Introduction to STAT 385

James Joseph Balamuta

Department of Informatics, Statistics
University of Illinois at Urbana-Champaign

June 12th, 2017

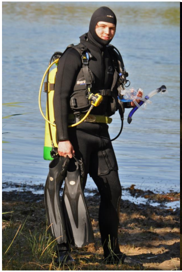
On the Agenda

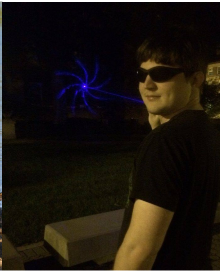
On the Agenda

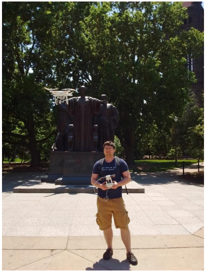
Hello
my name is

James

Who am I?

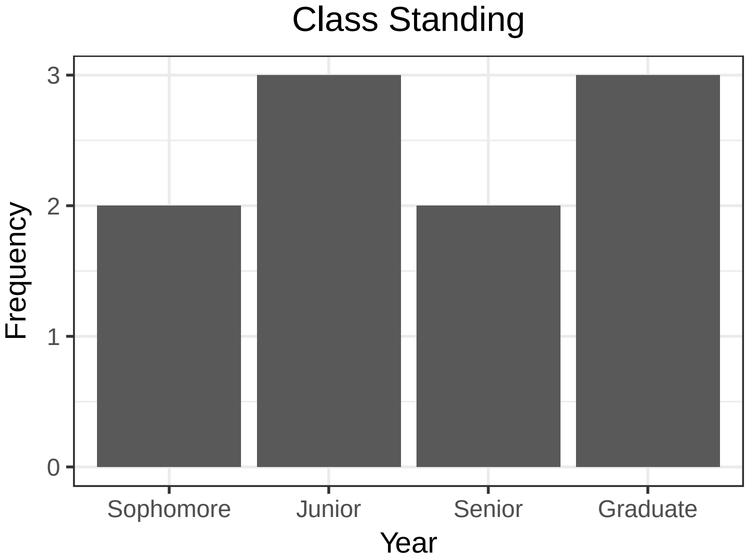




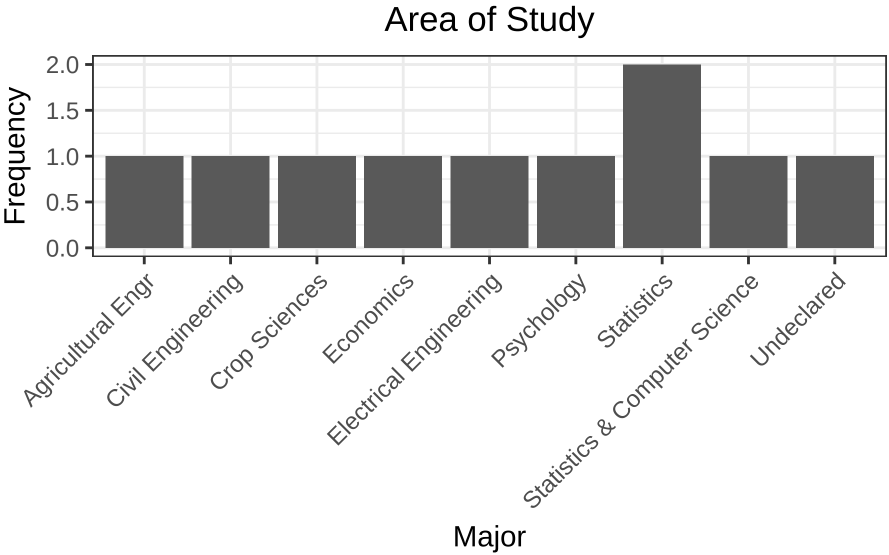


- 4th Year PhD Statistics/Informatics
- Research
  - NASA Carbon Monitor System Project
  - Time Series Latent Variable Estimation
  - Choice in Psychometric Models
- Teaching
  - List of Excellent Teachers (SU 2014)
  - Created three courses:
    - STAT 330: Data Visualization
    - STAT 385: Statistics Programming Methods (yes, **this** course!)
    - STAT 480: Data Science Foundations

You are...


