An Introduction to Programming in R

Drew Schmidt

November 13, 2015

About

This tutorial:

- This is not a "how to fit a linear model in R" tutorial.
- Slides and exercises are available at: http://wrathematics.info/handouts/bas2015.html
- I try to stick to base R as much as possible, even when better alternatives exist.
- Mostly independent; we can skip something that's boring!

Me:

- I am a very productive R package developer.
- I mostly do not show you the things I make.
- My expertise is R in HPC (contact me if interested!)

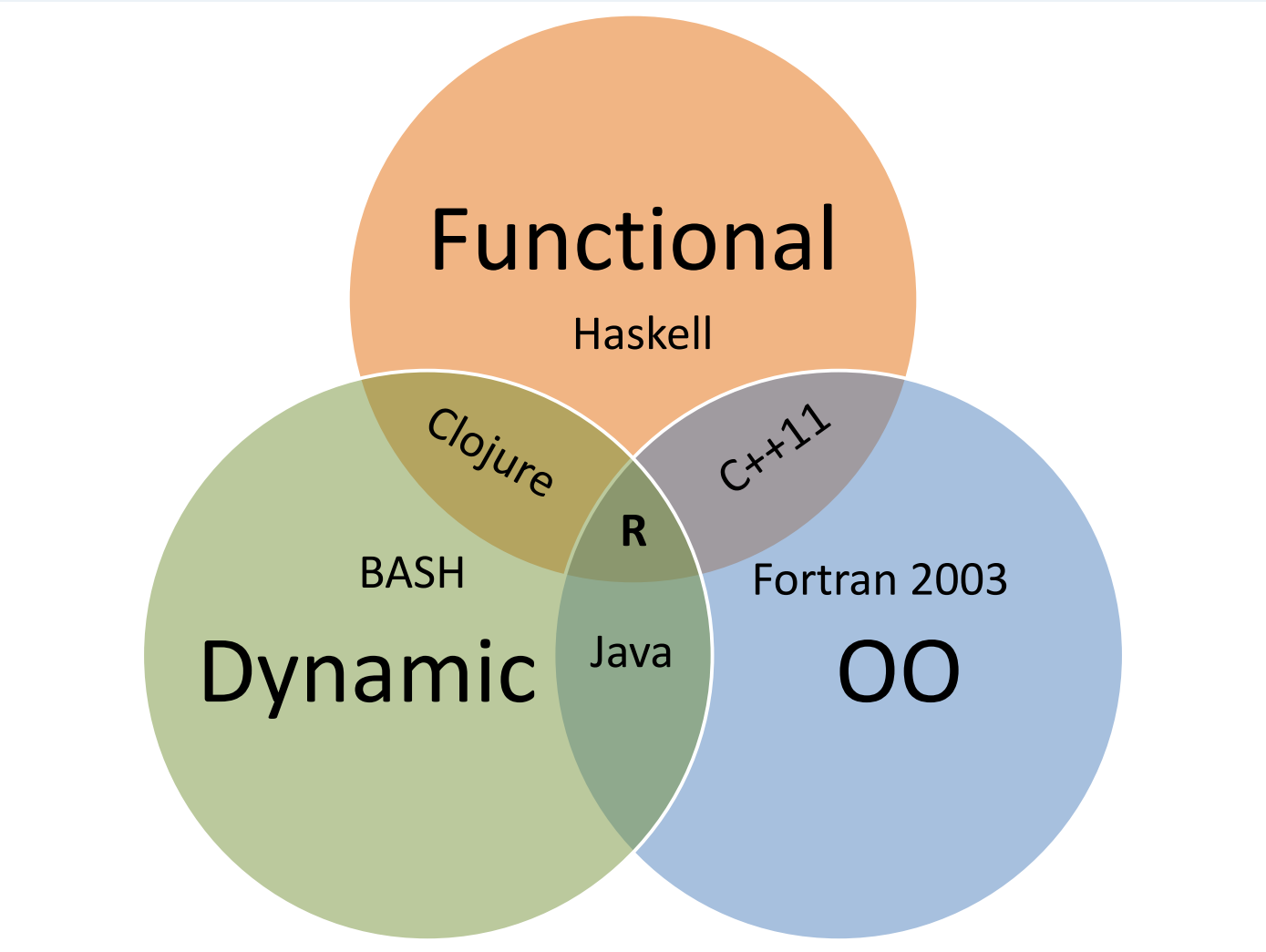
What is R?

- *lingua franca* for data analytics and statistical computing.
- Part programming language, part data analysis package.
- Dialect of S (Bell Labs).
- Syntax designed for data.

Who uses R?


ORACLE
MERCK
KICKSTARTER
JOHN DEERE
ebay
Bank of America
The New York Times
NIST
mozilla FOUNDATION
ORBITZ
cfpb
Shell
Pfizer
FDA
NOVARTIS
LLOYD'S
bing
Google
Ford
Zillow
okcupid

