Introduction to Statistics in R

Presented by:

BROWN
Center for
Biomedical Informatics

BROWN
Center for
Biomedical Informatics

Introduction to Statistics in R

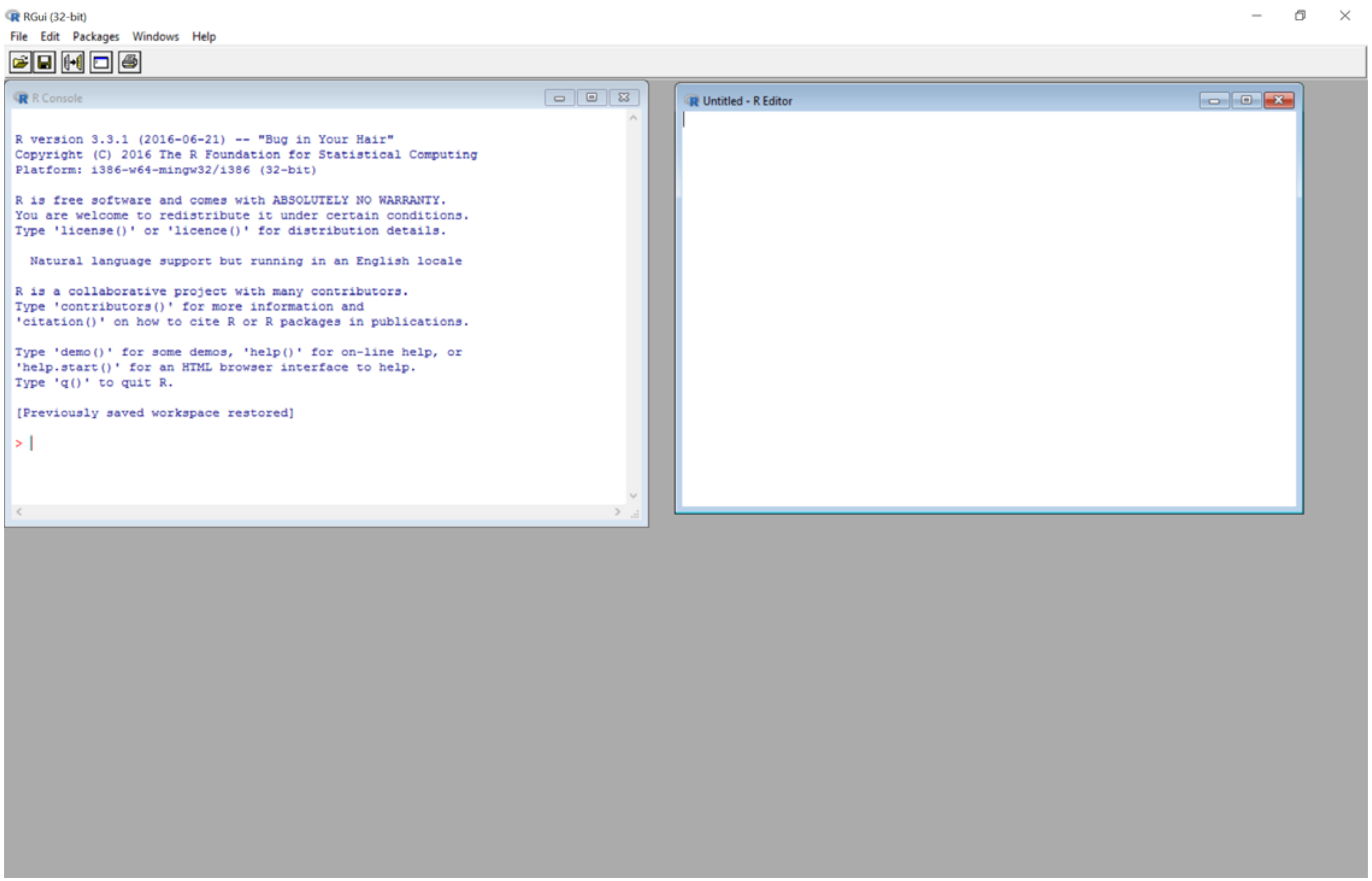
Day 1 - Getting Started with R/Basic Stats

Adam J Sullivan

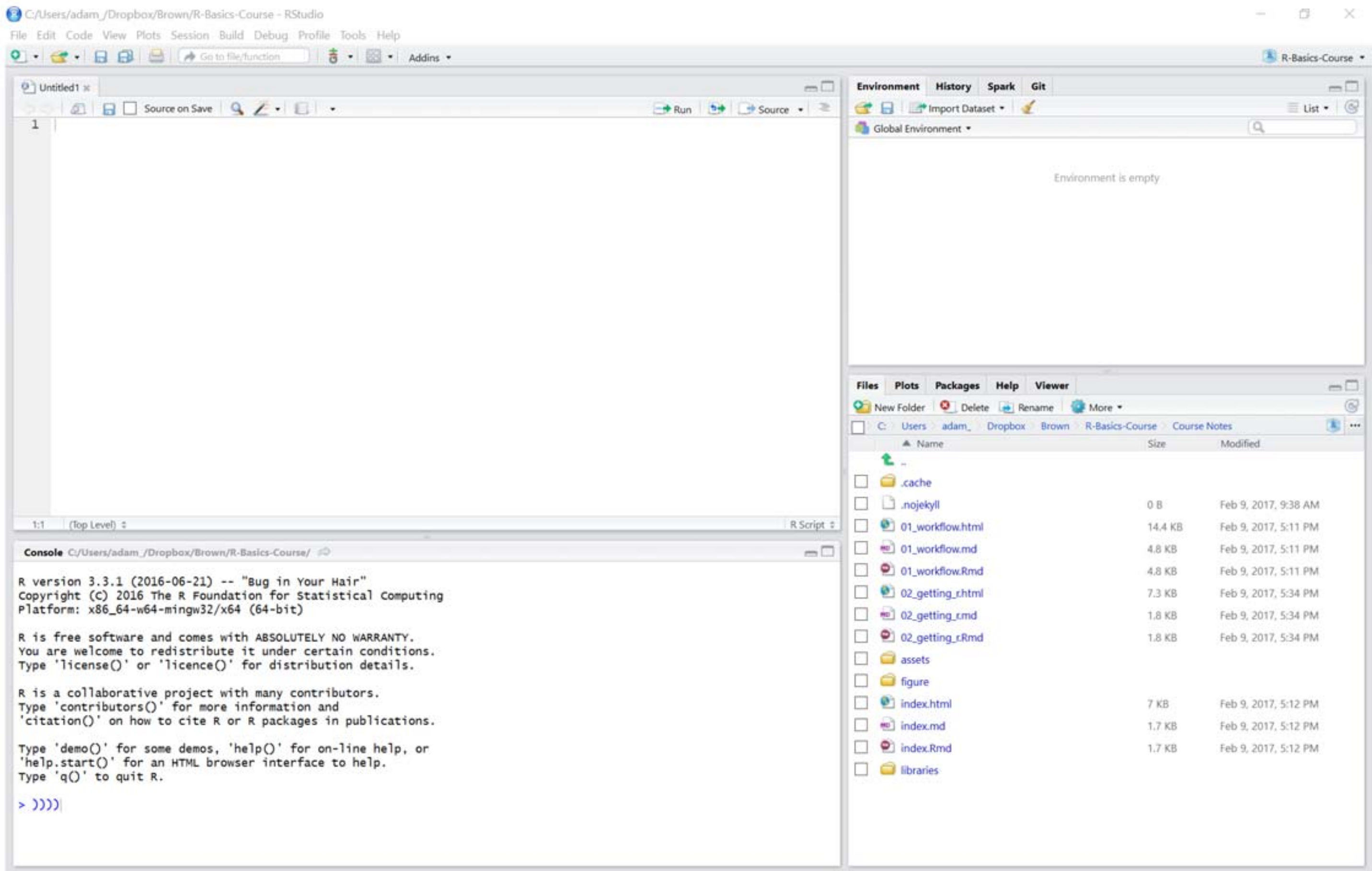
(http://creativecommons.org/licenses/by-nc-nd/3.0/)

