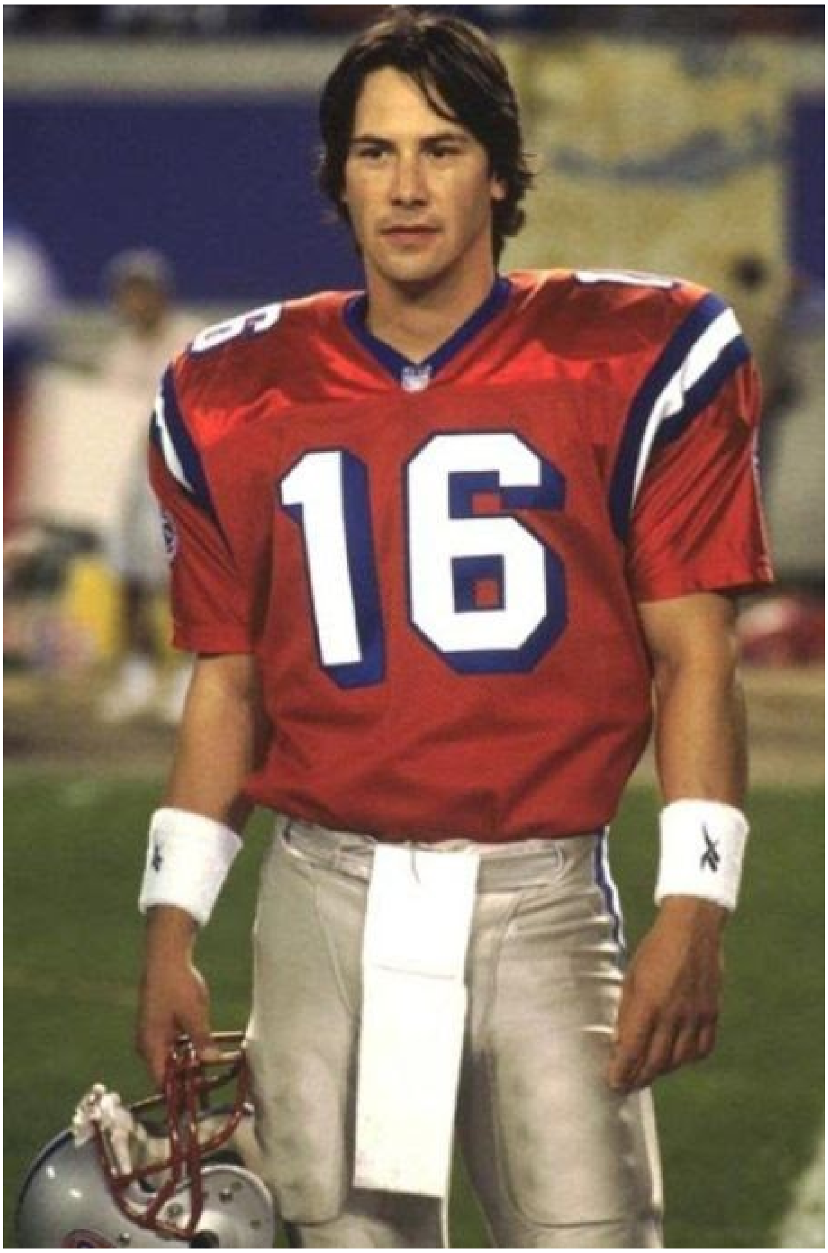
Requisite packages and data:

```
library(dplyr)

# data
library(AmesHousing)

# Load the housing data, clean names, then grab just six columns
# to simplify the dataset for display purposes.
ames_data <- make_ames() %>%
  janitor::clean_names() %>%
  select(sale_price, bsmt_fin_sf_1, first_flr_sf,
         total_bsmt_sf, neighborhood, gr_liv_area)
```



Superceding Functions

There are two major families of functions that supercede old functionality

The Replacements:

- **`across()`**

- **`slice()`**

In addition, there are some deprecated functions that are worth noting as well as some new mutate() arguments. In this post, I'll walk through some examples of each of these changes.