Class Standing
Frequency
3
2
1
0
Sophomore
Junior
Senior
Graduate
Year

You are. . .


Area of Study
Frequency
2.0
1.5
1.0
0.5
0.0
Agricultural Engr
Civil Engineering
Crop Sciences
Economics
Electrical Engineering
Psychology
Statistics
Statistics & Computer Science
Undeclared
Major

Course Websites

- **Main Website:**
  http://stat385.thecoatlessprofessor.com/su2017
- **Discussion Forum:**
  http://github.com/stat385uiuc/su2017-disc (invite-only)
- **Source Code Repository:**
  http://github.com/stat385uiuc/su2017

About the Course

- Emphasize computing theory and methods for statistical algorithms
- Learn about computing to use in a future career or graduate school
- Will primarily cover R and C++ (through Rcpp)

Course Objectives

- View different statistical concepts presented from a programming perspective instead of a more theoretical framework.
- Implement different statistical algorithms.
- Use version control (github).
- Distributed computing.
- Group capstone project.

On the Agenda

1. Introduction
   - Profiles
   - Course Structure
   - Basics
2. Intro to R
   - Background of R
   - RStudio IDE
   - Sample Code
3. Types of Languages
   - Interpreters
   - Compilers

Age Old Question

"If a tree falls in a forest and no one is around to hear it, does it make a sound?"

Age Old Question Redux

Original:

"If a tree falls in a forest and no one is around to hear it, does it make a sound?"

Changes to:

"If a *statistical algorithm* exists and no one *uses* it, does it really exist?"

Technology?

Today, most people are *users* of a computer.

Simply put, they do **not** need to know how a computer works in order to use it.

The majority of folks simply turn on technology and immediately see a graphic that they can click or tap with a finger.

Take for example getting a sports score from ESPN.

Only *how* to interact with a computer program is **known** *not* the behind the scenes aggregation of *data*, recognition of *gestures*, and so on.

Computer == Scary?

This rationale has existed for awhile since computers are a bit scary...

Like spiders...

Figure 1: Michigan Wolf Spider

What is programming?

Definition:

Programming is the art of instructing a computer to do exactly what you say through an *algorithm*.

Definition:

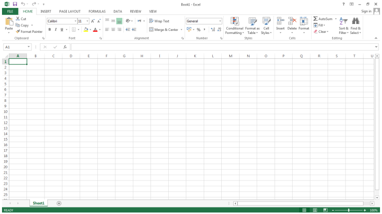
Algorithms are a process or set of rules to be followed in calculations or other problem-solving operations.

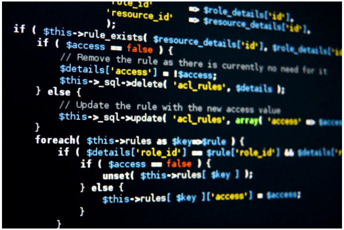
Why study programming now?

Programming has been available from the advent of computers. But, why am I hearing about it now?

What's changed?

A shift from working within **excel** to **automation**





Plus, a lot more CPU power on a traditional desktop.

Benefits

The adoption of programming methods within ones skillset and organization has several notable benefits:

- Speed
- Consistency
- Resources
- Computer Savviness
- Logic

On the Agenda

Learning a new language is HARD!

Ready?

What is *R*?

- *R* is a language designed specifically for statistical computing and graphics
- *R* is an interactive interface to many different tools.
- *R* is based on the S language, which was developed by Bell laboratories
- *R* is an open source (e.g. free) project that is cross platform (macOS, Windows, and Linux)
- *R* is available on *The R Project for Statistical Computing* website `http://www.r-project.org`

Why *R*?

