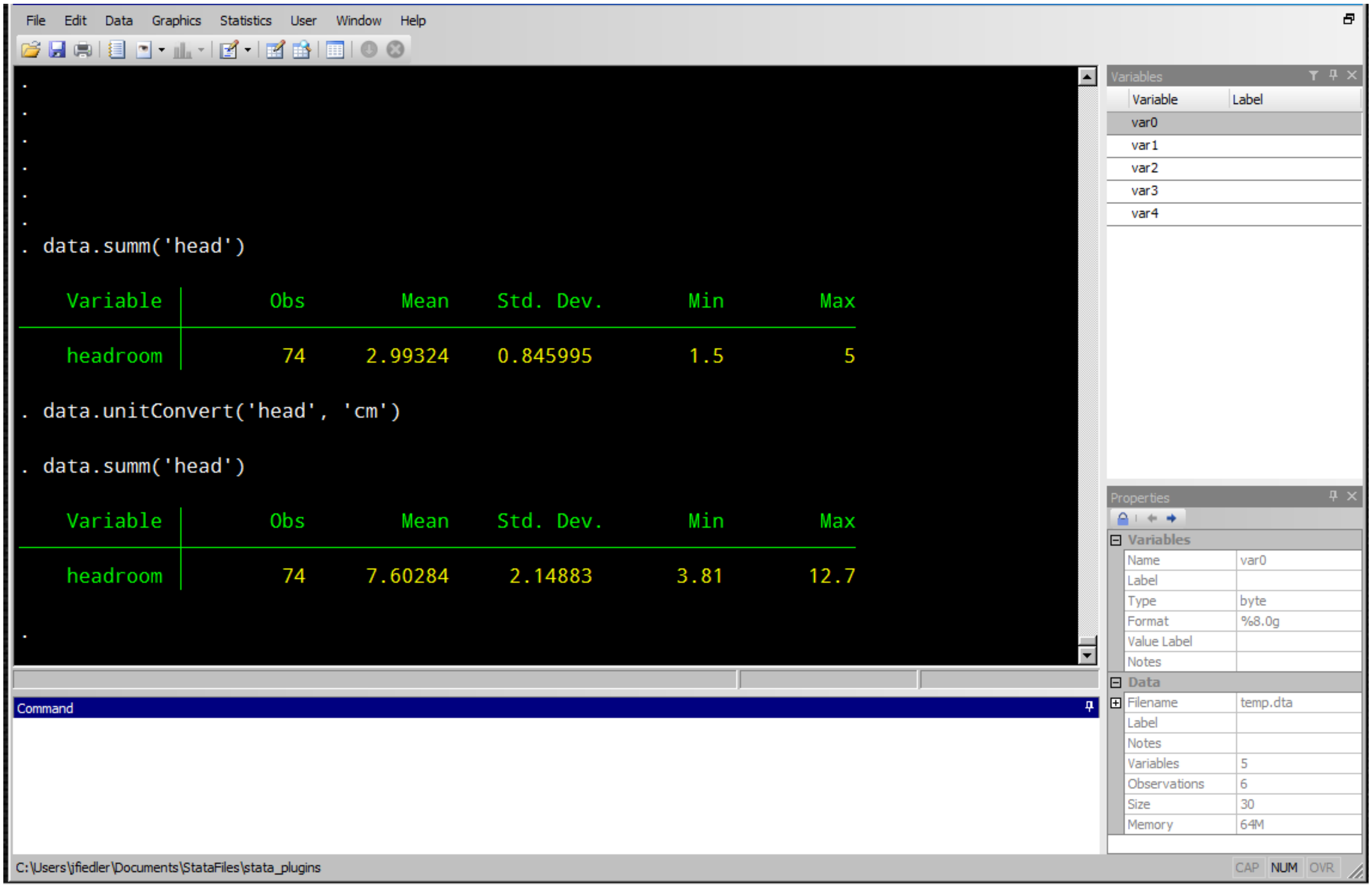
What else could we do with Stata datasets in Python? Here's an idea: how about adding functionality for recording the data units? Obviously, knowing the data units is necessary for understanding the contents of a dataset. There already are places available for recording this information, such as in variable 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.