Across

All *_if(), *_at() and *_all() function variants were superseded in favor of across(). across() makes manipulating multiple columns more intuitive and consistent with other dplyr syntax.

across() is my favorite new dplyr function because I've always had to stop and think and pull up the docs when using mutate_if() and mutate_at(). Most notably, I appreciate the use of tidy selection rather than the vars() method used in mutate_at().

Let's see across() in action. Let's say we want to convert all the square foot variables to square yards. When we take a look at our data, we see that all of the square foot variables either contain "area" or "_sf" in their names.

```
feet_to_yards <- function(x) {x / 9}
```

Here is the old way this was done with mutate_at():

```
ames_data %>%
  mutate_at(.vars = vars(contains("_sf") | contains("area")) , .funs =
feet_to_yards)
```

Now we use across() in combination with a vector. In this case, we used contains() to grab variable names that contain "_sf" or "area".

```
ames_data %>%
  mutate(across(.cols = c(contains("_sf"), contains("area")), .fns =
feet_to_yards)) %>%
  head()
```

```
## # A tibble: 6 x 6
##   sale_price bsmt_fin_sf_1 first_flr_sf total_bsmt_sf neighborhood
gr_liv_area
##
## 1     215000         0.222        184             120  North_Ames          184
## 2     105000         0.667         99.6            98  North_Ames
99.6
## 3     172000         0.111        148.            148. North_Ames          148.
## 4     244000         0.111        234.            234. North_Ames          234.
## 5     189900         0.333        103.            103. Gilbert             181
## 6     195500         0.333        103.            103. Gilbert             178.
```

across() can also replace mutate_if() in combination with where().

Old way with mutate_if():

```
ames_data %>%
  mutate_if(is.numeric, log)
```

New way with across(where()):

```
## new dplyr(log transform numeric values)
ames_data %>%
  mutate(across(where(is.numeric), log)) %>%
  head()
```

```
## # A tibble: 6 x 6
```

```
##   sale_price bsmt_fin_sf_1 first_flr_sf total_bsmt_sf neighborhood
gr_liv_area
##
## 1       12.3          0.693          7.41          6.98 North_Ames
7.41
## 2       11.6          1.79           6.80          6.78 North_Ames
6.80
## 3       12.1          0              7.19          7.19 North_Ames
7.19
## 4       12.4          0              7.65          7.65 North_Ames
7.65
## 5       12.2          1.10           6.83          6.83 Gilbert
7.40
## 6       12.2          1.10           6.83          6.83 Gilbert
7.38
```

summarize() now uses the same across() and where() syntax that we used above with mutate. Let's find the average of all numerics columns for each neighborhood.

```
ames_data %>%
  group_by(neighborhood) %>%
  summarize(across(where(is.numeric), mean, .names = "mean_{col}")) %>%
  head()
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 6 x 6
##   neighborhood mean_sale_price mean_bsmt_fin_s… mean_first_flr_…
##
## 1 North_Ames            145097.             3.66            1175.
## 2 College_Cre…          201803.             4.01            1173.
## 3 Old_Town              123992.             5.80             945.
## 4 Edwards               130843.             4.27            1115.
## 5 Somerset              229707.             4.59            1188.
## 6 Northridge_…          322018.             3.99            1613.
## # … with 2 more variables: mean_total_bsmt_sf , mean_gr_liv_area
```

As you can see, we calculated the neighborhood average for all numeric values. On top of that, we were able to easily prefix the column names with "mean_" thanks to another useful across() argument called ".names".

Not only that, in conjunction with the where() helper, across() unifies "_if" and "_at" semantics, allowing more intuitive and elegant column selection. For example, let's mutate the square footage variables that are integers (like mutate_if()), and the square footage variables that end with "_sf" (like mutate_at()) to make them doubles.

```
ames_data %>%
  mutate(across(where(is.integer) & ends_with("_sf"), as.double))
```

```
## # A tibble: 2,930 x 6
##    sale_price bsmt_fin_sf_1 first_flr_sf total_bsmt_sf neighborhood
gr_liv_area
##
##  1     215000             2         1656          1080 North_Ames
1656
##  2     105000             6          896           882 North_Ames
896
##  3     172000             1         1329          1329 North_Ames
1329
##  4     244000             1         2110          2110 North_Ames
```

```
2110
## 5     189900              3          928           928 Gilbert
1629
## 6     195500              3          926           926 Gilbert
1604
## 7     213500              3         1338          1338 Stone_Brook
1338
## 8     191500              1         1280          1280 Stone_Brook
1280
## 9     236500              3         1616          1595 Stone_Brook
1616
## 10    189000              7         1028           994 Gilbert
1804
## # … with 2,920 more rows
```

