SNU - PSIR
Introduction to R, with Hands-On Exercises

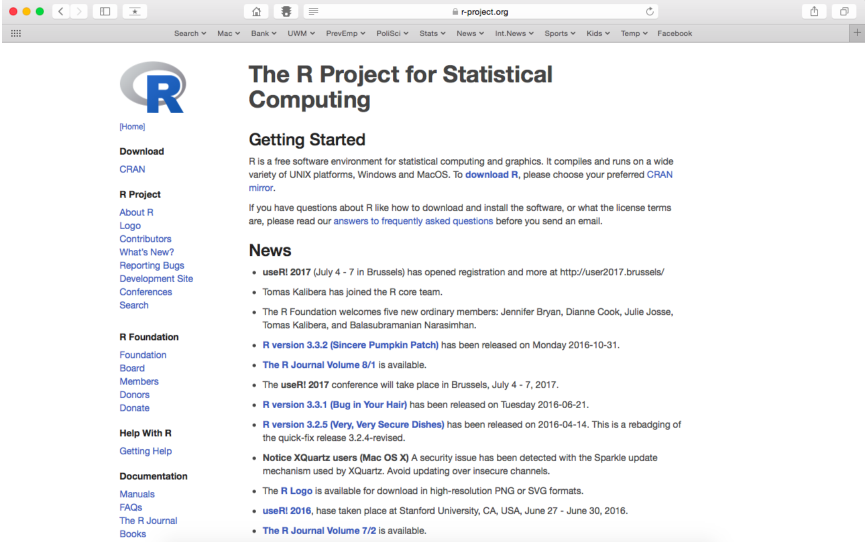
Instructor: **Hong Min Park**

Associate Professor
Department of Political Science
University of Wisconsin-Milwaukee
hmpark1@uwm.edu

Install R on your machine

- R is **free**, open-source, easily-extensible statistical software and programing environment.

- `http://www.r-project.org`
- `https://cran.r-project.org`


r-project.org
Search Mac Bank UWM PrevEmp PoliSci Stats News Int.News Sports Kids Temp Facebook
[Home]
Download
CRAN
R Project
About R
Logo
Contributors
What's New?
Reporting Bugs
Development Site
Conferences
Search
R Foundation
Foundation
Board
Members
Donors
Donate
Help With R
Getting Help
Documentation
Manuals
FAQs
The R Journal
Books
The R Project for Statistical Computing
Getting Started
R is a free software environment for statistical computing and graphics. It compiles and runs on a wide variety of UNIX platforms, Windows and MacOS. To download R, please choose your preferred CRAN mirror.
If you have questions about R like how to download and install the software, or what the license terms are, please read our answers to frequently asked questions before you send an email.
News
useR! 2017 (July 4 - 7 in Brussels) has opened registration and more at http://user2017.brussels/
Tomas Kalibera has joined the R core team.
The R Foundation welcomes five new ordinary members: Jennifer Bryan, Dianne Cook, Julie Josse, Tomas Kalibera, and Balasubramanian Narasimhan.
R version 3.3.2 (Sincere Pumpkin Patch) has been released on Monday 2016-10-31.
The R Journal Volume 8/1 is available.
The useR! 2017 conference will take place in Brussels, July 4 - 7, 2017.
R version 3.3.1 (Bug in Your Hair) has been released on Tuesday 2016-06-21.
R version 3.2.5 (Very, Very Secure Dishes) has been released on 2016-04-14. This is a rebadging of the quick-fix release 3.2.4-revised.
Notice XQuartz users (Mac OS X) A security issue has been detected with the Sparkle update mechanism used by XQuartz. Avoid updating over insecure channels.
The R Logo is available for download in high-resolution PNG or SVG formats.
useR! 2016, hase taken place at Stanford University, CA, USA, June 27 - June 30, 2016.
The R Journal Volume 7/2 is available.

cran.r-project.org

Search Mac Bank UWM PrevEmp PoliSci Stats News Int.News Sports Kids Temp Facebook

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!

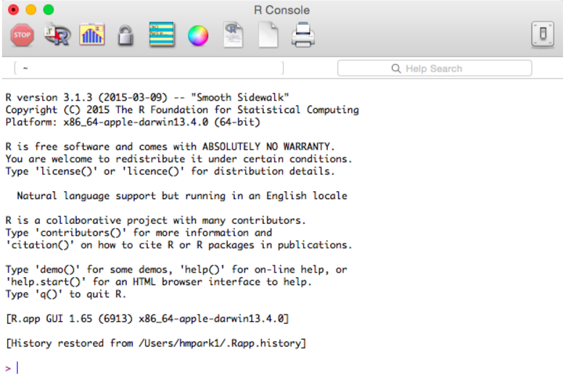
- The latest release (Monday 2016-10-31, Sincere Pumpkin Patch) R-3.3.2.tar.gz, read what's new in the latest version.
- Sources of R alpha and beta releases (daily snapshots, created only in time periods before a planned release).
- Daily snapshots of current patched and development versions are available here. Please read about new features and bug fixes before filing corresponding feature requests or bug reports.
- Source code of older versions of R is available here.
- Contributed extension packages

Questions About R

- If you have questions about R like how to download and install the software, or what the license terms are, please read our answers to frequently asked questions before you send an email.

What are R and CRAN?

R is ‘GNU S’, a freely available language and environment for statistical computing and graphics which provides a wide variety of statistical and graphical techniques: linear and nonlinear modelling, statistical tests, time series analysis, classification, clustering, etc. Please consult the R project homepage for further information.


R Console
STOP
Help Search
R version 3.1.3 (2015-03-09) -- "Smooth Sidewalk"
Copyright (C) 2015 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin13.4.0 (64-bit)
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.
Natural language support but running in an English locale
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
[R.app GUI 1.65 (6913) x86_64-apple-darwin13.4.0]
[History restored from /Users/hmpark1/.Rapp.history]
>

- Other options: SPSS, Stata, SAS, Matlab
- Why R?
  - Free
  - Community based on the statistics orientations
  - Flexible in custom programming
  - Excellent in visualization

- Other options: SPSS, Stata, SAS, Matlab
- Why R?
    - Free
    - Community based on the statistics orientations
    - Flexible in custom programming
    - Excellent in visualization
    - (more practical reason) More jobs waiting...

