Coding for Economists

A Language-Agnostic Guide to Programming for Economists

Ljubica "LJ" Ristovska

Organized by
Harvard Economics Professional Development
Spring 2019

Goals

The first part of the presentation focuses on general computer science concepts, guidelines, and programming tips with an aim to:

- Make managing data and programming easier
- Maintain accuracy and efficiency
- Ensure reproducibility of results
- Preserve an organized workflow throughout the project's lifecycle
- Preserve your sanity

The second part of the presentation with Frank Pinter will introduce version control via Git.

Disclaimers

1. I am not a computer science expert - the content of this presentation is assembled based on my computer science background and my own programming experience.

2. I have not personally used *all* of the below tricks in my work, but am providing them here for your reference.

3. The presentation is meant to be language-agnostic:
   - We will not be learning the concepts in a specific programming language.
   - Instead, we will focus on general principles that apply to any language.

General Principles

The two main principles for coding and managing data are[1]:

1. Make things easier for your future self
2. Don't trust your future self

[1] Like the agents we study, we too as programmers can be time inconsistent - the goal is to go from a naif to a sophisticate.

General Principles

The two main principles for coding and managing data are[1]:

1. Make things easier for your future self
2. Don't trust your future self

Specifically:

- Be consistent in formatting, style, organization, and naming
- Make the computer do the work
- Reduce copy-pasting and repetition
- Test often and test modularly
- Document often but "minimally"
- Increase efficiency

[1] Like the agents we study, we too as programmers can be time inconsistent - the goal is to go from a naif to a sophisticate.

Outline

1. Organization
2. Data types, structures, and automation
3. Abstraction
4. Debugging and testing
5. Documentation
6. Efficiency
7. Miscellaneous
8. References and further resources

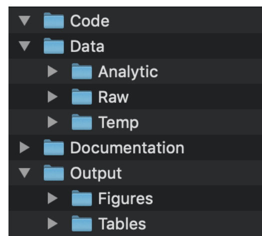
Be consistent

Adopt an organizational system and adhere to it. This system should include consistency in:

- Directory structure
- Code and comment style
- File naming conventions
- Variable naming conventions
- Output data structure

Directory structure

- Data
  - Raw
  - Analytic
  - Temp
- Code
- Output
  - Tables
  - Figures
- Documentation

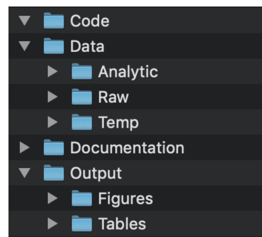


Wow, I know where everything is!

The documentation directory is optional - I personally like to keep any documentation that came with the data in there. A short README.txt file that describes how to run the code is also recommended.

Directory structure

- Data
  - Raw
  - Analytic
  - Temp
- Code
- Output
  - Tables
  - Figures
- Documentation



Wow, I know where everything is!

The documentation directory is optional - I personally like to keep any documentation that came with the data in there. A short README.txt file that describes how to run the code is also recommended.

Why is this useful? Preserves organization over time and makes automating certain tasks (e.g., file input and output) easier.

Code and comment style

- Small things like consistent spacing matter
- One statement per line
- Indent code that "belongs together" (i.e., anything within braces and/or parentheses)
- Use block commenting for separating code sections
- Write comments *before* the line that you want the comment to apply to
- Break long lines (I like to indent after breaking)
- Break down long algebraic expressions

```
********************************
* ESTIMATE EARNINGS BY GENDER
********************************
```

Block commenting: Here begins a long section of code that estimates earnings by gender.

Code and comment style

- Small things like consistent spacing matter
- One statement per line
- Indent code that "belongs together" (i.e., anything within braces and/or parentheses)
- Use block commenting for separating code sections
- Write comments *before* the line that you want the comment to apply to
- Break long lines (I like to indent after breaking)
- Break down long algebraic expressions

```
*******************************
* ESTIMATE EARNINGS BY GENDER
*******************************
```

Block commenting: Here begins a long section of code that estimates earnings by gender.

