R's `tidyverse`Statistics 506

The Tidyverse

The "Tidyverse" is a series of R packages developed primarily by Hadley Wickham and his team at Posit (formerly RStudio). In its own words, it is an "opinionated collection of R packages designed for data science".

Proponents of the tidyverse (so-named because one of the original packages was **tidyr**) argue that it provides a consistent "grammar" of statistics that is easier for new users to understand. Whether this is true or not remains to be seen.

The primary package in the tidyverse is **dplyr** which we will be going over. Additionally the **tibble** package introduces the tibble, which is an extension of a `data.frame`. There are a number of other packages which are more niche:

- **tidyr**: Reshaping data (wide to long)
- **readr**: Reading in CSV data
- **purrr**: Functional programming
- **stringr**: String manipulation
- **forcats**: `factor` manipulation

Finally, the **ggplot2** predates anything about the tidyverse, but none-the-less is now considered part of the tidyverse. We will be covering **ggplot2** in a separate set of notes.

In addition to these formal tidyverse packages, you will find many packages written by other authors which interact with the tidyverse. These typically aren't as "opinionated" and can be used with or without the rest of the tidyverse. For example,

- **haven**: Reading and writing data from Stata, SAS and SPSS
- **lubridate**: Working with datetime variables
- **rvest**: Web-scraping

```
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.3     v readr     2.1.4
v forcats   1.0.0     v stringr   1.5.0
v ggplot2   3.4.3     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 becom
```

When loading the meta-library **tidyverse**, the main above packages also get loaded, as seen in that note.

Piping

The tidyverse is heavily invested in the idea of "piping". The "pipe" operator is formally defined in the **magittr** package.

```r
x <- rnorm(10)
mean(x)
```

```
[1] -0.007047451
```

```r
x %>% mean
```

```
[1] -0.007047451
```

```r
x %>% mean()
```

```
[1] -0.007047451
```

The left side of the pipe gets included as the *first* argument of the right side function. Additional arguments can be passed as needed.

```r
x[1] <- NA
x %>% mean(na.rm = TRUE)
```

```
[1] -0.01171557
```

The object can be passed into different slots with the `.`:

```
data(mtcars)
lm(mpg ~ wt, data = mtcars)
```

```

Call:
lm(formula = mpg ~ wt, data = mtcars)

Coefficients:
(Intercept)           wt
     37.285       -5.344
```

```
mtcars %>% lm(mpg ~ wt)
```

```
Error in as.data.frame.default(data): cannot coerce class '"formula"' to a data.frame
```

```
mtcars %>% lm(mpg ~ wt, data = .)
```

```

Call:
lm(formula = mpg ~ wt, data = .)

Coefficients:
(Intercept)           wt
     37.285       -5.344
```

Note that as of R 4.1, R has it's own base pipe, `|>`:

```
x |> mean(na.rm = TRUE)
```

```
[1] -0.01171557
```

```
mtcars |> lm(mpg ~ wt, data = _)
```

```
Call:
lm(formula = mpg ~ wt, data = mtcars)

Coefficients:
(Intercept)           wt
     37.285       -5.344
```

There are a **lot** of differences between `%>%` and `|>`, which this stackoverflow answer goes into great detail about, but in most situations, they will function identically.

Of note is that `|>` is *substantially* faster, primarily because it does simple substitution: `x |> mean()` simply processes `mean(x)` without any additional processing. `%>%` does a lot of additional processing, which does enable some other features, but those features are not commonly used.

Should you use pipes?

There is nothing pipes can do that cannot be accomplished without their use. The choice between using pipes is (speed-considerations of `%>%` vs `|>` aside) entirely a personal code style choice.

dplyr

We will be using the 2009 RECS data to demonstrate the functionality of **dplyr**. We'll approach this as a case study in which we set out to answer the question:

> Which state has the highest proportion of single-family attached homes?

There are five main functions that **dplyr** uses. There are, of course, many more, but these are the most common ones.

- `select()` picks variables based on their names.
- `filter()` picks cases based on their values.
- `arrange()` changes the ordering of the rows.
- `mutate()` adds new variables that are functions of existing variables
- `summarize()` reduces multiple values down to a single summary.

