Re-imagining a Stata/Python combination

James Fiedler

Stata Conference 2013

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.

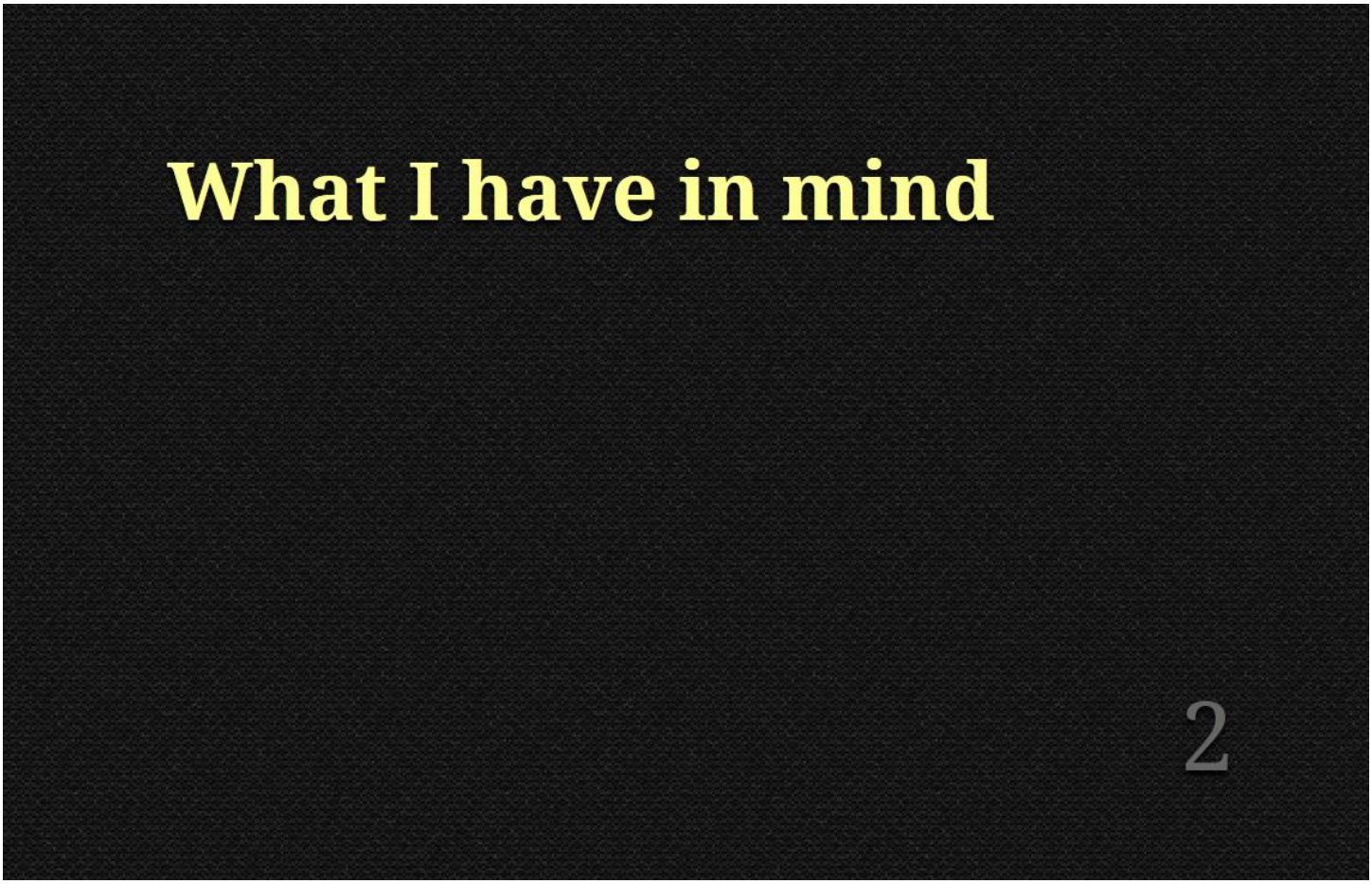
Support

where I work

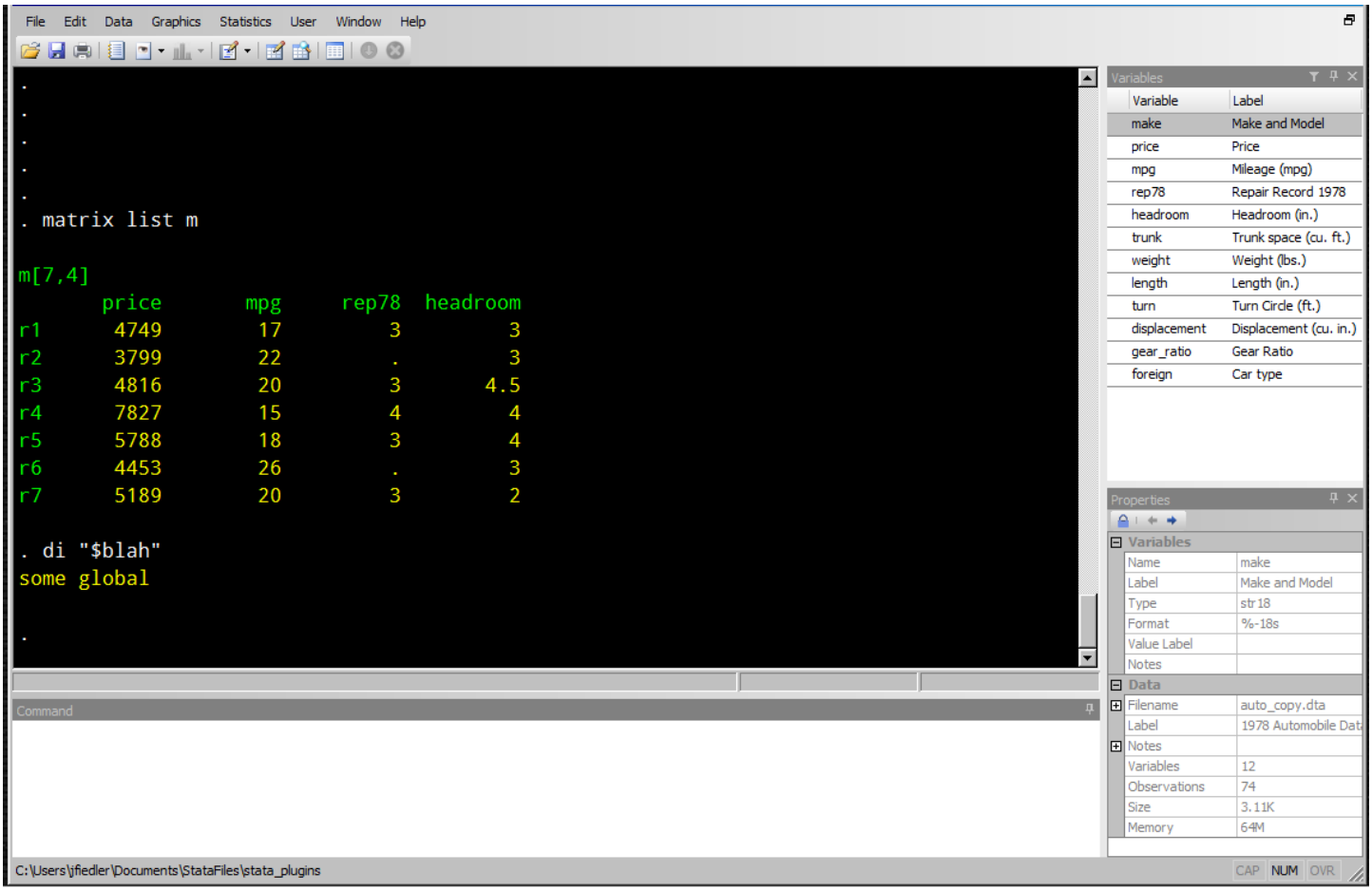
NASA Johnson Space Center

who I work for

Universities Space Research Association

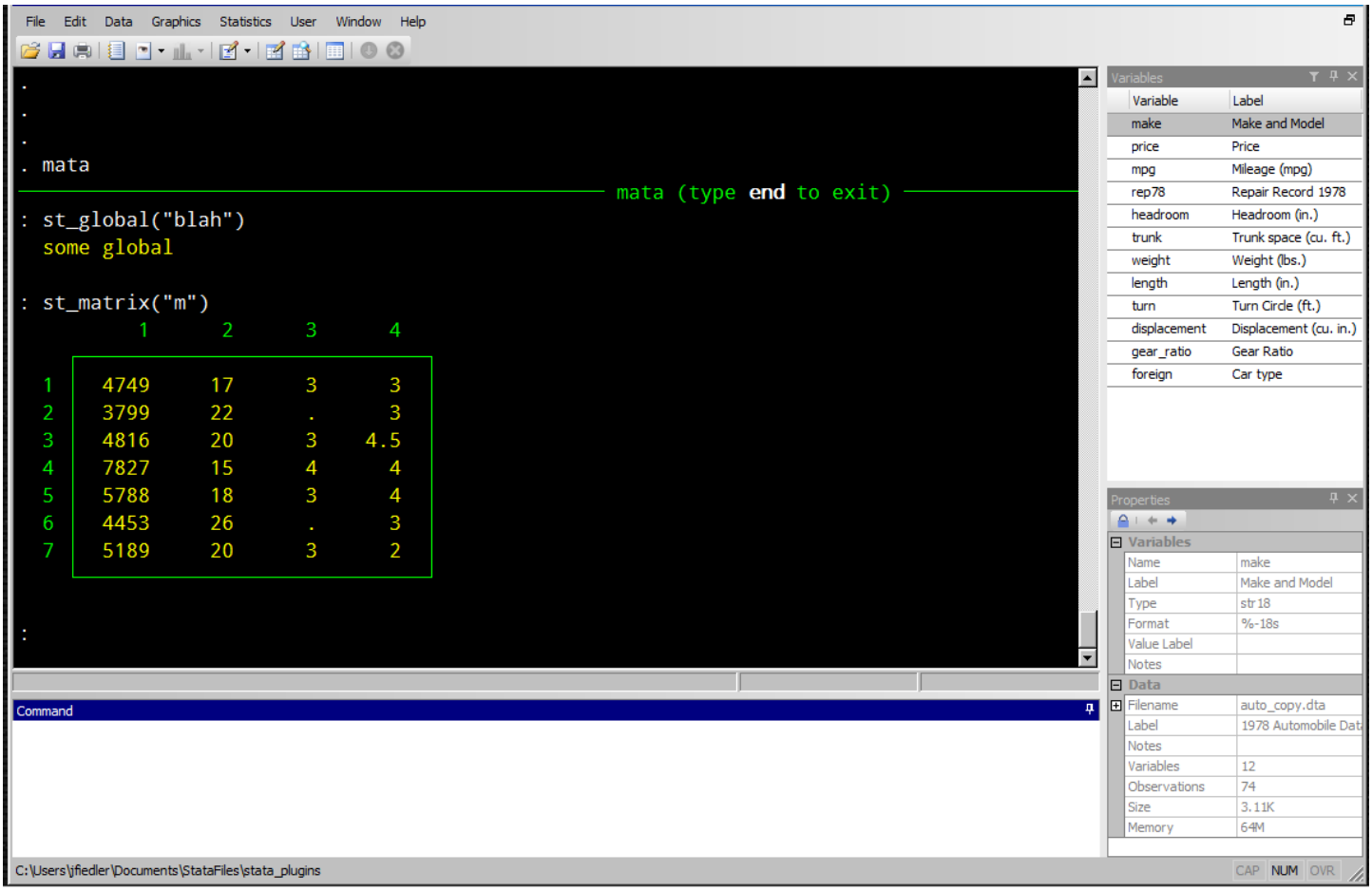


To start off, let me describe the kind of “Stata/Python combination” I have in mind for this talk.