Why is this useful? Makes code readable for people (and future you), not just machines.

More resources for code style by Google: for Python and R, applicable broadly.

File and variable naming conventions

- Use distinctive and informative file and variable names (e.g., `temp` vs. `gender`).
- Be generous and consistent with use of mixed case and underscores (e.g., `laborElasticity` vs. `laborelasticity` vs. `labor_elasticity`).
  - Note: not all languages will distinguish between upper and lower case characters in variable names – make sure to check!
- Length of variable names should be inversely proportional to how often you use it (e.g., iterator `i` vs. `coefficient_relative_risk_aversion`).
- But when in doubt, be more descriptive rather than less.

Output data structure

When creating a data set, particularly when saving one in that special "Analytic" directory, adhere to the following:

- Know that there are different types of input data sets (e.g., .csv, .xls, .dat) and each programming language has a specific output data set format (e.g., .mat, .dta, etc.)
- Know which variables uniquely identify an observation for each data set (called **keys**)
- Values do not have any internal structure, i.e., do not need to process variables to use them
- The data does not contain any redundant information, e.g., do not save a panel data set as a set of dummies year_1980, year_1981...

Output data structure

When creating a data set, particularly when saving one in that special "Analytic" directory, adhere to the following:

- Know that there are different types of input data sets (e.g., .csv, .xls, .dat) and each programming language has a specific output data set format (e.g., .mat, .dta, etc.)
- Know which variables uniquely identify an observation for each data set (called **keys**)
- Values do not have any internal structure, i.e., do not need to process variables to use them
- The data does not contain any redundant information, e.g., do not save a panel data set as a set of dummies year_1980, year_1981...

Why is this useful? Clarifies relationships between different data sets. Makes for easy merges of data sets.

Outline

1. Organization
2. Data types, structures, and automation
3. Abstraction
4. Debugging and testing
5. Documentation
6. Efficiency
7. Miscellaneous
8. References and further resources

Data types

Every programming language generally has the following "primitive" data types:

- **booleans**: stores binary 0 or 1 values, typically 32 bit, maybe less
- **integers**: stores mathematical integers, typically 32 bit
  - Integers between -2,147,483,648 and 2,147,483,647 can be represented accurately
- Decimal number storage, from least precise to most precise:
  - **floats**: typically 32 bit
  - **doubles**: typically 64 bit
  - **decimals**: typically 128 bit, not available in every language
- **strings**: stores sequences of alphanumeric characters

Data types

Every programming language generally has the following "primitive" data types:

- **booleans**: stores binary 0 or 1 values, typically 32 bit, maybe less
- **integers**: stores mathematical integers, typically 32 bit
  - Integers between -2,147,483,648 and 2,147,483,647 can be represented accurately
- Decimal number storage, from least precise to most precise:
  - **floats**: typically 32 bit
  - **doubles**: typically 64 bit
  - **decimals**: typically 128 bit, not available in every language
- **strings**: stores sequences of alphanumeric characters

Why is this useful?

- Generally cannot combine types
- With large data sets, the choice of how to store information will save you space on disk and also run time – different data types have different storage requirements and they are more suited for certain operations
- Careful when doing operations or merging data with decimal types or strings
  - Ensure that precision and string length is the same across merging data sets or variables to avoid truncation

Data structures

- While every language has the aforementioned primitive data types, some languages have more advanced **data structures** (e.g., vectors, matrices, lists, data frames, stacks, queues, dictionaries...).
- Typically these data structures consist of multiple instances of the primitive data types.
- Different data structures are stored/linked differently and are more conducive to certain types of operations than others.

Data structures

- While every language has the aforementioned primitive data types, some languages have more advanced **data structures** (e.g., vectors, matrices, lists, data frames, stacks, queues, dictionaries...).
- Typically these data structures consist of multiple instances of the primitive data types.
- Different data structures are stored/linked differently and are more conducive to certain types of operations than others.