Language Paradigms



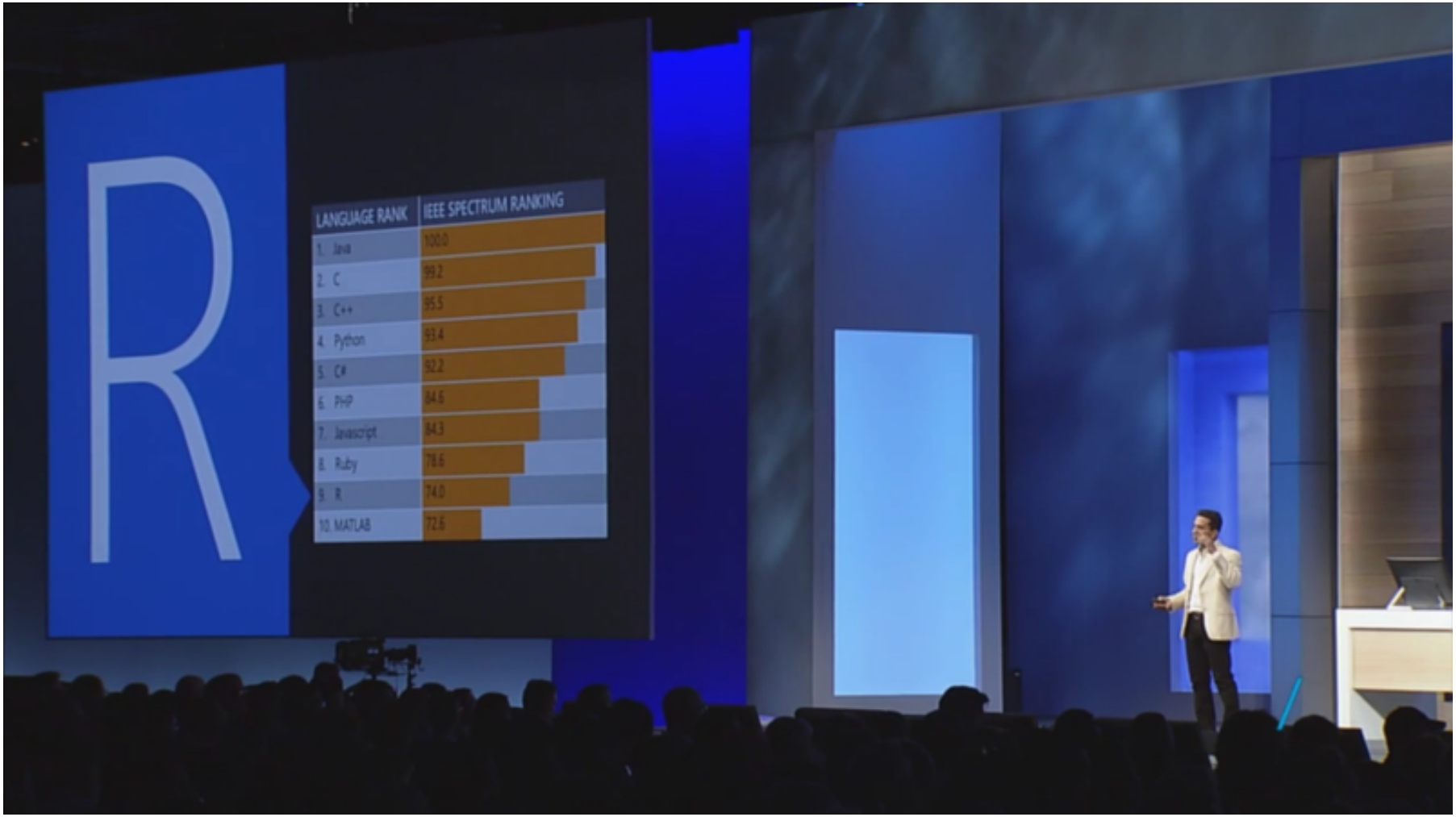
Very Popular for a DSL!

IEEE Spectrum's 2014 Ranking of Programming Languages

Language Rank	Types	Spectrum Ranking
1. Java		100.0
2. C		99.3
3. C++		95.5
4. Python		93.4
5. C#		92.4
6. PHP		84.7
7. Javascript		84.4
8. Ruby		78.8
9. R		74.2
10. MATLAB		72.9

See:

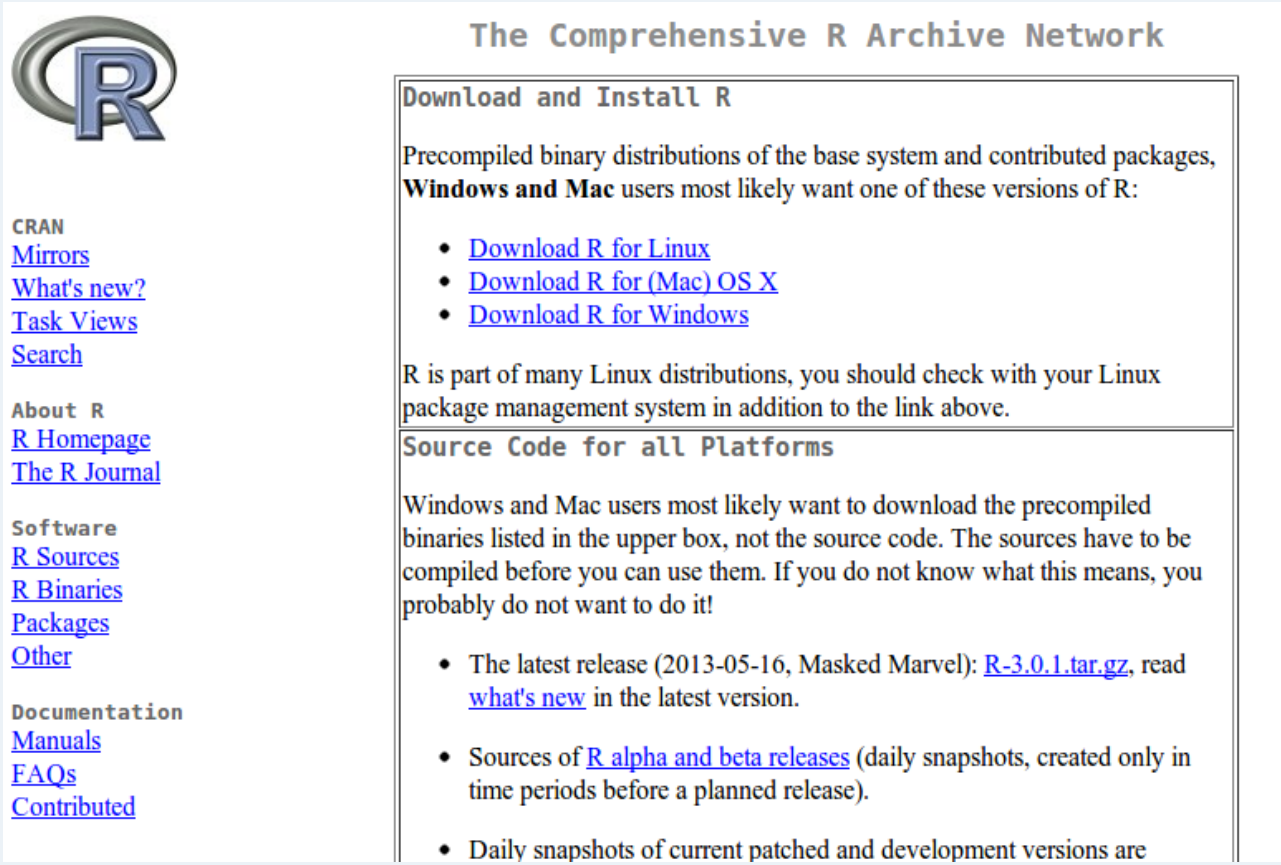
http://spectrum.ieee.org/static/interactive-the-top-programming-languages#index