Ways to Use R

Base R

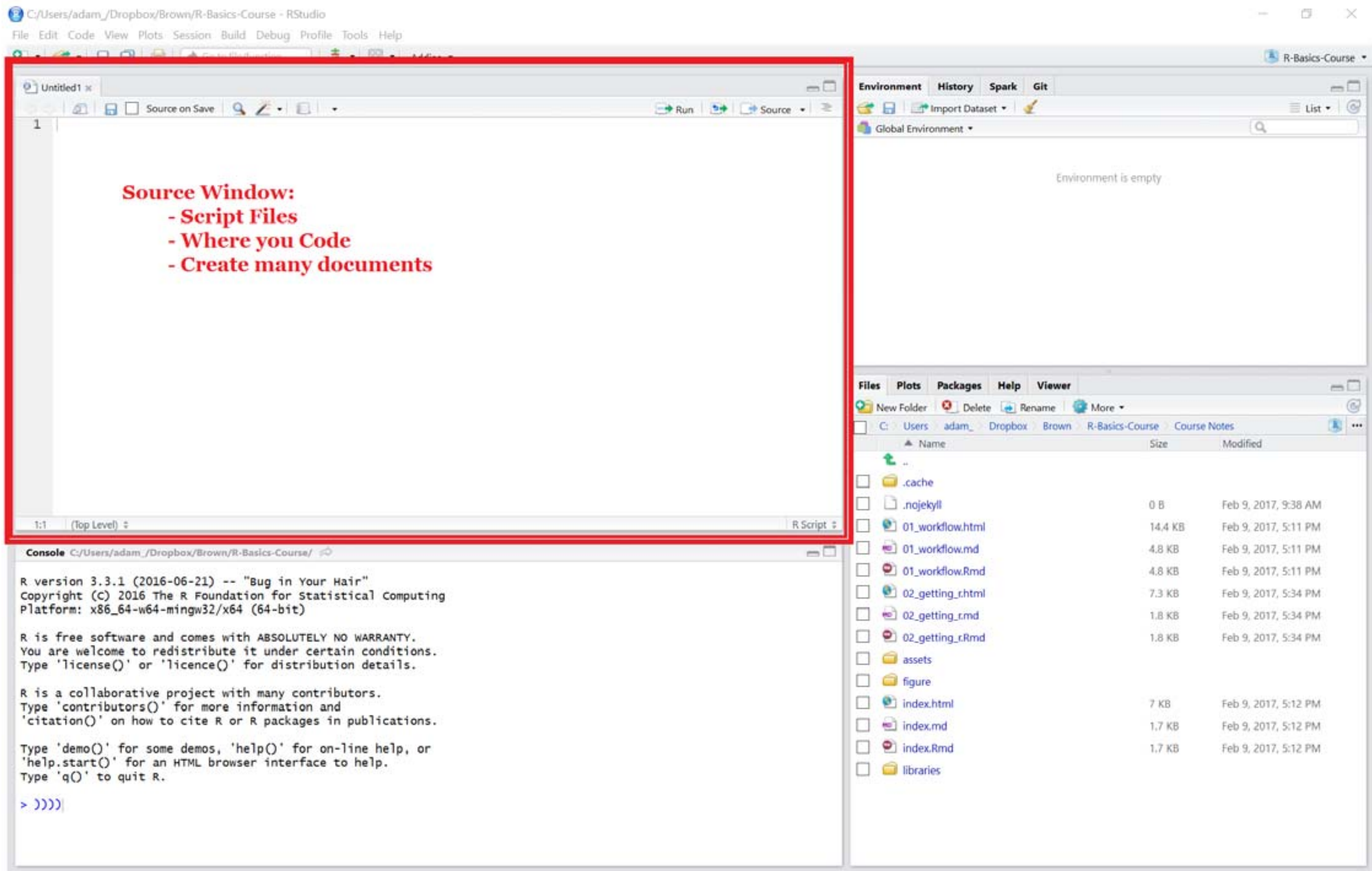

RGui (32-bit)
File Edit Packages Windows Help
R Console
R version 3.3.1 (2016-06-21) -- "Bug in Your Hair"
Copyright (C) 2016 The R Foundation for Statistical Computing
Platform: i386-w64-mingw32/i386 (32-bit)
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it 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.
[Previously saved workspace restored]
>
Untitled - R Editor

