Introduction to dplyr and the Tidyverse

Introduction to the Tidyverse

The Tidyverse is a collection of R packages designed for data science. Your textbook, R for Data Science, focuses on the Tidyverse. [LINK AND CHECK] These packages work in harmony to clean, process, analyze, and visualize data. The Tidyverse philosophy emphasizes the importance of tidy data, where each variable forms a column, each observation forms a row, and each type of observational unit forms a table.

Key Tidyverse Packages

- **dplyr** for data manipulation. (Our focus today.)
- **ggplot2** for data visualization. (We will learn about this later.)
- **readr** for reading data.(This may be useful.)
- **tidyr** for tidying data.
- **purrr** for functional programming.

Focus on dplyr

`dplyr` is a package that provides a set of tools for efficiently manipulating datasets in R. It's designed to be user-friendly.

Why dplyr?

- Simplified syntax.
- Fast and consistent performance.
- Integrates well with other Tidyverse packages.

Working with Tibbles

Tibbles are a modern take on data frames, but they are tweaked to work better with the Tidyverse. Converting a data frame to a tibble is straightforward:

```
library(tidyverse)
```

```
## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.4     v readr     2.1.4
## v forcats   1.0.0     v stringr   1.5.1
## v ggplot2   3.4.4     v tibble    3.2.1
## v lubridate 1.9.3     v tidyr     1.3.0
## v purrr     1.0.2
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become error
```

```
my_data_frame <- data.frame(x = 1:5, y = letters[1:5], z = c(TRUE,FALSE,TRUE,FALSE,TRUE))
my_data_frame
```

```
##   x y     z
## 1 1 a  TRUE
## 2 2 b FALSE
## 3 3 c  TRUE
## 4 4 d FALSE
## 5 5 e  TRUE
```

```
my_tibble <- as_tibble(my_data_frame)
my_tibble
```

```
## # A tibble: 5 x 3
##       x y     z
##   <int> <chr> <lgl>
## 1     1 a     TRUE
## 2     2 b     FALSE
## 3     3 c     TRUE
## 4     4 d     FALSE
## 5     5 e     TRUE
```

```
glimpse(my_tibble) #(like str())
```

```
## Rows: 5
## Columns: 3
## $ x <int> 1, 2, 3, 4, 5
## $ y <chr> "a", "b", "c", "d", "e"
## $ z <lgl> TRUE, FALSE, TRUE, FALSE, TRUE
```

Basic `dplyr` Functions

The `dplyr` package has a wide range of functions (or "verb") for manipulating tibbles in a user-friendly way. See this online resource[https://dplyr.tidyverse.org/reference/index.html] for a helpful list.

Operations on ROWS

filter() Filters rows based on a condition.

```
filtered_data <- filter(my_tibble, x > 3)
filtered_data
```

```
## # A tibble: 2 x 3
##       x y     z
##   <int> <chr> <lgl>
## 1     4 d     FALSE
## 2     5 e     TRUE
```

arrange()

Reorders rows.

```
arranged_data <- arrange(my_tibble, desc(x))
arranged_data
```

```
## # A tibble: 5 x 3
##       x y     z
##   <int> <chr> <lgl>
## 1     5 e     TRUE
## 2     4 d     FALSE
## 3     3 c     TRUE
## 4     2 b     FALSE
## 5     1 a     TRUE
```

distict()

Keeps unique/distinct rows.

```
# create tibble with duplicates
my_tibble_dup <- as_tibble(data.frame(id=c(1,1,1,3),x=c(3,3,4,5)))
my_tibble_dup
```

```
## # A tibble: 4 x 2
##      id     x
##   <dbl> <dbl>
## 1     1     3
## 2     1     3
## 3     1     4
## 4     3     5
```

```
# remove rows that are duplicate across all columns
distinct_data <- distinct(my_tibble_dup)
distinct_data
```

```
## # A tibble: 3 x 2
##      id     x
##   <dbl> <dbl>
## 1     1     3
## 2     1     4
## 3     3     5
```

```
# remove duplicates of id, keeping first one if different values for x
distinct_id <- distinct(my_tibble_dup, id, .keep_all = TRUE)
distinct_id
```

```
## # A tibble: 2 x 2
##      id     x
##   <dbl> <dbl>
## 1     1     3
## 2     3     5
```

slice()

Extract specific rows.

```
# get rows 1, 2 and 3
sliced_data1 <- slice(my_tibble,c(1,2,3))
sliced_data1
```

```
## # A tibble: 3 x 3
##       x y     z
##   <int> <chr> <lgl>
## 1     1 a     TRUE
## 2     2 b     FALSE
## 3     3 c     TRUE
```

```
# get rows with 2 top maximum values of a variable
sliced_data2 <- slice_max(my_tibble,x,n=2)
sliced_data2
```

```
## # A tibble: 2 x 3
##       x y     z
##   <int> <chr> <lgl>
## 1      5 e     TRUE
## 2      4 d     FALSE
```