At Build 2015 Microsoft CVP Joseph Sirosh called R the "language of data" and said *"if there is a single language that you choose to learn today .. let it be R"*.

Installing R

Go to http://cran.r-project.org/ for a download (binary or source)



CRAN
Mirrors
What's new?
Task Views
Search

About R
R Homepage
The R Journal

Software
R Sources
R Binaries
Packages
Other

Documentation
Manuals
FAQs
Contributed

The Comprehensive R Archive Network

Download and Install R

Precompiled binary distributions of the base system and contributed packages, **Windows and Mac** users most likely want one of these versions of R:

- Download R for Linux
- Download R for (Mac) OS X
- Download R for Windows

R is part of many Linux distributions, you should check with your Linux package management system in addition to the link above.

Source Code for all Platforms

Windows and Mac users most likely want to download the precompiled binaries listed in the upper box, not the source code. The sources have to be compiled before you can use them. If you do not know what this means, you probably do not want to do it!

- The latest release (2013-05-16, Masked Marvel): R-3.0.1.tar.gz, read what's new in the latest version.
- Sources of R alpha and beta releases (daily snapshots, created only in time periods before a planned release).
- Daily snapshots of current patched and development versions are

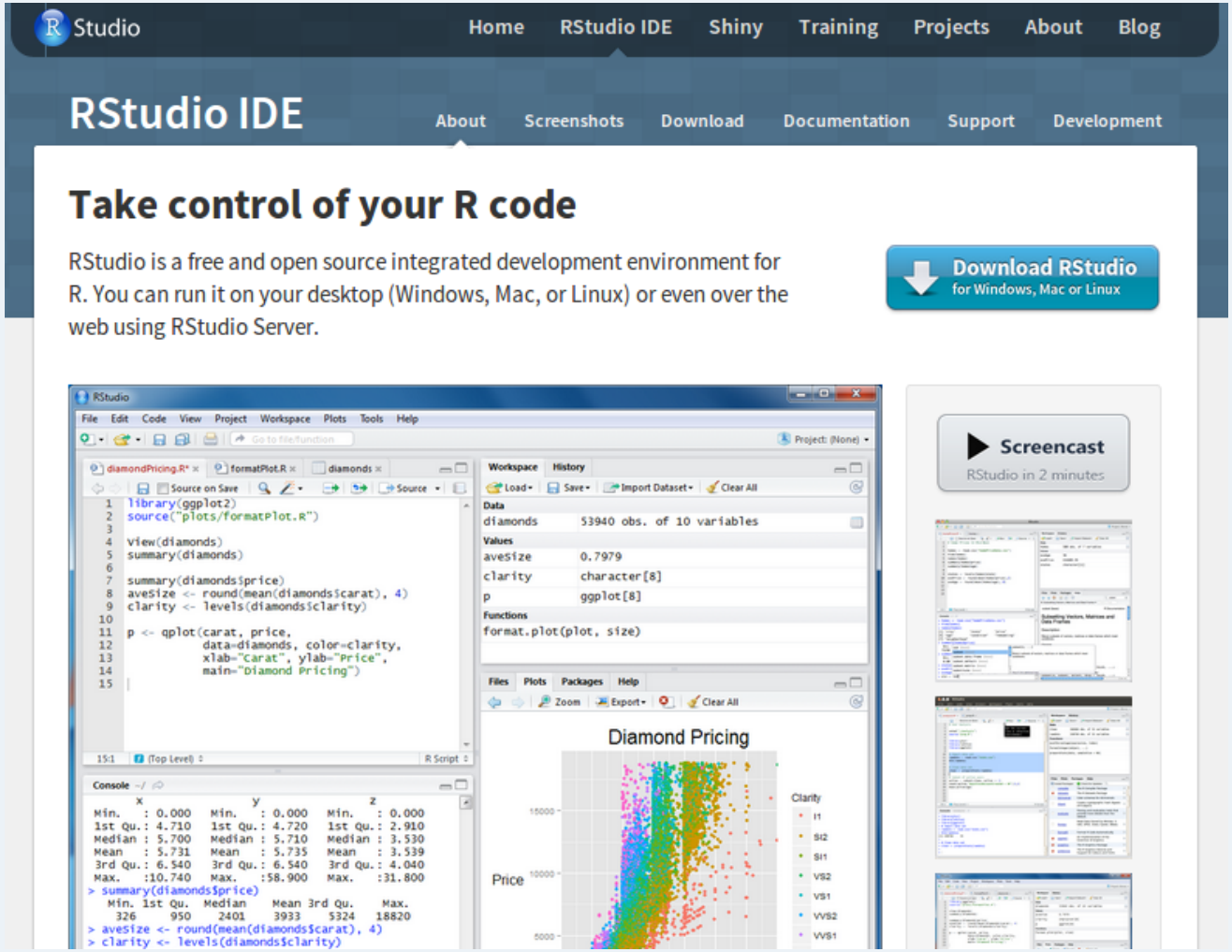
Installing R

Windows users should also install Rtools
http://cran.r-project.org/bin/windows/Rtools/

