

R crash course - Data frames

Jongbin Jung

Dependencies

- ▶ Latest version of R (*free* from <https://www.r-project.org/>)
- ▶ Latest version of Rstudio (also *free* from <https://www.rstudio.com/>)
- ▶ A bunch of *free* packages

```
# The single tidyverse package now includes  
# dplyr and tidyr  
install.packages('tidyverse')  
# install.packages('dplyr')  
# install.packages('tidyr')  
# install.packages('readr')  
# install.packages('ggplot2')
```

```
## Warning: replacing previous import by 'tidyr::%>%'  
## when loading 'broom'
```

```
## Warning: replacing previous import by  
## 'tidyr::gather' when loading 'broom'
```

Data Frames: Introduction

- ▶ Data frames are the primary representation of data in R
- ▶ You can think of a data frame as a two-dimensional *table* of data
- ▶ It helps your sanity to always think of data frames as a table where

Each column represents a variable/feature

Each row represents an observation/instance

- ▶ Conceptually, a data frame is also a collection of vectors, i.e., each column is a vector that belongs to the (parent) data frame
- ▶ The fastest path to achieving R-ninja status is to get familiar with data frames

Data Frames: First Impression

- ▶ Let's take a look at an existing dataset, `txhousing` (loaded with `ggplot2` — which is loaded with `tidyverse`)
- ▶ Contains data (9 variables) on 8,602 observations about the housing market in Texas, from 2000 to 2015
- ▶ See documentation for details on what the 9 variables are

```
?txhousing
```

Data Frames: First Impression (cont'd)

```
str(txhousing) # take a peek at the data frame
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    8602 obs. of 10 variables:
## $ city      : chr  "Abilene" "Abilene" "Abilene" "Abilene" "Abilene" "Abilene" "Abilene" "Abilene" "Abilene" "Abilene"
## $ year      : int   2000  2000  2000  2000  2000  2000  2000  2000  2000  2000
## $ month     : int    1  2  3  4  5  6  7  8  9  10 ...
## $ sales     : num   72  98 130  98 141 156 152 131 104 101
## $ volume    : num  5380000 6505000 9285000 9730000 10590000
## $ median    : num   71400 58700 58100 68600 67300 66900 66500 66100 65700 65300
## $ listings  : num   701 746 784 785 794 780 742 765 771 777
## $ inventory : num    6.3 6.6 6.8 6.9 6.8 6.6 6.2 6.4 6.5 6.6
## $ date      : num   2000 2000 2000 2000 2000 2000 ...
```

Some Question

- ▶ What questions could you ask (and answer) with this data?
 - ▶ which cities had the highest/lowest volume in sales?
 - ▶ what was the annual/monthly sales per city?
 - ▶ are there monthly trends across multiple years in listings?
 - ▶ what else?
- ▶ By the end of this session, we'll have the tools to answer most (if not all) of the questions you can come up with!

Data Frame Basics

tibble

- ▶ A tibble is a trimmed down version of `data.frame` that is more convenient to work with
- ▶ Throughout this course, we will use the terms tibble and data frames interchangeably, which is technically incorrect, but easier to read

Simple Example

- ▶ Use `tibble()` function (from `tidyverse`) to create a tibble data frame
- ▶ Arguments of `tibble()` are vectors or lists (of equal length) that constitute each column (variable)
- ▶ For example, let's create a data frame of the following table:

Age	Personality	Income
24	Good	2000
22	Bad	5800
23	Good	4200
25	Bad	1500
22	Good	6000

Simple Example (cont'd)

- ▶ We'll save the data frame to an object (I'll call mine data)

```
data <- tibble( # start tibble()
  age = c(24, 22, 23, 25, 22),
  personality = c('g', 'b', 'g', 'b', 'g'),
  income = c(2000, 5800, 4200, 1500, 6000)
) # finish the tibble() function
```

- ▶ Note that the new lines are just a matter of coding style, i.e., it makes the code easier to read
- ▶ The same data frame can be created in a single line:

```
data <- tibble(age = c(24, 22, 23, 25, 22),
  personality = c('g', 'b', 'g', 'b', 'g'), income
= c(2000, 5800, 4200, 1500, 6000))
```

Simple Example (cont'd)

- ▶ Let's take a look at our new data frame

```
data
```

```
## # A tibble: 5 x 3
##   age personality income
##   <dbl> <chr>      <dbl>
## 1  24.0 g         2000
## 2  22.0 b         5800
## 3  23.0 g         4200
## 4  25.0 b         1500
## 5  22.0 g         6000
```