Why is this useful? Using the appropriate data structure for the task at hand can improve code readability and efficiency.

Automation basics

Every programming language must have syntax for the following:

- **if-then-else statements**: execute a piece of code only if a specified statement evaluates to true
  - ```
    if this_lecture_is_useful==true then
       stay_at_lecture()
    else
       go_do_work()
    ```
    ```

Automation basics

Every programming language must have syntax for the following:

- **if-then-else statements**: execute a piece of code only if a specified statement evaluates to true
  - ```
    if this_lecture_is_useful==true then
       stay_at_lecture()
    else
       go_do_work()
    ```
- **for-loops**: execute a piece of code repeatedly using a loop counter (typically used when you know the number of iterations ahead of time)
  - ```
    for i=0 to last_minute_of_lecture {
       pay_attention()
    }
    ```

Automation basics

Every programming language must have syntax for the following:

- **if-then-else statements**: execute a piece of code only if a specified statement evaluates to true
  - ```
    if this_lecture_is_useful==true then
       stay_at_lecture()
    else
       go_do_work()
    ```
- **for-loops**: execute a piece of code repeatedly using a loop counter (typically used when you know the number of iterations ahead of time)
  - ```
    for i=0 to last_minute_of_lecture {
       pay_attention()
    }
    ```
- **while-loop**: execute a piece of code repeatedly until a specified condition fails to be satisfied (very useful when you *don't know* the number of iterations ahead of time, but beware of infinite loops!)
  - ```
    while this_lecture_is_boring ==false {
       stay()
    }
    ```
    ```

Automation basics

Every programming language must have syntax for the following:

- **if-then-else statements**: execute a piece of code only if a specified statement evaluates to true
  - ```
    if this_lecture_is_useful==true then
       stay_at_lecture()
    else
       go_do_work()
    ```
- **for-loops**: execute a piece of code repeatedly using a loop counter (typically used when you know the number of iterations ahead of time)
  - ```
    for i=0 to last_minute_of_lecture {
       pay_attention()
    }
    ```
- **while-loop**: execute a piece of code repeatedly until a specified condition fails to be satisfied (very useful when you *don't know* the number of iterations ahead of time, but beware of infinite loops!)
  - ```
    while this_lecture_is_boring ==false {
       stay()
    }
    ```

Pseudo code can be really useful when programming complicated procedures. It is language-agnostic and suitable for conveying general concepts without the burden of syntax.
    ```

Automation tips

Always use scripts, even for testing.

- Very rarely should you be typing directly into the interpreter.
- Whatever you don't want to keep, you can comment out, flag out, or save in a separate file (Is it really a good idea to delete it? More with version control in part two of this presentation.)
- Highly recommended to keep a master script, which executes all scripts in order as intended.
  - For example, a script `master.do` in Stata will call on, in order, the scripts `data_cleaning.do`, `sample_selection.do`, `summary_stats.do`, `regressions.do`

Automation tips

Always use scripts, even for testing.

- Very rarely should you be typing directly into the interpreter.
- Whatever you don't want to keep, you can comment out, flag out, or save in a separate file (Is it really a good idea to delete it? More with version control in part two of this presentation.)
- Highly recommended to keep a master script, which executes all scripts in order as intended.
    - For example, a script `master.do` in Stata will call on, in order, the scripts `data_cleaning.do`, `sample_selection.do`, `summary_stats.do`, `regressions.do`

Why is this useful? Maintains a clear record of steps that were taken and done on the project.

Automation tips

Automate as much as possible.

- A frequent mistake: tabulate the data and get the min/max/mean values of a variable from that tabulation and hard code it for use later on.
    - Instead, consider tabulating the data and storing the min/max/mean values of the variable you want to iterate through in a local variable.
    - Then, whenever the data changes, you do not need to change these values, they will automatically update.
- Consider using Makefile: a list of shell commands containing rules that dictate what files to execute.

Automation tips

Automate as much as possible.

- A frequent mistake: tabulate the data and get the min/max/mean values of a variable from that tabulation and hard code it for use later on.
  - Instead, consider tabulating the data and storing the min/max/mean values of the variable you want to iterate through in a local variable.
  - Then, whenever the data changes, you do not need to change these values, they will automatically update.
- Consider using Makefile: a list of shell commands containing rules that dictate what files to execute.

Why is this useful? Automation saves future-you time, reduces errors, and makes it easy to replicate results.

An aside: macro variables

Most programming languages have a version of "macro" variables – these are objects defined as independent of the data (i.e., they are not data set variables).

An aside: macro variables

Most programming languages have a version of “macro” variables – these are objects defined as independent of the data (i.e., they are not data set variables).

Two types of macro variables:

- **global variables**: defined outside of any function, and can be accessed from anywhere
- **local variables**: defined within a specific function, and can only be accessed within that function

An aside: macro variables

Most programming languages have a version of "macro" variables – these are objects defined as independent of the data (i.e., they are not data set variables).

Two types of macro variables:

- **global variables**: defined outside of any function, and can be accessed from anywhere
- **local variables**: defined within a specific function, and can only be accessed within that function

Generally it is not a good idea to use globals – it is difficult to keep track of which functions have access to it and update it.

An aside: macro variables

Most programming languages have a version of “macro” variables – these are objects defined as independent of the data (i.e., they are not data set variables).

Two types of macro variables:

- **global variables**: defined outside of any function, and can be accessed from anywhere
- **local variables**: defined within a specific function, and can only be accessed within that function

Generally it is not a good idea to use globals – it is difficult to keep track of which functions have access to it and update it.

Be careful how these variables are passed on to functions – some programming languages pass them on by value, and others by reference.

An aside: macro variables

Most programming languages have a version of "macro" variables – these are objects defined as independent of the data (i.e., they are not data set variables).

Two types of macro variables:

- **global variables**: defined outside of any function, and can be accessed from anywhere
- **local variables**: defined within a specific function, and can only be accessed within that function

Generally it is not a good idea to use globals – it is difficult to keep track of which functions have access to it and update it.

Be careful how these variables are passed on to functions – some programming languages pass them on by value, and others by reference.

Why is this useful? Locals are very useful for defining fixed parameters (e.g., a CRRA of 3), doing text substitution, or using them for automating file input/output.

Automation tips

Automate file input and output.

- This is where maintaining a consistent directory structure and file naming convention comes useful.
- With a consistent directory structure and file name conventions, can easily define the project's directory at the top of the code and re-use code across projects by only changing the directory name.
- Generally, the base file path is one of the few things that is "allowed" to be hard coded in your script.

Automation tips

Automate file input and output.

- This is where maintaining a consistent directory structure and file naming convention comes useful.
- With a consistent directory structure and file name conventions, can easily define the project's directory at the top of the code and re-use code across projects by only changing the directory name.
- Generally, the base file path is one of the few things that is "allowed" to be hard coded in your script.
- ```
  local dir "insert directory name here"
  local outcomes "insert list of outcomes here"

  for each outcome in local `outcomes' {
     graph_outcome()
     save "dir/histogram_`outcome'.pdf"

  }
  ```
- With a consistent folder structure, you can refer to each of the subdirectories of your project without having to re-type the entire directory path again.
  ```

Automation tips

Automate file input and output.

- This is where maintaining a consistent directory structure and file naming convention comes useful.
- With a consistent directory structure and file name conventions, can easily define the project's directory at the top of the code and re-use code across projects by only changing the directory name.
- Generally, the base file path is one of the few things that is "allowed" to be hard coded in your script.
- ```
  local dir "insert directory name here"
  local outcomes "insert list of outcomes here"

  for each outcome in local `outcomes' {
     graph_outcome()
     save "dir/histogram_`outcome'.pdf"

  }
  ```
- With a consistent folder structure, you can refer to each of the subdirectories of your project without having to re-type the entire directory path again.