For a complete set of installation instructions, see
http://www.r-pbd.org/install.html

Installing Rstudio

Go to http://www.rstudio.com/ide/



Important things we can't cover

- R and version control
- Developing R packages
- Performance/profiling
- Graphics/visualization

R Resources

- *The Art of R Programming* by Norm Matloff: http://nostarch.com/artofr.htm
- *An Introduction to R* by Venables, Smith, and the R Core Team: http://cran.r-project.org/doc/manuals/R-intro.pdf
- *The R Inferno* by Patrick Burns: http://www.burns-stat.com/pages/Tutor/R_inferno.pdf
- Mathesaurus: http://mathesaurus.sourceforge.net/
- R programming for those coming from other languages: http://www.johndcook.com/R_language_for_programmers.html
- *aRrgh: a newcomer's (angry) guide to R*, by Tim Smith and Kevin Ushey: http://tim-smith.us/arrgh/

Tutorials

- *R Programming*, Coursera course through Johns Hopkins
  https://www.coursera.org/course/rprog
- *Statistics One* Coursera course through Princeton
  https://www.coursera.org/course/stats1
- *High Performance Computing with R*
  https://github.com/wrathematics/2015hpcRworkshop/blob/master/README.md

Other Invaluable Resources

- *R Installation and Administration*: http://cran.r-project.org/doc/manuals/R-admin.html
- *Task Views*: http://cran.at.r-project.org/web/views
- *Writing R Extensions*: http://cran.r-project.org/doc/manuals/R-exts.html
- Mailing list archives: http://tolstoy.newcastle.edu.au/R/
- The [R] stackoverflow tag.
- The #rstats hastag on Twitter.

Comments and Advice

- R is part statistics package, part programming language.
- R is slow; if you don't know what you're doing, it's *really* slow.
- There is an R help mailing list. Use stackoverflow instead....
- Learn to love the R help system.
- If something appears broken in core R, it's (probably) not them, it's you.
- Try to avoid "super functions".

Interacting with the R Terminal

The default thing for R to do is print/show:

```
1
1+1
print(1+1)
sum

+ # ctrl+c to break
"+"
`+`
```

Arithmetic

```
1 7+4
2 7-4
3 7*4
4 7/4
5 7^4
6 7%%4
```

Assignment

R naming rules can be quite lax. For all practical purposes:

- Start with a letter
- Should consist of letters (any case), numbers, .'s, and _'s
- Very strange things are possible, however...

```
rm(list=ls())
```

```
"&*()" <- 3

"2" <- 1
`2` + 1
## [1] 2

"\U0001f431" <- "even unicode"
cat(ls())
## &*() 🐱 2
```

Assignment

```
1 # Local
2 myvar <- 1
3 myvar = 4
4
5 # Global
6 myvar <<- 2
7
8 # Whever you want
9 assign(x="myvar", value=3)
```

Case and Spacing

R is case sensitive, but fairly lax about spacing:

```
1 x <- 1:5
2 x
3 X
4
5 1      +  1
6 sum(x)
7 sum ( x    )
8 s um(x)
```

Finding Help

- R has its own manual system.
- Most of the answers to your questions lie within.
- Find help using ? or help(), or search across all help with ??.

```
1 ?sum
2 ??sum
3 help("sum")
4
5 ?+
6 ?"+"
```

Ways to Run R

- Interactive: typing commands into the console.
- Batch: Rscript, R CMD BATCH.
- From an IDE/GUI.
- Batch from a GUI: source()

On the note of directories...

- Windows paths use / or
  , e.g. c:/myfile.r
- Get current working directory: getwd()
- Set current working directory: setwd("path/to/my/dir")

R Extensions

- R is great, but limited.
- Has a great package extension system.
- R doesn't do what you want? Make it!

"*...if I don't know how to fix it, I can hire somebody else to fix it for me.*"
— Matt Dowle, developer of the data.table package.

Packages

- "R extensions"
- Comprehensive R Archive Network (CRAN).
- 7474 packages (nrow(available.packages()))
- CRAN Task Views http://cran.r-project.org/web/views/

Terminology

- A package is a collection of code usable by R.
- A library is a collection of packages.
- library() loads a package.
- Don't think too much about this...

Terminology

- We call the extension a *package*.
- *Packages* go into a *library*.
- **This is dumb and confusing, but there's nothing we can ever do about it.**

Example Confusion

- I wrote a library.
- I put the library in a package.
- I install the package ...into a library.
- I load the package with library() ???

BOOM

Installing R Packages

```
1 install.packages("devtools")
```

```
1 install.packages("devtools", lib="some/place/on/disk")
```

```
1 R CMD INSTALL devtools_1.6.tar.gz -l /some/place/on/disk
```

Repositories

- This basically assumes you're using CRAN.
- Lots of exciting development is happening outside of CRAN these days.
- Other binary package repositories: Bioconductor, R-forge (Windows)
- Other source repositories: GitHub, Bitbucket, ...

GitHub Binaries?

Karl Broman @kwbroman · Sep 27

.@hadleywickham @millerdl It would be great to have a service that would complie windows/mac binaries of #rstats pkgs on github.

Reply Retweet Favorite More

Hadley Wickham
@hadleywickham

Follow

@kwbroman @millerdl agreed. Plans are in motion

Installing Packages from Source

To install packages from source, you need some compilers:

- **Windows**: Install Rtools.
- **Mac**: Install Xcode from the app store, possibly some other things from here. If you need OpenMP, god help you.
- **Linux and FreeBSD**: You're good to go.