Indexing: The \$ Operator

- ▶ The \$ operator lets you reference elements of an object (e.g., column vectors of a data frame) in R

```
data$age
```

```
## [1] 24 22 23 25 22
```

```
data$personality
```

```
## [1] "g" "b" "g" "b" "g"
```

Indexing: The [Operator

- ▶ The [operator, similar to the \$ operator, lets you reference elements of an object (e.g., column vectors of a data frame) either by name *or by index*

```
data['age']
```

```
## # A tibble: 5 x 1
##   age
##   <dbl>
## 1  24.0
## 2  22.0
## 3  23.0
## 4  25.0
## 5  22.0
```

Indexing: The [Operator (cont'd)

```
data[1]
```

```
## # A tibble: 5 x 1
```

```
##   age
```

```
##   <dbl>
```

```
## 1  24.0
```

```
## 2  22.0
```

```
## 3  23.0
```

```
## 4  25.0
```

```
## 5  22.0
```

Indexing: The `[[` Operator (cont'd)

- ▶ Note that when using the `[]` operator, unlike `$`, you get a new data frame of the indexed column, *not* a vector.
- ▶ Use the `[[` operator if you want a vector instead

```
data[[1]]
```

```
## [1] 24 22 23 25 22
```

Indexing: Numeric Row/Column

- ▶ Since a data frame is a table of data, you can treat it like a matrix, and index its entries by [row #, col #] notation

```
data[2, 3] # item in row 2 column 3
```

```
## # A tibble: 1 x 1  
##   income  
##   <dbl>  
## 1   5800
```

```
data[, 2] # entire column 2
```

```
## # A tibble: 5 x 1  
##   personality  
##   <chr>  
## 1 g  
## 2 b  
## 3 g  
## 4 b  
## 5 g
```


Indexing: Named Variables

- ▶ Since the columns represent variables with names, you can index columns by a string representing variable names

```
data[, 'age'] # entire 'age' column
```

```
## # A tibble: 5 x 1
```

```
##   age
```

```
##   <dbl>
```

```
## 1  24.0
```

```
## 2  22.0
```

```
## 3  23.0
```

```
## 4  25.0
```

```
## 5  22.0
```

```
# entries 3~5 of 'personality' column
```

```
data[3:5, 'personality']
```

```
## # A tibble: 3 x 1
```

Indexing: Vectors

- ▶ As with vectors/matrices, you can index a data frame with vectors (either numeric or string)

```
data[1:3, c('age', 'income')]
```

```
## # A tibble: 3 x 2
##   age income
##   <dbl> <dbl>
## 1  24.0  2000
## 2  22.0  5800
## 3  23.0  4200
```

```
data[c(1, 4), 2:3]
```

```
## # A tibble: 2 x 2
##   personality income
##   <chr>         <dbl>
## 1 g           2000
```

Conditional Indexing

- Pick out entries that match specific criteria by first creating a binary vector for indexing

```
# find the 22-year-olds  
ind <- data$age == 22  
data[ind, ] # index rows by binary vector ind
```

```
## # A tibble: 2 x 3  
##   age personality income  
##   <dbl> <chr>      <dbl>  
## 1  22.0 b          5800  
## 2  22.0 g          6000
```

Chained Indexing

- ▶ Note that
 - ▶ when you index rows of a single column, the result is a vector
 - ▶ when you index multiple columns, the result is a new data frame
- ▶ You can chain indices to pin-point elements of a data frame
- ▶ For example, all of the following operations are (almost) equivalent – can you tell how the first two are different from the last two?

```
# (Almost) Equivalent operations to get the age of  
# third observation (row 3)  
data[3, 1] # if you know that 'age' is column 1  
data[3, 'age']  
data[3, ]$age # get 'age' of row 3  
data$age[3] # get third observation of 'age' variable
```

Column (Variable) Names

- ▶ To see the column name of a data frame, use the `colnames()` function
- ▶ The column names can be changed by directly assigning a new vector of names to the `colnames()` function

```
colnames(data)
```

```
## [1] "age"          "personality" "income"
```

```
colnames(data) <- c('age', 'attitude', 'income')  
data
```

```
## # A tibble: 5 x 3  
##   age attitude income  
##   <dbl> <chr>      <dbl>  
## 1  24.0 g         2000  
## 2  22.0 b         5800  
## 3  23.0 g         4200  
## 4  25.0 b         1500
```

Write Data Frames to Files

- ▶ Use `write_tsv()` (from the `readr` package) to write data frames to tab-separated (text) files
- ▶ The syntax is

```
write_tsv(x, path = "")
```

- ▶ For example, to save our sample data to a file named `data.tsv` with the entries of each row separated by a tab character, write

```
write_tsv(data, path = 'data.tsv')
```

