Lecture 02

Shell and Julia

Ivan Rudik
AEM 7130

Software and stuff

Necessary things to do:

- Install VSCode with these extensions: `Julia, Git Graph, Project Manager`
- Windows users: Install Windows Subsystem for Linux and a Unix distribution or use https://repl.it
- Install Julia with these Julia packages: `Expectations, Distributions, LinearAlgebra, BenchmarkTools`
    - Or use homebrew/other package manager

Visual Studio Code

VSCode

- Extensions
- Terminal/command line
- Julia (interactive)
- Project manager
- Git

The shell

Why learn the shell?

What is the shell?

The shell is the interface for interacting with your operating system, typically we are referring to the command line interface (terminal, command prompt, bash, etc)

Why learn the shell?

What is the shell?

The shell is the interface for interacting with your operating system, typically we are referring to the command line interface (terminal, command prompt, bash, etc)

A lot of what you can do in the shell can be done in Julia itself, why bother with it?

Why learn the shell?

Not everything can be done directily in your usual programming language

Why learn the shell?

Not everything can be done directily in your usual programming language

Command line is fast, powerful, and relatively easy to use, especially with modern shells like zsh or fish

Why learn the shell?

Not everything can be done directily in your usual programming language

Command line is fast, powerful, and relatively easy to use, especially with modern shells like zsh or fish

Writing shell scripts is reproducible and fast, unlike clicking buttons on a GUI

Why learn the shell?

Not everything can be done directily in your usual programming language

Command line is fast, powerful, and relatively easy to use, especially with modern shells like zsh or fish

Writing shell scripts is reproducible and fast, unlike clicking buttons on a GUI

If you want to use servers or any high performance computing you are likely to need to use shell

Why learn the shell?

You can automate your entire research pipeline with shell scripts (e.g. write something that calls multiple languages to execute your code then compiles your latex for the paper)

Why learn the shell?

You can automate your entire research pipeline with shell scripts (e.g. write something that calls multiple languages to execute your code then compiles your latex for the paper)

It really gets to the fundamentals of interacting with a computer (loops, tab-completions, saving scripts, etc)

Why learn the shell?

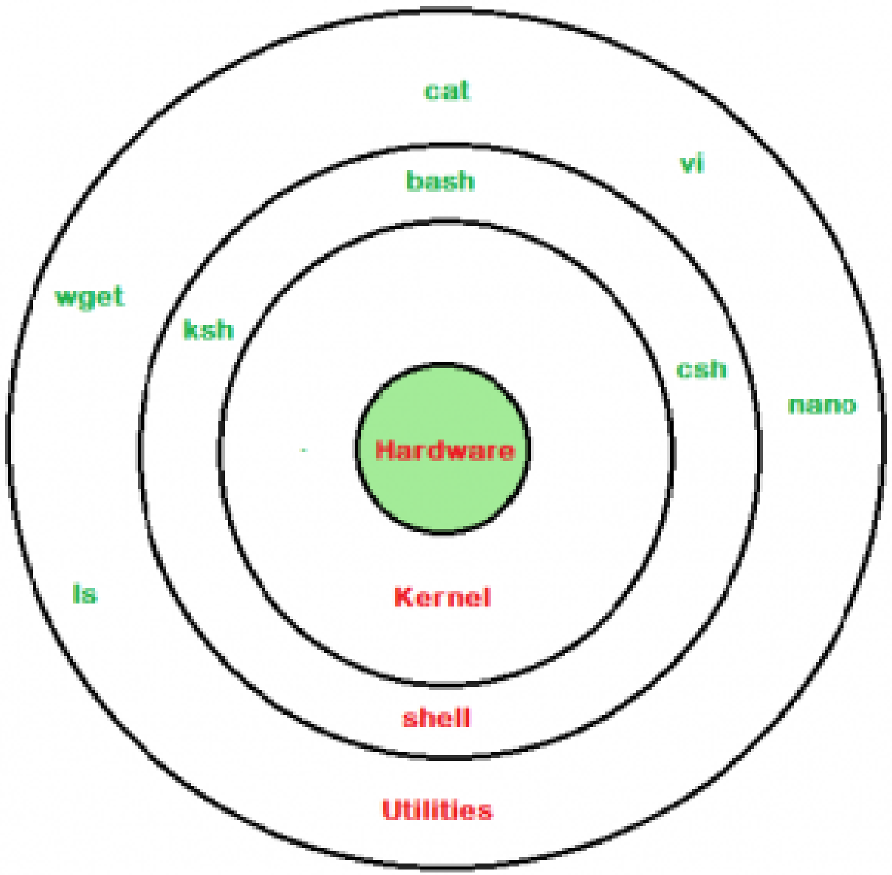
You can automate your entire research pipeline with shell scripts (e.g. write something that calls multiple languages to execute your code then compiles your latex for the paper)

It really gets to the fundamentals of interacting with a computer (loops, tab-completions, saving scripts, etc)

It gets you understanding how to write code in terms of functions which will be important for any programming you do in scripting languages like Julia, R, MATLAB, or Stata

What is the shell?

The shell is basically just a program where you can type in commands to interact with the **kernel** and hardware



What is the shell?

The most common one is **bash**, Bourne again shell, because it comes default on Linux and old Macs

What is the shell?

The most common one is **bash**, Bourne again shell, because it comes default on Linux and old Macs

I use **fish**, friendly interactive shell, because it comes default with a lot of nice features, all the commands still work identically to bash

Shell basics

When you open up the shell you should see a prompt, usually starting with `$` (don't type this)

```
$
```

Shell basics

We can type in one **command**, `ls` which lists the contents of your current directory

Shell basics

We can type in one **command**, `ls` which lists the contents of your current directory

```
$ ls
```

```
## 02-coding.Rmd
## 02-coding.html
## 02-coding_files
## figures
## my-css.css
## sandbox
```

My current directory is the one for this set of slides

Shell basics

Commands come with potential **options** or **flags** that modify how they act

```
$ ls
```

```
## 02-coding.Rmd
## 02-coding.html
## 02-coding_files
## figures
## my-css.css
## sandbox
```

```
$ ls -l # long form command
```

```
## total 368
## -rw-r--r--   1 ir229  CORNELL\Domain Users   44780 Jan 31 15:42 02-coding.Rmd
## -rw-r--r--@  1 ir229  CORNELL\Domain Users  125586 Jan 31 15:44 02-coding.html
## drwxr-xr-x   4 ir229  CORNELL\Domain Users     128 Jan 31 15:44 02-coding_files
## drwxr-xr-x   8 ir229  CORNELL\Domain Users     256 Jul 12  2022 figures
```

Shell basics

Options start with a dash and then a sequence of letters denoting which options you want

e.g. this lists files in long form `l`, sorted descending by size (`S`), with sizes in a human-readable format (`h`)

```
$ ls -lSh
```

```
## total 368
## -rw-r--r--@  1 ir229  CORNELL\Domain Users   123K Jan 31 15:44 02-coding.html
## -rw-r--r--   1 ir229  CORNELL\Domain Users    44K Jan 31 15:42 02-coding.Rmd
## -rw-r--r--@  1 ir229  CORNELL\Domain Users   8.2K Jan 22 17:44 my-css.css
## drwxr-xr-x  10 ir229  CORNELL\Domain Users   320B Jan 31 15:40 sandbox
## drwxr-xr-x   8 ir229  CORNELL\Domain Users   256B Jul 12  2022 figures
## drwxr-xr-x   4 ir229  CORNELL\Domain Users   128B Jan 31 15:44 02-coding_files
```

Shell basics

Finally commands have an **argument** that the command operates on

Shell basics

Finally commands have an **argument** that the command operates on

The previous `ls` calls were operating on the current directory,
but we could use it on any directory we want

Shell basics

Finally commands have an **argument** that the command operates on

The previous `ls` calls were operating on the current directory, but we could use it on any directory we want

```
$ ls -lSh ~/Desktop/git
```

```
## total 0
## drwxr-xr-x  73 ir229  CORNELL\Domain Users   2.3K Jan 29 09:05 climate-trade
## drwxr-xr-x  26 ir229  CORNELL\Domain Users   832B Mar 22  2022 aem2850
## drwxr-xr-x@ 26 ir229  CORNELL\Domain Users   832B Jan 23 10:54 aem4510
## drwxr-xr-x  20 ir229  CORNELL\Domain Users   640B Nov 30 15:21 irudik.github.io
## drwxr-xr-x  19 ir229  CORNELL\Domain Users   608B Jan 29 14:18 hurricane-forecasts
## drwxr-xr-x  19 ir229  CORNELL\Domain Users   608B Nov 15 20:52 lead-education
## drwxr-xr-x  18 ir229  CORNELL\Domain Users   576B Dec 28 13:31 enviro-transport
## drwxr-xr-x@ 16 ir229  CORNELL\Domain Users   512B Jan 31 15:46 aem7130
## drwxr-xr-x  14 ir229  CORNELL\Domain Users   448B May 25  2022 biodiversity
## drwxr-xr-x  14 ir229  CORNELL\Domain Users   448B Jan 25 20:06 climate-networks
```

Shell basics

To see what commands and their options do, use the `man` (manual) command

`q` exits the man page, and spacebar lets you skip down by a page

```
$ man ls
```

```
## LS(1)                       General Commands Manual                      LS(1)
##
## NNAAMMEE
##      llss – list directory contents
##
## SSYYNNOOPPSSIISS
##      llss [--@@AABBCCFFGGHHIILLOOPPRRSSTTUUWWaabbccddeeffgghhiikkllmmnnooppqqrrssttuuvvwwxxyy11%%,,]
##         [--DD _f_o_r_m_a_t] [_f_i_l_e _._._.]
##
## DDEESSCCRRIIPPTTIIOONN
##      For each operand that names a _f_i_l_e of a type other than directory, llss
##      displays its name as well as any requested, associated information.  For
```

Shell basics

Pressing `h` within a man page brings up the help page for how to navigate them

`/terms_here` lets you search within the man page for particular terms

Use `n` and `shift+n` to move forward and backward between matches

Navigation

We already learned how to list the files in a particular directory, but we need a few other tools to navigate around our machine

Navigation

We already learned how to list the files in a particular directory, but we need a few other tools to navigate around our machine

We often will want to know our **current working directory** so we know where we are before we start running commands

Navigation

We already learned how to list the files in a particular directory, but we need a few other tools to navigate around our machine

We often will want to know our **current working directory** so we know where we are before we start running commands

We do this with `pwd`

Navigation

We already learned how to list the files in a particular directory, but we need a few other tools to navigate around our machine

We often will want to know our **current working directory** so we know where we are before we start running commands