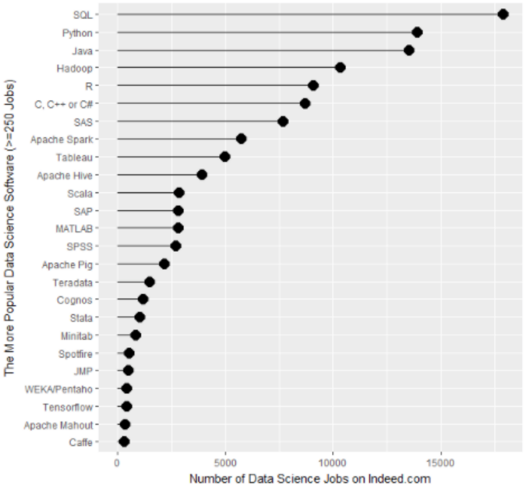


Figure 1a. The number of data science jobs for the more popular software (those with 250 jobs or more, 2/2017).

Figure 1c. Data science job trends for R (blue) and SAS (orange).

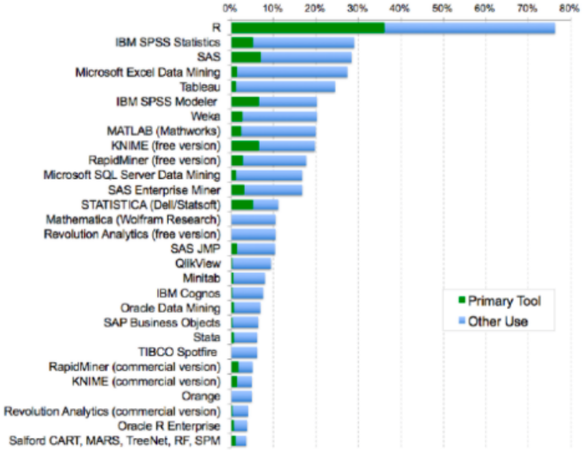


Figure 4a. Analytics tools used by respondents to the 2015 Rexer Analytics Survey. In this view, each respondent was free to check multiple tools.

Basics

- When R starts, the command prompt $>$ is shown.
  - This is where we type commands to be processed by R.
  - And, this happens when we hit the ENTER key.

Basics

- When R starts, the command prompt $>$ is shown.
  - This is where we type commands to be processed by R.
  - And, this happens when we hit the ENTER key.
- Some preliminaries on entering commands.
  - Expressions and commands are case-sensitive.
  - Command lines do not need to be separated by any special character like a semicolon as in SAS.
  - Anything following the pound character (#) is ignored - usually used as a comment.
  - You can use the arrow keys (up and down) on the keyboard to scroll back to previous commands.

Basics

- When R starts, the command prompt $>$ is shown.
  - This is where we type commands to be processed by R.
  - And, this happens when we hit the ENTER key.
- Some preliminaries on entering commands.
  - Expressions and commands are case-sensitive.
  - Command lines do not need to be separated by any special character like a semicolon as in SAS.
  - Anything following the pound character (#) is ignored - usually used as a comment.
  - You can use the arrow keys (up and down) on the keyboard to scroll back to previous commands.
- Help?
  - Type `?`, `help()`, `help.search()`, or `help.start()`
  - Or, use `www.rseek.org` (web-based search powered by Google)

Start R

- One way to run R is to have a script file open in an separate editor and run it periodically from the R window.
    - Why???

Start R

- One way to run R is to have a script file open in an separate editor and run it periodically from the R window.
  - Why???
  - RStudio becomes useful here (www.rstudio.com).

Start R

- One way to run R is to have a script file open in an separate editor and run it periodically from the R window.
  - Why???
  - RStudio becomes useful here (www.rstudio.com).
- If no path is specified to the script file, R assumes that the file is located in the current working directory.
  - Working directories can be viewed and changed

```
> getwd()
[1] "/Users/hmpark1"
> setwd("/Users/hmpark1/Dropbox/TALK/2018SNUmethods/R")
> dir()
[1] "Rfor2018SNU.R"
> getwd()
[1] "/Users/hmpark1/Dropbox/TALK/2018SNUmethods/R"
```

R as calculator

- The simplest usage of R is performing basic math operations.

```
> 3 + 2
[1] 5
> 3^2
[1] 9
> (2-4)*6
[1] -12
> 2-4*6
[1] -22
```

- Note that the answer is printed starting with a `[1]`.

R as calculator

- The simplest usage of R is performing basic math operations.

```
> 3 + 2
[1] 5
> 3^2
[1] 9
> (2-4)*6
[1] -12
> 2-4*6
[1] -22
```

  - Note that the answer is printed starting with a `[1]`.