- ▶ Recall, the default directory is the current working directory, specified with `setwd()`, and retrieved with `getwd()`
- ▶ For more options, see documentation

```
?write_tsv
```

Read Data Frames from Files

- ▶ To read data frames that exist as text files, use the general `read_table()` function
- ▶ Note that specific options for `read_table()` will depend on the structure of the text file you wish to read (e.g., comma-separated or tab-separated)
- ▶ Some shortcuts for pre-defined for (commonly used) formats

```
read_csv(file)           # comma-separated values
read_tsv(file)           # tab-separated values
read_delim(file, delim)  # custom delimiters
```

Read Data Frames from Files (cont'd)

- ▶ For example, to read the tsv file we just saved,

```
data <- read_tsv('data.tsv', col_names = TRUE)
```

```
## Parsed with column specification:  
## cols(  
##   age = col_integer(),  
##   attitude = col_character(),  
##   income = col_double()  
## )
```

- ▶ Notice how `read_tsv()` guesses the type of your columns
- ▶ It's good practice to specify column types manually so that
 1. you know exactly what you want, and
 2. you get what you want

Specify types for columns

- ▶ You can use the output message from the previous `read_tsv()` call as a boiler plate input to the `col_types` parameter, e.g.,

```
data <- read_tsv('data.tsv', col_types = cols(  
  age = col_double(),  
  personality = col_character(),  
  income = col_double()  
) , col_names = TRUE)
```

```
## Warning: The following named parsers don't match  
## the column names: personality
```

- ▶ Or, you can use a compact specification,

```
data <- read_tsv('data.tsv', col_types = 'dcd',  
  col_names = TRUE)
```

- ▶ Or, you can use a compact specification from the `col_types` package

Read Data from Online Database

- ▶ `read_*()` can also load data frames from an online database
- ▶ While loading data directly from the web is not recommended, this can be useful when making a local copy of an online database
- ▶ For example, to make a local copy of the dataset saved in <https://goo.gl/MGzatX>

```
address <- 'https://goo.gl/MGzatX'  
data <- read_tsv(address)  
write_tsv(data, path='data.tsv')
```

- ▶ Note that you can read data in one format (e.g., comma-separated) and save the local copy in another (e.g., tab-separated)

Exploring Data Frames

Display Structure with `str()`

- ▶ Let's start looking at the txhousing data frame
- ▶ The str() function is useful for exploring the overall structure of a data frame

```
str(txhousing)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':      8602 obs. of 10 variables:
## $ city      : chr  "Abilene" "Abilene" "Abilene" ..
## $ year      : int   2000  2000  2000  2000  2000  2000 ..
## $ month     : int    1  2  3  4  5  6  7  8  9 10 ...
## $ sales     : num   72  98 130  98 141 156 152 131 ..
## $ volume    : num  5380000 6505000 9285000 973000 ..
## $ median    : num   71400 58700 58100 68600 67300 ..
## $ listings  : num   701 746 784 785 794 780 742 7..
## $ inventory : num    6.3 6.6 6.8 6.9 6.8 6.6 6.2 6..
## $ date      : num  2000 2000 2000 2000 2000 ...
```

Factors

- ▶ Note that the city column should be a factor – a data frame representation of categorical variables
- ▶ Use `factor()` to force variables into factor variables

```
txhousing$city <- factor(txhousing$city)
```

- ▶ The entries of a factor variable is defined by `levels`

```
levels(txhousing$city)
```

unique values

- Use `unique()` to list the unique values of any variable

```
unique(txhousing$year)
```

```
##   [1] 2000 2001 2002 2003 2004 2005 2006 2007 2008  
##  [10] 2009 2010 2011 2012 2013 2014 2015
```

Data Frame Basics: Exercise

- ▶ From the `txhousing` data
 - ▶ create a new data frame for the city "Paris" (i.e., `city == "Paris"`)
 - ▶ generate a `summary()` of the inventory in Paris
- ▶ These are just (very) basic operations
- ▶ For more complicated operations, we'll use `dplyr` and `tidyr` (both part of `tidyverse` and covered next)

Exercise Solution

WARNING

- ▶ Solutions to the exercise are presented in the next slide
- ▶ Try the exercise before proceeding!

