Learning to Use the Shell

Moving Towards Text Based Computing

Programming Practices for Economics Research

Department of Economics, Univeristy of Zurich

Winter 2019

Learning Objectives

At the end of the session you will:

1. Be able to provide text based instructions to your computer to achieve certain tasks
2. Understand the structure of your computer a bit better
3. Know where to look for help

What is the Shell?

A program that accepts text based instructions to produce one or more outputs

What Computers do

At a high level, computers do four things:

- run programs
- store data
- communicate with each other
- interact with us

The Shell Runs Programs

The shell is a **comand line interface (CLI)**

- offers a way to interact with the computer via text
- is a program like any other but it's main job is to run other programs
- It works in a **read-evaluate-print loop (REPL)**.
- When you type a command and press `Return`,
  1. The shell **reads** your command
  2. The shell **evaluates** what it means and executes it
  3. The shell **prints** the output of the command

Some Definitions

Don't be thrown off by terminology:

- The "shell", "terminal", "tty", "command prompt", etc.
- These are essentially different names for the same thing.
- They are all referring to a *command line interface* (CLI).

There are many shell variants, but we're going to focus on **Bash**

- i.e. **B**ourne **a**gain **sh**ell.
- Default shell on Linux and MacOS.
- Windows users should have Cygwin which emulates the behaviour of a Bash shell

Why bother with the shell?

- Power
  - Both for executing commands and for fixing problems. There are some things you just can't do in an IDE or GUI.
- Reproducibility
  - Scripting is reproducible, while clicking is not.

Why bother with the shell?

- Interacting with servers and super computers
  - The shell is often the only game in town for high performance computing. We'll get to this later in the course.
- Automating workflow and analysis pipelines
  - Easily track and reproduce an entire project (e.g. use a Makefile to combine multiple programs, scripts, etc.)

Getting Started

Opening the Shell

Open up your Bash shell.

- Linux
- Mac
- Windows

First Look

You should see something like:

```
username@hostname:~$
```

This is shell-speak for: "Who am I and where am I?"

- `username` denotes a specific user (one of potentially many on this computer).
- `@hostname` denotes the name of the computer or server.
- `:~` denotes the directory path (where `~` signifies the user's home directory).
- `$` denotes the start of the command prompt.

Syntax

```
$ command -option(s) (or --longoption(s)) arguments
```

- a `whitespace` on the command line is an argument separator,
- a `-` starts options,
- a `--` starts longoptions.
- . . . but it's in the programers freedom to violate this standard

Basic Bash

- The dollar sign stands for a prompt waiting for input

```
$
```

- Type `whoami` and press Enter to return name of current user

```
$ whoami
```

Basic Bash

- When type `whoami`:
  1. Shell finds the program
  2. The program is run
  3. The output of the program is shown
  4. A new prompt is displayed, indicating that it's ready for new commands

Files and Directories

Before we get started...

- The part of the Operating System that handles files and directories is called the filesystem
- We differentiate between files which hold information and directories (or folders) which hold files
- A handful of commands are used frequently to interact with these structures.

Directory structure

- To understand what our home directory is, let's look at the directory structure
- Organized as a tree with the root directory / at the very top
- Everything else is contained in it
- / refers to the leading slash
  - in /Users/me (Mac and Linux)
  - /cygdrive (Windows with cygwin)

Directory structure

Mac and Linux:

- Underneath `/Users` the data of user accounts on the machine is stored
  - E.g. `/Users/someusername`
- If we see `/Users/me`, we are inside `/Users`.
  - Similarly, `/Users` resides in the root `/`

Windows:

- Underneath `/cygdrive` you find the drives of your system (i.e, C, D, etc.)

Listing Files in a Directory with `ls`

- list directory contents

```
$ ls [directory ...]
```

- some important options:
  - `-F` (for flag); distinguish directories ('/'), executables ('*'), symbolic links, etc.
  - `-a` (for all); include directory entries whose names begin with a dot (i.e., .git)
  - `-l` (for long); prints the output in the long format
  - `-h` (for human readable): prints filesize in KB, MB, GB, TB instead of #Bytes
  - `-d` (for directories): show directories only

How can I change my working directory?

