A Beginner's Guide to Programming in R

Drew Schmidt

June 9, 2015

About This Tutorial

Slides and Exercises

The slides and exercises are available at:

http://wrathematics.info/handouts/dance2015.html

Following Along

- Log in to `arronax.nics.utk.edu` (you may wish to forward X11)
- Run `module load r/3.0.1`
- Run `R`

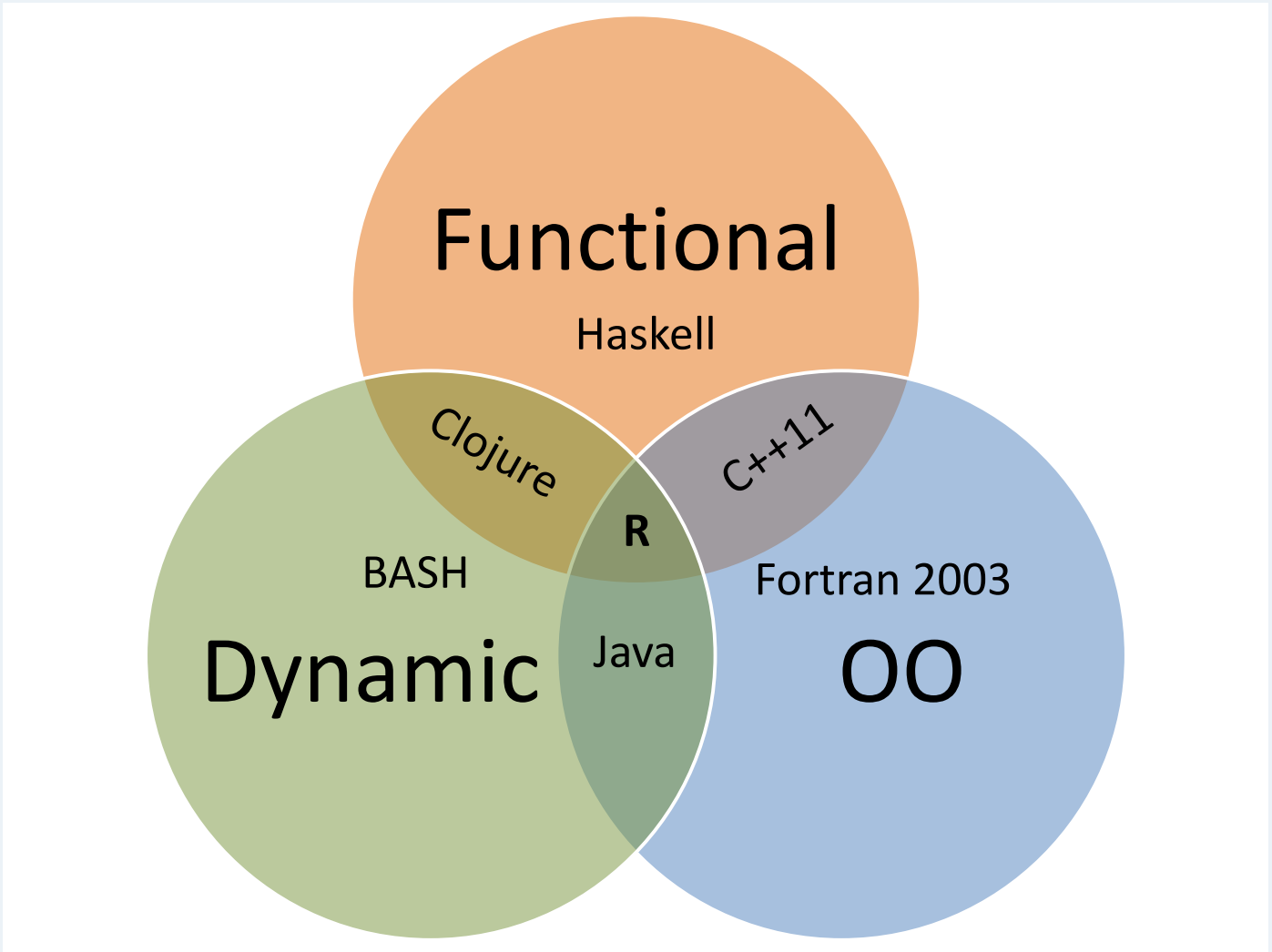
What is R?