Solution

```
ind_paris <- txhousing$city == "Paris"  
paris <- txhousing[ind_paris, ]  
summary(paris$inventory)
```

```
##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.   
##  4.500   8.775   10.000   10.170   11.600   14.500   
##      NA's   
##         19
```

Wrangling Data with dplyr

Introduction to dplyr

- ▶ dplyr is a package that provides a convenient framework (along with a handful of useful functions) for wrangling data (frames)
- ▶ dplyr is a part of the tidyverse, so if you load tidyverse, dplyr is also loaded
- ▶ You can also, but don't have to, install and load the dplyr as a standalone package like you would any other R package

```
# Only need to do this once on a single machine.  
install.packages("dplyr")  
# load package into workspace  
library("dplyr")
```

Data Frames: First Impression

- ▶ We'll primarily use the `diamonds` data that's included with `tidyverse`
- ▶ Take a look by typing `diamonds` in the R console
- ▶ `diamonds` is a dataset containing the prices and other attributes of almost 54,000 diamonds. Included variables are:
 - ▶ price, carat, cut, color, clarity, dimensions (x, y, z, depth, table)
- ▶ See documentation for more details

```
?diamonds
```

Verbs

- ▶ A *verb* in the world of `dplyr` is a function that
 - ▶ takes a data frame as its first argument, and
 - ▶ returns another data frame as a result
- ▶ For example, the `head()` function can be considered a verb

```
head(diamonds, n = 3)
```

- ▶ Note that the result of the `head()` function is another data frame (in this case, with 3 rows)
- ▶ The **core idea of `dplyr`** is that most of your data manipulation needs can be satisfied with 5 basic verbs (or 4, depending on how you categorize them)

Five basic verbs

- ▶ The five basic verbs of dplyr and associated actions are presented below

verb	action
<code>filter()</code>	select a subset of <i>rows</i> by specified conditions
<code>select()</code>	select a subset of <i>columns</i>
<code>mutate()</code>	create a <i>new column</i> (often from existing columns)
<code>arrange()</code>	reorder (sort) <i>rows</i> by values of specified <i>column(s)</i>
<code>summarize()</code>	aggregate values and reduce to single value

- ▶ Some verbs have additional options or convenient wrappers

Selecting rows (1/5)

Selecting Rows: `filter()`

- ▶ Select a subset of *rows*
- ▶ Multiple conditions can be used
- ▶ Use `&` to specify AND conditions
- ▶ Use `|` to specify OR conditions
- ▶ `AND(&)/OR(|)` operations can be used together (where default behavior for multiple conditions is AND)

```
filter(diamonds, cut == "Ideal" & carat > 3)
filter(diamonds, carat <= .2 | carat >= 3)
filter(diamonds,
       carat <= .2 | carat >= 3, color == "D")
```

- ▶ Use `%in%` to match on a collection of values

```
filter(diamonds, cut %in% c("Fair", "Ideal"))
```

Selecting Rows: `slice()`

- ▶ To select rows by numerical index (position), use `slice()`
- ▶ For example, to select the first 10 rows

```
slice(diamonds, 1:10)
```

- ▶ or to select the last 10 rows

```
slice(diamonds, (n() - 9):n())
```

- ▶ Use `n()` inside a dplyr verb to indicate the *number of rows* of the data frame

Selecting columns (2/5)

Selecting Columns: `select()`

- ▶ Select a subset of *columns*
- ▶ Either specify the columns that you want to select

```
select(diamonds, cut, price)
```

- ▶ Or specify the columns you wish to drop

```
select(diamonds, -x, -y, -z)
```

Selecting Columns: `select()` (cont'd)

- ▶ `dplyr` provides useful helper functions you can use to `select()` columns that match specific criteria such as
 - ▶ `starts_with(x)`: names that start with `x`
 - ▶ `ends_with(x)`: names that end with `x`
 - ▶ `contains(x)`: names that contain `x`
 - ▶ `matches(x)`: names that match the (regular expression) `x`
- ▶ See the documentation for more details

```
?dplyr::select
```

- ▶ While you can assign new column names with `select()` the convenience function `rename()` lets you rename columns while retaining the rest of the data frame

```
rename(diamonds, cut_type = cut)
```

Creating new columns (3/5)

Create New Columns: `mutate()`

- ▶ Create new columns, usually as a function of existing columns
- ▶ You can refer to new columns you just created, inside the same `mutate()` function

```
mutate(diamonds,  
       price_per_carat = price / carat,  
       volume = x * y * z,  
       # Use the volume column we just created  
       # to create a price_per_volume column  
       price_per_volume = price / volume)
```

- ▶ Use `transmute()` to create a new data frame *just from* the new column(s)

```
transmute(diamonds, carat,  
          price_per_carat = price / carat)
```

Sorting (4/5)

Sorting Rows by Column Value: `arrange()`

- ▶ Reorder the rows of a data frame by the specified column's value
- ▶ Multiple conditions are arranged from left to right
- ▶ Use `desc()` to arrange in descending order

```
arrange(diamonds, carat, price)
arrange(diamonds, carat, desc(price))
arrange(diamonds, desc(carat), desc(price))
```

