

# SML 201 – Week 3

*John D. Storey*

*Spring 2016*

## Contents

<b>Functions</b>	<b>4</b>
Rationale . . . . .	4
Defining a New Function . . . . .	4
Example 1 . . . . .	4
Example 2 . . . . .	4
Example 3 . . . . .	5
Default Function Argument Values . . . . .	5
The Ellipsis Argument . . . . .	6
Argument Matching . . . . .	6
<b>Organizing Your Code</b>	<b>6</b>
Suggestions . . . . .	6
Suggestions (cont'd) . . . . .	7
Where to Put Files . . . . .	7
<b>Environment</b>	<b>7</b>
Loading .RData Files . . . . .	7
Listing Objects . . . . .	8
Removing Objects . . . . .	8
Advanced . . . . .	8
<b>Packages</b>	<b>9</b>
Rationale . . . . .	9
Contents of a Package . . . . .	9
Installing Packages . . . . .	9
Installing Packages (cont'd) . . . . .	10

Loading Packages . . . . .	10
Getting Started with a Package . . . . .	10
Specifying a Function within a Package . . . . .	11
More on Packages . . . . .	11
<b>Subsetting R Objects</b>	<b>11</b>
Subsetting Vectors . . . . .	11
Subsetting Vectors . . . . .	12
Subsetting Matrices . . . . .	12
Subsetting Matrices . . . . .	12
Subsetting Matrices . . . . .	13
Subsetting Lists . . . . .	13
Subsetting Data Frames . . . . .	14
Subsetting Data Frames . . . . .	14
Note on Data Frames . . . . .	15
Missing Values . . . . .	15
Subsetting by Matching . . . . .	16
Advanced Subsetting . . . . .	16
<b>Tidy Data</b>	<b>16</b>
Definition . . . . .	16
Definition (cont'd) . . . . .	17
Example: Titanic Data . . . . .	17
Intuitive Format . . . . .	17
Tidy Format . . . . .	17
Wide vs. Long Format . . . . .	17
<b>reshape2</b> Package . . . . .	18
Air Quality Data Set . . . . .	18
Melt . . . . .	19
Guided Melt . . . . .	19
Casting . . . . .	20
<b>dcast</b> . . . . .	20

<b>Manipulating Data Frames</b>	<b>20</b>
dplyr Package . . . . .	20
Grammar of dplyr . . . . .	21
Grammar of dplyr . . . . .	21
Example: Baby Names . . . . .	21
babynames Object . . . . .	22
Peek at the Data . . . . .	22
%>% Operator . . . . .	23
filter() . . . . .	23
arrange() . . . . .	24
arrange() . . . . .	24
rename() . . . . .	25
select() . . . . .	25
Renaming with select() . . . . .	26
mutate() . . . . .	26
No. Individuals by Year and Sex . . . . .	26
summarize() . . . . .	27
group_by() . . . . .	27
No. Individuals by Year and Sex . . . . .	28
How Many Distinct Names? . . . . .	28
Most Popular Names . . . . .	29
Most Popular Names . . . . .	29
Most Popular Female Names . . . . .	30
Most Popular Male Names . . . . .	30
Additional Examples . . . . .	32
Additional dplyr Features . . . . .	32
<b>Extras</b>	<b>33</b>
License . . . . .	33
Source Code . . . . .	33
Session Information . . . . .	33

# Functions

## Rationale

- Writing functions is a core activity of an R programmer. It represents the key step of the transition from a mere user to a developer who creates new functionality for R.
- Functions are often used to encapsulate a sequence of expressions that need to be executed numerous times, perhaps under slightly different conditions.
- Functions are also often written when code must be shared with others or the public.

From *R Programming for Data Science*

## Defining a New Function

- Functions are defined using the `function()` directive
- They are stored as variables, so they can be passed to other functions and assigned to new variables
- Arguments and a final return object are defined

## Example 1

```
> my_square <- function(x) {  
+   x*x # can also do return(x*x)  
+ }  
>  
> my_square(x=2)  
[1] 4  
>  
> my_fun2 <- my_square  
> my_fun2(x=3)  
[1] 9
```

## Example 2

```

> my_square_ext <- function(x) {
+   y <- x*x
+   return(list(x_original=x, x_squared=y))
+ }
>
> my_square_ext(x=2)
$x_original
[1] 2

$x_squared
[1] 4
>
> z <- my_square_ext(x=2)

```

### Example 3

```

> my_power <- function(x, e, say_hello) {
+   if(say_hello) {
+     cat("Hello World!")
+   }
+   x^e
+ }
>
> my_power(x=2, e=3, say_hello=TRUE)
Hello World!
[1] 8
>
> z <- my_power(x=2, e=3, say_hello=TRUE)
Hello World!
> z
[1] 8

```

### Default Function Argument Values

Some functions have default values for their arguments:

```

> str(matrix)
function (data = NA, nrow = 1, ncol = 1, byrow = FALSE,
         dimnames = NULL)

```

You can define a function with default values by the following:

```
f <- function(x, y=2) {  
  x + y  
}
```

If the user types `f(x=1)` then it defaults to `y=2`, but if the user types `f(x=1, y=3)`, then it executes with these assignments.

