The Python plugin for Stata, version 0.2.0

James Fiedler

Contents

1 Introduction

This document describes a Stata plugin for embedding the Python programming language within Stata. In short, the plugin gives the user the ability to use Python to interact with Stata data values, matrices, macros, and numeric scalars. The plugin can be used interactively inside the Stata GUI, or can be used to execute Python files. Python files can be used separately or in combination with `.ado` or `.do` files.

This code has been tested only on Windows 7, 64-bit computers, in Stata versions 12.1 and 13.0, with Python 3.3. Python 3.2 and 3.1 can probably be used instead of Python 3.3, but that has not been tested. The plugin will not work with Python 2. The code was developed for Stata 12.1 but works in Stata 13.0, except that string values must be ASCII and be no more than 244 characters long. In other words, the code works in Stata 13.0 when used with string values allowed in Stata 12.1.

Users will need to compile the plugin themselves. Instructions for compiling on Windows are given in §4. Users will need Stata, Python (specifically, CPython, the most common version), and a C compiler. Users will also need access to the file `Python.h`, which is included in many distributions of Python. The Windows installer at `http://www.python.org/getit/` will install all of the Python files you need.

This document assumes the reader has some experience with Python, but extensive experience is not required.

2 What's new

Changes in version 0.2.0

- `st_Matrix` and `st_View` (capital `M` and `V`) have been replaced with `st_matrix` and `st_view`.
- The usual "`>>>`" Python prompt has been added to help differentiate Python mode from Stata Ado mode. Unfortunately, the default dot prompt remains, so the full prompt is "`>>>.`".
- New `st_mirror` function. See the description of `st_mirror` in §8.2. See example usage in §11.10.

- New `stata_math` module. See §9 and see §11.10 for example usage with `st_mirror`.
- `python.ado` will now search for Python files when using the `file` option. The referenced Python file can be anywhere in your Ado path.
- In the `st_matrix` and `st_view` returned objects (instances of `StataMatrix` and `StataView` classes), the "camelCase" method names have been replaced with "underscore_case" names. Also, these objects now do not show their contents as their default representation. To see their contents, use their `list` method, as in examples §11.8 and §11.9.

3 Use with caution

The plugin and helper files described here are experimental. Save your data before using the plugin. There is currently one known limitation/bug which can crash Stata. There may be other, unknown bugs that can crash Stata, too.

3.1 Limitations

1. Dropping a Stata program that uses the Python plugin and then re-running it can crash Stata, depending on what Python modules are used in the program, and whether it's the only Stata program that uses the plugin. For many Python modules this is not a problem. Nor does it seem to be a problem to drop and re-run `python.ado`, even if it's the only program using the plugin.

   Remedy: It's not clear what is causing this problem, but there seems to be a simple solution. If wanting to drop a program that uses the plugin, make sure that another program also uses it—for example, use `python.ado` at least once—or declare the plugin in Stata directly, with `program python_plugin, plugin`.

2. The interactive Python interpreter within Stata is limited to single-line inputs. Unfortunately there is no remedy for this at the moment. With some creativity, though, quite a bit of Python code can be packed into a single line, or combinations of single-line inputs. If more than one line of input is needed in a single statement, you can write the code in a Python `.py` file, and run the file using the `file` option of `python.ado` or `import` it in an interactive session.

3. The Stata GUI's Break button does not interrupt the plugin. There is not recourse for infinite loops in the plugin besides closing Stata.

4. The plugin does not have continuous access to user input. Python code requiring continuous control over `stdin`, such as the `input()` function, will not work.

5. Calling `sys.exit()` in a Python file will close Stata. In the interactive interpreter, `sys.exit()` may be safely used to exit the plugin only.

4 Installing

The necessary files for this project, besides those that come with your Python installation, are

- `stplugin.h` (from `http://www.stata.com/plugins/`)
- `stplugin.c` (from `http://www.stata.com/plugins/`)
- `python_plugin.c`
- `python.ado`
- `stata.py`
- `stata_missing.py`

4.1 Windows, using Visual Studio Express

Below are the steps I used for compiling the plugin using Visual Studio Express 2012 and Python version 3.3 on Windows 7. StataCorp has notes for compiling plugins for other versions of Visual Studio at `http://www.stata.com/plugins/`

0. You will need Stata, Python, and Visual Studio Express 2012 installed. You will also need the Stata plugin header file `stplugin.h` and C file `stplugin.c` from section 2 of `http://www.stata.com/plugins/`.
1. Open Visual Studio. From the main menu at the top, select **File** > **New Project**.
2. A window pops up. Under the menu on the left, expand the **Visual C++** item, then select **Win32**. In the center pane of the window choose **Win32 Project**. On the bottom, change the name and solution name, if desired, then click **OK**.
3. Another window pops up. Click on **Next**. On the next screen, under **Application type**, choose **DLL**. Below that, check the box for **empty project**. Click on **Finish**.
4. In the main application window, on the right hand side, find **Resource Files**. Right click, select **Add** > **Existing Item**. Add each of these (you might have to right click and choose **Add** > **Existing Item** multiple times):
   (a) `python_plugin.c`
   (b) `stplugin.h`

(c) `stplugin.c`

(d) `python33.lib` (for me this resides in `C:/Python33/libs`)

5. Under **Resource Files**, click on `python_plugin.c` so that it's highlighted. In the main menu bar (at the top of the Visual Studio) select **VIEW** > **Property Pages**.

   A new window pops up. On the left, select **C/C++** > **General**. On the right, click in the field next to **Additional Include Directories**, and type in the directory for Python.h (for me it is `C:/Python33/include`). Press enter or click on **OK**.

6. At the top, find **Debug** *below* the main menu bar (not the **DEBUG** *in* the main menu bar), and change this to **Release**. (Alternately, you could rename the Python file `python33.lib` to `python33_d.lib`.)

   You might have to repeat this and the previous step if you make other changes to settings or do these steps out of order.

7. If you have an x64 machine, change the field next to **Debug** from **Win32** to **x64**. This will require several steps. First, click on the field to open the menu, and choose **Configuration Manager...**. A new window pops up. Under platform, select **New...**, then select x64. Click on **OK**, then **Close**.

8. In the main menu bar select **BUILD** > **Build Solution** or use the shortcut, F7. You should get a message in the Output window, below the main window, that says the project was successfully compiled.

9. Rename the compiled dll (if necessary) to `python_plugin.plugin`.

   Using the default settings, for me the compiled dll is found in

   `C:/Users/<my username>/My Documents/Visual Studio 2012/ Projects/<project name>/x64/Release`

   (with `<my username>` and `<project name>` replaced).

   Put `python_plugin.plugin` and `python.ado` in Stata's Ado path (in Stata use command `adopath` to see the Ado path), and put `stata.py` and `stata_missing.py` in Python's path (in Python use `import sys` then `sys.path` to see directories in the path).

   As an alternative to putting files in the Ado path and/or the Python path, you can put some or all of these files into a common directory and `cd` to that directory in Stata before first calling the plugin. This works because the current working directory is always in the Ado path, and the directory in which the Python plugin was first called will be in the Python path.

10. Open Stata and type `python`. If everything has worked, this should start an interactive session of Python within Stata. A horizontal line should appear, with text to indicate a Python interactive session has started,

similar to when starting an interactive session of Mata. Try some of the examples from §11. If error messages and results are not printed to the screen, check to make sure `stata.py` is somewhere that Python can find it.

4.2 Mac OS X

(thanks to Kit Baum for working on this)

The plugin was successfully installed on Mac OS X with the following steps. First, make sure that Python3.3 is installed. An OS X installer can be found at `http://www.python.org/getit/`. After installing Python3.3, you might need change the definition of `python` to point to the `python3.3` executable. You can do this by renaming `/usr/local/python` to `/usr/local/python2.7` (assuming Python2.7 is the default version) and then adding a symlink from `/usr/local/python` to `/usr/local/bin/python3.3`.

You will also need `gcc`. You can get `gcc` with Xcode (`https://developer.apple.com/xcode/`), or "Command Line Tools for Xcode" (see, for example, `http://www.mkyong.com/mac/how-to-install-gcc-compiler-on-mac-os-x/`).

Next, make sure `python_plugin.c`, `stplugin.c`, and `stplugin.h` reside in the same directory. To compile the plugin, start with the compiler command from `http://www.stata.com/plugins/`, modified for this plugin:

```
gcc -bundle -DSYSTEM=APPLEMAC stplugin.c
  python_plugin.c -o python_plugin.plugin
```

Add to that compiler and linker flags for Python, which can be obtained as in `http://docs.python.org/3.3/extending/embedding.html#compiling-and-linking-under-unix-like-systems`.

After compiling, `python_plugin.plugin` and `python.ado` need to be put in Stata's Ado path and `stata.py` and `stata_missing.py` need to be put in the Python path. Alternately, any or all of these files can be in the directory from which the `python` command is first invoked, because that directory should be in both the Ado path and Python path.

5 Syntax of `python.ado`

The syntax for `python.ado` is

```
python [varlist] [if] [in] [, file(some_file.py)
                              args(some_args) ]
```

If no file is specified, an interactive session is begun. The number of arguments in the `args` option is stored in Stata local `_pynargs`, and the arguments are stored in `_pyarg0`, `_pyarg1`, etc. The number of variables in the varlist and their names are stored in Stata locals `_pynvars`, and `_pyvar0`, `_pyvar1`, etc. (see example in §11.13).