Why is this useful? Avoids manual file input/output, allows you to re-use code across projects by only changing the directory, avoids the need for re-inputting/outputting data when the data changes.
  ```

Outline

1. Organization
2. Data types, structures, and automation
3. Abstraction
4. Debugging and testing
5. Documentation
6. Efficiency
7. Miscellaneous
8. References and further resources

With great power...

Suppose you perform the same repetitive set of commands in multiple sections of one script, or across many scripts, but you just change the inputs to this set of commands.

Many programming languages allow you to write your own **functions and modularize code**.

What does modularization/abstraction mean? Break up code based on its use, save it as an "abstracted" piece of code in a separate script, and call it from many different scripts while only changing the inputs and outputs.

- If you foresee that you will be copy-pasting a piece of code a lot across different scripts but changing a fixed set of parameters, it's time to invest some of your time into abstraction!

… comes great responsibility

Abstraction is very powerful. It is clean, less error-prone, and allows for re-use.

Not everything should be abstracted. **The key theme is that things should be separated based on function.** Random pieces of code should not belong in a function that has a specific role and/or encapsulates a certain behavior.

... comes great responsibility

Abstraction is very powerful. It is clean, less error-prone, and allows for re-use.

Not everything should be abstracted. **The key theme is that things should be separated based on function.** Random pieces of code should not belong in a function that has a specific role and/or encapsulates a certain behavior.

Save your functions in separate scripts and name them well!

... comes great responsibility

Abstraction is very powerful. It is clean, less error-prone, and allows for re-use.

Not everything should be abstracted. **The key theme is that things should be separated based on function.** Random pieces of code should not belong in a function that has a specific role and/or encapsulates a certain behavior.

Save your functions in separate scripts and name them well!

Why is this useful? Abstraction is one of the most powerful concepts of good computer programming. Because it describes behavior, it allows for re-using of code across many different projects and applications. It saves time, and increases readability of code.

The perfect example of the power of abstraction is that I have been using the same function I wrote to output summary statistics from Stata into Excel using the format that I like for the past 5 years for many problem sets and projects.

Outline

1. Organization
2. Data types, structures, and automation
3. Abstraction
4. Debugging and testing
5. Documentation
6. Efficiency
7. Miscellaneous
8. References and further resources

Hey, I wrote some code! Does it work?

There are two types of checks you should be performing:

- **Debugging**: The code is not compiling/not running
- **Testing**: The code runs, but is it accurate?

Hey, I wrote some code! Does it work?

There are two types of checks you should be performing:

- **Debugging**: The code is not compiling/not running
- **Testing**: The code runs, but is it accurate?

Programming languages offer a lot of help with debugging – most programming languages provide very informative prompts and error messages when code does not compile or run. Read those warnings carefully!

Test often

Testing is trickier, since it depends on the code's function. The key questions to ask while testing are:

- What is the code supposed to be doing?
- Is it doing what it is supposed to be doing?
- What are potential cases that might break the code?

Test often

Testing is trickier, since it depends on the code's function. The key questions to ask while testing are:

- What is the code supposed to be doing?
- Is it doing what it is supposed to be doing?
- What are potential cases that might break the code?

Ask these questions often, and verify the answers.

Test modularly

In addition to testing often, you should **test modularly**.

What does this mean? Test code piece by piece, ensuring that each piece works correctly before moving on to the next.

- This becomes crucial when dealing with abstraction – generally good practice to ensure a function works correctly on a simple example, before checking that the code that calls that function works.
- Modular testing is made easy by splitting up your code into multiple scripts based on function.

General workflow for testing:

1. Focus on a piece of code