RStudio



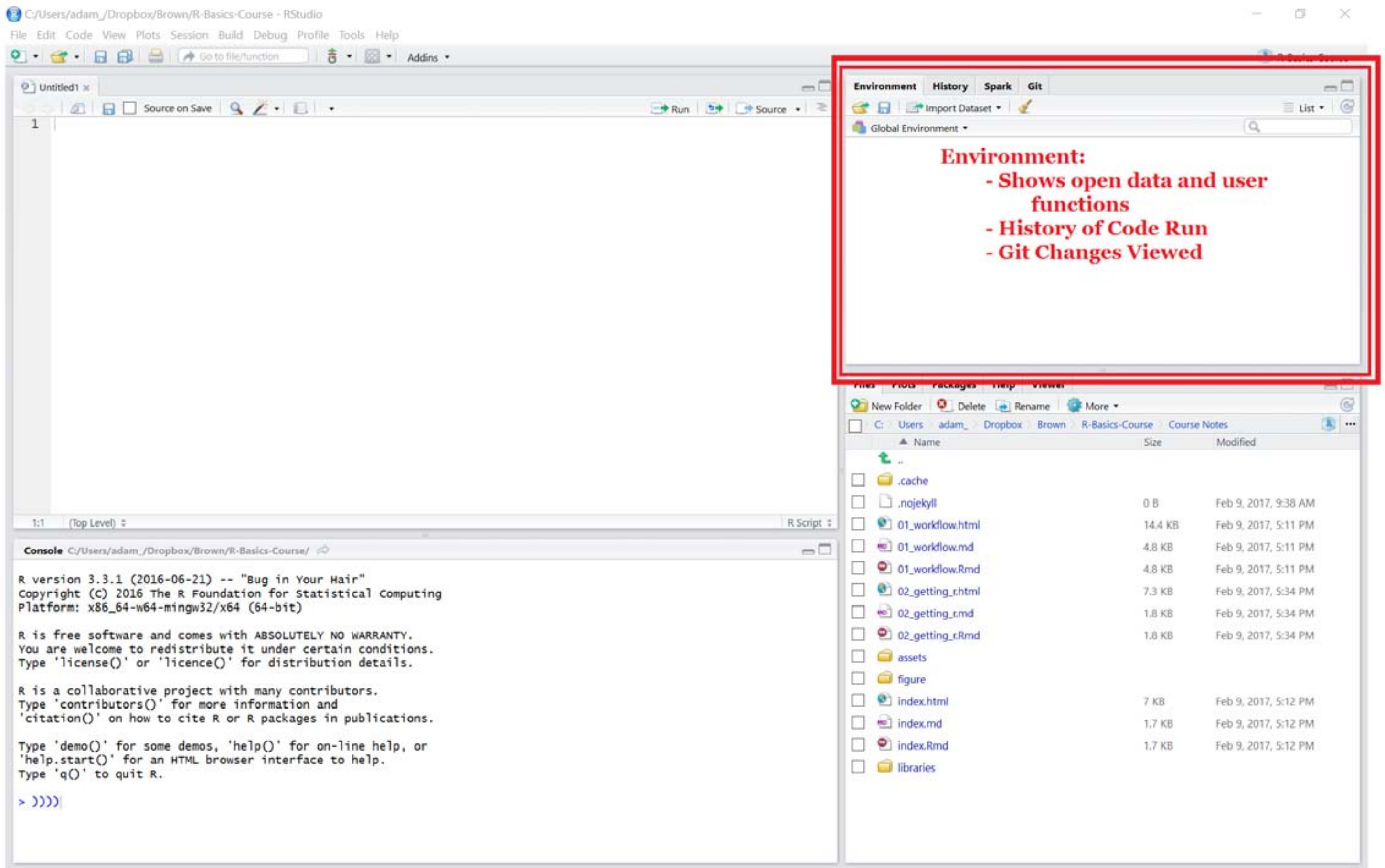
C:/Users/adam_/Dropbox/Brown/R-Basics-Course - RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Addins
R-Basics-Course
Untitled1
Source on Save
Run
Source
Environment History Spark Git
Import Dataset
List
Global Environment
Environment is empty
Files Plots Packages Help Viewer
New Folder Delete Rename More
C: Users adam_ Dropbox Brown R-Basics-Course Course Notes
Name Size Modified
.cache
.nojekyll 0 B Feb 9, 2017, 9:38 AM
01_workflow.html 14.4 KB Feb 9, 2017, 5:11 PM
01_workflow.md 4.8 KB Feb 9, 2017, 5:11 PM
01_workflow.Rmd 4.8 KB Feb 9, 2017, 5:11 PM
02_getting_r.html 7.3 KB Feb 9, 2017, 5:34 PM
02_getting_r.md 1.8 KB Feb 9, 2017, 5:34 PM
02_getting_r.Rmd 1.8 KB Feb 9, 2017, 5:34 PM
assets
figure
index.html 7 KB Feb 9, 2017, 5:12 PM
index.md 1.7 KB Feb 9, 2017, 5:12 PM
index.Rmd 1.7 KB Feb 9, 2017, 5:12 PM
libraries
1:1 (Top Level) R Script
Console C:/Users/adam_/Dropbox/Brown/R-Basics-Course/
R version 3.3.1 (2016-06-21) -- "Bug in Your Hair"
Copyright (C) 2016 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (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.
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.
>))))