- to change your working directory

```
$ cd [directory]
```

- some shortcuts:
  - change to the current directory: `$ cd .`
  - change to the parent directory: `$ cd ..`
  - change to the home directory: `$ cd ~` or `cd`
  - change to previous directory: `$ cd -`
  - tab completion (press TAB once, twice)

A Little More on Those Shortcuts

These should work in combination with any command

- `.` : the current directory
- `..` : the parent directory
  - the one containing this one
- `~` : your home directory
- `-` : the previous directory

Navigation & Whitespace

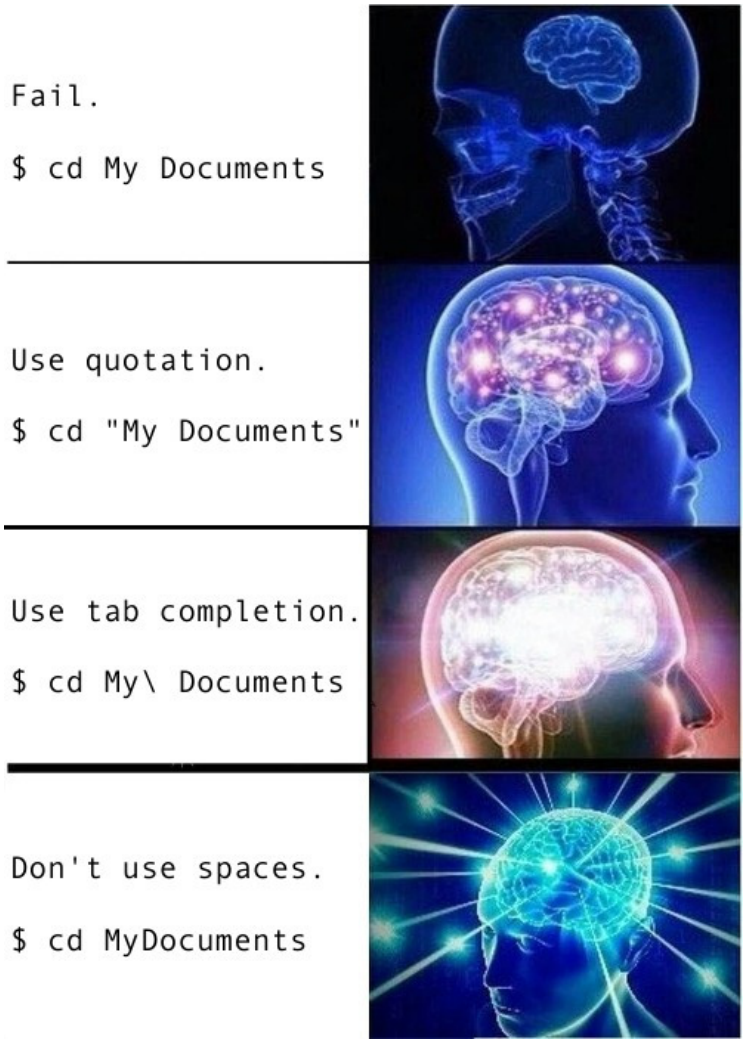

Figure 1

Exercise: relative path resolution

Using the filesystem diagram below, if `pwd` displays `/Users/thing`, what is the output of `ls -F ../backup`?

1. `../backup: No such file or directory`
2. `2012-12-01 2013-01-08 2013-01-27`
3. `2012-12-01/ 2013-01-08/ 2013-01-27/`
4. `original/ pnas_final/ pnas_sub/`