We do this with `pwd`

```
$ pwd
```

```
## /Users/ir229/Desktop/git/aem7130/lecture-notes/02-coding
```

Navigation

Directories are organized in a hierarchical structure, at the top is the root directory, `/`

```
$ ls /
```

```
## Applications
## Library
## Network
## System
## Users
## Volumes
## bin
## cores
## dev
## etc
## home
## opt
## private
## sbin
```

Navigation

The root directory contains everything else

Navigation

The root directory contains everything else

Other directories are inside the root directory and come afterward in the file path separated by forward slashes `/`

```
$ ls -lSh /Users
```

```
## total 0
## drwxr-x---+ 68 ir229           CORNELL\Domain Users   2.1K Jan 29 09:05 ir229
## drwxr-x---+ 19 cals_oit        staff                  608B Feb  9  2022 cals_oit
## drwxr-x---+ 15 ag-jjp34-doc10  CORNELL\Domain Users   480B Feb  9  2022 ag-jjp34-doc10
## drwxrwxrwt  14 root            wheel                  448B Jan 17 17:21 Shared
```

Navigation

The root directory contains everything else

Other directories are inside the root directory and come afterward in the file path separated by forward slashes `/`

```
$ ls -lSh /Users/ir229
```

```
## total 0
## drwx------@ 453 ir229  CORNELL\Domain Users    14K Jan 31 15:30 Downloads
## drwx------@  93 ir229  CORNELL\Domain Users   2.9K May 20  2022 Library
## drwx------@  43 ir229  CORNELL\Domain Users   1.3K Jan 18 14:47 Dropbox
## drwxr-xr-x   33 ir229  CORNELL\Domain Users   1.0K Dec 12 14:52 languageserver-library
## drwxr-xr-x   18 ir229  CORNELL\Domain Users   576B Jan 17 17:21 Zotero
## drwx------@  16 ir229  CORNELL\Domain Users   512B Sep  8 12:39 Box-Box-Backup
## drwx------@   6 ir229  CORNELL\Domain Users   192B Jan 10 08:11 Desktop
## drwx------+   5 ir229  CORNELL\Domain Users   160B Oct  5 12:03 Documents
## drwx------    5 ir229  CORNELL\Domain Users   160B May 10  2022 Movies
## drwx------+   5 ir229  CORNELL\Domain Users   160B May 10  2022 Music
```

Navigation

Next we need to be able to change directories, we can do this with `cd` (change directory)

Navigation

Next we need to be able to change directories, we can do this with `cd` (change directory)

```
$ cd /Users/ir229/Desktop/git
$ pwd
```

```
## /Users/ir229/Desktop/git
```

Navigation

Next we need to be able to change directories, we can do this with `cd` (change directory)

```
$ cd /Users/ir229/Desktop/git
$ pwd
```

```
## /Users/ir229/Desktop/git
```

When navigating, it is often easier and more reproducible to use **relative paths**

Navigation

Next we need to be able to change directories, we can do this with `cd` (change directory)

```
$ cd /Users/ir229/Desktop/git
$ pwd
```

```
## /Users/ir229/Desktop/git
```

When navigating, it is often easier and more reproducible to use **relative paths**

This is when arguments are relative to your current working directory, instead of using absolute paths (e.g. /Users/ir229/Desktop/git)

Navigation

There's a few expressions that make this possible

- `~` is your **home directory**
- `.` is your current directory
- `..` is the parent directory
- `-` is the previous directory you were in

Navigation

```
$ cd ~ # move to home directory (will vary computer-to-computer)
$ pwd
$ cd - # move to previous directory (lecture notes 2 directory)
$ cd .. # move to parent directory (general lecture notes directory)
$ pwd
$ cd . # move to current directory (nothing changes)
$ pwd
```

```
## /Users/ir229
## /Users/ir229/Desktop/git/aem7130/lecture-notes/02-coding
## /Users/ir229/Desktop/git/aem7130/lecture-notes
## /Users/ir229/Desktop/git/aem7130/lecture-notes
```

Navigation

```
$ cd ~ # move to home directory (will vary computer-to-computer)
$ pwd
$ cd - # move to previous directory (lecture notes 2 directory)
$ cd .. # move to parent directory (general lecture notes directory)
$ pwd
$ cd . # move to current directory (nothing changes)
$ pwd
```

```
## /Users/ir229
## /Users/ir229/Desktop/git/aem7130/lecture-notes/02-coding
## /Users/ir229/Desktop/git/aem7130/lecture-notes
## /Users/ir229/Desktop/git/aem7130/lecture-notes
```

You can see `.` and `..` in your current directory when using `ls` with the `a` flag

```
$ ls -a
```

```
## .
```

Navigation

This makes navigation much easier

If we wanted to move from the current directory to the parent directory for this year's course we can just do

Navigation

This makes navigation much easier

If we wanted to move from the current directory to the parent directory for this year's course we can just do

```
$ pwd
$ cd ../..
$ pwd
```

```
## /Users/ir229/Desktop/git/aem7130/lecture-notes/02-coding
## /Users/ir229/Desktop/git/aem7130
```

Navigation

This makes navigation much easier

If we wanted to move from the current directory to the parent directory for this year's course we can just do

```
$ pwd
$ cd ../..
$ pwd
```

```
## /Users/ir229/Desktop/git/aem7130/lecture-notes/02-coding
## /Users/ir229/Desktop/git/aem7130
```

instead of

```
$ cd /Users/ir229/Desktop/git/aem7130
$ pwd
```

Navigation

Relative paths are very important for reproducible code

Navigation

Relative paths are very important for reproducible code

Your directory structure starting at root `/` will not be the same as someone else's

Navigation

Relative paths are very important for reproducible code

Your directory structure starting at root `/` will not be the same as someone else's

Using relative paths lets you circumvent this as long as the project's directory structure is consistent

Navigation

Relative paths are very important for reproducible code

Your directory structure starting at root `/` will not be the same as someone else's

Using relative paths lets you circumvent this as long as the project's directory structure is consistent

If you use Git or Dropbox it should be

Creating files and directories

We learned how to move around directories but how do we make them?

Creating files and directories

We learned how to move around directories but how do we make them?

We do so with `mkdir` (make directory)

Creating files and directories

We learned how to move around directories but how do we make them?

We do so with `mkdir` (make directory)

```
$ mkdir test_directory
$ ls
```

```
## 02-coding.Rmd
## 02-coding.html
## 02-coding_files
## figures
## my-css.css
## sandbox
## test_directory
```

Creating files and directories

We create blank files using `touch`

Creating files and directories

We create blank files using `touch`

```
$ touch test_directory/test.txt test_directory/test1.txt
$ ls test_directory
```

```
## test.txt
## test1.txt
```

`touch` is useful if you have a program that can't create a file itself but can edit them

Creating files and directories

If you have a Unix system pre-installed with nano you can use `nano` to create and edit the file

```
$ nano test_directory/test.txt
```

Creating files and directories

Here are some tips for naming files and directories

Creating files and directories

Here are some tips for naming files and directories

1. **DON'T USE SPACES**
    - Spaces are used to separate commands, you generally want to avoid them in names in favor of underscores or dashes
2. Use letters, numbers, underscores, periods, and dashes only

Creating files and directories

If you really, really, want to use spaces in names you'll have to do one of two things, enclose in quotes or backslash the space

Creating files and directories

If you really, really, want to use spaces in names you'll have to do one of two things, enclose in quotes or backslash the space

```
$ mkdir "123test directory"
$ ls
```

```
## 02-coding.Rmd
## 02-coding.html
## 02-coding_files
## 123test directory
## figures
## my-css.css
## sandbox
## test_directory
```

Creating files and directories

If you really, really, want to use spaces in names you'll have to do one of two things, enclose in quotes or backslash the space

```
$ mkdir 123test\ directory\ 2
$ ls
```

```
## 02-coding.Rmd
## 02-coding.html
## 02-coding_files
## 123test directory
## 123test directory 2
## figures
## my-css.css
## sandbox
## test_directory
```

```
## 02-coding.Rmd
## 02-coding.html
## 02-coding_files
```

Moving files and directories

We can move files and directories with `mv` (move)

Moving files and directories

We can move files and directories with `mv` (move)

```
$ mv test_directory/test.txt ..
$ ls ..
```

The first argument is the (relative) path of the file you want to move, the second argument is where you're moving it to

Moving files and directories

```
$ mv test_directory/test.txt ..
$ ls ..
```

```
## 01-intro
## 02-coding
## 03-git
## 04-optimization
## 05-dynamic-programming
## 06-function-approximation
## 07-dp-solution-methods
## 08-optimal-control
## 09-advanced-dp-methods
## 10-spatial-models
## archive
## figures
## make-pdf.R
## test.txt
```

We moved the test.txt file from `test_directory` to the parent directory

Moving files and directories

```
$ mv ../test.txt test_directory
$ ls test_directory
```

```
## test.txt
## test1.txt
```

Here we moved it from the parent directory back to `test_directory`

Moving files and directories

```
$ mv ../test.txt test_directory
$ ls test_directory
```

```
## test.txt
## test1.txt
```

Here we moved it from the parent directory back to `test_directory`

Note that `mv` will overwrite any file with the move, use the `-i` option to make it ask you for confirmation

Moving files and directories

`mv` can also be used to rename files by just moving them to the same directory

Moving files and directories

`mv` can also be used to rename files by just moving them to the same directory

```
$ mv test_directory/test.txt test_directory/test_new_name.txt
$ ls test_directory
```

```
## test1.txt
## test_new_name.txt
```

```
$ mv test_directory/test_new_name.txt test_directory/test.txt
$ ls test_directory
```

```
## test.txt
## test1.txt
```

Moving files and directories

Now that we've made the directory and file, how do we get rid of them? With

`rm`

Moving files and directories

Now that we've made the directory and file, how do we get rid of them? With `rm`

```
$ rm test_directory/test.txt
$ ls
```

```
## 02-coding.Rmd
## 02-coding.html
## 02-coding_files
## figures
## my-css.css
## sandbox
## test_directory
```

Moving files and directories

We remove directories with `rmdir`

Moving files and directories

We remove directories with `rmdir`

```
$ rmdir test_directory
$ ls
```

```
## rmdir: test_directory: Directory not empty
## 02-coding.Rmd
## 02-coding.html
## 02-coding_files
## figures
## my-css.css
## sandbox
## test_directory
```

Notice that if a directory isn't empty you can't delete it

Moving files and directories

To delete a non-empty directory, you need to use `rm` on the directory, but apply the recursive option `-r` to delete everything inside of it first