## The Ellipsis Argument

You will encounter functions that include as a possible argument the ellipsis:

...

This basically holds arguments that can be passed to functions called within a function. Example:

```
> double_log <- function(x, ...) {  
+   log((2*x), ...)  
+ }  
>  
> double_log(x=1, base=2)  
[1] 1  
> double_log(x=1, base=10)  
[1] 0.30103
```

## Argument Matching

*R Programming for Data Science* spends several pages discussing how R deals with function calls when the arguments are not defined explicitly. For example:

```
x <- matrix(1:6, nrow=2, ncol=3, byrow=TRUE) # versus  
x <- matrix(1:6, 2, 3, TRUE)
```

I strongly recommend that you define arguments explicitly. For example, I can never remember which comes first in `matrix()`, `nrow` or `ncol`.

## Organizing Your Code

### Suggestions

RStudio conveniently tries to automatically format your R code. We suggest the following in general.

1. No more than 80 characters per line (or fewer depending on how R Markdown compiles):

```
really_long_line <- my_function(x=20, y=30, z=TRUE,  
                               a="Joe", b=3.8)
```

2. Indent 2 or more characters for nested commands:

```
for(i in 1:10) {  
  if(i > 4) {  
    print(i)  
  }  
}
```

## Suggestions (cont'd)

3. Generously comment your code.

```
# a for-loop that prints the index  
# whenever it is greater than 4  
for(i in 1:10) {  
  if(i > 4) {  
    print(i)  
  }  
}  
# a good way to get partial credit  
# if something goes wrong :-)
```

4. Do not hesitate to write functions to organize tasks. These help to break up your code into more understandable pieces, and functions can often be used several times.

## Where to Put Files

See *Elements of Data Analytic Style*, Chapter 12 (“Reproducibility”) for suggestions on how to organize your files.

In this course, we will keep this relatively simple. We will try to provide you with some organization when distributing the projects.

## Environment

### Loading .RData Files

An `.RData` file is a binary file containing R objects. These can be saved from your current R session and also loaded into your current session.

```

> # generally...
> # to load:
> load(file="path/to/file_name.RData")
> # to save:
> save(file="path/to/file_name.RData")

> # assumes file in working directory
> load(file="project_1_R_basics.RData")

```

```

> # loads from our GitHub repository
> load(file=url("https://github.com/SML201/project1/raw/
+             master/project_1_R_basics.RData"))

```

## Listing Objects

The objects in your current R session can be listed. An environment can also be specified in case you have objects stored in different environments.

```

> ls()
[1] "num_people_in_precept"      "SML201_grade_distribution"
[3] "some_ORFE_profs"
>
> ls(name=globalenv())
[1] "num_people_in_precept"      "SML201_grade_distribution"
[3] "some_ORFE_profs"
>
> # see help file for other options
> ?ls

```

## Removing Objects

You can remove specific objects or all objects from your R environment of choice.

```

> rm("some_ORFE_profs") # removes variable some_ORFE_profs
>
> rm(list=ls()) # Removes all variables from environment

```

## Advanced

The R environment is there to connect object names to object values.



The *R Programming for Data Science* chapter titled “Scoping Rules of R” discussed environments and object names in more detail than we need for this course.

A useful discussion about environments can also be found on the *Advanced R* web site.

## Packages

### Rationale

“In R, the fundamental unit of shareable code is the package. A package bundles together code, data, documentation, and tests, and is easy to share with others. As of January 2015, there were over 6,000 packages available on the **C**omprehensive **R** **A**rchive **N**etwork, or **CRAN**, the public clearing house for R packages. This huge variety of packages is one of the reasons that R is so successful: the chances are that someone has already solved a problem that you’re working on, and you can benefit from their work by downloading their package.”

From <http://r-pkgs.had.co.nz/intro.html> by Hadley Wickham

### Contents of a Package

- R functions
- R data objects
- Help documents for using the package
- Information on the authors, dependencies, etc.
- Information to make sure it “plays well” with R and other packages

### Installing Packages

From CRAN:

```
install.packages("dplyr")
```

From GitHub (for advanced users):

```
library("devtools")  
install_github("hadley/dplyr")
```

From Bioconductor (basically CRAN for biology):

```
source("https://bioconductor.org/biocLite.R")
biocLite("qvalue")
```

We will (probably) only be using packages from CRAN. Be *very* careful about dependencies when installing from GitHub.

## Installing Packages (cont'd)

Multiple packages:

```
install.packages(c("dplyr", "ggplot2"))
```

Install all dependencies:

```
install.packages(c("dplyr", "ggplot2"), dependencies=TRUE)
```

Updating packages:

```
update.packages()
```

## Loading Packages

Two ways to load a package:

```
library("dplyr")
library(dplyr)
```

I prefer the former.

## Getting Started with a Package

When you install a new package and load it, what's next? I like to look at the help files and see what functions and data sets a package has.

```
library("dplyr")
help(package="dplyr")
```

## Specifying a Function within a Package

You can call a function from a specific package. Suppose you are in a setting where you have two packages loaded that have functions with the same name.

```
dplyr::arrange(mtcars, cyl, disp)
```

This calls the `arrange` function specifically from `dplyr`. The package `plyr` also has an `arrange` function.