Data cleaning

Let's begin by creating a clean and *tidy* data frame with the necessary variables. We'll need to keep the two variables of interest and the sample weight. Later we will also make use of the replicate weights to compute standard errors.

Here we read in the data, either from a local file or directly from the web.

```
recs_tib <- readr::read_delim("data/recs2009_public.csv")
```

Note the use of **readr** rather than `read.csv` to stick within the tidyverse. `recs_tib` is now a `tibble`. We will go into more detail later about tibbles, for now they are mostly just `data.frame`s.

Next, we'll use `select()` to drop all but a subset of variables. We'll need to keep "REPORTABLE_DOMAIN" which records the State, "TYPEHUQ" which records the type of houses, and "NWEIGHT" which records the weight for the record which we'll need to use later. (Sampling weights is a massive topic outside of scope for this class; for now just understand that by using these weights in our analysis [e.g. weighted means or weighted least squares], we can obtain estimates which are appropriate for the entire US population.)

```
recs_homes <- recs_tib %>%
  select(REPORTABLE_DOMAIN,
         TYPEHUQ,
         NWEIGHT) %>%
  rename(state = REPORTABLE_DOMAIN,
         type = TYPEHUQ,
         weight = NWEIGHT)
recs_homes
```

```
# A tibble: 12,083 x 3
   state  type weight
   <dbl> <dbl>  <dbl>
 1    12     2  2472.
 2    26     2  8599.
 3     1     5  8970.
 4     7     2 18004.
 5     1     3  6000.
 6    10     2  4232.
 7     3     2  7862.
 8    17     2  6297.
 9     5     3 12157.
10    12     2  3242.
```

```
# i 12,073 more rows
```

Stylistically, note the convention of ending each line on the pipe.

Next, we clean up the values to something more easily interpreted. The values used here come from the code book available here.

```
recs_homes <- recs_homes %>%
  mutate(state = sapply(state, function(x) {
    switch(x,
           "CT, ME, NH, RI, VT", "MA", "NY", "NJ", "PA",
           "IL", "IN, OH", "MI", "WI", "IA, MN, ND, SD",
           "KS, NE", "MO", "VA", "DE, DC, MD, WV", "GA",
           "NC, SC" , "FL", "AL, KY, MS", "TN",
           "AR, LA, OK", "TX", "CO", "ID, MT, UT, WY", "AZ",
           "NV, NM", "CA", "AK, HI, OR, WA")
  }), type = sapply(type, function(x) {
    switch(x,
           "MobileHome",
           "SingleFamilyDetached",
           "SingleFamilyAttached",
           "ApartmentFew",
           "ApartmentMany")
  }))
recs_homes
```

```
# A tibble: 12,083 x 3
   state              type                 weight
   <chr>              <chr>                 <dbl>
 1 MO                 SingleFamilyDetached  2472.
 2 CA                 SingleFamilyDetached  8599.
 3 CT, ME, NH, RI, VT ApartmentMany         8970.
 4 IN, OH             SingleFamilyDetached 18004.
 5 CT, ME, NH, RI, VT SingleFamilyAttached  6000.
 6 IA, MN, ND, SD     SingleFamilyDetached  4232.
 7 NY                 SingleFamilyDetached  7862.
 8 FL                 SingleFamilyDetached  6297.
 9 PA                 SingleFamilyAttached 12157.
10 MO                 SingleFamilyDetached  3242.
# i 12,073 more rows
```

It probably would have been cleaner to write those functions externally. They certainly would be easier to test.

Aggregating by group

Recall that we are interested in computing the proportion of each housing type by state. We can do this using a split-apply-combine paradigm. We *split* the data by a grouping variable, *apply* a function to each split of the data, then *combine* the results back into a single dataset.