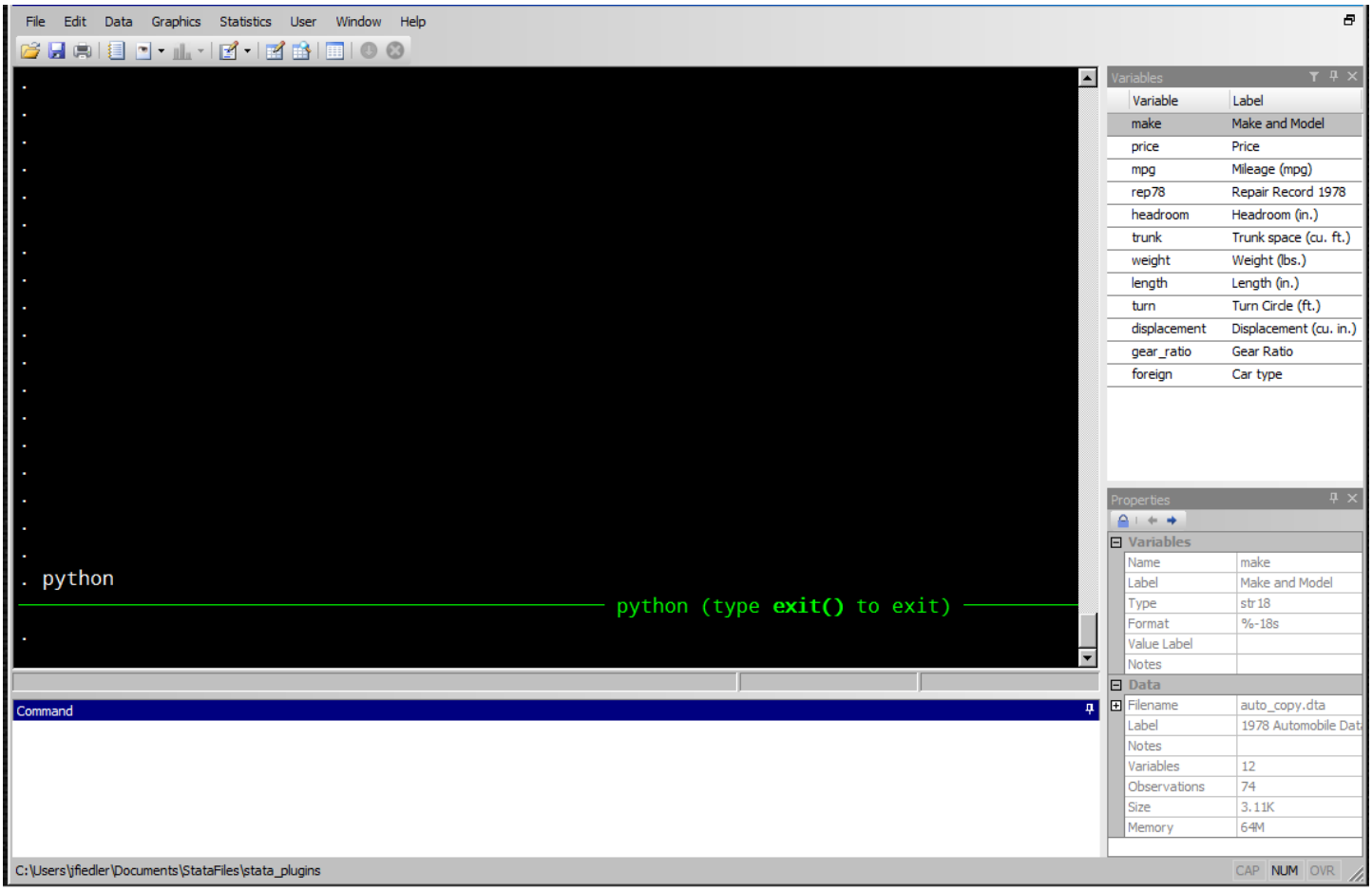
(In order to have some values to work with, I have loaded a copy of the auto data set, created a matrix `m` from a subset of the data values, and created a global macro `blah`.)

What I'd like to see is the ability to use Python interactively within Stata. Currently, there are two interactive modes. There's (what I call) "ado mode", which is what I used in this slide.

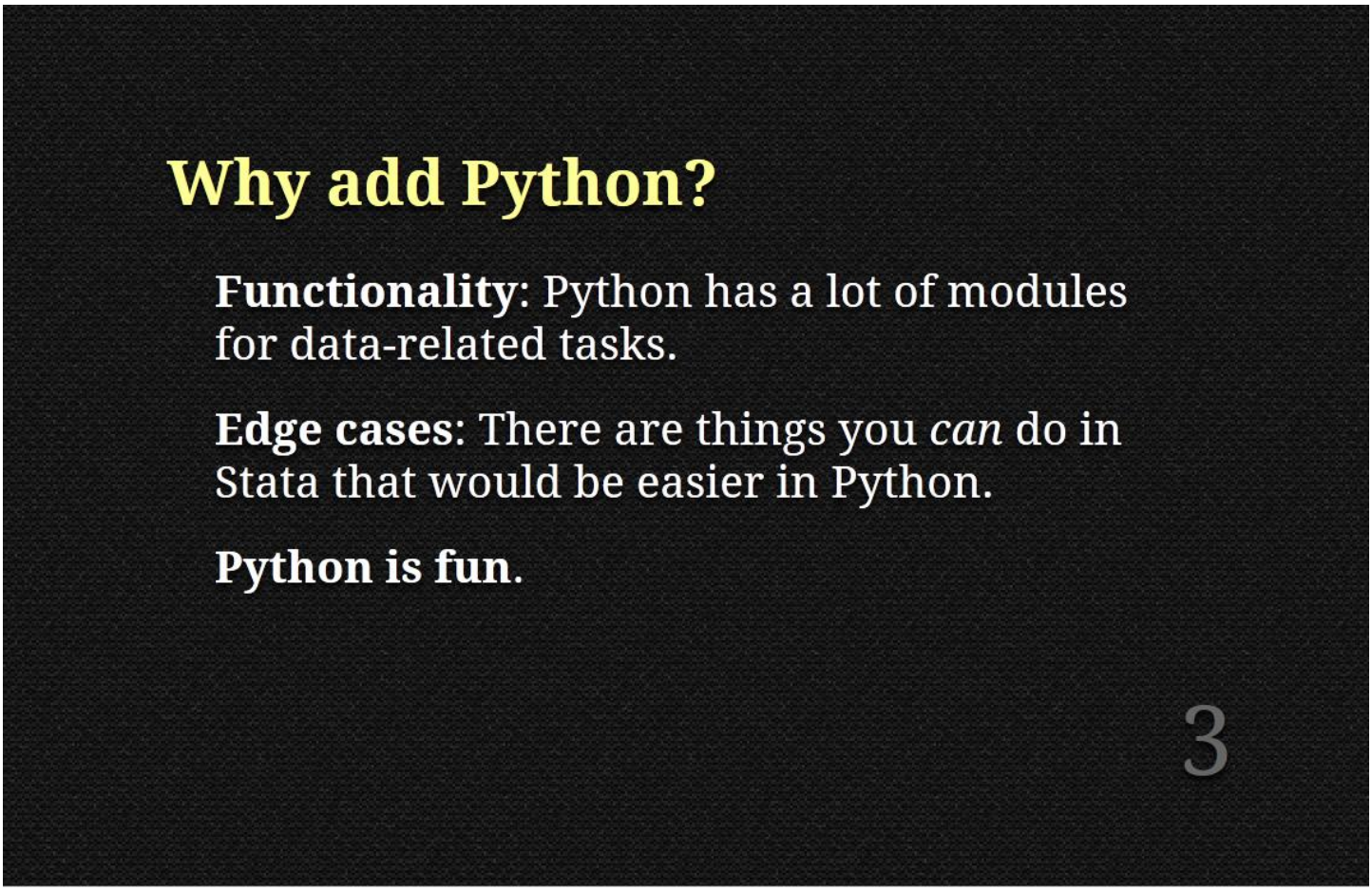


And with the `mata` command the user can enter a Mata interactive mode. Mata has different syntax, data structures, and functionality, but it's not completely disconnected from "ado mode" because there are functions for interacting with values defined in ado mode. For example, `st_global()` can be used to bring an ado-mode macro into Mata as a string and `st_matrix()` can be used to bring an ado-mode matrix in as a Mata matrix.

What I'd like to see is another interactive mode where Python can be used.



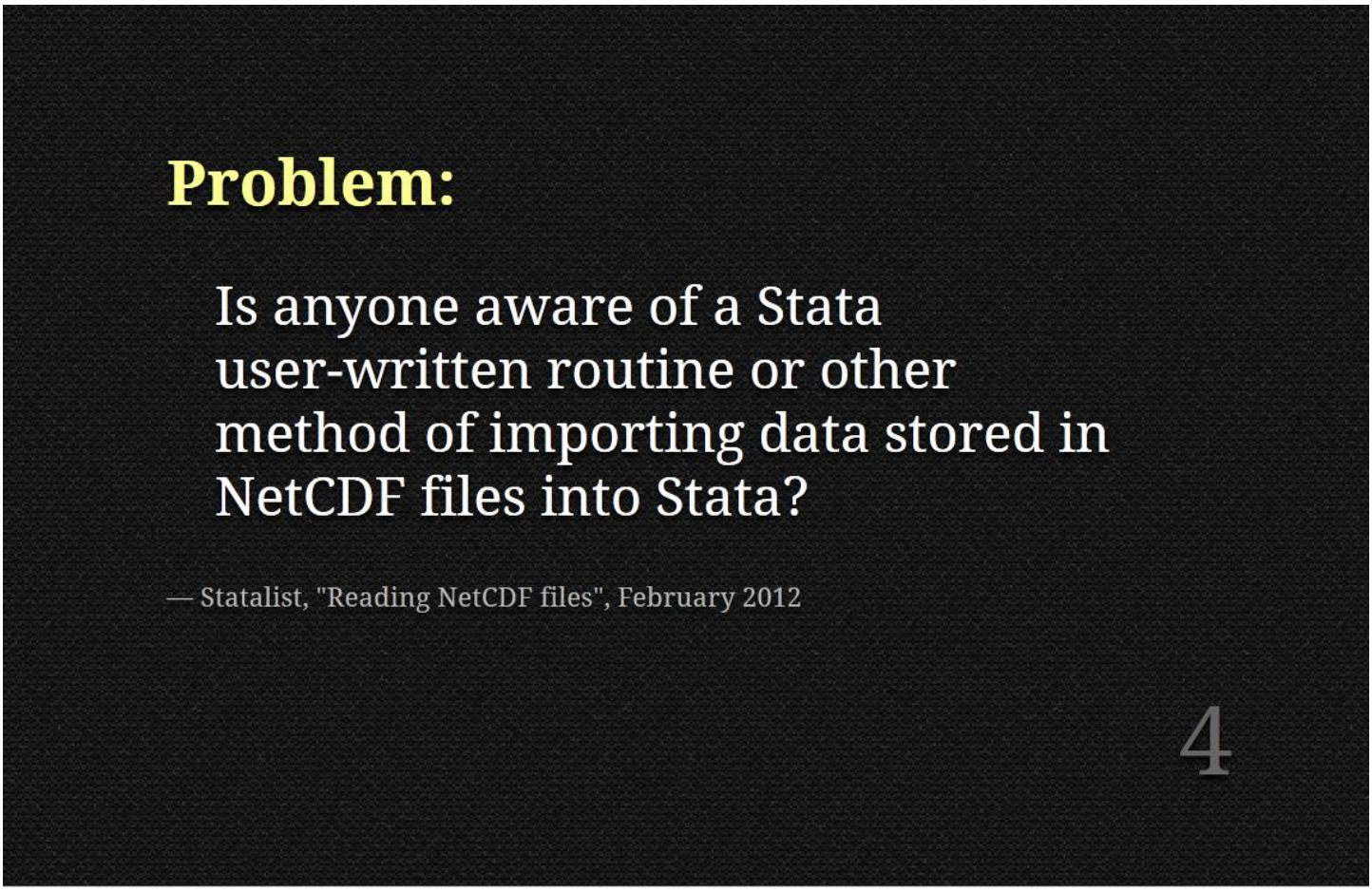
Just as there's a `mata` command in Stata, which puts the user into an interactive Mata session, there could be a `python` command which puts the user into an interactive Python session. And just as there are Mata functions for interacting with Stata values from Mata, there could be Python functions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



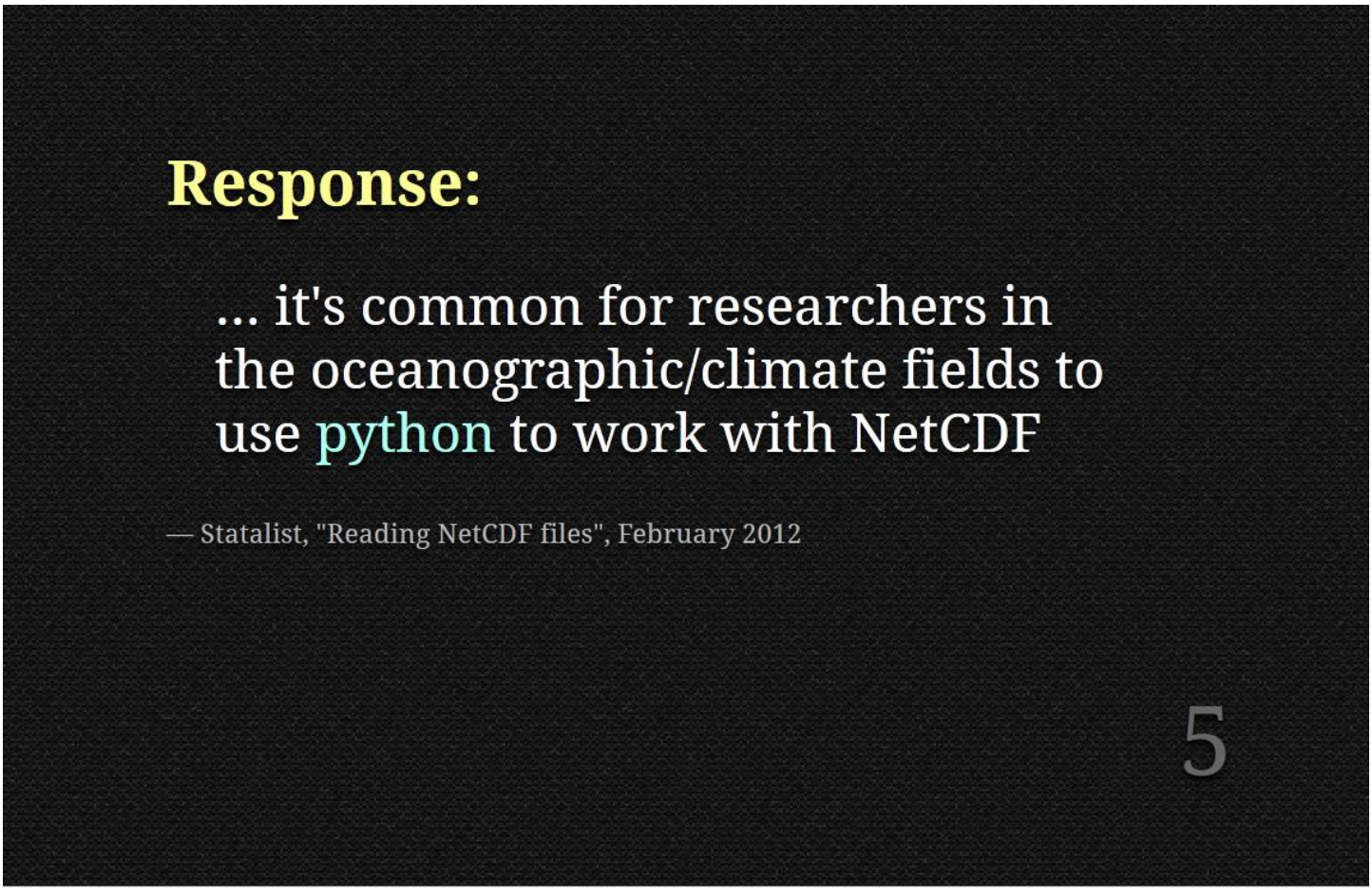
The first, obvious reason for adding Python is that doing so would add functionality. There is a large community of people using Python for data analysis, numerical computing, scientific computing, etc. As a consequence, there is a lot of Python code for doing the kinds of things that Stata users do. Having access to this code would expand what users can do in Stata, or at least expand what can be done practically.