## More on Packages

We will be covering several highly used R packages in depth this semester, so we will continue to learn about packages, how they are organized, and how they are used.

You can download the “source” of a package from R and take a look at the contents if you want to dig deeper. There are also many good tutorials on creating packages, such as <http://hilaryparker.com/2014/04/29/writing-an-r-package-from-scratch/>.

## Subsetting R Objects

### Subsetting Vectors

```
> x <- 1:8
>
> x[1]           # extract the first element
[1] 1
> x[2]           # extract the second element
[1] 2
>
> x[1:4]         # extract the first 4 elements
[1] 1 2 3 4
>
> x[c(1, 3, 4)] # extract elements 1, 3, and 4
[1] 1 3 4
> x[-c(1, 3, 4)] # extract all elements EXCEPT 1, 3, and 4
[1] 2 5 6 7 8
```

## Subsetting Vectors

```
> names(x) <- letters[1:8]
> x
a b c d e f g h
1 2 3 4 5 6 7 8
>
> x[c("a", "b", "f")]
a b f
1 2 6
>
> s <- x > 3
> s
      a      b      c      d      e      f      g      h
FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
> x[s]
d e f g h
4 5 6 7 8
```

## Subsetting Matrices

```
> x <- matrix(1:6, nrow=2, ncol=3, byrow=TRUE)
> x
      [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
>
> x[1,2]
[1] 2
> x[1, ]
[1] 1 2 3
> x[ ,2]
[1] 2 5
```

## Subsetting Matrices

```
> colnames(x) <- c("A", "B", "C")
>
> x[ , c("B", "C")]
      B C
[1,] 2 3
```

```

[2,] 5 6
>
> x[c(FALSE, TRUE), c("B", "C")]
B C
5 6
>
> x[2, c("B", "C")]
B C
5 6

```

## Subsetting Matrices

```

> s <- (x %% 2) == 0
> s
      A      B      C
[1,] FALSE TRUE FALSE
[2,]  TRUE FALSE  TRUE
>
> x[s]
[1] 4 2 6
>
> x[c(2, 3, 6)]
[1] 4 2 6

```

## Subsetting Lists

```

> x <- list(my=1:3, favorite=c("a", "b", "c"),
+         course=c(FALSE, TRUE, NA))
>
> x[[1]]
[1] 1 2 3
> x[["my"]]
[1] 1 2 3
> x$my
[1] 1 2 3

```

```

> x[[c(3,1)]]
[1] FALSE
> x[[3]][1]
[1] FALSE

```

```

> x[c(3,1)]
$course
[1] FALSE TRUE NA

$my
[1] 1 2 3

```

## Subsetting Data Frames

```

> x <- data.frame(my=1:3, favorite=c("a", "b", "c"),
+               course=c(FALSE, TRUE, NA))
>
> x[[1]]
[1] 1 2 3
> x[["my"]]
[1] 1 2 3
> x$my
[1] 1 2 3

```

```

> x[[c(3,1)]]
[1] FALSE
> x[[3]][1]
[1] FALSE

```

```

> x[c(3,1)]
  course my
1  FALSE 1
2   TRUE 2
3    NA 3

```

## Subsetting Data Frames

```

> x <- data.frame(my=1:3, favorite=c("a", "b", "c"),
+               course=c(FALSE, TRUE, NA))
>
> x[1, ]
  my favorite course
1 1      a  FALSE
> x[ ,3]
[1] FALSE TRUE NA
> x[ ,"favorite"]

```

```
[1] a b c
Levels: a b c
```

```
> x[1:2, ]
  my favorite course
1 1          a FALSE
2 2          b  TRUE
> x[,2:3]
  favorite course
1          a FALSE
2          b  TRUE
3          c  NA
```

## Note on Data Frames

R often converts character strings to factors unless you specify otherwise.

In the previous slide, we saw it converted the “favorite” column to factors. Let’s fix that…

```
> x <- data.frame(my=1:3, favorite=c("a", "b", "c"),
+               course=c(FALSE, TRUE, NA),
+               stringsAsFactors=FALSE)
>
> x[, "favorite"]
[1] "a" "b" "c"
> class(x[, "favorite"])
[1] "character"
```

## Missing Values

```
> data("airquality", package="datasets")
> head(airquality)
  Ozone Solar.R Wind Temp Month Day
1   41    190  7.4  67     5   1
2   36    118  8.0  72     5   2
3   12    149 12.6  74     5   3
4   18    313 11.5  62     5   4
5   NA     NA 14.3  56     5   5
6   28     NA 14.9  66     5   6
> dim(airquality)
[1] 153  6
```

```

> which(is.na(airquality$Ozone))
 [1]  5 10 25 26 27 32 33 34 35 36 37 39 42 43
[15] 45 46 52 53 54 55 56 57 58 59 60 61 65 72
[29] 75 83 84 102 103 107 115 119 150
> sum(is.na(airquality$Ozone))
 [1] 37

```

## Subsetting by Matching

```

> letters
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n"
[15] "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
> vowels <- c("a", "e", "i", "o", "u")
>
> letters %in% vowels
 [1]  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE
[10] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
[19] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
> which(letters %in% vowels)
 [1]  1  5  9 15 21
>
> letters[which(letters %in% vowels)]
 [1] "a" "e" "i" "o" "u"

```

## Advanced Subsetting

The *R Programming for Data Science* chapter titled “Subsetting R Objects” contains additional material on subsetting that you should know.