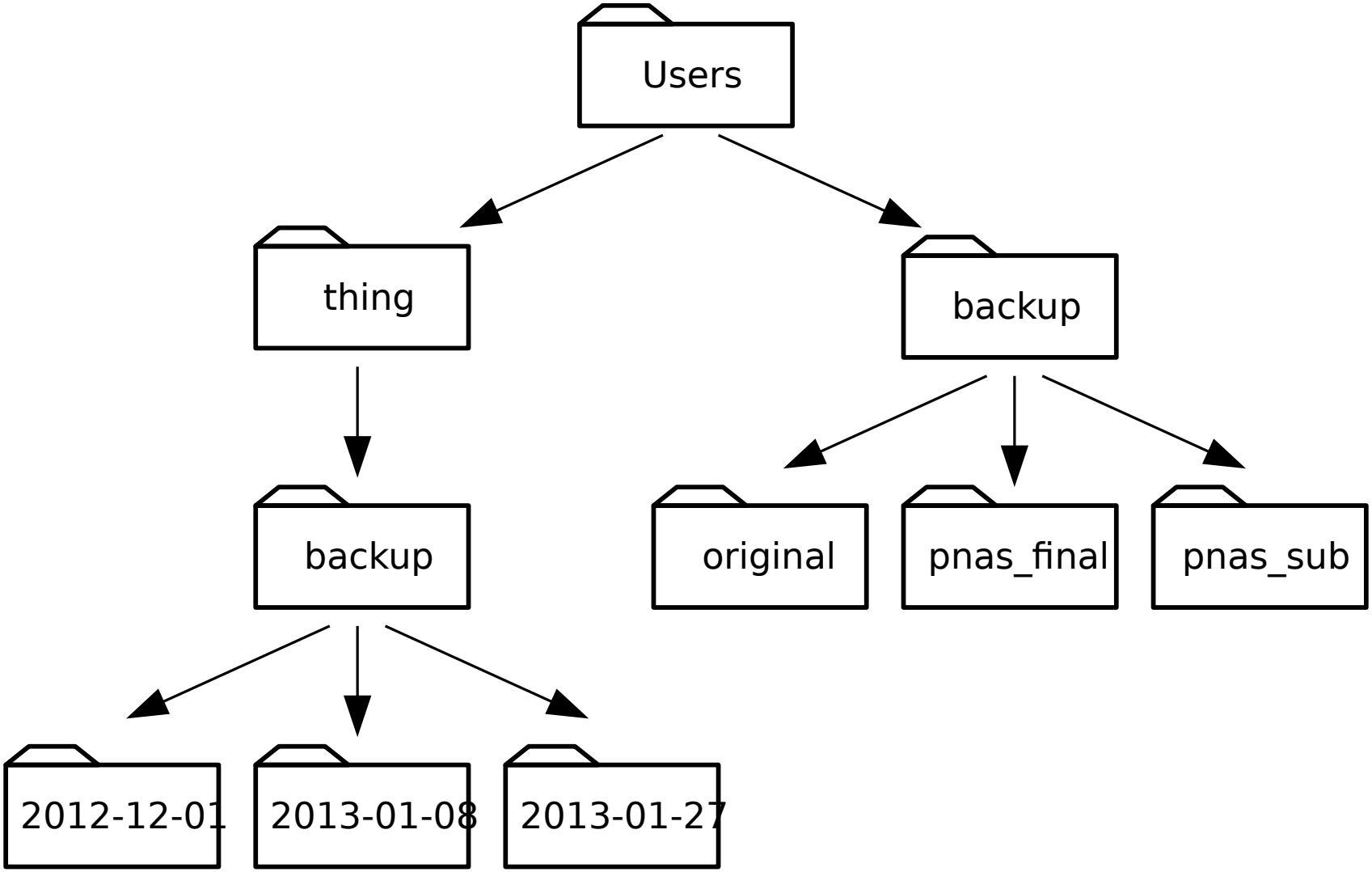

Figure 2: Example Filesystem

Viewing Content of a File

- View the file in the shell

```
$ less [filename]
```

- to navigate in less:
  - space: jump a page
  - b: jump a page back
  - /: search and highlight string in file/manualpage
  - q: quit
- Print out the file into the shell

```
$ cat [filename]
$ tail [filename]
$ head [filename]
$ more [filename] # less is more more
```

When Unsure - Search For Help

Getting help

- `$ whatis [command]`; display a brief description of a command
- `$ apropos [string]`; search the whatis database
- `$ man [executable]`; most executables provide a piece of documentation, called the manual page. Don't google a command, rtfm
- `$ help [builtins]`; help facility for shell builtins
- `$ [executable] --help`; option that displays a description of the command's supported syntax and options