Second, when there is overlap in what can be done in Stata and Python, sometimes it'll be easier to do it in Python.

Finally, one of the things Python programmers will consistently tell you is that Python is a fun language to use.



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

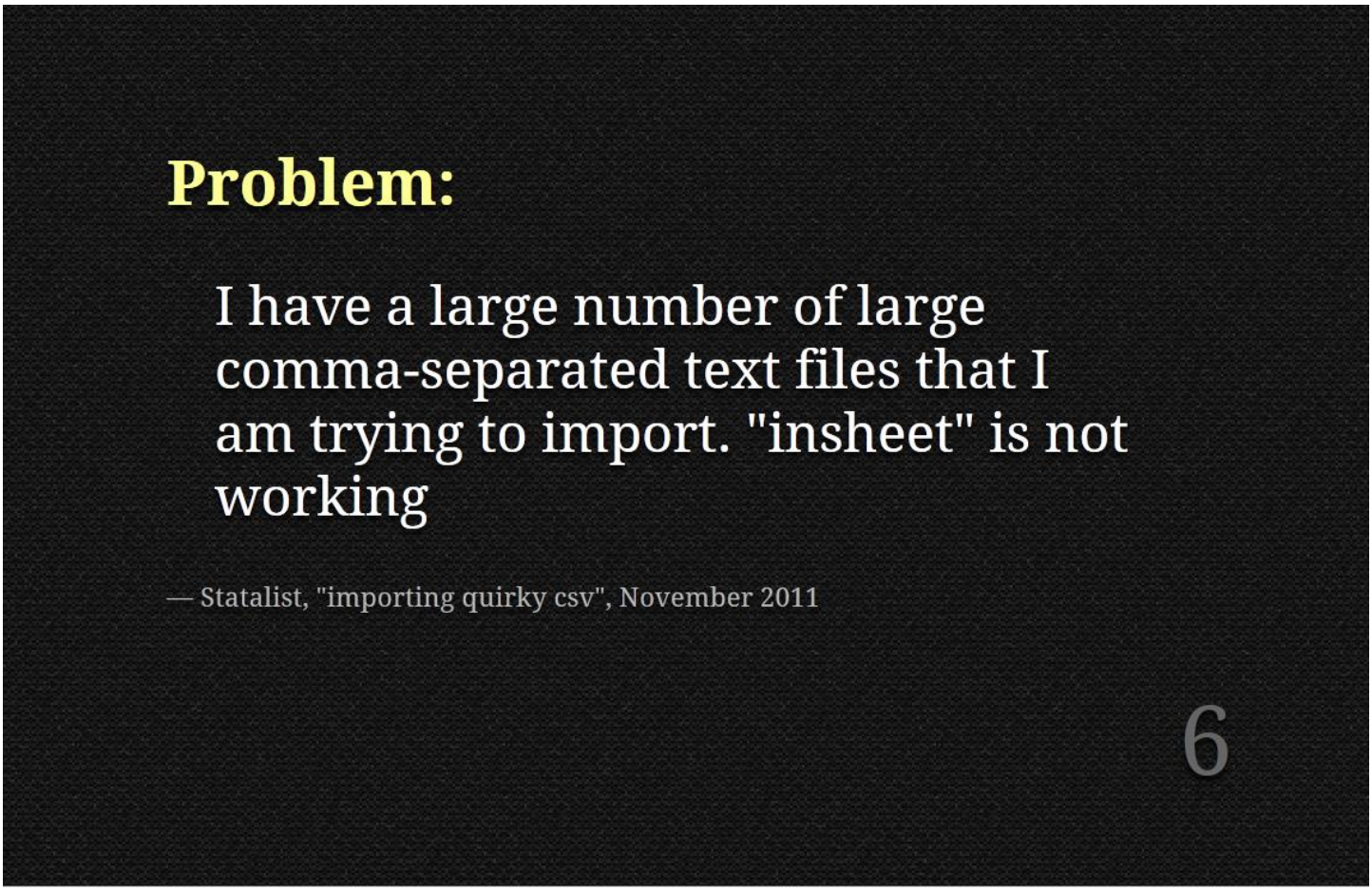


There weren't any Stata commands for opening the file, so one person suggested using Python.

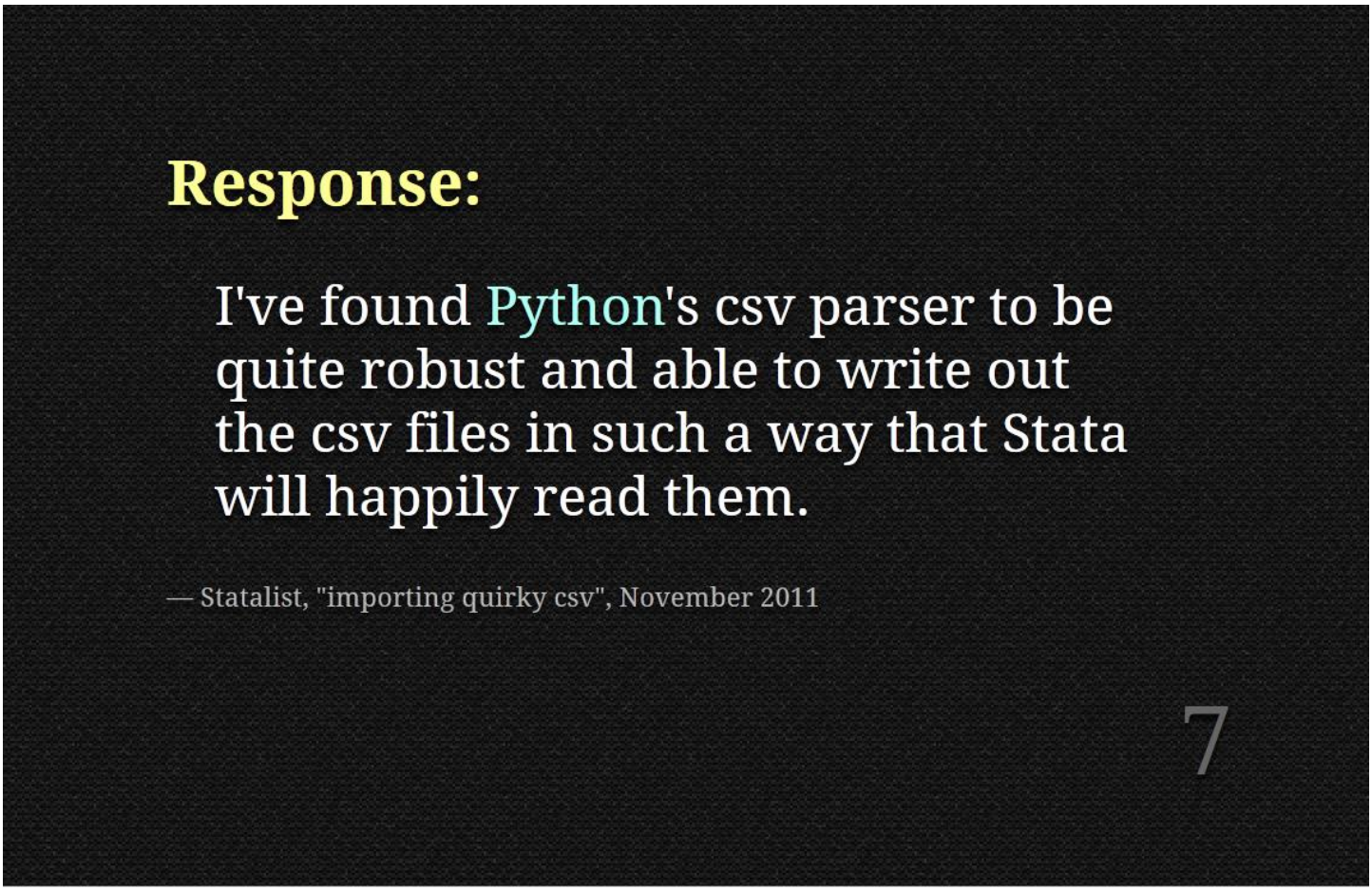
The file was handled some other way, but the user wrote back with some conclusions, one of which was:

> 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 and a module for reading NetCDF files it would probably only take a few lines of code to get this file into Stata.

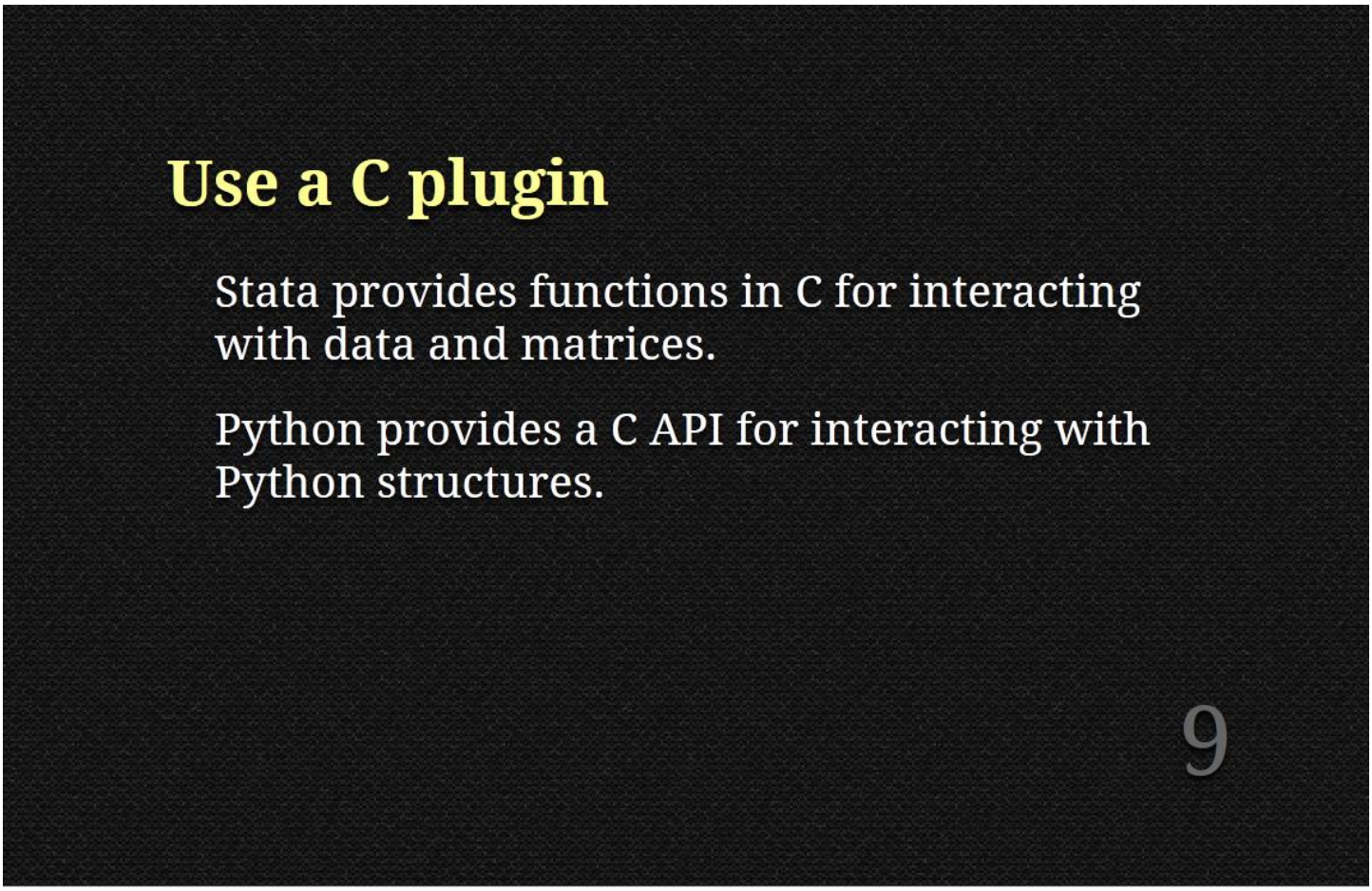


Here's an example of an edge case, again from Statalist. This person has a CSV file, which, because of some funky formatting, the user hasn't been able to import with the `insheet` command.