- An error message is generated for an "illegitimate" command.

```
> 2 ^^ 2
Error: unexpected '^' in "2 ^^"
```

- $+$, instead of $>$, will appear if the command is "incomplete".

```
> 2 *
+ 3
[1] 6
```

- $+$, instead of $>$, will appear if the command is "incomplete".

```
> 2 *
+ 3
[1] 6
```

- Functions

```
> sqrt(4)
[1] 2
> exp(3)
[1] 20.08554
> log(10)
[1] 2.302585
```

- +, instead of >, will appear if the command is "incomplete".

```
> 2 *
+ 3
[1] 6
```

- Functions

```
> sqrt(4)
[1] 2
> exp(3)
[1] 20.08554
> log(10)
[1] 2.302585
```

- Variables

```
> x <- 5   ## a command, giving x the value 5
> x = 5    ## a command, giving x the value 5
> x == 5   ## a question, with answers TRUE and FALSE
[1] TRUE
```

Data

- We can use c() to enter data.

```
> height <- c(5.2, 5.5, 6.2, 6.0, 5.7, 5.8, 6.4, 5.6)
> height
[1] 5.2 5.5 6.2 6.0 5.7 5.8 6.4 5.6
> heightX <- c(5.2, 5.5, 6.2, 6.0)
> heightY <- c(5.8, 5.2, 6.0, 5.0)
> c(heightX, heightY)
[1] 5.2 5.5 6.2 6.0 5.8 5.2 6.0 5.0
```

Data

- We can use c() to enter data.

```
> height <- c(5.2, 5.5, 6.2, 6.0, 5.7, 5.8, 6.4, 5.6)
> height
[1] 5.2 5.5 6.2 6.0 5.7 5.8 6.4 5.6
> heightX <- c(5.2, 5.5, 6.2, 6.0)
> heightY <- c(5.8, 5.2, 6.0, 5.0)
> c(heightX, heightY)
[1] 5.2 5.5 6.2 6.0 5.8 5.2 6.0 5.0
```

- Later, we will encounter data that look like an Excel spreadsheet. And, it is useful to know that this is an extension of c(), just like vector being a special case of matrix.

- Data vectors have a type.
  - One restriction is that all the values have the same type.
  - This can be numeric, as in `height` or character strings, as in

```
> park.kim <- c("Hong Min", "Yu Ha", "Chloe", "Steven")
> park.kim
[1] "Hong Min" "Yu Ha"    "Chloe"    "Steven"
```

  - If we mix the type within a data vector, the data will be coerced into a common type, which is usually a character.

```
> mix <- c("Park", 1)
> mix
[1] "Park" "1"
```

- A data vector can have its entries named.

```
> names(park.kim) <- c("dad", "mom", "daughter", "son")
> park.kim
       dad         mom    daughter         son
"Hong Min"    "Yu Ha"     "Chloe"    "Steven"
```

- A data vector can have its entries named.

```
> names(park.kim) <- c("dad", "mom", "daughter", "son")
> park.kim
       dad          mom    daughter         son
"Hong Min"    "Yu Ha"    "Chloe"    "Steven"
```

- Functions can be used on a data vector

```
> sum(height)
[1] 46.4
> length(height)
[1] 8
> sort(height)
[1] 5.2 5.5 5.6 5.7 5.8 6.0 6.2 6.4
> min(height)
[1] 5.2
> max(height)
[1] 6.4
```

- Column and row vectors
  - R uses the `c()` command to create a vector with values.
  - `cbind()` concatenates the objects by columns.

```
> CXY <- cbind(heightX, heightY)
> CXY
     heightX heightY
[1,]     5.2     5.8
[2,]     5.5     5.2
[3,]     6.2     6.0
[4,]     6.0     5.0
```

  - `rbind()` concatenates the objects by rows.

```
> RXY <- rbind(heightX, heightY)
> RXY
        [,1] [,2] [,3] [,4]
heightX  5.2  5.5  6.2    6
heightY  5.8  5.2  6.0    5
```

- We can get summary statistics on the data using `summary()` and we can determine the dimensionality using `dim()` or `nrow()` and `ncol()` commands.

```
> summary(CXY)
    heightX          heightY
 Min.   :5.200   Min.   :5.00
 1st Qu.:5.425   1st Qu.:5.15
 Median :5.750   Median :5.50
 Mean   :5.725   Mean   :5.50
 3rd Qu.:6.050   3rd Qu.:5.85
 Max.   :6.200   Max.   :6.00
> dim(CXY) # rows x columns
[1] 4 2
> ncol(CXY)
[1] 2
> nrow(CXY)
[1] 4
```

- Observe the difference: `c()` vs. `cbind()` and `rbind()`.

```
> a <- c(1,2,3)
> b <- cbind(1,2,3)
> c <- rbind(1,2,3)
> a
[1] 1 2 3
> b
     [,1] [,2] [,3]
[1,]    1    2    3
> c
     [,1]
[1,]    1
[2,]    2
[3,]    3
> dim(a)
NULL
> dim(b)
[1] 1 3
> dim(c)
[1] 3 1
```

```
> cbind(a,a)   ## a <- c(1,2,3)
     a a
[1,] 1 1
[2,] 2 2
[3,] 3 3
> cbind(b,b)   ## b <- cbind(1,2,3)
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    2    3    1    2    3
> cbind(c,c)   ## c <- rbind(1,2,3)
     [,1] [,2]
[1,]    1    1
[2,]    2    2
[3,]    3    3
```

```
> rbind(a,a)   ## a <- c(1,2,3)
  [,1] [,2] [,3]
a    1    2    3
a    1    2    3
> rbind(b,b)   ## b <- cbind(1,2,3)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    1    2    3
> rbind(c,c)   ## c <- rbind(1,2,3)
     [,1]
[1,]    1
[2,]    2
[3,]    3
[4,]    1
[5,]    2
[6,]    3
```

- Matrix can also be constructed by itself:

```
> C <- matrix(c(1,2,3,4), nrow=2)
> C
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

- Matrix can also be constructed by itself:

```
> C <- matrix(c(1,2,3,4), nrow=2)
> C
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

- R also does matrix algebra.

```
> D <- matrix(c(5,6,7,8), nrow=2)
> C*D     ## inner product
     [,1] [,2]
[1,]    5   21
[2,]   12   32
> C %*% D     ## multiplication
     [,1] [,2]
[1,]   23   31
[2,]   34   46
```

- Creating structured data
  - Simple sequences

```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
> rev(1:10)
 [1] 10  9  8  7  6  5  4  3  2  1
> 10:1
 [1] 10  9  8  7  6  5  4  3  2  1
```

  - Sequence by step size and starting/ending points

```
> seq(1, 9, by=2)    ## odd numbers
[1] 1 3 5 7 9
> seq(1, 9, length=5)
[1] 1 3 5 7 9
```

  - A vector of repeated numbers

```
> rep(1, 10)
[1] 1 1 1 1 1 1 1 1 1 1
> rep(1:3, 3)
[1] 1 2 3 1 2 3 1 2 3
```

- Accessing data by using indices
    - Each observation $x_1, x_2, \cdots, x_n$ is referred to by its index using square brackets, as in `x[1]`, `x[2]`, $\cdots$, `x[n]`.