Pros:

- It's free!
- Large repository of packages that often contain the latest breakthrough statistical methods
- Able to integrate *Fortran*, *C*, *C++*, and *Python* code via wrappers
- Large adoption of *R* in industry and academia

Cons:

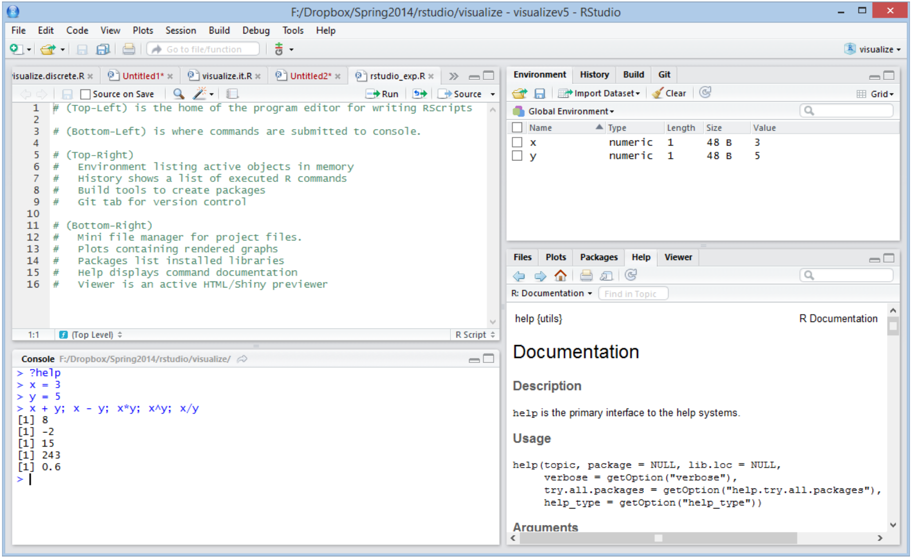
- Unable to handle large amounts of data
- Many idiosyncracies that break the mold of traditional programming languages

RStudio

RStudio is an Integrated Developer Environment (IDE) for *R*.

- Advanced GUI that emphasizes a project workflow
- Provides support for a novice user and an advanced user
- Open source (e.g. free) project that is cross platform (macOS, Windows, and Linux)
- Download RStudio via http://www.rstudio.com

RStudio View

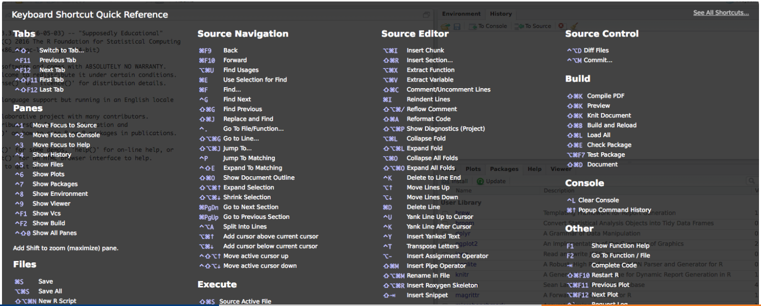


Know thine RStudio Environment

There are a *lot* of keyboard shortcuts in RStudio (Talk: Kevin Ushey, Talk: Sean Lopp). These shortcuts are meant to speed up your work.