Aggregating (5/5)

Aggregate Data: summarize()

- ▶ Aggregate/collapse the data into a single row
- ▶ Think of as applying a function to columns

```
summarize(diamonds, avg_price = mean(price))
```

```
## # A tibble: 1 x 1  
##   avg_price  
##   <dbl>  
## 1      3933
```

Recap:

- ▶ The five basic verbs:

verb	action
<code>filter()</code>	select a subset of <i>rows</i> by specified conditions
<code>select()</code>	select a subset of <i>columns</i>
<code>mutate()</code>	create a <i>new column</i> (often from existing columns)
<code>arrange()</code>	reorder (sort) <i>rows</i> by values of specified <i>column(s)</i>
<code>summarize()</code>	aggregate values and reduce to single value

- ▶ But what about ...
 - ▶ Average price of diamonds for each cut type?
 - ▶ Largest (carat) diamond in each color category?
 - ▶ ...

Bad example:

- ▶ A natural, but tedious way to compute:
- ▶ Average price of diamonds for each cut type?
 - ▶ use `filter` to create five different data frames, one for each cut type
 - ▶ use `summarize` to compute the mean price for each data frame
- ▶ Largest (carat) diamond in each color category?
 - ▶ use `filter` to create seven different data frames, one for each color category
 - ▶ use `arrange` to sort in descending order of carat for each data frame
 - ▶ use `slice` to get the first row from each of the arranged data frames
- ▶ The pattern:
 - ▶ split the data, grouping by some categorical value
 - ▶ do some operations, but to each category of the group

Split-Apply-Combine

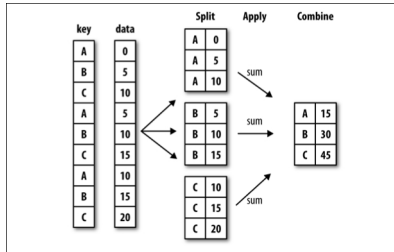


Figure 1: Illustration of SAC

Grouped Operations

- ▶ If a data frame is *grouped*, operations are applied to each group separately, and the results are combined back to a single data frame
- ▶ The `group_by()` verb lets you specify the *grouping* variables (e.g., `cut`, `color`)

```
diamonds_by_cut <- group_by(diamonds, cut)
```

- ▶ When the data frame is `group_by`'d, all verbs/functions will operate as if each category of the grouping variable is its own data frame, e.g.,

verb	group specific action
<code>arrange()</code>	sort rows within each group
<code>slice()</code>	extract rows within each group
<code>summarize()</code>	aggregate values group-wise
<code>n()</code>	count the number of rows in each group

Grouped slice()

- Retrieve one diamond from each cut

```
slice(diamonds_by_cut, 1)
```

```
## # A tibble: 5 x 10
## # Groups:   cut [5]
##   carat cut    color clar~ depth table price     x
##   <dbl> <ord> <ord> <ord> <dbl> <dbl> <int> <dbl>
## 1 0.220 Fair  E     VS2    65.1  61.0   337  3.87
## 2 0.230 Good  E     VS1    56.9  65.0   327  4.05
## 3 0.240 Very~ J     VVS2   62.8  57.0   336  3.94
## 4 0.210 Prem~ E     SI1    59.8  61.0   326  3.89
## 5 0.230 Ideal E     SI2    61.5  55.0   326  3.95
## # ... with 2 more variables: y <dbl>, z <dbl>
```


Grouped summarize()

- Retrieve (1) number of diamonds and (2) average price by cut type

```
summarize(diamonds_by_cut,  
          count = n(),  
          avg_price = mean(price))
```

```
## # A tibble: 5 x 3  
##   cut      count avg_price  
##   <ord>    <int>    <dbl>  
## 1 Fair      1610      4359  
## 2 Good      4906      3929  
## 3 Very Good 12082      3982  
## 4 Premium   13791      4584  
## 5 Ideal     21551      3458
```

Multiple (Chained) Operations

Multiple (Chained) Operations

- ▶ Wish to compute: Proportion of different colors in each cut category
- ▶ We can achieve the desired result with a few operations
 1. `group_by` cut
 2. `mutate` to create a column with total diamonds for each cut
 3. `re-group_by` cut and color
 4. Use the new column from above to `summarize` the proportion of each color within each cut
- ▶ Note that `dplyr` verbs do **not** modify the original data frame (i.e., they don't have side effects)
 - ▶ This is generally a good thing, since it guarantees the integrity of your data
 - ▶ But it makes multiple operations on a data frame difficult