Installing R Packages

```
1 devtools::install_github("wrathematics/lineSampler")
```

Devtools Package Install Functions		
install_bitbucket	install_github	install_svn
install_deps	install_gitorious	install_url
install_git	install_local	install_version

I/O

Input	Output
readcsv()	writecsv()
readtable()	writetable()
readBin()	writeBin()
readLines()	writeLines()
save()	load()
saveRDS()	readRDS()
scan()	

"What about my really weird use-case?"

- NUMEROUS R packages
- XML, json, databases, ...
- See also: *R Data Import/Export* manual https://cran.r-project.org/doc/manuals/r-release/R-data.html

Reading and Writing a CSV

```
1 x <- matrix(1:30, 10)
2 write.csv(x, file="x.csv")
3 read.csv("x.csv")
4
5 write.csv(x, file="x.csv", row.names=FALSE)
6 read.csv("x.csv", header=TRUE)
```

The dreaded stringsAsFactors

- Default for R functions (e.g., read.csv()) is stringsAsFactors=TRUE.
- If you do not need to modify the "strings" (going straight to modeling), stringsAsFactors=TRUE.
- If you **DO** need to modify, stringsAsFactors=FALSE

Serializing

```
1  x <- letters
2  y <- 1:5
3  z <- list(list("a"), b=matrix(0))
4
5  save(x, y, z, file="robjects.RData")
6  rm(x)
7  rm(y)
8  rm(z)
9  x
10 load("robjects.RData")
```

See also saveRDS().

Why/Why Not Serialize?

Why:

- Serializing is a binary "as-is" format.
- Performance!

Why not:

- Only works with R (or something that can read R binary formats!)
- Endianness...

Some Notable Packages

General:

- rio https://cran.r-project.org/web/packages/io/index.html

CSV and friends:

- data.table (fread()) https://cran.r-project.org/web/packages/data.table/index.html
- readr https://cran.r-project.org/web/packages/readr/index.html

Readers of proprietary things:

- haven (SAS, SPSS, and STATA) https://cran.r-project.org/web/packages/haven/index.html
- readxl (Excel) https://cran.r-project.org/web/packages/readxl/index.html

The data() Command

- Many packages (and R itself) bundle data.
- Load data with data()
- For a list of R's: library(help="datasets")

Example

```
1 head(iris)
2 head(mtcars)
3
4 data(package="ggplot2")
5 diamonds # error
6 data(diamonds)
7 head(diamonds)
```

Strings

- "character" data (text).
- Internal storage scheme is complicated...
- Managing character data is its own course!
- Just the baby basics...

Quotes

- Three kinds: single quote ', double quote ", backtick `.
- Single and double create strings.
- Backticks access language objects (e.g., `+`).
- Escape within a string with a *single backslash*:

```
1 quoted_quote <- "\"\n"
2 quoted_quote
3 ## [1] "\"\n"
4 cat(quoted_quote)
5 ## "
```

Example 1

```
1  letters
2  ##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o"
       "p" "q" "r" "s"
3  ## [20] "t" "u" "v" "w" "x" "y" "z"
4
5  toupper(letters)
6  ##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O"
       "P" "Q" "R" "S"
7  ## [20] "T" "U" "V" "W" "X" "Y" "Z"
8
9  LETTERS
10 ##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O"
       "P" "Q" "R" "S"
11 ## [20] "T" "U" "V" "W" "X" "Y" "Z"
12
13 tolower(LETTERS)
14 ##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o"
       "p" "q" "r" "s"
15 ## [20] "t" "u" "v" "w" "x" "y" "z"
```

Example 2

```
1  x <- "Star Trek is objectively better than Star Wars"
2  strsplit(x, split=" ")
3  ## [[1]]
4  ## [1] "Star"        "Trek"        "is"          "objectively"
       "better"
5  ## [6] "than"        "Star"        "Wars"
6
7  y <- unlist(strsplit(x, split=""))
8  y
9  ##  [1] "S" "t" "a" "r" " " "T" "r" "e" "k" " " "i" "s" " " "o" "b"
       "j" "e" "c" "t"
10 ## [20] "i" "v" "e" "l" "y" " " "b" "e" "t" "t" "e" "r" " " "t" "h"
       "a" "n" " " "S"
11 ## [39] "t" "a" "r" " " "W" "a" "r" "s"
12
13 paste(rev(y), collapse="")
14 ## [1] "sraW ratS naht retteb ylevitcejbo si kerT ratS"
```

Example 3

```
1  paste(letters, letters)
2  ##  [1] "a a" "b b" "c c" "d d" "e e" "f f" "g g" "h h" "i i" "j j"
       "k k" "l l"
3  ## [13] "m m" "n n" "o o" "p p" "q q" "r r" "s s" "t t" "u u" "v v"
       "w w" "x x"
4  ## [25] "y y" "z z"
5
6  paste(letters, letters, sep="")
7  ##  [1] "aa" "bb" "cc" "dd" "ee" "ff" "gg" "hh" "ii" "jj" "kk" "ll"
       "mm" "nn" "oo"
8  ## [16] "pp" "qq" "rr" "ss" "tt" "uu" "vv" "ww" "xx" "yy" "zz"
9
10 paste(letters, letters, sep="", collapse="")
11 ## [1] "aabbccddeeffgghhiijjkkllmmnnooppqqrrsstuuvvwwxxyyzz"
12
13 paste(paste(letters, collapse=""), paste(LETTERS, collapse=""),
       sep="")
14 ## [1] "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

Example 4

```
1 x <- rnorm(1000, mean=10)
2
3 paste("The mean of 'x' is:", mean(x))
4 ## [1] "The mean of 'x' is: 10.0157377724829"
5
6 cat(sprintf("mean:\t%.2f\nvar:\t%.2f\n", mean(x), var(x)))
7 ## mean:    10.02
8 ## var:     0.95
```

Regular Expressions

- grep(), grepl()
- sub(), gsub()
- regexpr(), gregexpr()
- Some others...

Some Notable Packages

- stringi
  https://cran.r-project.org/web/packages/stringi/index.html
- tm https://cran.r-project.org/web/packages/tm/index.html

Dates

- Dates managed in the UNIX style.
- Probably very alien for you...
- Just use lubridate.

Formatting Dates with Lubridate

