Title

stata.com

python — Call Python from Stata

Description

`python` provides utilities for embedding Python code within Stata. With these utilities, users can invoke Python interactively or in do-files and ado-files.

`python` [`:`] creates a Python environment in which Python code can be executed interactively, just like a Python interpreter. In this environment, the classic "`>>>`" and "`...`" prompts are used to indicate the user input. All the objects inside this environment are created in the namespace of the `__main__` module.

`python:` *istmt* executes one Python simple statement or several simple statements separated by semicolons.

`python script` executes a Python script `.py` file. A list of arguments can be passed to the file by using option `args()`.

`python set exec` *pyexecutable* sets which Python version to use. *pyexecutable* specifies the full path of the Python executable. If the executable does not exist or does not meet the minimum version requirement, an error message will be issued.

`python set userpath` *path* [*path* ...] sets the user's own module search paths in addition to system search paths. Multiple paths may be specified. When specified, those paths will be loaded automatically when the Python environment is initialized.

`python describe` lists the objects in the namespace of the `__main__` module.

`python drop` removes the specified objects from the namespace of the `__main__` module.

`python clear` clears all the objects whose names are not prefixed with `_` from the namespace of the `__main__` module.

`python query` lists the current Python settings and system information.

`python search` finds the Python versions installed on the current operating system. Only Python 2.7 and greater will be listed. On Windows, the registry will be searched for official Python installation and versions installed through `Anaconda`. On Unix or Mac, the registry will be searched for Python installations in the `/usr/bin/`, `/usr/local/bin/`, `/opt/local/python/bin/`, `~/anaconda/bin`, or `~/anaconda3/bin` directories.

`python which` checks the availability of a Python module.

Syntax

Enter Python interactive environment

```
python[ : ]
```

Execute Python simple statements

```
python: istmt
```

Execute a Python script file

```
python script pyfilename [, args(args_list) global
    userpaths(user_paths[ , prepend])]
```

Set which version of Python to use

```
python set exec pyexecutable [ , permanently]
```

`set python_exec` is a synonym for `python set exec`.

Set user's additional module search paths

```
python set userpath path [path ...] [ , permanently prepend]
```

`set python_userpath` is a synonym for `python set userpath`.

List objects in the namespace of the `__main__` *module*

```
python describe [namelist] [ , all]
```

Drop objects from the namespace of the `__main__` *module*

```
python drop namelist
```

Clear objects from the namespace of the `__main__` *module*

```
python clear
```

Query current Python settings and system information

```
python query
```

Search for Python installations on the current system

```
python search
```

Check the availability of a Python module

```
python which modulename
```

istmt is either one Python simple statement or several simple statements separated by semicolons.

pyfilename specifies the name of a Python script file with extension `.py`.

pyexecutable specifies the executable of a Python installation, such as
`"C:\Program Files\Python36\python.exe"`,
`"/usr/bin/python"`,
`"/usr/local/bin/python"`,
`"~/anaconda3/bin/python"`, or
`"~/anaconda/bin/python"`.

namelist specifies a list of object names, such as `sys`, `spam`, or `foo`. Names can also be specified using the `*` and `?` wildcard characters:

`*` indicates zero or more characters.

`?` indicates exactly one character.

modulename specifies the name of a Python module. The module can be a system module or a user-written module. The name can be a regular single module name or a dotted module name, such as `sys`, `numpy`, or `numpy.random`.

Options

`args(`*args_list*`)` specifies a list of arguments, *args_list*, that will be passed to the Python script file and can be accessed through `argv` in Python's `sys` module. *args_list* may contain one argument or a list of arguments separated by spaces.

`global` specifies that the objects created in the Python script file be appended to the namespace of the `__main__` module so that they can be accessed globally. By default, the objects created in the script file are discarded after execution.

`userpaths(`*user_paths*`[ , prepend ])` specifies the additional module search paths that will be added to the system paths stored in `sys.path`. *user_paths* may be one or a list of paths separated either by spaces or by semicolons. By default, those paths will be added to the end of system paths. If `prepend` is specified, they will be added in front of the system paths.

