Programming Tips for Empirical Researchers

(recommendations for reproducible programming).

Sergiy Radyakin,
The World Bank

Reproducible Research and Modern Data Analysis: Concepts, Skills and Tools
Porto Business School and Bank of Portugal
16-17 December 2019

Contents

Motivation

Reproducible programming. Why?

- gain confidence in your results;
- reproduce the results on demand;
- re-run the analysis with different assumptions/parameters;
- address the concerns of reviewers/critics;
- re-run the analysis with different input data;
- let other researchers reproduce your results.

Challenges:

- discipline;
- time-pressure;
- over-confidence and over-trust to own memory;
- not knowing or not understanding the problem;
- coordination and common ground among multiple co-authors/collaborators.

Version control for your project

Version control systems

Apache Subversion

Mercurial

GNU Bazaar

Git

Version control hosting sites

Assembla

BitBucket

SourceForge

GitHub

Self-hosting

We use self-hosting when:

- we trust no-one, not even commercial service providers (top-secret/military projects, know-how algorithms, etc);
- it's too expensive (e.g. we want a completely free solution, yet the hoster requires us to pay X USD per user per month);
- we want to commit data into the project, yet the data license prevents us from uploading it anywhere in the internet;
- we want to commit large dependencies, and the total repository size overshoots the footprint for a project provided by the hoster (rare);
- we want to manage the VCS version manually, e.g. remain on an older version for compatibility/legacy reasons.

Installing and maintaining a VCS on your own project server is simple in theory, but not so much in practice: backups, maintenance, security are all now your responsibility.

Self-hosting

Consult the license for every file that you are planning to upload, before putting anything on the Internet!

Once something is out there, there is usually no way of taking it all down!

Your project

Project composition - big-picture:

- Operating system (OS): Windows, Linux, etc;
- Tools: Stata, SPSS, SAS, R, Matlab, etc
- Dependencies: scripts and libraries, that are not written by you, but used in your code;
- Scripts: your code (source);
- Input data;
- Services and accounts.

Choice of OS:

- may limit the tools available;
- may introduce a licensing dependency;
- may limit reproducibility (how many of us have access to *Windows Server Datacenter Edition*?).

Localized operating systems:

- Beware of hardcoding anything like *"C:\Eigene Dateien\"* into the code of your project;
- This folder most likely will not exist when you try to replicate your results on a computer outside of Germany.
- Also mind the decimal delimiter, if your code imports data from e.g. an ASCII/Unicode text file, the delimiter used by default may be affected by the localization of the OS.
- Localized OS may also produce e.g. *'Mittwoch'* instead of *'Wednesday'* when formatting the dates output, which may or may not be acceptable to your subsequent code and readers of the output. Consider writing a code that is either fully transparent to OS localization, or fully robust to it.

Choice of Tools:

- a single project will heavily depend on the choice of one or more tools (Stata, SPSS, SAS, R, Matlab, etc);
- selecting an exotic tool will limit the reproducibility (less people who have it and have knowledge about how it works);
- open-source tools may have many derivatives/custom builds/etc, which complicates finding a proper version.

Record the full and verbose version description of the tool used, for example:

Example

```
Stata/MP 14.0 for Windows (64-bit x86-64) 4 cores
Revision 03 Aug 2015
Compile number 376
```

If the tool has a modular structure (such as SAS), record which [minimal] set of modules is necessary for the run.

Example

```
SAS9.4 TS1M6 for Linux 64-bit
```

- `Base SAS`
- `SAS/STAT`
- `SAS/GRAPH`

when modules are versioned independently, also record versions of all the modules, for example SAS/STAT 15.1.

Backward compatibility

- tools are usually not committed to the project repository, but rather described and mentioned as a pre-requisite;
- it is commonly assumed that newer versions will be available and that they will be able to run older code;
- this is often, but not always the case:.