Multiple Operations: The OK Way

- ▶ One way to perform multiple operations is to save intermediate data frames as new data frames
- ▶ This method delivers desired results, but makes your workspace quite messy (i.e., you'll end up with a workspace full of intermediate results)

```
by_cut <- group_by(diamonds, cut)
count_cuts <- mutate(by_cut, N = n())
by_cut_and_color <- group_by(count_cuts, cut, color)
proportions <- summarize(by_cut_and_color,
                          prop = mean(n()/N))
```

- ▶ This method might be preferred if you need the intermediate results in the future
- ▶ If not, there is a better way to chain multiple operations with `dplyr`

The Pipe Operator %>%

- ▶ The pipe operator, takes the output from its left-hand side, and uses it as the first argument to whatever is on its right-hand side¹
- ▶ For example:

```
by_cut <- group_by(diamonds, cut)
count_cuts <- mutate(by_cut, N = n())
```

is equivalent to

```
count_cuts <- diamonds %>%
  group_by(cut) %>%
  mutate(N = n())
```

(except in the second case, the `by_cut` data frame is never created)

¹Ctrl + Shift + M will insert %>% in Rstudio

The Pipe Operator %>%

- ▶ Using the pipe, we can complete the entire task without saving any intermediate data frames

```
proportions <- diamonds %>%  
  group_by(cut) %>%  
  mutate(N = n()) %>%  
  group_by(cut, color) %>%  
  summarize(prop = mean(n()/N))
```

- ▶ No need to save intermediate results
- ▶ Easier to read (i.e., you can follow the operations step-by-step without too much mental accounting)

dplyr: Exercise

- ▶ Find the most expensive diamond for each cut.
- ▶ How many 1 carat diamonds are “Premium” cut, and what are the min/median/max prices?
- ▶ What is the average price for diamonds grouped by 0.1 carats?

Exercise Solution

WARNING

- ▶ Solutions to the exercise are presented in the next slide
- ▶ Try the exercise before proceeding!

Solutions

- Find the most expensive diamond for each cut.

```
diamonds %>%  
  group_by(cut) %>%  
  arrange(desc(price)) %>%  
  slice(1)
```

```
## # A tibble: 5 x 10  
## # Groups:   cut [5]  
##   carat cut    color clar~ depth table price      x  
##   <dbl> <ord> <ord> <ord> <dbl> <dbl> <int> <dbl>  
## 1  2.01 Fair  G      SI1    70.6  64.0 18574  7.43  
## 2  2.80 Good  G      SI2    63.8  58.0 18788  8.90  
## 3  2.00 Very~ G      SI1    63.5  56.0 18818  7.90  
## 4  2.29 Prem~ I      VS2    60.8  60.0 18823  8.50  
## 5  1.51 Ideal G      IF     61.7  55.0 18806  7.37  
## # ... with 2 more variables: y <dbl>, z <dbl>
```

Solutions

- How many 1 carat diamonds are “Premium” ‘cut?

```
diamonds %>%  
  filter(carat == 1, cut == "Premium") %>%  
  summarize(N = n(),  
            min = min(price),  
            med = median(price),  
            max = max(price))
```

```
## # A tibble: 1 x 4  
##       N     min   med   max  
##   <int> <dbl> <dbl> <dbl>  
## 1    462  1681  5118 10752
```

Solutions

- What is the average price for diamonds grouped by 0.1 carats?

```
diamonds %>%  
  mutate(carat_bin = round(carat, digits = 1)) %>%  
  group_by(carat_bin) %>%  
  summarize(avg_price = mean(price))
```

```
## # A tibble: 38 x 2  
##   carat_bin avg_price  
##   <dbl>     <dbl>  
## 1    0.200       506  
## 2    0.300       703  
## 3    0.400       923  
## 4    0.500      1590  
## 5    0.600      1872  
## 6    0.700      2623  
## 7    0.800      2998  
## 8    0.900      3940
```

Reference

- ▶ Rstudio comes loaded with a bunch of cheatsheets: see [Help] > [Cheatsheets]
- ▶ Introductory text book by the creator of tidyverse:
<http://r4ds.had.co.nz/>

Reshape Data with tidyr

Introduction to tidyr

- ▶ Recall, the preferred way to think about a data frame:
Each column represents a variable/feature
Each row represents an observation/instance
- ▶ Consider the following (fictional) data frame of students' homework grades

ID	HW1	HW2	HW3
jamie	6	7	3
cersei	8	5	2
hodor	9	10	9

- ▶ What are the variables of this data?
- ▶ What are the potential issues with this representation?
- ▶ What are the benefits of this representation?