Another user suggests that Python's `csv` module might help. Without more information, we can't be sure that the Python method would be easier in this case, but we can see that the respondent has found cases where it is.

How do we add Python?

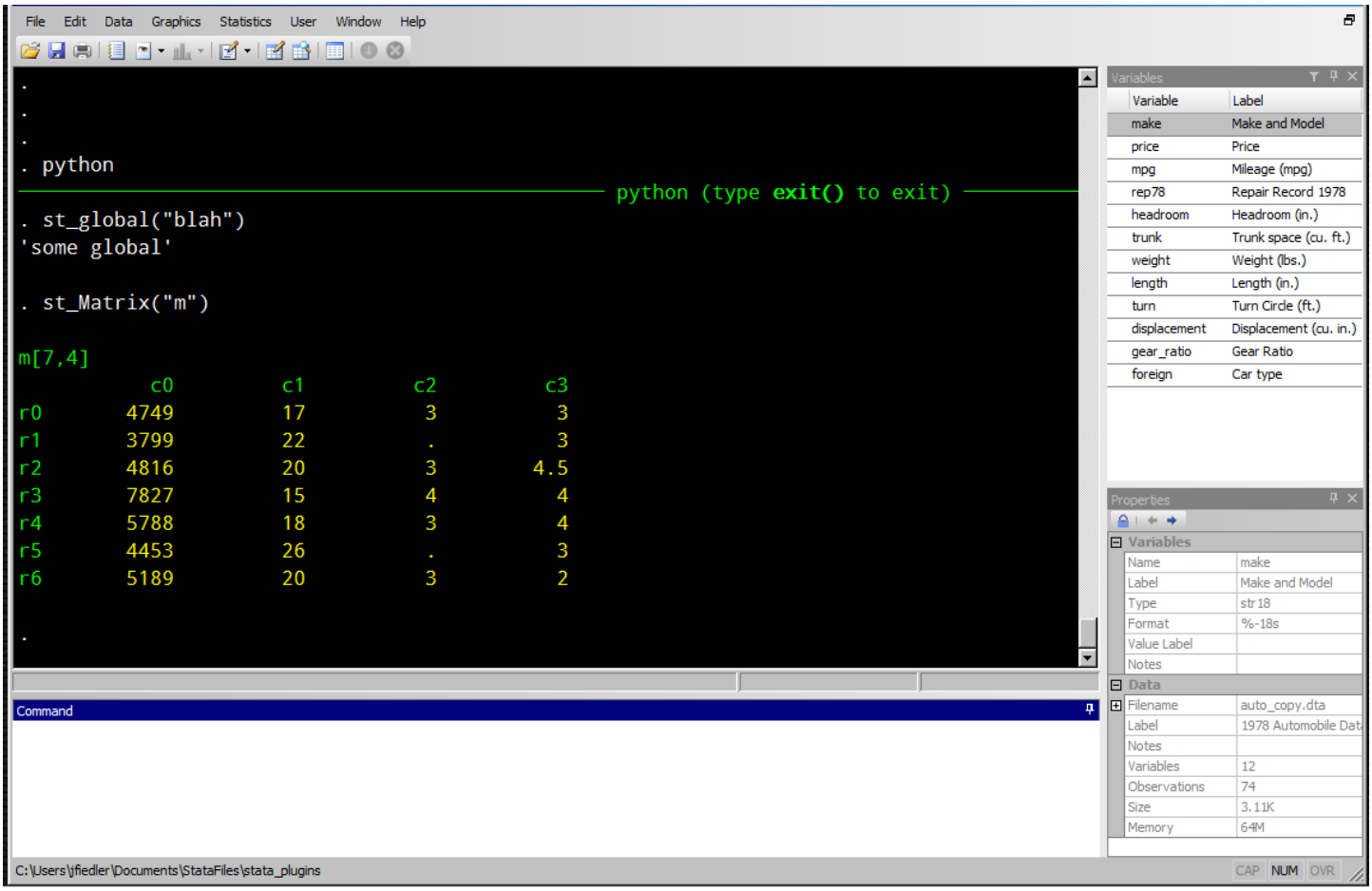


Stata, through its plugin system, provides C functions for interacting with Stata data, macros, matrices, etc. Python, through its C API (application programming interface) provides C functions for starting a Python interpreter and interacting with Python structures.

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

I think this is the way to go (i.e., much better than the methods I used at last year's Stata Conference), and I wish I could take credit for the idea. In fact, I'm pretty sure the idea came up in conversation with Stata Corp. employees last year, so credit for the idea should go to them.

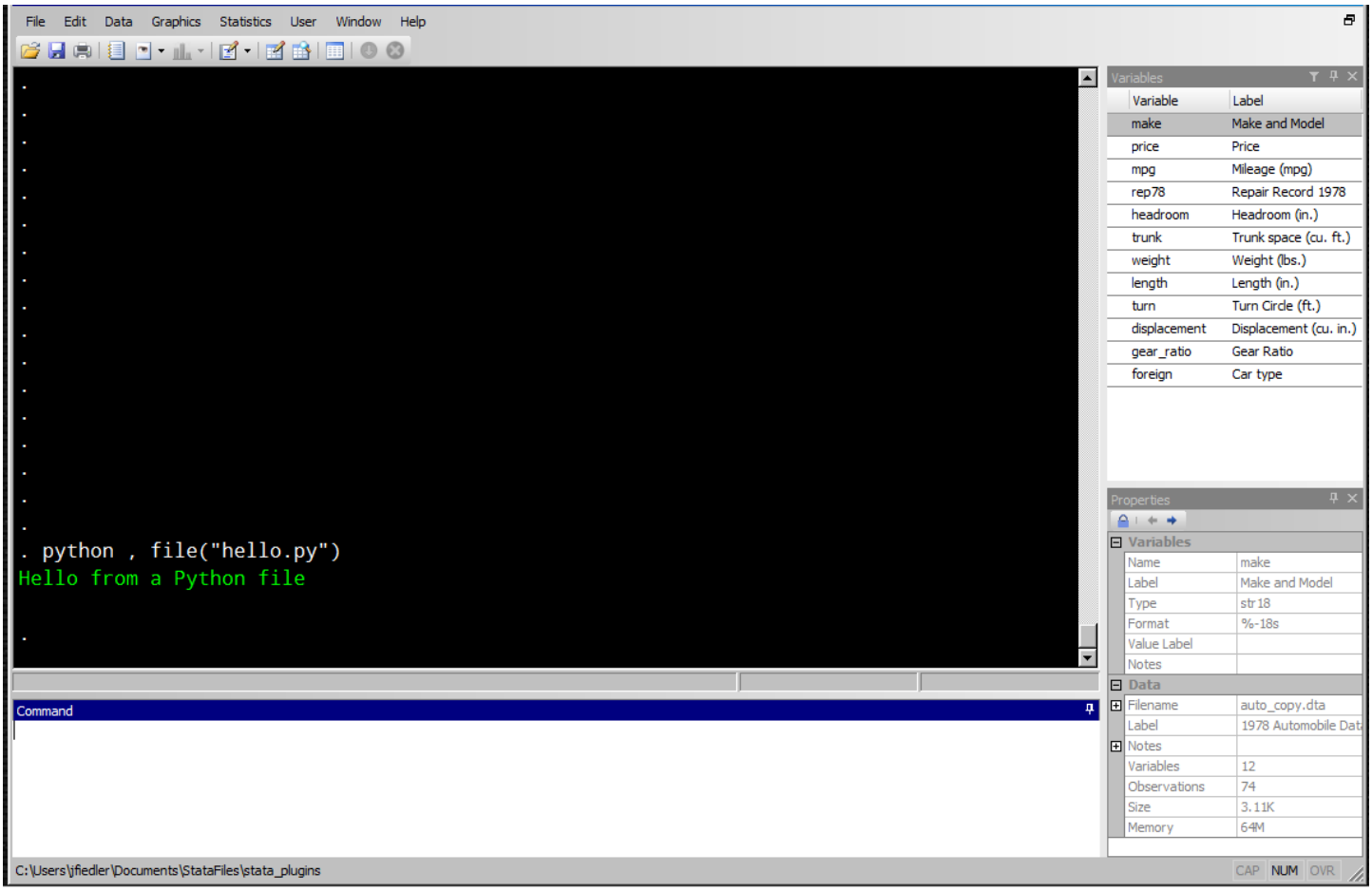
Before I begin a demonstration, I should probably make it clear that this isn't a simulation. Last year I used an imitation of the Stata GUI to demonstrate some ideas. This year there is no imitation Stata, and no imitation Python. This is a real instance of Stata, and I am really using Python within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



The idea, as I said, is to have the ability to use Python interactively within Stata. With Stata's C plugin functions, the idea can be implemented. In the slide above the command `python` puts the user in an interactive Python session.

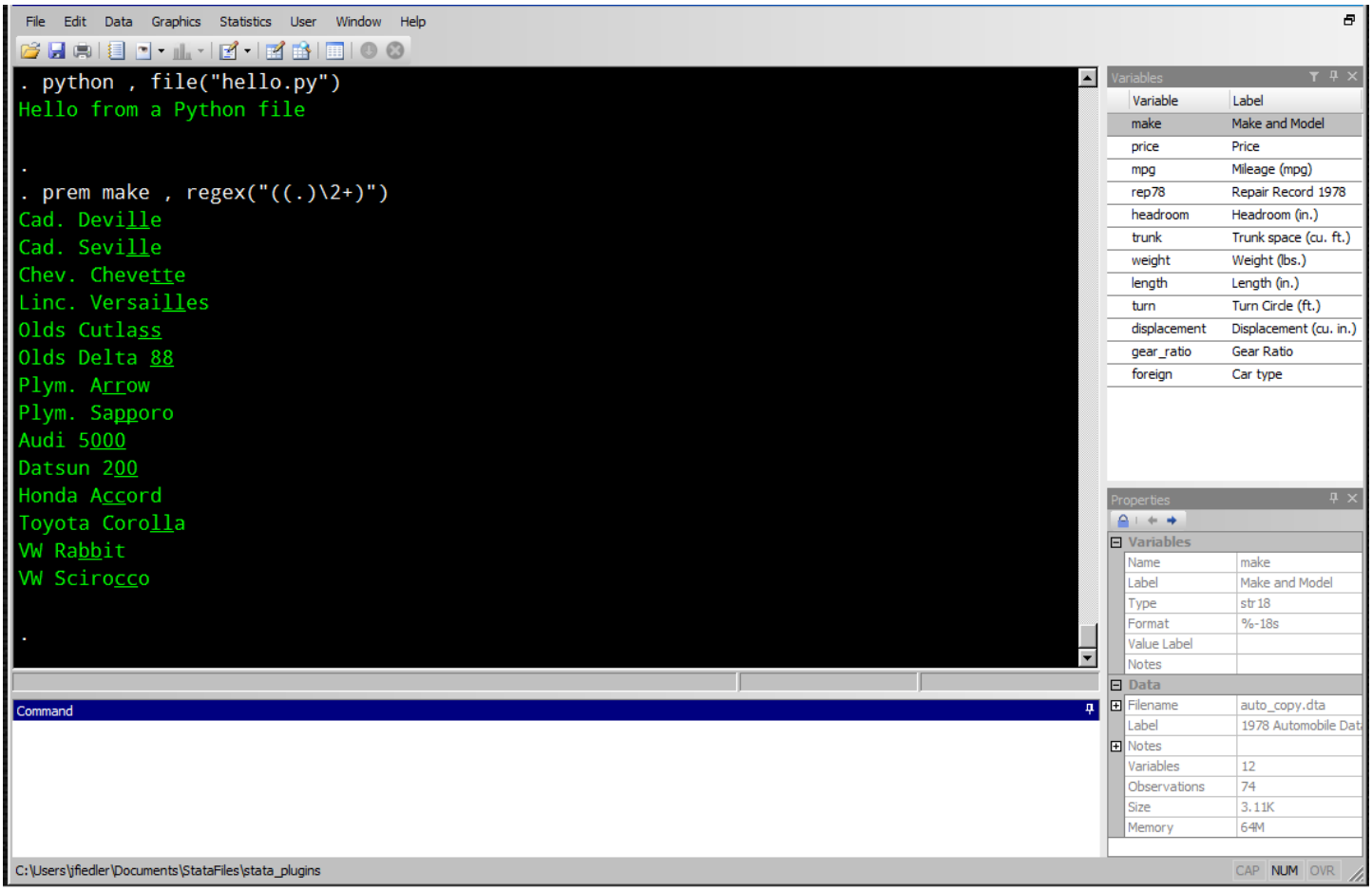
There's some leeway in how the C plugin functions are translated into Python functions. I've decided to make the Python functions mimic the Mata functions. The Python function for pulling in a global macros is `st_global()`, and the function for pulling in a Stata matrices is `st_Matrix()`.