Creating and Moving Files

Create a New File

```
$ touch [filename(s)]
```

Aside on Naming Conventions

- Files are typically named `something.extension`,
  - This is just a convention:
  - Two-part names help us (and programs) tell different kinds of files apart.
- The second part of such a name is called the filename extension,
  - Indicates what type of data the file holds
    - `.txt` signals a plain text file
    - `.pdf` indicates a PDF document,
    - `.png` is a PNG image, etc.
- Files contain bytes
  - it's up to us and our programs interpret those bytes according to the rules
    - Filename extensions typically specify a default behaviour

Create a New Directory

```
$ mkdir [directory]
```

or

```
$ mkdir -p [directory]
```

Remove a file

```
$ rm [filename(s)]
```

- there is **no undelete**
- important options
  - `-i` (for interactive); request confirmation before removing
  - `-v` (for verbose); show files which are being removed
  - `-r` (for recursive); required for directories; attempt to remove the file hierarchy rooted in each file argument

Remove an Empty Directory

```
$ rmdir [directory]
```

Exercise: Creating and Destroying

Perform the following tasks:

1. Create a new directory called `my_data`
2. Create 100 empty files `[number].dat` inside `my_data`
3. Create the subdirectory `new_data/2019-02` inside `my_data`
4. Create 20 empty files `[number].dat` inside `new_data/2019-02`
5. Remove the directory 2019-02 and all files

Copy file(s) to directory

```
$ cp [source file(s) ...] [target file | target directory]
```

- important options:
  - `-i` ; ask for permission before overwriting
  - `-r` ; required for directories
  - `-u` (for update); copy files that don't exist or are modified than in the existing directory
  - `-v` ; display messages

Rename and moving files and directories

```
$ mv [filename ...] [target file | target directory]
```

- important options
  - `-i` ; ask for permission before overwriting
  - `-v` ; display messages

Exercise

You are expecting a bunch of new data files that will need to be added to `my_data`. Like the current contents, the are numbered numerically. Perform two steps:

1. Create a back up of the data in a directory called `my_backup`
2. Move the data to a subdirectory `my_data/2018`

Wildcards

- Working with shell commands becomes powerful when you work with wildcards
- Wildcards are special characters that help you to rapidly specify groups of filenames
  - by matching one or more characters
- Four important wildcards are:
  - any character: `*`
  - any single character: `?`
  - any character that is a member of the set characters: `[characters]`
  - any character that is *not* a member of the set characters: `[!characters]`

Wildcards Examples

type	results
`*`	all files
`g*`	any file beginning with `g`
`b*.txt`	Any file beginning with `b` followed by any characters and ending with `.txt`
`Data???`	Any file beginning with `Data` followed by exactly three characters
`[abc]*`	Any file beginning with either `a`, `b`, or `c`

Exercise: Copying Structure, without files

You're starting a new experiment, and would like to duplicate the file structure from your previous experiment without the data files so you can add new data.

Assume that the file structure is in a folder called '2016-05-18-data', which contains a data folder that in turn contains folders named `raw` and `processed` that contain data files. The goal is to copy the file structure of the `2016-05-18-data` folder into a folder called `2016-05-20-data` and remove the data files from the directory you just created.

Which of the following set of commands would achieve this objective? What would the other commands do?

Exercise: Copying Structure, without files (cont.)

Option One:

```
$ cp -r 2016-05-18-data/ 2016-05-20-data/
$ rm 2016-05-20-data/raw/*
$ rm 2016-05-20-data/processed/*
```

Option 2:

```
$ rm 2016-05-20-data/raw/*
$ rm 2016-05-20-data/processed/*
$ cp -r 2016-05-18-data/ 2016-5-20-data/
```

Option 3:

```
$ cp -r 2016-05-18-data/ 2016-05-20-data/
$ rm -r -i 2016-05-20-data/
```

Redirections, Pipes, and Filters

I/O redirections

- Most programs read your input, execute it, and print output
- We call the input facility *standard input*
  - By default is your keyboard
- Our programs send their results to a special file called *standard output*
  - By default is print to the screen and not saved into the hard disk
- I/O redirection allows us to redefine where standard output goes.
  - Typically two choices
    1. Into a new file (or overwrite)
    2. Appended to an existing file