Introduction to tidyr: An Example

- ▶ For data manipulation/visualization we often prefer to have data in the *long* form
- ▶ The *long* form of the previous data would be

ID	HW	Score
jamie	1	6
jamie	2	7
⋮	⋮	⋮
hodor	2	10
hodor	3	9

- ▶ `tidyr` is a package that provides a tools for converting data between *long* and *wide* forms

Introduction to tidyr: Getting Started

- ▶ tidyr is also part of the tidyverse, so if you load tidyverse, tidyr is also loaded
- ▶ You can also, but don't have to, install and load tidyr as a standalone package like you would any other R package

```
# Only need to do this once on a single machine.  
install.packages('tidyr')  
# load package into workspace  
library('tidyr')
```

Introduction to tidyr: Getting Started (cont'd)

- Create some random data

```
grades <- tibble(  
  ID = c('jamie', 'cersei', 'hodor'),  
  HW1 = runif(3, 1, 10),  
  HW2 = runif(3, 1, 10),  
  HW3 = runif(3, 1, 10),  
  info = c('male/lannister',  
           'female/lannister',  
           'male/stark'))
```

```
grades
```

```
## # A tibble: 3 x 5
```

```
##   ID      HW1   HW2   HW3 info
```

```
##   <chr> <dbl> <dbl> <dbl> <chr>
```

```
## 1 jamie   3.84  9.05  5.18 male/lannister
```

```
## 2 cersei  5.36  9.43  1.28 female/lannister
```

```
## 3 hodor   3.25  7.86  5.15 male/stark
```

Create *long* data with `gather()`

- ▶ With the grades data, we would like to create a data frame in the form of

ID	info	HW	score
-	-	-	-

- ▶ The verb for gathering multiple columns into key-value pairs in `tidyr` is `gather()`
- ▶ The syntax is

```
gather(data, key, value, ...)
```

- ▶ where the `...` should be replaced by column specifications

Create *long* data with `gather()` (cont'd)

```
grades_tidy <- gather(grades, HW, score, HW1:HW3)
grades_tidy
```

```
## # A tibble: 6 x 4
##   ID      info      HW      score
##   <chr> <chr>    <chr> <dbl>
## 1 jamie  male/lannister HW1    3.84
## 2 cersei female/lannister HW1    5.36
## 3 hodor  male/stark    HW1    3.25
## 4 jamie  male/lannister HW2    9.05
## 5 cersei female/lannister HW2    9.43
## 6 hodor  male/stark    HW2    7.86
```

Split a Column to Multiple Variables with `separate()`

- ▶ Often, there will be column that you'd prefer to split into multiple variables, e.g., splitting a date column to year, month, and day
- ▶ From the grades data, notice that the info column combines two variables: sex and house
- ▶ We can split such columns to multiple variables with the `separate()` verb in `tidyr`

Split a Column to with separate() (cont'd)

```
grades_split <- separate(  
  grades_tidy, info,  
  into = c('sex', 'house'),  
  sep = '/'  
)  
grades_split
```

```
## # A tibble: 3 x 5  
##   ID      sex    house    HW    score  
##   <chr> <chr>   <chr>   <chr> <dbl>  
## 1 jamie  male   lannister HW1    3.84  
## 2 cersei female lannister HW1    5.36  
## 3 hodor  male    stark    HW1    3.25
```

Chaining tidyr verbs

- ▶ Note that tidyr operations are also verbs that can be chained with the pipe operator %>%
- ▶ For example, we can do the previous operations on the grades data with the chained operation

```
grades_final <- grades %>%  
  gather(HW, score, HW1:HW3) %>%  
  separate(info, into=c('sex', 'house'), sep='/')
```

- ▶ tidyr verbs can also be chained with dplyr verbs (and any other function that qualifies as a verb, i.e., takes a data frame as the first argument and results in a new data frame)

Exercise

- ▶ Download and load some toy data

```
address <- "https://goo.gl/0hFk2w"  
finance <- read_tsv(address)
```

1. Tidy the data to fit the form

ID	type	year	amount
-	-	-	-

2. Find the mean and total Income/Expense for each ID across all years

Exercise Solution

WARNING

- ▶ Solutions to the exercise are presented in the next slide
- ▶ Try the exercise before proceeding!

