# COLUMBIA ECONOMICS PROGRAMMING TUTORIAL

GUY ARIDOR

## GOALS OF THE TUTORIAL

- Plan to give you a high-level overview of programming and (a somewhat opinionated view on) how to think about programming
- Help you figure out what tools exist out there and how to best utilize them
- Give you a brief introduction to R

## WHO AM I?

- Data TA for the semester, Office Hours are 4 - 6 PM in Lehman Group Study 1!
- Where does my perspective on programming come from?
  - For a (brief) period of time I was a software engineer, mainly working on server-side and client-side web development as well as dealing with distributed data storage + processing
  - Written production code in the following languages: Java, JavaScript, Python, PHP, C++
  - Worked on research projects in the following languages: Julia, R, Python

## OVERVIEW OF PROGRAMMING

- The art of telling a (dumb) computer what to do - no ambiguity!
- In general, worry about the following things
  - Correctness of code (does it do what you want it to do)
  - Readability and reasonability of code (can people understand what it does)
  - Extensibility of code (can you extend it to do things beyond what you originally intended)
  - Testability of code (programmatic testing)
  - Efficiency of code (how fast does it run, how does it scale)

## PROGRAMMING AS TRADEOFFS

- Every programming project requires a balance between the elements described
- Important to think about what matters most to us when doing research
  - Correctness dominates everything else, don’t want to be fooled into fake results. An elegant, efficient solution that you can’t convince yourself is 100% correct is not worth it
  - Should view efficiency in terms of its opportunity cost of time - efficient programs increase iteration speed (simulation takes 1 minute instead of 1 hour).
    - However, don’t do premature optimization (only optimize when it becomes necessary)
  - Code should be readable and reasonably self-documenting.
  - Worry about extensibility as a secondary concern in this context

## OVERVIEW OF PROGRAMMING LANGUAGES (RELEVANT TO YOU)

- Statistical Computing (data cleaning, estimation)
  - STATA - non-programmer friendly, but can be inflexible and awkward. Proprietary language, has a lot of econometric packages
  - R - Open source, slow at times but has a wonderful ecosystem from the statistics community
- General Purpose Programming Languages
  - Python - Using NumPy / Pandas / SciPy / StatsModels should let you approximately have a statistical computing environment, but Python is more general purpose (useful language to know in general!)
  - Julia - Developed with a focus on efficient scientific computing and ease of use. Modern language, very efficient but YMMV since it still is an immature language (https://julialang.org/blog/2012/02/why-we-created-julia)

## CHOOSING A PROGRAMMING LANGUAGE

- One reason there exists so many popular languages is that each has an advantage, each has its strengths and weaknesses
- Certain communities favor different languages
- Generally, picking a language should be a tradeoff between
  - your (and others involved) familiarity with the language
  - the timeframe you hope to solve the problem in
  - how good the language is for the job
  - (as an undergrad) the usefulness of a language for future projects

## EXAMPLES

- Project: Scrape data from the web and then run some linear regressions on the data
  - Python is likely a good choice here or scrape the data in Python, save to a csv and then use R / STATA
- Project: Run some computationally intensive simulations or many simulations
  - In these cases, efficiency matters and you want a language like Julia or Python (maybe MATLAB if you're in a macro project - but (personal opinion) Julia or Python are generally better
- Project: Some data cleaning and running some linear regressions on a small dataset (likely your project)
  - STATA or R likely ideal

## THE PRIMITIVES OF PROGRAMMING

- Variables
- Types (especially "number" types and how they are represented on a computer - this matters a lot!)
- Control Flow (if statements, for loops)
- Functions
- Highly recommend that you at least skim through an introductory CS course set of lecture slides to learn basic data structures and programming constructs

## HOW TO STRUCTURE YOUR CODE

- Most important thing is to have your work be reproducible with minimal frictions!
  - Should be able to run one script that does everything you need to do to produce your figures / estimates!
  - Single sources of truth - this saves a lot of headaches! If you have a definition of a variable of interest, define it once and reuse
- If you have many files, break up your scripts into files and directories that are sensible
- Structure is more of an art than a science and is hard but be conscious of your actions