Example

The Python language does not provide backward compatibility.

https://www.python.org/dev/peps/pep-0606/

Dependencies are (typically):

- focused on solving one particular task (e.g. a particular optimization method, or graphing library);
- partially or fully sourced (may still be compiled in the form of DLLs, Stata *.mlib libraries, etc).
- more volatile in terms of versions, releases may or may not be tied to the releases of the main tool;
- may or may not provide backward compatibility;
- may or may not be supported by the author;
- may or may not have a reliable distribution source/site;
- small enough to be committed to the project repository.

- If there are no licensing issues and size permits, commit the dependencies into the project repository.
- If the dependency is distributed as source code, commit the dependency source code, the build/make script, and the compiled version, as well as the reference to the distribution site.
- Make sure the correct dependency is used when running the project code, both you and collaborators may have multiple versions installed in different locations! (e.g. for different projects)

See also DLL hell and dependency hell problems at Wikipedia.

Dependency customization

- When a dependency contains a bug or a limitation, we sometimes resort to making changes to the officially distributed version to fix the bug and/or alleviate the limitation;
- This is usually when the author is unreachable or not willing to make changes to the official version (which is the preferred course of action).
- (of course the dependency must be distributed as a source, not a compiled version).
- All such customizations must be clearly described and the customized version included into the project repository.

Example

`spmap.ado` v1.2.0 by Mauricio Pisati from Jan 5, 2010, with customization by Sergiy Radyakin to support patterned fills in choropleth maps from Mar 12, 2011.

Obscure dependencies

- sometimes the dependency is used by the researcher so frequently and for such a long period that he/she forgets that this is not part of the official tool distribution (install and forget about it);
- This is also happening sometimes when such dependencies are centrally stored/managed, e.g. a Stata ado/plus folder managed centrally on a network drive and available to all the students of the university using the same lab.
- It may also be the case of indirect dependencies: you are using only calls to dependency A, which you declare in the description and include into the project repository, yet A requires B to run, which you've installed during installation of A, but forgot to include into the repository.

Expansive dependencies

A small dependency distributed as a source file may still refer to a major tool or package for performing a particular task!

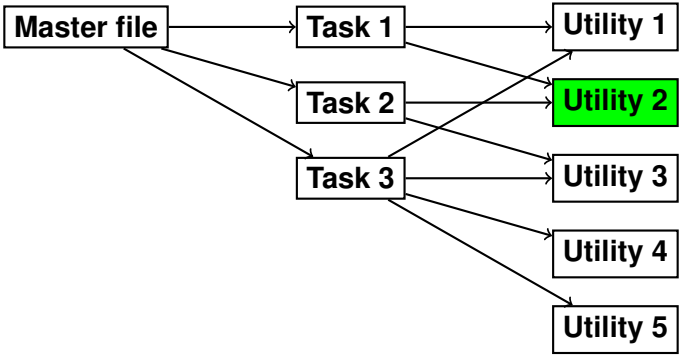
Example

`runmplus.ado` by Rich Jones, is implemented as a Stata ado file, but requires the MPLUS [commercial] package to be installed on the user's machine.

Many modules will refer to another statistical package, data conversion package, GIS, or publishing system (like LaTeX) to perform a certain operation. Make sure you've documented these other tools involved.

Your code/script

- this is usually most transparent part of the project - the author would know own files;
- commit all the source code written for the project into the repository.


Master file
Task 1
Task 2
Task 3
Utility 1
Utility 2
Utility 3
Utility 4
Utility 5

Your code/script

- Master file is the project startup file, will usually define all global parameters;
- Task files are the the particular steps of your project (data preparation / cleanup / merging / analytics / production of results, etc).
- Utils are portions of code that perform specific, identifiable, possibly repeatitive tasks.
- Utils may be project-specific (e.g. a procedure to pull the data from a particular project database) or generic (imputation procedure, graphing or tabulation procedure).
- Utils may be written by you (typically the project-specific ones) or by others (dependencies).