`permanently` specifies that, in addition to making the change right now, the setting be remembered and become the default setting when you invoke Python.

`prepend` specifies that instead of adding the user's additional module search paths to the end of system paths, the paths are to be added in front of the system paths.

`all` specifies that all the objects in the namespace of the `__main__` module be listed. By default, only objects that do not begin with an underscore will be listed.

Remarks and examples

stata.com

Remarks are presented under the following headings:

Invoking Python interactively

You type `python` or `python:` (with the colon) to enter the interactive environment.

```
. python
------------------------------------------------- python (type end to exit) -----
>>>
```

Within the interactive environment, we use three greater-than signs (`>>>`) as the primary prompt and three dots (`...`) as the secondary prompt for continuation lines. When you type a statement in the environment, the Python interpreter will compile what you typed, and if it is compiled without error, the statement will be executed. Note that within the Python environment, all the statements need to follow Python's style, such as for indentation and line breaks. For example,

```
>>> word = 'Python'
>>> word[0], word[-1]
('P', 'n')
>>> len(word)
6
>>> squares = [1,4,9,16,25]
>>> squares
[1, 4, 9, 16, 25]
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1,8)]
['3.1', '3.14', '3.142', '3.1416', '3.14159', '3.141593', '3.1415927']
>>>
>>> for i in range(3):
...     print(i)
...
0
1
2
```

When you are done using Python, type `end` following the `>>>` prompt:

```
>>> end
------------------------------------------------------------------------------
```

When you exit from the Python interactive environment back into Stata, the environment does not clear itself; so if you later type `python` or `python:` again, you will be right back where you were.

All the objects created in the interactive environment are stored in the namespace of the `__main__` module, and they can be accessed later when you exit Python and come back. In Stata, you can use `python describe`, `python drop`, and `python clear` to manipulate those objects.

Within the interactive environment, only Python statements are accepted. To execute a Stata command while in the Python environment, prefix the Stata command with `stata:`. For example, suppose `auto.dta` is in memory and we want to run a regression of `mpg` on `weight` and `foreign` using the `regress` command. We can type

```
>>> stata: regress mpg weight foreign
```

and the output would match what is produced in Stata. This syntax only works in the Python interactive environment. It will not work in a Python script, nor embedded within compound statements, such as `def` or `if`, in an interactive environment. Instead, use the `stata()` function, one of the functions defined in the Python class `SFIToolkit` within the `sfi` (Stata Function Interface) module, to execute Stata commands within script files and compound statements.

In the interactive environment, when a statement fails to compile, a stack trace will be printed and an error code will be issued. For example,

```
>>> spam
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
r(7102);
>>>
```

The stack trace issued by the Python interpreter states that a `NameError` occurs because the variable `spam` has not been defined. The error code `r(7102)` tells Stata that something is wrong with the Python environment. See *Error codes* for a detailed description.

The distinction between python and python:

Issuing `python` (without a colon) will allow you to remain in the Python environment despite errors. Issuing `python:` will allow you to work in the Python environment but will return control to Stata when you encounter a Python error. For example, consider the following (using `python` without the colon):

```
python
a = a + 2
b = 6
end
```

In the above code, the variable `a` is not defined, so the statement `a = a + 2` will throw a Python error. Because we used `python` without the colon, the incorrect line would be issued, and we would remain in the Python environment until the `end` statement. Python would not tell Stata that anything went wrong! This could have serious consequences. On the other hand, if we had used `python:` (with a colon), the same error would return control to Stata and issue an error code; the second statement (`b = 6`) would not be executed at all.

Embedding Python code in a do-file

Typing statements interactively can be prone to error, especially when you type a compound statement using indentation. Instead, you can write Python code within a do-file and run multiple statements consecutively. All you need to do is place the Python code within a `python[:]` and `end` block. By placing your code in a do-file, you can mix Stata code and Python code in a single file, execute it all at once, and even run it multiple times. For example,

begin pyex1.do

```
version 16.1
local a = 2
local b = 3

python:
from sfi import Scalar
def calcsum(num1, num2):
    res = num1 + num2
    Scalar.setValue("result", res)

calcsum('a', 'b')
end

display result
```

end pyex1.do

In the above do-file, we defined two local macros in Stata, `a` and `b`, which we use as arguments later. Within the `python:` and `end` block, we first defined a function, `calcsum()`, that calculated the sum of two numbers. We passed the result back to Stata as a scalar named `result` by using the `setValue()` function of the `Scalar` class defined in the `sfi` module. Finally, the function was called.