```
1  rightnow <- Sys.time()
2  rightnow
3
4  library(lubridate)
5  year(rightnow)
6  ## [1] 2015
7
8  month(rightnow)
9  ## [1] 11
10 month(rightnow, label=TRUE)
11 ## [1] Nov
12
13 wday(rightnow, label=TRUE)
14 ## [1] Mon
```

And much more!

See the package vignette: https://cran.r-project.org/web/packages/lubridate/vignettes/lubridate.html

Example: Timezone Lookup

```
1  timezone <- function()
2  {
3    time <- Sys.time()
4    ret <- list(timezone=format(time, format="%Z"),
         UTC.offset=format(time, format="%z"))
5    class(ret) <- "tzlookup"
6    return(ret)
7  }
8
9  print.tzlookup <- function(x)
10 {
11   maxlen <- max(sapply(names(x), nchar))
12   spacenames <- c("timezone:    ", "UTC Offset:")
13   cat(paste(spacenames, x, sep=" ", collapse="\n"), "\n")
14 }
15
16 timezone()
17 ## timezone:     EST
18 ## UTC Offset: -0500
```

Some Notable Packages

- lubridate https://cran.r-project.org/web/packages/lubridate/index.html

Motivation

- For many, dataframes are the most important/common object.
- ENORMOUS topic.
- We cover some important techniques.

Modifying Elements

```
### Matrix-like notation
iris[1,1] <- 0
iris[, 1] <- 0

edit(iris)
```

Adding Variables

```
index <- 1:nrow(iris)
iris$index <- index
head(iris)

rm(iris)
iris <- cbind(iris, index)
head(iris)

rm(iris)
iris[, ncol(iris)+1] <- index
head(iris)

rm(iris)
```

Adding Rows

```
1 nrow(iris)
2 set.seed(12345)
3 newrow <- iris[sample(nrow(iris), size=1), ]
4
5 iris <- rbind(iris, newrow)
6 nrow(iris)
7
8 iris[nrow(iris)+1, ] <- newrow
9 nrow(iris)
```

Adding Variables

```
1  index <- 1:nrow(iris)
2  iris$index <- index
3  head(iris)
4
5  rm(iris)
6  head(iris)
7
8  iris <- cbind(iris, index)
9  head(iris)
10
11 rm(iris)
```

Subsetting

```
1 subset(iris, Sepal.Length > 7)
2
3 subset(iris, Sepal.Length > 7 & Petal.Length < 6)
4
5 subset(iris, Species == "setosa" | Species == "versicolor")
```

Sorting/Ordering Rows

```
df1 <- iris[order(iris$Sepal.Length), ]
head(df1)

df2 <- iris[order(-iris$Sepal.Length), ]
head(df2)

df3 <- iris[order(-iris$Sepal.Length, iris$Sepal.Width), ]
head(df3)
```

Sorting/Ordering Columns

```
1  iris <- iris[1:5, ]
2
3  iris[5:1]
4  set.seed(12345)
5  iris[sample(1:ncol(iris))]
6
7  iris[1]
8
9  iris[sort(colnames(iris))]
10
11 iris[sort(colnames(iris), decreasing=TRUE)]
12
13 rm(iris)
```

Dealing with Duplicates

```
1 any(duplicated(iris))
2 which(duplicated(iris))
3
4 nrow(unique(iris))
```

Applying Functions

```
1 colMeans(iris)
2 colMeans(iris[, -5])
3
4 apply(iris[, -5], MARGIN=2, FUN=median)
```

Some Notable Packages

Dataframe-like things:

- data.table https://cran.r-project.org/web/packages/data.table/index.html
- dplyr https://cran.r-project.org/web/packages/dplyr/index.html

Restructuring helpers:

- reshape2 https://cran.r-project.org/web/packages/reshape2/index.html
- tidyr https://cran.r-project.org/web/packages/tidyr/index.html
- broom https://cran.r-project.org/web/packages/broom/index.html

The R Language

- Storage: logical, int, double, double complex, character (strings)
- Structures: vector, matrix, array, list, dataframe, environment (hashtable)
- Caveats: (Logical) TRUE, FALSE, NA
- In fact, there's an NA for each type:
  - Integer: $-\left(2^{-31}-1\right)$
  - Double: value at address 0x7FF00000000007A2LL
- 3 official OOP systems, several unofficial ones.
- Several unofficial packages supporting C++.

Data Types

- logical
- int
- double
- double complex
- character

Data Types

R is *dynamically typed*. You do not have to declare what kind of data a variable is before you start using it:

```
1 x <- 1
2 typeof(x)
3 x <- 1:2
4 typeof(x)
```

Other

There are 4 "other" data "types":

- Inf
- NaN
- NULL
- NA

Inf

Numerical infinity

```
Inf
## [1] Inf

typeof(Inf)
## [1] "double"

is.finite(Inf)
## [1] FALSE

is.infinite(Inf)
## [1] TRUE

Inf+Inf
## [1] Inf

1/0
## [1] Inf
```

NaN

Not a Number; numerical undefinedness.

```
NaN
## [1] NaN

typeof(NaN)
## [1] "double"

is.nan(NaN)
## [1] TRUE

Inf-Inf
## [1] NaN

sin(Inf)
## [1] NaN
## Warning message:
## In sin(Inf) : NaNs produced
```

NULL

The null object; a sort of placeholder for something undefined. Like a non-numeric NaN.