In **dplyr** the `group_by` function handles the *split* step, typically `summarize` handles the *apply* step, and `ungroup` (optionally) handles the *combine* step.

```
recs_homes_group_states <- recs_homes %>%
  group_by(state, type)
recs_homes_group_states
```

```
# A tibble: 12,083 x 3
# Groups:   state, type [134]
   state              type                 weight
   <chr>              <chr>                 <dbl>
 1 MO                 SingleFamilyDetached  2472.
 2 CA                 SingleFamilyDetached  8599.
 3 CT, ME, NH, RI, VT ApartmentMany         8970.
 4 IN, OH             SingleFamilyDetached 18004.
 5 CT, ME, NH, RI, VT SingleFamilyAttached  6000.
 6 IA, MN, ND, SD     SingleFamilyDetached  4232.
 7 NY                 SingleFamilyDetached  7862.
 8 FL                 SingleFamilyDetached  6297.
 9 PA                 SingleFamilyAttached 12157.
10 MO                 SingleFamilyDetached  3242.
# i 12,073 more rows
```

Note the tibble keeping track of the grouping. Next, the aggregation:

```
recs_type_state_sum <- recs_homes_group_states %>%
  summarize(homes = sum(weight))
```

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

```
recs_type_state_sum
```

```
# A tibble: 134 x 3
# Groups:   state [27]
   state          type                    homes
   <chr>          <chr>                   <dbl>
 1 AK, HI, OR, WA ApartmentFew          374743.
 2 AK, HI, OR, WA ApartmentMany         946196.
 3 AK, HI, OR, WA MobileHome            384298.
 4 AK, HI, OR, WA SingleFamilyAttached  189645.
 5 AK, HI, OR, WA SingleFamilyDetached 2833057.
 6 AL, KY, MS     ApartmentFew          183983.
 7 AL, KY, MS     ApartmentMany         201344.
 8 AL, KY, MS     MobileHome            422086.
 9 AL, KY, MS     SingleFamilyAttached  192720.
10 AL, KY, MS     SingleFamilyDetached 3637141.
# i 124 more rows
```

Pay close attention to the change in grouping. When `summarize()` is called we lose the most nested group.

Finally we can optionally `ungroup`. The reason it is optional is that a lot of functions are not aware of the grouping, so it rarely is wrong to simply leave it grouped. However, there are issues that can occur when leaving something grouped, so for safety I recommend always `ungroup`ing.

```r
recs_types_state_sum <- recs_type_state_sum %>%
  ungroup()
recs_types_state_sum
```

```
# A tibble: 134 x 3
   state          type                    homes
   <chr>          <chr>                   <dbl>
 1 AK, HI, OR, WA ApartmentFew          374743.
 2 AK, HI, OR, WA ApartmentMany         946196.
 3 AK, HI, OR, WA MobileHome            384298.
 4 AK, HI, OR, WA SingleFamilyAttached  189645.
 5 AK, HI, OR, WA SingleFamilyDetached 2833057.
 6 AL, KY, MS     ApartmentFew          183983.
 7 AL, KY, MS     ApartmentMany         201344.
 8 AL, KY, MS     MobileHome            422086.
 9 AL, KY, MS     SingleFamilyAttached  192720.
10 AL, KY, MS     SingleFamilyDetached 3637141.
# i 124 more rows
```

Note that we could have done this in one step:

```
recs_types_state_sum <- recs_homes %>%
  group_by(state, type) %>%
  summarize(homes = sum(weight)) %>%
  ungroup()
```

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

```
recs_types_state_sum
```

```
# A tibble: 134 x 3
   state          type                   homes
   <chr>          <chr>                  <dbl>
 1 AK, HI, OR, WA ApartmentFew         374743.
 2 AK, HI, OR, WA ApartmentMany        946196.
 3 AK, HI, OR, WA MobileHome           384298.
 4 AK, HI, OR, WA SingleFamilyAttached  189645.
 5 AK, HI, OR, WA SingleFamilyDetached 2833057.
 6 AL, KY, MS     ApartmentFew         183983.
 7 AL, KY, MS     ApartmentMany        201344.
 8 AL, KY, MS     MobileHome           422086.
 9 AL, KY, MS     SingleFamilyAttached  192720.
10 AL, KY, MS     SingleFamilyDetached 3637141.
# i 124 more rows
```