2. Is it doing what it is supposed to be doing?
3. Try to break it (e.g., pass in wrong type of input, or try edge cases)

Test modularly

In addition to testing often, you should **test modularly**.

What does this mean? Test code piece by piece, ensuring that each piece works correctly before moving on to the next.

- This becomes crucial when dealing with abstraction – generally good practice to ensure a function works correctly on a simple example, before checking that the code that calls that function works.
- Modular testing is made easy by splitting up your code into multiple scripts based on function.

General workflow for testing:

1. Focus on a piece of code
2. Is it doing what it is supposed to be doing?
3. Try to break it (e.g., pass in wrong type of input, or try edge cases)

Why is this useful? Testing code all at once takes longer and makes it easy to miss errors/mistakes. Testing it modularly allows you to verify accuracy of code piece by piece and narrow down the areas where there could be a mistake.

Other testing tips

- Do not test in the interpreter, test in a script (and keep those scripts).
- Check values of key variables and data sets.
- Use the built-in debugger: adding break-points is very helpful for checking values of key variables/data sets while the code runs.
- With large data, you might not want to test the code on the first pass with the entire data set. Many languages allow you to run a piece of code on a sub-sample of 100 observations. Make use of this feature to save some time when testing!
- If you have code that runs slowly, print notes to yourself to delineate what pieces of code have finished running. That way, if the code stops due to an error, you'll know where the error was.
- **Don't trust your future self**: instead of adding comments in the code that say "check for XYZ when updated data arrives" or "be careful: data must be in numeric format", add your own assertions and error checks.
- **Read your results! Think through them!** You'll be surprised how many special cases and errors you can catch that way.

Outline

1 Organization
2 Data types, structures, and automation
3 Abstraction
4 Debugging and testing
5 Documentation
6 Efficiency
7 Miscellaneous
8 References and further resources

Documentation tips

Find a good documentation strategy that works for you and is easy for future you and others to understand.

Most people document in-code, which means that they add comments documenting variables and functions where appropriate. This is the easiest to maintain, but it requires you to remember to update the comments when you change something.

Documentation tips

Find a good documentation strategy that works for you and is easy for future you and others to understand.

Most people document in-code, which means that they add comments documenting variables and functions where appropriate. This is the easiest to maintain, but it requires you to remember to update the comments when you change something.

Documentation varies based on individual preferences, but general tips include:

- Document unique identifiers (keys) and where parameters/data came from.
- Document decisions made, and **why** those decisions were made.
- Document inputs to your functions.
- I personally like to save data use agreements (DUAs) and dates when raw data was saved/downloaded in the 'Documentation' folder.
- Only write documentation that you know you will update.
- Write code to be self documenting:
  - If variable names are descriptive, then there is no need to document what each variable means.
- Use software (e.g., Stata and R Markdown).

Outline

1. Organization
2. Data types, structures, and automation
3. Abstraction
4. Debugging and testing
5. Documentation
6. Efficiency
7. Miscellaneous
8. References and further resources

One rule of fight club

Only improve efficiency after you have ensured accuracy.

Previously mentioned tips

- Automate and abstract!
- Test first passes of a code on a subset of the data.
- Print notes to yourself when running code while testing to benchmark the execution of the code.
- Use assertions and error checks.

Other tips for saving time and speeding up code

- Do not run things in the graphical user interface – learn how to run code from the terminal.
- Use flags in your master script to turn on/off scripts to run.
- **Parallelize**: makes your computer executes many operations simultaneously (e.g., use multiple cores of your processor, or use multiple processors on different machines - clusters).
  - When working with small data sets, parallelizing won't save you much time – better optimize your sequential code.