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

Language Paradigms

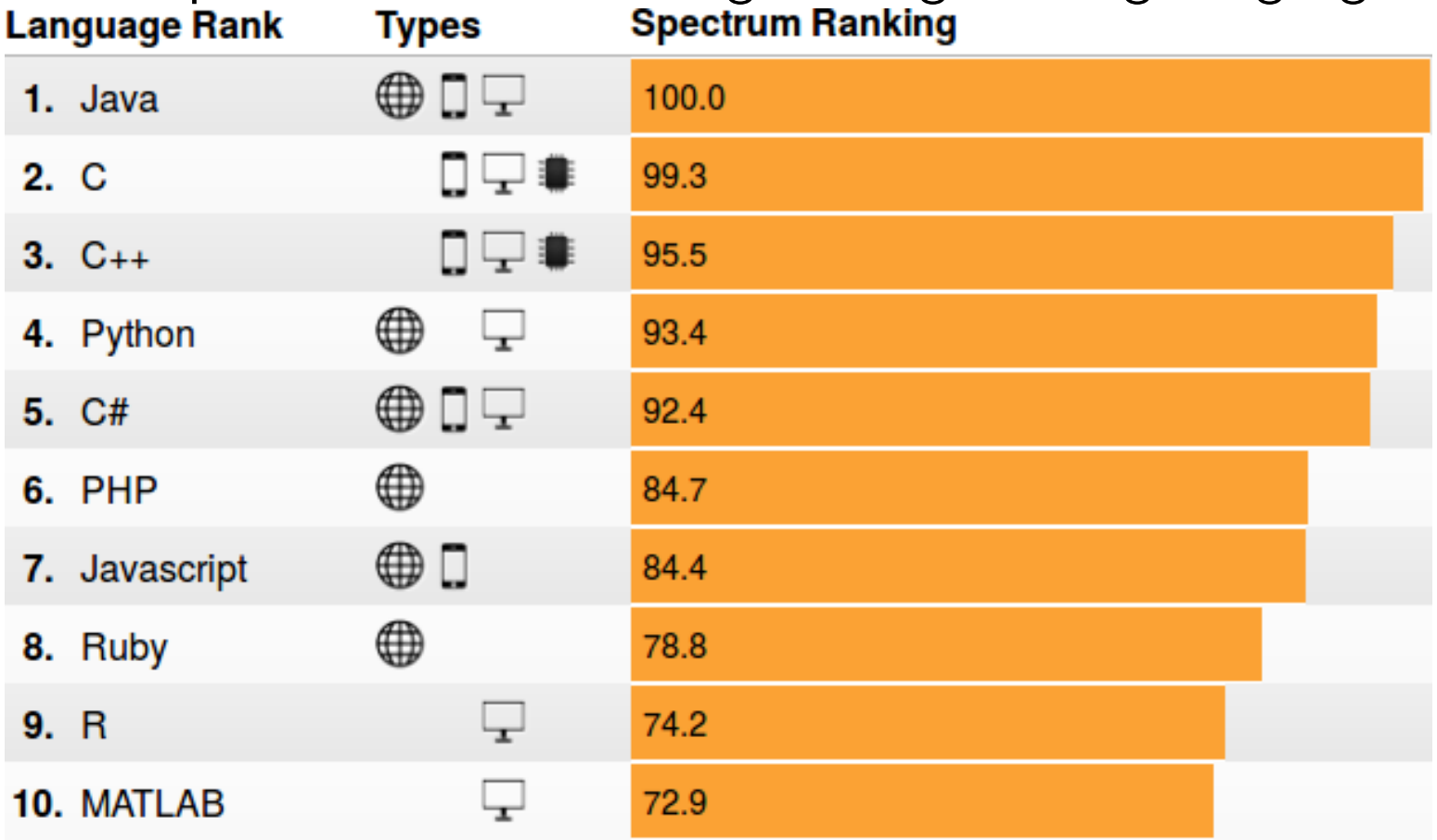


The R Language

- R is slow; if you don't know what you're doing, it's *really* slow.
- High-level scripting language.
- Syntax designed for data: models are first-class constructs, missingness is built into the core of the language, ...

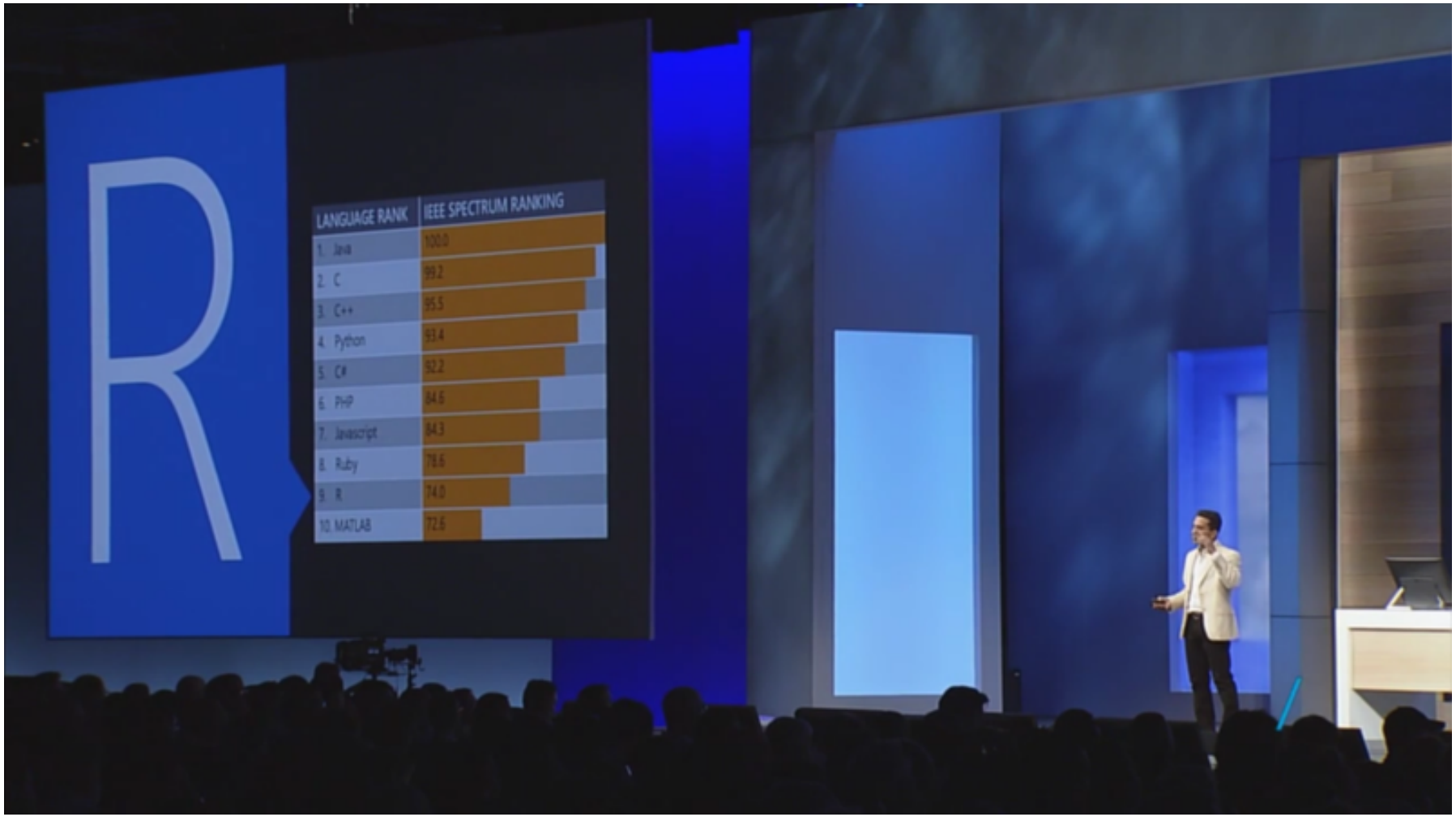
But you can't deny its popularity!

IEEE Spectrum's 2014 Ranking of Programming Languages



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"*.

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.
- There are always 100 ways to do anything, only one of them efficient.
- 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 a bug (it's you).
- Just because something is on the CRAN does not mean it's of any value (or even functional).
- Indent your code.

Comments and Advice

- Be consistent.
- Generally try to use instructive names for things.
- Make your code concise, but not obtuse (don't play golf).
- Try to avoid "super functions". Breaking up complicated ideas into modular pieces can help with readability, debugging, reusability, etc.
- Google has an R style guide: https://google-styleguide.googlecode.com/svn/trunk/google-r-style.html
- The R community has few computer scientists.

Starting R

1. Interactive: `R`
2. Batch: `Rscript`

Interacting with the Terminal

The default method in R is `show()`, which usually amounts to printing:

```
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...

```
1 "&*()" <- 3
2 "&*()"
3 "2" <- 1
4 `2` + 1
5 "    _    " <- "even unicode"
```

Assignment

```
1  myvar <- 1
2  myvar
3
4  myvar <<- 2
5  myvar
6
7  assign(x="myvar", value=3)
8  myvar
9
10 myvar = 4
11 myvar
```

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 ?+ # ctrl+c to break
6 ?"+"
```

RNG

R contains many powerful random number generators:

Beta	Binomial	Cauchy
Chi-square	Exponential	F
Gamma	Geometric	Hypergeometric
Logistic	Log Normal	Negative Binomial
Normal	Poisson	Student t
Uniform	Weibull	Wilcoxon Rank Sum

```
1 runif(5, min=0, max=10)
2 rnorm(5, mean=0, sd=1)
3 rgamma(5, shape=1)
4
5 set.seed(10)
6 runif(5)
7 set.seed(10)
8 runif(5)
```

Strings

- "Character" data (text).
- Internal storage scheme is complicated...
- Mostly works like most high-level languages.
- Can have a vector, matrix, dataframe, or list of strings.

Strings: Example 1

```
1 letters
2 toupper(letters)
3 LETTERS
4 tolower(LETTERS)
```

Strings: Example 2

```
1 x <- "Star Trek is objectively better than Star Wars"
2 strsplit(x, split=" ")
3
4 y <- unlist(strsplit(x, split=""))
5 y
6 paste(rev(y), collapse="")
```

Strings: Example 3

```
1 paste(letters, letters)
2 paste(letters, letters, sep="")
3 paste(letters, letters, sep="", collapse="")
4
5 paste(paste(letters, collapse=""), paste(LETTERS, collapse=""),
      sep="")
```

Strings: Example 4

```
1 x <- rnorm(1000, mean=10)
2
3 paste("The mean of 'x' is:", mean(x))
4
5 cat(sprintf("mean:\t%.2f\nvar:\t%.2f\n", mean(x), var(x)))
```

Regular Expressions

- `grep()`, `grepl()`
- `sub()`, `gsub()`
- Some others...

I/O

- `write.csv()`, `read.csv()`,
- `write.table()`, `read.table()`
- `scan()`
- `save()` and `load()`

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=F)
6 read.csv("x.csv", header=T)
```

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()`.

Plotting

- R is *world class* for graphics and visualization.
- Biggest package for plotting is **ggplot2** by Hadley Wickham.
- We will focus on some examples from core R...

Scatterplots

```
1 x <- 1:10
2 y <- rnorm(10)
3 plot(x, y)
4 text(x=5.5, y=0, "MY PLOT", col='red')
```

Piecharts

```
1 pie(1:5)
```

Barplots

```
1 barplot(1:5)
```

Histograms

```
1 hist(rnorm(500))
```

Saving Plots

```
1 pdf("myscatterplot.pdf")
2 plot(rnorm(4), 1:4)
3 dev.off()
4
5 png("mybarplot.png")
6 barplot(1:10)
7 dev.off()
```

Plotting

- These are all fairly "cookie-cutter".
- These can do more complicated things, but you're only making things difficult for yourself...
- Best to learn a full graphics package.
- The biggest are **ggplot2** and **lattice** (I recommend ggplot...)

Lattice: http://www.statmethods.net/advgraphs/trellis.html
ggplot2: http://www.statmethods.net/advgraphs/ggplot2.html

Packages

- Comprehensive R Archive Network (CRAN).
- 6735 packages.
- CRAN Task Views http://cran.r-project.org/web/views/

Installing Packages

From CRAN

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

From GitHub

```
1 library("devtools")
2 install_github("wrathematics/lineSampler")
```

Namespaces

All packages have namespaces:

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

Libraries

- 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...

The R Language

- Storage: logical, int, double, double complex, character (strings)
- Structures: vector, matrix, array, list, dataframe
- Caveats: (Logical) `TRUE`, `FALSE`, `NA`
- In fact, there's an `NA` for each type:
  - Integer: $-(2^{-31}-1)$
  - Double: value at address `0x7FF00000000007A2LL`
- 3 official OOP systems (none of them work like anything you're used to), several unofficial ones.
- No official support for C++; 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

```
1 Inf
2 typeof(Inf)
3 is.finite(Inf)
4 is.infinite(Inf)
5 Inf+Inf
6 1/0
```

NaN

Not a Number; numerical undefinedness.

```
1 NaN
2 typeof(NaN)
3 is.nan(NaN)
4 Inf-Inf
5 sin(Inf)
```

NULL

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

```
1 NULL
2 typeof(NULL)
3 is.null(NULL)
4 NULL+NULL
```

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)
is.vector(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 x
3
4 x[1:5, ] <- 0
5 x[7, 3] <- NA
6 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: no
- Example deque https://github.com/wrathematics/dequer

Logic

- Possible values are `TRUE`, `FALSE`, and `NA`
- Relational operations: ==, !=, <, <=, >, and >=

Logic

```
1  1==1
2  1!=1
3  1<1
4  1<=1
5
6  TRUE==FALSE
7
8  TRUE==1
9  FALSE==0
10
11 TRUE==T
12 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(T, F, F, T, T, F)
2 x==T
3 !x
```

Logic

Some very important functions for logical evaluations:

```
1 x <- c(T, F, F, T, T, F)
2 any(x)
3 all(x)
4 which(x)
5
6 y <- -5:5
7 which(y%%2==0)
8 y[which(y%%2==0)]
```

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    print(i)
3  }
4
5  x <- matrix(1:30, 10)
6  colmax <- numeric(3)
7  for (i in 1:3){
8    colmax[i] <- max(x[, i])
9  }
10 colmax
```

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 n
14 i
```

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.

apply

```
1  x <- matrix(1:30, 10)
2  x
3  apply(X=x, MARGIN=1, FUN=min)
4  apply(X=x, MARGIN=2, FUN=min)
5
6  out <- numeric(3)
7  for (i in 1:3){
8    out[i] <- min(x[, i])
9  }
10 out
```

lapply and sapply

```
1 x <- 1:5
2 lapply(X=x, FUN=sqrt)
3
4 sapply(X=x, FUN=sqrt)
```

*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

```
# No initialization
system.time({
  x <- 5000:1
  sin <- numeric(0)
  for (i in 1:length(x)){
    sin[i] <- sin(x[i])
  }
  sin
})

# With initialization
system.time({
  x <- 5000:1
  sin <- numeric(length(x))
  for (i in 1:length(x)){
    sin[i] <- sin(x[i])
  }
  sin
})
```

Loop Speeds

```
1 # lapply
2 system.time(lapply(x, sin))
3 system.time(sapply(x, sin))
4
5 # vectorized
6 system.time(sin(x))
```

Functions

- Self-contained input/output machine.
- Reusable blocks of code.
- First class objects.
- Can not modify in place*; use a list.
- *Evaluating the Design of the R language*, http://r.cs.purdue.edu/pub/ecoop12.pdf

Functions: Example 1

```
1  f <- function(x)
2  {
3    ret <- x+1
4    return(ret)
5  }
6
7  f(1)
8  f(2)
9  f(5)
10 f
```

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, ncol=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   return(list(plus=a+b, minus=a-b, times=a*b, division=a/b))
4 }
5
6 g(5, 2)
7 g(1, 0)
8 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 }
```

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

Object Inspection Tools

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

Object Inspection Tools: print()

Basic printing:

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

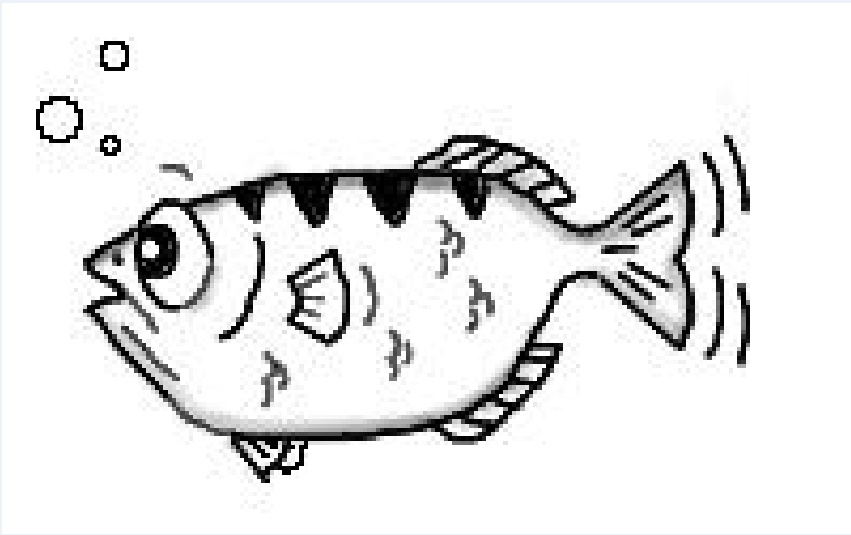
Using the Debugger

Example Debugger Interaction

```
1  > f <- function(x){y <- z+1;z <- y*2;z}
2  > f(1)
3  Error in f(1) : object 'z' not found
4  > debug(f)
5  > f(1)
6  debugging in: f(1)
7  debug at #1: {
8      y <- z + 1
9      z <- y * 2
10     z
11 }
12 Browse[2]>
13 debug at #1: y <- z + 1
14 Browse[2]>
15 Error in f(1) : object 'z' not found
16 >
```

Debugging Compiled Code

- Reasonably easy to use gdb and Valgrind (from command line).
- gdb — The GNU Debugger; general purpose debugging.
- Valgrind — Memory debugger.
- For gdb, start R interactively.
- For Valgrind, need a batch script.



Debugging with gdb

Suppose we have:

- R function: `fooR()`
- Calls the C function: `fooC()`

We can debug `fooC()` via gdb by executing the following from a shell:

```
R -d gdb
b fooC
signal 0
fooR(10)
```

Debugging with Valgrind

Put the R code you wish to profile in `myscript.r` and execute the following from a shell:

```
R -d "valgrind --tool=memcheck --leak-check=full" --vanilla <
    myscript.r
```

Getting Started

- Log in to `arronax.nics.utk.edu`
- Run `module load r/3.0.1`
- Run `cd /lustre/medusa/proj/dance/data/day_0129_csv`

Managing the Data

- `ls`
- `ls | grep Unit989`

Managing the Data

reconfigure.sh

```
#!/bin/sh

for file in `ls | grep Unit989`; do
  cat $file >> Unit989.csv
done
```

```
$ chmod +x reconfigure.sh
$ ./reconfigure.sh
$ wc -l Unit989.csv
```

Managing the Data

reconfigure.sh

```
#!/bin/sh

for folder in `ls -d */ | grep -E
    "day_[0-9][0-9][0-9][0-9]_csv"`; do
  for file in `ls $folder | grep Unit989`; do
    cat $folder/$file >> Unit989.csv
  done
done
```

```
$ chmod +x reconfigure.sh
$ ./reconfigure.sh
$ wc -l Unit989.csv
```

Loading

```
1 file <- "Unit989-20150129-000102.csv"
2 file.info(file)
3 data <- read.csv(file)
4 head(data)
5 str(data)
6 data <- read.csv(file, header=FALSE, stringsAsFactors=FALSE)
```

Inspecting

```
1 head(data)
2 tail(data)
3 dim(data)
4 str(data)
5 summary(data)
6 sd(data$V5)
7 object.size(data)
```

xts: The eXtensible Time Series package

```
1 library(xts)
```

Time Series

```
1 date <- as.POSIXlt(paste(data$V1, data$V2))
2 x <- xts(data$V4, date)
3 head(x)
4 str(x)
```

Time Series

```
1 first(x, "3 seconds")
2 first(x, "1 minute")
3 last(first(x, "1 minute"), "3 seconds")
```

Time Series

```
1 ndays(x)
2 nhours(x)
3 nminutes(x)
```

Time Series

```
1 to.daily(x)
2 to.hourly(x)
3 to.minutes30(x)
4 to.minutes15(x)
```

Time Series

```
1 endpoints(x, "hours")
2 period.apply(x, INDEX=endpoints(x, "hours"), FUN=mean)
```

Time Series

```
1 min30 <- to.minutes30(x)
2
3 plot(x)
4 plot(min30)
5
6 plot(as.zoo(min30))
7 plot(as.zoo, plot.type="single")
8 plot(as.zoo, plot.type="single", col=1:4)
```