Moving files and directories

To delete a non-empty directory, you need to use `rm` on the directory, but apply the recursive option `-r` to delete everything inside of it first

```
$ rm -r test_directory
$ ls
```

```
## 02-coding.Rmd
## 02-coding.html
## 02-coding_files
## figures
## my-css.css
## sandbox
```

Sometimes you might want to add the force option `-f` so it doesn't ask you if you want to delete each file

Copying files and directories

To copy files and directories just use `cp`, it works similarly to `mv`

Copying files and directories

To copy files and directories just use `cp`, it works similarly to `mv`

```
$ mkdir test_directory
$ touch test_directory/test_copy.txt
$ cp test_directory/test_copy.txt .
$ ls test_directory
```

```
## test_copy.txt
```

Copying files and directories

```
$ mkdir test_directory
$ touch test_directory/test_copy.txt
$ cp test_directory/test_copy.txt .
$ ls test_directory
```

```
$ ls
```

```
## 02-coding.Rmd
## 02-coding.html
## 02-coding_files
## figures
## my-css.css
## sandbox
## test_copy.txt
## test_directory
```

Copying files and directories

You copy directories the same way, but if you want to copy the full file contents you need to apply the recursive option `-r`

Copying files and directories

You copy directories the same way, but if you want to copy the full file contents you need to apply the recursive option `-r`

```
$ cp -r test_directory ..
$ ls .. test_directory
```

```
## ..:
## 01-intro
## 02-coding
## 03-git
## 04-optimization
## 05-dynamic-programming
## 06-function-approximation
## 07-dp-solution-methods
## 08-optimal-control
## 09-advanced-dp-methods
## 10-spatial-models
## archive
## figures
```

Copying multiple files

How do we copy multiple files?

Let's make two directories for copying and a set of similar files

```
$ mkdir main_directory copy_directory
$ touch main_directory/file1.txt main_directory/file2.txt main_directory/file3.txt
$ ls main_directory
```

```
## file1.txt
## file2.txt
## file3.txt
```

Copying multiple files

To copy them we can just use `cp` as we did before

Copying multiple files

To copy them we can just use `cp` as we did before

```
$ cp main_directory/file1.txt main_directory/file2.txt main_directory/file3.txt copy_directory
$ ls copy_directory
```

```
## file1.txt
## file2.txt
## file3.txt
```

Copying multiple files

To copy them we can just use `cp` as we did before

```
$ cp main_directory/file1.txt main_directory/file2.txt main_directory/file3.txt copy_directory
$ ls copy_directory
```

```
## file1.txt
## file2.txt
## file3.txt
```

We remove them the same way with `rm`

Copying multiple files

To copy them we can just use `cp` as we did before

```
$ cp main_directory/file1.txt main_directory/file2.txt main_directory/file3.txt copy_directory
$ ls copy_directory
```

```
## file1.txt
## file2.txt
## file3.txt
```

We remove them the same way with `rm`

```
$ rm main_directory/file1.txt main_directory/file2.txt main_directory/file3.txt
```

Or we could use the `mv` rename trick into a new directory named `copy_directory`

Renaming multiple files

We can rename multiple files in an easier way using `rename` (`brew install rename` to install using Homebrew)

We can change all of our txt files to csvs, `-s` indicates that the first argument is to be the text we are changing, and the second argument is the text we are changing it to, the third argument is the location of the files we are renaming

Renaming multiple files

```
$ ls copy_directory
```

```
## file1.txt
## file2.txt
## file3.txt
```

```
$ rename -s .txt .csv copy_directory/*
$ ls copy_directory
```

```
## bash: rename: command not found
## file1.txt
## file2.txt
## file3.txt
```

We can change all of our txt files to csvs, `-s` indicates that the first argument is to be the text we are changing, and the second argument is the text we are changing it to, the third argument is the location of the files we are renaming

Accessing multiple files

We can access multiple things at once using **wildcards** `*`, which replaces zero to any number of characters in the expression

Accessing multiple files

We can access multiple things at once using **wildcards** `*`, which replaces zero to any number of characters in the expression

```
$ touch copy_directory/test1.txt copy_directory/test2.txt copy_directory/test3.txt copy_directory
$ ls copy_directory/* # return everything in copy_directory
```

```
## copy_directory/file1.txt
## copy_directory/file2.txt
## copy_directory/file3.txt
## copy_directory/test1.txt
## copy_directory/test123.txt
## copy_directory/test2.txt
## copy_directory/test3.txt
```

Word count

The shell really shines when you try to combine multiple commands into one

Lets play around with the `sandbox` directory and count the number of words in `animals.txt` using `wc`

Word count

```
$ ls sandbox
$ wc sandbox/animals.txt
```

```
## animals.txt
## classes
## hey_jude.txt
## lengths.txt
## lucy_in_the_sky.txt
## shell_script.sh
## trees.txt
##        0       7      33 sandbox/animals.txt
```

The first number is the number of lines, the second is the number of words, and the third is the number of characters

Word count

We can run this using the wildcard for all text files and also get the totals

```
$ wc sandbox/*.txt
```

```
##        0       7      33 sandbox/animals.txt
##       29     195     979 sandbox/hey_jude.txt
##        6      12     167 sandbox/lengths.txt
##       45     220    1191 sandbox/lucy_in_the_sky.txt
##        6      15     101 sandbox/trees.txt
##       86     449    2471 total
```

Redirecting

Now suppose we had 1 million files and wanted to find the one with the most words? Just printing to the screen doesn't work, we'd want to save the output and use it somewhere else, we can do that by **redirecting** with the greater than symbol `>`

Redirecting

Now suppose we had 1 million files and wanted to find the one with the most words? Just printing to the screen doesn't work, we'd want to save the output and use it somewhere else, we can do that by **redirecting** with the greater than symbol `>`

```
$ wc -w sandbox/*.txt > sandbox/lengths.txt
$ ls sandbox
```

```
## animals.txt
## classes
## hey_jude.txt
## lengths.txt
## lucy_in_the_sky.txt
## shell_script.sh
## trees.txt
```

Printing and cating

We can print the file to the screen using `cat` (print the full file) or `less` (one screenful)

Printing and cating

We can print the file to the screen using `cat` (print the full file) or `less` (one screenful)

```
$ cat sandbox/lengths.txt
```

```
##        7 sandbox/animals.txt
##      195 sandbox/hey_jude.txt
##        0 sandbox/lengths.txt
##      220 sandbox/lucy_in_the_sky.txt
##       15 sandbox/trees.txt
##      437 total
```

The `w` option made it so we only got the number of words, not characters or lines

Sorting

If we want to return the output sorted we can use `sort`

Sorting

If we want to return the output sorted we can use `sort`

```
$ sort -n sandbox/lengths.txt
```

```
##         0 sandbox/lengths.txt
##         7 sandbox/animals.txt
##        15 sandbox/trees.txt
##       195 sandbox/hey_jude.txt
##       220 sandbox/lucy_in_the_sky.txt
##       437 total
```

where the `n` option means to sort numerically

Sorting

We can look at only the first few lines using `head`, (`tail` gets the last lines)

Sorting

We can look at only the first few lines using `head`, (`tail` gets the last lines)

```
$ head -n 1 sandbox/lengths.txt
```

```
##        7 sandbox/animals.txt
```

Where the 1 means we only want the first line

Redirecting

`>` will always overwrite a file, we can use the double greater than symbol `>>` to append to a file

Redirecting

`>` will always overwrite a file, we can use the double greater than symbol `>>` to append to a file

Lets use `echo` for an example which prints text

Redirecting

`>` will always overwrite a file, we can use the double greater than symbol `>>` to append to a file

Lets use `echo` for an example which prints text

```
$ echo Hello world!
```

```
## Hello world!
```

```
$ echo \ walnut >> sandbox/trees.txt
$ cat sandbox/trees.txt
```

```
## maple pine birch oak beechnut palm fig redwood walnut walnut
##  walnut
##  walnut
##  walnut
##  walnut
```

Piping

We've learned a few options for manipulating text files, we can combine them in easy ways using **piping** (same idea as Julia's queryverse and R's tidyverse)

Piping

We've learned a few options for manipulating text files, we can combine them in easy ways using **piping** (same idea as Julia's queryverse and R's tidyverse)

Pipes `|` allow you sequentially write out commands that use the previous command's output as the next command's input

Piping

Suppose we wanted to find the file in a directory with the most number of characters, we could do this with

Piping

Suppose we wanted to find the file in a directory with the most number of characters, we could do this with

```
$ wc -m sandbox/* | sort -n | tail -n 2
```

```
## wc: sandbox/classes: read: Is a directory
##     1191 sandbox/lucy_in_the_sky.txt
##     2875 total
```

`lucy_in_the_sky.txt` is the longest in the sandbox directory

Piping

Look at the file `sandbox/hey_jude.txt`, how would we get the second verse?

Piping

Look at the file `sandbox/hey_jude.txt`, how would we get the second verse?

We can pipe a `head` and `tail` together:

Piping

Look at the file `sandbox/hey_jude.txt`, how would we get the second verse?

We can pipe a `head` and `tail` together:

```
$ head -n 9 sandbox/hey_jude.txt | tail -n 4
```

```
## Hey jude, don't be afraid.
## You were made to go out and get her.
## The minute you let her under your skin,
## Then you begin to make it better.
```

`head` grabs the first two verses (with the empty line inbetween),
`tail` grabs second verse

Looping example 1

What if we wanted the second verse of *multiple songs*?

Looping example 1

What if we wanted the second verse of *multiple songs*?

We can do that with a loop

Looping example 1

What if we wanted the second verse of *multiple songs*?