  - Very useful to know how to parallelize if you are working with large data sets.
- **Vectorize**: subset of parallelization, but does not require separate cores/clusters. Involves using data structures that are conducive to applying a procedure to multiple items .
  - Know your data structures and know when vectors/matrices and matrix algebra is faster.
  - Matrix algebra can in some instances replace loops and provide efficiency.

Outline

1. Organization
2. Data types, structures, and automation
3. Abstraction
4. Debugging and testing
5. Documentation
6. Efficiency
7. Miscellaneous
8. References and further resources

Command line basics (Linux/Mac)

- `pwd` prints working directory
- `cd "insert path here"` changes directory to specified one
  - `cd ..` goes up one directory
  - `cd ~` goes to root directory
- `ls` lists all files in current directory
- `ls -a` lists all files, including hidden files
- `mkdir "name or file path"` creates the specified directory
  - Can type `.` to denote the current directory instead of typing up the entire path
- `rmdir "name or file path"` deletes the specified directory
- `cp "origin path" "destination path"` copies origin file/directory to destination
- `mv "origin path" " destination path"` moves the origin file/directory to the destination path
- `top` provides a list of running processes, along with their process ID (PID)
- `kill "insert PID here"` stops the process with the specified PID

Regular expression basics

Regular expressions represent a way to denote patterns in variable, value, file, even directory names.

Why is this useful? Makes searching for values/files that satisfy a particular pattern easy.

Regular expression basics

- `.` denotes any character
  - e.g., `a.t` matches art, ant, aft, a2t, artsy...
  - e.g., `b..l` matches ball, bawl, bowl, b65l, bawling, bowling,...
- `[]` denotes a range of characters
  - e.g., `l[oe]t` matches let, lot, lots, letting,... but not lit, latitude,...
  - e.g., `l[1-8]t` matches l1t, l2t, ..., l8t,... – can use it with different ranges, e.g., 1-49, a-t etc.
- ^ denotes "not"
  - e.g., `l[^oe]t` matches lit, but not let or lot
- `*` denotes something occurring zero or more times, applied to the character directly in front of it
  - e.g., `let*` matches let, letting, lets,...
- + denotes something occurring one or more times, applied to the character directly in front of it
  - e.g., `let+` matches letting, lets but not let

General Principles

The two main principles for coding and managing data are:

1. Make things easier for your future self
2. Don't trust your future self

Specifically:

- Be consistent in formatting, style, organization, and naming
- Make the computer do the work
- Reduce copy-pasting and repetition
- Test often and test modularly
- Document often but "minimally"
- Increase efficiency

Big picture advice

The following big picture advice is **always** applicable[2]:

- Know the basics (data types, structures, automation, abstraction)
- Turn bugs into test cases.
- Divide (by function) and conquer.
- Investing time to write a function for a set of commands you use often pays off in the long run.
- Most things you want to program have already been programmed and are available online:
  - Even complicated concepts have pseudo code available that you can adapt.
- Learn the vocabulary of your programming language:
  - The answers to your questions are probably online, and you will find them easily if you know what to search for.

[2] Many thanks to Alex Measure for some of this great advice!

Outline

1. Organization
2. Data types, structures, and automation
3. Abstraction
4. Debugging and testing
5. Documentation
6. Efficiency
7. Miscellaneous
8. References and further resources

References and other helpful resources

- Gentzkow, M. and Shapiro, J.
  Code and Data for the Social Sciences: A Practitioner's Guide
- Wilson, Gregory V. et al. (2014). "Best Practices for Scientific Computing"
- Hans-Martin von Gaudecker's course:
  Effective programming practices for economists
  Specific to Python, but contains a lot of general programming tips
- MIT Open Courseware course:
  Introduction to Computer Science and Programming in Python
  Specific to Python, but is a great intro to computer science concepts.
- Grant McDermott's data science course:
  Environmental Economics and Data Science.
  Specific to R, but has wonderful general tips for programming.
- Intro to parallel computing in R: Quick Intro to Parallel Computing in R.
- Intro to Regular Expressions
- Jesús Fernández-Villaverde's course: Computational Methods
  Starts the second half of this semester at Harvard!

Thank you!