```
NULL
## NULL

typeof(NULL)
## [1] "NULL"

is.null(NULL)
## [1] TRUE

NULL+NULL
## numeric(0)
```

NA

Missingness; not merely undefined, but *unknown*.

- Each type (logical, int, double, ...) has its own NA
- R is thus not boolean: TRUE, FALSE, NA
- Most R methods have ways of removing NA's

Data Structures

- vector
- matrix
- array
- factor
- list
- dataframe

Vectors

```
1:10
10:1

c(1, 3, 5, 7, 9)
seq(from=1, to=10, by=2)

x <- 1:5
length(x)
```

Matrices

```
1  matrix(1:10)
2  matrix(1:10, nrow=5)
3  matrix(1:10, ncol=5)
4
5  x <- 1:10
6  y <- as.matrix(x)
7  dim(x)
8  dim(y)
9  dim(x) <- c(1, 10)
10 x
```

Extraction

```
1  x <- matrix(1:30, 10)
2  x
3
4  x[-1, ]
5  x[, -1]
6  x[1:5, -1]
7  x[c(2,5,7), c(1,3)]
8
9  y <- x[, -2]
10 dim(y) <- NULL
11 y
```

Replacement

```
1 x <- matrix(1:30, 10)
2
3 x[1:5, ] <- 0
4 x[7, 3] <- NA
5 x
```

Factors

```
1 factor(1:5)
2 factor(c("a", "b", "b", "a", "c"))
3
4 x <- factor(-1:1)
5 x
6 as.numeric(x)
7 as.numeric(as.character(factor(-1:1)))
```

Dataframes

```
1  c(1, "a")
2  matrix(c(1, "a"))
3
4  x <- data.frame(1, "a")
5  x
6  x[1, 1]
7  is.numeric(x[1,1])
8  x[1, 2]
9  is.numeric(x[1,2])
10
11 data.frame(a=1:5,b=5:1)
```

Lists

- Super structures
- Items can be any structure (even other lists)
- Dataframe is really just a special list

Lists

```
1 list(1)
2 x <- list(list(1), "a")
3 x
4 x[[1]]
5
6 x <- list(a=list("b", 1), z=1:5)
7 x$z
```

Other Structures?

- stacks, heaps, queues, graphs, ...
- tl;dr: mostly no
- Hash table - environments
- Example deque https://github.com/wrathematics/dequer

Logic

- Possible values are TRUE, FALSE, and NA
- Operations: &&, ||, &, |
- Comparators: ==, !=, <, <=, >, and >=

Logic

```
1==1
1!=1
1<1
1<=1

TRUE==FALSE

TRUE==1
FALSE==0

TRUE==T
FALSE==F
```

Logic

```
1  NA==NA
2  is.na(NA)
3
4  NULL==NULL
5  is.null(NULL)
6
7  NaN==NaN
8  is.nan(NaN)
9
10 Inf==Inf
11 is.infinite(Inf)
```

Logic

Vectors of logicals are evaluated element-wise:

```
1 x <- c(TRUE, FALSE, FALSE, TRUE)
2 x==TRUE
3 !x
```

Logic

Some very important functions for logical evaluations:

```
x <- c(TRUE, FALSE, FALSE, TRUE)
any(x)
all(x)
which(x)

y <- -5:5
which(y%%2==0)
y[which(y%%2==0)]
```

Comparisons

Comparing to...

- Comparing to NULL: is.null(x)
- Comparing to NA: is.na(x)
- Comparing to TRUE: isTRUE(x)

Comparing two...

- Tread carefully...: x == y
- Numerics: all.equal(x, y)
- EXACTLY THE SAME: identical(x, y)

Why all this trouble?

BECAUSE COMPUTERS ARE TERRIBLE

```
1 x <- 1
2 for (i in 1:10) x <- x - .1
3 x
```

Another famous example

```
sprintf("%.17f", 0.1+0.2)
## [1] "0.30000000000000004"
```

This even has its own website! http://0.30000000000000004.com/

Loops

- Instruction set to be repeatedly executed until some condition is satisfied (possibly forever).
- R has for() and while().
- for(): Iterates over a list or vector
- while(): Performs operations until logical condition is satisfied.

for Loop

```
1  for (i in 1:10){
2    cat(i, " ")
3  }
4  ## 1  2  3  4  5  6  7  8  9  10
5
6  x <- matrix(1:30, 10)
7  colmax <- numeric(3)
8  for (i in 1:3){
9    colmax[i] <- max(x[, i])
10 }
11
12 colmax
13 ## [1] 10 20 30
```

while Loop

```
1  i <- 1
2  while (i < 11){
3    print(i)
4    i <- i+1
5  }
6
7  n <- 2
8  i <- 1
9  while (n < 1000){
10   n <- n^2
11   i <- i*2
12 }
13
14 n
15 ## [1] 65536
16 i
17 ## [1] 16
```

The *ply Family

- apply(): Apply function across "margin" (dimension) of matrix.
- lapply(): Apply function to input data object; returns a list.
- sapply(): Same as sapply() but
- mapply(): Multivariate sapply().
- vapply(): Essentially sapply(), sometimes faster.
- tapply(): Applying a function to a subset of a vector.
- ...

We will only discuss the first 3.

What is it?

- Syntax to loop without writing a loop.
- Inspired by functional programming.
- **Not the same thing as vectorization** (but it looks vectorized).

apply

```
x <- matrix(1:30, 10)
x

apply(X=x, MARGIN=1, FUN=min)
## [1]  1  2  3  4  5  6  7  8  9 10
apply(X=x, MARGIN=2, FUN=min)
## [1]  1 11 21

out <- numeric(3)
for (i in 1:3){
  out[i] <- min(x[, i])
}
out
## [1]  1 11 21
```

lapply and sapply