Services and Accounts

- More and more projects utilize internet services to perform a certain task through API consumption. Typical example: geocoding.
- Normally we do not have access to the server-side of the service, don't have exact information about the code/method and input data used there to provide the service, we treat it as the black box.
- While some services may provide a warning about the forthcoming change and maintain a compatibility/legacy API, often such services are completely silent about any changes under the hood.

Services and Accounts

This brings additional risks for reproducibility:

- The service may become unavailable.
- The service may change from free to paid with significant costs, restrict access to users from a particular country, or otherwise become inaccessible.
- The service may radically change the API rendering old code invalid.
- The service may change/update the underlying data used to provide the service.
- The service may update the algorithm used to provide the service.

The first 3 will result in inability to re-run the project, the latter 2 may run, but result in different data returned to the user and ultimately in possibly different results altogether.

Services and Accounts

- We rarely commit the private data (like the account/password) used to access the service into the repository.
- Instead, commit the instructions indicating which service is used, which site is used for registering with this service, any special requirements for registering or accessing the service.
- For example, *site X provides free geocoding services, but limited to 10,000 queries per 24-hour period*.
- The person replicating the project should be encouraged to continue, yet be warned about the possible obstacles.
- We, obviously, can't preserve the whole service itself.

Services and Accounts

If there is a risk that the service may not be available/accessible in the future:

- Try to localize service consumption as a separate step.
- Your project should prepare the queries/input data as a preceeding step.
- When you actually run the project: log the queries submitted to the service and the responses received.
- Commit the queries/responses to the project repository.
- During the project replication stage: use the stored information to simulate the response of the service.

Services and accounts - possible problems

Storing the results of queries to web-services is not without problems:

- service may return the response that is not unique (e.g. affected by randomness at the server side);
- region-specific depending on where the query originates from;
- privacy/licensing issues may prohibit committing the queries/responses into the project repository! (e.g. real addresses of the survey respondents).

Services and Accounts

Storing the responses of the web service will help to trace why your calculations yielded a certain result even if the service is no longer available, yet (in general) you will not be able to reproduce your results on a different/larger dataset.

Consider using offline alternatives!

Your project code

Master file

- meta-info: declare project name, purpose, author, contact info (if there is no separate readme file);
- describe the requirements;
- start the logging;
- setup the environment;
- declare dependencies;
- execute task files one after the other;
- finish logging and possibly assemble the output into a deliverable package.

Describe the requirements

- OS requirement? e.g. Windows-only
- CPU requirement, e.g. 64-bit CPU required.
- memory requirement, if any/known, e.g. min 16GB RAM required;
- disk space requirement, if any/known, e.g. min 2GB HDD space for temporary files during the execution;
- internet connectivity, if needed;
- tool requirement, e.g. Stata 13.0 or newer, etc.

Setup the environment

Example

```
version 13.0
clear all
global projpath "C:\PROJ\AER2019\"
global datapath "$projpath\data"
global outpath "$projpath\out"
set seed 123456
adopath ++ "$projpath\ado"
```

Setup the environment - paths

Example

```
global datapath "$projpath\data"
```

- Stata is agnostic to the use of forward slash and backward slash in paths, which makes the above a valid path in any OS, whichever convention it uses for writing paths to files.
- Yet when you pass this path to another tool in your toolset, it may very well object to the path written in this way, which may not be compliant with the convention of the OS.

Setup the environment - paths

Instead use the path management tools provided by your tool/language to combine paths:

Example

```
mata st_global("projdata", pathjoin(st_global("projpath"),"DATA"))
```

- This approach results in a full path valid for the current OS.
- Similarly, e.g. in C# .Net we write `Path.Combine()` and avoid mentioning the specific path delimiter in our code.

Version

- The version command is intended to declare how Stata should behave when executing commands.
- E.g. when `version 13.0` is specified, Stata 14.0 will behave the same as Stata 13.0

Version

But not so simple!