Notice, the “first_flr_sf” was converted to a double, but the “gr_living_area” remains an integer because it doesn’t fit the criteria aends_with(“_sf”).

across() can also perform mutate_all() functionality with across(everything(), …

Slice

---

top_n(), sample_n(), and sample_frac() have been superseded in favor of a new family of slice() helpers.

Reasons for future deprecation:

- top_n() – has a confusing name that might reasonably be considered to filter for the min or the max rows. For example, let’s stay we have data for a track and field race that records lap times. One might reasonable assume that top_n() would return the fastest times but they actually return the times that took the longest. top_n() has been superseded by slice_min(), and slice_max().
- sample_n() and sample_frac() – it’s easier to remember (and pull up documentation for) two mutually exclusive arguments to one function called slice_sample().

```
# Old way to grab the five most expensive homes by sale price
ames_data %>%
  top_n(n = 5, wt = sale_price)

# New way to grab the five most expensive homes by sale price
ames_data %>%
  slice_max(sale_price, n = 5)
```

```
## # A tibble: 5 x 6
##   sale_price bsmt_fin_sf_1 first_flr_sf total_bsmt_sf neighborhood
gr_liv_area
##
## 1     755000             3         2444          2444 Northridge
4316
## 2     745000             3         2411          2396 Northridge
4476
## 3     625000             3         1831          1930 Northridge
3627
## 4     615000             3         2470          2535 Northridge_He…
2470
## 5     611657             3         2364          2330 Northridge_He…
2364
```

```
# You can also grab the five cheapest homes
ames_data %>%
```

```
  slice_min(sale_price, n = 5)

## # A tibble: 5 x 6
##   sale_price bsmt_fin_sf_1 first_flr_sf total_bsmt_sf neighborhood
gr_liv_area
##
## 1      12789             7          832           678 Old_Town
832
## 2      13100             5          733             0 Iowa_DOT_and_…
733
## 3      34900             6          720           720 Iowa_DOT_and_…
720
## 4      35000             7          498           498 Edwards
498
## 5      35311             2          480           480 Iowa_DOT_and_…
480

# Old way to sample four random rows(in this case properties)
ames_data %>%
  sample_n(4)

# New way to sample four random rows(in this case properties)
ames_data %>%
  slice_sample(n = 4)

## # A tibble: 4 x 6
##   sale_price bsmt_fin_sf_1 first_flr_sf total_bsmt_sf neighborhood
gr_liv_area
##
## 1     119000             6          948           948 Edwards
948
## 2     156000             1          990           990 College_Creek
990
## 3     245700             3         1614          1614 Northridge_He…
1614
## 4     108000             2         1032          1032 Old_Town
1032

# Old way to sample a random 0.2% of the rows
ames_data %>%
  sample_frac(0.002)

# New way to sample a random 0.2% of the rows
ames_data %>%
  slice_sample(prop = 0.002)

## # A tibble: 5 x 6
##   sale_price bsmt_fin_sf_1 first_flr_sf total_bsmt_sf neighborhood
gr_liv_area
##
## 1     110000             7          682           440 Old_Town
1230
## 2     136000             6         1040          1040 North_Ames
1040
## 3     208000             1         1182           572 Crawford
1982
## 4     115000             1          789           789 Old_Town
789
## 5     145500             1         1053          1053 North_Ames
```

```
1053
```

Additionally, slice_head() and slice_tail() can be used to grab the first or last rows, respectively.

Nest By

---

nest_by() works similar to group_by() but is more visual because it changes the structure of the tibble instead of just adding grouped metadata. With nest_by(), the tibble transforms into a rowwised dataframe (Run vignette("rowwise") to learn more about the revised rowwise funtionality in dplyr 1.0.0).