RStudio

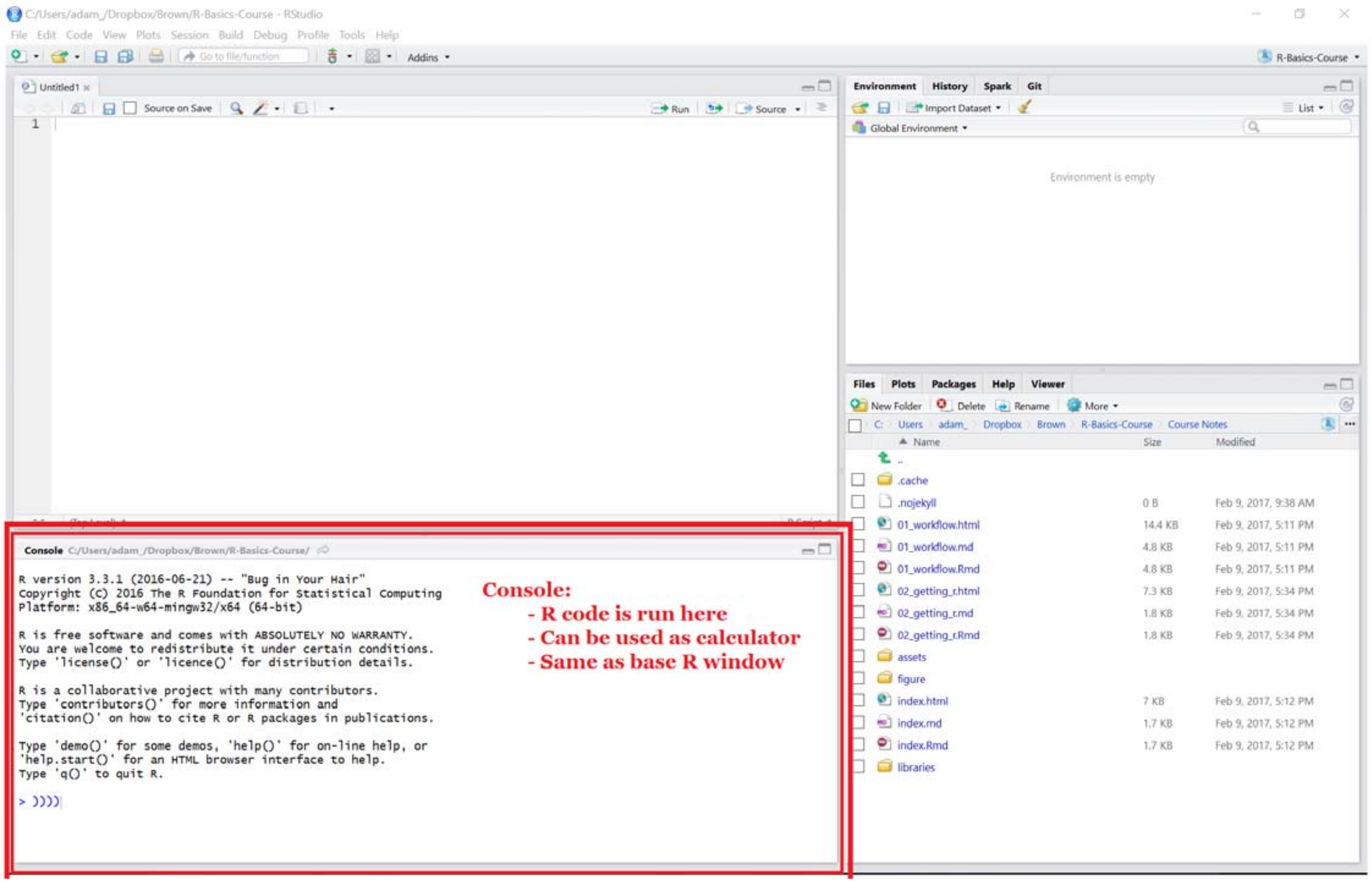


Source Window:
- Script Files
- Where you Code
- Create many documents

RStudio

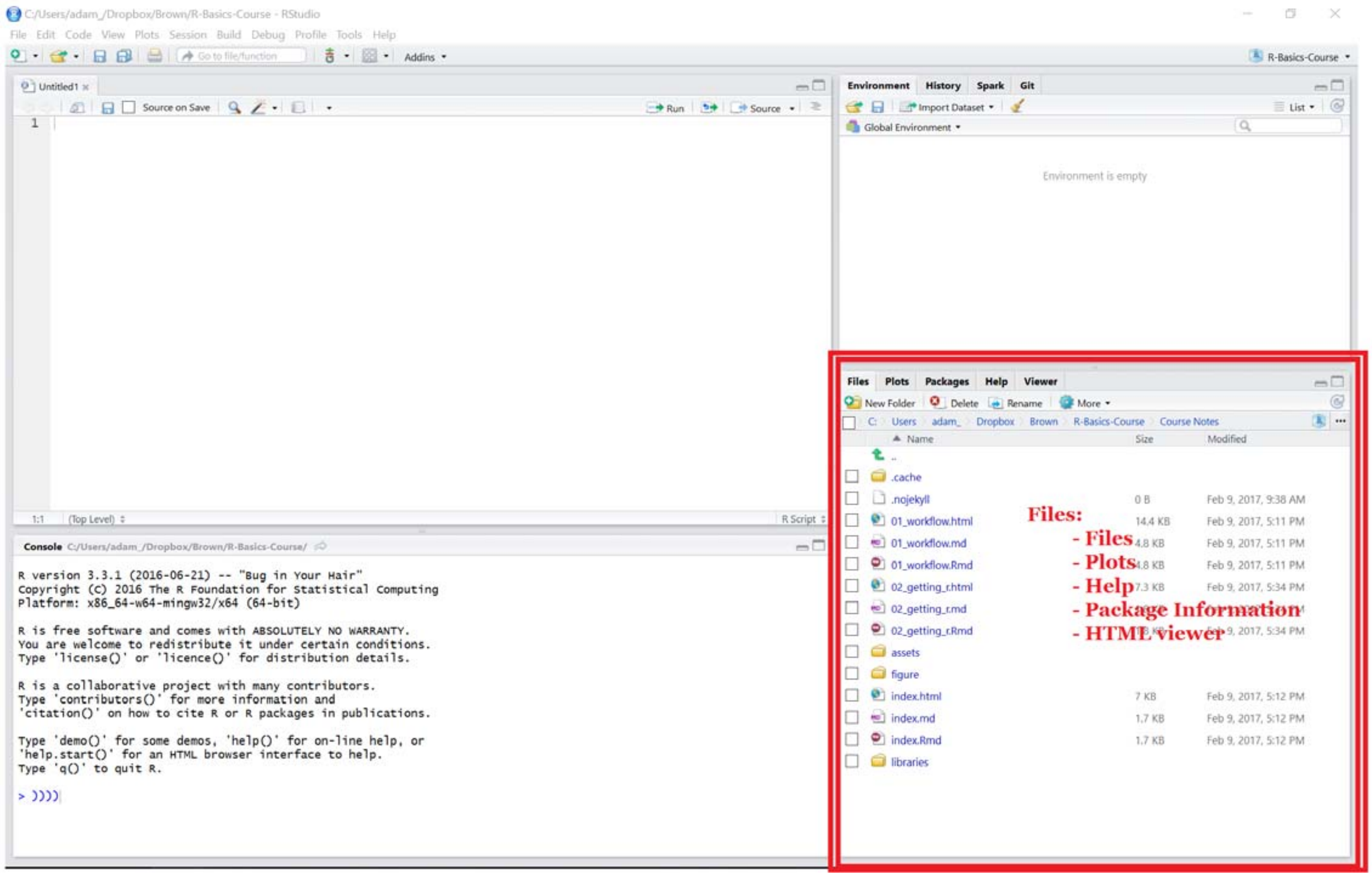

Environment:
- Shows open data and user functions
- History of Code Run
- Git Changes Viewed

RStudio



C:/Users/adam_/Dropbox/Brown/R-Basics-Course - RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Addins
R-Basics-Course
Untitled1
Source on Save
Run
Source
Environment History Spark Git
Import Dataset
List
Global Environment
Environment is empty
Files Plots Packages Help Viewer
New Folder Delete Rename More
C: Users adam_ Dropbox Brown R-Basics-Course Course Notes
Name Size Modified
.cache
.nojekyll 0 B Feb 9, 2017, 9:38 AM
01_workflow.html 14.4 KB Feb 9, 2017, 5:11 PM
01_workflow.md 4.8 KB Feb 9, 2017, 5:11 PM
01_workflow.Rmd 4.8 KB Feb 9, 2017, 5:11 PM
02_getting_r.html 7.3 KB Feb 9, 2017, 5:34 PM
02_getting_r.md 1.8 KB Feb 9, 2017, 5:34 PM
02_getting_r.Rmd 1.8 KB Feb 9, 2017, 5:34 PM
assets
figure
index.html 7 KB Feb 9, 2017, 5:12 PM
index.md 1.7 KB Feb 9, 2017, 5:12 PM
index.Rmd 1.7 KB Feb 9, 2017, 5:12 PM
libraries
Console C:/Users/adam_/Dropbox/Brown/R-Basics-Course/
R version 3.3.1 (2016-06-21) -- "Bug in Your Hair"
Copyright (C) 2016 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (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.
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.
>))))
Console:
- R code is run here
- Can be used as calculator
- Same as base R window

RStudio



Using R as a Calculator

Arithmetic Operators

OPERATOR	DESCRIPTION
+	Addition
-	Subtraction
*****	Multiplication
/	Division
**^ or ** **	Exponentiation

R as a Calculator

The most simple procedures that we can do in R is using R as a calculator. For example:

```
# Addition
5+4
```

```
## [1] 9
```

```
# Subtraction
124 - 26.82
```

```
## [1] 97.18
```

R as a Calculator

```
# Multiplication
5*4
```

```
## [1] 20
```

```
# Division
35/8
```

```
## [1] 4.375
```

More Math in R

R works simply as a calculator but also can be used for more advanced operations as well.

```
# Exponentials
3^(1/2)
```

```
## [1] 1.732051
```

```
# Exponential Function
exp(1.5)
```

```
## [1] 4.481689
```

More Math in R

```
# Log base e
log(4.481689)
```

```
## [1] 1.5
```

```
# Log base 10
log10(1000)
```

```
## [1] 3
```

Logical Operators

Once we have used some basic arithmetic operators we move into logic.