If a file is specified and it is not specified with an asolute path, then the file is searched for in the `adopath` (not the Python path). Thus you can keep Python files with related `.ado` or `.do` files by giving the Python file a similar name. To prevent searches along the `adopath`, specify the absolute path the file.

There is one drawback to using `python.ado` rather than using the plugin directly. With `python.ado` the user will have access only to those locals defined within `python.ado` or within the Python session or script. Any locals defined interactively before starting an interactive session will be invisible if using `python.ado` to invoke the interactive session. See example in §11.7.

6 Using the plugin directly

6.1 Syntax

Two ways of calling the plugin will be shown here, the minimal syntax, which is not generally recommended, and then the recommended syntax, which is more cumbersome. With either syntax, the plugin will need to be introduced to Stata with `program python_plugin, plugin`.

Contrary to usual plugin usage, arguments for the plugin should not be included in the plugin call. Arguments can instead be put in locals and accessed through `st_local` inside the plugin, as done in `python.ado` (see §5).

The mininal syntax for calling the plugin is

```
plugin call python_plugin [varlist] [, file_name ]
```

but this syntax should not be used if wanting to interact with Stata variables. In fact, because of reasons described below, the plugin has been written so that the varlist is seen to be empty when using the minimal syntax. If wanting to use the plugin to interact with Stata variables, the plugin should be called with

```
ereturn clear
local _pynallvars = 0
if (c(k) > 0) {
  foreach var of varlist * {
    local _pyallvars`_pynallvars' = "`var'"
    local _pynallvars = `_pynallvars' + 1
  }
}
plugin call python_plugin `=cond(c(k) > 0, "*", "")' ///
    [, file_name ]
```

With either version, the plugin runs the file if `file_name` is given. Unlike with the `python` command, a specified `file_name` is not searched for. You must provide the path, or the file must be in the same directory. The Mata function `findfile()` is useful for finding the path to a file in the `adopath` (see §11.13).

If no `file_name` is given, an interactive session is begun. With the recommended syntax above, variables of interest can be specified in locals, as is done in `python.ado` (see §5).

The recommended way to call the plugin solves several problems:

1. Estimation commands can introduce "hidden" variables that are partially visible in the plugin and occupy a position in the varlist, but cannot be interacted with (as far as I know).

   The purpose of the `ereturn clear` is to clear any hidden variables.

2. If a subset of variables could be specified, indexing of variables can be inconsistent between functions in the plugin. Some functions index relative to the specified set, and some index relative to the entire varlist (including hidden variables). Clearing hidden variables with `ereturn clear` and using `` `=cond(c(k) > 0, "*", "")' `` in the plugin call help to ensure that the indexing is consistent.

   The minimal, non-recommeded syntax above implies that a subset of variables can be specified. In fact, while the syntax is allowed, the plugin tries to disallow actually using subsets because of the problems discussed here.

3. The remainder of the extra lines in the second version provide the variable names to the plugin so that `st_varname` can look up names by index and `st_varindex` can return the index for a name. Supplying the variable names in this way also allows the C code to be written so that if the simpler, not-recommended syntax is used, the user is presented with an empty varlist rather than inconsistent indexing and hidden variables.

7 The `stata_missing` module

The purpose of the `stata_missing` module is to implement an analog of Stata's missing values. This is accomplished with the class `MissingValue`. The module contains analogs of the 27 usual missing values, `.`, `.a`, `.b`, etc., in a tuple called `MISSING_VALS`. Stata supports missing values other than these, but the `stata_missing` module does not. The analog of Stata's `.` missing value, `MISSING_VALS[0]`, is also given the name `MISSING` within the `stata_missing` module.

Users wanting direct access to analogs of Stata's missing values should use the existing instances of `MissingValue` rather than construct new instances. Users wanting to determine which instance of `MissingValue` corresponds to a large floating point number should use the function `getMissing`, which takes a single `float` or `int` argument and returns an instance of `MissingValue`.

Any plugin function that sets Stata numeric values can accept a `float`, `int`, `None`, or an instance of `MissingValue`. In such cases, `None` will translated into Stata's `.` missing value.

Example usage of the `stata_missing` module is given in §11.2, §11.8, and §11.9.

8 The stata module

The stata module provides functions for interacting with Stata variables, matrices, macros, and numeric scalars. The module is automatically imported with the first use of the plugin, just as if the user had typed from stata import *. The module is imported whether the plugin was invoked directly ("plugin call ...") or through python.ado, and for both the interactive interpreter and running files.

Almost all of the functionality provided by the stata module is mirrored by functionality in Mata (but not vice versa). In an effort to be easier to use, the functions in the stata module were made to mimic functions in Mata. For example, in Mata the function st_local retrieves the value of a local macro if given one argument (its name) or sets the value of the macro if given two arguments (name and value). In the stata module there is a function st_local with the same behavior. Of course, some changes had to be made between Mata and Python versions of functions. In Mata, when the user tries to access a nonexistent numeric scalar, an empty matrix J(0,0) is returned. Python has no inherent notion of a matrix, so instead the Python function raises a ValueError.

Many of the functions in the stata module have indexing arguments. Keep in mind that while Stata uses 1-based indexing (i.e., the first meaningful index is 1 for data variables, observations, and matrix elements), the Python convention is to begin indexing at 0. Thus, if the first variable in a dataset is numeric, its first observation can be obtained via either

```
_st_data(0, 0)
```

or

```
st_data(0, 0),
```

and its value can be changed via

```
_st_store(0, 0, some_val)
```

or

```
st_store(0, 0, some_val).
```

8.1 List of functions

st_cols	st_ifobs	st_isnumfmt
_st_data	st_in1	st_isnumvar
st_data	st_in2	st_isstrfmt
_st_display	st_isfmt	st_isstrvar
_st_error	st_islmname	st_isvarname
st_format	st_ismissing	st_local
st_global	st_isname	st_matrix

st_matrix_el	_st_sdata	st_varindex
st_mirror	st_sdata	st_varname
st_nobs	_st_sstore	st_view
st_numscalar	st_sstore	st_viewobs
st_nvar	_st_store	st_viewvars
st_rows	st_store	

8.2 Function descriptions

`st_cols(matname)`

arguments: `matname` str
returns: int

Get number of columns in given matrix. Returns 0 if there is no Stata matrix with name `matname`.

`_st_data(obsnum, varnum)`

arguments: `obsnum` int
`varnum` int
returns: float or `MissingValue`

Get value in given observation and Stata numeric variable. The allowed argument values are

$$-\texttt{st\_nobs()} \leq \texttt{obsnum} < \texttt{st\_nobs()}$$

and

$$-\texttt{st\_nvar()} \leq \texttt{varnum} < \texttt{st\_nvar()}$$

(assuming plugin is called through `python.ado` or used in accordance with recommendations made in §6). Negative values are interpreted in the usual way for Python indices. Values outside of these ranges will cause an `IndexError`. Note this last detail is unlike in Mata, where `_st_data()` does not abort with error for invalid indices, but instead returns a `.` missing value.

`st_data(obsnums, vars)`

arguments: `obsnums` single int or iterable of int
`vars` single int, single str, or iterable of int or str
returns: list of lists of float or `MissingValue`

Get values in given observations and given nuemric Stata variables. The function returns a list of lists, with one sub-list for each observation. See §11.5 for example usage and return values.

This function uses `_st_data()`, so `obsnums` and `var` (if integer) can be negative and if out of range will raise an `IndexError`. If strings are used in `vars`,

a `ValueError` will be raised for ambiguous or incorrect abbreviations.

`_st_display(text)`

arguments: `text` str
returns: `None`

Print text in Stata's results window, with included SMCL tags interpreted. The usual `print` function is routed through `_st_display`, so there's usually no need to call `_st_display` directly. Unlike most other functions listed here, this function is not autmatically imported into the main namespace. To use it, first import it with `from stata import _st_display`.

`_st_error(text)`

arguments: `text` str
returns: `None`

Print text as error. There's usually no need to call this function directly. Python errors are automatically routed through `_st_error`, and if wanting to display a message as an error, the user can simply use `print("{err}<message>")`. Like `_st_display`, this function is not automatically imported into the main namespace. To use it, first import it with `from stata import _st_error`.

`st_format(fmt, value)`

arguments: `fmt` str
`value` int, float, `MissingValue`, or `None`
returns: bool

Return string representation of `value` according to Stata format given in `fmt`. The first argument should be a valid Stata format, but the function will return a meaningful string regardless.

`st_global(macroname)`
`st_global(macroname, value)`

with 1 argument:
arguments: `macroname` str
returns: str

with 2 arguments:
arguments: `macroname` str
`value` str
returns: None

Get value from given global macro if using 1-argument version, or set the value of the global macro if using the 2-argument version. In the 1-argument version,

if the global macro does not exist the return value will be the empty string. In either version, if the global macro name is malformed a `ValueError` will be raised.

Unlike Mata's `st_global`, the `st_global` here cannot access characteristics and cannot access `r()`, `e()`, `s()`, and `c()` macros.

`st_ifobs(obsnum)`

arguments: `obsnum` int
returns: bool

Query the `if` condition (specified when invoking `python` or the plugin) for the given observation number. If no `if` condition was specified, this will evaluate to `True` for all observations. The allowed values for `obsnum` are

$$-\texttt{st\_nobs()} \leq \texttt{obsnum} < \texttt{st\_nobs()}$$

with negative values interpreted in the usual way. Values outside this range will cause an `IndexError`.

`st_in1()`

returns: int

Get the first index in the `in` range (specified when invoking `python` or the plugin). If no `in` was specified, this will evaluate to `0`.