To view all the options, you must engage the keyboard shortcut that rules them all:

- Windows: `Alt` + `Shift` + `K`
- macOS: `Option` + `Shift` + `K`



My Favorites

1. Runs the current line and/or current selection from the editor to the console and runs it
   - Windows: `Ctrl` + `Enter`
   - macOS: `Cmd` + `Enter`
2. Comment multiple lines.
   - Windows: `Ctrl` + `Shift` + `C`
   - macOS: `Command` + `Shift` + `C`
3. Multicursor:
   - Both: `Ctrl` + `Alt` + `Up` (or `Down`)
4. Reindent Code:
   - Windows: `Ctrl` + `L`
   - macOS: `Command` + `I`
5. Autocomplete command
   - Both: `Tab`

Disable persistent workspaces

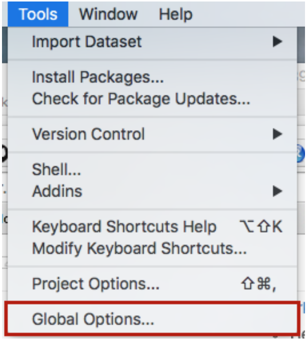
Within *R*, there is an option to have the persistent workspaces where previous computations can be made available. However, this option often leads to students and researchers quickly running into many unforseen difficulties.

- Not being able to reproduce the initial computation
- Slow program start up times
- Messy global environments.

Therefore, we believe it is important to disable this feature.

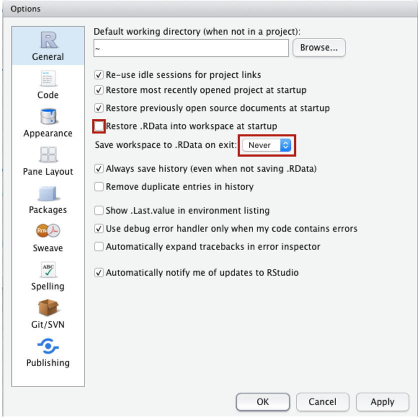
Preparing your environment

Navigate to Tools $\Rightarrow$ Global Options



Preparing your environment

- Uncheck `Restore .RData` into workspace at startup
- For `Save workspace to .RData on exit`, select the `Never` option from the dropdown.



Warming up to *R*

To begin our exploration of the *R* language, we'll use *R* to mimic a scientific calculator. Scientific calculators are able to:

- Compute mathematical expressions.
- Temporarily store values in a variable.

Explanations of the code, are given by comments predated by a #.

Output from the code is given by two ##.

Storing Values and Calculations

```r
# Create numeric object with values
x = 3
y = 5

# Perform calculations
x + y
```

```
## [1] 8
```

```r
x - y
```

```
## [1] -2
```

```r
# x*y; x/y; x^y;
```

Vectors

In *R*, a number like **5** is treated as a vector, or a collection of values, with **length of 1**.

We will see at a later time that this behavior, while odd, is actual pretty great to *vectorize* computations.

Vectors

```r
x = c(1,2,3,4,5) # Create vector
y = 6:10         # Shorthand

cbind(x, y)      # Combine Columns to form Matrix: 5 x 2
```

```
##      x  y
## [1,] 1  6
## [2,] 2  7
## [3,] 3  8
## [4,] 4  9
## [5,] 5 10
```

```r
rbind(x, y)      # Combine Rows to form Matrix: 2 x 5
```

```
##   [,1] [,2] [,3] [,4] [,5]
## x    1    2    3    4    5
## y    6    7    8    9   10
```

Built in Functions and Loops

Like any good programming language before it, *R* has built in functions to aide in the workflow.

A small sampling of functions is:

- `sum` - Summation over elements $\sum_{i=1}^{n} x_i$
- `mean` - Average over elements $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$
- `sd` - Standard Deviation over elements $\sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}$
- `sample` - Random sample from $x_1, x_2, \ldots, x_i, \ldots, x_n$

To view the function's help documentation, use:

```
?function_name
```

Built-in Functions & Loops

```
x = seq(1, 10, by = 2)   # 1, 3, 5, 7, 9
y = seq(10, 30, by = 5)  # 10, 15, .. , 30

result = numeric(1)      # Storage
for(i in 1:length(x)){   # (variable in sequence)
  result = x[i] + result
}                        # Loop (slow)

(out = sum(x))           # Vectorized Function (faster)
```

```
## [1] 25
```

```
all.equal(result, out)   # Same value
```

```
## [1] TRUE
```

```
#sd(x)                   # Standard Deviation
#cor(x,y)                # Correlation
#cov(x,y)                # Covariance
```

On the Agenda

Talking to a Computer

- In order to talk to a computer, you must speak its dialect.
- The dialect though is normally in 1's and 0's (or binary).
- Until Rear Admiral Grace M. Hopper came along...