Redirection to a new file

- Redirect the output to a file instead of to the screen

```
$ ls -l [directory] > [filename]
```

- Overwrites `[filename]` in the process

Redirection by appending to a file

- Redirect the output to a file by appending instead of overwriting

```
$ ls -l [directory] >> [filename]
```

Exercise

Consider our file `data-shell/data/animals.txt`. After these commands:

```
$ head -n 3 animals.txt > animalsUpd.txt
$ tail -n 2 animals.txt >> animalsUpd.txt
```

What text is conatined in `animalsUpd.txt`?

Example: Read all tables into one file

```
$ cat table0* > table.txt
```

Pipelines

- So far: redirecting to file
- Can also redirect to be (the first) argument of next command
  - called *piping*
  - uses the `pipe operator`, `|`
- The general structure is

```
$ command arguments | command
```

- Pipes allow you to do complex data manipulations in one line, the pipeline

Piping Examples

- Example 1:

```
$ ls -l Data | less
```

- Example 2:

```
$ history | grep cp
```

Filters

- Filters take input, change it somehow, and then output it
  - Change is normally by reducing content to a summary form
- Some useful filters are the following:
  - `$ sort`
  - `$ uniq`
  - `$ wc`
  - `$ head` and `$ tail`

sort

- Sort lines of text files and writes to standard output; it does not change the file

```
$ sort [filename]
```

- some options:
  - `-f` (for fold); fold lower case to upper case characters
  - `-n` (for numerical); compare according to string numerical value
  - `-r` ; reverse the result of comparisons

uniq

- Report or filter out repeated lines in a file

```
$ uniq [input file] [ouput file]
```

- often used with `sort`
- useful option
  - `-d` (for duplicates); print list of duplicates

Piping with Filters - Example

Explain what the the following code does:

```
$ cat table*.dat  | sort -n | uniq | less
```

wc

- Count number of words, lines, characters, and bytes count

```
$ wc [file]
```

- can restrict count to certain fields using
  - `-w`: words
  - `-l`: lines
  - `-m`: characters

Pipes and Filters Example 2:

Explain what the the following code does:

```
$ cat table*.dat  | sort -n | uniq | wc -l > lines.txt
```

Exercise: Which pipe?

The file `animals.txt` contains 586 lines of data formatted as follows:

```
2012-11-05,deer
2012-11-05,rabbit
2012-11-05,raccoon
2012-11-06,rabbit
...
```

Assuming your current directory is `data-shell/data/`, what command would you use to produce a table that shows the total count of each type of animal in the file?

1. `grep {deer, rabbit, raccoon, deer, fox, bear} animals.txt | wc -l`
2. `sort animals.txt | uniq -c`
3. `sort -t, -k2,2 animals.txt | uniq -c`
4. `cut -d, -f 2 animals.txt | uniq -c`
5. `cut -d, -f 2 animals.txt | sort | uniq -c`
6. `cut -d, -f 2 animals.txt | sort | uniq -c | wc -l`

Finding Files and File Content

Finding Files

```
$ find [path] [expression]
```

- helpful versions:
  - `$ find . -type d`; find directories in current working directory
  - `$ find . -type f`; find files in current working directory
  - `$ find . -maxdepth 1 -type f`; restrict the depth of search to current level
  - `$ find . -mindepth 2 -type f`; find all files that are two or more levels below
  - `$ find . -name *.txt`; find all txt files

Finding Lines Within a File

```
$ grep [pattern] [file ...]
```

- example: print lines containing “beta”:

```
$ history | grep find
```

- Options include:
  - `-w` word; restrict matches to lines containing the word on its own (i.e., if beta, not beta1)
  - `-i` insensitive; makes search case-insensitive
  - `-n` number; number the lines that match
  - `-v` invert; print the lines that do not match
  - with "" phrase;
- check `man grep`

Regular expressions

- `grep` becomes powerful when combined with *regular expressions*
- Regex are used to identify *regular strings*;
  - specific strings such as phone numbers or email addresses . . .
  - or anything you know how to specify

Regular Strings?

