Programming Practices for Research in Economics

Introduction & Motivation

Ulrich Bergmann Ursina Schaede Dora Simon Carlo Zanella Christian Zünd

Department of Economics, Univeristy of Zurich

Fall 2018

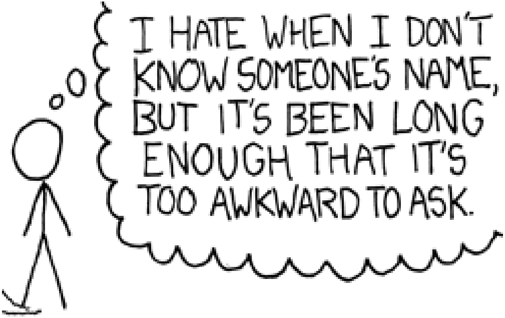
Welcome!

Introductions: Who We Are

5 PhD students

- Uli
- Christian
- Ursina
- Dora
- Carlo

Introductions: Who are You


I HATE WHEN I DON'T KNOW SOMEONE'S NAME, BUT IT'S BEEN LONG ENOUGH THAT IT'S TOO AWKWARD TO ASK.

Logistics: Basic Information

group is a mix of “for credit” and audit students

- For credit students need to * enrol using sheet we will pass around in the last week * register for course on `UZH module booking` * hand in assignment 4 weeks after course ends * grading: pass/fail based on final assignment

sessions are designed to be interactive

- mix of *live coding*, *small challenges*, *longer exercises*
- We want to get you comfortable using your computing environment to solve problems
  - Bring your laptop!
  - We expect you have completed the installation guide and have all software installed.
  - Ask questions!

Logistics: Structure of each day

- Session 1: 9.00-12.00
- Session 2: 13.00 - 16.00
- Expect coffee breaks in each session
  - Exactly when depends on the leader of a session, and the material
- No scheduled office hours
  - Talk to us during the day
  - Email for appointment after class if want to discuss assignment

Logistics: Where to Find Information

- Course website:
  - `pp4rs.github.io/2018-uzh`
- Installation Guide:
  - `pp4rs.github.io/installation-guide`
- Course Chatter:
  - `pp4rs.slack.com/`, #pp4rs-2018
- GitHub repositories:
  - `github.com/pp4rs`

Logistics: Assignment

The basics

- One final assignment
- Can be submitted in groups of 1-3 people
- Due 4 weeks *after* last class
- Propose to us an idea before you start

Use what you learn in this course to solve a non-trivial economic problem

- Code must be in split into meaningful sub-files
- Solution must be submitted using GitHub
- Solution must be executable using a single line of code, e.g. using`Snakemake`, a `Shell` script, `R Markdown`

Logistics: Social Event

- Join us for casual drinks
- When: This Friday (August 31st), after class
- Location: TBA

Motivation

Why? – Practical reasons

Broad Goals for the Course

1. Improve computing skills, so you can do things you could not do before
2. Show how can do a given set of things with less effort
3. Increase the confidence in results that are produced this way (both yours and others' results)

Why? – Academia

EU policy for all publicly funded research being *open* by 2020

- 4Rs (Pagan and Torgler, Nature 2015)
    - Reproduction: Can others reproduce you results using your data?
    - Replication: Can others replicate your results using new data?
    - Robustness: Do your results depent on the assumptions you made?
    - Revelation: Do you communicate the reasoning for your conclusions transparently?

Our generation must adapt to the challenge of "open science"

- But we are lacking the skillset to do so
- Lack of proper training opportunites (particularly with a Social Science focus)

We hope this is a first step in filling this gap

Why? – The real world

Econ PhD graduates report that

- R/Python is the software used required in their first job
- You need to be able to cooperate on coding tasks with others

The big shots (Google, Amazon, Netflix, Pandora) complain that

- Hiring Econ PhD is a massive cost factor because we need to learn to use standard software and to code on the job
  - ... they really hate this
- FYI: These are former Econ graduates themselves