Now, we have the option of:

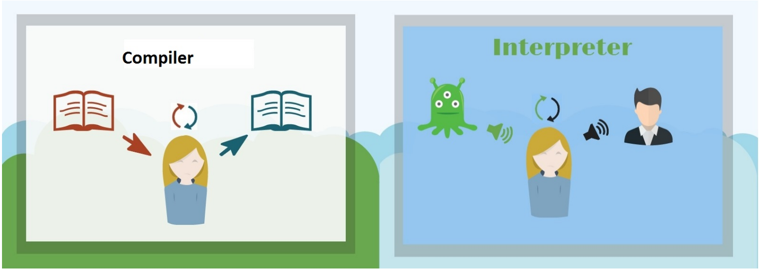


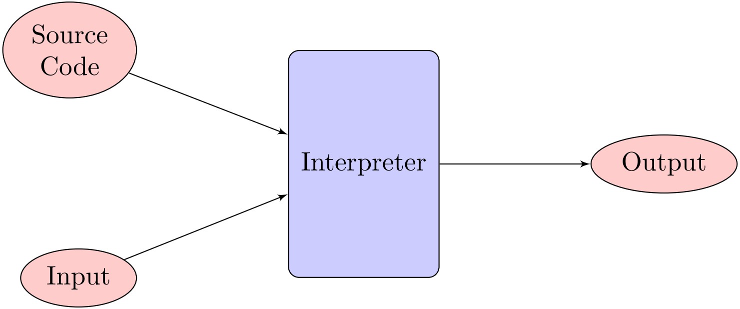
Image Source StackOverflow: Interpreter vs. compiler

What is an Interpreter?

An *interpreter* is a program that translates a high-level language into a low-level one, but it does it at the moment the program is run.

So, the interpreter takes the source code, one line at a time, and translates each line before executing it every time the program runs.

Think of like a person providing a “real time translation” to a conversation.



Interpreters: Pros & Cons

As a result of the program being instantly translated, it is able to immediately provide feedback (e.g. output, errors, etc).

For example, entering the following into R yields:

```r
3+4
```

```
## [1] 7
```

The downside to this approach are:

1. The lack of optimized code
2. Constant translation

Downsides of Interpreter Translation

Consider the effect of a slight change of code placement within a loop

```
bad.loop = function(){
  sum = 0
  for(i in 1:1000){
    a = 1/sqrt(2) # In loop
    sum = (sum+i)*a
  }
  sum
}
```

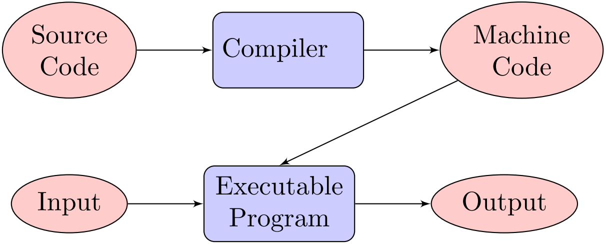
```
good.loop = function(){
  sum = 0
  a = 1/sqrt(2) # Out of Loop
  for(i in 1:1000){
    sum = (sum+i)*a
  }
  sum
}
```

test	replications	elapsed	relative	user.self	sys.self
good.loop()	100	0.007	1.000	0.008	0
bad.loop()	100	0.015	2.143	0.014	0

What is a Compiler? Can it do my taxes?

A compiler takes source code tries to optimize it before converting it into machine language **once**. After it is done compiling, the code can then be ran again and again without ever needing to be recompiled until the code is changed once more.