Typing `do pyex1` returns a result of `5`.

As you can see, we called the function with `calcsum(‘a’, ‘b’)` After macro expansion, this line became `calcsum(2, 3)` and the values `2` and `3` were passed to the function. Macro substitution is a convenient way to pass values from Stata to Python. You can use macros when typing Python statements interactively in the Command window or when writing Python statements in a do-file. You just need to follow Stata's quotes notation.

When you run the do-file and the `python:` line is executed, it will enter the interactive environment and run Python code line by line. After the `end` line is executed, it will exit Python and enter Stata again.

Because the Python code is executed in the interactive environment, all objects defined in the Python block within a do-file are automatically added to the namespace of the `__main__` module. Thus, they can be accessed later when you enter Python statements interactively or in another Python block within a do-file. For example, we can rewrite the above do-file as follows, and it will lead to the same result:

begin pyex2.do

```
version 16.1
local a = 2
local b = 3

python:
from sfi import Scalar
def calcsum(num1, num2):
     res = num1 + num2
     Scalar.setValue("result", res)

end

python: calcsum(‘a’, ‘b’)
display result
```

end pyex2.do

Here we called the function `calcsum()` by using the simple statement syntax outside the first Python block, and the argument values were passed in through macro substitution. We will discuss macro substitution and the simple statement syntax further in *Embedding Python code in an ado-file*.

Running a Python script file

Be aware that Stata and Python use different syntax, data structures and types, language infrastructures, etc. They even have different rules for handling comments and indentations.

Because of these differences, it may be best to isolate Stata and Python code. This can be achieved by writing Python code in a `.py` script file, and then running `python script` in Stata to execute it. For example, let's isolate the Stata and Python code from the example above.

We first write the Python code in a script file, say, pyex.py:

begin pyex.py

```
from sfi import Macro, Scalar
def calcsum(num1, num2):
     res = num1 + num2
     Scalar.setValue("result", res)

pya = int(Macro.getLocal("a"))
pyb = int(Macro.getLocal("b"))
calcsum(pya, pyb)
```

end pyex.py

In this script file, we first defined the function calcsum() as we did before. We called the function getLocal(), defined in the Macro class within the sfi module, to get the local macro values a and b from Stata. Because getLocal() returns a string value, we called Python's built-in function int() to get the numeric values, and we passed them to calcsum().

Next we call this script file in a separate do-file, say, pyex3.do:

begin pyex3.do

```
version 16.1
local a = 2
local b = 3

python script pyex.py
display result
```

end pyex3.do

In the do-file, we first defined two local macros and passed them to the calcsum() function. Next we ran the script file with the python script command and obtained the scalar result.

By default, all the objects defined in the script file are discarded after execution; they are not added to the namespace of the __main__ module. In other words, the execution of a script file does not share the same namespace with the __main__ module, which means you cannot access objects defined in the __main__ module from the script file and vice versa.

To use objects in the namespace of the __main__ module in a script file, you can import them with the import or import-from statement. For example, you can include

```
import __main__
```

in a script file to access each object defined in the __main__ module.

On the other hand, if you want the interactive environment to have access to the objects defined in the script file after it has been executed, you can specify the global option in the python script command. By specifying this option, all the objects are copied to the namespace of the __main__ module, so they can be used directly without having to import them. This is useful when you define functions, classes, etc., in a script file and want to access them interactively or in a do-file. However, you should use this option with caution because those objects will overwrite objects defined in the namespace of the __main__ module with the same name.

You can pass arguments from Stata to a script file with the args() option of python script. To access those arguments in the script file, use the argv list defined in Python's sys module. Let's use the above example to illustrate.