- What is a regular string?
  - It's any string that can be generated by a series of linear rules,
- Examples:
  1. Write the letter "a" at least once.
  2. Contains the letter "b" exactly five times.
  3. Starts with a certain pattern.
  4. Ends with a certain pattern.
  5. Contains certain group of characters
- Can chain these rules together

Example: Valid Email addresses

- Rule 1: The first part of an email address contains at least one of the following: uppercase letters, lowercase letters, the numbers 0-9, periods (.), plus signs (+), or underscores (_).
- Rule 2: After this, the email address contains the @ symbol.
- Rule 3: The email address then must contain at least one uppercase or lowercase letter.
- Rule 4: This is followed by a period (.).
- Rule 5: Finally, the email address ends with *com*, *org*, *edu*, or *net*
  - in reality, there are many possible top-level domains

Solution:

```
[A-Za-z0-9\._+]+@[A-Za-z]+\.(com|org|edu|net)
```

Commonly used regular expressions

Symbols	Meaning	Example	Ex Matches
*	Matches the preceding character, subexpression, or bracketed character, 0 or more times	a*b*	aaaaaaaa, aaabbbbb, bbbbbb
+	Matches the preceding character, subexpression, or bracketed character, 1 or more times	a+b+	aaaaaaaab, aaabbbbb, abbbbbb
[]	Matches any character within the brackets (i.e., "Pick any one of these things")	[A-Z]*	APPLE, CAPITALS, QWERTY

Commonly used regular expressions

Symbols	Meaning	Example	Ex Matches
`()`	A grouped subexpression (these are evaluated first, in the "order of operations" of regular expressions)	`(a*b)*`	`aaabaab`, `abaaab`, `ababaaaaab`
`{m, n}`	Matches the preceding character, subexpression, or bracketed character between m and n times (inclusive)	`a{2,3}b{2,3}`	`aabbb`, `aaabbb`, `aabb`

Commonly used regular expressions

Symbols	Meaning	Example	Ex Matches
`[^]`	Matches any single character that is not in the brackets	`[^A-Z]*`	`apple`, `lowercase`, `qwerty`
`|`	Matches any character, string of characters, or subexpression, separated by the "|" (a vertical bar, or "pipe," not a capital "i")	`b(a|i|e)d`	`bad`, `bid`, `bed`
`.`	Matches any single character (including symbols, numbers, a space, etc.)	`b.d`	`bad`, `bzd`, `b$d`, `b d`

Commonly used regular expressions

Symbols	Meaning	Example	Ex Matches
^	Indicates that a character or subexpression occurs at the beginning of a string	`^a`	`apple, asdf, a`
\	An escape character (this allows you to use "special" characters as their literal meaning)	`\. | \\`	`. | \`

Commonly used regular expressions

Symbols	Meaning	Example	Ex Matches
`$`	Often used at the end of a regular expression, it means "match this up to the end of the string." Without it, every regular expression has a defacto `".*"` at the end of it, accepting strings where only the first part of the string matches.	`[A-Z]*[a-z]*$`	`ABCabc`, `zzzyx`, `Bob`

Commonly used regular expressions

Symb	Meaning	Example	Ex Matches
`?!`	"Does not contain." This pairing of symbols, immediately preceding a character (or regular expression), indicates that that character should not be found in that specific place in the larger string. If trying to eliminate a character entirely, use in conjunction with a ^ and $ at either end.	`^((?![A-Z]).)*$`	`no-caps-here`, `$ymb0ls a4e f!ne`

Acknowledgements

- This course is designed after and borrows a lot from:
  - Effective Programming Practices for Economists, a course by Hans-Martin von Gaudecker
  - Software Carpentry's Unix Shell Lesson
  - `Data Science for Economists`, a course by Grant McDermott
- The course material from above sources is made available under a Creative Commons Attribution License, as is this courses material.

Programming Practices Team

Programming Practices for Economics Research 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 2019 Foundations edition, conducted by

* Lachlan Deer
* Julian Langer