So, a compiler is like an editor who is asked to look over a paper. If it thinks something can be better, then it will take the initiative and implement that option.



Compilers: Pros

Compilers will attempt to optimize the code that they are given.

After a succesful compilation, the code will not need to be compiled again*.

Base *R*

```r
good.loop = function(){
  sum = 0
  a = 1/sqrt(2) # Out of Loop
  for(i in 1:1000){
    sum = (sum+i)*a
  }
  sum
}
```

Bytecompiled *R*

```
## list(.Code, list(10L, LDCONST.OP, 1L, SETVAR.OP, 3L, POP.OP,
##     LDCONST.OP, 5L, BASEGUARD.OP, 7L, 15L, LDCONST.OP, 8L, SQRT.
##     7L, DIV.OP, 9L, SETVAR.OP, 10L, POP.OP, LDCONST.OP, 5L, LDCO
##     13L, COLON.OP, 14L, STARTFOR.OP, 16L, 15L, 43L, GETVAR.OP,
##     3L, GETVAR.OP, 15L, ADD.OP, 18L, GETVAR.OP, 10L, MUL.OP,
##     19L, SETVAR.OP, 3L, POP.OP, STEPFOR.OP, 30L, ENDFOR.OP, POP.
##     GETVAR.OP, 3L, RETURN.OP), list({
##     sum = 0
##     a = 1/sqrt(2)
##     for (i in 1:1000) {
##         sum = (sum + i) * a
##     }
##     sum
## }, 0, structure(c(2L, 3L, 2L, 9L, 3L, 9L, 2L, 2L), srcfile = <en
##     sum, sum = 0, 1, structure(c(3L, 3L, 3L, 15L, 3L, 15L, 3L,
##     3L), srcfile = <environment>, class = "srcref"), sqrt(2),
##     2, 1/sqrt(2), a, a = 1/sqrt(2), structure(c(4L, 3L, 6L, 3L,
##     3L, 3L, 4L, 6L), srcfile = <environment>, class = "srcref"),
##     1000, 1:1000, i, for (i in 1:1000) {
##         sum = (sum + i) * a
##     }, structure(c(5L, 5L, 5L, 19L, 5L, 19L, 5L, 5L), srcfile =
##     sum + i, (sum + i) * a, sum = (sum + i) * a, structure(c(7L,
##     3L, 7L, 5L, 3L, 5L, 7L, 7L), srcfile = <environment>, class
##     structure(c(NA, 1L, 1L, 4L, 4L, 4L, 5L, 5L, 7L, 7L, 7L, 8L,
```

Compilers: Cons

Compilers spend a lot of time analyzing and processing the program.

In order to analyze the program, they require the **ENTIRE** program to be sent.

There is additional storage requirement due to machine generated code.

Errors will only appear **AFTER** the entire program is analyzed.

A looping example redux

After applying a compiler, there should be a noticeable change. . .

```
library("compiler")
good.comp = cmpfun(good.loop)
bad.comp = cmpfun(bad.loop)
```

test	replications	elapsed	relative	user.self	sys.self
good.loop	100	0.012	1.000	0.007	0
good.comp	100	0.017	1.417	0.007	0
bad.loop	100	0.017	1.417	0.014	0
bad.comp	100	0.026	2.167	0.014	0

Both of compiled version outperformed their respective interpreter version. However, the “bad” loop was not optimized to match the performance of the “good” loop due to limitations non-native compilation options.

To summarize. . .

Compilers

Pros:

1. Code is optimized
2. Program runs faster
3. Compiles once*

Cons:

1. Lots of time spent analyzing and optimizing code
2. More storage required due to machine code
3. Errors only show at the end

Interpreters

Pros:

1. Immediate Feedback
2. Less storage required
3. Friendlier

Cons:

1. Takes a single instruction as input
2. Constant compiles from high-level to low-level
3. Slow program execution

*no errors, changes, etc.