OPERATOR	**DESCRIPTION**
`<`	Less Than
`>`	Greater Than
`<=`	Less Than or Equal To
`>=`	Greater Than or Equal To
`==`	Exactly Equal To
`!=`	Not Equal To
`!a`	Not a
a&b	a **AND** b

Logic in R Example

We can then see an example of this:

```
a <- c(1:12)
# a>9 OR a<4
# Gives us 1 2 3 10 11 12

#Having R do this
a
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

```
a>9
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
## [12]  TRUE
```

Logic in R Example

```
a<4
```

```
##  [1]  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE
```

```
a>9 | a<4
```

```
##  [1]  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
## [12]  TRUE
```

```
a[a>9 | a<4]
```

```
## [1]  1  2  3 10 11 12
```

Further Operators

Some other operators we may want to use are listed below

DESCRIPTION	**R SYMBOL**	**EXAMPLE**
Comment	#	# This is a comment
DESCRIPTION	**R SYMBOL**	**EXAMPLE**
Assignment	<-	x <- 5
Assignment	->	5 -> x
Assignment	=	x = 5
Concatenation operator	c	c(1,2,4)
Modular	\%\%	25 \%\% 6
Sequence from a to b by h	seq	seq(a,b,h)
Sequence Operator	:	0:3

Math Functions in R

We also have access to a wide variety of mathematical functions that are already built into R.

DESCRIPTION	R SYMBOL
Square Root	`sqrt`
`floor(x)`	`floor`
`\ceil(x)`	`ceiling`
Logarithm	`log`
Exponential function, e^x	`exp`
Factorial, !	`factorial`

Getting Help in R

The `help()` Function

- To get online help within an R session we use the `help()` function.
- For example if we want to create a sequence and know that we can use the function `seq()` but are unsure of the arguments

Help Function Example

```
# help() function with seq as argument
help(seq)

# Shortcut for help() is ?
?seq
```

Further Help Funcion Use

We can also get help on characters and words by placing them in quotations

```
#Special characters < (all of these display the same information)
help("<")
help('<')
?"<"
?'<'

# Help for words (same use of quotations above work
?"for"
```

The `example()` Function

- Many times we just need to see some examples rather than read the entire documentation of a function or command.
- In this situation we would use the `example()` function

```
example(seq)
```

The `example()` Function

- We can then see numerous examples that R has run for us.
- The benefit of this command comes when you are interested in seeing examples of graphics, where just seeing the command and not the final product may not be as intuitive for us

```
#Example of Perspective Plots
example(persp)
```

The `help.search()` Function

The other help items are great if you know what function you are looking for. Many times we do not know exactly what we are looking for and need to do a more comprehensive search to find a proper function.

```
#Search for information about normal
help.search("normal")
```

Importing Data into R

Ways to get Data into R?

We use

- Built in Data
- .txt. .xls, ...
- SPSS, SAS, Stata
- Web Scraping
- Databases

Built in Data

- R has a wealth of data built in.
- We can use `data()` function to find it

Built in Data

- List all Datasets

```
data()
```

- Specific packages data

```
data(package="tidyr")
```

Delimited Files

- There are many packages out there which handle all of these things.
- We will stick to using the tidyverse packages.
- This will provide consistency with all we do.

`readr` in Tidyverse

- `readr` is a collection of many functions
    - `read_csv()`: comma separated (CSV) files
    - `read_tsv()`: tab separated files
    - `read_delim()`: general delimited files
    - `read_fwf()`: fixed width files
    - `read_table()`: tabular files where columns are separated by white-space.
    - `read_log()`: web log files
- `readxl` reads in Excel files.

Reading Delimited Files

- Many files are separated by delimiters.
- Common Ones are
    - comma (`,`)
    - semi-colon (`;`)
    - tab (or `\t`)
- We can use various functions to read these files in.

Reading Delimited Files

- In the third session we will use the following functions in practice:

    - `read.csv()`

    - `read.delim()`

Importing From Other Software

- R can read files from many other software types.
    - SAS
    - Stata
    - SPSS

Enter Haven Package

- `haven` is part of tidyverse.
- It contains the functions to read many different files.
- It can also write to those same data types.

For SAS

```
read_sas(data_file, catalog_file = NULL, encoding = NULL)

write_sas(data, path)
```

For Stata

```
read_dta(file, encoding = NULL)

read_stata(file, encoding = NULL)

write_dta(data, path, version = 14)
```

For SPSS

```
read_sav(file, user_na = FALSE)

read_por(file, user_na = FALSE)

write_sav(data, path)

read_spss(file, user_na = FALSE)
```

Tibbles in R

Tibbles

Previously we have worked with data in the form of

- Vectors
- Lists
- Arrays
- Dataframes

Tibbles

- *"Tibbles"* are a new modern data frame.
- It keeps many important features of the original data frame.
- It removes many of the outdated features.

Compared to Data Frames

- A *tibble* never changes the input type.
    - No more worry of characters being automatically turned into strings.
- A tibble can have columns that are lists.
- A tibble can have non-standard variable names.
    - can start with a number or contain spaces.
    - To use this refer to these in a backtick.
- It only recycles vectors of length 1.
- It never creates row names.

Column-Lists

```
library(tidyverse)
## Warning: package 'tidyverse' was built under R version 3.3.2
## Warning: package 'ggplot2' was built under R version 3.3.2
## Warning: package 'tidyr' was built under R version 3.3.2
try <- tibble(x = 1:3, y = list(1:5, 1:10, 1:20))
try
## # A tibble: 3 × 2
##       x          y
##   <int>     <list>
## 1     1  <int [5]>
## 2     2 <int [10]>
## 3     3 <int [20]>



#try <- as_data_frame(c(x = 1:3, y = list(1:5, 1:10, 1:20)))
#try
# Leads to error
```

Non-Standard Names

```
names(data.frame(`crazy name` = 1))
## [1] "crazy.name"
names(tibble(`crazy name` = 1))
## [1] "crazy name"
```

Coercing into Tibbles

- A tibble can be made by coercing `as_tibble()`.
- This works similar to `as.data.frame()`.
- It works efficiently.

Coercing into Tibbles

```
l <- replicate(26, sample(100), simplify = FALSE)
names(l) <- letters

microbenchmark::microbenchmark(
  as_tibble(l),
  as.data.frame(l)
)
## Unit: microseconds
##              expr      min       lq     mean    median        uq      max
##      as_tibble(l)  299.879  337.363  385.665  368.3775  411.6635  775.132
##  as.data.frame(l) 1357.485 1504.300 1747.447 1578.1535 1826.2675 3112.575
##  neval cld
##    100  a
##    100   b
```

Tibbles vs Data Frames

There are a couple key differences between tibbles and data frames.

- Printing.
- Subsetting.

Printing

- Tibbles only print the first 10 rows and all the columns that fit on a screen. - Each column displays its data type.
- You will not accidentally print too much.