Reshaping and formatting results for presentation

To proceed, let's reshape the data to have one row per state. We can do this using the `tidyr::pivot_wider()` function. The **tidyr** package is designed for

```
recs_type_state <- recs_type_state_sum %>%
  tidyr::pivot_wider(names_from = type,
                     values_from = homes)
recs_type_state
```

```
# A tibble: 27 x 6
# Groups:   state [27]
```

```
   state              ApartmentFew ApartmentMany MobileHome SingleFamilyAttached
   <chr>                     <dbl>         <dbl>      <dbl>                <dbl>
 1 AK, HI, OR, WA          374743.       946196.    384298.              189645.
 2 AL, KY, MS              183983.       201344.    422086.              192720.
 3 AR, LA, OK              322290.       605024.    239154.              214708.
 4 AZ                       24143.       380745.    336741.               77391.
 5 CA                     1034231.      2871668.    394079.              856699.
 6 CO                      147208.       260461.     97400.              203527.
 7 CT, ME, NH, RI, VT      422981.       501581.     45209.              144269.
 8 DE, DC, MD, WV          109699.       634137.    253861.              590254.
 9 FL                      414436.      1143320.    974800.              261688.
10 GA                      124408.       463603.    127089.              101213.
# i 17 more rows
# i 1 more variable: SingleFamilyDetached <dbl>
```

Next, compute all proportions

```
recs_type_state <- recs_type_state %>%
  mutate(Total = sum(ApartmentFew, ApartmentMany, MobileHome,
                     SingleFamilyAttached, SingleFamilyDetached,
                     na.rm = TRUE),
         ApartmentFew = 100 * ApartmentFew / Total,
         ApartmentMany = 100 * ApartmentMany / Total,
         MobileHome = 100 * MobileHome / Total,
         SingleFamilyAttached = 100 * SingleFamilyAttached / Total,
         SingleFamilyDetached = 100 * SingleFamilyDetached / Total) %>%
  select(-Total) %>% # Drop total
  arrange(SingleFamilyAttached)
recs_type_state
```

```
# A tibble: 27 x 6
# Groups:   state [27]
   state          ApartmentFew ApartmentMany MobileHome SingleFamilyAttached
   <chr>                 <dbl>         <dbl>      <dbl>                <dbl>
 1 TN                     4.52          18.4       9.66                 1.91
 2 MI                     5.26          15.3       7.27                 2.78
 3 GA                     3.59          13.4       3.66                 2.92
 4 IL                    11.5           19.9      NA                    3.03
 5 NC, SC                 6.40          15.3      13.6                  3.24
 6 AZ                     1.06          16.7      14.8                  3.40
 7 FL                     5.93          16.4      14.0                  3.75
 8 AK, HI, OR, WA         7.93          20.0       8.13                 4.01
```

```
 9 TX                        5.39          16.8        7.20                     4.11
10 AL, KY, MS                3.97           4.34       9.10                     4.16
# i 17 more rows
# i 1 more variable: SingleFamilyDetached <dbl>
```

A comment about `arrange`: Pass the variable into `desc()` to reverse the order. E.g. `arrange(desc(SingleFamilyAttached))`.

Subsetting rows

Next we take a quick look at just Michigan to demonstrate the use of `filter()`.

```r
recs_type_state %>% filter(state == 'MI')
```

```
# A tibble: 1 x 6
# Groups:   state [1]
  state ApartmentFew ApartmentMany MobileHome SingleFamilyAttached
  <chr>        <dbl>         <dbl>      <dbl>                <dbl>
1 MI            5.26          15.3       7.27                 2.78
# i 1 more variable: SingleFamilyDetached <dbl>
```

We might also want to find all states with at least 25% of people living in apartments,

```r
recs_type_state %>% filter(ApartmentFew + ApartmentMany >= 25)
```

```
# A tibble: 7 x 6
# Groups:   state [7]
  state              ApartmentFew ApartmentMany MobileHome SingleFamilyAttached
  <chr>                     <dbl>         <dbl>      <dbl>                <dbl>
1 IL                        11.5           19.9      NA                    3.03
2 AK, HI, OR, WA             7.93          20.0       8.13                 4.01
3 CT, ME, NH, RI, VT        13.9           16.5       1.49                 4.75
4 NY                        16.9           33.4       1.61                 5.29
5 MA                        24.4           21.1       1.58                 5.70
6 NJ                        11.3           14.5       1.88                 5.93
7 CA                         8.47          23.5       3.23                 7.01
# i 1 more variable: SingleFamilyDetached <dbl>
```

tibble

Tibbles are defined by the **tibble** package.

```
tb <- tibble(a = 1:3, b = letters[10:12])
tb
```

```
# A tibble: 3 x 2
      a b
  <int> <chr>
1     1 j
2     2 k
3     3 l
```

```
class(tb)
```

```
[1] "tbl_df"     "tbl"        "data.frame"
```

```
typeof(tb)
```

```
[1] "list"
```

As you can see, tibbles extend `data.frame` and by extension, extends `list`. So at its core, a tibble is again just a list of equally-lengthed vectors.