(Notice the capital "M" in `st_Matrix()`. 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), so I'm back in ado mode, and I've used the `python` function with the file option. The `hello.py` file is just a single line:

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

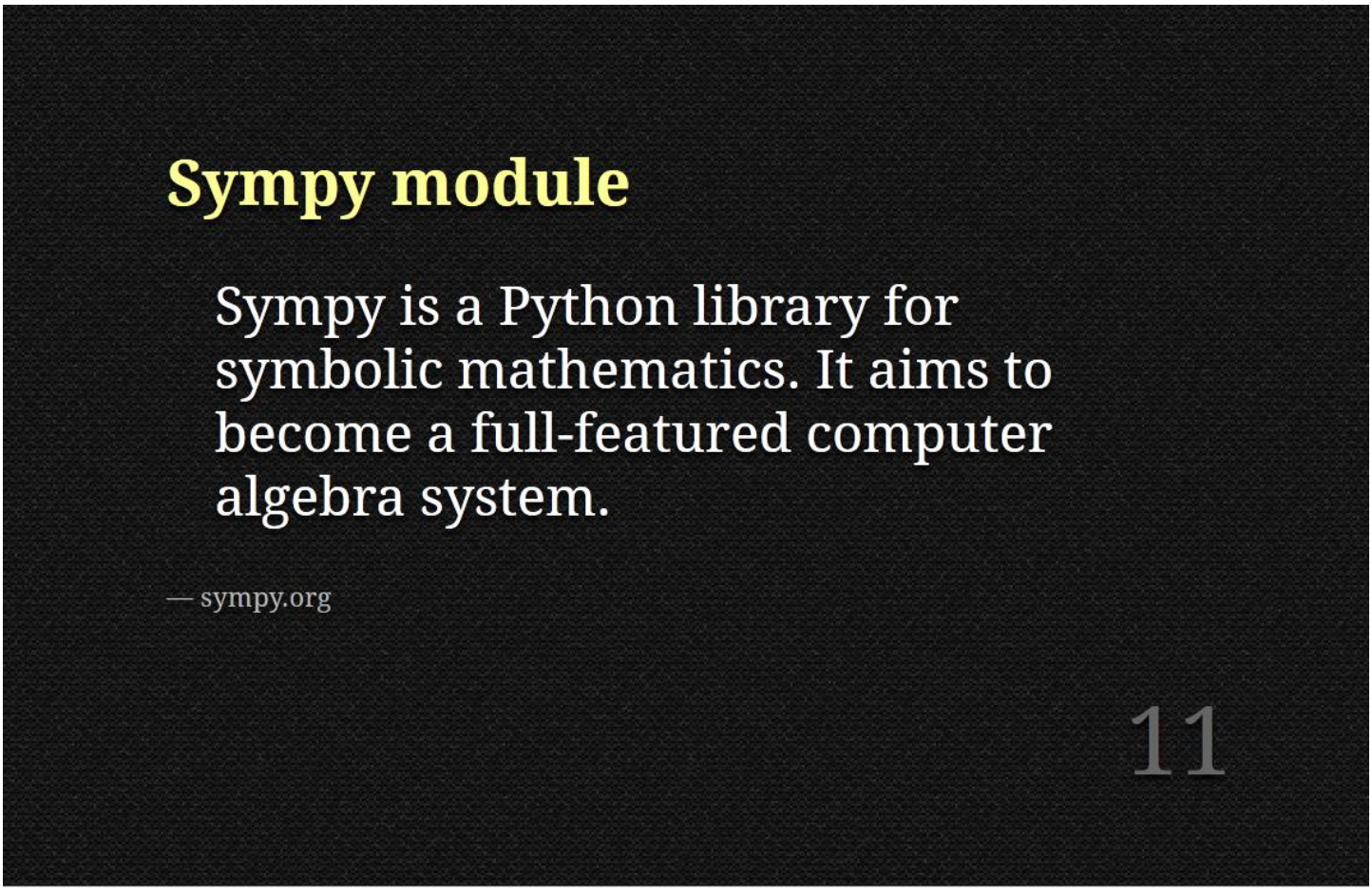


(This example is in ado mode.)

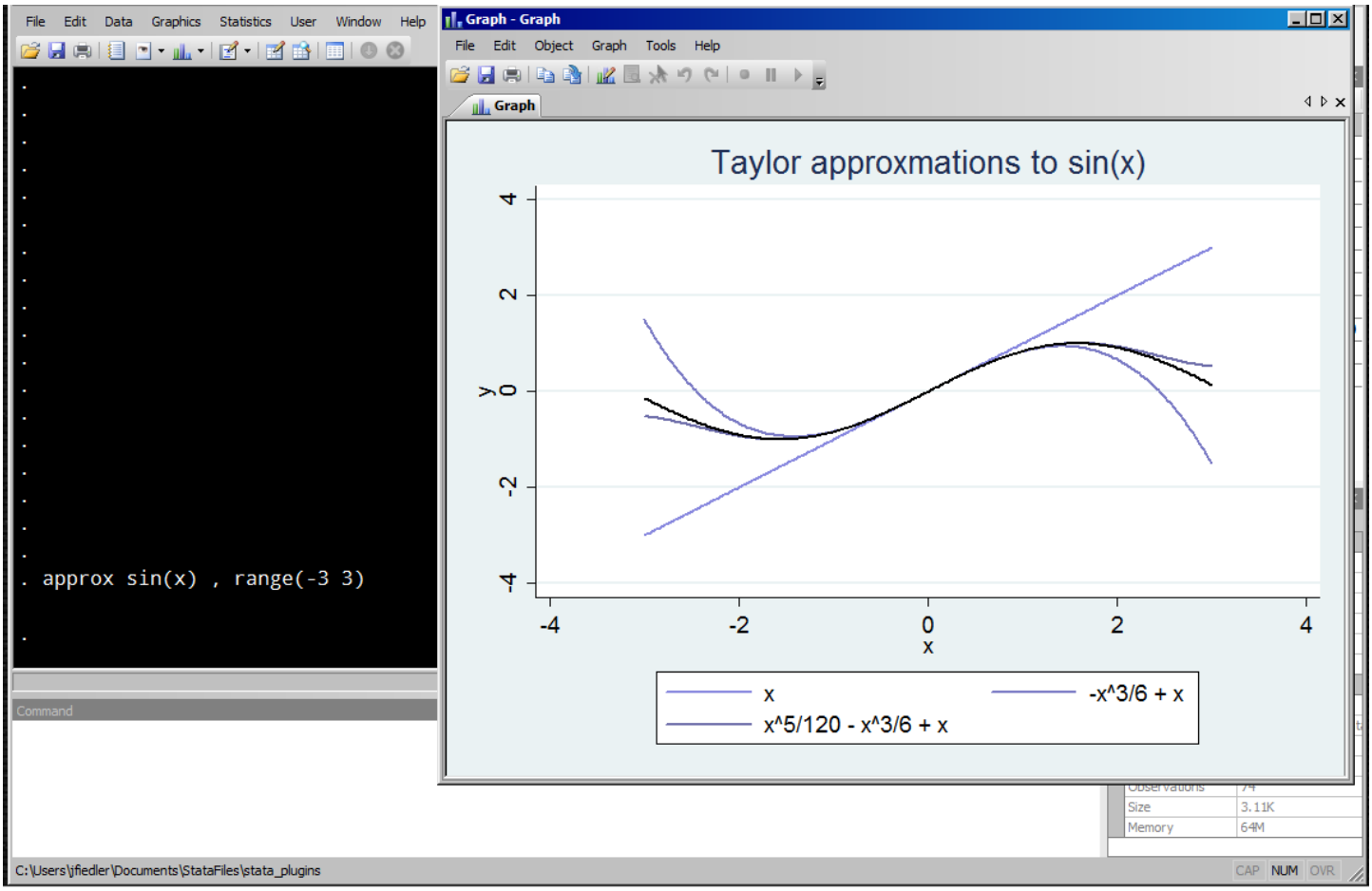
The previous page shows a Python file used in isolation. Here a Python file will be used together with an ado file to find regular expression matches.

If you've never used regular expressions, these are basically a cryptic method of specifying patterns in text. If you have used regular expressions, Stata's regular expressions are probably sufficient for the vast majority of uses. Occasionally, though, some Stata users want to do something more complicated than what Stata's regular expressions were designed to handle.

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 entries repeated characters—for example, it finds the "bb" in "Rabbit" and the "000" in "5000"—then prints it to the screen with the match underlined.



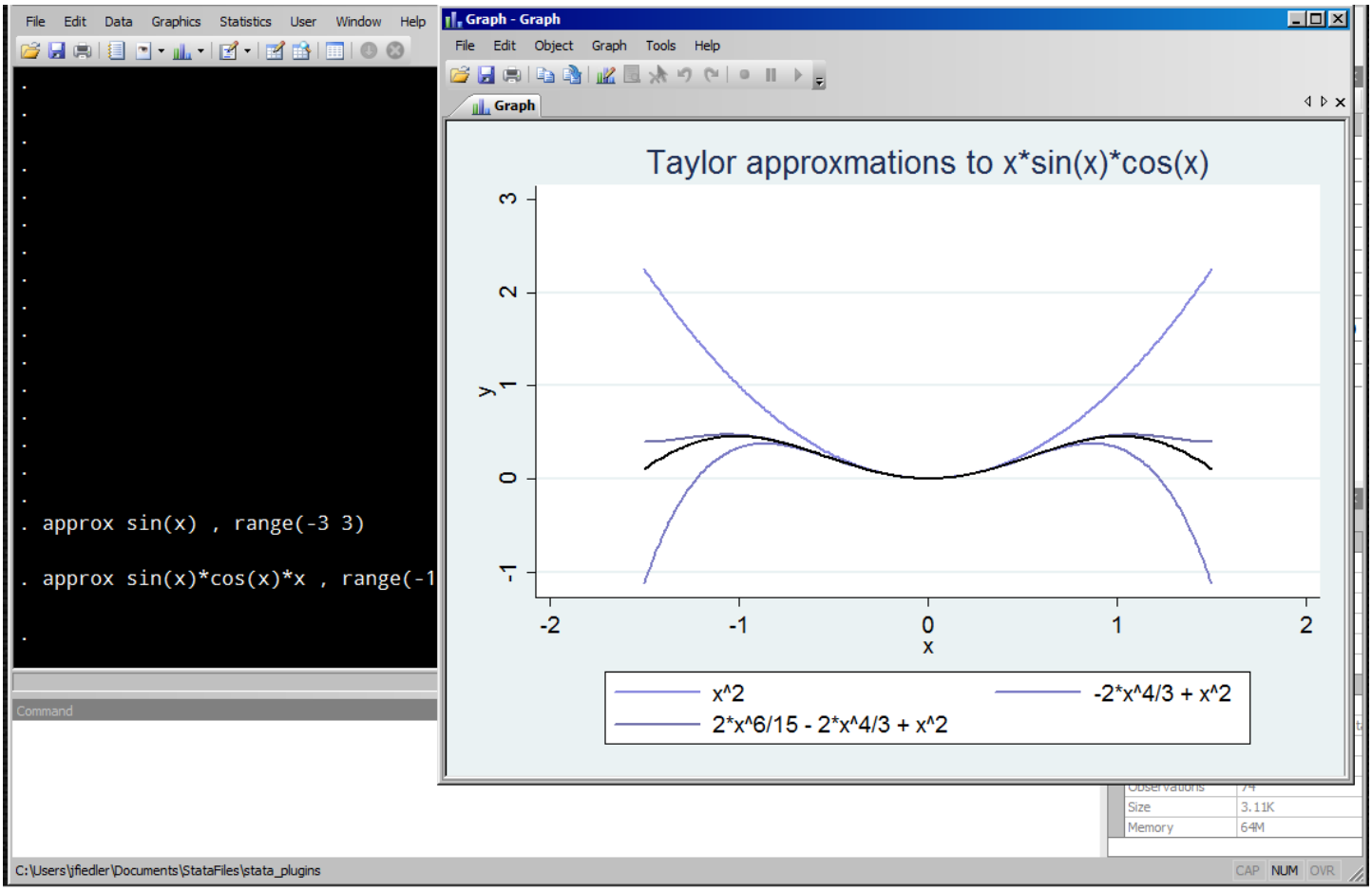
For the next couple examples I'll be using a symbolic math module called Sympy. With Sympy you can do things like take the limit of a function, factorize polynomials, and get the Taylor series expansion of a function.