We rewrote the script file and the do-file as follows:

begin pyex2.py

```
import sys
pya = int(sys.argv[1])
pyb = int(sys.argv[2])
from sfi import Macro, Scalar
def calcsum(num1, num2):
     res = num1 + num2
     Scalar.setValue("result", res)
calcsum(pya, pyb)
```

end pyex2.py

begin pyex4.do

```
version 16.1
local a = 2
local b = 3
python script pyex2.py, args(‘a’ ‘b’)
display result
```

end pyex4.do

In the script file, we imported the `sys` module and then got the arguments through the `sys.argv` list. Because we will pass two arguments to the script file, we access the argument values with `sys.argv[1]` and `sys.argv[2]`. Note that when executing a script file, `sys.argv[0]` stores the script name, which is `pyex2.py` in this case. In the do-file, we passed the macro values to the Python script file by listing them in the `args()` option of `python script`.

Another option you may find useful when running a script file in Stata is `userpaths()`, which allows you to find and import modules defined in your private paths. By default, the paths you specified are appended to the end of the list. You can prepend them to the beginning of the list by using the `prepend` suboption.

These paths are only added temporarily to `sys.path`, which means they will be used only when executing the script file. After that, they will be discarded from the list. To add a path permanently, use `python set userpath`. See *Locating modules* for a detailed discussion about setting module search paths.

Embedding Python code in an ado-file

Python code can be embedded and executed in ado-files too. This is useful when you are interested in extending Stata by adding a new command. Below, we use an example to illustrate this purpose.

Suppose that we want to write a new command for Stata that will report the sum of one variable. We might do this as follows:

begin varsum.ado

```
program varsum
    version 16.1
    syntax varname [if] [in]
    marksample touse
    python: calcsum("`varlist'", "`touse'")
    display as txt "  sum of `varlist': " as res r(sum)
end

version 16.1
python:
from sfi import Data, Scalar

def calcsum(varname, touse):
    x = Data.get(varname, None, touse)
    Scalar.setValue("r(sum)", sum(x))

end
```

end varsum.ado

We load `auto.dta` and run this program from Stata. It will result in the following output:

```
. varsum price
  sum of price: 456229
```

Let's explain what happened in the ado-file step by step:

1. The ado-file has both ado-code and Python code in it.
2. The ado-code handled all parsing and identified the subsample of the data to be used.
3. The ado-code called the Python function `calcsum()` to perform the calculation using the simple statement syntax `python:` *istmt*.
4. The Python code first imported two classes, `Data` and `Scalar`, from the `sfi` module. Then it defined the function `calcsum()`, which received as arguments the names of two variables in the Stata dataset: the variable on which the calculation was to be made and the variable that identified the subsample of the data to be used.
5. The Python function returned the result in `r()`, where the ado-code can access it.

In the ado-file, the Python code was defined within the `python:` and `end` block. You can treat this block as a Python script file, meaning that you can write any Python statement within it. Here we define only one function, `calcsum()`, which acts as a connection between the ado-code and the Python code.

In a connection like this, you have two paramount interests: getting values defined in the ado-code into the Python function, and getting results returned by the function back to your ado-code. For `calcsum()`, the values defined in the ado-code are passed to the function as arguments. When we called the function

```
python: calcsum("`varlist'", "`touse'")
```

this line was automatically expanded and turned into something like

```
python: calcsum("price", "__0001dc")
```

The `__0001dc` variable is a temporary variable created by the `marksample` command earlier in our ado-file. `price` was the variable specified by the user. After expansion, the arguments were nothing more than strings, and those strings were passed to `calcsum()`.

Macro substitution is the most common way values are passed from Stata to Python functions. When writing your Python function, keep in mind the arguments that Stata will find convenient to pass and that Python will make convenient to use:

1. numbers, such as 2 and 3 (`‘a’` and `‘b’` in pyex1.do)
2. names of variables, macros, scalars, matrices, etc., such as `"price"` and `"__0001dc"` (`"‘varlist’"` and `"‘touse’"`)

To receive arguments of type 1, you code numeric type in the function declaration for the argument, and then pass in the value using Stata's quotes notation. To receive arguments of type 2, you code string type in the function declaration for the argument, and then use classes and functions defined in the sfi module to extract the contents from the name.