Differences from `data.frame`

Non-syntactically valid names

Tibbles do not enforce names to be syntactically valid.

```
df <- data.frame(a = 1:3,
                 "123" = 4:6,
                 "my data" = 7:9)
df
```

```
  a X123 my.data
1 1    4       7
2 2    5       8
3 3    6       9
```

```
tb <- tibble(a = 1:3,
             "123" = 4:6,
             "my data" = 7:9)
tb
```

```
# A tibble: 3 x 3
      a `123` `my data`
  <int> <int>     <int>
1     1     4         7
2     2     5         8
3     3     6         9
```

However, to refer to these non-syntactically valid names, you need to use the backticks.

```
tb$`123`
```

```
[1] 4 5 6
```

```
select(tb, `my data`)
```

```
# A tibble: 3 x 1
  `my data`
      <int>
1         7
2         8
3         9
```

Lazy evaluation

Tibbles are created sequentially rather than in parallel:

```
df <- data.frame(a = 1:3)
df$b <- df$a + 2
df
```

```
  a b
1 1 3
2 2 4
3 3 5
```

```r
tb <- tibble(a = 1:3,
             b = a + 2)
tb
```

```
# A tibble: 3 x 2
      a     b
  <int> <dbl>
1     1     3
2     2     4
3     3     5
```

row.names

Tibbles do not support row names.

```r
df
```

```
  a b
1 1 3
2 2 4
3 3 5
```

```r
tb
```

```
# A tibble: 3 x 2
      a     b
  <int> <dbl>
1     1     3
2     2     4
3     3     5
```

```r
row.names(df)
```

```
[1] "1" "2" "3"
```

```r
row.names(tb)
```

```
[1] "1" "2" "3"
```

```
row.names(df) <- letters[21:23]
```

```
row.names(tb) <- letters[21:23]
```

```
Warning: Setting row names on a tibble is deprecated.
```

```
df
```

```
  a b
u 1 3
v 2 4
w 3 5
```

```
tb
```

```
# A tibble: 3 x 2
      a     b
* <int> <dbl>
1     1     3
2     2     4
3     3     5
```

Watch out for this - it can lead to weird bugs if you try and use row names.

Recycling vectors

`data.frames` can recycle vectors as normal. Tibbles only recycle length-1 vectors. Imagine we're trying to create a data set containing each pairwise combination of "temperature" and "direction"

```
temperature <- c("low", "medium", "high")
setting <- c("forward", "backwards")
results <- rnorm(6)
df <- data.frame(temperature, setting, results)
df
```

```
  temperature   setting      results
1         low   forward -1.644495975
2      medium backwards  1.063998152
3        high   forward -0.007910344
4         low backwards -1.717917447
5      medium   forward -0.170544568
6        high backwards  0.274487266
```

```r
tibble(temperature, setting, results)
```

```
Error in `tibble()`:
! Tibble columns must have compatible sizes.
* Size 3: Existing data.
* Size 2: Column at position 2.
i Only values of size one are recycled.
```

```r
tb <- as_tibble(df)
tb
```

```
# A tibble: 6 x 3
  temperature setting    results
  <chr>       <chr>        <dbl>
1 low         forward   -1.64
2 medium      backwards  1.06
3 high        forward   -0.00791
4 low         backwards -1.72
5 medium      forward   -0.171
6 high        backwards  0.274
```

Subsetting

Subsetting a `data.frame` with `[]` can yield a vector or a `data.frame`, where-as a tibble always subsets to a tibble.

```r
df[, 2:3]
```

```
    setting      results
1   forward -1.644495975
2 backwards  1.063998152
```

```
3   forward -0.007910344
4 backwards -1.717917447
5   forward -0.170544568
6 backwards  0.274487266
```

```
tb[, 2:3]
```

```
# A tibble: 6 x 2
  setting    results
  <chr>        <dbl>
1 forward   -1.64
2 backwards  1.06
3 forward   -0.00791
4 backwards -1.72
5 forward   -0.171
6 backwards  0.274
```

```
df[, 3]
```

```
[1] -1.644495975  1.063998152 -0.007910344 -1.717917447 -0.170544568
[6]  0.274487266
```

```
tb[, 3]
```

```
# A tibble: 6 x 1
   results
     <dbl>
1 -1.64
2  1.06
3 -0.00791
4 -1.72
5 -0.171
6  0.274
```