The *Advanced R* website contains more detailed information on subsetting that you may find useful.

## Tidy Data

### Definition

Tidy datasets are easy to manipulate, model and visualize, and have a specific structure: each variable is a column, each observation is a row, and each type of observational unit is a table.

From Wickham (2014), “Tidy Data”, *Journal of Statistical Software*



## Definition (cont'd)

A dataset is a collection of values, usually either numbers (if quantitative) or strings (if qualitative). Values are organized in two ways. Every value belongs to a variable and an observation. A variable contains all values that measure the same underlying attribute (like height, temperature, duration) across units. An observation contains all values measured on the same unit (like a person, or a day, or a race) across attributes.

From: Wickham H (2014), “Tidy Data”, *Journal of Statistical Software*

## Example: Titanic Data

According to the `Titanic` data from the `datasets` package: 367 males survived, 1364 males perished, 344 females survived, and 126 females perished.

How should we organize these data?

## Intuitive Format

	Survived	Perished
Male	367	1364
Female	344	126

## Tidy Format

fate	sex	number
perished	male	1364
perished	female	126
survived	male	367
survived	female	344

## Wide vs. Long Format

Tidy data come in wide and long formats.

Wide format data have a column for each variable and there is one observed unit per row.

The simplest long format data have two columns. The first column contains the variable names and the second column contains the values for the variables.

There are “wider” long format data that have additional columns that identify connections between observations.

Wide format data is useful for some analyses and long format for others.

## reshape2 Package

The `reshape2` package has three important functions: `melt`, `dcast`, and `acast`. It allows one to move between wide and long tidy data formats.

```
> library("reshape2")
> library("datasets")
> data(airquality, package="datasets")
> names(airquality)
[1] "Ozone" "Solar.R" "Wind" "Temp" "Month" "Day"
> dim(airquality)
[1] 153 6
```

## Air Quality Data Set

```
> head(airquality)
Source: local data frame [6 x 6]

  Ozone Solar.R Wind Temp Month Day
  (int)  (int) (dbl) (int) (int) (int)
1    41    190  7.4   67    5    1
2    36    118  8.0   72    5    2
3    12    149 12.6   74    5    3
4    18    313 11.5   62    5    4
5     NA     NA 14.3   56    5    5
6    28     NA 14.9   66    5    6
```

```
> tail(airquality)
Source: local data frame [6 x 6]

  Ozone Solar.R Wind Temp Month Day
  (int)  (int) (dbl) (int) (int) (int)
1    14     20 16.6   63    9   25
2    30    193  6.9   70    9   26
3     NA    145 13.2   77    9   27
4    14    191 14.3   75    9   28
5    18    131  8.0   76    9   29
6    20    223 11.5   68    9   30
```

## Melt

Melting can be thought of as melting a piece of solid metal (wide data), so it drips into long format.

```
> aql <- melt(airquality)
No id variables; using all as measure variables
> head(aql)
  variable value
1   Ozone   41
2   Ozone   36
3   Ozone   12
4   Ozone   18
5   Ozone   NA
6   Ozone   28
```

```
> tail(aql)
  variable value
913   Day    25
914   Day    26
915   Day    27
916   Day    28
917   Day    29
918   Day    30
```

## Guided Melt

In the previous example, we lose the fact that a set of measurements occurred on a particular day and month, so we can do a guided melt to keep this information.

```
> aql <- melt(airquality, id.vars = c("Month", "Day"))
> head(aql)
  Month Day variable value
1     5  1   Ozone   41
2     5  2   Ozone   36
3     5  3   Ozone   12
4     5  4   Ozone   18
5     5  5   Ozone   NA
6     5  6   Ozone   28
```

```
> tail(aql)
  Month Day variable value
607    9  25     Temp    63
608    9  26     Temp    70
```

609	9	27	Temp	77
610	9	28	Temp	75
611	9	29	Temp	76
612	9	30	Temp	68

## Casting

Casting allows us to go from long format to wide format data. It can be visualized as pouring molten metal (long format) into a cast to create a solid piece of metal (wide format).

Casting is more difficult because choices have to be made to determine how the wide format will be organized. It often takes some thought and experimentation for new users.

Let's do an example with `dcast`, which is casting for data frames.

### `dcast`

```
> aqw <- dcast(aql, Month + Day ~ variable)
> head(aqw)
```

	Month	Day	Ozone	Solar.R	Wind	Temp
1	5	1	41	190	7.4	67
2	5	2	36	118	8.0	72
3	5	3	12	149	12.6	74
4	5	4	18	313	11.5	62
5	5	5	NA	NA	14.3	56
6	5	6	28	NA	14.9	66

```
> tail(aqw)
```

	Month	Day	Ozone	Solar.R	Wind	Temp
148	9	25	14	20	16.6	63
149	9	26	30	193	6.9	70
150	9	27	NA	145	13.2	77
151	9	28	14	191	14.3	75
152	9	29	18	131	8.0	76
153	9	30	20	223	11.5	68

## Manipulating Data Frames

### `dplyr` Package

`dplyr` is a package with the following description:

A fast, consistent tool for working with data frame like objects, both in memory and out of memory.