On the other hand, you may use other functions defined in those classes to return results to Stata too. For example, you can return results in `r()`—as we did in our example—or in `e()` or `s()`. You can also create Stata macros, scalars, matrices, and even Mata objects from within Python.

Here are some general guidelines for passing values between Python and Stata:

1. If you are dealing with a variable name, you will want to read about the functions defined in the Data and Frame classes.
2. If you are dealing with local or global macros, scalars, or matrices, you will want to see the Macro, Scalar, and Matrix classes.
3. Refer to the sfi module for more detailed descriptions and additional functions.

Remember that all Python objects defined in an ado-file are private and cannot be accessed outside of it. So you cannot use those objects in the interactive environment, in a do-file, or in another ado-file. However, you can still access objects defined in the namespace of the `__main__` module by using the `import` or `import-from` statement within the `python[:]` and `end` block of an ado-file.

In the above example, we put the `calcsum()` function in the ado-file within the `python:` and `end` block. You can also write the function in a Python module file and import the function from the module to the ado-file. Let's restructure our ado-file to save `calcsum()` in a `.py` file.

First, we simply write the function in a `.py` file named `pyex3.py`, as follows:

begin pyex3.py

```
from sfi import Data, Scalar
def calcsum(varname, touse):
    x = Data.get(varname, None, touse)
    Scalar.setValue("r(sum)", sum(x))
```

end pyex3.py

Next, we import the function from the module to our ado-file, so it now reads

begin varsum.ado

```
program varsum
     version 16.1
     syntax varname [if] [in]
     marksample touse
     python: calcsum("`varlist'", "`touse'")
     display as txt "  sum of `varlist': " as res r(sum)
end

version 16.1
python:
from pyex3 import calcsum
end
```

end varsum.ado

Note the following:

1. All the original Python code within the `python:` and `end` block was moved to the Python module file `pyex3.py`.
2. The `python:` and `end` block now has only one statement, which imports the function `calcsum()` from the module by using the `import-from` syntax. Alternatively, you can import the function from the module by using the `import` syntax, depending on your preference. For example, you can import the whole module by using `import pyex3` and then call `python: pyex3.calcsum("`varlist'", "`touse'")` in the ado-code.
3. To make `import` in note 2 work, the module file must be placed where Stata can find it. See *Locating modules* for details on how Stata searches for modules.

Each of the two alternatives to write your Python function has its own advantages. You can choose which one to use based on your preference.

1. Putting the Python code right in the ado-file is easier, and it sure is convenient. You only need to handle a single file.
2. Saving the Python code in a module file makes the Python utilities (`calcsum()` here) available for use in your other ado-files. Compared with the Python code being restricted to the ado-file in note 1, this is more useful if you call the same Python utility in various ado-files.
3. You can combine the two alternatives under some circumstances. For example, suppose you have a few utility functions defined in an existing module file—say, `pyutil.py`—and you want to call those utilities in the `calcsum()` function. You do not need to copy those utilities to the ado-file to use them in `calcsum()`. Instead, you can just import them from the existing module and use them directly in the ado-file.

Stata Function Interface (sfi) module

The Stata Function Interface (`sfi`) module allows users to interact Python's capabilities with core features of Stata. The module can be used interactively or in do-files and ado-files.

Within the module, classes are defined to provide access to Stata's characteristics, current dataset, data and time, macros, scalars, matrices, value labels, global Mata matrices, and so on. The following is a summary of them:

Class	Description
`Characteristic`	This class provides access to Stata characteristics.
`Data`	This class provides access to the Stata dataset in memory.
`Datetime`	This class provides access to Stata date and time values.
`Frame`	This class provides access to a Stata data frame.
`FrameError`	This class indicates that an exceptional condition has occurred within a frame.
`Macro`	This class provides access to Stata macros.
`Mata`	This class provides access to global Mata matrices.
`Matrix`	This class provides access to Stata matrices.
`Missing`	This class provides tools for handling Stata missing values.
`Platform`	A set of utilities for getting platform information.
`Preference`	A set of utilities for loading and saving preferences.
`Scalar`	This class provides access to Stata scalars.
`SFIError`	This class is the base class for other exceptions defined in this module.
`SFIToolkit`	This class provides a set of core tools for interacting with Stata.
`StrLConnector`	This class facilitates access to Stata's `strL` data type.
`ValueLabel`	This class provides access to Stata's value labels.