- Accessing data by using indices
  - Each observation $x_1, x_2, \cdots, x_n$ is referred to by its index using square brackets, as in `x[1]`, `x[2]`, $\cdots$, `x[n]`.
  - eBay's weekly stock price for the last two months:

```
> ebay <- c(88.8, 88.3, 90.2, 93.5, 95.2, 94.7, 99.2, 99.4, 101.6)
> length(ebay)
[1] 9
```

  - We can get the first and last values directly:

```
> ebay[1]     ## first value
[1] 88.8
> ebay[length(ebay)]     ## last value
[1] 101.6
```

- Accessing data by using indices
  - Each observation $x_1, x_2, \cdots, x_n$ is referred to by its index using square brackets, as in `x[1]`, `x[2]`, $\cdots$, `x[n]`.
  - eBay's weekly stock price for the last two months:

```
> ebay <- c(88.8, 88.3, 90.2, 93.5, 95.2, 94.7, 99.2, 99.4, 101.6)
> length(ebay)
[1] 9
```

  - We can get the first and last values directly:

```
> ebay[1]     ## first value
[1] 88.8
> ebay[length(ebay)]     ## last value
[1] 101.6
```

  - Slicing

```
> ebay[1:4]
[1] 88.8 88.3 90.2 93.5
> ebay[c(1,5,9)]
[1]  88.8  95.2 101.6
```

- Using indices (Cont'd)
  - Negative indices: if `i` is negative and no less than `-n`, then a useful convention is employed to return all but the i^{th} value of the vector.

```
> ebay[-1]    ## all but the first
[1]  88.3  90.2  93.5  95.2  94.7  99.2  99.4 101.6
> ebay[-c(1:4)]    ## all but the first - fourth
[1]  95.2  94.7  99.2  99.4 101.6
> ebay[-(c(1:4,6))]    ## all but the first - fourth, and sixth
[1]  95.2  99.2  99.4 101.6
```

- Using indices (Cont'd)
  - Negative indices: if `i` is negative and no less than `-n`, then a useful convention is employed to return all but the i^{th} value of the vector.

```
> ebay[-1]    ## all but the first
[1]  88.3  90.2  93.5  95.2  94.7  99.2  99.4 101.6
> ebay[-c(1:4)]    ## all but the first - fourth
[1]  95.2  94.7  99.2  99.4 101.6
> ebay[-(c(1:4,6))]    ## all but the first - fourth, and sixth
[1]  95.2  99.2  99.4 101.6
```

- Accessing data by using names: when data vector has names, then the values can be accessed by their names.

```
> x <- c(1,2,9)
> names(x) <- c("one", "two", "three")
> x["two"]
two
  2
```

- Assigning values to data vector
  - The simplest case, `x[i] <- a`

```
> ebay[1] <- 76.0
> ebay
[1]  76.0  88.3  90.2  93.5  95.2  94.7  99.2  99.4 101.6
```

- Assigning values to data vector
  - The simplest case, `x[i] <- a`

```
> ebay[1] <- 76.0
> ebay
[1]  76.0  88.3  90.2  93.5  95.2  94.7  99.2  99.4 101.6
```

  - We can assign more than one value at a time.

```
> ebay[c(10:13)] <- c(97.0, 99.3, 102.0, 101.8)
> ebay
 [1]  76.0  88.3  90.2  93.5  95.2  94.7  99.2  99.4 101.6  97.0
[11]  99.3 102.0 101.8
```

- What about using indices for matrix?

```
> ebayT <- matrix(ebay[-1], nrow=3)
> ebayT
     [,1] [,2]  [,3]  [,4]
[1,] 88.3 95.2  99.4  99.3
[2,] 90.2 94.7 101.6 102.0
[3,] 93.5 99.2  97.0 101.8
> ebayT[1,2]
[1] 95.2
> ebayT[1,]
[1] 88.3 95.2 99.4 99.3
> ebayT[,3]
[1]  99.4 101.6  97.0
```

Logical values

- Logical values: R expressions which involves just values of TRUE or FALSE

```
> ebay > 100
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
[11] FALSE  TRUE  TRUE
```

Logical values

- Logical values: R expressions which involves just values of TRUE or FALSE

```
> ebay > 100
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
[11] FALSE  TRUE  TRUE
```

- We can select several entries from data by using logical values.

```
> ebay[ebay>100] # values greater than 100
[1] 101.6 102.0 101.8
> which(ebay>100) # which indices
[1]  9 12 13
> ebay[c(9,12,13)]
[1] 101.6 102.0 101.8
```

- `sum` function for logical vectors will add up all the `TRUE` values as 1 and all the `FALSE` values as 0.

```
> sum(ebay > 100)
[1] 3
```

- `sum` function for logical vectors will add up all the `TRUE` values as 1 and all the `FALSE` values as 0.

```
> sum(ebay > 100)
[1] 3
```

- Logical operators include $<, <=, >, >=, ==, != , \&, |$.

```
> x > 3 & x < 7
> x > 3 | x < 7
> x == 7
> x != 7
> x %in% c(1:3, 7) # range of values use %in%
```

Missing values

- R uses NA to indicate a missing value.

```
> shuttle <- c(0,1,0,NA,0,0)
> shuttle
[1]  0  1  0 NA  0  0
> is.na(shuttle)
[1] FALSE FALSE FALSE  TRUE FALSE FALSE
> sum(shuttle)
[1] NA
> sum(shuttle, na.rm=TRUE)
[1] 1
```