`st_in2()`

returns: int

Get the second index of the `in` range (specified when invoking `python` or the plugin), plus one. If no `in` condition was specified, this will return the maximum observation index, plus one. The reason for returning the first index *beyond* the `in` range, rather that *last index within*, is to facilitate the common Python syntax of index slicing. For example, if Python variable `v` is an instance of `st_view` (see below), then `v[st_in1():st_in2(), ]` would be a reference to the observations within the `in` condition. (See §11 for other examples of slice indexing.)

`st_isfmt(fmt)`

arguments: `fmt` str
returns: bool

Determine if given `fmt` is a valid Stata format. In calendar formats, the calendar name is not checked for validity.

`st_islmname(name)`

arguments: `name` str
returns: bool

Determine if given `name` is a valid local macro name.

`st_ismissing(value)`

arguments: `value` any Python object
returns: bool

Determine if the given value is considered a missing value. The function returns `False` if the value is not a `float`, `int`, `MissingValue` instance or `None`. It returns `True` if `value` is `None` or a `MissingValue` instance. If `value` is `float`, the function tests whether the value is inside the non-missing range for doubles in Stata, which is approximately $[-1.798 \times 10^{308},\ 8.988 \times 10^{307}]$ (see `help dta` in Stata, specifically, "Representation of numbers"), returning `True` if it's outside this range, `False` if it's inside this range.

`st_isname(name)`

arguments: `name` str
returns: bool

Determine if given `name` is a valid name, for example, for scalars or global macros. To test for validity as a local macro name use `st_islmname`. To test for validity as a Stata variable name use `st_isvarname`.

`st_isnumfmt(fmt)`

arguments: `fmt` str
returns: bool

Determine if given `fmt` is a valid Stata numeric format. Numeric formats are any valid formats that are not string formats.

`st_isnumvar(var)`

arguments: `var` int or str
returns: bool

Determine if given Stata variable is a numeric variable. The variable can be specified by its integer index, by its name, or by abbreviation of its name. If `var` is an integer, then it should be in the range

$$-\texttt{st\_nvar()} \leq \texttt{var} < \texttt{st\_nvar()}$$

with negative values interpreted in the usual way. Values outside this range will cause an `IndexError`. If `var` is a string, an invalid or ambiguous abbreviation will cause a `ValueError`.

`st_isstrfmt(fmt)`

arguments: `fmt` str
returns: bool

Determine if given `fmt` is a valid Stata string format.

`st_isstrvar(var)`

arguments: `var` int or str
returns: bool

Check whether given Stata variable is a string variable. The variable can be specified by its integer index, by its name, or by abbreviation of its name. If `var` is an integer, then it should be in the range

$$-\texttt{st\_nvar()} \leq \texttt{var} < \texttt{st\_nvar()}$$

with negative values interpreted in the usual way. Values outside this range will cause an `IndexError`. If `var` is a string, an invalid or ambiguous abbreviation will cause a `ValueError`.

`st_isvarname(name)`

arguments: `name` str
returns: bool

Determine if given `name` is a valid Stata variable name. See manual [U] §11.3 Naming conventions.

`st_local(macroname)`
`st_local(macroname, value)`

with 1 argument:
arguments: `macroname` str
returns: str

with 2 arguments:
arguments: `macroname` str
`value` str
returns: None

Get value from given local macro if using 1-argument version, or set the value of the local macro if using the 2-argument version. In the 1-argument version, if the local macro does not exist the return value will be the empty string. In

either version, if the local name is malformed a `ValueError` will be raised.

`st_matrix(matname)`

arguments: `matname` str
returns: instance of `StataMatrix` class

This function creates a *view* onto a Stata matrix. See §11.9 for example usage. If no matrix is found with name `matname`, a `ValueError` is raised.

Unlike Mata's `st_matrix`, the object returned by the `st_matrix` here is only a view on the Stata matrix. If you change elements of the Python object you are actually changing the matrix in Stata. Also unlike Mata's `st_matrix`, the `st_matrix` here cannot access `r()` and `e()` matrices.

`st_matrix_el(matname, row, col)`
`st_matrix_el(matname, row, col, value)`

with 3 arguments:
arguments: `matname` str
`row` int
`col` int
returns: float or `MissingValue`

with 4 arguments:
arguments: `matname` str
`row` int
`col` int
`value` int, float, `MissingValue`, or `None`
returns: `None`

Get the value in the given matrix, row, and column if using the 3-argument version, or replace the value if using the 4-argument version. If no matrix is found with name `matname`, a `ValueError` is raised.

`st_mirror()`

returns: instance of `StataMirror` class

This function creates a view onto the current Stata data set. Unlike `st_view`, the object returned by `st_mirror` is 'aware' of changes made in Stata. However, the main advantage of `st_mirror` is that the returned object provides quick access to Stata variables as attributes. See §11.10 for example usage. See §11.11 for a comparison of `st_mirror` and `st_view`.

`st_nobs()`

returns: int

Get the number of observations in the current Stata data set.

`st_numscalar(scalarname)`
`st_numscalar(scalarname, value)`

with 1 argument:
arguments: `scalarname` str
returns: float or `MissingValue`

with 2 arguments:
arguments: `scalarname` str
`value` int, float, `MissingValue`, or `None`
returns: `None`

Get contents of given numeric scalar if using 1-argument version, or set the contents if using 2-argument version. In the 1-argument version, if the scalar does not exist a `ValueError` will be raised. Note this is unlike Mata, where `st_numscalar` returns `J(0,0,.)` if the scalar does not exist. In both 1-argument and 2-argument versions, if the scalar name is malformed a `ValueError` will be raised.

Unlike Mata's `st_numscalar`, this `st_numscalar` cannot access `r()`, `e()`, and `c()` numeric scalars.

`st_nvar()`

returns: int

Get the number of Stata variables in the current data set.

`st_rows(matname)`

arguments: `matname` str
returns: int

Get the number of rows in given matrix. Returns 0 if there is no Stata matrix with name `matname`.

`_st_sdata(obsnum, varnum)`

arguments: `obsnum` int
`varnum` int
returns: str

Get value in given Stata string variable and observation. The allowed argument values are

$$-\texttt{st\_nobs()} \leq \texttt{obsnum} < \texttt{st\_nobs()}$$

and

$$-\texttt{st\_nvar()} \leq \texttt{varnum} < \texttt{st\_nvar()}$$

(assuming plugin is called through `python.ado` or used in accordance with recommendations made in §6). Negative values are interpreted in the usual way for Python indices. Values outside of these ranges will cause an `IndexError`. Note this is unlike in Mata, where `_st_sdata()` does not abort with error for invalid indices, but instead returns an empty string.

`st_sdata(obsnums, vars)`

arguments:	`obsnums`	single int or iterable of int
	`vars`	single int, single str, or iterable of int or str
returns:	list of lists of str	

Get values in given observations and given Stata string variables. The function returns a list of lists, with one sub-list for each observation. See §11.5 for example usage and return values.

This function uses `_st_sdata()`, so `obsnums` and `var` (if integer) can be negative and if out of range will raise an `IndexError`. If strings are used in `vars`, a `ValueError` will be raised for ambiguous or incorrect abbreviations.

`_st_sstore(obsnum, varnum, value)`

arguments:	`obsnum`	int
	`varnum`	int
	`value`	str
returns:	`None`	

Set value in given Stata string variable in given observation. The allowed argument values are

$$-\texttt{st\_nobs()} \leq \texttt{obsnum} < \texttt{st\_nobs()}$$

and

$$-\texttt{st\_nvar()} \leq \texttt{varnum} < \texttt{st\_nvar()}$$

(assuming plugin is called through `python.ado` or used in accordance with recommendations made in §6). Negative values are interpreted in the usual way for Python indices. Values outside of these ranges will cause an `IndexError`. Note this is unlike in Mata, where `_st_sstore()` does not abort with error for invalid indices.

`st_sstore(obsnums, vars, values)`

arguments:	`obsnums`	single int or iterable of int
	`vars`	single int, single string, or iterable of int or str
	`values`	iterable of str
returns:	None	

Set values in given observations and given Stata string variables. The dimensions of the input `values` should match the dimensions implied by `obsnums` and `vars`.

For example, if `obsnums` is `(0,1,2)` and `vars` is `(2,4)` (and if those are valid for the loaded data set), then any of these input `values` would be valid:

```
values = [['a','b'], ['c','d'], ['e','f']]
values = (('a','b'), ('c','d'), ('e','f'))
values = (['a','b'], ['c','d'], ['e','f'])
values = [['a']*2]*3
```

and these would be invalid:

```
values = [['a','b','c'], ['d','e','f']]
values = (('a','b','c','d','e','f'))
values = ('a','b','c','d','e','f')
```

See §11.5 for other examples.

This function uses `_st_sstore()`, so `obsnums` and `var` (if integer) can be negative and if out of range will raise an `IndexError`. If there is an invalid index, some values may be set before the `IndexError` is raised. If strings are used in `vars`, a `ValueError` will be raised for ambiguous or incorrect abbreviations.

`_st_store(obsnum, varnum, value)`

arguments:	`obsnum`	int
	`varnum`	int
	`value`	int, float, `MissingValue`, or `None`
returns:	`None`	

Set value in given Stata numeric variable in given observation. The allowed argument values are

$$-\texttt{st\_nobs()} \leq \texttt{obsnum} < \texttt{st\_nobs()}$$

and

$$-\texttt{st\_nvar()} \leq \texttt{varnum} < \texttt{st\_nvar()}$$

(assuming plugin is called through `python.ado` or used in accordance with recommendations made in §6). Negative values are interpreted in the usual way for Python indices. Values outside of these ranges will cause an `IndexError`. Note this is unlike in Mata, where `_st_store()` does not abort with error for invalid indices.