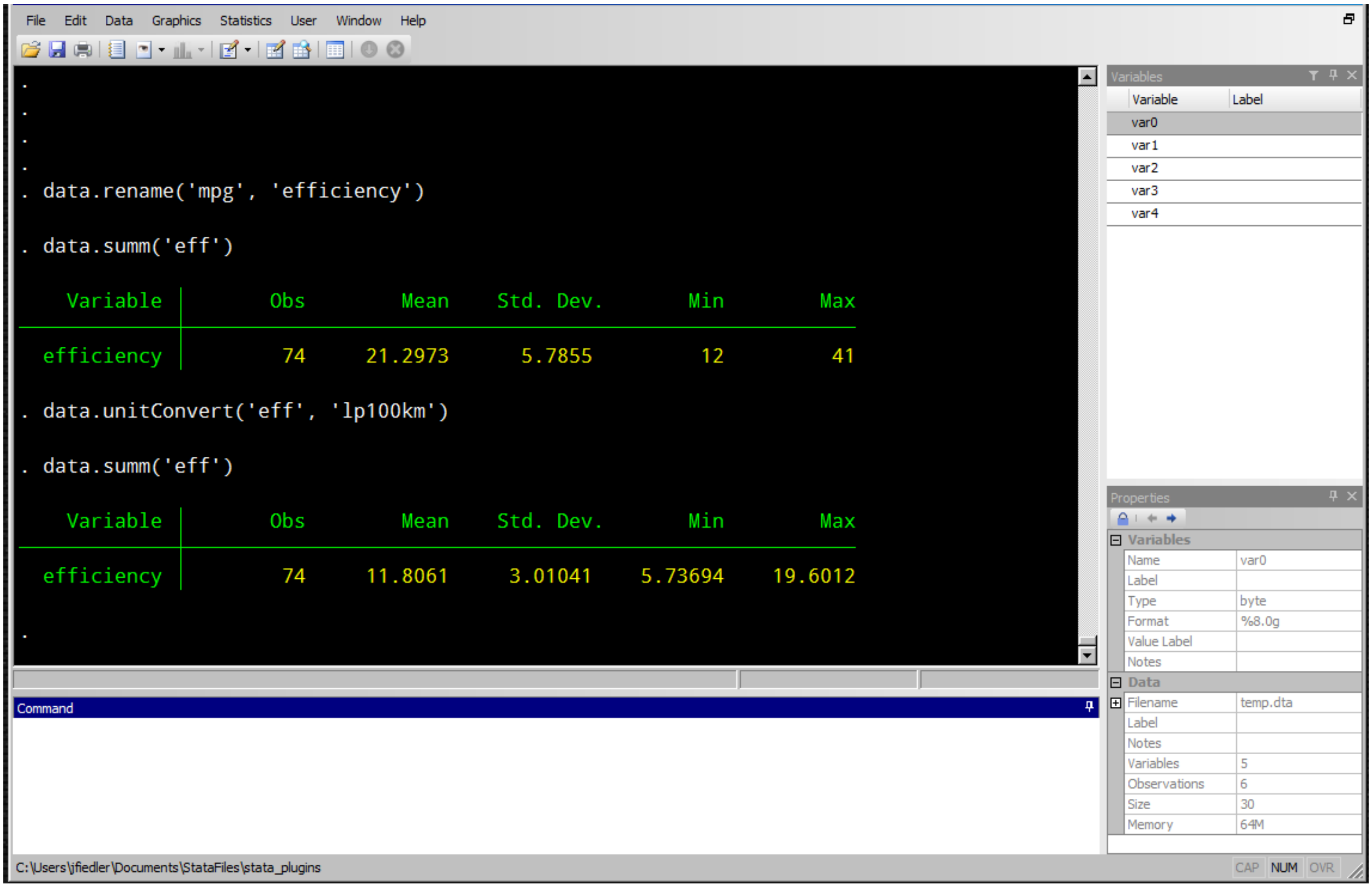
Here I've entered the Python interactive interpreter, then imported `UDta` from the `units_dta` module. `UDta` is subclass of `Dta`. It has all of the functionality that `Dta` has, but I've also added some functions for working with units. Above, I've used `UDta` to open a saved dataset