Within Python, you can use

```
import sfi
```

or

```
from sfi import *
```

to import the whole module. Alternatively, you can import specific classes. For example, to import `Data` and `Macro`, you can use

```
from sfi import Data, Macro
```

After the classes are imported, you can invoke the various functions defined within them. See Stata's Python API for detailed documentation about each class and function.

Configuring Python

Currently, Python has two major versions: Python `2` and Python `3`. Stata supports both of them starting from Python `2.7`. The first time you call `python` in Stata, Stata will search for Python installations on the system and choose the one with the highest version. Stata will search the official Python installations and Python installations bundled with Anaconda or Miniconda. The installation must contain the corresponding Python dynamically linked library. For example, for Python 3.6, it would be something like `python36.dll` on Windows, `libpython3.6.so` on Linux, and `libpython3.6.dylib` on Mac. Otherwise, it will not be found and used as a candidate. Once Stata finds the candidate with the highest version, it will save that information to use in the future. You can see which Python version Stata will use by typing `python query`.

You can type `python search` to conduct a search. It will list all the Python executables on the system. On Windows, it looks for `python.exe`. On Linux or Mac, it looks for `/usr/bin/python`, `/usr/bin/python3`, `/usr/local/bin/python`, `/usr/local/bin/python3`, `~/anaconda/bin/python`, `~/anaconda3/bin/python`, etc.

If you want to use a Python version different from the default, you can type `python set exec` to change the setting. For example, on Linux, you can type

```
python set exec "/usr/local/bin/python"
```

If `python search` does not find the Python environment you wanted (for example, a user-created virtual environment), you can type `python set exec` to use the version of choice.

Setting the Python version is optional, but if set, it must be done before the initialization of Python. Otherwise, an error will be issued. The setting will be available only for the current Stata session. If you want Stata to remember the setting and use that Python version by default the next time you launch Stata, then use the `permanently` option:

```
python set exec "/usr/local/bin/python", permanently
```

Locating modules

According to the Python documentation (sec. 6.1.2 and sec. 6.1.3), "When a module named `spam` is imported, the interpreter first searches for a built-in module with that name. If not found, it then searches for a file named `spam.py` in a list of directories given by the variable `sys.path`." When the interpreter is initialized in Stata, Stata's system directories (`sysdir`) and a `py/` directory within each system directory, except the `STATA` directory, are added to the list following the default module search paths. For example, on a particular Windows computer, the following paths are added:

```
C:\Program Files\Stata16\
C:\Program Files\Stata16\ado\base\
C:\Program Files\Stata16\ado\base\py\
C:\Program Files\Stata16\ado\site\
C:\Program Files\Stata16\ado\site\py\
C:\ado\plus\
C:\ado\plus\py\
C:\ado\personal\
C:\ado\personal\py\
C:\ado\
C:\ado\py\
```

If you want to add other paths to the module search path list, you can type `python set userpath` to add a list of paths at once. For example,

```
python set userpath "C:\mymodules1\" "C:\mymodules2\"
```

By default, those paths are added to the end of the list so that modules in those directories are searched last. If you want those paths to be searched first, you can specify the `prepend` option, which will add those paths to the beginning of the module search path. Paths added in this way will be kept in the module search path list and be searched for the whole Stata session. This is different from specifying the `userpaths()` option with `python script`, which removes the paths from the module search path list once the script is executed.

Specifying additional module search paths is optional, but if specified, it must be done before the initialization of Python. Otherwise, an error will be issued. The setting will be available only for the current Stata session. If you want Stata to remember the setting and use the additional paths by default the next time you launch Stata, then use the `permanently` option.