`st_store(obsnums, vars, values)`

arguments:	`obsnums`	single int or iterable of int
	`vars`	single int, single string, or iterable of int or str
	`values`	iterable of int, float, `MissingValue`, or None
returns:	None	

Set values in given observations and given Stata numeric variables. The dimensions of the input `values` should match the dimensions implied by `obsnums` and `var`. For example, if `obsnums` is `(0,1,2)` and `vars` is `(2,4)` (and if those are valid for the loaded data set), then any of these input `values` would be valid:

```
values = [[0,1], [2,3], [4,5]]
values = ((0,1), (2,3), (4,5))
values = ([0,1], [2,3], [4,5])
values = [[0]*2]*3
```

and these would be invalid:

```
values = [[0,1,2], [3,4,5]]
values = ((0,1,2,3,4,5))
values = (0,1,2,3,4,5)
```

See §11.5 for other examples.

This function uses `_st_store()`, so `obsnums` and `var` (if integer) can be negative and if out of range will raise an `IndexError`. If there is an invalid index, some values may be set before the `IndexError` is raised. If strings are used in `vars`, a `ValueError` will be raised for ambiguous or incorrect abbreviations.

`st_varindex(varname)`
`st_varindex(varname, abbr_ok)`

arguments:	`varname`	str
	`abbr_ok`	bool or coercible to bool
returns:	int	

Find the index of the given Stata variable. Abbreviations are allowed if using the two-argument version and the second argument is truthy. Otherwise, `varname` must match a Stata variable name exactly. A `ValueError` will be raised if the text does not match a Stata variable or if the abbreviation is ambiguous. Unlike Mata's `st_varindex`, this `st_varindex` only allows a single name or abbreviation per call.

`st_varname(varnum)`

arguments:	`varnum`	int
returns:	str	

Return the name of the Stata variable at the given index. The allowed argument values are

$$-\texttt{st\_nvar()} \leq \texttt{varnum} < \texttt{st\_nvar()}$$

(assuming plugin is called through `python.ado` or used in accordance with recommendations made in §6). Negative values are interpreted in the usual way

for Python indices. A value outside of this ranges will cause an `IndexError`.

`st_view(rownums, varnums, selectvar)`

arguments:	`obsnums`	single int or iterable of int
	`varnums`	single int or iterable of int
	`selectvar`	string, int, `None`, or `MissingValue` instance
returns:	instance of `StataView` class	

This function returns a view onto the current Stata data set. See §11.8 for example usage. Unlike Mata's `st_view`, the `st_view` here allows access to both numeric and string variables.

If `rownums` is not specified, is `None`, or is an instance of `MissingValue`, `rownums` will be set to all possible row numbers. Likewise for `varnums`.

As in Mata's `st_view`, `selectvar` is used to indicate which rows will be included. If `selectvar` is not specified or is the empty string, all rows in `rownums` will be included. If `selectvar` is set to an int or string representing a data variable, all rows in `rownums` will be included where the `selectvar` variable is non-zero. If `selectvar` is `None` or is a `MissingValue` instance, all rows in `rownums` will be included where *none* of the `varnums` variables are missing.

`st_viewobs(view_obj)`

arguments:	`view_obj`	instance of `StataView`
returns:	tuple of int	

Return tuple containing observation numbers in the `StataView` instance.

`st_viewvars(view_obj)`

arguments:	`view_obj`	instance of `StataView`
returns:	tuple of int	

Return tuple containing the variable indices in the `StataView` instance.

9 The `stata_math` module

Python's built-in math functions won't work with missing values or Stata variables. Users could check each input to verify that it's not a missing value, or use `try ... except` blocks with each function invocation. For example,

```
. sysuse auto
(1978 Automobile Data)
. python
------------------------------------------------- python (type exit() to exit) -----
>>>. v = st_view()
```

```
>>>. v[:4, :4].list()
  obs: 4
 vars: 4
            c0        c1        c2        c3
r0 AMC Concord      4099        22         3
r1   AMC Pacer      4749        17         3
r2  AMC Spirit      3799        22         .
r3 Buick Centu      4816        20         3
>>>. v.get(2, 3)
.
>>>. from math import sin
>>>. sin(v.get(2, 2))
-0.008851309290403876
>>>. sin(v.get(2, 3))
Traceback (most recent call last):
  File "<string>", line 1, in <module>
TypeError: a float is required
>>>. from stata_math import st_sin
>>>. st_sin(v.get(2, 2))
-0.008851309290403876
>>>. st_sin(v.get(2, 3))
.
>>>. exit()
```

The `stata_math` module provides Python versions of all of Stata's math functions. These functions understand missing values and they understand Stata variables obtained from `st_mirror`. See §11.10 for example usage with `st_mirror`.

9.1 List of functions

`st_abs`	`st_digamma`	`st_min`
`st_acos`	`st_exp`	`st_mod`
`st_acosh`	`st_floor`	`st_reldif`
`st_asin`	`st_int`	`st_round`
`st_asinh`	`st_invcloglog`	`st_sign`
`st_atan`	`st_invlogit`	`st_sin`
`st_atan2`	`st_ln`	`st_sinh`
`st_atanh`	`st_lnfactorial`	`st_sqrt`
`st_ceil`	`st_lngamma`	`st_sum`
`st_cloglog`	`st_log`	`st_tan`
`st_comb`	`st_log10`	`st_tanh`
`st_cos`	`st_logit`	`st_trigamma`
`st_cosh`	`st_max`	

Usage of each is similar to the corresponding Stata function. Two differences are knwon: `st_round` rounds differently because Python 3 uses "banker's rounding", and `st_trigamma` gives values that are very similar but different. For more information, use the Python `help` function:

```
>>>. from stata_math import st_round
>>>. help(st_round)
Help on function st_round in module stata_math:
st_round(x, y=1)
    Rounding function.

    Parameters
    ----------
    x : float, int, MissingValue instance, or None
    y : float, int, MissingValue instance, or None;
        y is optional, default value is 1

    Returns
    -------
    If both x and y are non-missing, returns x / y rounded to
        the nearest integer times y (but see notes below).
    If y is 1 or y is not specified, returns x rounded to the
        nearest integer (but see notes below).
    If y is zero, returns x.
    If y is missing, MISSING (".") is returned.
    If x is missing and y is non-missing, returns MissingValue
        corresponding to x.
    If both x and y are missing, returns MISSING (".").

    Notes
    -----
    Though Python 3 uses "banker´s rounding" or "round half to even",
    this function uses "round half up". For example, with Python 3´s
    `round` function, `round(3.5)` and `round(4.5)` are both 4, but
    `st_round(3.5)` is 4 and `st_round(4.5)` is 5.

    Keep in mind that floating point imprecision of inputs may affect
    the output.
```

10 Miscellanea

You can use `python.ado` or the plugin to run a python file in `.do` and `.ado` files. You can also start an interactive session from `.do` or `.ado` files, but you cannot use Python statements in `.do` or `.ado` files.

If an interactive session is begun in a `.do` or `.ado` file, execution of that file is effectively halted. When the interactive interpreter is exited, execution of the file resumes from that point. Here is an example of a file called `do_example.do` that starts an interactive Python session:

```
noi di "in do file"
noi python
noi di "back in do file"
```

Example usage:

```
. run do_example
in do file
```

```
------------------------------------------------------- python (type exit() to exit)
>>>. "in python"
´in python´
>>>. exit()
--------------------------------------------------------------------------------
back in do file
```

Here is another example, with a file called `ado_example.ado` (see §6):

```
program ado_example
  noi di "in ado_example"
  plugin call python_plugin
  noi di "back in ado_example"
  noi di "`scname' = " scalar(`scname')
end

program python_plugin, plugin
```

Example usage:

```
. ado_example
in ado_example
------------------------------------------------------- python (type exit() to exit)
>>>. st_local("scname", "the_scalar")
>>>. st_numscalar("the_scalar", 12345)
>>>. exit()
--------------------------------------------------------------------------------
back in ado_example
the_scalar = 12345
```

11 Examples

11.1 The interactive interpreter takes single-line inputs

The error below comes from trying to use a multi-line statement (the user hit the `enter` key after typing `for i in range(5):`). The following lines in the example show ways to squeeze moderately complicated statements into a single line.

```
. python
------------------------------------------------------- python (type exit() to exit)
>>>. for i in range(5):
  File "<string>", line 1
    for i in range(5):
                      ^
SyntaxError: unexpected EOF while parsing
>>>. for i in range(5): print(i)
0
1
2
```

```
3
4
>>>. def mlf(): print("multi-", end="") ; print("line", end=" ") ; print("function
> ")
>>>. mlf()
multi-line function
>>>. exit()
```

11.2 Missing values

In plugin functions that set the value of a numeric quantity, `None` can be used in input to represent Stata's `.` missing value. In general, though, `None` should not be thought of as the analog of Stata's `.` value.