We can do that with a loop

```
$ for thing in list
$ do
$     operation_using $thing    # Indentation is good style
$ done
```

`$` preprends any variables, here the variables are the things we're looping over

Looping example 1

```
$ for song in sandbox/hey_jude.txt sandbox/lucy_in_the_sky.txt
$ do
$     head -n 9 $song | tail -n 4
$ done
```

```
## Hey jude, don't be afraid.
## You were made to go out and get her.
## The minute you let her under your skin,
## Then you begin to make it better.
## Cellophane flowers of yellow and green
## Towering over your head
## Look for the girl with the sun in her eyes
## And she's gone
```

Looping example 2

How about a more realistic one that is real world useful (taken from Grant Mcdermott)

Looping example 2

How about a more realistic one that is real world useful (taken from Grant Mcdermott)

Let's combine a bunch of csvs using the shell

This is particularly useful with many or large datasets, because when done through shell, you do not need to load them into memory

Looping example 2

How about a more realistic one that is real world useful (taken from Grant Mcdermott)

Let's combine a bunch of csvs using the shell

This is particularly useful with many or large datasets, because when done through shell, you do not need to load them into memory

The files are in `/sandbox/classes` and report a fake class schedule

Let's combine them into one

Looping example 2

First we need to make our class schedule file

Looping example 2

First we need to make our class schedule file

```
$ touch sandbox/classes/class_schedule.csv
```

Looping example 2

First we need to make our class schedule file

```
$ touch sandbox/classes/class_schedule.csv
```

Then we need to add each day's schedule to the file

Looping example 2

First we need to make our class schedule file

```
$ touch sandbox/classes/class_schedule.csv
```

Then we need to add each day's schedule to the file

```
$ for day in $(ls sandbox/classes/*day.csv)
$ do
$     cat $day >> sandbox/classes/class_schedule.csv
$ done
```

where we treat what `ls` returns as a variable since its the output of a command

Looping example 2

```
$ cat sandbox/classes/class_schedule.csv
```

```
## day,morning,afternoon,evening
## friday,nothing,workshop,nothing
## day,morning,afternoon,evening
## monday,micro,macro,metrics
## day,morning,afternoon,evening
## thursday,game theory,seminar,nothings
## day,morning,afternoon,evening
## tuesday,game theory,seminar,nothings
## day,morning,afternoon,evening
## wednesday,micro,macro,metrics
```

Looping example 2

```
$ cat sandbox/classes/class_schedule.csv
```

```
## day,morning,afternoon,evening
## friday,nothing,workshop,nothing
## day,morning,afternoon,evening
## monday,micro,macro,metrics
## day,morning,afternoon,evening
## thursday,game theory,seminar,nothings
## day,morning,afternoon,evening
## tuesday,game theory,seminar,nothings
## day,morning,afternoon,evening
## wednesday,micro,macro,metrics
```

Looks like it worked but we have the header every other line, how do we get rid of it?

Looping example 2

Hint: we only need the header once, and then we want the last line of the csv for each file

Looping example 2

Hint: we only need the header once, and then we want the last line of the csv for each file

First lets remove the old file

Looping example 2

Hint: we only need the header once, and then we want the last line of the csv for each file

First lets remove the old file

```
$ rm -f sandbox/classes/class_schedule.csv
```

Looping example 2

Hint: we only need the header once, and then we want the last line of the csv for each file

First lets remove the old file

```
$ rm -f sandbox/classes/class_schedule.csv
```

Next create the new file by grabbing the header from Monday

Looping example 2

Hint: we only need the header once, and then we want the last line of the csv for each file

First lets remove the old file

```
$ rm -f sandbox/classes/class_schedule.csv
```

Next create the new file by grabbing the header from Monday

```
$ head -1 sandbox/classes/monday.csv > sandbox/classes/class_schedule.csv
$ cat sandbox/classes/class_schedule.csv
```

```
## day,morning,afternoon,evening
```

Looping example 2

So we've got the file started, now we need to fill in the days using our looping skills

Looping example 2

So we've got the file started, now we need to fill in the days using our looping skills

We need to add each day's schedule to the file

Looping example 2

So we've got the file started, now we need to fill in the days using our looping skills

We need to add each day's schedule to the file

```
$ for day in $(ls sandbox/classes/*day.csv)
$ do
$     tail -1 $day | cat >> sandbox/classes/class_schedule.csv
$ done
$ cat sandbox/classes/class_schedule.csv
```

```
## day,morning,afternoon,evening
## friday,nothing,workshop,nothing
## monday,micro,macro,metrics
## thursday,game theory,seminar,nothings
## tuesday,game theory,seminar,nothings
## wednesday,micro,macro,metrics
```

Finding things

How can we find things within files?

Finding things

How can we find things within files?

We use the command `grep` (global/regular expression/print)

Finding things

How can we find things within files?

We use the command `grep` (global/regular expression/print)

`grep` finds and prints lines that match a certain pattern

For example, lets find the lines in Hey Jude that contain "make"

Finding things

How can we find things within files?

We use the command `grep` (global/regular expression/print)

`grep` finds and prints lines that match a certain pattern

For example, lets find the lines in Hey Jude that contain "make"

```
$ grep make sandbox/hey_jude.txt
```

```
## Hey jude, don't make it bad.
## Take a sad song and make it better.
## Then you can start to make it better.
## Then you begin to make it better.
## Then you can start to make it better.
## Hey jude, don't make it bad.
## Take a sad song and make it better.
```

Finding things

```
$ grep make sandbox/hey_jude.txt
```

Here `make` is the pattern we are searching for inside Hey Jude

Finding things

Now lets search Lucy in the Sky for "in"

Finding things

Now lets search Lucy in the Sky for "in"

```
$ grep in sandbox/lucy_in_the_sky.txt | head -5
```

```
## Picture yourself in a boat on a river
## With tangerine trees and marmalade skies
## Towering over your head
## Look for the girl with the sun in her eyes
## Lucy in the sky with diamonds
```

Finding things

Now lets search Lucy in the Sky for "in"

```
$ grep in sandbox/lucy_in_the_sky.txt | head -5
```

```
## Picture yourself in a boat on a river
## With tangerine trees and marmalade skies
## Towering over your head
## Look for the girl with the sun in her eyes
## Lucy in the sky with diamonds
```

This gave us words that contained "in" but weren't actually the word "in"

Finding things

We can restrict the search to words with the `w` option

Finding things

We can restrict the search to words with the `w` option

```
$ grep -w in sandbox/lucy_in_the_sky.txt | head -5
```

```
## Picture yourself in a boat on a river
## Look for the girl with the sun in her eyes
## Lucy in the sky with diamonds
## Lucy in the sky with diamonds
## Lucy in the sky with diamonds
```

Grepping

`grep`s real power comes from using **regular expressions**

These are complex expressions that allow you to search for very specific things

Grepping

`grep`s real power comes from using **regular expressions**

These are complex expressions that allow you to search for very specific things

For example, lets find lines with "a" as the second letter

Grepping

`grep`s real power comes from using **regular expressions**

These are complex expressions that allow you to search for very specific things

For example, lets find lines with "a" as the second letter

```
$ grep -E "^.a" sandbox/lucy_in_the_sky.txt | head -5
```

```
## Waiting to take you away
```

Finding things

`grep` shows up in most programming languages as well

You can imagine using it to do things like dynamically renaming a set of variables, dealing with weirdly reported FIPS codes, etc

Shell scripts

A nice thing about shell that's pretty underused by economists is putting the commands into scripts so we can re-use them

Shell scripts

A nice thing about shell that's pretty underused by economists is putting the commands into scripts so we can re-use them

```
$ # writing a shell script using echo is kind of silly
$ # but I want to show you what I'm doing on the slides
$ touch sandbox/shell_script.sh
$ echo echo "Hello World!" >> sandbox/shell_script.sh
$ cat sandbox/shell_script.sh
```

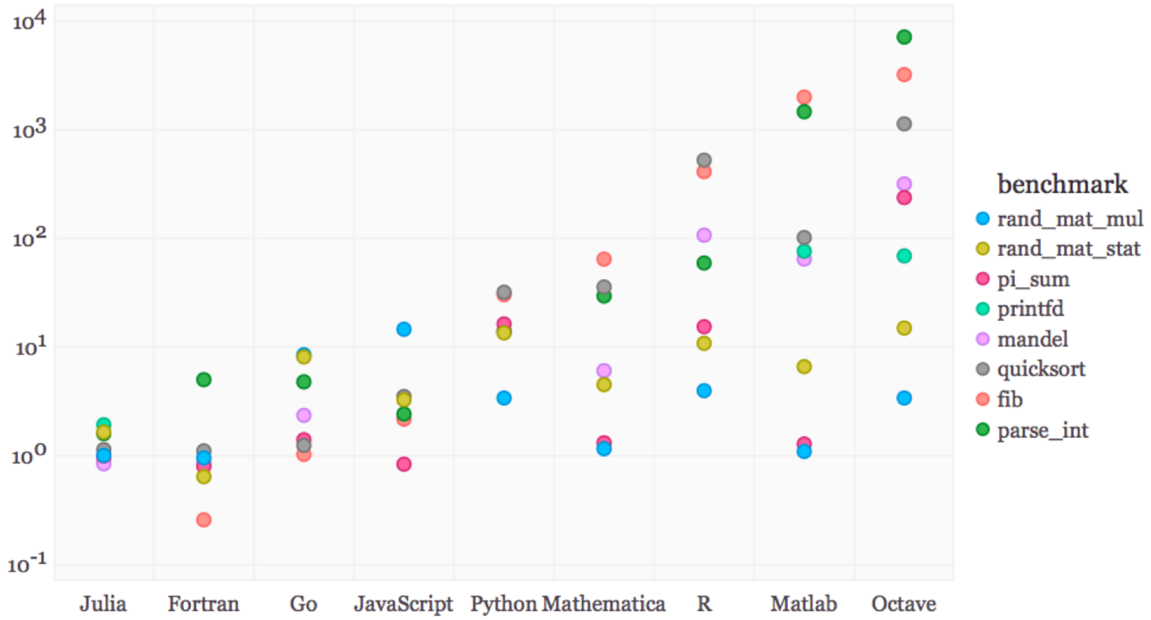
```
## echo Hello World!
## echo Hello World!
## echo Hello World!
## echo Hello World!
## echo Hello World!
## echo Hello World!
## echo Hello World!
## echo Hello World!
## echo Hello World!
## echo Hello World!
```

Julia

Why am I doing this to you?

Why are we using Julia?