Missing values

- R uses NA to indicate a missing value.

```
> shuttle <- c(0,1,0,NA,0,0)
> shuttle
[1]  0  1  0 NA  0  0
> is.na(shuttle)
[1] FALSE FALSE FALSE  TRUE FALSE FALSE
> sum(shuttle)
[1] NA
> sum(shuttle, na.rm=TRUE)
[1] 1
```

- We can recode values in data ($1 \rightarrow 0, 2 \rightarrow 1, 8\&9 \rightarrow \mathtt{NA}$).

```
> test <- c(1,1,2,1,1,8,1,2,1,9,1,8,2,1,9,1,2,9,9,1)
> test.new <- replace(test, test==8 | test==9, NA)
> test.new <- replace(test.new, test.new==1, 0)
> test.new <- replace(test.new, test.new==2, 1)
> test.new
 [1]  0  0  1  0  0 NA  0  1  0 NA  0 NA  1  0 NA  0  1 NA NA  0
```

Table

- The `table` command computes the frequency of variable values.

```
> test
 [1] 1 1 2 1 1 8 1 2 1 9 1 8 2 1 9 1 2 9 9 1
> t <- table(test)
> t
test
 1  2  8  9
10  4  2  4
```

Table

- The `table` command computes the frequency of variable values.

```
> test
 [1] 1 1 2 1 1 8 1 2 1 9 1 8 2 1 9 1 2 9 9 1
> t <- table(test)
> t
test
 1  2  8  9
10  4  2  4
```

- What if the variable has some `NA`'s?

```
> table(test.new)
test.new
 0  1
10  4
> table(test.new, useNA = "always")
test.new
   0    1 <NA>
  10    4    6
```

- The `table` command produced so-called "one-way" table.
- After the break, we will learn data (and `data.frame` in R). Then, the `table` command will become much more useful – enabling us to navigate "two-way" tables.

```
> grade <- c("A", "B", "A", "A", "C")
> attend <- c("good", "good", "good", "bad", "bad")
> table(grade, attend)
     attend
grade bad good
    A   1    2
    B   0    1
    C   1    0
```

Loop

- It may be necessary to loop through an index and perform an operation at each iteration. The most common format is:

  `for (name in vec) { commands }`

  - Most commonly, `vec` is an integer range, like 1 : 20.
  - So, `name` iterates through those integers
  - And, `commands` uses `name` as a variable

```
> for (i in 1:5) {
+   print(i)
+ }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

Control statement

- R has an if-else statement in the format of:

  `if (logical expr) { commands 1 } else { commands 2 }`

  - If the logical expression in `logical expr` is TRUE, execute commands in `commands 1`.
  - Otherwise, execute commands in `commands 2`.

```
> last.name <- "park"
> if (last.name == "park") {
+   print("yay!!!")
+ } else {
+   print("hmm..")
+ }
[1] "yay!!!"
```

Function `function`

- R allows the user to write functions. The format for creating the function named `newfn` is:

`newfn <- function(` <u>`var1`</u>`,` <u>`var2`</u>`, ··· ) {` <u>`my arguments`</u> `}`

  - This new function takes <u>`var1`</u>, <u>`var2`</u>, ··· as input.
  - It returns the value of the last command in <u>`my arguments`</u>.

```
> newfn <- function(a) {
+   b <- 2*a + 5
+   return(b)
+ }
> newfn(2)
[1] 9
```

More on R

- Managing the work environment
  - If R sessions run long enough, there may by more variables than you can remember.
  - The `ls()` and `objects()` functions will list all the objects (variables, functions, etc.).
  - To reduce the work environment, we can use the functions, `rm()` or `remove()`.

More on R

- Managing the work environment
  - If R sessions run long enough, there may by more variables than you can remember.
  - The `ls()` and `objects()` functions will list all the objects (variables, functions, etc.).
  - To reduce the work environment, we can use the functions, `rm()` or `remove()`.
- Reading data
  - If the data is already in some format, R can read it in.
  - `read.table()` reads a file in table format and creates a data frame from it, with cases corresponding to lines and variables to fields in the files.

More on R

- Managing the work environment
  - If R sessions run long enough, there may by more variables than you can remember.
  - The `ls()` and `objects()` functions will list all the objects (variables, functions, etc.).
  - To reduce the work environment, we can use the functions, `rm()` or `remove()`.
- Reading data
  - If the data is already in some format, R can read it in.
  - `read.table()` reads a file in table format and creates a data frame from it, with cases corresponding to lines and variables to fields in the files.
  - The package `foreign` allows us to use Stata or SPSS files by using the functions, `read.dta()` or `read.spss()`.

- Packages
  - R is designed to be a small code of kernel, with additional functionality provided by external packages.
  - An example is the `foreign` package or the dataset for textbook.
  - Most packages are not loaded by default and can be selectively loaded either using `library()` or `require()`.
  - In some cases, packages should be installed first by users.

```
> install.packages("UsingR")
> require(UsingR)
Loading required package: UsingR
```

- Packages
  - R is designed to be a small code of kernel, with additional functionality provided by external packages.
  - An example is the `foreign` package or the dataset for textbook.
  - Most packages are not loaded by default and can be selectively loaded either using `library()` or `require()`.
  - In some cases, packages should be installed first by users.

```
> install.packages("UsingR")
> require(UsingR)
Loading required package: UsingR
```

  - It could be easier if we use "Package Installer" on the menu. $\Rightarrow$ my recommendation for you?

Too Basic or Too Much?

Will Come Back After SHORT Break

data.frame object

- A `data.frame` object in R looks just like a matrix.
  - The standard is to put data for one sample across a row and covariates (or variables) as columns.
  - (*technical note*) On one level, a `data.frame` is a list, in which each component corresponds to a variable (i.e. the vector of values of a given variable).