```
tibble(
  a = lubridate::now() + runif(1e3) * 86400,
  b = lubridate::today() + runif(1e3) * 30,
  c = 1:1e3,
  d = runif(1e3),
  e = sample(letters, 1e3, replace = TRUE)
)
```

Printing

```
## # A tibble: 1,000 x 5
##                      a          b     c          d     e
##                 <dttm>     <date> <int>      <dbl> <chr>
## 1  2017-02-18 05:28:37 2017-03-08     1 0.02150370     f
## 2  2017-02-17 22:08:24 2017-03-08     2 0.08031493     k
## 3  2017-02-18 02:03:13 2017-03-07     3 0.11670172     u
## 4  2017-02-18 15:16:10 2017-03-08     4 0.24552337     h
## 5  2017-02-18 00:41:20 2017-03-04     5 0.11232662     b
## 6  2017-02-18 06:26:41 2017-03-08     6 0.52834632     m
## 7  2017-02-18 10:08:57 2017-03-15     7 0.78928491     v
## 8  2017-02-18 13:28:41 2017-03-15     8 0.80388276     h
## 9  2017-02-18 11:35:47 2017-03-18     9 0.45767339     d
## 10 2017-02-18 05:40:18 2017-02-24    10 0.18177950     t
## # ... with 990 more rows
```

Subsetting

- We can index a tibble in the manners we are used to
  - `df$x`
  - `df[["x"]]`
  - `df[[1]]`
- We can also use a `pipe` which we will learn about later.
  - `df %>% .$x`
  - `df %>% .[["x"]]`

Subsetting

```
df <- tibble(
  x = runif(5),
  y = rnorm(5)
)

df$x
## [1] 0.6227033 0.7363213 0.8551199 0.9173554 0.5542486
df[["x"]]
## [1] 0.6227033 0.7363213 0.8551199 0.9173554 0.5542486
df[[1]]
## [1] 0.6227033 0.7363213 0.8551199 0.9173554 0.5542486
```

Subsetting

```
df %>% .$x
## [1] 0.6227033 0.7363213 0.8551199 0.9173554 0.5542486
df %>% .[["x"]]
## [1] 0.6227033 0.7363213 0.8551199 0.9173554 0.5542486
df %>% .[[1]]
## [1] 0.6227033 0.7363213 0.8551199 0.9173554 0.5542486
```

Piping or Chaining Data

Piping or Chaining

- We will discuss a concept that will help us greatly when it comes to working with our data.
- The usual way to perform multiple operations in one line is by nesting.

Piping or Chaining

To consider an example we will look at the data provided in the 'fivethirtyeight' package:

```
library(fivethirtyeight)
drinks
```

```
## # A tibble: 193 x 5
##    country           beer_servings spirit_servings wine_servings total_li~
##    <chr>                     <int>           <int>         <int>     <dbl>
##  1 Afghanistan                   0               0             0     0
##  2 Albania                      89             132            54     4.90
##  3 Algeria                      25               0            14     0.700
##  4 Andorra                     245             138           312    12.4
##  5 Angola                      217              57            45     5.90
##  6 Antigua & Barbuda           102             128            45     4.90
##  7 Argentina                   193              25           221     8.30
##  8 Armenia                      21             179            11     3.80
##  9 Australia                   261              72           212    10.4
## 10 Austria                     279              75           191     9.70
## # ... with 183 more rows
```

Nesting vs Chaining

- Let's say that we want to consider the beer and wine drinking habits of Australia.
- Traditionally speaking we could do this in a nested manner:

```
filter(select(drinks, country, beer_servings, wine_servings), country=="Australia")
```

Nesting vs Chaining

- It is not easy to see exactly what this code was doing but we can write this in a manner that follows our logic much better.
- The code below represents how to do this with chaining.

```
drinks %>%
    select(country, beer_servings, wine_servings) %>%
    filter(country=="Australia")
```

Breaking Down the Code

- We now have something that is much clearer to read.
- Here is what our chaining command says:

1. Take the `drinks` data
2. Select the variables: `country`, `beer_servings` and `wine_servings`.
3. Only keep information from Australia.

- The nested code says the same thing but it is hard to see what is going on if you have not been coding for very long.

Breaking Down the Code

- The result of this search is below:

```
## # A tibble: 1 x 3
##   country   beer_servings wine_servings
##   <chr>             <int>         <int>
## 1 Australia           261           212
```

What is `%>%`

- In the previous code we saw that we used `%>%` in the command you can think of this as saying ***then***.
- For example:

```
drinks %>%
    select(country, beer_servings, wine_servings) %>%
    filter(country=="Australia")
```

What Does this Mean?

- This translates to:
    - Take drinks ***then*** select these columns country, beer_servings, wine_servings ***then*** filter out so we only keep Australia.

Why Chain?

- We still might ask why we would want to do this.
- Chaining increases readability significantly when there are many commands.
- With many packages we can replace the need to perform nested arguments.
- The chaining operator is automatically imported from the magrittr (https://github.com/smbache/magrittr) package.

User Defined Function

- Let's say that we wish to find the Euclidean distance between two vectors say, `x1` and `x2`.
- We could use the math formula:

$$\sqrt{sum(x1 - x2)^2}$$

User Defined Function

- In the nested manner this would be:

```
x1 <- 1:5; x2 <- 2:6
sqrt(sum((x1-x2)^2))
```

User Defined Function

- However, if we chain this we can see how we would perform this mathematically.

```
# chaining method
(x1-x2)^2 %>% sum() %>% sqrt()
```

- If we did it by hand we would perform elementwise subtraction of `x2` from `x1` ***then*** we would sum those elementwise values ***then*** we would take the square root of the sum.

User Defined Function

```
# chaining method
(x1-x2)^2 %>% sum() %>% sqrt()
```

```
## [1] 2.236068
```

- Many of us have been performing calculations by this type of method for years, so that chaining really is more natural for us.

User Defined Function

In the nested manner this would be:

```
x1 <- 1:5; x2 <- 2:6
sqrt(sum((x1-x2)^2))
```

However, if we chain this we can see how we would perform this mathematically.

```
# chaining method
(x1-x2)^2 %>% sum() %>% sqrt()
```

If we did it by hand we would perform elementwise subtraction of `x2` from `x1` ***then*** we would sum those elementwise values ***then*** we would take the square root of the sum.

User Defined Function

```
# chaining method
(x1-x2)^2 %>% sum() %>% sqrt()
```

```
## [1] 2.236068
```

Many of us have been performing calculations by this type of method for years, so that chaining really is more natural for us.

The `dplyr` Package

The `dplyr` Package

- We will begin with how we can start to manipulate data.
- We may wish to add additional variables.
- Perhaps we also wish to only look at data that meets a certain requirement.
- The `dplyr` package allows us to further work with our data.

`dplyr` Functionality

- With `dplyr` we have five basic verbs that we will learn to work with:

```
- filter()

- select()

- arrange()

- mutate()

- summarize()
```

Data We will Use

- For the purposes of this example we will consider looking at the package `fivethirtyeight`.
- We also will be using the `dyplr` package from `tidyverse`:

```
library(dplyr)
library(fivethirtyeight)
```

Filtering

- At this point we will consider how we pick the rows of the data that we wish to work with.
- If you consider many modern data sets, we have so much information that we may not want to bring it all in at once.
- We have not discussed exactly how R works at this point, however R brings data into the RAM of your computer.
- This means you can be limited for what size data you can bring in at once.
- Very rarely do you need the entire data set.
- We will focus on how to pick the rows or observations we want now.

Enter the `filter()` Function

- The `filter()` function chooses rows that meet a specific criteria.
- We can do this with Base R functions or with dplyr`.

The Data

- Lets consider the the data from the story:
- 41 Percent Of Fliers Think You're Rude If You Recline Your Seat (https://fivethirtyeight.com/features/airplane-etiquette-recline-seat/).
- This FiveThirtyEight data contains the following variables:

VARIABLE	DESCRIPTION
respondent_id	RespondentID
	gender
	age
	height
	children_under_18
	household_income
	education
	location
	frequency
	recline_frequency
	recline_obligation

Indexing