1. It's a high-level language, much easier to use than C++, Fortran, etc
2. It delivers C++ and Fortran speed



Intro to programming

Programming $\equiv$ writing a set of instructions

1. There are hard and fast rules you can't break if you want it to work
2. There are elements of style (e.g. Strunk and White) that make for clearer and more efficient code

Intro to programming

If you will be doing computational work there are:

1. Language-independent coding basics you should know
    - Arrays are stored in memory in particular ways
2. Language-independent best practices you should use
    - Indent to convey program structure (or function in Python)
3. Language-dependent idiosyncracies that matter for function, speed, etc
    - Julia: type stability; R: vectorize

Intro to programming

Learning these early will:

1. Make coding a lot easier

Intro to programming

Learning these early will:

1. Make coding a lot easier
2. Reduce total programmer time

Intro to programming

Learning these early will:

1. Make coding a lot easier
2. Reduce total programmer time
3. Reduce total computer time

Intro to programming

Learning these early will:

1. Make coding a lot easier
2. Reduce total programmer time
3. Reduce total computer time
4. Make your code understandable by someone else or your future self

Intro to programming

Learning these early will:

1. Make coding a lot easier
2. Reduce total programmer time
3. Reduce total computer time
4. Make your code understandable by someone else or your future self
5. Make your code flexible

A broad view of programming

Your goal is to make a **program**

A program is made of different components and sub-components

A broad view of programming

Your goal is to make a **program**

A program is made of different components and sub-components

The most basic component is a **statement**, more commonly called a **line of code**

A broad view of programming

Here's pseudoprogram:

```
deck = ["4 of hearts", "King of clubs", "Ace of spades"]
shuffled_deck = shuffle(deck)
first_card = shuffled_deck[1]
println("The first drawn card was " * shuffled_deck ".")
```

This program is real simple:

1. Create a deck of cards

A broad view of programming

Here's pseudoprogram:

```
deck = ["4 of hearts", "King of clubs", "Ace of spades"]
shuffled_deck = shuffle(deck)
first_card = shuffled_deck[1]
println("The first drawn card was " * shuffled_deck ".")
```

This program is real simple:

1. Create a deck of cards
2. Shuffle the deck

A broad view of programming

Here's pseudoprogram:

```
deck = ["4 of hearts", "King of clubs", "Ace of spades"]
shuffled_deck = shuffle(deck)
first_card = shuffled_deck[1]
println("The first drawn card was " * shuffled_deck ".")
```

This program is real simple:

1. Create a deck of cards
2. Shuffle the deck
3. Draw the top card

A broad view of programming

Here's pseudoprogram:

```
deck = ["4 of hearts", "King of clubs", "Ace of spades"]
shuffled_deck = shuffle(deck)
first_card = shuffled_deck[1]
println("The first drawn card was " * shuffled_deck ".")
```

This program is real simple:

1. Create a deck of cards
2. Shuffle the deck
3. Draw the top card
4. Print it

A broad view of programming

```
deck = ["4 of hearts", "King of clubs", "Ace of spades"]
shuffled_deck = shuffle(deck)
first_card = shuffled_deck[1]
println("The first drawn card was " * shuffled_deck ".")
```

What are the parentheses and why are they different from square brackets?

How does shuffle work?

What's println?

It's important to know that a good program has understandable code

Julia specifics

We will discuss coding in the context of Julia but a lot of this ports to Python, MATLAB, etc

To do:

1. Types
2. Operators
3. Scope
4. Generic functions
5. Multiple dispatch

Types

All languages have some kind of **data types** like integers or arrays

Types

All languages have some kind of **data types** like integers or arrays

The first type you will often use is a boolean (`Bool`) variable that takes on a value of `true` or `false`:

```
x = true
```

```
## true
```

```
typeof(x)
```

```
## Bool
```

Types

We can save the boolean value of actual statements in variables this way:

```
@show y = 1 > 2
```

```
## y = 1 > 2 = false
```

```
## false
```

`@show` is a Julia macro for showing the operation

Numbers

Two other data types you will use frequently are integers

```
typeof(1)
```

```
## Int64
```

Numbers

Two other data types you will use frequently are integers

```
typeof(1)
```

```
## Int64
```

and floating point numbers

```
typeof(1.0)
```

```
## Float64
```

Numbers

Two other data types you will use frequently are integers

```
typeof(1)
```

```
## Int64
```

and floating point numbers

```
typeof(1.0)
```

```
## Float64
```

Recall from lecture 1 the 64 means 64 bits of storage for the number, which is probably the default on your machine

Numbers

You can always instantiate alternative floating point number types

```
converted_int = convert(Float32, 1.0);
typeof(converted_int)
```

```
## Float32
```

Numbers

Math works like you would expect:

```
a = 2; b = 1.0;
a * b
```

```
## 2.0
```

Numbers

Math works like you would expect:

```
a = 2; b = 1.0;
a * b
```

```
## 2.0
```

```
a^2
```

```
## 4
```

Numbers

```
2a - 4b
```

```
## 0.0
```

Numbers

```
2a - 4b
```

```
## 0.0
```

```
@show 4a + 3b^2
```

```
## 4a + 3 * b ^ 2 = 11.0
```

```
## 11.0
```

Numbers

```
2a - 4b
```

```
## 0.0
```

```
@show 4a + 3b^2
```

```
## 4a + 3 * b ^ 2 = 11.0
```

```
## 11.0
```

You dont need `*` inbetween numeric literals (numbers) and variables

Strings

Strings store sequences of characters

Strings

Strings store sequences of characters

You implement them with double quotations:

```
x = "Hello World!";
typeof(x)
```

```
## String
```

Strings

Strings store sequences of characters

You implement them with double quotations:

```
x = "Hello World!";
typeof(x)
```

```
## String
```

Note that `;` suppresses output for that line of code but is unnecessary in Julia

Strings

It's easy to work with strings, use `$` to interpolate a variable/expression

```
x = 10; y = 20; println("x + y =  $(x+y).")
```

```
## x + y =  30.
```

Strings

It's easy to work with strings, use `$` to interpolate a variable/expression

```
x = 10; y = 20; println("x + y =  $(x+y).")
```

```
## x + y =  30.
```

Use `*` to concatenate strings

```
a = "Aww"; b = "Yeah!!!"; println(a * " " * b)
```

```
## Aww Yeah!!!
```

Strings

It's easy to work with strings, use `$` to interpolate a variable/expression

```
x = 10; y = 20; println("x + y =  $(x+y).")
```

```
## x + y =  30.
```

Use `*` to concatenate strings

```
a = "Aww"; b = "Yeah!!!"; println(a * " " * b)
```

```
## Aww Yeah!!!
```

You probably won't use strings too often unless you're working with text data or printing output

Containers

Containers are types that store collections of data

Containers

Containers are types that store collections of data

The most basic container is the `Array` which is denoted by square brackets

Containers

Containers are types that store collections of data

The most basic container is the `Array` which is denoted by square brackets

```
a1 = [1 2; 3 4]; typeof(a1)
```

```
## Matrix{Int64} (alias for Array{Int64, 2})
```

Containers

Containers are types that store collections of data

The most basic container is the `Array` which is denoted by square brackets

```
a1 = [1 2; 3 4]; typeof(a1)
```

```
## Matrix{Int64} (alias for Array{Int64, 2})
```

Arrays are **mutable** which means you can change their values

Containers

Containers are types that store collections of data

The most basic container is the `Array` which is denoted by square brackets

```
a1 = [1 2; 3 4]; typeof(a1)
```

```
## Matrix{Int64} (alias for Array{Int64, 2})
```

Arrays are **mutable** which means you can change their values

```
a1[1,1] = 5; a1
```

```
## 2×2 Matrix{Int64}:
##  5  2
##  3  4
```

Containers

An alternative to the `Array` is the `Tuple` which is denoted by parentheses

Containers

An alternative to the `Array` is the `Tuple` which is denoted by parentheses

```
a2 = (1, 2, 3, 4); typeof(a2)
```

```
## NTuple{4, Int64}
```

`a2` is a `Tuple` of 4 `Int64`s, tuples have no dimension

Containers

Tuples are **immutable** which means you **can't** change their values

```
try
  a2[1,1] = 5;
catch
  println("Error, can't change value of a tuple.")
end
```

```
## Error, can't change value of a tuple.
```

Containers

Tuples don't need parentheses (but it's probably best practice for clarity)

```
a3 = 5, 6; typeof(a3)
```

```
## Tuple{Int64, Int64}
```

Containers

Tuples can be **unpacked** (see `NamedTuple` for an alternative and more efficient container)

Containers

Tuples can be **unpacked** (see `NamedTuple` for an alternative and more efficient container)

```
a3_x, a3_y = a3;
a3_x
```

```
## 5
```

```
a3_y
```

```
## 6
```

Containers

Tuples can be **unpacked** (see `NamedTuple` for an alternative and more efficient container)

```
a3_x, a3_y = a3;
a3_x
```

```
## 5
```

```
a3_y
```

```
## 6
```

This is basically how functions return output when you call them

Containers

A `Dictionary` is the last main container type, they are arrays but are indexed by keys (names) instead of numbers

Containers

A `Dictionary` is the last main container type, they are arrays but are indexed by keys (names) instead of numbers

```
d1 = Dict("class" => "AEM7130", "grade" => 97);
typeof(d1)
```

```
## Dict{String, Any}
```

Containers

A `Dictionary` is the last main container type, they are arrays but are indexed by keys (names) instead of numbers

```
d1 = Dict("class" => "AEM7130", "grade" => 97);
typeof(d1)
```

```
## Dict{String, Any}
```

`d1` is a dictionary where the key are strings and the values are any kind of type

Containers

Reference specific values you want in the dictionary by referencing the key

Containers

Reference specific values you want in the dictionary by referencing the key

```
d1["class"]
```

```
## "AEM7130"
```

```
d1["grade"]
```

```
## 97
```

Containers

If you just want all the keys or all the values you can use the base functions

Containers

If you just want all the keys or all the values you can use the base functions

```
keys_d1 = keys(d1)
```

```
## KeySet for a Dict{String, Any} with 2 entries. Keys:
##   "class"
##   "grade"
```

```
values_d1 = values(d1)
```

```
## ValueIterator for a Dict{String, Any} with 2 entries. Values:
##   "AEM7130"
##   97
```

Iterating

As in other languages we have loops at our disposal:

`for` loops iterate over containers

```
for count in 1:10
  random_number = rand()
  if random_number > 0.2
    println("We drew a $random_number.")
  end
end
```

```
## We drew a 0.2296328225210792.
## We drew a 0.4970731729200491 6.
## We drew a 0.856092069452262 6.
## We drew a 0.262402485876497 4.
## We drew a 0.665060001812365 8.
## We drew a 0.650341730067200 2.
## We drew a 0.4929268795537094.
```

Iterating

`while` loops iterate until a logical expression is false

```
while rand() > 0.5
  random_number = rand()
  if random_number > 0.2
    println("We drew a $random_number.")
  end
end
```

Iterating

An `Iterable` is something you can loop over, like arrays

Iterating

An `Iterable` is something you can loop over, like arrays

```
actions = ["codes well", "skips class"];
for action in actions
    println("Charlie $action")
end
```

```
## Charlie codes well
## Charlie skips class
```

Iterating

There's a type that's a subset of iterables, `Iterator`, that are particularly convenient

Iterating

There's a type that's a subset of iterables, `Iterator`, that are particularly convenient

These include things like the dictionary keys:

```
for key in keys(d1)
  println(d1[key])
end
```

```
## AEM7130
## 97
```

Iterating

Iterating on `Iterator`s is more memory efficient than iterating on arrays

Iterating

Iterating on `Iterator`s is more memory efficient than iterating on arrays

Here's a **very** simple example, the top function iterates on an `Array`, the bottom function iterates on an `Iterator`:

Iterating

Iterating on `Iterator`s is more memory efficient than iterating on arrays

Here's a **very** simple example, the top function iterates on an `Array`, the bottom function iterates on an `Iterator`:

```
function show_array_speed()
  m = 1
  for i = [1, 2, 3, 4, 5, 6]
    m = m*i
  end
end;

function show_iterator_speed()
  m = 1
  for i = 1:6
    m = m*i
  end
end;
```

Iterating

```
using BenchmarkTools
@btime show_array_speed()
```

```
##   17.827 ns (1 allocation: 112 bytes)
```

```
@btime show_iterator_speed()
```

```
##   2.125 ns (0 allocations: 0 bytes)
```

The `Iterator` approach is faster and allocates no memory

`@btime` is a macro from `BenchmarkTools` that shows you the elasped time and memory allocation

Neat looping

The nice thing about Julia vs MATLAB is your loops can be much neater since you don't need to index if you just want the container elements

Neat looping

The nice thing about Julia vs MATLAB is your loops can be much neater since you don't need to index if you just want the container elements

```
f(x) = x^2;
x_values = 0:20:100;
for x in x_values
  println(f(x))
end
```

```
## 0
## 400
## 1600
## 3600
## 6400
## 10000
```

Neat looping

The loop directly assigns the elements of `x_values` to `x` instead of having to do something clumsy like `x_values[i]`

Neat looping

The loop directly assigns the elements of `x_values` to `x` instead of having to do something clumsy like `x_values[i]`

`0:20:100` creates something called a `StepRange` (a type of `Iterator`) which starts at `0`, steps up by `20` and ends at `100`

Neat looping

You can also pull out an index and the element value by enumerating

```
f(x) = x^2;
x_values = 0:20:100;
for (index, x) in enumerate(x_values)
  println("f(x) at value $index is $(f(x)).")
end
```

```
## f(x) at value 1 is 0.
## f(x) at value 2 is 400.
## f(x) at value 3 is 1600.
## f(x) at value 4 is 3600.
## f(x) at value 5 is 6400.
## f(x) at value 6 is 10000.
```

`enumerate` basically assigns an index vector

Neat looping

There is also a lot of Python-esque functionality

Neat looping

There is also a lot of Python-esque functionality

For example: `zip` lets you loop over multiple different iterables at once

Neat looping

There is also a lot of Python-esque functionality

For example: `zip` lets you loop over multiple different iterables at once

```
last_name = ("Lincoln", "Bond", "Walras");
first_name = ("Abraham", "James", "Leon");

for (first_idx, last_idx) in zip(first_name, last_name)
  println("The name's $last_idx, $first_idx $last_idx.")
end
```

```
## The name's Lincoln, Abraham Lincoln.
## The name's Bond, James Bond.
## The name's Walras, Leon Walras.
```

Neat looping

Nested loops can also be made very neatly

Neat looping

Nested loops can also be made very neatly

```
for x in 1:3, y in 3:-1:1
  println(y-x)
end
```

```
## 2
## 1
## 0
## 1
## 0
## -1
## 0
## -1
## -2
```

Neat looping

Nested loops can also be made very neatly

```
for x in 1:3, y in 3:-1:1
  println(y-x)
end
```

```
## 2
## 1
## 0
## 1
## 0
## -1
## 0
## -1
## -2
```

The first loop is the inner loop, the second loop is the outer loop

Comprehensions: the neatest looping

Comprehensions are super nice ways to use iterables that make your code cleaner and more compact

Comprehensions: the neatest looping

Comprehensions are super nice ways to use iterables that make your code cleaner and more compact

```
squared = [y^2 for y in 1:2:11]
```

```
## 6-element Vector{Int64}:
##    1
##    9
##   25
##   49
##   81
##  121
```

Comprehensions: the neatest looping

Comprehensions are super nice ways to use iterables that make your code cleaner and more compact

```
squared = [y^2 for y in 1:2:11]
```

```
## 6-element Vector{Int64}:
##     1
##     9
##    25
##    49
##    81
##   121
```

This created a 1-dimension `Array` using one line

Comprehensions: the neatest looping

We can also use nested loops for comprehensions

Comprehensions: the neatest looping

We can also use nested loops for comprehensions

```
squared_2 = [(y+z)^2 for y in 1:2:11, z in 1:6]
```

```
## 6×6 Matrix{Int64}:
##    4    9   16   25   36   49
##   16   25   36   49   64   81
##   36   49   64   81  100  121
##   64   81  100  121  144  169
##  100  121  144  169  196  225
##  144  169  196  225  256  289
```

Comprehensions: the neatest looping

We can also use nested loops for comprehensions

```
squared_2 = [(y+z)^2 for y in 1:2:11, z in 1:6]
```

```
## 6×6 Matrix{Int64}:
##    4    9   16   25   36   49
##   16   25   36   49   64   81
##   36   49   64   81  100  121
##   64   81  100  121  144  169
##  100  121  144  169  196  225
##  144  169  196  225  256  289
```

This created a 2-dimensional `Array`

Comprehensions: the neatest looping

We can also use nested loops for comprehensions

```
squared_2 = [(y+z)^2 for y in 1:2:11, z in 1:6]
```

```
## 6×6 Matrix{Int64}:
##     4    9   16   25   36   49
##    16   25   36   49   64   81
##    36   49   64   81  100  121
##    64   81  100  121  144  169
##   100  121  144  169  196  225
##   144  169  196  225  256  289
```

This created a 2-dimensional `Array`

Use this (and the compact nested loop) sparingly since it's hard to follow

Dot syntax: broadcasting/vectorization

Vectorizing operations (e.g. applying it to a whole array or vector at once) is easy in Julia, just use dot syntax like you would in MATLAB, etc

Dot syntax: broadcasting/vectorization

Vectorizing operations (e.g. applying it to a whole array or vector at once) is easy in Julia, just use dot syntax like you would in MATLAB, etc

```
g(x) = x^2;
squared_2 = g.(1:2:11)
```

```
## 6-element Vector{Int64}:
##     1
##     9
##    25
##    49
##    81
##   121
```

Dot syntax: broadcasting/vectorization

Vectorizing operations (e.g. applying it to a whole array or vector at once) is easy in Julia, just use dot syntax like you would in MATLAB, etc

```
g(x) = x^2;
squared_2 = g.(1:2:11)
```

```
## 6-element Vector{Int64}:
##     1
##     9
##    25
##    49
##    81
##   121
```

This is actually called **broadcasting**

Dot syntax: broadcasting/vectorization

Vectorizing operations (e.g. applying it to a whole array or vector at once) is easy in Julia, just use dot syntax like you would in MATLAB, etc

```
g(x) = x^2;
squared_2 = g.(1:2:11)
```

```
## 6-element Vector{Int64}:
##     1
##     9
##    25
##    49
##    81
##   121
```

This is actually called **broadcasting**

When broadcasting, you might want to consider **pre-allocating** arrays

Dot syntax: broadcasting/vectorization

Vectorization creates *temporary allocations*, temporary arrays in the middle of the process that aren't actually needed for the final product

Julia can do broadcasting in a nicer, faster way by **fusing** operations together and avoiding these temporary allocations

Dot syntax: broadcasting/vectorization

Let's write two functions that do the same thing:

```
function show_vec_speed(x)
  out = [3x.^2 + 4x + 7x.^3 for i = 1:1]
end
function show_fuse_speed(x)
  out = @. [3x.^2 + 4x + 7x.^3 for i = 1:1]
end
```

The top one is vectorized for the operations, the `@.` in the bottom one vectorizes everything in one swoop: the function call, the operation, and the assignment to a variable

Dot syntax: broadcasting/vectorization

First, precompile the functions

```
x = rand(10^6);
@time show_vec_speed(x);
@time show_fuse_speed(x);
```

```
@time show_vec_speed(x)
```

```
##   0.004672 seconds (13 allocations: 45.777 MiB)
```

```
## 1-element Vector{Vector{Float64}}:
##  [12.874822850326872, 2.725670724402299, 2.8566366599700235, 11.811970611339124, 0.8959328634610417,
```

```
@time show_fuse_speed(x)
```

```
##   0.000925 seconds (3 allocations: 7.630 MiB)
```

Dot syntax: vectorization

Not pre-allocated:

```
h(y,z) = y^2 + sin(z);    # function to evaluate
y = 1:2:1e6+1;            # input y
z = rand(length(y));      # input z
```

Dot syntax

Here we are vectorizing the function call

```
# precompile h so first timer isn't picking up on compile time
h(1,2)
```

```
@time out_1 = h.(y,z)    # evaluate h.(y,z) and time
```

```
##   0.034353 seconds (246.22 k allocations: 16.154 MiB, 90.35% compilation time)
```

```
## 500001-element Vector{Float64}:
##    1.270990279629937
##    9.118939012142084
##   25.30930830725158
##   49.73109386642228
##   81.46268082472055
##  121.19774297963545
##  169.776117811416
##  225.12227681514867
```

Dot syntax: vectorization

Here we are vectorizing the function call and assignment

```
out_2 = similar(out_1)
```

```
@time out_2 .= h.(y,z)
```

```
##   0.014353 seconds (38.86 k allocations: 1.784 MiB, 77.51% compilation time)
```

```
## 500001-element Vector{Float64}:
##     1.270990279629937
##     9.118939012142084
##    25.30930830725158
##    49.73109386642228
##    81.46268082472055
##   121.19774297963545
##   169.776117811416
##   225.12227681514867
##   289.3627061432049
```

Dot syntax: vectorization

Here we are vectorizing the function call, assignment, and operations

```
out_3 = similar(out_1)
```

```
@time out_3 = @. h(y,z)
```

```
##   0.003161 seconds (4 allocations: 3.815 MiB)
```

```
## 500001-element Vector{Float64}:
##    1.270990279629937
##    9.118939012142084
##   25.30930830725158
##   49.73109386642228
##   81.46268082472055
##  121.19774297963545
##  169.776117811416
##  225.12227681514867
##  289.3627061432049
```

Logical operators work like you'd think

- `==` (equal equal) tests for equality

Logical operators work like you'd think

- `==` (equal equal) tests for equality

```
1 == 1
```

```
## true
```

Logical operators work like you'd think

- `==` (equal equal) tests for equality

```
1 == 1
```

```
## true
```

- `!=` (exclaimation point equal) tests for inequality

Logical operators work like you'd think

- `==` (equal equal) tests for equality

```
1 == 1
```

```
## true
```

- `!=` (exclamation point equal) tests for inequality

```
2 != 2
```

```
## false
```