and then summarize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



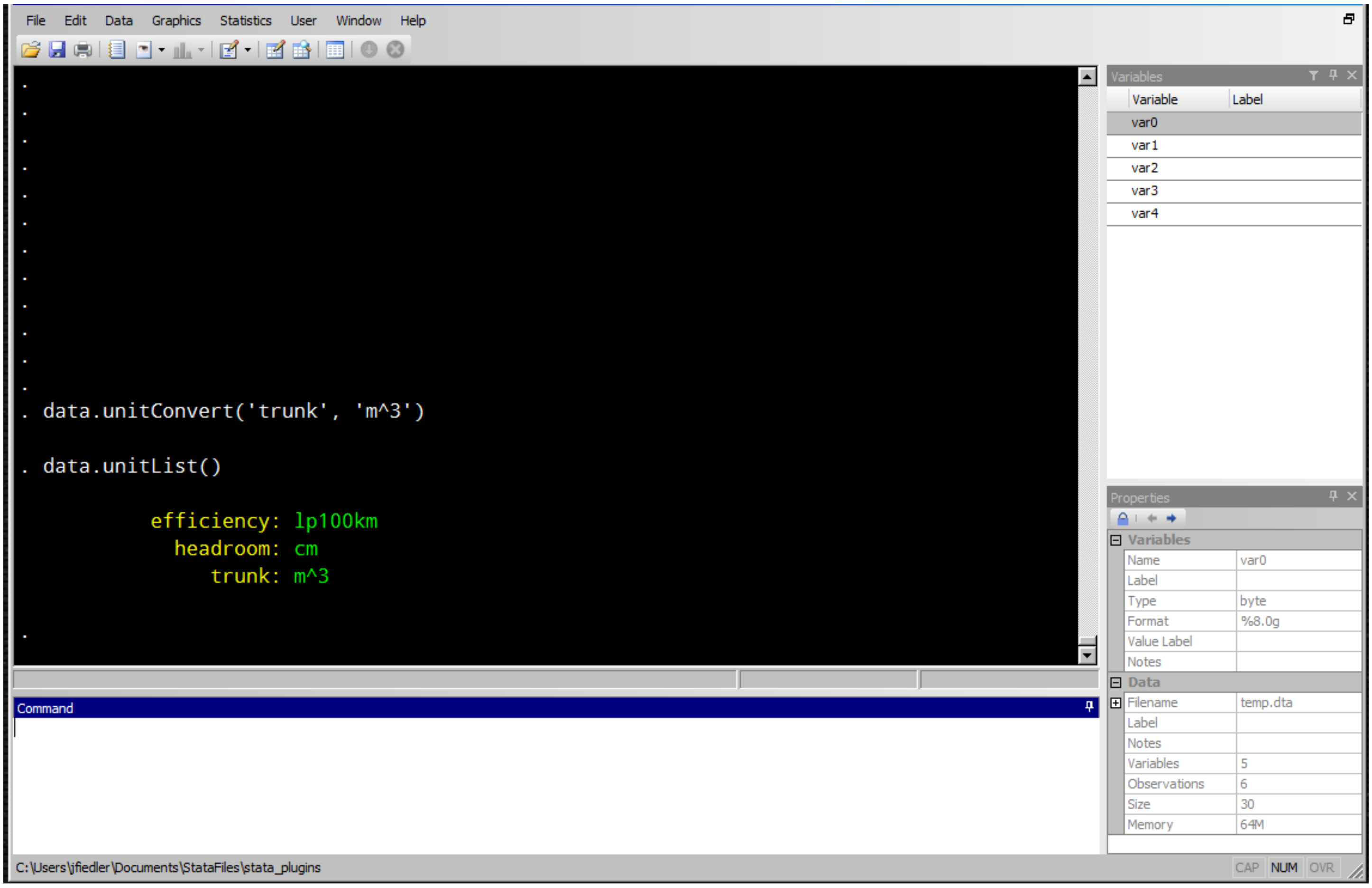
Part of the added functionality in `UDta` is a method `unitSet()` for setting the units. Here I've set the units for `mpg`, `headroom`, and `trunk`. In normal usage I would set the rest of the units likewise.