Missing values are implemented in the `stata_missing` module, see §7, and will often have to be imported before using directly. In the following example, though, notice that `st_numscalar("new")` returned a `MissingValue` instance before anything was imported from `stata_missing`.

```
. python
                                                  python (type exit() to exit)
>>>. st_numscalar("new", None)
>>>. st_numscalar("new")
.
>>>. 0 < 100 < float("inf")
True
>>>. 0 < . < float("inf")
  File "<string>", line 1
    0 < . < float("inf")
        ^
SyntaxError: invalid syntax
>>>. from stata_missing import MISSING as mv
>>>. 0 < mv < float("inf")
True
>>>. mv
.
>>>. mv.value
8.98846567431158e+307
>>>. from stata_missing import MISSING_VALS as mvs
>>>. mvs
(., .a, .b, .c, .d, .e, .f, .g, .h, .i, .j, .k, .l, .m, .n, .o, .p, .q, .r, .s,
>  .t, .u, .v, .w, .x, .y, .z)
>>>. st_numscalar("new", mvs[14])
>>>. st_numscalar("new")
.n
>>>. mv == mvs[0]
True
>>>. exit()
```

11.3 Basic functions

```
. clear
. sysuse auto
(1978 Automobile Data)
. list make-trunk in 1/5
```

	make	price	mpg	rep78	headroom	trunk
1.	AMC Concord	4,099	22	3	2.5	11
2.	AMC Pacer	4,749	17	3	3.0	11
3.	AMC Spirit	3,799	22	.	3.0	12
4.	Buick Century	4,816	20	3	4.5	16
5.	Buick Electra	7,827	15	4	4.0	20

```
. python
------------------------------------------------------ python (type exit() to exit)
>>>. st_varindex("not_a_variable")
variable not_a_variable not found
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ValueError: no Stata variable found
>>>. st_varindex("make")
0
>>>. st_isstrvar(0)
True
>>>. _st_sdata(0,0)
´AMC Concord´
>>>. st_varindex("rep")
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ValueError: no Stata variable found (abbrev. not allowed)
>>>. st_varindex("rep78")
3
>>>. st_varindex("rep", True)
3
>>>. st_isnumvar("rep")
True
>>>. _st_data(0,3)
3.0
>>>. exit()
-------------------------------------------------------------------------------
```

11.4 Stata variable types do not change on replacement

In Stata, if you replace a numeric variable's value with a value outside its type range, the value gets promoted (unless the original type is `float`). For example, the range of non-missing values in `byte` is −127 to 100. If you replace a `byte` value with something outside of this range, the variable will be promoted to a type that can hold larger values. If you replace an integer variable value with a non-integer like 1.5, the type will be promoted to `double`. However, if you make these replacements in Python, the type will not be promoted. If you replace

with a value outside the type's range, a missing value will be inserted. If you replace an integer variable value with a non-integer like 1.5, the value will be truncated to an integer. (By the way, this also happens when replacing Stata variable values using Mata.)

First, in Stata.

```
. clear
. set obs 1
obs was 0, now 1
. gen byte b = 0
. gen int i = 0
. gen long l = 0
. replace b = 101 in 1
b was byte now int
(1 real change made)
. replace i = 120000 in 1
i was int now long
(1 real change made)
. replace l = 1.5 in 1
l was long now double
(1 real change made)
. list
```

	b	i	l
1.	101	120000	1.5

Now in Python.

```
. clear
. set obs 1
obs was 0, now 1
. gen byte b = 0
. gen int i = 0
. gen long l = 0
. python
------------------------------------------------------ python (type exit() to exit)
>>>. _st_store(0, 0, 101)
>>>. _st_store(0, 1, 120000)
>>>. _st_store(0, 2, 1.5)
>>>. exit()
--------------------------------------------------------------------------------
. list
```

	b	i	l
1.	.	.	1

11.5 Data and store functions

This example demonstrates the usage of functions that get and set Stata data values: `st_data`, `st_store`, `st_sdata`, and `st_sstore`. Some of the other data functions are used elsewhere, `_st_data` and `_st_sdata` in §11.3 and the `st_view` class in §11.8.

We will use a copy of the auto data set rather than the original because values will be replaced.

```
. clear
. sysuse auto
(1978 Automobile Data)
. save auto_copy
file auto_copy.dta saved
. python
---------------------------------------------------------------------- python (type exit() to exit)
>>>. st_data(0, 0)
Traceback (most recent call last):
  File "<string>", line 1, in <module>
  File "stata.py", line 206, in st_data
    raise TypeError("only numeric Stata variables allowed")
TypeError: only numeric Stata variables allowed
>>>. st_sdata(0, 0)
[[´AMC Concord´]]
>>>. st_sdata(range(0,74,10), 0)
[[´AMC Concord´], [´Cad. Deville´], [´Dodge Diplomat´], [´Merc. Marquis´], [´Ol
> ds Toronado´], [´Pont. Phoenix´], [´Honda Accord´], [´VW Diesel´]]
>>>. for row in st_sdata(range(0,74,10), 0): print(row)
[´AMC Concord´]
[´Cad. Deville´]
[´Dodge Diplomat´]
[´Merc. Marquis´]
[´Olds Toronado´]
[´Pont. Phoenix´]
[´Honda Accord´]
[´VW Diesel´]
>>>. st_data(0, 1)
[[4099.0]]
>>>. st_data(0, range(1,12,3))
[[4099.0, 2.5, 186.0, 3.5799999237060547]]
>>>. st_data(range(0,74,10), range(1,12,3))
[[4099.0, 2.5, 186.0, 3.5799999237060547], [11385.0, 4.0, 221.0, 2.279999971389
> 7705], [4010.0, 4.0, 206.0, 2.4700000286102295], [6165.0, 3.5, 212.0, 2.25999
> 9990463257], [10371.0, 3.5, 206.0, 2.4100000858306885], [4424.0, 3.5, 203.0,
> 3.0799999237060547], [5799.0, 3.0, 172.0, 3.049999952316284], [5397.0, 3.0, 1
> 55.0, 3.7799999713897705]]
>>>. for row in st_data(range(0,74,10), range(1,12,3)): print(row)
[4099.0, 2.5, 186.0, 3.5799999237060547]
[11385.0, 4.0, 221.0, 2.2799999713897705]
[4010.0, 4.0, 206.0, 2.4700000286102295]
[6165.0, 3.5, 212.0, 2.259999990463257]
[10371.0, 3.5, 206.0, 2.4100000858306885]
[4424.0, 3.5, 203.0, 3.0799999237060547]
[5799.0, 3.0, 172.0, 3.049999952316284]
[5397.0, 3.0, 155.0, 3.7799999713897705]
```

```
>>>. st_store(range(0,74,10), range(1,12,3), [[None]*3])
Traceback (most recent call last):
  File "<string>", line 1, in <module>
  File "stata.py", line 226, in st_store
    obs, cols, vals = _parseObsColsVals(obs, cols, vals)
  File "stata.py", line 193, in _parseObsColsVals
    raise ValueError("length of value does not match number of rows")
ValueError: length of value does not match number of rows
>>>. st_store(range(0,74,10), range(1,12,3), [[None]*3]*8)
Traceback (most recent call last):
  File "<string>", line 1, in <module>
  File "stata.py", line 226, in st_store
    obs, cols, vals = _parseObsColsVals(obs, cols, vals)
  File "stata.py", line 195, in _parseObsColsVals
    raise ValueError("inner dimensions do not match number of columns")
ValueError: inner dimensions do not match number of columns
>>>. st_store(range(0,74,10), range(1,12,3), [[None]*4]*8)
>>>. for row in st_data(range(0,74,10), range(1,12,3)): print(row)
[., ., ., .]
[., ., ., .]
[., ., ., .]
[., ., ., .]
[., ., ., .]
[., ., ., .]
[., ., ., .]
[., ., ., .]
>>>. exit()
```

```
. list price head length gear if mod(_n-1, 10) == 0
```

	price	headroom	length	gear_r~o
1.	.	.	.	.
11.	.	.	.	.
21.	.	.	.	.
31.	.	.	.	.
41.	.	.	.	.
51.	.	.	.	.
61.	.	.	.	.
71.	.	.	.	.

```
. python
```

---

python (type exit() to exit)

```
>>>. from stata_missing import MISSING_VALS as mvs
>>>. [ 0, 1, mvs[1], mvs[4] ]
[0, 1, .a, .d]
>>>. st_store(range(0,74,10), range(1,12,3), [ [0, 1, mvs[1], mvs[4]] ]*8)
>>>. for row in st_data(range(0,74,10), range(1,12,3)): print(row)
[0.0, 1.0, .a, .d]
[0.0, 1.0, .a, .d]
[0.0, 1.0, .a, .d]
[0.0, 1.0, .a, .d]
[0.0, 1.0, .a, .d]
[0.0, 1.0, .a, .d]
[0.0, 1.0, .a, .d]
[0.0, 1.0, .a, .d]
```

```
>>>. exit()
```

```
. list price head length gear if mod(_n-1, 10) == 0
```

	price	headroom	length	gear_r~o
1.	0	1.0	.a	.d
11.	0	1.0	.a	.d
21.	0	1.0	.a	.d
31.	0	1.0	.a	.d
41.	0	1.0	.a	.d
51.	0	1.0	.a	.d
61.	0	1.0	.a	.d
71.	0	1.0	.a	.d