Logical operators work like you'd think

- `==` (equal equal) tests for equality

```
1 == 1
```

```
## true
```

- `!=` (exclaimation point equal) tests for inequality

```
2 != 2
```

```
## false
```

- You can also test for approximate equality with ≈ (type `\approx<TAB>`)

Logical operators work like you'd think

- `==` (equal equal) tests for equality

```
1 == 1
```

```
## true
```

- `!=` (exclaimation point equal) tests for inequality

```
2 != 2
```

```
## false
```

- You can also test for approximate equality with ≈ (type `\approx<TAB>`)

```
1.00000001 ≈ 1
```

Scope

The **scope** of a variable name determines when it is valid to refer to it

Scope

The **scope** of a variable name determines when it is valid to refer to it

Scope can be a frustrating concept

Scope

The **scope** of a variable name determines when it is valid to refer to it

Scope can be a frustrating concept

If you want to dive into the details: the type of scoping in Julia is called **lexical scoping**

Scope

The **scope** of a variable name determines when it is valid to refer to it

Scope can be a frustrating concept

If you want to dive into the details: the type of scoping in Julia is called **lexical scoping**

Different scopes can have the same name, i.e. `saving_rate`, but be assigned to different variables

Scope

The scope of a variable name determines when it is valid to refer to it

Scope can be a frustrating concept

If you want to dive into the details: the type of scoping in Julia is called **lexical scoping**

Different scopes can have the same name, i.e. `saving_rate`, but be assigned to different variables

Let's walk through some simple examples to see how it works

Scope

First, functions have their own local scope

Scope

First, functions have their own local scope

```
ff(xx) = xx^2;
yy = 5;
ff(yy)
```

```
## 25
```

`xx` isn't bound to any values outside the function `ff`

This is pretty natural for those of you who have done any programming before

Scope

Locally scoped functions allow us to do things like:

```
xx = 10;
fff(xx) = xx^2;
fff(5)
```

```
## 25
```

Although `xx` was declared equal to 10, the function still evaluated at 5

Scope

Locally scoped functions allow us to do things like:

```
xx = 10;
fff(xx) = xx^2;
fff(5)
```

```
## 25
```

Although `xx` was declared equal to 10, the function still evaluated at 5

This is all kind of obvious so far

Scope

But, this type of scoping also has (initially) counterintuitive results like:

```
zz = 0;
for ii = 1:10
  zz = ii
end
println("zz = $zz")
```

```
## zz = 0
```

Scope

What happened?

Scope

What happened?

The `zz` outside the for loop has a different scope, the **global scope**, than the `zz` inside it

Scope

What happened?

The `zz` outside the for loop has a different scope, the **global scope**, than the `zz` inside it

The global scope is the outer most scope, outside all functions and loops

Scope

What happened?

The `zz` outside the for loop has a different scope, the **global scope**, than the `zz` inside it

The global scope is the outer most scope, outside all functions and loops

The `zz` inside the for loop has a scope **local** to the loop

Scope

What happened?

The `zz` outside the for loop has a different scope, the **global scope**, than the `zz` inside it

The global scope is the outer most scope, outside all functions and loops

The `zz` inside the for loop has a scope **local** to the loop

Since the outside `zz` has global scope the locally scoped variables in the loop can't change it

Scope

Generally you want to avoid global scope because it can cause conflicts, slowness, etc, but you can use `global` to force it if you want something to have global scope

```
zz = 0;
for ii = 1:10
  global zz
  zz = ii
end
println("zz = $zz")
```

```
## zz = 10
```

Scope

Local scope kicks in whenever you have a new block keyword (i.e. you indented something) except for `if`

Global variables inside a local scope are inherted for **reading**, not writing

```
x, y = 1, 2;
function foo()
  x = 2          # assignment introduces a new local
  return x + y # y refers to the global
end;
foo()
```

```
## 4
```

```
x
```

```
## 1
```

Scope

Important piece: nested functions can modify their parent scope's **local** variables

Scope

Important piece: nested functions can modify their parent scope's local variables

```
x, y = 1, 2; # set globals

function f_outer()
  x = 2                    # introduces a new local
  function f_inner()
    x = 10                 # modifies the parent's x
    return x + y           # y is global
  end
  return f_inner() + x # 12 + 10 (x is modified in call of f_inner())
end;
f_outer()
x, y                       # verify that global x and y are unchanged
```

Scope

```
function f_outer()
  x = 2                 # introduces a new local
  function f_inner()
    x = 10              # modifies the parent's x
    return x + y        # y is global
  end
  return f_inner() + x # 12 + 10 (x is modified in call of f_inner())
end;
f_outer()
```

```
## 22
```

```
x, y                    # verify that global x and y are unchanged
```

```
## (1, 2)
```

Scope

```
function f_outer()
  x = 2                   # introduces a new local
  function f_inner()
    x = 10                # modifies the parent's x
    return x + y          # y is global
  end
  return f_inner() + x # 12 + 10 (x is modified in call of f_inner())
end;
f_outer()
x, y                      # verify that global x and y are unchanged
```

Scope

```
function f_outer()
  x = 2                     # introduces a new local
  function f_inner()
    x = 10                  # modifies the parent's x
    return x + y            # y is global
  end
  return f_inner() + x # 12 + 10 (x is modified in call of f_inner())
end;
f_outer()
x, y                        # verify that global x and y are unchanged
```

If `f_inner` was not nested and was in the global scope we'd get `14` not `22`, this is also a way to handle the issue with loops editing variables not created in their local scope

Scope

We can fix looping issues with global scope by using a wrapper function that doesn't do anything but change the parent scope so it is not global

```
function wrapper()
  zzz = 0;
  for iii = 1:10
    zzz = iii
  end
  println("zzz = $zzz")
end
```

```
## wrapper (generic function with 1 method)
```

```
wrapper()
```

```
## zzz = 10
```

Closures

These inner functions we've been looking at are called **closures**

When a function `f` is parsed in Julia, it looks to see if any of the variables have been previously defined in the current scope

```
a = 0.2;
f(x) = a * x^2;     # refers to the `a` in the outer scope
```

```
## f (generic function with 1 method)
```

```
f(1)                # univariate function
```

```
## 0.2
```

Closures

```
function g(a)
    f(x) = a * x^2; # refers to the `a` passed in the function
    f(1);           # univariate function
end
```

```
## g (generic function with 1 method)
```

```
g(0.2)
```

```
## 0.2
```

Closures

```julia
function g(a)
    f(x) = a * x^2; # refers to the `a` passed in the function
    f(1);           # univariate function
end
```

```
## g (generic function with 1 method)
```

```julia
g(0.2)
```

```
## 0.2
```

In both of these examples `f` is a closure designed to **capture** a variable from an outer scope

Closures

Here's a complicated example that actually returns a closure (a function!) itself:

Closures

```
x = 0;
function toplevel(y)
  println("x = ", x, " is a global variable")
  println("y = ", y, " is a parameter")
  z = 2
  println("z = ", z, " is a local variable")

  function closure(v)
    println("v = ", v, " is a parameter")
    w = 3
    println("w = ", w, " is a local variable")
    println("x = ", x, " is a global variable")
    println("y = ", y, " is a closed variable (a parameter of the outer function)")
    println("z = ", z, " is a closed variable (a local of the outer function)")
  end;
  return closure
end;
```

What will be returned when we call these functions?

Closures

Here's a complicated example:

```
c_func = toplevel(10)
```

```
## x = 0 is a global variable
## y = 10 is a parameter
## z = 2 is a local variable
```

```
## (::var"#closure#230"{Int64, Int64}) (generic function with 1 method)
```

```
c_func(20)
```

```
## v = 20 is a parameter
## w = 3 is a local variable
## x = 0 is a global variable
## y = 10 is a closed variable (a parameter of the outer function)
## z = 2 is a closed variable (a local of the outer function)
```

Generic functions

If you use Julia to write code for research you should aim to write **generic functions**

Generic functions

If you use Julia to write code for research you should aim to write **generic functions**

These are functions that are flexible (e.g. can deal with someone using an `Int` instead of a `Float`) and have high performance (e.g. comparable speed to C)

Generic functions

If you use Julia to write code for research you should aim to write **generic functions**

These are functions that are flexible (e.g. can deal with someone using an `Int` instead of a `Float`) and have high performance (e.g. comparable speed to C)

Functions are made generic by paying attention to types and making sure types are **stable**

Generic functions

Type stability: Given an input into a function, operations on that input should maintain the type so Julia **knows** what its type will be throughout the full function call

Generic functions

Type stability: Given an input into a function, operations on that input should maintain the type so Julia **knows** what its type will be throughout the full function call

This allows it to compile type-specialized versions of the functions, which will yield higher performance

Generic functions

Type stability: Given an input into a function, operations on that input should maintain the type so Julia **knows** what its type will be throughout the full function call

This allows it to compile type-specialized versions of the functions, which will yield higher performance

The question you might have is: Type stability sounds like mandating types (e.g. what C and Fortran do, not what R/Python/etc do), so how do we make it flexible?

Generic functions

Type stability: Given an input into a function, operations on that input should maintain the type so Julia **knows** what its type will be throughout the full function call

This allows it to compile type-specialized versions of the functions, which will yield higher performance

The question you might have is: Type stability sounds like mandating types (e.g. what C and Fortran do, not what R/Python/etc do), so how do we make it flexible?