- bugs are not subject to such version control; if there was a bug and it was fixed in a newer version, it will yield different results, even under version control.
- some commands will change the behavior anyway, for example `save` will always save in the most recent format, regardless of the version statement. Stata itself will be able to read the file, but may be a problem passing such a file to an external tool that may not be aware of the recent Stata format.
- the `version` command affects Stata only! User-written modules have their own life and versioning, so an update to such an additional module will not obey the version command.
- update to such a module may increase the required base tool. For example, a module may get updated and require e.g. Stata 15.0 to run. Even if you've carefully written your code to be executed in Stata 13.0 and declared that in the `version` command, the requirement will bump up to 15.0 because of the dependency's requirement.

Randomness

- Many projects utilize randomness;
- Most will utilize pseudo-randomness, an RNG that produces a reproducible sequence of seemingly random numbers.
- Typically used for drawing samples, Monte-Carlo method and various simulations.
- Hence `set seed NNN` command ensures the same starting number and the same sequence every time the code is running.

Randomness

But not so simple

- An RNG may change! this happened during transition from Stata 10 to Stata 11, and again from Stata 13 to Stata 14, switching from the 32-bit KISS to the 64-bit Mersenne Twister (but which RNG to use may still be controlled by an appropriate command or under the version cmd);
- While setting the seed as a number is acceptable for most situations, StataCorp recommends setting the seed with the state string. The state string describes fully the state of the random number generator, and can be displayed with `c(seed)`, for example in Stata 13.0 the default state string is
  `X075bcd151f123bb5159a55e50022865700043e55`
- In Stata 14.0 it is different and about a page long.

See more here: https://www.stata.com/manuals13/rsetseed.pdf
and here: https://www.stata.com/statalist/archive/2011-12/msg00010.html
and here: https://arxiv.org/pdf/1810.10985.pdf

Randomness

But not so simple

- there could be more than one RNG involved!
- Stata uses randomness during sorting of data (totally not obvious to a novice user) and this randomness doesn't come from the same general purpose RNG, but from a specialized generator controlled with set sortseed.
- StataCorp writes:

help sortseed

`set sortseed` *should be used rarely. Access to the sorting seed is provided solely for those doing replication studies, where occasionally it is necessary to set the sort seed to produce numerically identical results for computations whose results may change slightly when the computations are run repeatedly. These changes are typically in the digits beyond those displayed by the command, but these might be compared nevertheless in a replication study.*

Randomness

But not so simple

- Some projects may utilize true randomness, see e.g. `http://random.org`; if so, the results better be stable regardless of the randomness (e.g. if randomness controls breaking the ties in an optimization procedure, it better converge to the same extremum, regardless of the ties broken, albeit at different number of iterations).

True randomness

Random.Org uses atmospheric noise to create random numbers that do not have a reproducible (pseudo-random) nature/sequense.

- It might be necessary to record the sequence returned by random.org as an input file for reproducibility of the calculation.

Precision

- Precision of calculation may be affected by the particular hardware used (CPU, FPU, GPU)

Stata

Results will differ slightly on other platforms because of compiler and hardware differences; Stata runs the same numerical code on all platforms.

`https://www.stata.com/support/cert/nist/`

- When presenting results, round them to precision appropriate for your method, or practical considerations. This will minimize the fluctuations of results on different machines/hardware.
- If epsilon-tolerances affect your results qualitatively, then, probably, the method employed is not robust.

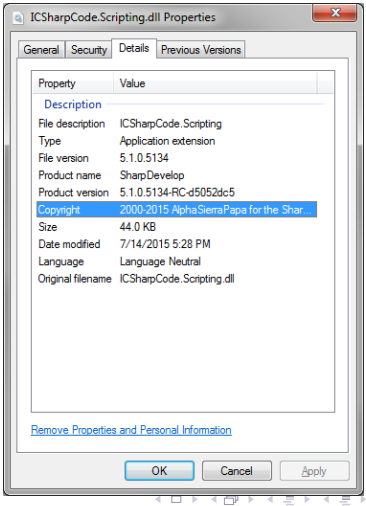
Precision

You have probably chosen a wrong method or approach if you are getting qualitatively different results (e. g. opposite conclusions) when:

- you re-run your project on a different hardware;
- you reduce/increase your sample by 1 (or a handful) of observations;
- when you change the calibrating parameters by a small margin, such as bandwidth in a smoothing function.