```
> park.family <- data.frame(first = c("Hong Min", "Yu Ha", "Chloe", "Steven"),
+                           age = c(41, 36, 6, 4),
+                           gender = c("M", "F", "F", "M"),
+                           spicy = c("hate", "like", "hate", "hate"))
> park.family
     first age gender spicy
1 Hong Min  41      M  hate
2    Yu Ha  36      F  like
3    Chloe   6      F  hate
4   Steven   4      M  hate
```

- Entries are indexed just like a matrix.

```
> park.family[1,2]
[1] 41
```

- Entries are indexed just like a matrix.

```
> park.family[1,2]
[1] 41
```

- Variable names can be used.

```
> names(park.family)
[1] "first"  "age"    "gender" "spicy"
> park.family$age
[1] 41 36  6  4
```

- Entries are indexed just like a matrix.

```
> park.family[1,2]
[1] 41
```

- Variable names can be used.

```
> names(park.family)
[1] "first"  "age"    "gender" "spicy"
> park.family$age
[1] 41 36  6  4
```

- The big difference is that it may contain categorical data as well as numeric.

```
> class(park.family$age)
[1] "numeric"
> class(park.family$first)
[1] "factor"
```

  - Note also that character vectors are always stored as a "factor" instead of "string."

- More variables can be added to a `data.frame`.

```
> park.family$food <- c("noodle", "soup", "pasta", "salad")
> park.family
     first age gender spicy   food
1 Hong Min  41      M  hate noodle
2    Yu Ha  36      F  like   soup
3    Chloe   6      F  hate  pasta
4   Steven   4      M  hate  salad
```

  - All the variables should have the same lengths.

- More observations can be added, too.

```
> more <- data.frame(first = c("Addy", "Bunny"),
+                    age = c(6, 4),
+                    gender = c("F", "M"),
+                    spicy = c(NA, NA),
+                    food = c(NA, NA))
> park.family <- rbind(park.family, more)
> park.family
     first age gender spicy   food
1 Hong Min  41      M  hate noodle
2    Yu Ha  36      F  like   soup
3    Chloe   6      F  hate  pasta
4   Steven   4      M  hate  salad
5     Addy   6      F  <NA>   <NA>
6    Bunny   4      M  <NA>   <NA>
```

  - A new `data.frame`, instead of just a matrix, should be added in order to preserve the data types (i.e. numeric vs. factor).
  - And, as before, the dimension is important.

Selecting and sorting `data.frame`

- Commonly, we will want to select those rows (or observations) in a `data.frame` in which one of the variables has specific values.
  - For example, the entries in `park.family` with age $\geq$ 18 are found:

```
> adults <- park.family[park.family$age >= 18,]
> adults
     first age gender spicy   food
1 Hong Min  41      M  hate noodle
2    Yu Ha  36      F  like   soup
```

  - Note that logical values are used in the place for the "row" index, and logical operators with multiple conditions can also be used

```
> a.m.cond <- park.family$age >= 18 & park.family$gender == "M"
> a.m <- park.family[a.m.cond,]
> a.m
     first age gender spicy   food
1 Hong Min  41      M  hate noodle
```

- Often data are better viewed when sorted.

```
> order(park.family$age)
[1] 4 6 3 5 2 1
```

  - Note that we have raw numbers instead of values themselves. And, this can be used as an index.

```
> park.family[order(park.family$age),]
     first age gender spicy   food
4   Steven   4      M  hate  salad
6    Bunny   4      M  <NA>   <NA>
3    Chloe   6      F  hate  pasta
5     Addy   6      F  <NA>   <NA>
2    Yu Ha  36      F  like   soup
1 Hong Min  41      M  hate noodle
```

- Often data are better viewed when sorted.

```
> order(park.family$age)
[1] 4 6 3 5 2 1
```

  - Note that we have raw numbers instead of values themselves. And, this can be used as an index.

```
> park.family[order(park.family$age),]
     first age gender spicy   food
4   Steven   4      M  hate  salad
6    Bunny   4      M  <NA>   <NA>
3    Chloe   6      F  hate  pasta
5     Addy   6      F  <NA>   <NA>
2    Yu Ha  36      F  like   soup
1 Hong Min  41      M  hate noodle
```

  - What about the reverse order? Try:

```
> park.family[rev(order(park.family$age)),]
```

Two-way table

- Two-way table will be very useful to navigate the relationship among variables in a `data.frame`.

```
> table(park.family$gender, park.family$spicy)

  hate like
F    1    1
M    2    0
```

  - TIP: It is useful for categorical variables, but not for continuous variables. So, sometimes, you will want to transform variables:

```
> park.family$adult <- rep(0, 6)   ## create new variable
> park.family$adult <- replace(park.family$adult,
+                              park.family$age >= 18, 1)
> table(park.family$spicy, park.family$adult)

     0 1
hate 2 1
like 0 1
```

Read data

- By default, R uses the `read.table` command to read a variety of delimited files.