Why? – The real world #2

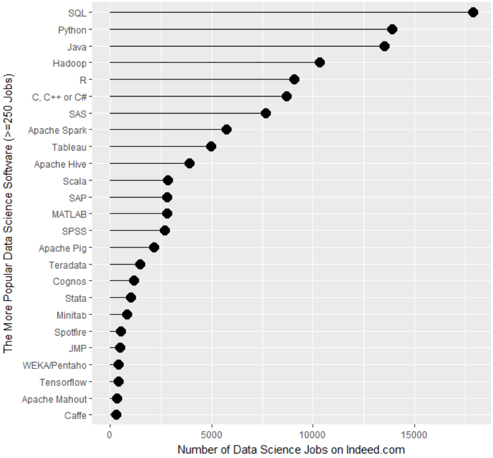


Figure 1: Required software for data-science jobs

What We Teach

Core topics:

1. Unix shell
   - Text based interface to computing
   - Automate repetitive tasks
2. Git
   - Track/control and share work
3. Snakemake
   - Automate the execution of your research project
4. Python/R
   - Build modular code to solve typical economics problems

Applications:

5. Machine Learning
6. Web scrapping
7. Geo-spatial Data

We Cannot Cover Everything

We miss (important) topics such as:

- Databases: SQL, mySQL, SQLite etc.
- Unit testing
- Complete documentation of Research Projects
- High Performance Computing
- many others ..

Guiding Principles

Rule 1: Write Programs for People Not Computers

Code that a computer can understand $\neq$ code a human understands

- Does your difficult-to-understand code do what it's suppose to?
- Makes it hard for other collaborators and researchers to use your code
  - Future you *is an* **other** *collaborator*

Rule 1a: Write Many Short Scripts / Code

- Short-term memory can hold 7 ± 2 items

Functions:

- short, readable, take only a few inputs each
- **Rule of thumb**: code looking very complex = you're probably doing it wrong
- think of a function/script like a paragraph * limit it to one idea, e.g.
  - `transform()` transforms x to x'
  - `do_stuff()` works with x'

Rule 1a: Write Many Short Scripts / Code

Scripts:

- **What not to do:** 5000 line scripts that execute an entire project
  - What does the variable on line 4100 mean?
  - How does it relate to it's initial defintion on line 1350?
- **What to do:** Split into meaningful smaller tasks, e.g.
  - `clean-data.R` cleans the data
  - `create-vars.R` creates auxiliary variables
  - `regressions.R` does the regressions
  - `tables.R` makes the tables
  - `plots.R` makes the plots

Rule 1b: Use meaningful variable names

- `p` less useful for long term memory than `price`
- `i`, `j` are (almost) OK for indices in small scopes
    - `i_subject` and `i_trial` might be better
- be careful with the use of ambiguous names like `temp`

Rule 1c: Make code style and formatting consistent

Which rules don't matter — having rules does

- brain assumes all differences are significant
  - inconsistency slows comprehension
  - annoying not to remember what your function x was called
  - freaks out your co-authors

What to do:

- consistent naming conventions
  - **Camel Case:** `cleanData()`
  - **Snake Case:** `clean_data()`
  - **Kebab Case:** `clean-data()`
- keep each line of code within 80 characters
- whitespace, identation & comments are your friend

Rule 1d: Document your code well

Document design and purpose, not mechanics

- focus on what the code doesn't say
- or doesn't say clearly
  - e.g., file formats
  - an example is worth a thousand words…
- makes the next person's life easier

Rule 2: Use a version control system

- tracks changes
- allows them to be undone
- keeps folders clean
- supports independent parallel development
- essential for collaboration

  Email is not version control.