Declaring dependencies

- Most utils/libraries, especially circulated over the internet have meta-information attached to it to explain the version.
- For example, for DLLs a set of standard attributes is defined in the library's properties:
- These properties may be used to require a particular version of the library for the project, or can be inspected and reported on startup.



Declaring dependencies

- In Stata the command which gives information about the command implemented as an *ado* file.
- The set of attributes is not standardized, but usually includes the command purpose, version number, author name and contact information. For example:
- These properties are not saved into Stata's saved results and thus are not accessible from the syntax.

```
. which radar
c:\ado\plus\r\radar.ado
*! Date        : 27 March 2017
*! Version     : 1.10
*! Authors     : Adrian Mander
*! Email       : adrian.mander@mrc-bsu.cam.ac.uk
*! Description : Radar graphs

. return list

.
```

Declaring dependencies

One way of declaring dependencies is to simply mention them as a comment:

Example

```
// This project requires the following additional modules:
// - radar.ado v1.10 by Adrian Mander,
// - tabout.ado v2.0.7 by Ian Watson,
// - ...
```

This of course doesn't guarantee that the modules will be installed and will be of the corresponding versions.

Declaring dependencies

A better way is to mention each with a which command at startup of your project:

Example

```
// This project requires the following additional modules:
which radar // - radar.ado v1.10 by Adrian Mander,
which tabout // - tabout.ado v2.0.7 by Ian Watson,
which ...  // - ...
```

This gives the following benefits:

- if the command is not installed the project will stop right away at the initialization;
- if the command is installed, its actual version will be recorded in the log file.

Still, no guarantee that a specific version is used.

Declaring dependencies

A better way:

Example

```
// This project requires the following additional modules:
which radar // - radar.ado v1.10 by Adrian Mander,
findfile radar.ado
checksum r(fn)
assert r(checksum)==3932438196

which tabout // - tabout.ado v2.0.7 by Ian Watson,
findfile tabout.ado
checksum r(fn)
assert r(checksum)==508354043

which ...  // - ...
```

Write your code clearly - structure

- Decide on a structure of the code, stick to it. For example, every command in a separate file, or one file with public entry procedure and subroutine name as a parameter.
- When defining own commands, document any and all parameters/arguments (possible values, ranges, relationships).
- Document what would be the output of your procedure. Avoid mega-procedures that "do everything" (e.g. clean data, interpolate data, build a chart/graph).

Write your code clearly - assumptions

- When we write our code we (consciously or subconsciously) make assumptions.
- For example, “all incomes must be positive”, or “all shares of food-expenditures are between 0 and 1”
- Assert your assumptions!

Example

```
assert IncomeHH>0
assert FoodExpShare>=0 & FoodExpShare<=1
```

- Assertions work better than documentation - they are enforced.

Write your code clearly - style

- Use tabulation/indentation for nested loops, etc.
- Use comments to explain the idea of the step, not to duplicate help for the command being used.
- Write commands fully without abbreviation
  (is `tab` short for `table` or `tabulate` ?)
- Use `i` , `j` , etc for loop variables
- Use informative names for your variables of interest: `NumPersonsInvalidAge` , `PercUnemployed2YearsOrMore`
- Don't do `ПроцБезработных2ГодаИлиБольше`

Write your code clearly - reflect the idea

Consider a data cleaning code for the following task: a social support program makes payments to eligible citizens; no payment may be negative, but the data still has some negative values; substitute them with missing codes.