This package offers a “grammar” for manipulating data frames.

Everything that `dplyr` does can also be done using basic R commands – however, it tends to be much faster and easier to use `dplyr`.

## Grammar of `dplyr`

Verbs:

- **filter**: extract a subset of rows from a data frame based on logical conditions
- **arrange**: reorder rows of a data frame
- **rename**: rename variables in a data frame
- **select**: return a subset of the columns of a data frame, using a flexible notation

Partially based on *R Programming for Data Science*

## Grammar of `dplyr`

Verbs (continued):

- **mutate**: add new variables/columns or transform existing variables
- **distinct**: returns only the unique values in a table
- **summarize**: generate summary statistics of different variables in the data frame, possibly within strata
- **group\_by**: breaks down a dataset into specified groups of rows

Partially based on *R Programming for Data Science*

## Example: Baby Names

```
> library("dplyr", verbose=FALSE)
> library("babynames")
> ls()
character(0)
> babynames <- babynames::babynames
> ls()
[1] "babynames"
```

## babynames Object

```
> class(babynames)
[1] "tbl_df"      "tbl"        "data.frame"
> dim(babynames)
[1] 1792091      5
```

```
> babynames
Source: local data frame [1,792,091 x 5]
```

	year (dbl)	sex (chr)	name (chr)	n (int)	prop (dbl)
1	1880	F	Mary	7065	0.07238359
2	1880	F	Anna	2604	0.02667896
3	1880	F	Emma	2003	0.02052149
4	1880	F	Elizabeth	1939	0.01986579
5	1880	F	Minnie	1746	0.01788843
6	1880	F	Margaret	1578	0.01616720
7	1880	F	Ida	1472	0.01508119
8	1880	F	Alice	1414	0.01448696
9	1880	F	Bertha	1320	0.01352390
10	1880	F	Sarah	1288	0.01319605
...	...	...	...	...	...

## Peek at the Data

```
> set.seed(201)
> sample_n(babynames, 10)
Source: local data frame [10 x 5]
```

	year (dbl)	sex (chr)	name (chr)	n (int)	prop (dbl)
1	1991	F	Sayra	29	1.426546e-05
2	1932	F	Wannell	5	4.520211e-06
3	1966	M	Rey	26	1.430083e-05
4	1905	F	Samuel	7	2.258975e-05
5	1992	F	Sherron	17	8.483034e-06
6	1927	F	Pierrette	7	5.662116e-06
7	1907	M	Nolen	6	3.783293e-05
8	1967	F	Cheri	1305	7.602543e-04
9	1920	M	Tyson	11	9.991662e-06
10	1955	F	Gay	493	2.459665e-04

```
> # try also sample_frac(babynames, 6e-6)
```

## %>% Operator

Originally from R package `magrittr`. Provides a mechanism for chaining commands with a forward-pipe operator, `%>%`.

```
> x <- 1:10
>
> x %>% log(base=10) %>% sum
[1] 6.559763
>
> sum(log(x,base=10))
[1] 6.559763
```

```
> babynames %>% sample_n(5)
Source: local data frame [5 x 5]

  year  sex  name    n      prop
  (dbl) (chr) (chr) (int) (dbl)
1  1978   M   Toy      8 4.681892e-06
2  1995   M Derron   32 1.591702e-05
3  1990   M  Jacob 22000 1.022964e-02
4  1979   F  Clara   342 1.985056e-04
5  1983   M  Jerid   35 1.879331e-05
```

## filter()

```
> filter(babynames, year==1880, sex=="F")
Source: local data frame [942 x 5]

  year  sex  name    n      prop
  (dbl) (chr) (chr) (int) (dbl)
1  1880   F   Mary  7065 0.07238359
2  1880   F   Anna  2604 0.02667896
3  1880   F   Emma  2003 0.02052149
4  1880   F Elizabeth 1939 0.01986579
5  1880   F   Minnie 1746 0.01788843
6  1880   F Margaret 1578 0.01616720
7  1880   F     Ida 1472 0.01508119
8  1880   F   Alice 1414 0.01448696
9  1880   F  Bertha 1320 0.01352390
10 1880   F   Sarah 1288 0.01319605
... ..
> # same as filter(babynames, year==1880 & sex=="F")
```

```
> filter(babynames, year==1880, sex=="F", n > 5000)
Source: local data frame [1 x 5]
```

	year	sex	name	n	prop
	(dbl)	(chr)	(chr)	(int)	(dbl)
1	1880	F	Mary	7065	0.07238359

## arrange()

```
> arrange(babynames, name, year, sex)
Source: local data frame [1,792,091 x 5]
```

	year	sex	name	n	prop
	(dbl)	(chr)	(chr)	(int)	(dbl)
1	2007	M	Aaban	5	2.260668e-06
2	2009	M	Aaban	6	2.835010e-06
3	2010	M	Aaban	9	4.392374e-06
4	2011	M	Aaban	11	5.433940e-06
5	2012	M	Aaban	11	5.447022e-06
6	2013	M	Aaban	14	6.998380e-06
7	2011	F	Aabha	7	3.625123e-06
8	2012	F	Aabha	5	2.590107e-06
9	2003	M	Aabid	5	2.381787e-06
10	2008	F	Aabriella	5	2.405002e-06
...	...	...	...	...	...

