Intro to `dplyr`

Jongbin Jung

Wrangling Data with `dplyr`

Exercise Solution

Reshape Data with `tidyr`

Exercise Solution

Combining Data with `joins`

Dependencies

- Latest version ($\geq$ 3.1.2) of R (*free* from https://www.r-project.org/)
- Latest version of Rstudio (also *free* from https://www.rstudio.com/)
- tidyverse package

```
# tidyverse package now includes
install.packages("tidyverse")
```

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 load an existing data frame to take a look at

```
# install data package (only need to do once)
install.packages("nycflights13")
```

```
# load data package to workspace
library("nycflights13")
```

- The `nycflights13` package contains a single data frame named `flights`
- Contains data (16 variables) on all 336,776 flights that departed NYC (i.e. JFK, LGA, or EWR) in 2013
- See documentation for details on what the 16 variables are

```
?flights
```

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

```
# Install, if you haven"t already.
# Only need to do this once on a single machine.
install.packages("dplyr")
# load package into workspace
library("dplyr")
```

- We"ll primarily use the diamonds data that"s included with tidyverse

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(flights, n = 10)
```

- 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
`filter()`	select a subset of *rows* by specified conditions
`select()`	select a subset of *columns*
`mutate()`	create a *new column* (often from existing columns)
`arrange()`	reorder (sort) *rows* by values of specified *column*(s)
`summarize()`	aggregate values and reduce to single value

- Some verbs have additional options or convenient wrappers

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 == "J")
```

- 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(flights, 1:10)
```

- or to select the last 10 rows

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

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

Selecting Columns: `select()`

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

```
select(flights, carrier, tailnum)
```

- Or specify the columns you wish to drop

```
select(flights, -year, -month, -day)
```

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

```
select(flights, tail_num = tailnum)
rename(flights, tail_num = tailnum)
```

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(flights, gain = arr_delay - dep_delay,
       speed = distance / air_time * 60,
       # use the gain column we just created
       # to create yet another gain_per_hour column
       gain_per_hour = gain / (air_time / 60)
       )
```

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

```
transmute(flights, gain = arr_delay - dep_delay)
```

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(flights, year, month, day)
arrange(flights, year, desc(month), day)
arrange(flights, year, month, desc(day))
arrange(flights, year, desc(month), desc(day))
```

Aggregate Data: `summarize()`

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

```
summarize(flights, delay = mean(dep_delay))
# Note that the mean function need help
# handling NA values
summarize(flights,
          delay = mean(dep_delay, na.rm = TRUE))
```

- More useful as a grouped operation (see next)

Split-Apply-Combine

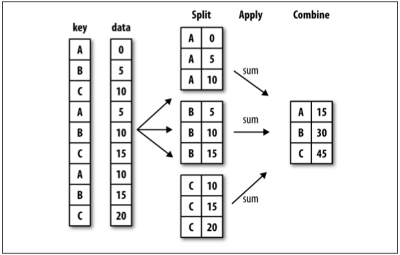

Figure 1: Illustration of SAC

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
- Use the `group_by()` verb to specify variables to use for generating groups

```
flights_by_day <- group_by(flights, day)
```

- Some verbs have specific behavior when applied to grouped data

verb	group specific action
`arrange()`	sort rows within each group
`slice()`	extract rows within each group
`summarize()`	aggregate values group-wise

Grouped slice()

- Retrieve the first 2 departures (rows) of each day

```
slice(flights_by_day, 1:2)
```

```
## Source: local data frame [62 x 19]
## Groups: day [31]
##
##      year month   day dep_time sched_dep_time
##     <int> <int> <int>    <int>          <int>
## 1    2013     1     1      517            515
## 2    2013     1     1      533            529
## 3    2013     1     2       42           2359
## 4    2013     1     2      126           2250
## 5    2013     1     3       32           2359
## 6    2013     1     3       50           2145
## 7    2013     1     4       25           2359
## 8    2013     1     4      106           2245
## 9    2013     1     5       14           2359
```

Grouped summarize()

- Retrieve (1) number of departures (observations), (2) average distance, and (3) average arrival delay for each day (i.e., for flights grouped by day)

```
summarize(flights_by_day,  count = n(),
          dist = mean(distance, na.rm=TRUE),
          delay = mean(arr_delay, na.rm=TRUE))
```

```
##    day count dist   delay
## 1    1 11036 1039  7.3637
## 2    2 10808 1047  6.7681
## 3    3 11211 1041  4.4699
## 4    4 11059 1038 -1.7827
## 5    5 10858 1038  0.4925
## 6    6 11059 1041 -1.7489
```

Multiple (Chained) Operations

- Consider the following task

  find days when the mean arrival delay OR departure delay was greater than 30

- We can achieve the desired result with three operations
  1. **`group_by`** date (`year`, `month`, `day`)
  2. **`summarize`** mean arrival/departure delay
  3. **`filter`** summarized results (i.e., mean `arr_delay` > 30 | mean `dep_delay` > 30)
- Note that `dplyr` verbs do **not** modify the original data frame
  - This is generally a good thing, since it guarantees the integrity of your data
  - But it makes multiple operations on a data frame difficult
- There are two (acceptable) ways to apply multiple operations on a data frame, and one is definitely prefered to the other

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)

```
flights_by_date <- group_by(flights, year, month, day)
summary_by_date <- summarize(
  flights_by_date,
  arr = mean(arr_delay, na.rm=TRUE),
  dep = mean(dep_delay, na.rm=TRUE))
big_delay_dates <- filter(
  summary_by_date,
  arr > 30 | dep > 30)
```

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

The Pipe Operator %>%

- The pipe operator, aka the "magic" operator, takes the output from the verb on its left-hand side, and uses it as the first argument (data frame) for the verb on the right-hand side

```
big_delay_dates <- flights %>%
  group_by(year, month, day) %>%
  summarize(arr = mean(arr_delay, na.rm=TRUE),
            dep = mean(dep_delay, na.rm=TRUE)) %>%
  filter(arr > 30 | dep > 30)
```

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

The Pipe Operator %>% – Best Practice

- Always using the pipe operator – across multiple lines – makes it easier to add/remove/move things around, even for single expressions, i.e.,

```
flights_by_date <- flights %>%
  group_by(year, month, day)
```

`dplyr`: Exercise

- With the `flights` data
  1. find the average speed (`distance / air_time * 60`) by each carrier (ignore `NA`), and sort the data in descending order of average speed
  2. find the number of flights and average flight time of all flights greater than 10 hours by each carrier in April

Exercise Solution

WARNING

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

Solution 1

```
speed_by_carrier <- flights %>%
  group_by(carrier) %>%
  mutate(speed = distance / air_time * 60) %>%
  summarize(avg_speed = mean(speed, na.rm=TRUE)) %>%
  arrange(desc(avg_speed))
speed_by_carrier
```

```
## # A tibble: 6 × 2
##   carrier avg_speed
##     <chr>     <dbl>
## 1      HA     480.4
## 2      VX     446.2
## 3      AS     443.7
## 4      F9     425.2
## 5      UA     420.9
## 6      DL     418.5
```

Solution 2

```
april_long_flights <- flights %>%
  group_by(month, carrier) %>%
  filter(month == 4 & hour > 10) %>%
  summarize(avg = mean(hour, na.rm=TRUE), count = n())
april_long_flights
```

```
## Source: local data frame [6 x 4]
## Groups: month [1]
##
##   month carrier   avg count
##   <int>   <chr> <dbl> <int>
## 1     4      9E 16.68  1170
## 2     4      AA 15.55  1704
## 3     4      AS 18.00    30
## 4     4      B6 16.89  3012
## 5     4      DL 15.81  2747
## 6     4      EV 16.08  3048
```

Reshape Data with `tidyr`

Introduction to `tidyr`

- Recall, the prefered 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
$\vdots$	$\vdots$	$\vdots$
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

```
# Install, if you haven"t already.
# 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 × 5
##       ID   HW1   HW2   HW3             info
##    <chr> <dbl> <dbl> <dbl>            <chr>
## 1  jamie 1.462 8.051 6.853   male/lannister
## 2 cersei 5.798 8.816 9.162 female/lannister
## 3  hodor 5.573 8.110 8.247       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 × 4
##       ID              info    HW score
##    <chr>             <chr> <chr> <dbl>
## 1  jamie   male/lannister   HW1 1.462
## 2 cersei female/lannister   HW1 5.798
## 3  hodor       male/stark   HW1 5.573
## 4  jamie   male/lannister   HW2 8.051
## 5 cersei female/lannister   HW2 8.816
## 6  hodor       male/stark   HW2 8.110
```

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 × 5
##         ID    sex     house    HW score
##      <chr>  <chr>     <chr> <chr> <dbl>
## 1   jamie   male lannister   HW1 1.462
## 2  cersei female lannister   HW1 5.798
## 3   hodor   male     stark   HW1 5.573
```

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

- Create some random 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 × 4
##      ID   type  year  amount
##   <chr>  <chr> <chr>   <dbl>
## 1  leia Income  2013  4174.3
## 2   han Income  2013 11064.9
## 3  luke Income  2013 11341.8
## 4  leia Income  2014 17343.7
## 5   han Income  2014 -5023.7
## 6  luke Income  2014  3547.0
## 7  leia Income  2015  3027.6
## 8   han Income  2015  -757.4
```

Solution 2

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

```
## Source: local data frame [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

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

```
## # A tibble: 3 × 4
##    name       job   score strength
##   <chr>     <chr>   <dbl>    <dbl>
## 1  tony scientist -0.3955   6.0759
## 2  tony     tiger -0.1033       NA
## 3   rey scavenger -0.9967   0.5665
```

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

```
## # A tibble: 3 × 4
##    name       job   score strength
##   <chr>     <chr>   <dbl>    <dbl>
## 1  tony scientist -0.3955   6.0759
## 2  hank scientist      NA   1.0291
```

Inner join

- Retain only the rows that have matches on both datasets

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

```
## # A tibble: 2 × 4
##    name       job   score strength
##   <chr>     <chr>   <dbl>    <dbl>
## 1  tony scientist -0.3955   6.0759
## 2   rey scavenger -0.9967   0.5665
```

Outer (Full) join

- Retain all rows

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

```
## # A tibble: 4 × 4
##    name        job   score strength
##   <chr>      <chr>   <dbl>    <dbl>
## 1  tony scientist -0.3955   6.0759
## 2  tony      tiger -0.1033       NA
## 3   rey scavenger -0.9967   0.5665
## 4  hank scientist      NA   1.0291
```

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 × 4
##    name        job   score strength
##   <chr>      <chr>   <dbl>    <dbl>
## 1  tony scientist -0.3955   6.0759
## 2  tony      tiger -0.1033   0.0000
## 3   rey scavenger -0.9967   0.5665
## 4  hank scientist  0.0000   1.0291
```

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`