John's code

```
replace transfer=.  in 17
replace transfer=.  in 29
replace transfer=.  in 77
...
```

This solution is sensitive to the order of data and will create a mess if the order of observations changes, say if you add an additional earlier cleaning step.

Write your code clearly - reflect the idea

Consider a data cleaning code for the following task: a social support program makes payments to eligible citizens; no payment may be negative, but the data still has some negative values; substitute them with missing codes.

Jack's code

```
replace transfer=.  if transfer==-1108
replace transfer=.  if transfer==-589
replace transfer=.  if transfer==-2784
...
```

This solution is no longer sensitive to the order of data, but will not be effective if the data gets extended with new program participants data (replication on a larger scale).

Write your code clearly - reflect the idea

Consider a data cleaning code for the following task: a social support program makes payments to eligible citizens; no payment may be negative, but the data still has some negative values; substitute them with missing codes.

Jessica's code

```
replace transfer=.  if transfer<0
```

This solution directly expresses the idea formulated above and will work even if the dataset becomes larger/smaller, whether it has or doesn't have this problem.

Tests

- Not productive code;
- Manifest requirements to code, reflect documentation;
- Help maintain integrity of the project with large teams, temporary contributors, etc.
- Require additional time to write and maintain, save time in making sure the new version ‘works’;
- Inevitable in large projects.
- Daily practice of reproducibility.

Writing Tests

- Maximize the code coverage, not number of benchmarks;
- Include corner cases, not just mainstream;
- Cover invalid situations/parameter values;
- Match tests to project specification/requirements;
- Update your tests if your specification changes;
- Do not accept the code commit unless it passes all the tests.

Writing Tests

Where to get test values?

- the project specification may suggest;
- for simple procedures may work out an example manually;
- for complex procedures compare with alternatives;
- for optimized algorithms - compare with non-optimized version.

Test data

- you may need some data to test your procedures;
- you may be inclined to commit your original project data;
- not always a good idea - privacy, size;
- consider writing a code to generate hypothetical data;
- has known properties, patterns;
- smaller size - often a few lines can generate a few GBs of data (if necessary);
- possible to evolve the test data generating code with project changes.

Practice

- Have your tests in a different set of files, perhaps a different project folder;
- Typically a different master file for testing is created, e.g. test_all calls test1, test2, ...;
- Stata conveniently provides an **assert** command for writing test cases:

example

```
mysum 1 2
assert r(sum)==3
myagecalc bday intervday, generate(age)
assert age>=0
```

- In many cases testing is aborted on the first test that doesn't pass.
- Yet running the remaining tests (and seeing which others have failed) often helps in locating the change that caused the breakdown.

Test the non-obvious

- It is obvious that your implementation of the OLS-method should return the correct values of the regression coefficients.
- It is not obvious (but often assumed by default), that your code should not modify/destroy the data in memory, erase files, or conflict with global variables, etc.
- Note that for the dataset Stata provides an off-the-shelf tool `datasignature` to detect changes in the data. In other languages one can either specify read only access to data or utilize own integrity checks as a last resort.

Things we always forget

Things we always forget: user/group profile

- The user profile (user configuration file) may have serious consequences for the smooth running of the whole project.
- We generally do not associate the user profile with a particular project,
- We set up the profile once when the tool is installed,
- And then we forget about it.
- When it comes to reproducing the results on a different machine, we usually have no idea what was written in the user profile on the original machine.

Example

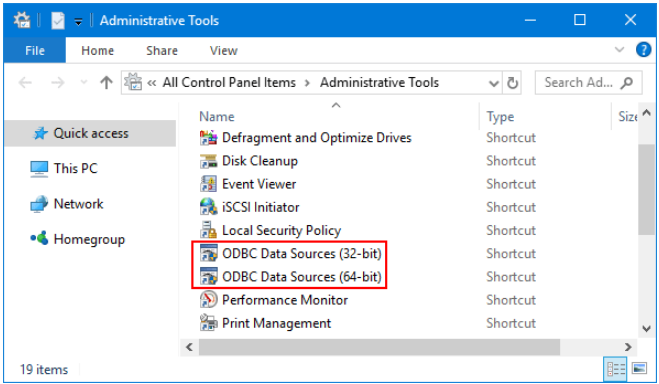
Stata has a default setting of 5,000 for number of variables that can be used in a dataset, but Stata can clearly work with larger datasets, though requires this setting to be changed.