First, for the sake of comparison, let's calculate the average sale price by neighborhood using group_by() and summarize():

```
ames_data %>%
  group_by(neighborhood) %>%
  summarise(avg_sale_price = mean(sale_price)) %>%
  ungroup() %>%
  head()
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 6 x 2
##   neighborhood       avg_sale_price
##
## 1 North_Ames                 145097.
## 2 College_Creek              201803.
## 3 Old_Town                   123992.
## 4 Edwards                    130843.
## 5 Somerset                   229707.
## 6 Northridge_Heights         322018.
```

The summarize() operation works well with group_by(), particularly if the output of the summarization function are single numeric values. But what if we want to perform a more complicated operation on the grouped rows? Like, for example, a linear model. For that, we can use nest_by() which stores grouped data not as metadata but as lists in a new column called "data".

```
nested_ames <- ames_data %>%
  nest_by(neighborhood)
```

```
head(nested_ames)
```

```
## # A tibble: 6 x 2
## # Rowwise:  neighborhood
##   neighborhood                     data
##                  >
## 1 North_Ames                [443 × 5]
## 2 College_Creek             [267 × 5]
## 3 Old_Town                  [239 × 5]
## 4 Edwards                   [194 × 5]
## 5 Somerset                  [182 × 5]
## 6 Northridge_Heights        [166 × 5]
```

As you can see, nest_by() fundementally changes the structure of the dataframe unlike group_by(). This feature becomes useful when you want to apply a model to each row of the nested data.

For example, here is a linear model that uses square footage to predict sale price applied to each neighborhood.

```
nested_ames_with_model <- nested_ames %>%
  mutate(linear_model = list(lm(sale_price ~ gr_liv_area, data = data)))
```

```
head(nested_ames_with_model)
```

```
## # A tibble: 6 x 3
## # Rowwise:  neighborhood
##   neighborhood                      data linear_model
##                  >
## 1 North_Ames                 [443 × 5]
## 2 College_Creek              [267 × 5]
## 3 Old_Town                   [239 × 5]
## 4 Edwards                    [194 × 5]
## 5 Somerset                   [182 × 5]
## 6 Northridge_Heights         [166 × 5]
```

It's important to note that the model must be vectorized, a tranformation performed here with list(). Let's take a look at the model that was created for the "North_Ames" neighborhood.

```
north_ames_model <- nested_ames_with_model %>%
  filter(neighborhood == "North_Ames") %>%
  pull(linear_model)

north_ames_model
```

```
## [[1]]
##
## Call:
## lm(formula = sale_price ~ gr_liv_area, data = data)
##
## Coefficients:
## (Intercept)  gr_liv_area
##    74537.97        54.61
```

The model shows that for each additional square foot, a house in the North Ames neighborhood is expected to sell for about $54.61 more.

Additional Mutate arguments

Control what columns are retained with ".keep"

```
# For example "used" retains only the columns involved in the mutate
ames_data %>%
  mutate(sale_price_euro = sale_price / 1.1, .keep = "used") %>%
  head()
```

```
## # A tibble: 6 x 2
##   sale_price sale_price_euro
##
## 1     215000         195455.
## 2     105000          95455.
## 3     172000         156364.
## 4     244000         221818.
## 5     189900         172636.
## 6     195500         177727.
```

Control where the new columns are added with ".before" and ".after"

```
# For example, make the "sale_price_euro" column appear to the left of the
"sale_price" column like this
ames_data %>%
```

```
  mutate(
    sale_price_euro = sale_price / 1.1, .keep = "used", .before = sale_price
  ) %>%
  head()
```

```
## # A tibble: 6 x 2
##   sale_price_euro sale_price
##
## 1         195455.     215000
## 2          95455.     105000
## 3         156364.     172000
## 4         221818.     244000
## 5         172636.     189900
## 6         177727.     195500
```

Conculsion

This was a short, high level look at my favorite new features coming in dplyr 1.0.0. The two major changes were the addition of across() and slice() which supercede old functionality. across() makes it easy to mutate specific columns or rows in a more intuitive, consistent way. slice() makes similar improvements to data sampling methods. I am also a big fan of the new nest_by() functionality, and plan to search for elegant ways to incorporate it in my upcoming R projects. These changes align dplyr syntax more closely with conventions common in the tidyverse. Thanks tidyverse team for continually pushing the boundaries to make data analytics easier in practice and to learn/teach!