## arrange()

```
> arrange(babynames, desc(name), desc(year), sex)
Source: local data frame [1,792,091 x 5]
```

	year	sex	name	n	prop
	(dbl)	(chr)	(chr)	(int)	(dbl)
1	2010	M	Zzyzx	5	2.440208e-06
2	2010	F	Zyyanna	6	3.068790e-06
3	2009	M	Zyvion	5	2.362508e-06
4	2010	M	Zytavious	6	2.928249e-06
5	2009	M	Zytavious	7	3.307511e-06
6	2007	M	Zytavious	6	2.712801e-06
7	2006	M	Zytavious	7	3.197154e-06
8	2005	M	Zytavious	5	2.353078e-06



```

9  2004      M Zytavious      6 2.841921e-06
10 2002      M Zytavious      6 2.905729e-06
..   ...      ...           ...           ...

```

## rename()

```

> rename(babynames, number=n)
Source: local data frame [1,792,091 x 5]

```

	year (dbl)	sex (chr)	name (chr)	number (int)	prop (dbl)
1	1880	F	Mary	7065	0.07238359
2	1880	F	Anna	2604	0.02667896
3	1880	F	Emma	2003	0.02052149
4	1880	F	Elizabeth	1939	0.01986579
5	1880	F	Minnie	1746	0.01788843
6	1880	F	Margaret	1578	0.01616720
7	1880	F	Ida	1472	0.01508119
8	1880	F	Alice	1414	0.01448696
9	1880	F	Bertha	1320	0.01352390
10	1880	F	Sarah	1288	0.01319605
..	...	...	...	...	...

## select()

```

> select(babynames, sex, name, n)
Source: local data frame [1,792,091 x 3]

```

	sex (chr)	name (chr)	n (int)
1	F	Mary	7065
2	F	Anna	2604
3	F	Emma	2003
4	F	Elizabeth	1939
5	F	Minnie	1746
6	F	Margaret	1578
7	F	Ida	1472
8	F	Alice	1414
9	F	Bertha	1320
10	F	Sarah	1288
..	...	...	...

```

> # same as select(babynames, sex:n)

```

## Renaming with select()

```
> select(babynames, sex, name, number=n)
Source: local data frame [1,792,091 x 3]
```

	sex (chr)	name (chr)	number (int)
1	F	Mary	7065
2	F	Anna	2604
3	F	Emma	2003
4	F	Elizabeth	1939
5	F	Minnie	1746
6	F	Margaret	1578
7	F	Ida	1472
8	F	Alice	1414
9	F	Bertha	1320
10	F	Sarah	1288
...	...	...	...

## mutate()

```
> mutate(babynames, total_by_year=round(n/prop))
Source: local data frame [1,792,091 x 6]
```

	year (dbl)	sex (chr)	name (chr)	n (int)	prop (dbl)	total_by_year (dbl)
1	1880	F	Mary	7065	0.07238359	97605
2	1880	F	Anna	2604	0.02667896	97605
3	1880	F	Emma	2003	0.02052149	97605
4	1880	F	Elizabeth	1939	0.01986579	97605
5	1880	F	Minnie	1746	0.01788843	97605
6	1880	F	Margaret	1578	0.01616720	97605
7	1880	F	Ida	1472	0.01508119	97605
8	1880	F	Alice	1414	0.01448696	97605
9	1880	F	Bertha	1320	0.01352390	97605
10	1880	F	Sarah	1288	0.01319605	97605
...	...	...	...	...	...	...

```
> # see also transmutate
```

## No. Individuals by Year and Sex

Let's put a few things together now adding the function `distinct()`...

```
> babynames %>% mutate(total_by_year=round(n/prop)) %>%
+   select(sex, year, total_by_year) %>% distinct()
Source: local data frame [268 x 3]
```

	sex (chr)	year (dbl)	total_by_year (dbl)
1	F	1880	97605
2	M	1880	118400
3	F	1881	98857
4	M	1881	108285
5	F	1882	115698
6	M	1882	122032
7	F	1883	120065
8	M	1883	112481
9	F	1884	137588
10	M	1884	122742
...	...	...	...