```
> h.pol <- read.table("ftp://k7moa.com/wf1/house_polarization_46_114.txt")
> head(h.pol)
  V1   V2    V3    V4    V5    V6    V7     V8     V9    V10    V11
1 46 1879 0.737 0.145 0.104 0.097 0.000 -0.030 -0.015 -0.378 -0.022
2 47 1881 0.743 0.134 0.114 0.084 0.003  0.008 -0.017 -0.392 -0.052
3 48 1883 0.690 0.222 0.265 0.132 0.000 -0.081 -0.016 -0.345 -0.033
4 49 1885 0.722 0.157 0.170 0.128 0.003 -0.058 -0.006 -0.373 -0.036
5 50 1887 0.741 0.173 0.200 0.118 0.006 -0.027 -0.017 -0.383 -0.078
6 51 1889 0.784 0.116 0.140 0.089 0.000  0.003 -0.012 -0.407 -0.085
    V12    V13   V14    V15   V16    V17    V18    V19    V20    V21
1 0.359 -0.044 0.363 -0.045 0.233 -0.025 -0.320 -0.041 -0.431 -0.003
2 0.351 -0.010 0.355 -0.007 0.310 -0.043 -0.307 -0.064 -0.464 -0.041
3 0.345  0.011 0.352  0.016 0.272 -0.046 -0.265 -0.015 -0.465 -0.059
4 0.350  0.034 0.357  0.040 0.231 -0.071 -0.282 -0.025 -0.473 -0.048
5 0.358  0.036 0.356  0.042 0.373 -0.030 -0.287 -0.091 -0.483 -0.064
6 0.377  0.054 0.376  0.058 0.389  0.011 -0.314 -0.116 -0.493 -0.057
```

The format of the Polarization Data files is:

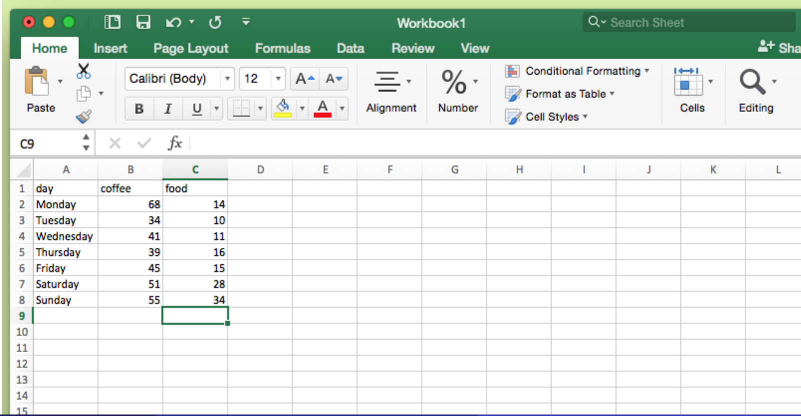
```
 1.  Congress Number
 2.  First Year of the Congress
 3.  Difference in Party Means - first dimension
 4.  Proportion Moderates
 5.  Proportion of moderate Democrats (-0.25 to +0.25)
 6.  Proportion of moderate Republicans (-0.25 to +0.25)
 7.  Overlap
 8.  Chamber Mean - first dimension
 9.  Chamber Mean - second dimension
10.  Democratic Party Mean - first dimension
11.  Democratic Party Mean - second dimension
12.  Republican Party Mean - first dimension
13.  Republican Party Mean - second dimension
14.  Northern Republican Mean - first dimension
15.  Northern Republican Mean - second dimension
16.  Southern Republican Mean - first dimension
17.  Southern Republican Mean - second dimension
18.  Northern Democrat Mean - first dimension
19.  Northern Democrat Mean - second dimension
20.  Southern Democrat Mean - first dimension
21.  Southern Democrat Mean - second dimension
```

- We choose variables that we need, and add variable names:

```
> h.pol.sub <- h.pol[,c(1,2,3,7)]
> names(h.pol.sub) <- c("cong", "year", "partydiff", "overlap")
```

- We can move between R and Excel.
  - In Excel, save the spreadsheet as a .csv file.
  - Then, you can read it in R

```
> coffee <- read.csv("excelData.csv", header=TRUE)
```



- Different types of data can be read by using packages

```
> require(foreign)
> gpa <- read.dta("http://fmwww.bc.edu/ec-p/data/wooldridge/gpa1.dta")
> head(gpa)
  age soph junior senior senior5 male campus business engineer colGPA
1  21    0      0      1       0    0      0        1        0    3.0
2  21    0      0      1       0    0      0        1        0    3.4
3  20    0      1      0       0    0      0        1        0    3.0
4  19    1      0      0       0    1      1        1        0    3.5
5  20    0      1      0       0    0      0        1        0    3.6
6  20    0      0      1       0    1      1        1        0    3.0
:
:
```

- Use appropriate command for different types of data.

Stata	`read.dta`
SPSS	`read.spss`
SAS	`read.ssd`

Write data

- We can store the data file on our computers.

```
> write.csv(park.family, "park.csv", row.names=FALSE)
> write.dta(park.family, "park.dta")
```

  - It is very important that you include directory information within the file name, unless you have already specified working directory.

    PC *C:/Documents and Settings/userName/Desktop*
    Mac */Users/userName/Desktop*

  - And, this applies to reading data from our computers.

Merge data

- We will need to merge different data files into one.

```
> fruit <- data.frame(day=c("Monday", "Tuesday", "Wednesday",
+                           "Thursday", "Friday", "Saturday"),
+                     fruit=c(6, 5, 6, 7, 7, 7))
> shop <- merge(coffee, fruit, by.x="day", by.y="day", all=TRUE)
> shop
        day coffee food fruit
1    Friday     45   15     7
2    Monday     68   14     6
3  Saturday     51   28     7
4    Sunday     55   34    NA
5  Thursday     39   16     7
6   Tuesday     34   10     5
7 Wednesday     41   11     6
```

Merge data

- We will need to merge different data files into one.

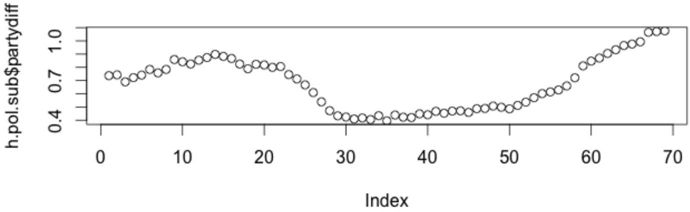
```
> fruit <- data.frame(day=c("Monday", "Tuesday", "Wednesday",
+                           "Thursday", "Friday", "Saturday"),
+                     fruit=c(6, 5, 6, 7, 7, 7))
> shop <- merge(coffee, fruit, by.x="day", by.y="day", all=TRUE)
> shop
        day coffee food fruit
1    Friday     45   15     7
2    Monday     68   14     6
3  Saturday     51   28     7
4    Sunday     55   34    NA
5  Thursday     39   16     7
6   Tuesday     34   10     5
7 Wednesday     41   11     6
```

- What if we change the option `all=FALSE`?

The plot function

- When it comes to figures in R, we will first need to utilize some commands that are automatically loaded in the base package.
  - Among others, the plot function performs almost all tasks.

```
> plot(h.pol.sub$partydiff)
```



  - It plots the h.pol.sub$partydiff values against a default index.

Scatter plot

- We are more often interested in bivariate relationships.

```
> plot(h.pol.sub$partydiff, h.pol.sub$overlap)
```

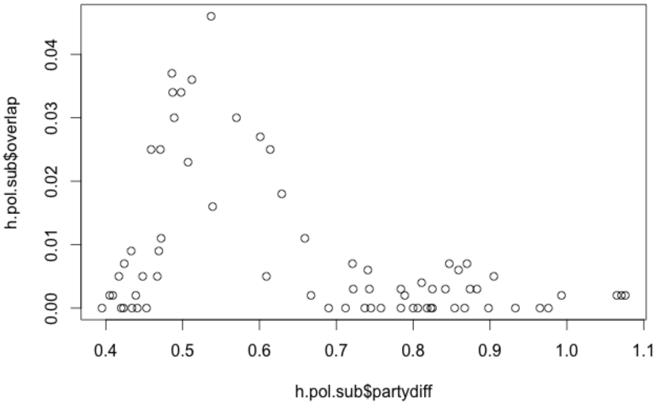


Figure construction

- There are a good number of options that can be combined in a number of ways to create figures that suit our specific needs.
  - **Coordinate system**: Our `x` and `y` inside the `plot` function must be of the same length.

```
> length(h.pol.sub$partydiff)
[1] 69
> length(h.pol.sub$overlap)
[1] 69
```

Figure construction

- There are a good number of options that can be combined in a number of ways to create figures that suit our specific needs.
  - **Coordinate system**: Our `x` and `y` inside the `plot` function must be of the same length.

```
> length(h.pol.sub$partydiff)
[1] 69
> length(h.pol.sub$overlap)
[1] 69
```

  - **Plot types**: We will want not only points but also other types.

`type="p"`	points (default)
`type="l"`	lines
`type="o"`	points and lines overlaid
`type="h"`	histogram-like vertical lines
`type="s"`	stair-step-like lines
`type="n"`	nothing

- **Axes**: It is possible to turn off the axes, to adjust the coordinate space, and to create our own labels for the axes.

`axes=F`	turn off the axes that are automatically given (better if used with the `axis` command)
`xlim=c(), ylim=c()`	expand the space from the R default
`xlab="", ylab=""`	create labels for the x- and y-axis

- **Axes**: It is possible to turn off the axes, to adjust the coordinate space, and to create our own labels for the axes.

`axes=F`	turn off the axes that are automatically given (better if used with the `axis` command)
`xlim=c(), ylim=c()`	expand the space from the R default
`xlab="", ylab=""`	create labels for the x- and y-axis

- **Style**: We can adjust the style in the figure.

`lty=`	select the line type (solid, dashed, dotted, short-long dashed, etc.)
`lwd=`	select the line width (fat or skinny)
`pch=`	select the plotting symbol - number (`pch=1`) or letter (`pch="R"`)
`col=`	select the color

- **Axes**: It is possible to turn off the axes, to adjust the coordinate space, and to create our own labels for the axes.

`axes=F`	turn off the axes that are automatically given (better if used with the `axis` command)
`xlim=c(), ylim=c()`	expand the space from the R default
`xlab="", ylab=""`	create labels for the x- and y-axis

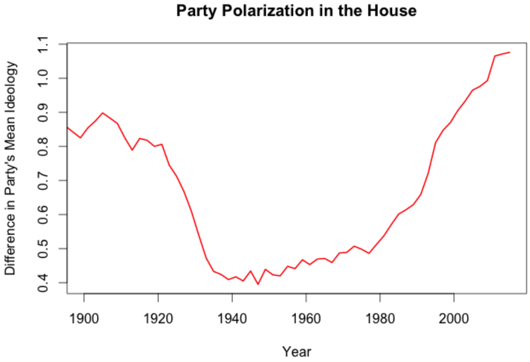
- **Style**: We can adjust the style in the figure.

`lty=`	select the line type (solid, dashed, dotted, short-long dashed, etc.)
`lwd=`	select the line width (fat or skinny)
`pch=`	select the plotting symbol - number (`pch=1`) or letter (`pch="R"`)
`col=`	select the color

- Additional features from the `par` function

```
> par(mfrow=c(1,2))     ## two plots in one figure
> plot(h.pol.sub$partydiff)
> plot(h.pol.sub$partydiff, h.pol.sub$overlap)
```

```
> plot(h.pol.sub$year, h.pol.sub$partydiff,
+       type="l", lty=1, lwd=2, col=2,
+       main="Party Polarization in the House",
+       xlim=c(1900, 2015),
+       xlab="Year", ylab="Difference in Party's Mean Ideology")
```



- Add-on functions: There are a number of add-on functions that we can use once the basic coordinate system has been created using `plot`.

`arrows(x1, x2, y1, y2)` Create arrows from `(x1, y1)` to `(x2, y2)` within the plot (useful for labeling particular data points)

`text(x1, y1, "text here")` Create text at `(x1, y1)` within the plot (modify the text size using the `cex` option)

`lines(x1, y1)` Connects lines on top of the existing plot

`points(x1, y1)` Draw a sequence of points on top of the existing plot

`legend(x1, y1, labels=c("",""))` Create a legend to identify the components in the figure

More...

- In addition to the `plot` function, we are able to utilize a variety of figures. To name a few:
    - dot chart
    - bar chart
    - pie chart
    - box plot
    - histogram
    - density plot
- We can also utilize several packages. The most popular ones are:
    - `lattice` package
    - `ggplot2` package

You are now an R expert!!!

ONLY IF you practice it AGAIN(!) at home

Questions? - hmpark1@uwm.edu