11.6 Data and store functions, string indices

This example repeats part of the previous example, but uses string indices for Stata variables. In the last few inputs, notice that the string indices can appear in a single string or in an iterable of separate strings, or both. String indices can contain the entire name of a variable, or any non-ambiguous abbreviation.

```
. clear
. use auto_copy
(1978 Automobile Data)
. python
```

```
>>>. st_data(0, "make")
Traceback (most recent call last):
  File "<string>", line 1, in <module>
  File "stata.py", line 20
> 6, in st_data
    raise TypeError("only numeric Stata variables allowed")
TypeError: only numeric Stata variables allowed
>>>. st_sdata(0, "make")
[[´AMC Concord´]]
>>>. st_sdata(range(0,74,10), 0)
[[´AMC Concord´], [´Cad. Deville´], [´Dodge Diplomat´], [´Merc. Marquis´], [´Ol
> ds Toronado´], [´Pont. Phoenix´], [´Honda Accord´], [´VW Diesel´]]
>>>. st_sdata(range(0,74,10), "make")
[[´AMC Concord´], [´Cad. Deville´], [´Dodge Diplomat´], [´Merc. Marquis´], [´Ol
> ds Toronado´], [´Pont. Phoenix´], [´Honda Accord´], [´VW Diesel´]]
>>>. for row in st_sdata(range(0,74,10), "make"): print(row)
[´AMC Concord´]
[´Cad. Deville´]
[´Dodge Diplomat´]
[´Merc. Marquis´]
[´Olds Toronado´]
[´Pont. Phoenix´]
[´Honda Accord´]
[´VW Diesel´]
>>>. st_data(0, "price")
[[4099.0]]
```

```
>>>. st_data(0, "price headroom length gear_ratio")
[[4099.0, 2.5, 186.0, 3.5799999237060547]]
>>>. st_data(0, "price headroom length gear_ratio") == st_data(0, "pr he le ge")
True
>>>. st_data(0, "price head length gear") == st_data(0, ("pr", "he", "le", "ge"))
True
>>>. st_data(0, "price head length gear") == st_data(0, ("pr he", "le ge"))
True
>>>. st_data(0, "price head length gear") == st_data(0, ("pr", 4, 7, "ge"))
True
>>>. exit()
```

11.7 Accessing locals

This example begins with defining local and global macros in Stata, then using `python.ado` and the Python plugin to access them.

```
. local a = "a local"
. global b = "a global"
. python
------------------------------------------------------------------------------ python (type exit() to exit)
>>>. st_local("a")
´´
>>>. st_global("b")
´a global´
>>>. st_local("a", "modified")
>>>. exit()
```

```
. di "`a´"
a local
```

The attempt to access and modify a local defined outside of `python.ado` failed because, when using `python.ado`, the plugin only has access to locals defined within `python.ado`. (The following doesn't use the recommended syntax from §6, but that's ok because we're not interacting with Stata variables.)

```
. program python_plugin, plugin
. plugin call python_plugin
------------------------------------------------------------------------------ python (type exit() to exit)
>>>. st_local("a")
´a local´
>>>. st_global("b")
´a global´
>>>. st_local("a", "modified")
>>>. exit()
```

```
. di "`a´"
modified
```

11.8 Using `st_view`

This example demonstrates use of `st_view` with the `auto` data set. Here we use a copy of the auto data set because we'll be assigning new values. We drop some data variables to make the output less wide.

```
. clear
. use auto_copy
(1978 Automobile Data)
. drop turn-foreign
. python
------------------------------------------------------------ python (type exit() to exit)
>>>. v = st_view()
>>>. v
<stata.StataView object at 0x0000000002834C88>
>>>. v.list()
  obs: 74
 vars:  8
             c0       c1       c2       c3       c4       c5       c6       c7
 r0 AMC Concord     4099       22        3      2.5       11     2930      186
 r1   AMC Pacer     4749       17        3        3       11     3350      173
 r2  AMC Spirit     3799       22        .        3       12     2640      168
 r3 Buick Centu     4816       20        3      4.5       16     3250      196
 r4 Buick Elect     7827       15        4        4       20     4080      222
 r5 Buick LeSab     5788       18        3        4       21     3670      218
 r6  Buick Opel     4453       26        .        3       10     2230      170
 r7 Buick Regal     5189       20        3        2       16     3280      200
 r8 Buick Rivie    10372       16        3      3.5       17     3880      207

--output shortened--
```

The last output has been shortened to save space; all 74 rows appear in the Stata output. If you want to see less output, or if you want select a subset of the data, you'll want to use indexing.

Indexing is done by appending [*rows*, *cols*] to the `st_view` instance, where *rows* and *cols* are either integers, iterable of integers (tuple, list, etc.), or slices (e.g., 3:10 to denote $3, 4, \ldots, 10$ or 3:10:2 to denote $3, 5, 7, 9$). The *cols* index is optional, but the separating comma is not optional, with or without *cols*.

The syntax for slices is *start*:*stop*:*step* and denotes "every $step^{\text{th}}$ value beginning at *start*, up to, but not including, *stop*". For example, `4:16:3` denotes $4, 7, 10, 13$, but not 16. Any of *start*, *stop*, *step* can be omitted, and if *step* is omitted then the second colon can also be omitted. An omitted *start* is taken to be zero. If *stop* is omitted the slice extends as far as possible in the context. An omitted *step* is taken to be 1. For example, `4::` is equivalent to `4:` is equivalent to `4:`(max+1)`:1`.

Indexing an object returned by `st_view` always returns another object of the same type. To get values out of an `st_view` instance, use *instance*`.to_list()` or *instance*`.get(`*row,col*`)`.

```
>>>. v[::6, ::2].list()
  obs: 13
```

```
 vars:  4
             c0        c2        c4        c6
 r0 AMC Concord        22       2.5      2930
 r6  Buick Opel        26         3      2230
r12 Cad. Sevill        21         3      4290
r18  Chev. Nova        19       3.5      3430
r24 Ford Mustan        21         2      2650
r30 Merc. Marqu        15       3.5      3720
r36 Olds Cutlas        19       4.5      3300
r42 Plym. Champ        34       2.5      1800
r48 Pont. Grand        19         2      3210
r54    BMW 320i        25       2.5      2650
r60 Honda Accor        25         3      2240
r66 Toyota Celi        18       2.5      2410
r72 VW Scirocco        25         2      1990
>>>. v[0,0].list()
  obs: 1
 vars: 1
             c0
r0 AMC Concord
>>>. v[0, ].list()
  obs: 1
 vars: 8
             c0       c1       c2       c3       c4       c5       c6       c7
r0 AMC Concord     4099       22        3      2.5       11     2930      186
>>>. v[(0,1,2), (0,1,2)].list()
  obs: 3
 vars: 3
             c0        c1        c2
r0 AMC Concord      4099        22
r1   AMC Pacer      4749        17
r2  AMC Spirit      3799        22
>>>. v[:3, :3].list()
  obs: 3
 vars: 3
             c0        c1        c2
r0 AMC Concord      4099        22
r1   AMC Pacer      4749        17
r2  AMC Spirit      3799        22
>>>. v[:3, :3] = [["A", "not num", 2.02], ["B", 11.11, 12.12], ["C", 21.21, 22.22
> ]]
Traceback (most recent call last):
  File "<string>", line 1, in <module>
  File "stata.py", line 586, in __setitem__
    setters[col](row, col, value[i][j])
TypeError: set value should be float, None, or a missing value
```

The error was caused by trying to assign a string value to a numeric Stata variable.

```
>>>. v[:3, :3] = [["A", 1.01, 2.02], ["B", 11.11, 12.12], ["C", 21.21, 22.22]]
>>>. v[:3, :3]
  obs: 3
 vars: 3
             c0        c1        c2
```

```
r0          A         1         2
r1          B        11        12
r2          C        21        22
```

The Stata variables in columns 1 and 2 (Stata columns 2 and 3) are `price` and `mpg`, which are both integer type. When floating point values are assigned to these columns through the plugin, their values are truncated (see §11.4).

```
>>>. from stata_missing import MISSING_VALS as mvs
>>>. v[1:3, 1:3] = [[mvs[0], mvs[1]], [mvs[2], mvs[3]]]
>>>. v[:3, :3].list()
  obs: 3
 vars: 3
           c0        c1        c2
r0          A         1         2
r1          B         .        .a
r2          C        .b        .c
>>>. exit()
```

```
. list make-mpg in 1/3
```

	make	price	mpg
1.	A	1	2
2.	B	.	.a
3.	C	.b	.c

```
. clear
```

11.9 Using `st_matrix`

```
. clear
. sysuse auto
(1978 Automobile Data)
. mkmat mpg rep78 headroom in 1/5, matrix(m)
. matrix list m
m[5,3]
         mpg     rep78  headroom
r1        22         3       2.5
r2        17         3         3
r3        22         .         3
r4        20         3       4.5
r5        15         4         4
. python
```

```
>>>. m = st_matrix("m")
>>>. m.nrows
5
>>>. m.rows
(0, 1, 2, 3, 4)
>>>. m.ncols
3
```

```
>>>. m
<stata.StataMatrix object at 0x00000000035BC2B0>
>>>. m.list()
m[5,3]
           c0         c1         c2
r0         22          3        2.5
r1         17          3          3
r2         22          .          3
r3         20          3        4.5
r4         15          4          4
. m.to_list()
[[22.0, 3.0, 2.5], [17.0, 3.0, 3.0], [22.0, ., 3.0], [20.0, 3.0, 4.5], [15.0, 4
> .0, 4.0]]
```

An instance of `st_matrix` is mostly just a view onto the Stata matrix. If you want a static list of values, use the `to_list()` method, which returns a list of lists (one sub-list for each row), or the `get(`*row,col*`)` method, which returns a single entry.