```
## Error in `[.tbl_df`(flying, recline_frequency == "Never", ): object 'recline_frequency' not found
```

Renaming Columns

- Many times we may wish to rename a column so that it makes more sense to us.
- The `select()` function can rename things for us as well.
- For example, there is a variable called `gender` in the `flying` data.
- This actually refers to binary sex and not what we know as gender. We could rename this to be:

```
flying %>%
  select(sex = gender)
```

Renaming Columns

```
## # A tibble: 1,040 x 1
##    sex
##    <chr>
##  1 <NA>
##  2 Male
##  3 Male
##  4 Male
##  5 Male
##  6 Male
##  7 Male
##  8 Male
##  9 <NA>
## 10 Male
## # ... with 1,030 more rows
```

Renaming Columns

- Note: We only kept the column of data that we renamed. If we had wanted to keep everything we could have used:

```
flying %>%
 select(sex = gender, everything())
```

Renaming Columns

Unique Observations

- Many times we have a lot of repeats in our data.
- If we just would like to have an account of all things included then we can use the `unique()` command.
- Let's assume that we just want each census region accounted for

```
flying %>%
  select(location) %>%
  unique()
```

Unique Observations

```
## # A tibble: 10 x 1
##    location
##    <chr>
##  1 <NA>
##  2 Pacific
##  3 East North Central
##  4 New England
##  5 Mountain
##  6 South Atlantic
##  7 East South Central
##  8 Middle Atlantic
##  9 West North Central
## 10 West South Central
```

Arranging the Data

- We also have need to make sure the data is ordered in a certain manner.
- This can be easily done in R with the `arrange()` function.
- Again we can do this in base R but this is not always a clear path.

Arranging the Data

- Let's say that we wish to look at only sex and frequency and we wish to order frequency from smallest to largest.
- In base R we would have to run the following command:

```
flying[order(flying$frequency), c("sex", "frequency")]
```

In this command we are ordering the rows by `frequency` and then only keeping `sex` and `frequency` in the end.

Enter the `arrange()` Function

We could do this in an easy manner using the `arrange()` function:

```
arrange(.data, ...)
```

Where

- `.data` is a data frame of interest.
- `...` are the variables you wish to sort by.

Arrange Function Example

```
flying %>%
    select(sex, frequency) %>%
    arrange(frequency)
```

Arrange Function Example

```
## # A tibble: 1,040 x 2
##    sex   frequency
##    <chr> <fctr>
##  1 Male  Never
##  2 Male  Never
##  3 Male  Never
##  4 Male  Never
##  5 Male  Never
##  6 Male  Never
##  7 Male  Never
##  8 Male  Never
##  9 Male  Never
## 10 Male  Never
## # ... with 1,030 more rows
```

Arrange Function Descending Example

- With `arrange()` we first use `select()` to pick the only columns that we want and then we arrange by the `dep_delay`.
- If we had wished to order them in a descending manner we could have simply used the `desc()` function:

```
flying %>%
    select(sex, frequency) %>%
    arrange(desc(frequency))
```

Arrange Function Descending Example

```
## # A tibble: 1,040 x 2
##    sex    frequency
##    <chr>  <fctr>
##  1 Male   Every day
##  2 Female Every day
##  3 Male   Every day
##  4 <NA>   A few times per week
##  5 <NA>   A few times per week
##  6 Male   A few times per week
##  7 Male   A few times per week
##  8 Male   A few times per month
##  9 Male   A few times per month
## 10 Male   A few times per month
## # ... with 1,030 more rows
```

More Complex Arrange

- Lets consider the a scenario where we want to look at people from each sex in each location and only consider the 3 highest when ranked by age.
- We then need to do the following:

1. Group by sex and location
2. Pick the top 3 ages
3. order them largest to smallest

More Complex Arrange

This can be done in the following manner:

```
flying %>%
  group_by(sex, location) %>%
  top_n(3, age) %>%
  arrange(desc(age))
```

More Complex Arrange

```
## # A tibble: 263 x 27
## # Groups: sex, location [20]
##    sex    respon~ age    height child~ hous~ educa~ loca~ freq~ recl~ recl~
##    <chr>    <dbl> <fctr> <fctr> <lgl>  <fct> <fctr> <chr> <fct> <fct> <lgl>
##  1 Male    3.43e9 > 60   "5'9\~ F      $50,~ Bache~ East~ Once~ Once~ T
##  2 Male    3.43e9 > 60   "6'5\~ F      $50,~ Bache~ West~ Once~ Never T
##  3 Male    3.43e9 > 60   "5'11~ F      $50,~ Some ~ West~ Once~ Alwa~ F
##  4 Male    3.43e9 > 60   "5'8\~ F      <NA>  Gradu~ Moun~ Once~ Never F
##  5 Male    3.43e9 > 60   "5'8\~ F      $100~ Some ~ West~ Once~ Abou~ F
##  6 Male    3.43e9 > 60   "5'7\~ F      $100~ Bache~ Moun~ Once~ Once~ F
##  7 Male    3.43e9 > 60   <NA>   NA     <NA>  Some ~ Midd~ Never <NA>  NA
##  8 Male    3.43e9 > 60   "5'11~ F      <NA>  Gradu~ Midd~ Once~ Once~ F
##  9 Male    3.43e9 > 60   "5'10~ F      <NA>  Bache~ Paci~ Once~ Once~ T
## 10 Male    3.43e9 > 60   "5'9\~ F      <NA>  Gradu~ Midd~ Once~ Never T
## # ... with 253 more rows, and 16 more variables: recline_rude <fctr>,
## #   recline_eliminate <lgl>, switch_seats_friends <fctr>,
## #   switch_seats_family <fctr>, wake_up_bathroom <fctr>, wake_up_walk
## #   <fctr>, baby <fctr>, unruly_child <fctr>, two_arm_rests <chr>,
## #   middle_arm_rest <chr>, shade <chr>, unsold_seat <fctr>, talk_stranger
## #   <fctr>, get_up <fctr>, electronics <lgl>, smoked <lgl>
```

Summarizing Data

- As you have seen in your own work, being able to summarize information is crucial.
- We need to be able to take out data and summarize it as well.
- We will consider doing this using the `summarise()` function.

Summarizing Data Example

Lets say we wish to:

1. Create a table grouped by `location`.
2. Summarize each group by taking mean of `recline_frequency`.

Summarizing Data Example Base R

```
head(with(flying, tapply(as.numeric(recline_frequency), location, mean, na.rm=TRUE)))
head(aggregate(as.numeric(recline_frequency)~location, flying, mean))
```

```
## East North Central East South Central    Middle Atlantic
##          2.647541          2.960000           3.090090
##          Mountain        New England            Pacific
##          2.629630          2.854545           3.021622
```

```
##             location as.numeric(recline_frequency)
## 1 East North Central                      2.647541
## 2 East South Central                      2.960000
## 3    Middle Atlantic                      3.090090
## 4           Mountain                      2.629630
## 5        New England                      2.854545
## 6            Pacific                      3.021622
```

Enter `summarise()` Function

The `summarise()` function is:

```
summarise(.data, ...)
```

where

- `.data` is the tibble of interest.
- `...` is a list of name paired summary functions

    - Such as:

        - `mean()`
        - `median`
        - `var()`
        - `sd()`
        - `min()`
        - `max()

Note: `summarise()` is Primarily useful with data that has been grouped by one or more variables.

Summarise Function Example

Our example:

```
flying %>%
    group_by(location) %>%
    summarise(avg_recline = mean(as.numeric(recline_frequency), na.rm=TRUE))
```

Consider the logic here:

1. Group observations by location.
2. Find the average recline frequency of the groups and call it `avg_recline`.

Summarise Function Example

This is much easier to understand than the Base R code and yields the table below:

```
## # A tibble: 10 x 2
##    location           avg_recline
##    <chr>                    <dbl>
##  1 East North Central        2.65
##  2 East South Central        2.96
##  3 Middle Atlantic           3.09
##  4 Mountain                  2.63
##  5 New England               2.85
##  6 Pacific                   3.02
##  7 South Atlantic            2.67
##  8 West North Central        2.68
##  9 West South Central        2.70
## 10 <NA>                      2.95
```

Another Example

Lets say that we would like to have more than just the averages but we wish to have the minimum and the maximum ages by location and:

```
flying %>%
    group_by(sex, location) %>%
    mutate(age_num = as.numeric(age)) %>%
    summarise_each(funs(min(., na.rm=TRUE), max(., na.rm=TRUE)), "age_num")
```

Another Example

```
## # A tibble: 21 x 4
## # Groups: sex [?]
##    sex    location           age_num_min age_num_max
##    <chr>  <chr>                    <dbl>       <dbl>
##  1 Female East North Central        1.00        4.00
##  2 Female East South Central        1.00        4.00
##  3 Female Middle Atlantic           1.00        4.00
##  4 Female Mountain                  1.00        4.00
##  5 Female New England               1.00        4.00
##  6 Female Pacific                   1.00        4.00
##  7 Female South Atlantic            1.00        4.00
##  8 Female West North Central        1.00        4.00
##  9 Female West South Central        1.00        4.00
## 10 Female <NA>                      2.00        4.00
## # ... with 11 more rows
```

Counting Events

There are various helper functions that we can use to count rows in a group. We will consider the following:

- `n()`
- `tally()`
- `n_distinct()`

Counting Events

Let's consider counting how many people are missing information on smoking when broken down by age and location.

```
flying %>%
    group_by(age, location) %>%
    summarise(person_count = n()) %>%
    arrange(desc(person_count))
```

Counting Events

```
## # A tibble: 41 x 3
## # Groups: age [5]
##    age    location           person_count
##    <fctr> <chr>                     <int>
##  1 30-44  Pacific                      60
##  2 18-29  Pacific                      56
##  3 > 60   Pacific                      54
##  4 > 60   South Atlantic               51
##  5 45-60  Middle Atlantic              48
##  6 45-60  Pacific                      48
##  7 45-60  South Atlantic               48
##  8 30-44  East North Central           42
##  9 45-60  East North Central           42
## 10 18-29  South Atlantic               38
## # ... with 31 more rows
```

Counting Events

We could also have used what is called the `tally()` function:

```
flying %>%
    group_by(age, location) %>%
    tally(sort = TRUE)
```

Counting Events

```
## # A tibble: 41 x 3
## # Groups: age [5]
##    age    location              n
##    <fctr> <chr>             <int>
##  1 30-44  Pacific              60
##  2 18-29  Pacific              56
##  3 > 60   Pacific              54
##  4 > 60   South Atlantic       51
##  5 45-60  Middle Atlantic      48
##  6 45-60  Pacific              48
##  7 45-60  South Atlantic       48
##  8 30-44  East North Central   42
##  9 45-60  East North Central   42
## 10 18-29  South Atlantic       38
## # ... with 31 more rows
```

Counting Unique Items

- The last way to count is if we want to find unique items.
- So we could look for one observation per age range and location that discusses how often one should get up during a flight.

```
flying %>%
    group_by(age,location) %>%
    summarise(person_count = n(), get_up_unique = n_distinct(get_up))
```

Counting Unique Items

```
## # A tibble: 41 x 4
## # Groups: age [?]
##    age    location           person_count get_up_unique
##    <fctr> <chr>                     <int>         <int>
##  1 18-29  East North Central           33             6
##  2 18-29  East South Central            7             3
##  3 18-29  Middle Atlantic              22             7
##  4 18-29  Mountain                     15             6
##  5 18-29  New England                  12             7
##  6 18-29  Pacific                      56             7
##  7 18-29  South Atlantic               38             6
##  8 18-29  West North Central           17             4
##  9 18-29  West South Central           19             6
## 10 18-29  <NA>                          1             1
## # ... with 31 more rows
```

Adding New Variables

- There is usually no way around needing a new variable in your data.
- For example, most medical studies have height and weight in them, however many times what a researcher is interested in using is Body Mass Index (BMI).
- We would need to add BMI in.

Using the `tidyverse` we can add new variables in multiple ways

- `mutate()`
- `transmute()`

The Mutate Function

With `mutate()` we have

```
mutate(.data, ...)
```

where

- `.data` is your tibble of interest.
- `...` is the name paired with an expression

The transmute Function

Then with `transmute()` we have:

```
transmute(.data, ...)
```

where

- `.data` is your tibble of interest.
- `...` is the name paired with an expression

Differences Between `mutate()` and `transmute()`

- There is only one major difference between `mutate()` and `transmutate` and that is what it keeps in your data.
- `mutate()`
  - creates a new variable
  - It keeps all existing variables
- `transmute()`
  - creates a new variable.
  - It only keeps the new variables

Mutate and Transmute Example

- Let's consider another dataset on drinking habits.
- Let's say we want to create a ratio of beer drinking to wine drinking. We can do this by:

$$\text{beer to wine ratio} = \frac{\text{beer servings}}{\text{wine servings}}$$

Mutate and Transmute Example

We can first do this with `mutate()`:

```
drinks %>%
  select(country, beer_servings, wine_servings) %>%
  mutate(beer_2_wine = beer_servings/wine_servings)
```

Mutate and Transmute Example

Notice with `mutate()` we kept all of the variables we selected and added beer to wine ratio to this.

```
## # A tibble: 193 x 4
##    country            beer_servings wine_servings beer_2_wine
##    <chr>                      <int>         <int>       <dbl>
##  1 Afghanistan                    0             0     NaN    
##  2 Albania                       89            54       1.65 
##  3 Algeria                       25            14       1.79 
##  4 Andorra                      245           312       0.785
##  5 Angola                       217            45       4.82 
##  6 Antigua & Barbuda            102            45       2.27 
##  7 Argentina                    193           221       0.873
##  8 Armenia                       21            11       1.91 
##  9 Australia                    261           212       1.23 
## 10 Austria                      279           191       1.46 
## # ... with 183 more rows
```

Mutate and Transmute Example

Now we can do the same with `transmute()`:

```
drinks %>%
  select(country, beer_servings, wine_servings) %>%
  transmute(beer_2_wine = beer_servings/wine_servings)
```

Mutate and Transmute Example

Now we can do the same with `transmute()`:

```
## # A tibble: 193 x 1
##    beer_2_wine
##          <dbl>
##  1     NaN
##  2       1.65
##  3       1.79
##  4       0.785
##  5       4.82
##  6       2.27
##  7       0.873
##  8       1.91
##  9       1.23
## 10       1.46
## # ... with 183 more rows
```