### summarize()

```
> summarize(babynames, mean_n = mean(n), median_n = median(n),
+           number_sex = n_distinct(sex),
+           distinct_names = n_distinct(name))
Source: local data frame [1 x 4]
```

	mean_n (dbl)	median_n (int)	number_sex (int)	distinct_names (int)
1	186.0496	12	2	92600

### group\_by()

```
> babynames %>% group_by(year, sex)
Source: local data frame [1,792,091 x 5]
Groups: year, sex [268]
```

	year (dbl)	sex (chr)	name (chr)	n (int)	prop (dbl)
1	1880	F	Mary	7065	0.07238359
2	1880	F	Anna	2604	0.02667896
3	1880	F	Emma	2003	0.02052149
4	1880	F	Elizabeth	1939	0.01986579
5	1880	F	Minnie	1746	0.01788843

```

6  1880    F  Margaret  1578  0.01616720
7  1880    F    Ida     1472  0.01508119
8  1880    F    Alice   1414  0.01448696
9  1880    F   Bertha  1320  0.01352390
10 1880    F    Sarah   1288  0.01319605
..   ...    ...      ...    ...      ...

```

## No. Individuals by Year and Sex

```

> babynames %>% group_by(year, sex) %>%
+   summarize(total_by_year=sum(n))
Source: local data frame [268 x 3]
Groups: year [?]

```

	year (dbl)	sex (chr)	total_by_year (int)
1	1880	F	90993
2	1880	M	110491
3	1881	F	91954
4	1881	M	100746
5	1882	F	107850
6	1882	M	113687
7	1883	F	112322
8	1883	M	104630
9	1884	F	129022
10	1884	M	114446
..	...	...	...

Compare to earlier slide. Why the difference?

## How Many Distinct Names?

```

> babynames %>% group_by(sex) %>%
+   summarize(mean_n = mean(n),
+             distinct_names_sex = n_distinct(name))
Source: local data frame [2 x 3]

```

	sex (chr)	mean_n (dbl)	distinct_names_sex (int)
1	F	155.5683	64089
2	M	230.4324	38601

## Most Popular Names

```
> top_names <- babynames %>% group_by(year, sex) %>%
+   summarize(top_name = name[which.max(n)])
>
> head(top_names)
Source: local data frame [6 x 3]
Groups: year [3]

  year  sex top_name
  (dbl) (chr)   (chr)
1  1880   F   Mary
2  1880   M   John
3  1881   F   Mary
4  1881   M   John
5  1882   F   Mary
6  1882   M   John
```

## Most Popular Names

### Recent Years

```
> tail(top_names, n=10)
Source: local data frame [10 x 3]
Groups: year [5]

  year  sex top_name
  (dbl) (chr)   (chr)
1  2009   F Isabella
2  2009   M   Jacob
3  2010   F Isabella
4  2010   M   Jacob
5  2011   F   Sophia
6  2011   M   Jacob
7  2012   F   Sophia
8  2012   M   Jacob
9  2013   F   Sophia
10 2013   M   Noah
```

## Most Popular Female Names

1990s

```
> top_names %>% filter(year >= 1990 & year < 2000, sex=="F")
Source: local data frame [10 x 3]
Groups: year [10]
```

	year (dbl)	sex (chr)	top_name (chr)
1	1990	F	Jessica
2	1991	F	Ashley
3	1992	F	Ashley
4	1993	F	Jessica
5	1994	F	Jessica
6	1995	F	Jessica
7	1996	F	Emily
8	1997	F	Emily
9	1998	F	Emily
10	1999	F	Emily

