Stata Dashboards using Python

A GUI pipeline for running "`command.ado`"

The dashboard keeps Stata as the statistical engine and uses Python as the interface layer.

Giovanni Cerulli
CNR-IRCrES

Italian Stata Conference 2026
Milan, 17th of September

The core idea

Python layer

- opens the GUI
- reads the Stata dataset
- lists numeric variables
- writes `run.do`
- shows the Stata log

Stata layer

- defines `command.ado`
- validates the varlist
- computes summary statistics
- writes output to `run.log`

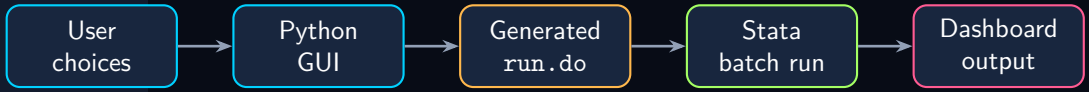
The dashboard wraps a Stata command in a user-friendly Python interface.

Project files

File	Role in the pipeline
`app.py`	Python GUI, command builder, batch runner, log reader
`command.ado`	Stata command executed by the dashboard
`mydata.dta`	sample Stata dataset with 74 observations and 12 variables
`requirements.txt`	Python dependencies: Pandas and PyReadStat
`run.do`	generated at runtime by Python
`run.log`	generated at runtime by Stata

`app.py` `mymean.ado` `mydata.dta` `run.do / run.log`

Dashboard pipeline



Pipeline logic: the GUI collects choices, Python converts them into Stata code, Stata runs the code, and Python brings the log back to the screen.

Step 1: write the Stata command “mymean”

```
program define mymean, rclass
    version 18.0

    syntax varlist(min=1 numeric) [if] [in] ///
        [, DETAIL FORMAT(string)]

    marksample touse
    ...
```

Stata responsibilities

- require at least one numeric variable
- accept optional `if` and `in`
- read `detail` and `format()`
- mark the valid sample

Step 2: compute inside Stata

```
foreach v of local varlist {
    quietly summarize `v' if `touse'

    display as text _newline "Variable: " ///
        as result "`v'"
    display as text "Mean = " ///
        as result `fmt' r(mean)

    if "`detail'" != "" {
        display as text "Std. dev. = " as result `fmt' r(sd)
        display as text "Minimum   = " as result `fmt' r(min)
        display as text "Maximum   = " as result `fmt' r(max)
        display as text "N         = " as result %9.0f r(N)
    }
}
```

Output contract

Stata prints plain text. Python does not need to parse a table or modify the results. It only displays the log.

“Tkinter” as the GUI layer

What Tkinter is

Tkinter is Python’s standard library for desktop interfaces. It creates windows, buttons, text fields, lists, checkboxes, and file dialogs without requiring a web server.

```
import tkinter as tk
from tkinter import filedialog, messagebox, ttk
```

How it helps a Stata dashboard

- `filedialog` selects the Stata executable and `.dta` file
- `Listbox` shows numeric variables
- `Checkbutton` activates `detail`
- `Entry` captures `format()`
- `Text` displays the generated command and Stata log

Step 3: build the Python window

```
class StataDashboard(tk.Tk):
    def __init__(self) -> None:
        super().__init__()
        self.title("MyMean - Stata Dashboard")
        self.geometry("820x680")

        self.dataset_path = None
        self.numeric_variables = []
        self.detail = tk.BooleanVar(value=False)
        self.number_format = tk.StringVar(value="%9.3f")
```

The GUI stores the current dataset, the selected variables, the `detail` option, the numeric display format, and the execution status.

Step 4: load a Stata dataset in Python

```
dataframe = pd.read_stata(
    path,
    convert_categoricals=False
)

self.numeric_variables = (
    dataframe
    .select_dtypes(include="number")
    .columns
    .tolist()
)
```

Dataset scan

- `mydata.dta`: 74 rows
- 12 variables
- 11 numeric variables
- numeric names populate the GUI list

Step 5: translate GUI choices into Stata syntax

```python
def build_command(self) -> str:
    variables = self.selected_variables()
    command = f"mymean {' '.join(variables)}"
    options = []

    if self.detail.get():
        options.append("detail")

    fmt = self.number_format.get().strip()
    if fmt:
        options.append(f"format({fmt})")

    if options:
        command += ", " + " ".join(options)
    return command
```

```
mymean price mpg weight, detail format(%9.3f)
```

Step 6: generate run.do

```
clear all
set more off

adopath ++ "/project/folder"
use "/project/folder/mydata.dta", clear

capture log close
log using "run.log", text replace

mymean price mpg, detail format(%9.3f)

log close
exit, clear
```

Why this matters

The generated do-file lets the audience inspect the exact Stata code behind the GUI action.

Step 7: run Stata from Python

```
result = subprocess.run(
    [str(executable), "-b", "do", do_path.name],
    cwd=APP_DIR,
    capture_output=True,
    text=True,
    check=False,
)
```

-b do
Stata runs the generated do-file in batch mode.

cwd=APP_DIR
The working folder controls where logs and generated files go.

threading
The GUI stays responsive during the Stata run.

Step 8: return the Stata log to the GUI

```
log_path = APP_DIR / "run.log"
if log_path.exists():
    output = log_path.read_text(
        encoding="utf-8",
        errors="replace"
    )

self.after(0, self._show_run_result, output, status)
```

The dashboard does not need a special Stata API for this example. The bridge uses files: Python writes `run.do`, Stata writes `run.log`, Python reads the log.

The whole construction recipe

The same pattern can wrap more complex Stata programs: estimators, post-estimation commands, simulations, tables, or graph production.

Demo storyline

What the audience sees

1. open the Python app
2. load `mydata.dta`
3. select numeric variables
4. choose `detail` and format
5. click `Run Stata`

What happens underneath

1. Python validates inputs
2. Python writes `run.do`
3. Stata executes `mymean`
4. Stata writes `run.log`
5. Python displays the log

Main takeaway

A Stata dashboard can be built as a transparent pipeline: GUI choices become Stata syntax, Stata runs the analysis, and the GUI displays the results.

Python handles interaction. Stata handles statistics. The generated do-file keeps the workflow reproducible.

Thank you