When you want to import third-party Python modules (such as `numpy`, `pandas`, etc.) in your Python code, you need to make sure that they are already installed in the Python version that you are currently using. Otherwise, an error will be issued claiming the specified module is not found. You can type `python which` to check whether a module is available in your current Python settings.

Error codes

When you run Python code within Stata, a Stata error code will be issued with the Python stack trace if an error occurs. Here is a list of them:

Code	Meaning
7100	error occurs when loading or freeing the Python dynamically linked library
7101	attempt to set a different Python version or add additional module search paths after the Python environment is initialized
7102	error occurs when executing Python in the interactive environment
7103	error occurs when running a Python script file or importing a Python module

To create custom errors in your Python code, invoke the `exit()` function defined in the Python class `SFIToolkit` within the `sfi` module. This is often used when you want to terminate execution of Python code when handling an error condition or exceptions. Let's use the following two script files as an illustration.

begin pyex4.py

```
from sfi import SFIToolkit
a = 3
if a > 4:
    SFIToolkit.displayln("continue execution")
else:
    SFIToolkit.errprintln("assertion failed")
    SFIToolkit.exit(198)
# This line will not be executed due to assertion failure.
SFIToolkit.displayln("never reached")
```

end pyex4.py

and

begin pyex5.py

```
from sfi import SFIToolkit
try:
    print(a)
except:
     SFIToolkit.errprintln("name a is not defined")
     SFIToolkit.exit(198)
# This line will not be executed due to assertion failure.
SFIToolkit.displayln("never reached")
```

end pyex5.py

Here `errprintln()` is used to output a string to the Stata Results window as an error. `displayln()` is used to output a string as normal text. They both honor any SMCL tags contained in the string. Executing the above script files results in the following output:

```
. python script pyex4.py
assertion failed
r(198);

. python script pyex5.py
name a is not defined
r(198);
```

Stored results

`python query` stores the following in `r()`:

Scalars

`r(initialized)`	whether Python environment initialized (`0` or `1`)

Macros

`r(execpath)`	Python executable path
`r(userpath)`	Python user path
`r(version)`	Python version
`r(arch)`	Python architecture (`64-bit` or `32-bit`)
`r(libpath)`	Python shared library

Acknowledgment

The thought of embedding Python code within Stata was inspired by the Python plugin for Stata, which was written by James Fiedler, Universities Space Research Association.

References

Huber, C. 2020a. Stata/Python integration, part 1: Setting up Stata to use Python. *The Stata Blog: Not Elsewhere Classified.* https://blog.stata.com/2020/08/18/stata-python-integration-part-1-setting-up-stata-to-use-python/.

——. 2020b. Stata/Python integration, part 2: Three ways to use Python in Stata. *The Stata Blog: Not Elsewhere Classified.* https://blog.stata.com/2020/08/25/stata-python-integration-part-2-three-ways-to-use-python-in-stata/.

——. 2020c. Stata/Python integration, part 3: How to install Python packages. *The Stata Blog: Not Elsewhere Classified.* https://blog.stata.com/2020/09/01/stata-python-integration-part-3-how-to-install-python-packages/.

——. 2020d. Stata/Python integration, part 4: How to use Python packages. *The Stata Blog: Not Elsewhere Classified.* https://blog.stata.com/2020/09/10/stata-python-integration-part-4-how-to-use-python-packages/.

——. 2020e. Stata/Python integration, part 5: Three-dimensional surface plots of marginal predictions. *The Stata Blog: Not Elsewhere Classified.* https://blog.stata.com/2020/09/14/stata-python-integration-part-5-three-dimensional-surface-plots-of-marginal-predictions/.

——. 2020f. Stata/Python integration, part 6: Working with APIs and JSON. *The Stata Blog: Not Elsewhere Classified.* https://blog.stata.com/2020/09/29/stata-python-integration-part-6-working-with-apis-and-json-data/.

——. 2020g. Stata/Python integration part 7: Machine learning with support vector machines. *The Stata Blog: Not Elsewhere Classified.* https://blog.stata.com/2020/10/13/stata-python-integration-part-7-machine-learning-with-support-vector-machines/.

Also see

[P] **Java intro** — Introduction to Java plugins

[P] **javacall** — Call a Java plugin