Data Wrangling in R Using dplyr

Learning Objectives

1. View specific row(s) and/or column(s) of a data frame
2. Select observations by condition(s)
3. Change column name(s)
4. Find and remove missing data
5. Summarizing variable(s)
6. Summarizing variable by groups
7. Create a new variable based on existing variable(s)
8. Combine data frames and vectors
9. Export data frame in R to a .csv file
10. Sort data frame by column values
11. Pipes

using dplyr functions.

Previously, we were performing data wrangling operations using functions that are built in with base R. In this module, we will be using functions mostly from a package called `dplyr`, which can perform the same operations as well.

The `dplyr` package is a subset of the `tidyverse` package, so we can access these functions after installing and loading either package. After installing the `tidyverse` package, load it by typing

```
##library(dplyr) or
library(tidyverse)
```

We will continue to use the dataset ClassDataPrevious.csv as a working example. Download the dataset from Collab and read it into R

```
Data<-read.csv("ClassDataPrevious.csv", header=TRUE)
```

In the examples below, we are performing the same operations as in Module 1.2, but using `dplyr` functions instead of base R functions.

1. View specific row(s) and/or column(s) of a data frame

The `select()` function is used to select specific columns. There are a couple of ways to use this function. First

```
select(Data,Year)
```

to select the column `Year` from the data frame called `Data`.

Pipes

Alternatively, we can use pipes

```
Data%>%
  select(Year)
```

Pipes in R are typed using `%>%` or by pressing Ctrl + Shift + M on your keyboard. To think of the operations above, we can read the code as

1. take the data frame called Data
2. and then select the column named Year

We can interpret a pipe as "and then". Commands after a pipe should be placed on a new line (press enter). Pipes can be useful if we want to execute several commands in sequence, which we will see in later examples.

2. Select observations by condition(s)

The `filter()` function allows us to subset our data based on some conditions, for example, to select students whose favorite sport is soccer

```
filter(Data, Sport=="Soccer")
```

We can create a new data frame called `SoccerPeeps` that contains students whose favorite sport is soccer

```
SoccerPeeps<-Data%>%
  filter(Sport=="Soccer")
```

Suppose we want to have a data frame, called `SoccerPeeps_2nd`, that satisfies two conditions: that the favorite sport is soccer and they are 2nd years at UVa

```
SoccerPeeps_2nd<-Data%>%
  filter(Sport=="Soccer" & Year=="Second")
```

We can also set conditions based on numeric variables, for example, we want the students who sleep more than eight hours a night

```
Sleepy<-Data%>%
  filter(Sleep>8)
```

We can also create a data frame that contains observations as long as they satisfy at least one out of two conditions: the favorite sport is soccer or they sleep more than 8 hours a night

```
Sleepy_or_Soccer<-Data%>%
  filter(Sport=="Soccer" | Sleep>8)
```

3. Change column name(s)

It is straightforward to change the names of columns using the `rename()` function. For example

```
Data<-Data%>%
  rename(Yr=Year, Comp=Computer)
```

allows us to change the name of two columns: from `Year` and `Computer` to `Yr` and `Comp`.

4. Find and remove missing data

There are a few ways to locate missing data. Using the `is.na()` function directly on a data frame produces a lot of output that can be messy to view.

```
is.na(Data)
```

On the other hand, using the `complete.cases()` function is more pleasing to view

```
Data[!complete.cases(Data),]
```

```
##         Yr Sleep      Sport Courses                                      Major
## 103 Second    NA Basketball       7 psychology and youth and social innovation
## 206 Second     8       None       4                           Cognitive Science
##     Age Comp Lunch
## 103  19  Mac    10
## 206  19  Mac    NA
```

The code above will extract rows that are not complete cases, in other words, rows that have missing entries. The output informs us observation 103 has a missing value for `Sleep`, and observation 206 has a missing value for `Lunch`.

If you want to remove observations with a missing value, you can use the `drop_na()` function:

```
Data_nomiss<-Data %>%
  drop_na()
```

A word of caution: these lines of code will remove the entire row as long as at least a column has missing entries. As noted earlier, observation 103 has a missing value for only the `Sleep` variable. But this observation still provides information on the other variables, which are now removed.

5. Summarizing variable(s)

The `summarize()` function allows us to summarize a column. Suppose we want to find the mean value of the numeric columns: `Sleep`, `Courses`, `Age`, and `Lunch`

```
Data%>%
  summarize(mean(Sleep,na.rm = T),mean(Courses),mean(Age),mean(Lunch,na.rm = T))
```

```
##   mean(Sleep, na.rm = T) mean(Courses) mean(Age) mean(Lunch, na.rm = T)
## 1               155.5593      5.016779  19.57383                156.5942
```

The output looks a bit cumbersome. We can give names to each summary

```
Data%>%
  summarize(avgSleep=mean(Sleep,na.rm = T),avgCourse=mean(Courses),avgAge=mean(Age),
            avgLun=mean(Lunch,na.rm = T))
```

```
##   avgSleep avgCourse   avgAge   avgLun
## 1 155.5593  5.016779 19.57383 156.5942
```

As mentioned previously, the means look suspiciously high for a couple of variables, so looking at the medians may be more informative

```
Data%>%
  summarize(medSleep=median(Sleep,na.rm = T),medCourse=median(Courses),
            medAge=median(Age),medLun=median(Lunch,na.rm = T))
```

```
##   medSleep medCourse medAge medLun
## 1      7.5         5     19      9
```

Note: For a lot of functions in the `dplyr` package, using American spelling or British spelling works. So we can use `summarise()` instead of `summarize()`.

6. Summarizing variable by groups

Suppose we want to find the median amount of sleep 1st years, 2nd years, 3rd years, and 4th years get. We can use the `group_by()` function

```
Data%>%
  group_by(Yr)%>%
  summarize(medSleep=median(Sleep,na.rm=T))
```

```
## # A tibble: 4 x 2
##   Yr     medSleep
##   <chr>     <dbl>
## 1 First       8
## 2 Fourth      7
## 3 Second      7.5
## 4 Third       7
```

The way to read the code above is

1. Get the data frame called `Data`,
2. and then group the observations by `Yr`,
3. and then find the median amount of sleep by each `Yr` and store the median in a vector called `medSleep`.

As seen previously, the ordering of the factor levels is in alphabetical order. For our context, it is better to rearrange the levels to First, Second, Third, Fourth. We can use the `mutate()` function whenever we want to transform or create a new variable. In this case, we are transforming the variable `Yr` by reordering the factor levels with the `fct_relevel()` function

```
Data<- Data%>%
  mutate(Yr = Yr%>%
           fct_relevel(c("First","Second","Third","Fourth")))
```

1. Get the data frame called `Data`,
2. and then transform the variable called `Yr`,
3. and then reorder the factor levels.

Then, we use pipes, the `group_by()`, and `summarize()` functions like before.

```
Data%>%
  group_by(Yr)%>%
  summarize(medSleep=median(Sleep,na.rm=T))
```

```
## # A tibble: 4 x 2
##   Yr     medSleep
##   <fct>     <dbl>
## 1 First       8
## 2 Second      7.5
## 3 Third       7
## 4 Fourth      7
```

This output makes a lot more sense for this context.

To summarize a variable on groups formed by more than one variable, we just add the other variables in the `group_by()` function

```
Data%>%
  group_by(Yr,Comp)%>%
  summarize(medSleep=median(Sleep,na.rm=T))
```

```
## `summarise()` has grouped output by 'Yr'. You can override using the `.groups`
## argument.
```

```
## # A tibble: 9 x 3
## # Groups:   Yr [4]
##   Yr     Comp  medSleep
##   <fct>  <chr>    <dbl>
```

```
## 1 First  "Mac"     8
## 2 First  "PC"      7.5
## 3 Second ""        7
## 4 Second "Mac"     7.5
## 5 Second "PC"      7.5
## 6 Third  "Mac"     7.5
## 7 Third  "PC"      7
## 8 Fourth "Mac"     7
## 9 Fourth "PC"      7.25
```

7. Create a new variable based on existing variable(s)

As mentioned in the previous section, the `mutate()` function is used to transform a variable or to create a new variable. There are a few variations of this task, based on the type of variable you want to create, and the type of variable it is based on.

Create a numeric variable based on another numeric variable

The variable `Sleep` is in number of hours. Suppose we need to convert the values of `Sleep` to number of minutes, we can simply perform the following mathematical operation

```
Data<-Data%>%
  mutate(Sleep_mins = Sleep*60)
```

and store the transformed variable called `Sleep_mins` and add `Sleep_mins` to the data frame called `Data`.

Create a binary variable based on a numeric variable

Suppose we want to create a binary variable , called `deprived`. An observation will obtain a value of `yes` if they sleep for less than 7 hours a night, and `no` otherwise. We will then add this variable deprived to the data frame called `Data`.

```
Data<-Data%>%
  mutate(deprived=ifelse(Sleep<7, "yes", "no"))
```

Create a categorical variable based on a numeric variable

Suppose we want to create a categorical variable based on the number of courses a student takes. We will call this new variable `CourseLoad`, which takes on the following values

- `light` if 3 courses or less,
- `regular` if 4 or 5 courses,
- `heavy` if more than 5 courses

and then add `CourseLoad` to the data frame `Data`.

```
Data<-Data%>%
  mutate(CourseLoad=cut(Courses, breaks = c(-Inf, 3, 5, Inf),
                        labels = c("light", "regular", "heavy")))
```

Collapsing levels

Sometimes, a categorical variable has more levels than we need for our analysis, and we want to collapse some levels. For example, the variable Yr has four levels: First, Second, Third, and Fourth. Perhaps we are more interested in comparing between lower level students and upper level students, so we want to collapse First and Second Yrs into lower level students, and Third and Fourth Yrs into upper level students. We will use the `fct_collapse()` function

```
Data<-Data%>%
  mutate(lowup=fct_collapse(Yr,lower=c("First","Second"),upper=c("Third","Fourth")))
```

We are creating a new variable called `lowup`, which is done by collapsing `First` and `Second` into a new factor called `lower`, and collapsing `Third` and `Fourth` into a new factor called `upper`. `lowup` is also added to the data frame `Data`.

8. Combine data frames

To combine data frames which have different observations but the same columns, we can combine them using `bind_rows()`

```
dat1<-Data[1:3,1:3]
dat3<-Data[6:8,1:3]
res.dat2<-bind_rows(dat1,dat3)
head(res.dat2)
```

```
##       Yr Sleep      Sport
## 1 Second     8 Basketball
## 2 Second     7     Tennis
## 3 Second     8     Soccer
## 4  Third     7       None
## 5 Second     7 Basketball
## 6  First     7 Basketball
```

To combine data frames which have the same observations but different columns, we can combine them using `bind_cols()` or `data.frame()`

```
D1<-Data%>%
  select(Yr)
D2<-Data%>%
  select(Sport)
new.Data<-bind_cols(D1,D2)
head(new.Data)
```

```
##       Yr      Sport
## 1 Second Basketball
## 2 Second     Tennis
## 3 Second     Soccer
## 4  First Basketball
## 5 Second Basketball
## 6  Third       None
```

```
new.Data2<-data.frame(D1,D2)
head(new.Data2)
```

```
##       Yr      Sport
## 1 Second Basketball
## 2 Second     Tennis
## 3 Second     Soccer
## 4  First Basketball
## 5 Second Basketball
## 6  Third       None
```

9. Export data frame in R to a .csv file

We use the `write.csv()` function to export a data frame in R to a .csv file in our working directory

```
write.csv(Data, file="newdata.csv", row.names = FALSE)
```

This takes our dataframe called `Data`, and saves it as `newdata.csv` in our working directory. The argument `row.names=FALSE` prevents an index column from being created.

10. Sort data frame by column values

To sort your data frame in ascending order by `Age`,

```
Data_by_age<-Data%>%
  arrange(Age)
```

To sort in descending order by `Age`,

```
Data_by_age_des<-Data%>%
  arrange(desc(Age))
```

To sort in ascending order by `Age` first, then by `Sleep`,

```
Data_by_age_sleep<-Data%>%
  arrange(Age,Sleep)
```

11. Pipes

We have used pipes throughout all earlier examples.