Hence, many users include something like `set maxvar 32767` into their profile.do file.

On a new machine, when your code doesn't specially define the maxvar, it relies on the default setting, which (with a large dataset) may cause the program to stop with an error. When such error occurs deep in the toolchain, neither the cause of the problem, nor the solution to it may be obvious.

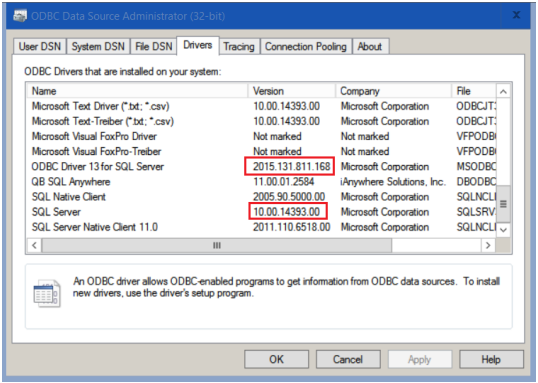
Things we always forget: ODBC configuration

Open Database Connectivity (ODBC) defines the database connections (at system level) which may be utilized by the programs on this computer.



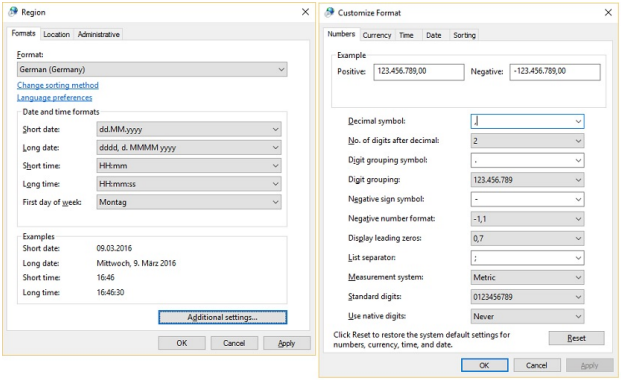
Things we always forget: ODBC configuration

What were the ODBC drivers used to connect to the data source? What versions? What locales? Where to acquire them in the future?



Things we always forget: locale configuration

The locale settings may affect all string-to-binary and binary-to-string conversions performed on the computer.



- Your collaborators most likely woudn't like to have to change the system settings to ones uncommon for their area.
- Instead, rely on directly specifying the delimiter in your code, perhaps defining the format as a configuration parameter in the master file.

Other things we always forget

Numerous other settings may affect the exact appearance of the output produced by your code. For example, in Stata:

- Screen (monitor) width and resolution may affect the line width in characters, causing the tables appear correctly on some machines, and line-wrapped (distorted) on others.
- Graphs are colored following the default color scheme, which may be different for different users.
- Current directory may point to Stata's program directory, or user's own directory, or whatever is set in the user's profile.

For more info

- Scott Chacon and Ben Straub. 2014. Pro Git. 2nd ed. ISBN: 9781484200766.
- Roy Osherove. 2013. The Art of Unit Testing. 2nd ed. ISBN: 9781617290893.
- Christopher F. Baum. 2016. An Introduction to Stata Programming. 2nd ed. ISBN: 9781597181501.
- Nicholas J. Cox. 2005 Suggestions on Stata programming style. The Stata Journal (2005) 5, Number 4, pp. 560-566.
- Nicholas J. Cox and H. Joseph Newton 2014 One Hundred Nineteen Stata Tips ISBN:9781597181433.
- Bill Gould. 2012. The Penultimate Guide to Precision. (Stata blog post).