```
1  x <- 1:5
2  lapply(X=x, FUN=sqrt)
3  ## [[1]]
4  ## [1] 1
5  ##
6  ## [[2]]
7  ## [1] 1.414214
8  ##
9  ## [[3]]
10 ## [1] 1.732051
11 ##
12 ## [[4]]
13 ## [1] 2
14 ##
15 ## [[5]]
16 ## [1] 2.236068
17
18 sapply(X=x, FUN=sqrt)
19 ## [1] 1.000000 1.414214 1.732051 2.000000 2.236068
```

*ply Functions Internally

- apply(): A for() loop.
- lapply(): Internal R voodoo; faster than a loop.
- sapply(): Essentially the same as lapply().

When does all this choice matter?

- loops are slow.
- apply() is sugar for an R for loop.
- lapply() different, often faster.
- Vectorization is fastest of all.

Loop Speeds

```
x <- 100000:1

system.time({ # No initialization
  sin <- numeric(0)
  for (i in 1:length(x)){
    sin[i] <- sin(x[i])
  }
  sin
})
##    user  system elapsed
## 17.320   1.072  18.376

system.time({ # With initialization
  sin <- numeric(length(x))
  for (i in 1:length(x)){
    sin[i] <- sin(x[i])
  }
  sin
})
##    user  system elapsed
##   0.172   0.000   0.171
```

Loop Speeds

```
1  system.time(lapply(x, sin))
2  ##   user  system elapsed
3  ## 0.056   0.004   0.059
4
5  system.time(sapply(x, sin))
6  ##   user  system elapsed
7  ## 0.048   0.004   0.053
8
9  system.time(sin(x))
10 ##   user  system elapsed
11 ## 0.004   0.000   0.004
```

Functions

- Self-contained input/output machine.
- Reusable blocks of code.
- First class objects.
- All functions have returns!
- *Evaluating the Design of the R language*, http://r.cs.purdue.edu/pub/ecoop12.pdf

Some Basic Rules

- Can not modify data in place* (multi-return with a list)
- Arguments can have defaults, specified with =.
- By default, no printing occurs (have to say print(x)).
- The last "thing done" is the return.
- People often use NULL or invisible(NULL) if return is unimportant.

Functions: Example 1

```
1  f <- function(x)
2  {
3    ret <- x+1
4    return(ret)
5  }
6
7  f(1)
8  ## [1] 2
9  f(2)
10 ## [1] 3
11 f(5)
12 ## [1] 6
13 f
14 ## function(x)
15 ## {
16 ##   ret <- x+1
17 ##   return(ret)
18 ## }
```

Functions: Example 2

```
1  f <- function (a, b)
2  {
3    a - b
4  }
5
6  f(a=1, b=2)
7  f(1, 2)
8  f(b=1, a=2)
9  f(b=1, 2)
10 f(1)
11 f(matrix(1:4, nrow=2), matrix(4:1, nrow=2))
```

Functions: Example 3

For complicated returns (especially of mixed type/class), use a list:

```
1  g <- function (a, b)
2  {
3    plus <- a+b
4    minus <- a-b
5    return(list(plus, minus))
6  }
7
8  g(5, 2)
9  g(1, 0)
10 g(f(2, 6), 2)
```

Functions: Example 4

R allows parameter defaults in functions

```
1 h <- function (a=1, b=2)
2 {
3   return(b-a)
4 }
5
6 h
7 h()
8 h(2, 1)
```

Recursive Functions

A *recursive function* is one that calls itself:

```
1  f <- function(n)
2  {
3    if (n==1)
4      return(1)
5    else {
6      if (n%%2==0)
7        return(f(n/2))
8      else
9        return(f(3*n+1))
10   }
11 }
12
13 f(22)
14 ## [1] 1
15 f(237)
16 ## [1] 1
17 sapply(1:37, f)
18 ##   [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
       1 1 1 1 1 1 1
```

Debugging R Code

- Very broad topic ...
- We'll hit the highlights.
- For more examples, see: cran.r-project.org/doc/manuals/R-exts.html#Debugging
- Debugging compiled code called by R (valgrind, gdb, ...) also possible...

Object Inspection Tools

- print()
- str()
- unclass()

Object Inspection Tools: print()

Basic printing:

```
1  x <- matrix(1:10, nrow=2)
2  print(x)
3  ##      [,1] [,2] [,3] [,4] [,5]
4  ## [1,]    1    3    5    7    9
5  ## [2,]    2    4    6    8   10
6
7  x
8  ##      [,1] [,2] [,3] [,4] [,5]
9  ## [1,]    1    3    5    7    9
10 ## [2,]    2    4    6    8   10
```

Object Inspection Tools: str()

Examining the **str**ucture of an R object:

```
1 x <- matrix(1:10, nrow=2)
2
3 str(x)
4 ##  int [1:2, 1:5] 1 2 3 4 5 6 7 8 9 10
```

Object Inspection Tools: unclass()

Exposing all data with unclass():

```
1 df <- data.frame(x=rnorm(10), y=rnorm(10))
2 mdl <- lm(y~x, data=df) ### That's a "tilde" character
3
4 mdl
5 print(mdl)
6
7 str(mdl)
8
9 unclass(mdl)
```

The R Debugger

- debug()
- debugonce()
- undebug()

Using The R Debugger

1. Declare function to be debugged: `debug(foo)`
2. Call function: `foo(arg1, arg2, ...)`
   - next: Enter or `n` followed by Enter.
   - break: Halt execution and exit debugging: `Q`.
   - exit: Continue execution and exit debugging: `c`.
3. Call `undebug()` to stop debugging

Using the Debugger

Example Debugger Interaction

```
> f <- function(x){y <- z+1;z <- y*2;z}
> f(1)
Error in f(1) : object 'z' not found
> debug(f)
> f(1)
debugging in: f(1)
debug at #1: {
    y <- z + 1
    z <- y * 2
    z
}
Browse[2]>
debug at #1: y <- z + 1
Browse[2]>
Error in f(1) : object 'z' not found
>
```

Thanks so much for attending!

Questions?