Rule 2a: Put everything that has been created manually in version control

Add all inputs

- things generated by you / others
- scripts, data (if not too large), images from other sources

Leave out outputs

- things generated by the computer
- use build tools to reproduce those instead
- unless they take a very long time to create

Rule 2b: Work and track in small steps

this allows to

- move back to exactly this point in
- understand the progress of your project better

How it's done:

- create snapshots regularly
- creates snapshots for logical steps, e.g.
  - "added best practice slides about clean code"
  - "added best practice slides about version control"

Rule 2c: Use an issue tracking tool

- a shared to-do list
- items can be assigned to people
- supports comments, links to code and papers, etc.
- “Version control is where we’ve been, the issue tracker is where we’re going”
- **Email is not an issue tracker.**
  - ... although my advisor would seemingly disagree ;-)
- `GitHub` has a build in issue tracker

Rule 2d: Use pre-merge code reviews

- Develop on different branches or forks
- Review changes before merging in version control
- This significantly reduces errors

Rule 3: Let the Computer Do the Work

Computers exist to repeat things quickly and accurately

- 99% accuracy vs 63% percent chance of error in *simple* tasks

Rule 3a: Let the computer repeat and execute tasks

Functions

- Rule of 3:
  - If you copy-paste code 3 times, write a function instead
  - This reduces error rates and work

Projects

- Write little programs for everything
  - Even if they're called scripts, macros, or aliases
- Easy to do with text-based programming compared to GUIs
- We will search for the 'magic button'
  - One command that will execute your entire project
  - . . . after you have written ordered instructions

Rule 3b: Use a build tool to automate workflows

- Build tools originally developed for compiling programs
- Workflow becomes explicit
- This will become your ‘magic button’

Rule 4: Define things once, and only once

Every input must have a single authoritative representation in the system.

- define something *exactly once*
  - make calls to that input each time you need to reference it
  - Example: Define important parameters in a dictionary, import into each script

Rule 5: Optimize Software Only After It Works Correctly

Even experts find it hard to predict performance bottlenecks

- get it right, then make it fast
- small changes can have dramatic impact on performance
- don't be scared to ask questions about how you could further improve

Use a profiler to identify bottlenecks

- reports how much time is spent on each line of code

Rule 6: Plan for Mistakes

No single practice catches everything

- turn bugs into test cases
- we can only try to prevent many
- practice makes perfect

Rule 6a: Add assertions to programs to check their operation

“This must be true here or there is an error”

- no point proceeding if the program is broken. . .
- error messages are implemented and expected by other users and programmers
- . . . they also help you to make less mistakes and find errors faster

Can you summarize all of that?!

1. Use text-based interfaces
2. Write simple and clean code
3. Put everything in version control
4. Turn history into scripts
5. Use test-driven development

A Warning

Where your brain may end up

why should i waste my precious time learning another faddish programming language?

A Warning

15 days × 6 hours/day = 90 hours of content

- that's a lot! ... and fast

You **will be tired** at various points

- but don't confuse that with questioning the point of the course

Nobody can transform their practices overmight ...

- but persistance will make your programming life much, much more efficient
- think of us as a 'kick in the arse' to get you started

Let's Get Started!

Acknowledgements

This module is based on the 2016 and 2017 versions of the course:

- Programming Practices For Economists, by Lachlan Deer, Adrian Etter, Julian Langer & Max Winkler

It is designed after and borrows a lot from:

- Effective Programming Practices for Economists, a course by Hans-Martin von Gaudecker
- Software Carpentry's Managing Software Research Projects lesson

Guiding Principles borrows a lot from the paper

- Wilson G, Aruliah DA, Brown CT, Chue Hong NP, Davis M, Guy RT, et al. (2014) Best Practices for Scientific Computing. PLoS Biol 12(1): e1001745.

License

Suggested Citation:

- Ulrich Bergmann, Lachlan Deer, Adrian Etter, Julian Langer, Ursina Schaede, Dora Simon, Max Winkler, Carlo Zanella & Christian Zund, 2018, `Introduction and Motivation`, Programming Practices for Research in Economics, University of Zurich

Programming Practices Team

Programming Practices for Research in Economics was created by

- Lachlan Deer
- Adrian Etter
- Julian Langer
- Max Winkler

at the Department of Economics, University of Zurich in 2016. These slides are from the 2018 edition, conducted by

- Ulrich Bergmann
- Ursina Schaede
- Dora Simon
- Carlo Zanella
- Christian Zünd