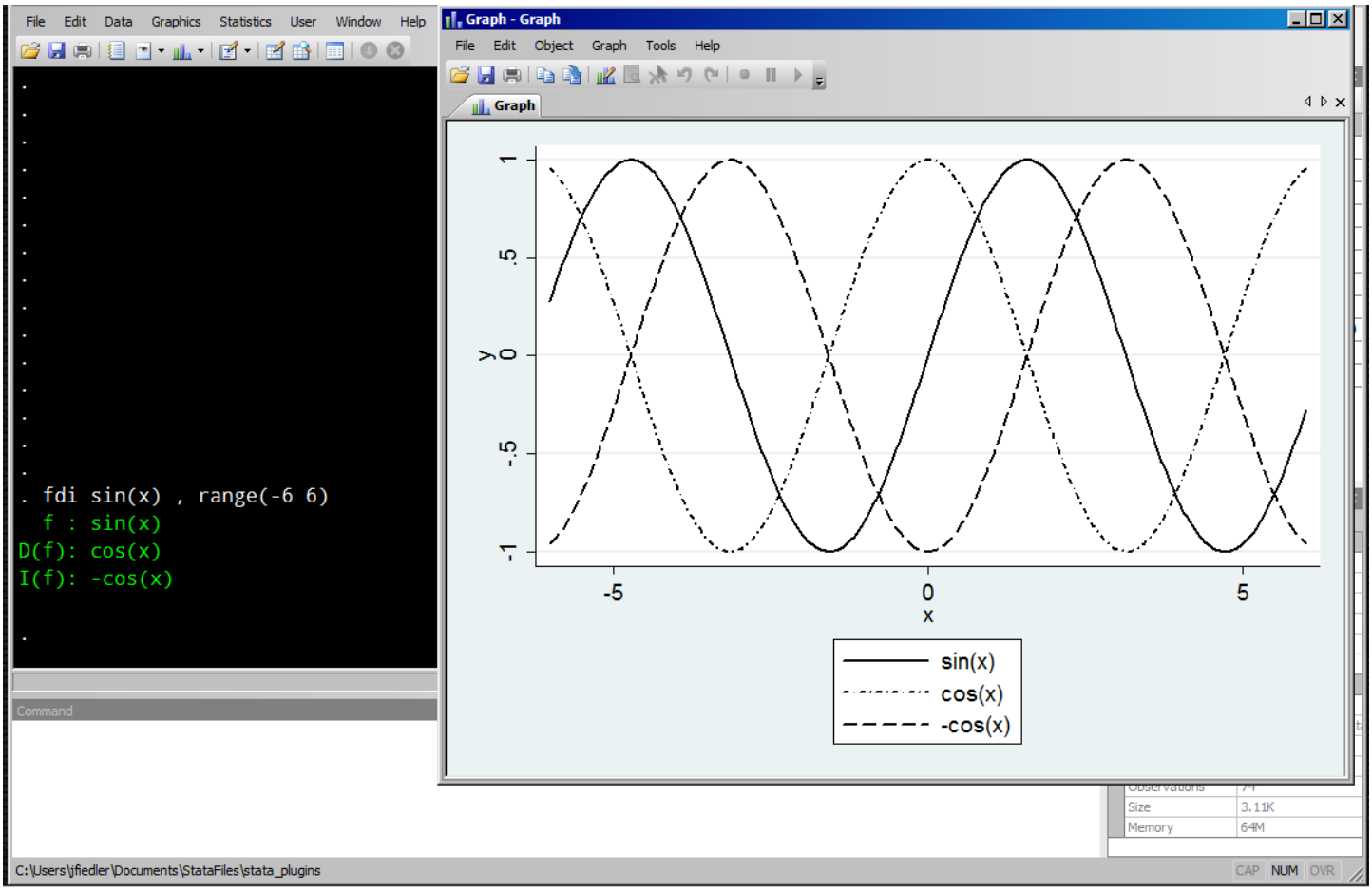
(This example is in ado mode.)

Here I'm using the command 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.

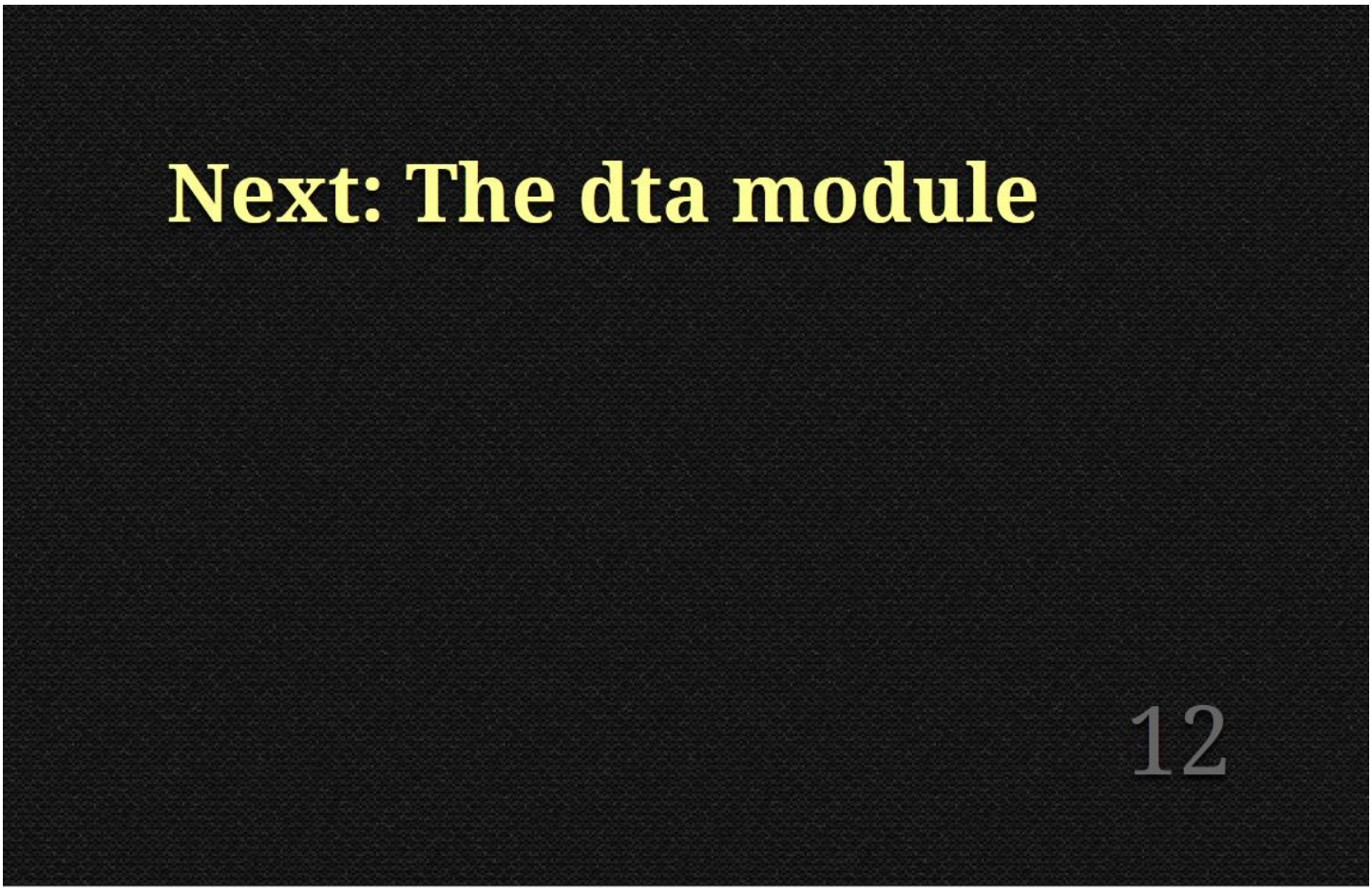
Again, the point of this is to show that ado files and Python files can work together seamlessly.



Here's another example with a more complicated function.

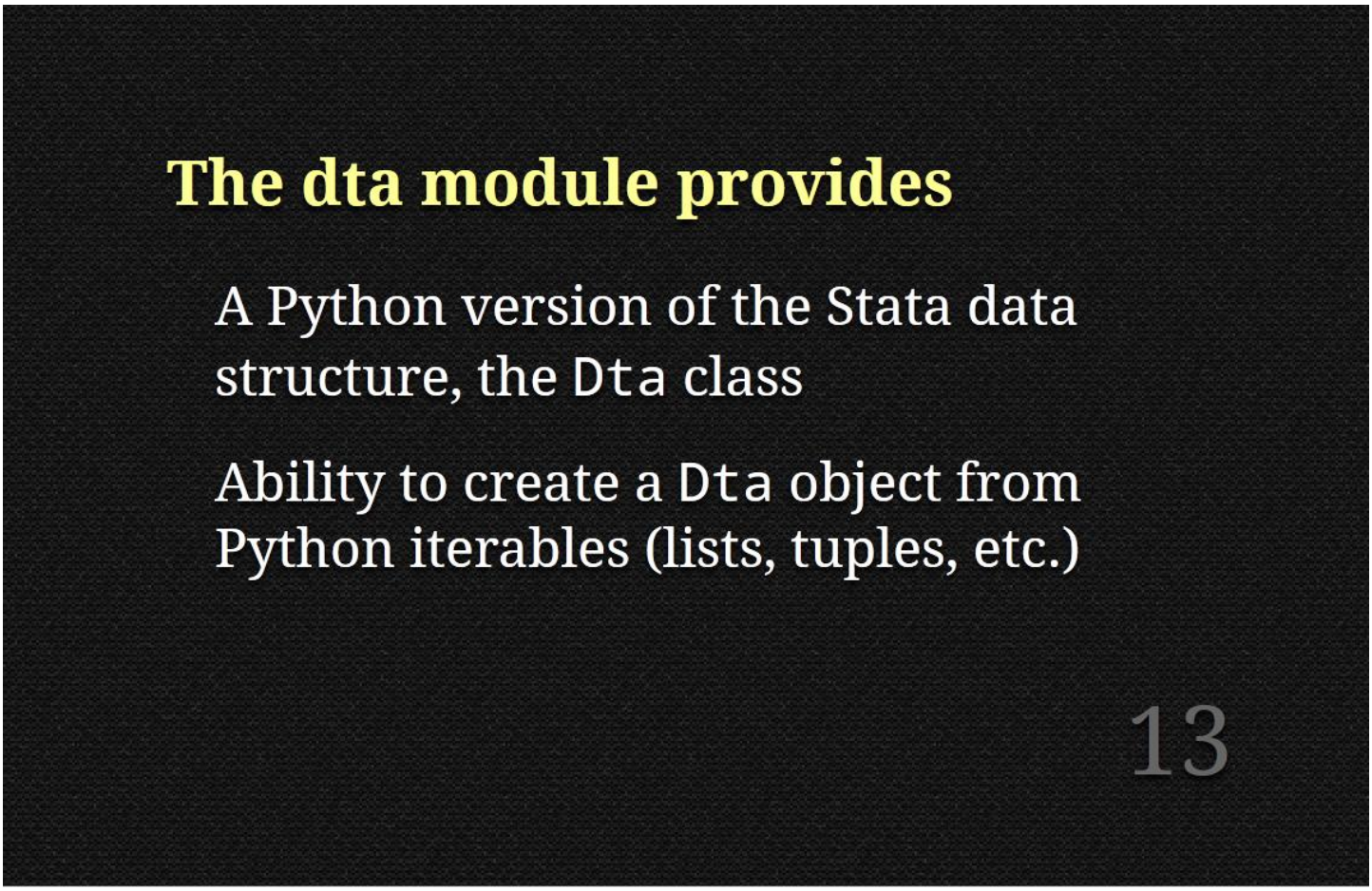


Sympy can also be used to find the functional form of derivatives and integrals. Here is a command for graphing a function with its derivative and integral.