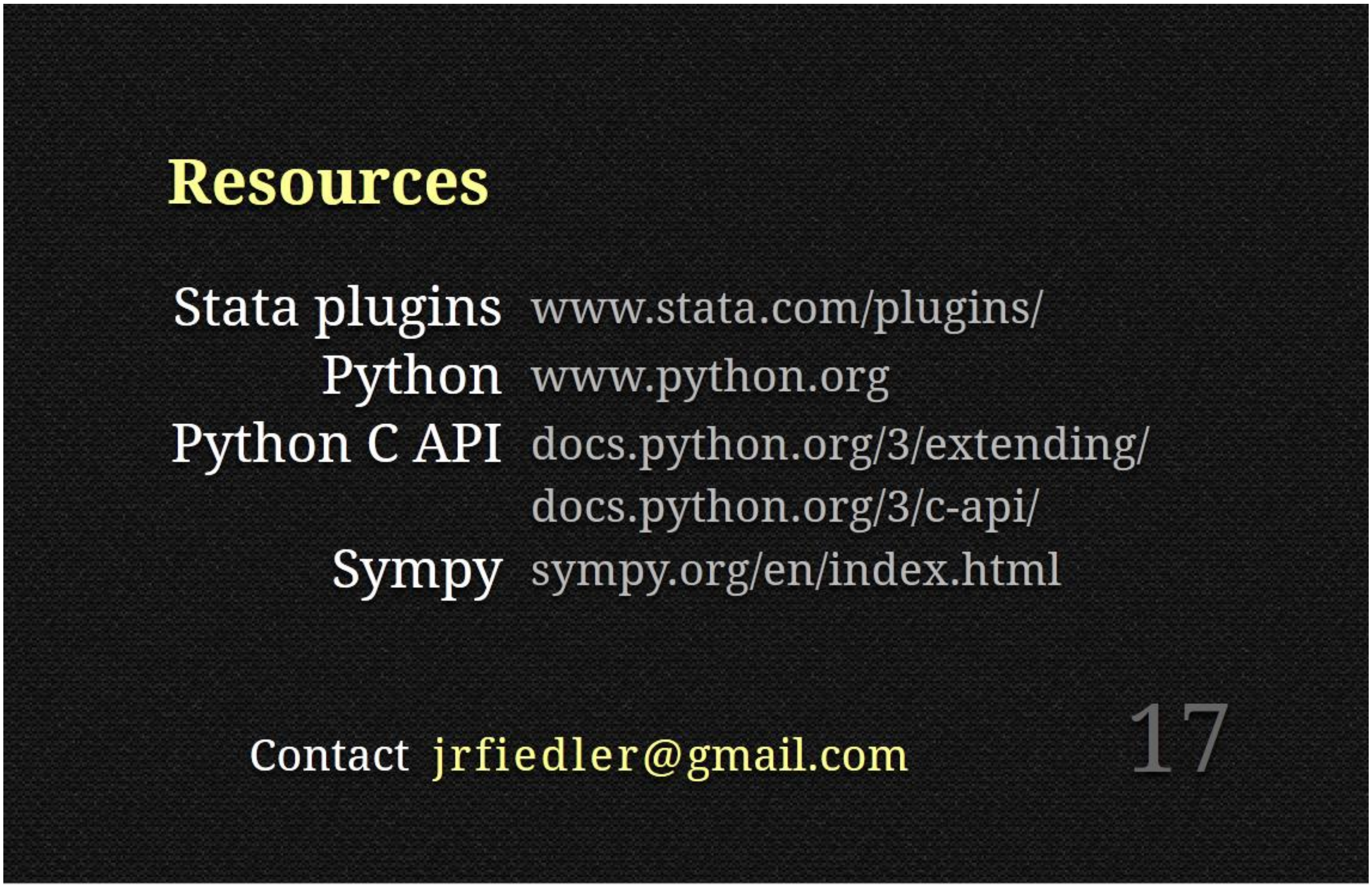
Now, suppose you're in the US (so these units probably seem natural to you), and you give this dataset to someone in Europe, who would probably prefer metric units. The `UDta` class has a method `unitConvert()` for making conversions. This method doesn't just replace the old unit with the new unit, it 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.



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".



Finally, I convert the `trunk` units, and then call the method `unitList()` to see the defined units.


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.