If you do want a single-column `data.frame`, you can pass the `drop` option into the subset:

```
df[, 3, drop = FALSE]
```

```
       results
1 -1.644495975
2  1.063998152
3 -0.007910344
4 -1.717917447
5 -0.170544568
6  0.274487266
```

(Tibbles support `drop = TRUE` if you do want it to return a vector.)

Additionally, tibbles do not support partial-matching with `$`

```r
names(df)
```

```
[1] "temperature" "setting"     "results"
```

```r
df$temp
```

```
[1] "low"    "medium" "high"   "low"    "medium" "high"
```

```r
names(tb)
```

```
[1] "temperature" "setting"     "results"
```

```r
tb$temp
```

```
Warning: Unknown or uninitialised column: `temp`.
```

```
NULL
```

Printing tibbles

The most visually distinguishing difference between tibbles and `data.frames` is how much it prints by default.

```r
data(starwars)
starwars
```

```
# A tibble: 87 x 14
   name     height  mass hair_color skin_color eye_color birth_year sex   gender
   <chr>     <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
 1 Luke Sk~    172    77 blond      fair       blue            19   male  mascu~
 2 C-3PO       167    75 <NA>       gold       yellow         112   none  mascu~
 3 R2-D2        96    32 <NA>       white, bl~ red             33   none  mascu~
 4 Darth V~    202   136 none       white      yellow          41.9 male  mascu~
 5 Leia Or~    150    49 brown      light      brown           19   fema~ femin~
 6 Owen La~    178   120 brown, gr~ light      blue            52   male  mascu~
 7 Beru Wh~    165    75 brown      light      blue            47   fema~ femin~
 8 R5-D4        97    32 <NA>       white, red red             NA   none  mascu~
 9 Biggs D~    183    84 black      light      brown           24   male  mascu~
10 Obi-Wan~    182    77 auburn, w~ fair       blue-gray       57   male  mascu~
# i 77 more rows
# i 5 more variables: homeworld <chr>, species <chr>, films <list>,
#   vehicles <list>, starships <list>
```

As you can see, a large number of columns and rows were suppressed from the output. If we were to convert this to a `data.frame` and print, it would display the entire results

```r
# not evaluated!
as.data.frame(starwars)
```

The `print` function can control tibbles performance:

```r
print(starwars, n = 3, width = 50)
```

```
# A tibble: 87 x 14
  name          height  mass hair_color skin_color
  <chr>          <int> <dbl> <chr>      <chr>
1 Luke Skywalk~    172    77 blond      fair
2 C-3PO            167    75 <NA>       gold
3 R2-D2             96    32 <NA>       white, bl~
# i 84 more rows
# i 9 more variables: eye_color <chr>,
#   birth_year <dbl>, sex <chr>, gender <chr>,
#   homeworld <chr>, species <chr>, films <list>,
#   vehicles <list>, starships <list>
```

Note that `width` controls the actual width of the output, not the number of columns.

Tidyverse vs base R

I personally restrict use of the tidyverse as much as possible. There are a number of reasons for this, a few include:

1. Tidyverse changes its API and deprecates functions very rapidly.
2. Tidyverse uses nonstandard evaluation frequently.
3. Tidyverse packages have no issue overloading function names which can lead to confusing results depending on the order in which packages are loaded.
4. It is often more complex to do basic operations in tidyverse than base R.
5. Debugging long piped operations is challenging (a pipe problem rather than a specific tidyverse problem).
6. Using the tidyverse adds a massive set of requirements to your analysis.

Here are two useful links. The first is tidyverse's own document showing the equivalency of dplyr and base R commands: https://dplyr.tidyverse.org/articles/base.html

This second is a document which explains a lot of the issues with the tidyverse and why it isn't necessarily the best way to learn R or move R forward: https://github.com/matloff/TidyverseSkeptic