The next parts of the example show the use of indexing to retrieve and set values in a sub-matrix. As with `st_view`, an `st_matrix` instance is indexed by appending [*rows*, *cols*] to it, where *rows* and *cols* are either integers, iterable of integers (tuple, list, etc.), or slices (for more info on slices, see example §11.8). The *cols* index is optional, but the separating comma is not optional, with or without *cols*. And, as with `st_view`, **indexing an object returned by `st_matrix` always returns the same kind of object**.

```
>>>. m[0,0]
<stata.StataMatrix object at 0x00000000036019B0>
>>>. m[0,0].list()
m[1,1]
           c0
r0         22
>>>. m[2,1].list()
m[1,1]
           c1
r2          .
>>>. m.get(2,1)
.
>>>. type(m.get(2,1))
<class ´stata_missing.MissingValue´>
>>>. from stata_missing import MISSING_VALS as mvs
>>>. m[(0,4), (0,2)].list()
m[2,2]
           c0         c2
r0         22        2.5
r4         15          4
>>>. mvs[1]
.a
>>>. m[(0,4), (0,2)] = mvs[1]
Traceback (most recent call last):
  File "<string>", line 1, in <module>
```

```
  File "stata.py", line 796, in __setitem__
    raise ValueError("length of value does not match number of rows")
ValueError: length of value does not match number of rows
>>>. [ [mvs[1]]*2 ]*2
[[.a, .a], [.a, .a]]
>>>. m[(0,4), (0,2)] = [ [mvs[1]]*2 ]*2
>>>. m.list()
m[5,3]
           c0         c1         c2
r0         .a          3         .a
r1         17          3          3
r2         22          .          3
r3         20          3        4.5
r4         .a          4         .a
>>>. m[(1,3), (0,2)] = [ [10101]*2 ]*2
>>>. m.list()
m[5,3]
           c0         c1         c2
r0         .a          3         .a
r1      10101          3      10101
r2         22          .          3
r3      10101          3      10101
r4         .a          4         .a
>>>. m[:, 1] = [[0.5]]*5
>>>. m.list()
m[5,3]
           c0         c1         c2
r0         .a         .5         .a
r1      10101         .5      10101
r2         22         .5          3
r3      10101         .5      10101
r4         .a         .5         .a
>>>. exit()
```

```
. matrix list m
m[5,3]
         mpg     rep78  headroom
r1         .a        .5         .a
r2     10101        .5     10101
r3        22        .5         3
r4     10101        .5     10101
r5         .a        .5         .a
```

11.10 Using `st_mirror`

Similar to `st_view`, `st_mirror` is used for accessing values from the currently loaded data set. In fact, the value returned from `st_mirror` (an instance of `StataMirror`) inherits much of its functionality from the `StataView` class (the class of the return value of `st_view`). The main purpose of adding `st_mirror` is to allows quick access to Stata variables. If `m = st_mirror()`, the variables of the current data set can be accessed as `m.varname_`, i.e., as an attribute with the variable name plus an underscore.

```
. clear
. use auto_copy
(1978 Automobile Data)
. python
──────────────────────────────────────────────────── python (type exit() to exit) ────
>>>. m = st_mirror()
>>>. m
<stata.StataMirror object at 0x000000000AF14FD0>
>>>. mpg = m.mpg_
>>>. mpg[:4]
[22.0, 17.0, 22.0, 20.0]
>>>. make = m.make_
>>>. for name in make[::10]: print(name)
AMC Concord
Cad. Deville
Dodge Diplomat
Merc. Marquis
Olds Toronado
Pont. Phoenix
Honda Accord
VW Diesel
```

You can use unambiguous abbreviations for the variable name, and you can use the `stata_math` functions on these mirror variables.

```
>>>. space = m.head_ + m.tru_
>>>. space[:16:2]
[13.5, 15.0, 24.0, 13.0, 20.5, 24.0, 16.0, 24.0]
>>>. from stata_math import st_round
>>>. st_round(space)[:16:2]
[14, 15, 24, 13, 20, 24, 16, 24]
```

As said above, the return value of `st_mirror` shares functionality with the return value of `st_view`. Compare the following lines with example §11.8.

```
. use auto_copy
(1978 Automobile Data)
. drop turn-foreign
. python
──────────────────────────────────────────────────── python (type exit() to exit) ────
>>>. [x for x in dir(st_view()) if not x.startswith("_")]
[´cols´, ´format´, ´get´, ´list´, ´ncols´, ´nrows´, ´rows´, ´to_list´]
>>>. [x for x in dir(st_mirror()) if not x.startswith("_")]
[´cols´, ´format´, ´get´, ´index´, ´list´, ´ncols´, ´nrows´, ´rows´, ´to_list´]
>>>. m[::6, ::2].list()

  obs: 13
 vars:  4

             c0        c2        c4        c6
 r0 AMC Concord        22       2.5      2930
 r6  Buick Opel        26         3      2230
r12 Cad. Sevill        21         3      4290
r18  Chev. Nova        19       3.5      3430
```

```
r24 Ford Mustan        21         2      2650
r30 Merc. Marqu        15       3.5      3720
r36 Olds Cutlas        19       4.5      3300
r42 Plym. Champ        34       2.5      1800
r48 Pont. Grand        19         2      3210
r54    BMW 320i        25       2.5      2650
r60 Honda Accor        25         3      2240
r66 Toyota Celi        18       2.5      2410
r72 VW Scirocco        25         2      1990
>>>. m[0,0].list()
  obs: 1
 vars: 1
            c0
r0 AMC Concord
>>>. m.get(0,0)
´AMC Concord´
>>>. m[:3,:3].list()
  obs: 3
 vars: 3
            c0        c1        c2
r0 AMC Concord      4099        22
r1   AMC Pacer      4749        17
r2  AMC Spirit      3799        22
>>>. m[:3,:3] = [["A", 1.01, 2.02], ["B", 11.11, 12.12], ["C", 21.21, 22.22]]
>>>. m[:3,:3].list()
  obs: 3
 vars: 3
            c0        c1        c2
r0           A         1         2
r1           B        11        12
r2           C        21        22
```

One advantage of `st_mirror` is that, unlike `st_matrix` or `st_view`, the object returned by `st_mirror` is aware of changes in the data set. (The following example continues from the previous one, using the `auto.dta` dataset with some variables dropped.)

```
>>>. m.nrows
74
>>>. m.ncols
8
>>>. exit()
```

```
. clear
. sysuse lifeexp
(Life expectancy, 1998)
. python
------------------------------------------------- python (type exit() to exit) ----
>>>. m.nrows
68
>>>. m.ncols
6
>>>. m.mpg_
```

```
Traceback (most recent call last):
  File "<string>", line 1, in <module>
  File "stata.py", line 639, in __getattr__
    varname = st_varname(st_varindex(name[:-1], True))
ValueError: no Stata variable found
```

```
>>>. m.country_[:4]
[´Albania´, ´Armenia´, ´Austria´, ´Azerbaijan´]
```

This benefit of being aware of data set changes doesn't extend to objects derived by subscripting. Subscripting a `StataMirror` object returns a `StataView` object, i.e., the type of object returned by `st_view`. Hence the subscript-derived objects will behave just as if they came from `st_view`, and these are not aware of data set changes (see example §11.11).

```
>>>. m[:, :]
<stata.StataView object at 0x000000000315B128>
```

```
>>>. type(m[:, :]) == type(st_view())
True
```

11.11 `st_matrix`, `st_view`, and `st_mirror` after changes in Stata

The object returned from `st_matrix` is not aware of changes made to Stata matrices, and the object returned by `st_view` and is not aware of changes made to the current data set. The following example demonstrates this for `st_matrix`.

```
. clear
```

```
. sysuse auto
(1978 Automobile Data)
```

```
. mkmat mpg rep head in 1/5, matrix(m)
```

```
. matrix list m
m[5,3]
         mpg     rep78  headroom
r1        22         3       2.5
r2        17         3         3
r3        22         .         3
r4        20         3       4.5
r5        15         4         4
```

```
. python
──────────────────────────────────────────────────────────────────────────────── python (type exit() to exit)
```

```
>>>. m = st_matrix("m")
```

```
>>>. m.list()
m[5,3]
           c0         c1         c2
r0         22          3        2.5
r1         17          3          3
r2         22          .          3
r3         20          3        4.5
r4         15          4          4
```

```
>>>. exit()
──────────────────────────────────────────────────────────────────────────────────────────────────
```

```
. mkmat mpg - weight in 1/5, matrix(m)
```

```
. matrix list m
m[5,5]
         mpg     rep78  headroom     trunk    weight
r1        22         3       2.5        11      2930
r2        17         3         3        11      3350
r3        22         .         3        12      2640
r4        20         3       4.5        16      3250
r5        15         4         4        20      4080
. python
------------------------------------------------------------ python (type exit() to exit)
>>>. m.list()
m[5,3]
           c0         c1         c2
r0         22          3        2.5
r1         17          3          3
r2         22          .          3
r3         20          3        4.5
r4         15          4          4
>>>. exit()
------------------------------------------------------------------------------
. mkmat mpg in 1/5, matrix(m)
. matrix list m
m[5,1]
    mpg
r1   22
r2   17
r3   22
r4   20
r5   15
. python
------------------------------------------------------------ python (type exit() to exit)
>>>. m.list()
Traceback (most recent call last):
  File "<string>", line 1, in <module>
  File "stata.py", line 648, in __repr__
    return header + "\n" + colTop + "\n" + "\n".join(rowGen)
  File "stata.py", line 646, in <genexpr>
    for r in rowNums)
  File "stata.py", line 645, in <genexpr>
    " ".join(st_format(fmt, st_matrix_el(matname, r, c)) for c in colNums)
IndexError: matrix col number out of range
>>>. exit()
```

The error comes from the Python object `m` thinking there are 4 columns and trying the access the values in them.

Here is short example demonstrating the same for `st_view`.

```
. clear
. sysuse auto
(1978 Automobile Data)
. python
------------------------------------------------------------ python (type exit() to exit) ---
>>>. v = st_view()
>>>. v[1, ::3].list()
```

```
  obs: 1
 vars: 4
           c0        c3        c6        c9
r1   AMC Pacer         3      3350       258
>>>. exit()
```

```
. clear
. sysuse lifeexp
(Life expectancy, 1998)
. di c(k)
6
. python
```

---

python (type exit() to exit)

```
>>>. v[1, ::3].list()
Traceback (most recent call last):
  File "<string>", line 1, in <module>
  File "stata.py", line 523, in __getitem__
    return StataView(rownums=sel_rows, colnums=sel_cols)
  File "stata.py", line 350, in __init__
    "%11s" if st_isstrvar(c) else "%9.0g" for c in colnums
  File "stata.py", line 350, in <listcomp>
    "%11s" if st_isstrvar(c) else "%9.0g" for c in colnums
IndexError: Stata variable number out of range
```

Objects returned by `st_mirror` (objects of `StataMirror` class) *are* aware of such changes.

```
. clear
. sysuse auto
(1978 Automobile Data)
. python
```

---

python (type exit() to exit)

```
>>>. mirror = st_mirror()
>>>. mirror[1, ::3].list()
  obs: 1
 vars: 4
           c0        c3        c6        c9
r1   AMC Pacer         3      3350       258
>>>. exit()
```

```
. clear
. sysuse lifeexp
(Life expectancy, 1998)
. python
```

---

python (type exit() to exit)

```
>>>. mirror[1, ::3].list()
  obs: 1
 vars: 2
          c0        c3
r1         1        74
>>>. exit()
```

However, this awareness is not transmitted to objects derived by subscripting. As discussed in §11.10, objects derived from a `StataMirror` instance by subscripting are the same type as returned by `st_view`, and are thus not automatically aware of data set changes.

11.12 `st_matrix`, `st_view`, and `st_mirror` objects are iterable

Objects returned by `st_matrix` are iterable over rows, and objects returned by `st_view` and `st_mirror` are iterable over observations. What this means, in particular, is that they are easy to traverse in `for` loops. The object returned with each iteration is a tuple of the values in the row or observation.

```
. clear
. sysuse auto
(1978 Automobile Data)
. mkmat price-head in 2/8, matrix(m)
. python
------------------------------------------------- python (type exit() to exit)
>>>. m = st_matrix("m")
>>>. import collections
>>>. isinstance(m, collections.Iterable)
True
>>>. for row in m: print(row)
(4749.0, 17.0, 3.0, 3.0)
(3799.0, 22.0, ., 3.0)
(4816.0, 20.0, 3.0, 4.5)
(7827.0, 15.0, 4.0, 4.0)
(5788.0, 18.0, 3.0, 4.0)
(4453.0, 26.0, ., 3.0)
(5189.0, 20.0, 3.0, 2.0)
>>>. for row in m[::2, (0, 0, 1, 1, 2, 2)]: print(row)
(4749.0, 4749.0, 17.0, 17.0, 3.0, 3.0)
(4816.0, 4816.0, 20.0, 20.0, 3.0, 3.0)
(5788.0, 5788.0, 18.0, 18.0, 3.0, 3.0)
(5189.0, 5189.0, 20.0, 20.0, 3.0, 3.0)
>>>. v = st_view()
>>>. isinstance(v, collections.Iterable)
True
>>>. for row in v[::8, :5]: print(row)
(´AMC Concord´, 4099.0, 22.0, 3.0, 2.5)
(´Buick Riviera´, 10372.0, 16.0, 3.0, 3.5)
(´Chev. Monte Carlo´, 5104.0, 22.0, 2.0, 2.0)
(´Ford Mustang´, 4187.0, 21.0, 3.0, 2.0)
(´Merc. XR-7´, 6303.0, 14.0, 4.0, 3.0)
(´Olds Toronado´, 10371.0, 16.0, 3.0, 3.5)
(´Pont. Grand Prix´, 5222.0, 19.0, 3.0, 2.0)
(´Datsun 210´, 4589.0, 35.0, 5.0, 2.0)
(´Renault Le Car´, 3895.0, 26.0, 3.0, 3.0)
(´VW Scirocco´, 6850.0, 25.0, 4.0, 2.0)
>>>. for row in v[::8, :3]: print("{0:>18} {1:>8} {2:>5}".format(*row))
       AMC Concord   4099.0  22.0
     Buick Riviera  10372.0  16.0
```

```
 Chev. Monte Carlo   5104.0  22.0
      Ford Mustang   4187.0  21.0
        Merc. XR-7   6303.0  14.0
     Olds Toronado  10371.0  16.0
  Pont. Grand Prix   5222.0  19.0
        Datsun 210   4589.0  35.0
    Renault Le Car   3895.0  26.0
       VW Scirocco   6850.0  25.0
>>>. exit()
```

11.13 Using Python files

To run a Python script, just use the `python` command with the `file` option (or call the plugin directly, see §6). With the `python` command, the Python file can be anywhere in your Ado path (including the working directory). If you want to use files with the plugin directly, you will have to specify the path if the file is not in the working directory, or use the Mata function `findfile()` to find the path.

For example, suppose you have a Python file called `pyfile.py` and its contents are

```
nvars = int(st_local("_pynvars"))
vars = [st_local("_pyvar" + str(i)) for i in range(nvars)]
print("\nselected vars are\n\t{}\n".format(vars))

nargs = int(st_local("_pynargs"))
args = [st_local("_pyarg" + str(i)) for i in range(nargs)]
print("arguments are\n\t{}\n".format(args))

firstRow = [_st_sdata(0, i)
            if st_isstrvar(i) else _st_data(0, i)
            for i in range(st_nvar())]
print("first row of data is\n\t{}\n\n".format(firstRow))
```

If the file is in the Ado path, you can run the file interactively using `python`:

```
. clear

. sysuse auto
(1978 Automobile Data)

. python mpg trunk weight gear, file(pyfile.py) args(plus "o t h e r" options)

selected vars are
        [´mpg´, ´trunk´, ´weight´, ´gear_ratio´]

arguments are
        [´plus´, ´o t h e r´, ´options´]

first row of data is
        [´AMC Concord´, 4099.0, 22.0, 3.0, 2.5, 11.0, 2930.0, 186.0, 40.0, 121.
> 0, 3.5799999237060547, 0.0]
```

To use Python files inside your own Ado command, just use the `python` command as above or call the plugin directly. If calling the plugin directly, consider the recommendations in §6. You will also probably want to use the Mata function `findfile()` to find the Python file.

Here is an example for a command called `prem`, for *p*rint *r*egular *e*xpression *m*atch. The command uses Python's regular expression module `re` to find pattern matches in a Stata string variable. The command uses an Ado file called `prem.ado`, which consists of

```
program prem
  version 12.1
  syntax varlist(string min=1 max=1) [if] [in] , regex(string)

  // Put all varnames of varlist in locals.
  // Used to create lookups name <-> index.
  ereturn clear // to clear hidden variables
  local _pynallvars = 0
  if (c(k) > 0) {
    foreach var of varlist * {
      local _pyallvars`_pynallvars' = "`var'"
      local _pynallvars = `_pynallvars' + 1
    }
  }

  // find prem.py in Ado path
  mata: st_local("filepath", findfile("prem.py"))
  if ("`filepath'" == "") {
    noi di as error "cannot find Python file prem.py"
    exit 601
  }

  plugin call python_plugin * `if' `in' , "`filepath'"
end

program python_plugin, plugin
```

The majority of the code in `prem.ado` can be considered a template for Stata commands that call a Python file. It implements the suggestions in §6 and uses Mata to find the desired Python file. The Python file `prem.py` consists of

```
import re

varNum = st_varindex(st_local("varlist"), True)
reComp = re.compile(st_local("regex"))

for i in range(st_in1(), st_in2()):
    if st_ifobs(i):
        obs = _st_sdata(i, varNum)
```

```
    m = reComp.search(obs)
    if m:
        beg, end = m.start(), m.end()
        s1, s2, s3 = obs[:beg], obs[beg:end], obs[end:]
        print(s1 + "{ul on}" + s2 + "{ul off}" + s3)
```

Both `prem.ado` and `prem.py` should be in Stata's `adopath`. The command can be used like so:

```
. clear
. sysuse auto
(1978 Automobile Data)
. prem make , regex("((.)\2+)")
Cad. Deville
Cad. Seville
Chev. Chevette
Linc. Versailles
Olds Cutlass
Olds Delta 88
Plym. Arrow
Plym. Sapporo
Audi 5000
Datsun 200
Honda Accord
Toyota Corolla
VW Rabbit
VW Scirocco
. prem make if foreign , regex("((.)\2+)")
Audi 5000
Datsun 200
Honda Accord
Toyota Corolla
VW Rabbit
VW Scirocco
```

11.14 The result of `exit()` is hard-coded

In Python, the label `exit` can be reassigned. If that is done in a typical interactive session, `exit()` will no longer exit the interpreter (depending, of course, on how `exit` was reassigned). In this interactive interpreter, the behavior of `exit()` is hard-coded to cause an exit.

```
. python
---------------------------------------------------- python (type exit() to exit)
>>>. exit = "blah"
>>>. exit()
--------------------------------------------------------------------------------

. python
---------------------------------------------------- python (type exit() to exit)
>>>. def exit(): return "will it exit?"
>>>. exit()
--------------------------------------------------------------------------------
```