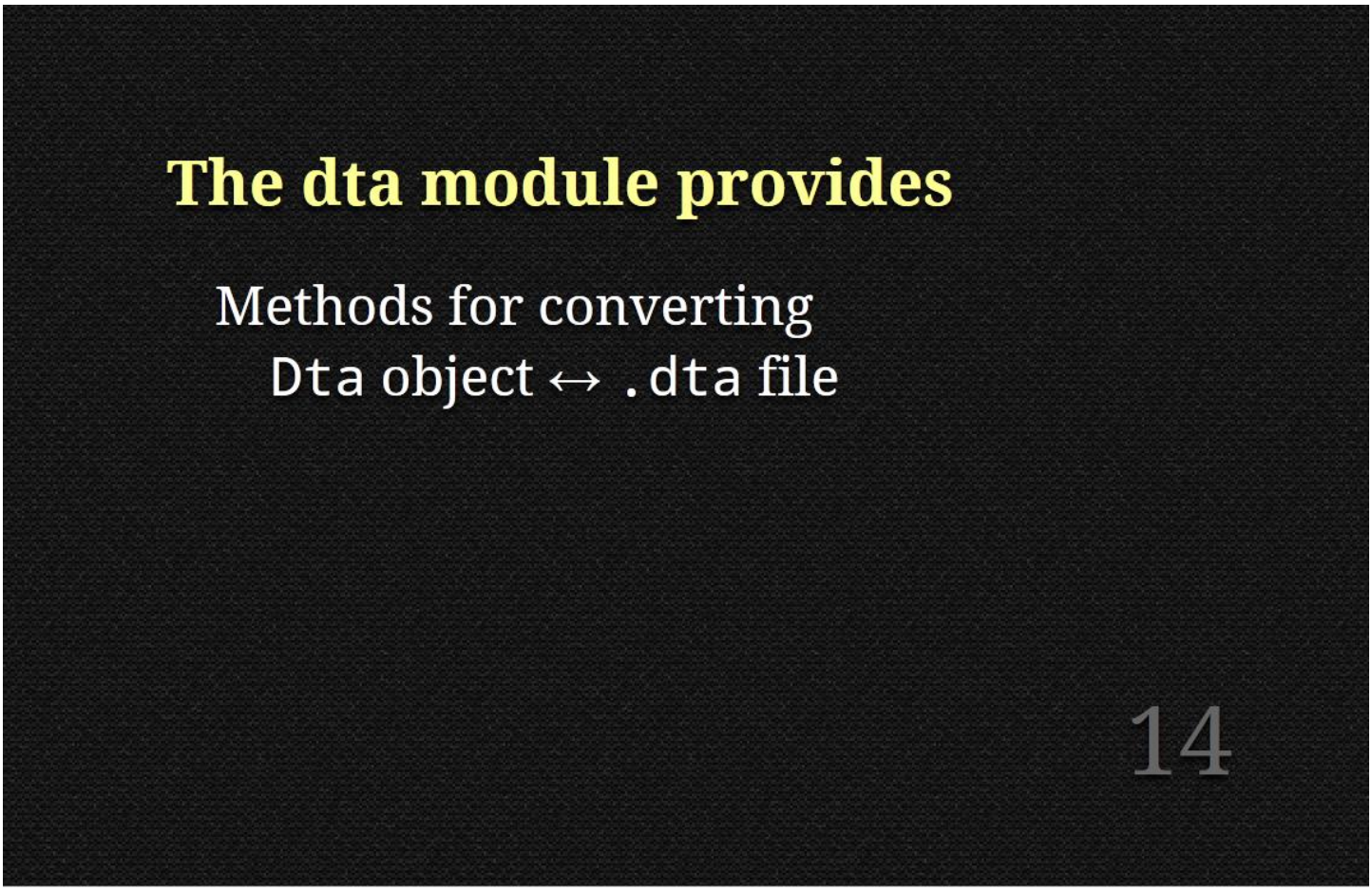
Looking ahead and imagining people using the Python plugin, I would guess a common use would be to build or modify datasets. Users might want to build their datasets from scratch or by importing CSV files, for example, then save the dataset as a dta file for analysis in Stata. Or they might want to open and modify existing dta files before analyzing them in Stata.

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



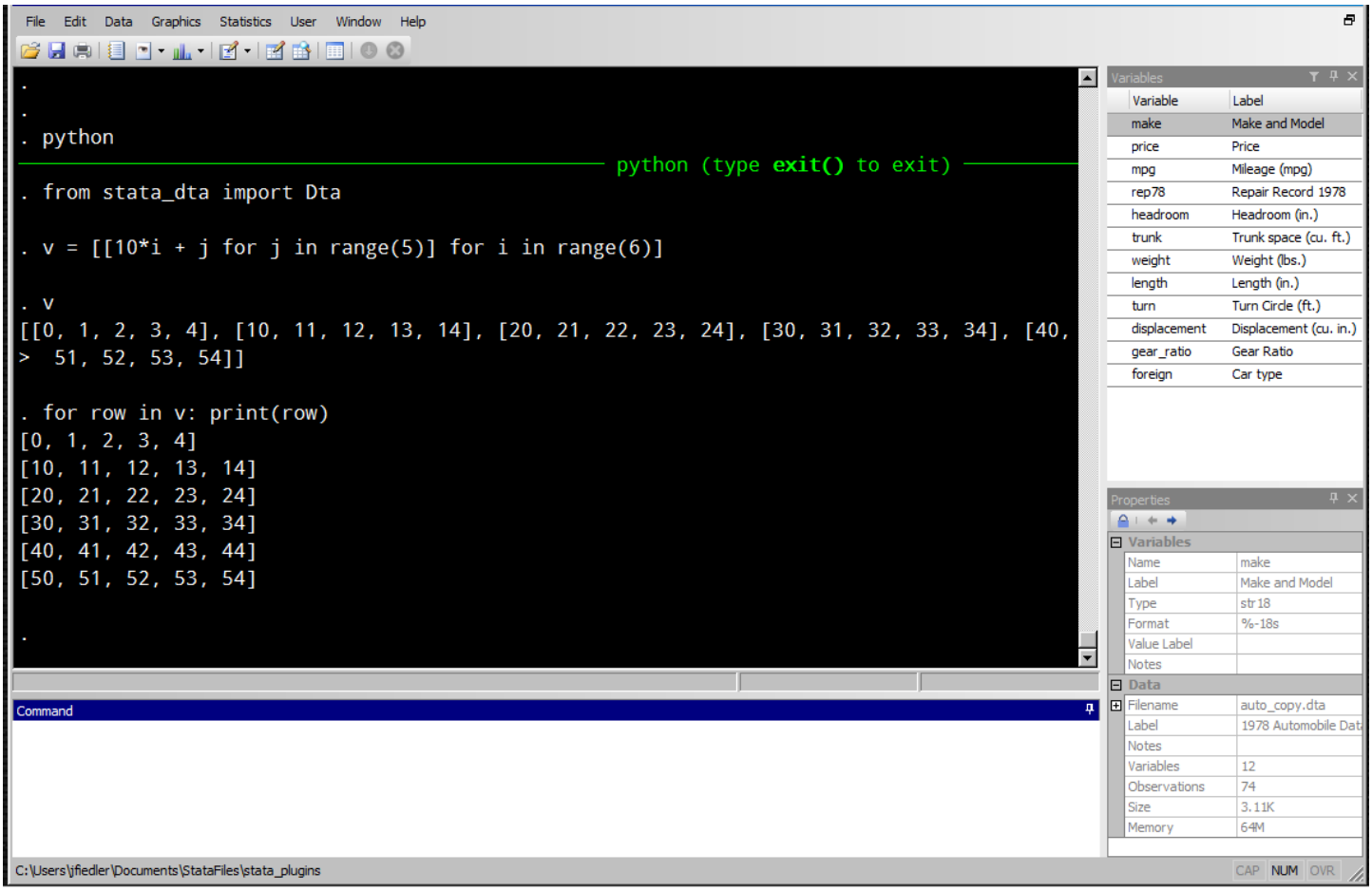
The module includes a complete Python version of the Stata data structure, or at least includes everything that a dta file would include.

The module provides the ability to create a Python dataset from Python array types ("iterable" types such as lists, tuples, generators, etc.).



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

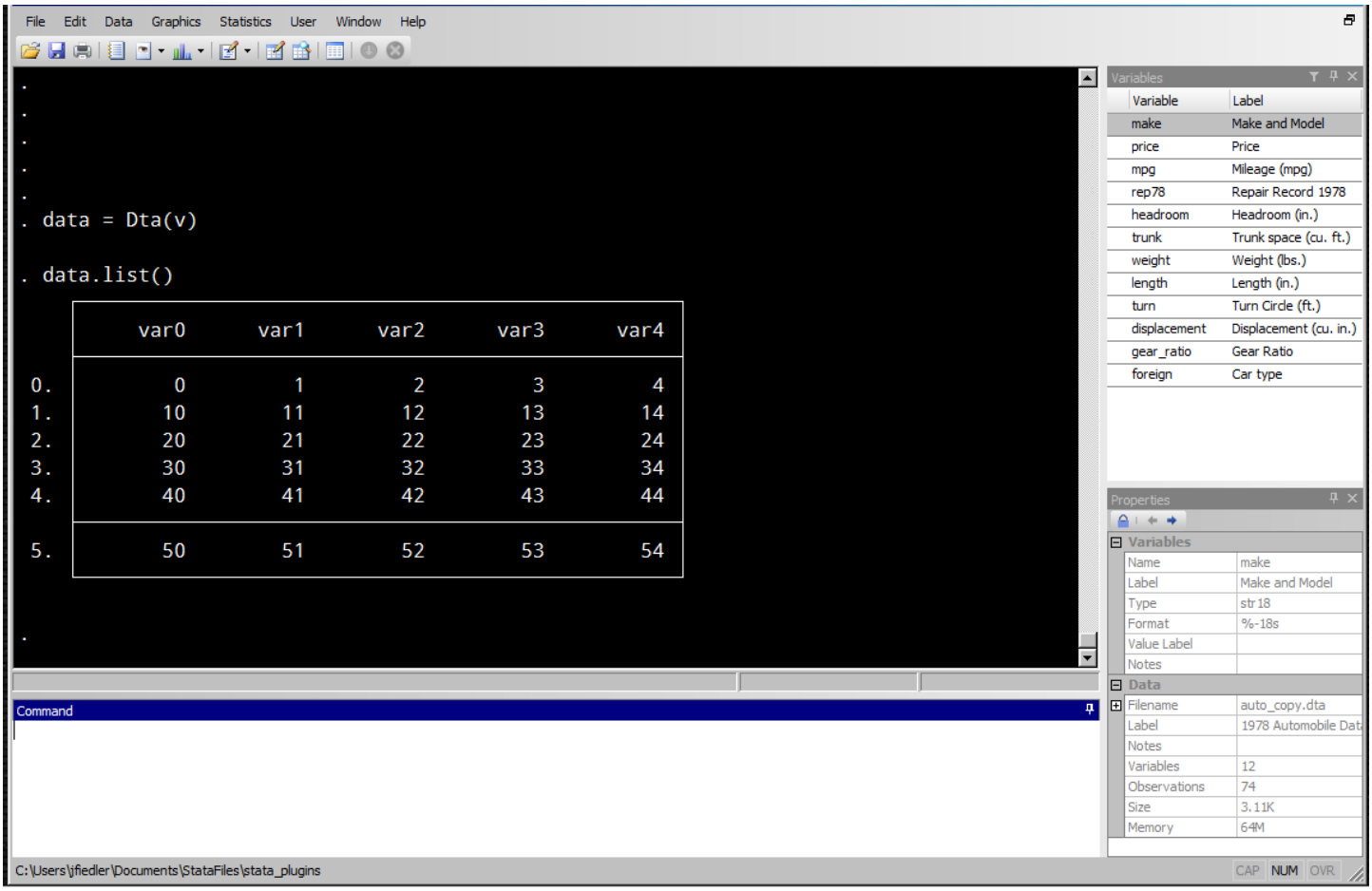
Demonstration



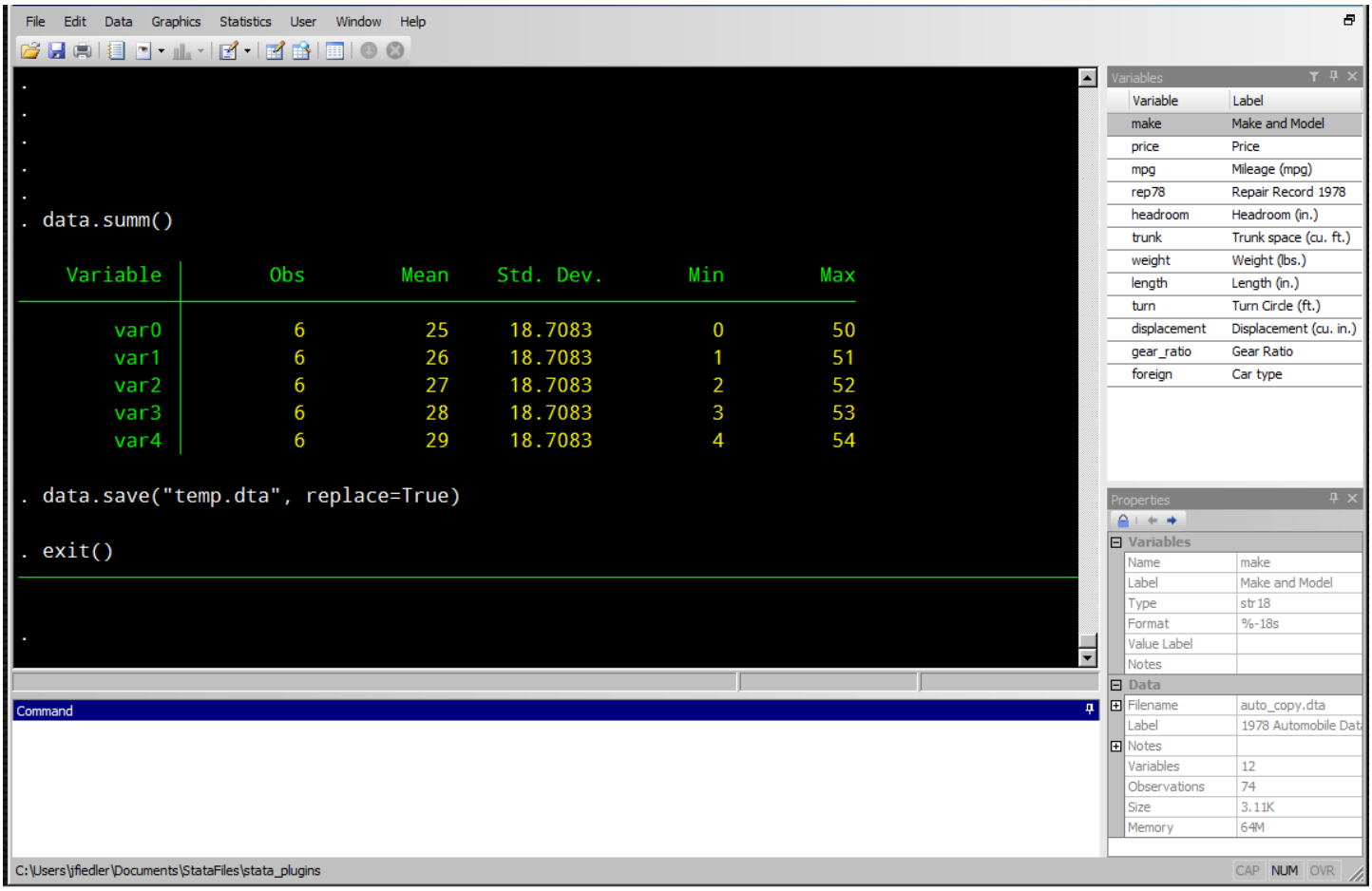
We need to do some preparatory work for the next example.