Operations on COLUMNS

select() Selects columns from a dataset.

```
selected_data <- select(my_tibble, x, y)
selected_data
```

```
## # A tibble: 5 x 2
##       x y
##   <int> <chr>
## 1     1 a
## 2     2 b
## 3     3 c
## 4     4 d
## 5     5 e
```

mutate() Adds new columns or transforms existing ones.

```
mutated_data <- mutate(my_tibble, w = x * 2)
mutated_data
```

```
## # A tibble: 5 x 4
##       x y     z         w
##   <int> <chr> <lgl> <dbl>
## 1     1 a     TRUE      2
## 2     2 b     FALSE     4
## 3     3 c     TRUE      6
## 4     4 d     FALSE     8
## 5     5 e     TRUE     10
```

combination of multiple functions used togeter

We can carry out numerous operations/manipulations on rows and columns at a time with the following format. We create a new tibble, assign it to the tibble we want to revise followed by %>%. Then we can call multiple dplyr functions/verbs in a slightly different format. Mainly, we no longer pass in the tibble to each function since we have already noted it. And, following each function/verb, we add another %>% - to the end of all except the final function.

```
new_data <- my_tibble %>%
  filter(z) %>%               # Filter rows where z is TRUE
  select(x, y) %>%            # Select columns x and y
  mutate(double_x = x * 2)    # Create a new column 'double_x' which is twice the value of x

new_data
```

```
## # A tibble: 3 x 3
##       x y     double_x
##   <int> <chr>    <dbl>
## 1     1 a            2
## 2     3 c            6
## 3     5 e           10
```

Operations on GROUPS of ROWS

summarize Compute summary statistics across rows (observations) for designated columns.

```
# compute the mean of x across all observations
summarized_x <- summarize(my_tibble,mean_x=mean(x))
summarized_x
```

```
## # A tibble: 1 x 1
##   mean_x
##    <dbl>
## 1      3
```

```
# we can also compute multiple summary statistics at one time:
summary_stats <- my_tibble %>%
  summarize(
    mean_x = mean(x),
    median_x = median(x),
    max_x = max(x)
  )
summary_stats
```

```
## # A tibble: 1 x 3
##   mean_x median_x max_x
##    <dbl>    <int> <int>
## 1      3        3     5
```

group_by Takes an existing tibble and converts it into a grouped tibble where operations are performed "by group."

```
# just grouping data - by whether z is TRUE OR FALSE
grouped_data <- group_by(my_tibble,z)

# if we print our data, it looks the same, but it tells us it is grouped and how
grouped_data
```

```
## # A tibble: 5 x 3
## # Groups:   z [2]
##       x y     z
##   <int> <chr> <lgl>
## 1     1 a     TRUE
## 2     2 b     FALSE
## 3     3 c     TRUE
## 4     4 d     FALSE
## 5     5 e     TRUE
```

```
# now if we run another dplyr function, it will take the grouping into account
grouped_data_means <- summarize(grouped_data,mean_x=mean(x))
grouped_data_means # note that this object is another tibble! abbreviated as tbl
```

```
## # A tibble: 2 x 2
##   z     mean_x
##   <lgl>  <dbl>
## 1 FALSE      3
## 2 TRUE       3
```

```
# we can combine things here...
summary_stats <- my_tibble %>%
    group_by(z) %>%
    summarize(
    mean_x = mean(x),
    median_x = median(x),
    max_x = max(x)
  )
summary_stats
```

```
## # A tibble: 2 x 4
##   z     mean_x median_x max_x
##   <lgl>  <dbl>    <dbl> <int>
## 1 FALSE      3        3     4
## 2 TRUE       3        3     5
```

Here is another example of using group_by with another row operation - arrange. In this case, the sorting will happen within groups defined by whether z is TRUE or FALSE. Note it does not sort by groups in the results. But this could be accomplished with another call to arrange().

```
# adding a variable for illustration
my_tibble_addw <- mutate(my_tibble, w = c(3,2,-4,0,1))
my_tibble_addw
```

```
## # A tibble: 5 x 4
##       x y     z         w
##   <int> <chr> <lgl> <dbl>
## 1     1 a     TRUE      3
## 2     2 b     FALSE     2
## 3     3 c     TRUE     -4
## 4     4 d     FALSE     0
## 5     5 e     TRUE      1
```

```
# arranging tbl by descending order of w, by group defined by z
arranged_grouped_data <- my_tibble_addw %>%
    group_by(z) %>%
    arrange(desc(w))
arranged_grouped_data
```

```
## # A tibble: 5 x 4
## # Groups:   z [2]
##       x y     z         w
##   <int> <chr> <lgl> <dbl>
## 1     1 a     TRUE      3
## 2     2 b     FALSE     2
## 3     5 e     TRUE      1
## 4     4 d     FALSE     0
## 5     3 c     TRUE     -4
```

Operations on GROUPS of COLUMNS

across Apply functions across multiple columns. It's particularly useful in conjunction with mutate() and summarize() functions. across() allows you to perform the same operation on multiple columns without having to write repetitive code for each column.

```
# Applying transformations
transformed_tibble <- my_tibble %>%
  mutate(
    # Convert logical columns to numeric (TRUE to 1, FALSE to 0)
    across(where(is.logical), as.numeric),
    # Append "_modified" to the character values in column 'y'
    across(where(is.character), ~ paste0(.x, "_modified"))
  )

transformed_tibble
```

```
## # A tibble: 5 x 3
##       x y              z
##   <int> <chr>      <dbl>
## 1     1 a_modified     1
## 2     2 b_modified     0
## 3     3 c_modified     1
## 4     4 d_modified     0
## 5     5 e_modified     1
```

Note when you're using across() within mutate() or summarize(), `.x` represents the data from the current column being processed. It's a way to refer to each element of the column(s) you're applying the function to.

Summary

The dplyr functionality is powerful for wrangling data. With practice, you will learn to harness and layer the wide variety of functions and verbs in order to conduct numerous operations. My advice is that you start slow and gradually add on more complex combinations.

Also, this notebook has covered just a subset of the *types* of functions/verbs - those that operate on single rows and columns or groups of rows and columns. As you will see in the documentation, there are other functions/verbs that operate on whole data frames or that operate at vector level. We will touch on these later when we discuss data merging (or joining), which refers to combining two datasets with overlapping observations.

Finally, here is a good reference[https://dplyr.tidyverse.org/articles/base.html] for comparing dplyr functions to their base R equivalents.