We'll see next

These two functions look the same, but are they?

```
function t1(n)
  s = 0
  t = 1
  for i in 1:n
     s += s/i
     t = div(t, i)
  end
  return t
end
```

```
function t2(n)
  s  = 0.0
  t = 1
  for i in 1:n
     s += s/i
     t = div(t, i)
  end
  return t
end
```

No! t1 is not type stable

No! t1 is not type stable

`t1` starts with `s` as an `Int64` but then we have `s += s/i` which will mean it must hold a `Float64`

No! t1 is not type stable

`t1` starts with `s` as an `Int64` but then we have `s += s/i` which will mean it must hold a `Float64`

It must be converted to `Float` so it is not type stable

No! t1 is not type stable

We can see this when calling the macro `@code_warntype` where it reports `t1` at some point handles `s` that has type `Union{Float64,Int64}`, either `Float64` or `Int64`

Julia now can't assume `s`'s type and produce pure integer or floating point code $\rightarrow$ performance degradation

```
Variables
  #self#::Core.Compiler.Const(t1, false)
  n::Int64
  s::Union{Float64, Int64}
  t::Int64
  @_5::Union{Nothing, Tuple{Int64,Int64}}
  i::Int64
```

```
Variables
  #self#::Core.Compiler.Const(t2, false)
  n::Int64
  s::Float64
  t::Int64
  @_5::Union{Nothing, Tuple{Int64,Int64}}
  i::Int64
```

THIS MATTERS

2x difference between two simple functions

```julia
# Type instable
function type_unstable()
  x = 1
  for i = 1:1e6
    x = x/2
  end
  return x
end
# Type stable
function type_stable()
  x = 1.0
  for i = 1:1e6
    x = x/2
  end
  return x
end
```

THIS MATTERS

2x difference between two simple functions

```
@time type_unstable()
```

```
##   0.004397 seconds
```

```
## 0.0
```

```
@time type_stable()
```

```
##   0.002884 seconds
```

```
## 0.0
```

Concrete vs abstract types

A **concrete type** is one that can be instantiated (`Float64` `Bool` `Int32`)

Concrete vs abstract types

A **concrete type** is one that can be instantiated (`Float64` `Bool` `Int32`)

An **abstract type** cannot (`Real`, `Number`, `Any`)

Concrete vs abstract types

Abstract types are for organizing the types

You can check where types are in the hierarchy

```
@show Float64 <: Real
```

```
## Float64 <: Real = true
```

```
## true
```

```
@show Array <: Real
```

```
## Array <: Real = false
```

```
## false
```

Concrete vs abstract types

You can see the type hierarchy with the supertypes and subtypes commands

```
using Base: show_supertypes
show_supertypes(Float64)
```

```
## Float64 <: AbstractFloat <: Real <: Number <: Any
```

Creating new types

We can actually create new composite types using `struct`

Creating new types

We can actually create new composite types using `struct`

```
struct FoobarNoType # This will be immutable by default
  a
  b
  c
end
```

Creating new types

This creates a new type called `FoobarNoType`, and we can generate a variable of this type using its **constructor** which will have the same name

Creating new types

This creates a new type called `FoobarNoType`, and we can generate a variable of this type using its **constructor** which will have the same name

```
newfoo = FoobarNoType(1.3, 2, "plzzz");
typeof(newfoo)
```

```
## FoobarNoType
```

```
newfoo.a
```

```
## 1.3
```

Creating new types

This creates a new type called `FoobarNoType`, and we can generate a variable of this type using its **constructor** which will have the same name

```
newfoo = FoobarNoType(1.3, 2, "plzzz");
typeof(newfoo)
```

```
## FoobarNoType
```

```
newfoo.a
```

```
## 1.3
```

You should always declare types for the fields of a new composite type

Creating new types

You can declare types with the double colon

```
struct FoobarType # This will be immutable by default
  a::Float64
  b::Int
  c::String
end
```

Creating new types

```
newfoo_typed = FoobarType(1.3, 2, "plzzz");
typeof(newfoo_typed)
```

```
## FoobarType
```

```
newfoo.a
```

```
## 1.3
```

This lets the compiler generate efficient code because it knows the types of the fields when you construct a `FoobarType`

Parametric types are what help deliver flexibility

We can create types that hold different types of fields by declaring subsets of abstract types

```
struct FooParam{t1 <: Real, t2 <: Real, t3 <: AbstractArray{<:Real}}
  a::t1
  b::t2
  c::t3
end
newfoo_para = FooParam(1.0, 7, [1., 4., 6.])
```

```
## FooParam{Float64, Int64, Vector{Float64}}(1.0, 7, [1.0, 4.0, 6.0])
```

Parametric types are what help deliver flexibility

We can create types that hold different types of fields by declaring subsets of abstract types

```
struct FooParam{t1 <: Real, t2 <: Real, t3 <: AbstractArray{<:Real}}
  a::t1
  b::t2
  c::t3
end
newfoo_para = FooParam(1.0, 7, [1., 4., 6.])
```

```
## FooParam{Float64, Int64, Vector{Float64}}(1.0, 7, [1.0, 4.0, 6.0])
```

The curly brackets declare all the different type subsets we will use in `FooParam`

Parametric types are what help deliver flexibility

We can create types that hold different types of fields by declaring subsets of abstract types

```
struct FooParam{t1 <: Real, t2 <: Real, t3 <: AbstractArray{<:Real}}
  a::t1
  b::t2
  c::t3
end
newfoo_para = FooParam(1.0, 7, [1., 4., 6.])
```

```
## FooParam{Float64, Int64, Vector{Float64}}(1.0, 7, [1.0, 4.0, 6.0])
```

The curly brackets declare all the different type subsets we will use in `FooParam`

This actually delivers high performance code!

Delivering flexibility

We want to make sure types are stable but code is flexible

Ex: if want to preallocate an array to store data, how do we know how to declare it's type?

Delivering flexibility

We want to make sure types are stable but code is flexible

Ex: if want to preallocate an array to store data, how do we know how to declare it's type?

We don't need to

Delivering flexibility

```
using LinearAlgebra                  # necessary for I
function sametypes(x)
  y = similar(x)                     # creates an array that is `similar` to x, use this for preal
  z = I                              # creates a scalable identity matrix
  q = ones(eltype(x), length(x))     # one is a type generic array of ones, fill creates the array
  y .= z * x + q
  return y
end
```

```
## sametypes (generic function with 1 method)
```

```
x = [5.5, 7.0, 3.1];
y = [7, 8, 9];
```

Delivering flexibility

We did not declare any types but the function is type stable

```
sametypes(x)
sametypes(y)
```

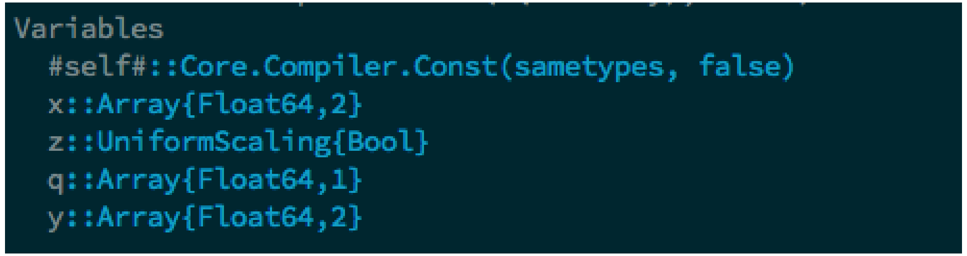
```
Variables
  #self#::Core.Compiler.Const(sametypes, false)
  x::Array{Float64,2}
  z::UniformScaling{Bool}
  q::Array{Float64,1}
  y::Array{Float64,2}
```

```
Variables
  #self#::Core.Compiler.Const(sametypes, false)
  x::Array{Int64,2}
  z::UniformScaling{Bool}
  q::Array{Int64,1}
  y::Array{Int64,2}
```

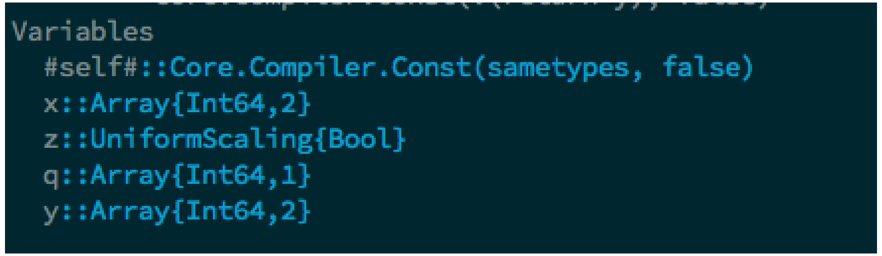
Delivering flexibility

We did not declare any types but the function is type stable

```
sametypes(x)
sametypes(y)
```



```
Variables
  #self#::Core.Compiler.Const(sametypes, false)
  x::Array{Float64,2}
  z::UniformScaling{Bool}
  q::Array{Float64,1}
  y::Array{Float64,2}
```



```
Variables
  #self#::Core.Compiler.Const(sametypes, false)
  x::Array{Int64,2}
  z::UniformScaling{Bool}
  q::Array{Int64,1}
  y::Array{Int64,2}
```

There's a lot of other functions out there that help with writing flexible, type stable code

Multiple dispatch

Why type stability really matters: multiple dispatch

Neat thing about Julia: the same function name can perform different operations depending on the underlying type of the inputs

A function specifies different **methods**, each of which operates on a specific set of types

Multiple dispatch

When you write a function that's type stable, you are actually writing many different methods, each of which are optimized for certain types

Multiple dispatch

When you write a function that's type stable, you are actually writing many different methods, each of which are optimized for certain types

If your function isn't type stable, the optimized method may not be used

This is why Julia can achieve C speeds: it compiles to C (or faster) code

Multiple dispatch

`/` has 103 different methods depending on the input types, these are 103 specialized sets of codes

```
methods(/)
```

```
## # 152 methods for generic function "/":
## [1] /(x::Union{Int128, Int16, Int32, Int64, Int8, UInt128, UInt16, UInt32, UInt64, UInt8}, y::Union{
## [2] /(x::T, y::T) where T<:Union{Float16, Float32, Float64} in Base at float.jl:386
## [3] /(x::Union{Integer, Complex{<:Union{Integer, Rational}}}, y::Rational) in Base at rational.jl:34
## [4] /(x::Union{Int16, Int32, Int64, Int8, UInt16, UInt32, UInt64, UInt8}, y::BigInt) in Base.GMP at
## [5] /(c::Union{UInt16, UInt32, UInt64, UInt8}, x::BigFloat) in Base.MPFR at mpfr.jl:441
## [6] /(c::Union{Int16, Int32, Int64, Int8}, x::BigFloat) in Base.MPFR at mpfr.jl:453
## [7] /(c::Union{Float16, Float32, Float64}, x::BigFloat) in Base.MPFR at mpfr.jl:465
## [8] /(U::Union{UnitUpperTriangular{var"#s886", S} where S<:AbstractMatrix{var"#s886"}, UpperTriangul
## [9] /(U::Union{UnitUpperTriangular{T, S} where S<:AbstractMatrix{T}, UpperTriangular{T, S} where S<:
## [10] /(L::Union{LowerTriangular{var"#s886", S} where S<:AbstractMatrix{var"#s886"}, UnitLowerTriangu
## [11] /(L::Union{LowerTriangular{T, S} where S<:AbstractMatrix{T}, UnitLowerTriangular{T, S} where S<
## [12] /(X::StridedArray{P}, y::P) where P<:Dates.Period in Dates at /Users/ir229/.julia/juliaup/julia
## [13] /(X::StridedArray{P}, y::Real) where P<:Dates.Period in Dates at /Users/ir229/.julia/juliaup/ju
```

Coding practices etc

See JuliaPraxis for best practices for naming, spacing, comments, etc