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

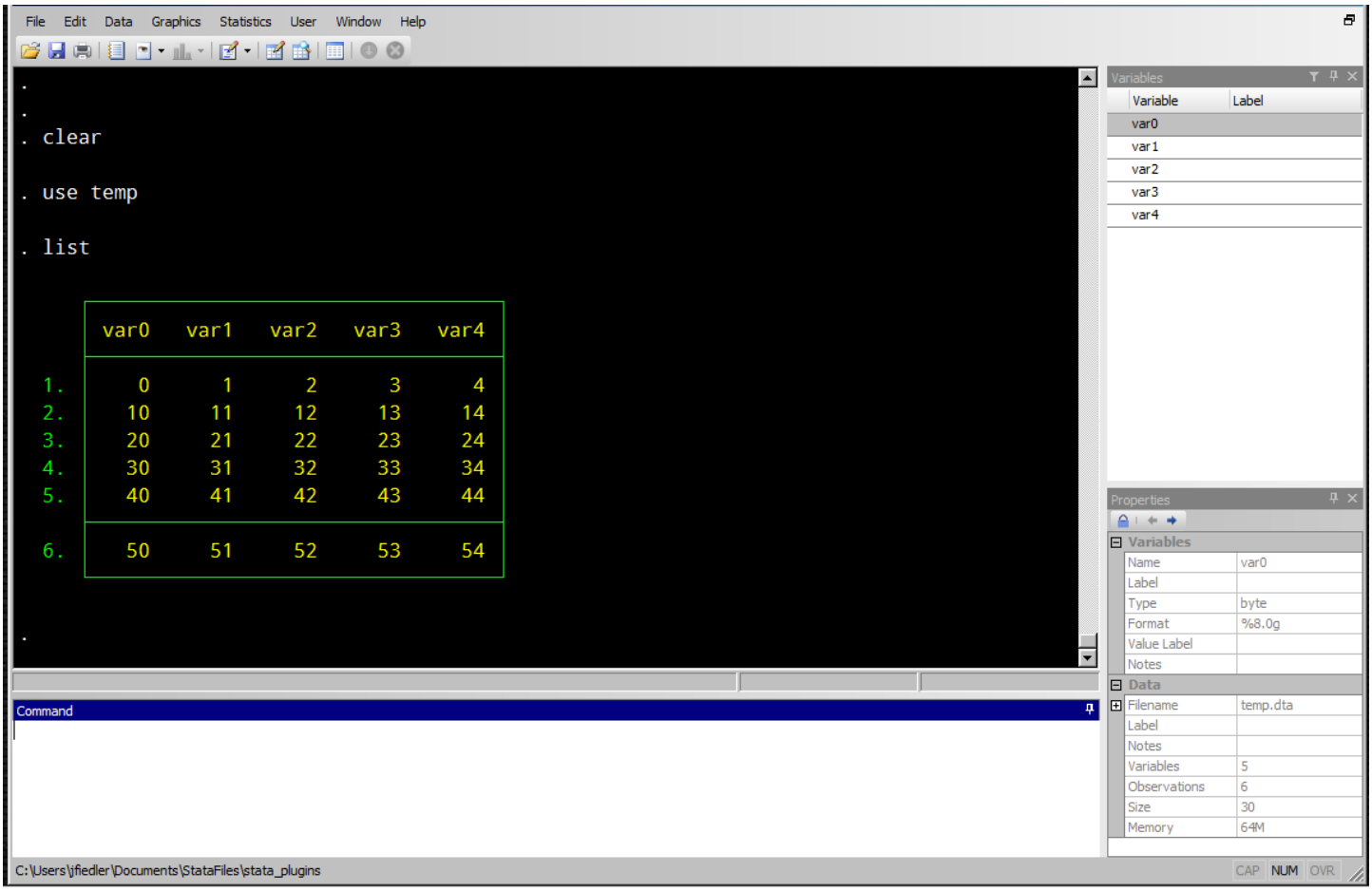
0. Start the Python interactive mode. (We were in ado mode before this command.)
1. The "dta module" is actually called `stata_dta`. This line brings `Dta`, the dataset constructor, into the interactive session.
2. This line creates an array of arrays of values called v.
3. This is just to show the value 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.



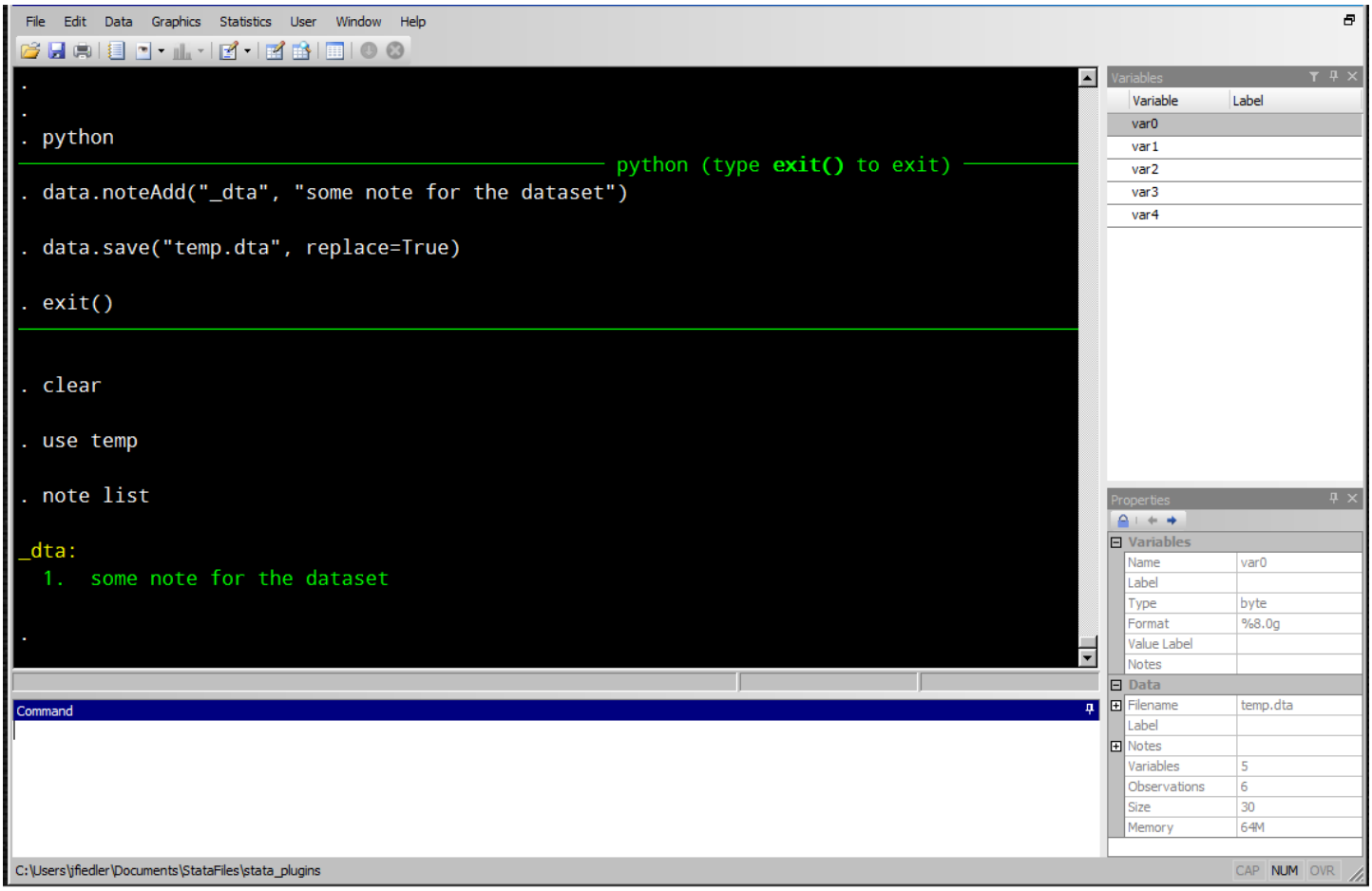
A dataset is created from `v` just by calling `Dta` with `v` as argument. To see the values in a dataset, the `Dta` class has a `list()` method.



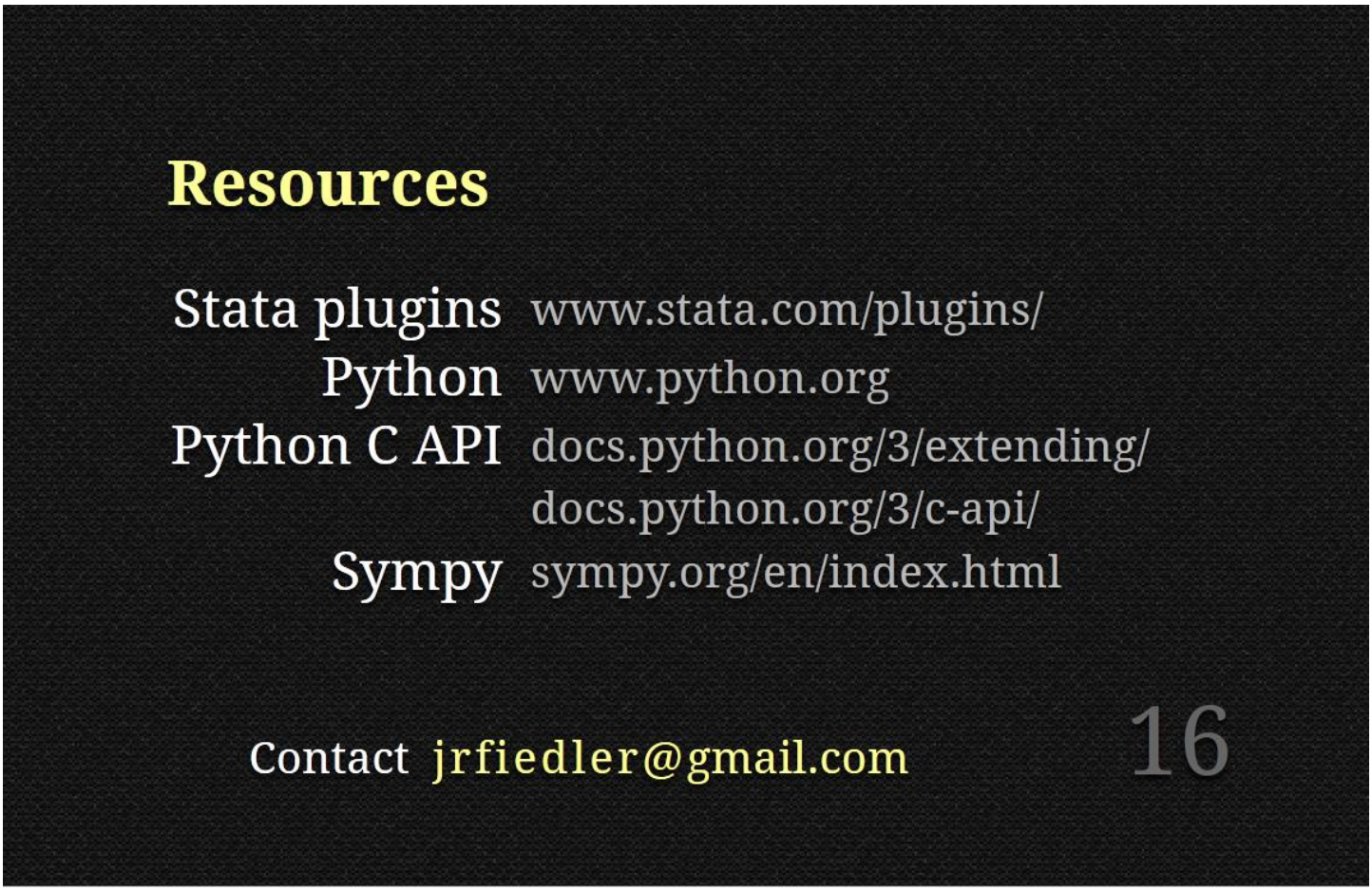
We can also summarize the data with the `Dta` class's `summ` method, and once we're satisfied with the dataset, we can save it as a `dta` file. Above, the optional `replace` argument is used because there's already a `temp.dta` file in this directory. In the last line above, `exit()` is used to return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



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



Python `Dta` objects can contain anything that a dta file can. The commands above show that notes can be added in Python. We see that when the dataset is saved (in Python) and opened in Stata, the note is there.


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
16

The end.