Solution 1

```
finance <- finance %>%  
  gather(year, amount, 2:7) %>%  
  separate(year, c('type', 'year'), sep='_')  
finance
```

```
## # A tibble: 8 x 4  
##   ID      type    year amount  
##   <chr> <chr>   <chr>  <dbl>  
## 1 leia   Income  2013    4174  
## 2 han    Income  2013   11065  
## 3 luke   Income  2013   11342  
## 4 leia   Income  2014   17344  
## 5 han    Income  2014   - 5024  
## 6 luke   Income  2014    3547  
## 7 leia   Income  2015    3028  
## 8 han    Income  2015   - 757
```

Solution 2

```
finance_summary <- finance %>%  
  group_by(ID, type) %>%  
  summarize(mean=mean(amount), total=sum(amount))  
finance_summary
```

```
## # A tibble: 6 x 4  
## # Groups:   ID [?]  
##   ID      type      mean total  
##   <chr> <chr>    <dbl> <dbl>  
## 1 han    Expense   8170 24509  
## 2 han    Income   1761  5284  
## 3 leia   Expense   8818 26453  
## 4 leia   Income   8182 24546  
## 5 luke   Expense  12820 38461  
## 6 luke   Income   8803 26408
```

Combining Data with joins

Basic concatenations

- ▶ `rbind`: concatenate rows
- ▶ `cbind`: concatenate columns
- ▶ For data frames, `bind_rows()` and `bind_cols()` from `dplyr` is usually much faster.

Example (with matrix, but works with data frames too)

```
A <- matrix(1:4, 2, 2)
B <- matrix(5:8, 2, 2)
rbind(A, B)
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
## [3,]    5    7
## [4,]    6    8
```

```
cbind(A, B)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5    7
## [2,]    2    4    6    8
```

Introduction to joins

- ▶ Sometimes, you will find relevant data across multiple datasets, e.g., a list of grades in one dataset and a list of contact information in another
- ▶ In such cases, you may want to join the two datasets into a single data frame prior to further analysis
- ▶ For a successful join you must determine
 1. One or more variables that uniquely identify matching observations (rows) in both datasets
 2. How you wish to join the data, i.e.,
 - ▶ **Left/right join** Retain one of the datasets entirely, while only keeping matching entries of the other, possibly resulting in a few missing values (NA)
 - ▶ **Inner join** Retain only complete matches, possibly dropping some rows of both datasets
 - ▶ **Outer (full) join** Retain all rows of both datasets, but potentially resulting in many missing values (NA)

Example Datasets

- For illustration, let's create two data frames

```
info <- tibble(  
  name=c('tony', 'tony', 'rey'),  
  job=c('scientist', 'tiger', 'scavenger'),  
  score=rnorm(3)  
)  
  
power <- tibble(  
  name=c('tony', 'hank', 'rey'),  
  job=c('scientist', 'scientist', 'scavenger'),  
  strength=rexp(3)  
)
```

- How is an observation (row) uniquely identified?

Left/Right join

- Retain rows of one dataset, and match the rows of the other

```
left_join(info, power, by=c('name', 'job'))
```

```
## # A tibble: 3 x 4
##   name  job      score strength
##   <chr> <chr>    <dbl>    <dbl>
## 1 tony  scientist -0.181    0.180
## 2 tony  tiger      0.263    NA
## 3 rey   scavenger  1.13     0.0408
```

```
right_join(info, power, by=c('name', 'job'))
```

```
## # A tibble: 3 x 4
##   name  job      score strength
##   <chr> <chr>    <dbl>    <dbl>
## 1 tony  scientist - 0.181    0.180
## 2 hank  scientist  NA        2.25
```

Inner join

- Retain only the rows that have matches on both datasets

```
inner_join(info, power, by=c('name', 'job'))
```

```
## # A tibble: 2 x 4
##   name  job      score strength
##   <chr> <chr>    <dbl>    <dbl>
## 1 tony  scientist -0.181    0.180
## 2 rey   scavenger  1.13     0.0408
```

Outer (Full) join

- Retain all rows

```
full_join(info, power, by=c('name', 'job'))
```

```
## # A tibble: 4 x 4
##   name  job      score strength
##   <chr> <chr>    <dbl>    <dbl>
## 1 tony  scientist - 0.181    0.180
## 2 tony  tiger      0.263    NA
## 3 rey   scavenger  1.13     0.0408
## 4 hank  scientist  NA       2.25
```

Tip: Replace NA entries

- ▶ To replace *all* NA entries in a data frame — assuming you know exactly what you want to change them to! — index by `is.na` and re-assign, e.g.,

```
everyone <- full_join(info, power, by=c('name', 'job'))  
  
# Replace the NAs with 0 (BAD IDEA!)  
everyone[is.na(everyone)] <- 0  
everyone
```

```
## # A tibble: 4 x 4  
##   name  job      score strength  
##   <chr> <chr>    <dbl>    <dbl>  
## 1 tony  scientist -0.181    0.180  
## 2 tony  tiger     0.263     0  
## 3 rey   scavenger 1.13     0.0408  
## 4 hank  scientist 0         2.25
```

Reference

- ▶ A great “cheat sheet” for wrangling data with `dplyr` and `tidyr` is available for free at <https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf>