`R` Handout: Introduction to `tidyverse`

Contents

Overview 1

Key Functions in `tidyverse` 1
- Installing `tidyverse` . . . 1

Functions to Subset and Organize Data Sets 2
- Select Columns or Variables using `select()` . . . 2
- Select Rows or Observations using `filter()` . . . 3
- Sorting Data using `arrange()` . . . 4

Functions to Summarize Data and Create New Variables 4
- Summarizing Data with `summarize()` . . . 4
- Creating New Variables using `mutate()` . . . 5

Piping with `%>%` 7
- Grouping Data with `group_by()` . . . 8

Overview

In the *Learning to Speak `R`* handout, we defined *packages*. Packages are collections of functions that we can use in `R` but which don't come installed by default with "Base `R`". One of the most important packages that we'll use is called `tidyverse`. This package provides a range of functions to help with data cleaning and analysis tasks.

Key Functions in `tidyverse`

This handout introduces the following functions:

- `select()` - pick out specific columns or variables from your data
- `filter()` - pick out specific rows or observations from your data
- `arrange()` - reorder or sort rows of your data
- `mutate()` - create new variables as functions of existing variables
- `summarize()` - creates summary statistics (with `group_by()`)

At the end of this handout, we'll discuss a final important concept - the pipe operator `%>%`. We'll see how this operator can be used to simplify data cleaning and make code easier to read.

Installing `tidyverse`

If you haven't installed the `tidyverse` on your computer yet, you can do so running the following command: `install.packages("tidyverse")`. You only need to run this command once on your computer (if you switch computers, you'll want to run it again).

Once you've installed this package, you can access it by using including the following command in your `R` script: `library(tidyverse)`. This tells `R`, "I want to use the functions included in this package." You'll want to include this line at the start of all your `R` scripts. Without running this command, you'll get error messages if you try to use `tidyverse` functions.

NOTE: When you run the `library` function above, `R` will return a series of messages. As long as the `tidyverse` package loads successfully (and you see something like "Attaching code tidyverse packages"), you're all set. If you're getting errors when trying to use `tidyverse` commands, we can revisit these messages and check that things loaded correctly.

Functions to Subset and Organize Data Sets

We can use `tidyverse` to perform basic data "manipulation" tasks like selecting certain columns (variables) or rows (observations). Let's see how these functions work by creating a sample data set:

```
# Set of vectors that we'll combine to create our sample data set

person.ID <- c(12, 24, 54, 65)
address   <- c("123 Main St", "274 Long St", "789 Right St", "467 Left St")
employed  <- c(TRUE, TRUE, FALSE, TRUE)
wage.inc  <- c(12500, 15750, 0, 14100)

sample.data <- data.frame(person.ID, address, employed, wage.inc)

sample.data
```

```
##   person.ID      address employed wage.inc
## 1        12  123 Main St     TRUE    12500
## 2        24  274 Long St     TRUE    15750
## 3        54 789 Right St    FALSE        0
## 4        65  467 Left St     TRUE    14100
```

Select Columns or Variables using `select()`

Suppose we want to create a new data frame that's just stores the `person.ID` and `address` variables from the `sample.data` data frame created above. We can use the `select()` function to do this.

SYNTAX: `select(name of data frame, name of variable to keep)`

Things to note:

- You can select multiple variables by separating each name with a comma.
- You *don't* need to put quotes around the variable names.

Let's use this function on our sample data set in the code below:

```
# Select person.ID and address columns from sample.data, and store it as
# a new data frame named ID.address.data - we'll then print this new data frame

ID.address.data <- select(sample.data, person.ID, address)

ID.address.data
```

```
##   person.ID      address
## 1        12  123 Main St
## 2        24  274 Long St
## 3        54 789 Right St
## 4        65  467 Left St
```

Select Rows or Observations using `filter()`

We saw how to select certain columns of our data above. Now, let's see how to select certain *rows* of our data. When we pick out specific rows of data, we're selecting particular *observations* within our data. To do this, we'll want to apply some set of criteria to decide which observations to keep and which to remove. We can do this using the `filter()` function.

SYNTAX: `filter(name of data frame, criteria for keeping observations)`

Things to note:

- In the code below, we'll see several commonly used filter criteria.
- When you want to match an exact criteria, use: `variable name == some condition`. You can read `==` here as "equal to". We'll see other examples below as well using `<` as well as combinations of criteria using `&` and `|`.
- As you work on more complex data analysis tasks, you might need to use other criteria - take a look at the `filter()` documentation on the `tidyverse` website for more examples!

Let's use this function on our sample data set in the code below. Here, instead of creating and saving new data set, we'll just call the `filter()` function directly, and let it print the resulting data frame without saving a new data frame.

```
# Keep observation with address equal to "123 Main St"

filter(sample.data, address == "123 Main St")
```

```
##   person.ID     address employed wage.inc
## 1        12 123 Main St     TRUE    12500
```

```
# Keep observations with wages greater than 14,000

filter(sample.data, wage.inc > 14000)
```

```
##   person.ID     address employed wage.inc
## 1        24 274 Long St     TRUE    15750
## 2        65 467 Left St     TRUE    14100
```

```
# Keep observations using two criteria using the & symbol - this syntax
# requires that BOTH conditions be true - you can read "&" as "AND"

filter(sample.data, employed == TRUE & person.ID < 50)
```

```
##   person.ID     address employed wage.inc
## 1        12 123 Main St     TRUE    12500
## 2        24 274 Long St     TRUE    15750
```

```
# We can use two conditions like above, but return rows that match EITHER
# condition using the | symbol which we can read as "OR"

filter(sample.data, employed == TRUE | person.ID < 50)
```

```
##   person.ID     address employed wage.inc
## 1        12 123 Main St     TRUE    12500
## 2        24 274 Long St     TRUE    15750
## 3        65 467 Left St     TRUE    14100
```

Sorting Data using `arrange()`

Suppose we want to sort our data by a particular column to check how we've coded things or create a table of output. We can use the `arrange()` function to do this.

SYNTAX: `arrange(name of data frame, name of variable to sort by)`

Things to note:

- The default sort is from smallest to largest values.
- You can reverse this by sort from largest to smallest by wrapping the variable name with the `desc()` function.
- You can sort by multiple variables by separating each name with a comma.

Let's use this function on our sample data set in the code below:

```
# Sort sample.data observations by wage.inc from smallest to largest

arrange(sample.data, wage.inc)
```

```
##   person.ID      address employed wage.inc
## 1        54 789 Right St    FALSE        0
## 2        12  123 Main St     TRUE    12500
## 3        65  467 Left St     TRUE    14100
## 4        24  274 Long St     TRUE    15750
```

```
# Sort by wage.inc in descending order from largest to smallest using desc()

arrange(sample.data, desc(wage.inc))
```

```
##   person.ID      address employed wage.inc
## 1        24  274 Long St     TRUE    15750
## 2        65  467 Left St     TRUE    14100
## 3        12  123 Main St     TRUE    12500
## 4        54 789 Right St    FALSE        0
```

```
# Sort first by employed (here, FALSE = 0, TRUE = 1), then by wage.inc

arrange(sample.data, employed, wage.inc)
```

```
##   person.ID      address employed wage.inc
## 1        54 789 Right St    FALSE        0
## 2        12  123 Main St     TRUE    12500
## 3        65  467 Left St     TRUE    14100
## 4        24  274 Long St     TRUE    15750
```

Functions to Summarize Data and Create New Variables

We've seen how to subset and sort data sets. Now, let's learn how to *transform* our data. In this section, we'll learn how to calculate summary statistics and how to create new variables.

Summarizing Data with `summarize()`

For simple summary statistics for a individual vector or column of data, its often easy enough to use the functions from the "Learning to Speak `R`" handout. However, `tidyverse` provides useful tools to make calculating summary statistics easy - particularly when we want to group our data beforehand (more on that below) or we want to calculate multiple summary statistics at one time. The `summarize()` function is the basic `tidyverse` function we'll use to perform these tasks.

SYNTAX: `summarize(name of data frame, name of summary stat = function(a variable in your data set)`.

Things to note:

- There's several important features of syntax above to note:
    1) This function returns a *new* data frame with your summary stat - "name of summary stat" specifies the column name for this new statistic
    2) You'll then specify a particular function corresponding to the particular summary statistic you want to calculate (listed below)
- Here's a list of commonly-used functions for calculating summary statistics, categorized by the information they convey:
    - Measures of central tendency: `mean()` and `median()`
    - Percentiles: `quantile(variable name, value between 0 and 1)`
    - Spread or distribution: `sd()` and `IQR()`
    - Range: `min()` and `max()`
    - Counts and unique values: `n()` and `n_distinct()`

The structure of the `summarize()` function might seem overly complicated when you first see it. Where this function really shines is when you want to calculate several summary stats or group your data. Let's take a look at several examples in the code below:

```
# Start with a simple example - calculating average wages. Remember that
# summarize will return a data frame. In this case, we'll have one column,
# named mean.wage.inc, with a single row storing the mean value of wage.inc.

summarize(sample.data, mean.wage.inc = mean(wage.inc))
```

```
##   mean.wage.inc
## 1       10587.5
```

```
# Suppose we want to calculate the median, 10th, and 90th percentiles of
# wage.inc. We can do this by adding additional arguments to our summarize
# function. Take a look at the resulting data frame - now we have three
# columns, with each corresponding to one of our summary statistics.

summarize(sample.data,
          median.wage.inc = median(wage.inc),
          pct.10.wage.inc = quantile(wage.inc, 0.1),
          pct.90.wage.inc = quantile(wage.inc, 0.9))
```

```
##   median.wage.inc pct.10.wage.inc pct.90.wage.inc
## 1           13300            3750           15255
```

Creating New Variables using `mutate()`

A common data cleaning task is to create new columns or variables in our data. We can use these new columns to store statistics that we calculate using other variables in our data set, as well as identify rows or observations in our data that fall into certain categories. We can use the `mutate()` function to perform these kinds of tasks.

SYNTAX: `mutate(name of data frame, name of new variable = function(existing variable in your data set)`.

Things to note:

- The syntax here should similar look to the `summarize()` syntax from above:
    1) This function adds a *new* column to your data frame - the "name of new variable" above is the column name for that new variable

2) You can then specify a particular function defining to the new variable you want to calculate (see examples below)

- Here's an abbreviated list of functions you can use with mutate to create new variables:
  - Basic operations like `+`, `-`, and `log()`
  - Time-series functions: `lead()` and `lag()`
  - Logical operations: `if_else()`, `recode()`, and `case_when()`
- Creating logical variables using `ifelse()` and `casewhen()` to flag certain observations can be especially helpful - see the examples below.
- For other functions, see the `tidyverse` website and documentation

Let's use this function on our sample data set in the code below. In each case, we'll call `mutate` to see what the output looks like. `R` will print new versions of our `sample.data` with the variables added, but it won't save over our old `sample.data` object unless we explicitly tell it to do so (we'll skip that step for now, more on this below in the piping section).

```
# To see the similarities between summarize and mutate, let's start by creating
# a new variable equal to average wages - with mutate, this will be stored as
# a new column in our data (with the same mean value repeated in each row)

mutate(sample.data, avg.wage.inc = mean(wage.inc))
```

```
##   person.ID      address employed wage.inc avg.wage.inc
## 1        12  123 Main St     TRUE    12500      10587.5
## 2        24  274 Long St     TRUE    15750      10587.5
## 3        54 789 Right St    FALSE        0      10587.5
## 4        65  467 Left St     TRUE    14100      10587.5
```

```
# We can also create a new variable using a function defined over multiple
# existing variables. The example below is just an illustration to show what
# mutate can do (the formula here doesn't make much economic sense!).

mutate(sample.data, new.var = log(person.ID)^2 + wage.inc/50)
```

```
##   person.ID      address employed wage.inc   new.var
## 1        12  123 Main St     TRUE    12500 256.17476
## 2        24  274 Long St     TRUE    15750 325.10003
## 3        54 789 Right St    FALSE        0  15.91199
## 4        65  467 Left St     TRUE    14100 299.42551
```

```
# Using the ifelse() function, we can create a new variable to identify
# observations that meet certain criteria defined by a single other variable.
# As an example, let's flag observations with wage.inc above 14,000. Our new
# variable will equal 1 when the condition holds (incomes > 14000) and 0
# when it doesn't hold (incomes < 14000).

mutate(sample.data, inc.above.14000 = ifelse(wage.inc > 14000, 1, 0))
```

```
##   person.ID      address employed wage.inc inc.above.14000
## 1        12  123 Main St     TRUE    12500               0
## 2        24  274 Long St     TRUE    15750               1
## 3        54 789 Right St    FALSE        0               0
## 4        65  467 Left St     TRUE    14100               1
```

```
# When we want to flag observations that meet criteria defined over multiple
# variables, or apply a range of restrictions on the same variable, its easier
# to use the case_when() function.
```

```
mutate(sample.data,
       id.wage.indicator = case_when(person.ID > 25 & wage.inc > 0  ~ 1,
                                     person.ID > 25 & wage.inc == 0 ~ 2,
                                     TRUE ~ 3))
```

```
##   person.ID      address employed wage.inc id.wage.indicator
## 1        12  123 Main St     TRUE    12500                 3
## 2        24  274 Long St     TRUE    15750                 3
## 3        54 789 Right St    FALSE        0                 2
## 4        65  467 Left St     TRUE    14100                 1
```

```
# In the example above, we have two sets of logical conditions that define
# three categories into which observations can be grouped (the third category
# covers observations that don't fall into either of the first two categories).
# These categories are defined over person.ID and wage.inc.

# The tilde denotes the value will be assigned to observations that meet the
# corresponding criteria. In the final line, "TRUE ~ 3" says that for any
# observations that don't meet the first two sets of conditions, just assign a
# value of 3. Check the tidyverse website + documentation for more examples!
```

Piping with %>%

Data cleaning and processing like the examples we've seen above often entails lots of operations on a single data frame. This can make code messy and hard to read! The `tidyverse` package includes what's known as the pipe operator, indicated by the key combination `%>%`, to make code easier to read (you can generate this combo of symbols by hitting `ctrl + shift + M` on your keyboard).

In this section, we'll take a look at several examples of how `%>%` let's us write cleaner, easier to read code. Here's several things to note at the start:

- Whenever you see the `%>%` symbols, its telling you "take what's on the left side of the symbol and send it to the right side (generally, onto the next line down)" – whatever is getting sent along will be used as an argument to the function that's getting used on the right side (next line down).
- Oftentimes, you'll see several sequential uses of `%>%` - you can read this as, take an object at the top of the code like a data frame or variable in a data frame, apply X intermediate functions to that object, then send that transformed object to the last function in that sequence.

The examples below should make all this clearer!

```
# A simple example to start - in the `mutate` section above, I mentioned that we
# weren't saving new versions of sample.data when we created new variables.
# Let's see how mutate can make this process cleaner by looking at two ways of
# saving a new variable. Start by considering the following...

sample.data <- mutate(sample.data, avg.wage.inc = mean(wage.inc))

# The code below does the same but uses %>% to make things clearer

sample.data <- sample.data %>%
  mutate(avg.wage.inc = mean(wage.inc))

# What's the advantage of the second option? It makes it clear that we're
# starting with sample.data, passing that data frame as an argument to
# mutate (notice that now sample.data doesn't appear directly in the mutate
# function call), then saving the output to sample.data. This is especially
# helpful when we're performing multiple tasks with the same data frame.

# Let's consider another example:

summary.stats.table <- sample.data %>%
  mutate(new.var = log(person.ID)^2 + wage.inc/50) %>%
  summarize(avg.new.var = mean(new.var))

# In the example above, we're starting with our sample.data data frame, then
# passing it as an argument to the mutate function in the next line and
# creating a new variable, then passing that updated data frame to summarize
# in the final line to calculate the average value of that new variable.
```

Grouping Data with `group_by()`

We've seen how to use the `summarize()` function to calculate summary statistics for a given column of our data across *all* observations. But what if we wanted to calculate summary statistics by *groups* of observations? For example, in our sample data set, we might be interested in average wages for people who (1) are employed and (2) are not employed. We can do this using the `group_by` function. This is one big advantage of using `%>%` – it streamlines combining `group_by` and `summarize()` to calculate averages across groups.

```
# Calculating average wages grouped by employed status using group_by()

group.avg.wages <- sample.data %>%
  group_by(employed) %>%
  summarize(mean.wages = mean(wage.inc))
```

Let's unpack what's going on in the code above! Reading from the top, we're telling `R` to do the following:

1) Start with our `sample.data` data frame and pass that as a argument to the `group_by` function on the second line
2) On the second line, take `sample.data`, and group every row in the data frame by `employed` (so that we'll have observations are grouped by whether their value of `employed` is `TRUE` or `FALSE`)
3) Send the grouped data from the second line to the third line, where it becomes an argument to the `summarize` function
4) In this final line, we're telling `R` to calculate average wages. Because we've grouped our data, `R` now calculates *two* averages - wages for those who are employed, and wages for those who aren't.
5) Finally, we save the resulting group average values data frame as the `group.avg.wages` object named at the top of the example.

Let's look at one more example:

```
# Create a new variable using mutate

sample.data <- sample.data %>%
  group_by(employed) %>%
  mutate(avg.wage = mean(wage.inc))
```

Just like in the previous example, we're grouping `sample.data` by `employed`. Here, however, we want to use `mutate` to add a new variable to our data set.

In the first line, we're telling `R`, "I want to use `sample.data` as an input to `group_by` and `mutate`, but at the end I want to save the output as the new / updated version of `sample.data`." In the last line, we calculate average wages by group, and save the resulting means for employed and unemployed observations in the new variable `avg.wage`.