## BIG PROBLEM OF STRUCTURE

- Projects change over time, hard to maintain documentation of a constantly evolving project without time investment
- Documentation is good, but not sufficient
- The problem: Suppose that someone (or you) come to your code months from now and need to grok what's going on and how certain files and functions are related - need to form a mental model of project structure
- May not have to worry about this (for small projects)!
- Gentzkow-Shapiro have a nice discussion about this here: https://web.stanford.edu/~gentzkow/research/CodeAndData.pdf

## READABILITY

- What does this mean?
  - You should write code that others can read (and verify correctness and understand what you're doing)
  - You should write code that your future self can read and understand
  - Quip: There are two hard problems in computer science: naming, cache-validation, and off-by-one errors
- How to do it?
  - Self-documenting code (good variable names, good function names)
  - Example: v = d/t (or velocity = distance / time)
  - Extensive documentation of files and project via comments - the most useful comments are block-level comments
    - Useful to have a README explaining how to get started and overall structure of code
    - Should view comments as explaining WHY something is done, not HOW it is done (or as anti-pattern)
    - Possible to use version control commit messages as an alternative

## VERSION CONTROL

- Very useful (especially for projects with several people), but there is somewhat of a learning curve
- High-level idea is to “checkpoint” activity and let you give historically evolving documentation
- Think of the last term paper that you wrote. Might have had files such as:
    - DRAFT_1_31, DRAFT_2_7, DRAFT_2_7_COMMENTS
- Now imagine you just had to compare about one file DRAFT and you could record (and view or revert to) each incremental change
- For your purposes, this is what version control gives you for your code and data (but there is a lot more)
- Links to learn git and some more documentation are found on the GitHub repo

## TESTING CORRECTNESS OF CODE

- Should always test your code in one way or another
- Think about cases where you know what the output should be given a particular input and test this!
- Can do this manually or can write programmatic testing scripts
  - Only really should do the latter if your project becomes larger and larger and there are many moving parts
  - Programmatic testing can be really annoying but you'll be very happy you did it when it's done - increases iteration speed and confidence in your results

## EFFICIENCY OF CODE

- For most of your projects, should think of efficiency optimizations as being second order
- Efficiency helps for iteration speed and (in more complex projects) tractability
- Things to look out for when you think things are running too slowly:
    - Vectorize code if possible, especially in R (vector operations instead of for-loops)
    - Think about parallelization (but be wary of Amdahl's Law - decreasing returns to increased number of processes)

## ABSTRACTION

- *Abstracting* something means to *give names* to things, so that the name captures the core of what a function or a whole program does.
- Allows us to break down our complex problems into smaller sub-problems so we can reason on a high-level about our complex problem.
- Example: Suppose I ask you to draw a square.
    - You immediately know what to do - draw a rectangle with all equal sides
    - How do you know what a rectangle is? To draw a rectangle you need to do the following => draw two lines parallel to each other, of the same length, and then add another two parallel lines perpendicular to the other two lines, again of the same length but possibly of different length than the first two
    - But what is parallel, perpendicular, line?

## ABSTRACTION IN PROGRAMMING

- Allows you to reason at a higher-level about your code.
- Concretely, think about the different libraries you utilize when you write code - don't need to worry about the underlying engineering or mathematical details for data processing or estimation
- A word of caution: be wary of leaky abstractions - sometimes for instance it's critical that you know the details of the implementation procedure. Think about this before you write or use any sort of abstraction (class, library, function, etc.)

## NOW TO R…

## R TUTORIAL INSTRUCTIONS

- Now that you have a high-level idea of how to program, it's easiest to learn how to program by actually doing it!
- The tutorial will probably take you more than an hour and part of it might be a bit difficult (ask any questions you need, now or in office hours…)
- Tutorial also has some supplementary reading material that you don't need to read now (but should read later)