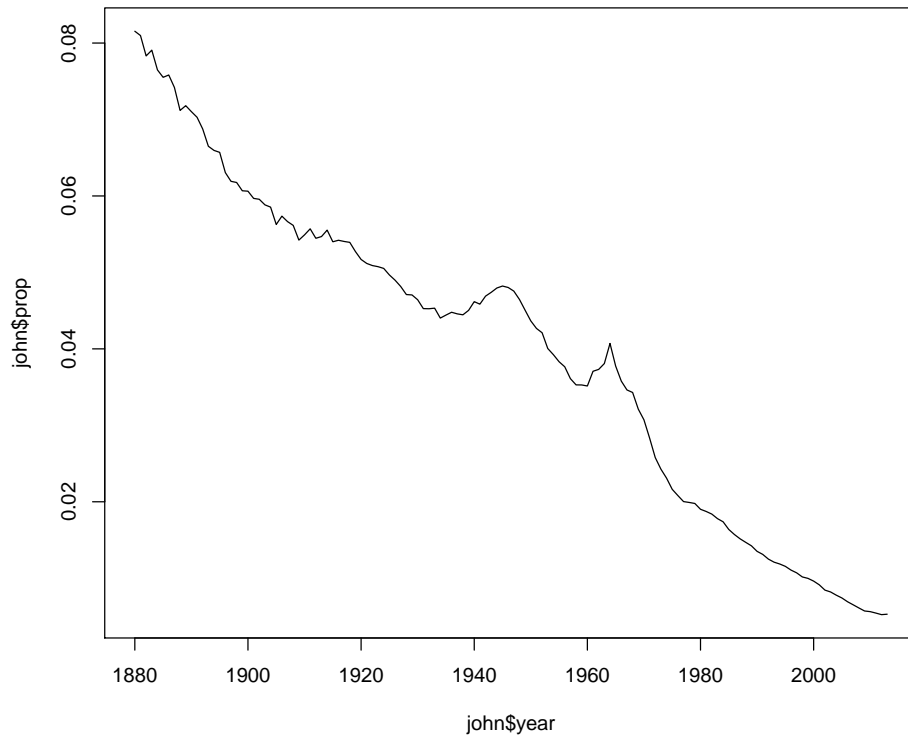
## Most Popular Male Names

1990s

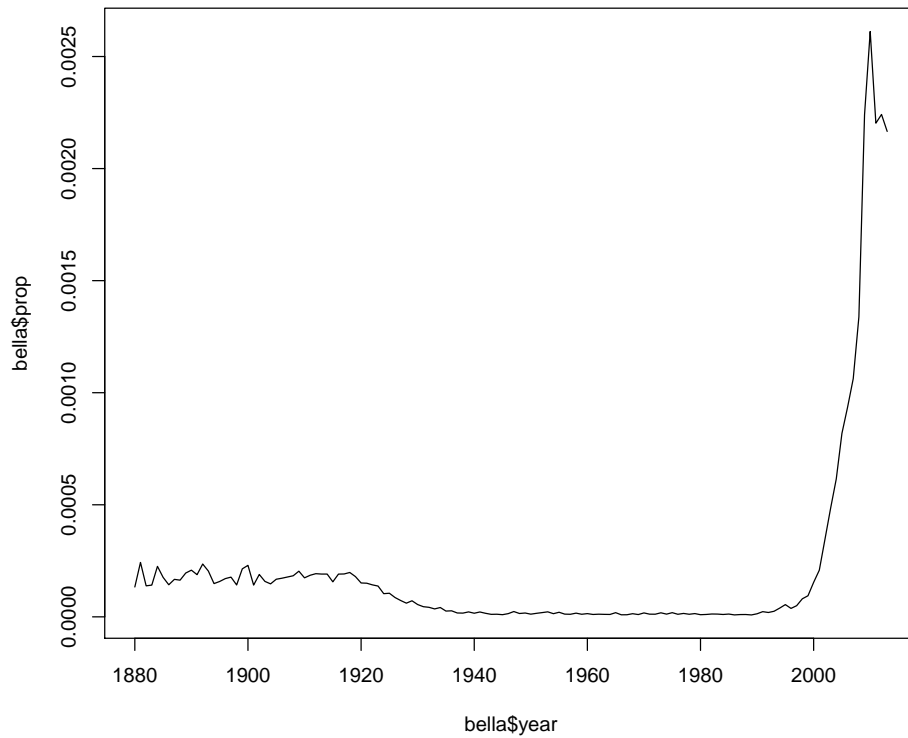
```
> top_names %>% filter(year >= 1990 & year < 2000, sex=="M")
Source: local data frame [10 x 3]
Groups: year [10]
```

	year (dbl)	sex (chr)	top_name (chr)
1	1990	M	Michael
2	1991	M	Michael
3	1992	M	Michael
4	1993	M	Michael
5	1994	M	Michael
6	1995	M	Michael
7	1996	M	Michael
8	1997	M	Michael
9	1998	M	Michael
10	1999	M	Jacob

```
> # Analyzing the name 'John'  
> john <- babynames %>% filter(sex=="M", name=="John")  
> plot(john$year, john$prop, type="l")
```



```
> # Analyzing the name 'Bella'  
> bella <- babynames %>% filter(sex=="F", name=="Bella")  
> plot(bella$year, bella$prop, type="l")
```



## Additional Examples

You should study additional tutorials of `dplyr` that utilize other data sets:

- Read the `dplyr` [introductory vignette](#)
- Read the examples given in *R Programming for Data Science*, the “Managing Data Frames with the `dplyr` Package” chapter

## Additional `dplyr` Features

- We’ve only scratched the surface – many interesting demos of `dplyr` can be found online
- `dplyr` can work with other data frame backends such as SQL databases
- There is an SQL interface for relational databases via the `DBI` package
- `dplyr` can be integrated with the `data.table` package for large fast tables
- There is a [healthy rivalry](#) between `dplyr` and `data.table`



## Extras

### License

<https://github.com/SML201/lectures/blob/master/LICENSE.md>

### Source Code

<https://github.com/SML201/lectures/tree/master/week3>

### Session Information

```
> sessionInfo()
R version 3.2.3 (2015-12-10)
Platform: x86_64-apple-darwin13.4.0 (64-bit)
Running under: OS X 10.11.3 (El Capitan)

locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

attached base packages:
[1] stats      graphics  grDevices  utils      datasets  methods
[7] base

other attached packages:
[1] babynames_0.1  dplyr_0.4.3    reshape2_1.4.1
[4] knitr_1.12.3   devtools_1.10.0

loaded via a namespace (and not attached):
[1] Rcpp_0.12.3    assertthat_0.1 digest_0.6.9
[4] R6_2.1.2       plyr_1.8.3     DBI_0.3.1
[7] formatR_1.2.1  magrittr_1.5   evaluate_0.8
[10] highr_0.5.1    stringi_1.0-1  lazyeval_0.1.10
[13] rmarkdown_0.9.2 tools_3.2.3    stringr_1.0.0
[16] parallel_3.2.3 yaml_2.1.13    memoise_1.0.0
[19] htmltools_0.3
```