Handout 08: Miscellaneous R

03 March 2017

Relabeling categories

Let's read in the bikes dataset from the first round of data analysis projects. Notice that the season and weather conditions are labeled as integers.

```
bikes <- read_csv("https://statsmaths.github.io/stat_data/bikes.csv")
bikes
```

```
## # A tibble: 731 × 9
##    season  year workingday weather      temp
##     <int> <int>      <int>   <int>     <dbl>
## 1       1     0          0       2  6.715022
## 2       1     0          0       2  7.989548
## 3       1     0          1       1 -3.039976
## 4       1     0          1       1 -2.800000
## 5       1     0          1       1 -1.020838
## 6       1     0          1       1 -2.513032
## 7       1     0          1       2 -3.029548
## 8       1     0          0       2 -5.110000
## 9       1     0          0       1 -6.870022
## 10      1     0          1       1 -6.045022
## # ... with 721 more rows, and 4 more
## #   variables: humidity <dbl>,
## #   windspeed <dbl>, registered <int>,
## #   count <int>
```

What if we wanted to create a new variable that had these properly labeled? To do so we can use the `ifelse` function together with mutate. We first set a default value, and then override the default one by one.

```
bikes <- mutate(bikes, season_name = "missing")
bikes <- mutate(bikes, season_name = ifelse(season == 1, "Winter", season_name))
bikes <- mutate(bikes, season_name = ifelse(season == 2, "Spring", season_name))
bikes <- mutate(bikes, season_name = ifelse(season == 3, "Summer", season_name))
bikes <- mutate(bikes, season_name = ifelse(season == 4, "Fall", season_name))
select(bikes, season, season_name)
```

```
## # A tibble: 731 × 2
##    season season_name
##     <int>       <chr>
## 1       1      Winter
## 2       1      Winter
```

```
## 3       1      Winter
## 4       1      Winter
## 5       1      Winter
## 6       1      Winter
## 7       1      Winter
## 8       1      Winter
## 9       1      Winter
## 10      1      Winter
## # ... with 721 more rows
```

It looks like this requires a lot of code, but its actually very little if you make good use of copy and paste.

This relabeling is very useful, and could be used by almost all of the datasets at some point. It can also be used to change labels that we already have or to create hand-constructed buckets.

Sampling and selecting data

Another function that we have not seen that may be useful with the larger datasets in data analysis two are `sample_frac` and `select`. We just saw select in the last code chunk; we give it a dataset name followed by the variable names we want to select from the data. These are separated by a comma.

```
select(bikes, season, year, weather)
```

```
## # A tibble: 731 × 3
##    season  year weather
##     <int> <int>   <int>
## 1       1     0       2
## 2       1     0       2
## 3       1     0       1
## 4       1     0       1
## 5       1     0       1
## 6       1     0       1
## 7       1     0       2
## 8       1     0       2
## 9       1     0       1
## 10      1     0       1
## # ... with 721 more rows
```

This is sometimes useful if you are doing something like a group summarize on a large dataset and do not want to be overwhelmed with the output. I often use it before running `left_join`, or the two functions we will see on Wednesday.

The function `sample_frac` is useful to take a random subset of your data. This is great for initial exploration of a large dataset, or for when trying to do very large scatter plot. We give it the dataset name followed by the percentage of the data to keep.

```
sample_frac(bikes, 0.1)
```

```
## # A tibble: 73 × 10
##    season  year workingday weather      temp
##     <int> <int>      <int>   <int>     <dbl>
## 1       1     0          0       2  6.715022
## 2       3     0          0       2 31.960022
## 3       1     0          1       2  4.890452
## 4       3     0          1       1 30.749978
## 5       1     0          0       1  5.945000
## 6       4     0          1       2 18.980000
## 7       3     0          0       2 28.660022
## 8       3     1          1       2 27.944978
## 9       2     1          0       1 27.835022
## 10      3     0          1       1 31.795022
## # ... with 63 more rows, and 5 more
## #   variables: humidity <dbl>,
## #   windspeed <dbl>, registered <int>,
## #   count <int>, season_name <chr>
```

Notice that the sample will be different each time this is run. That can be useful in some situations (with plots you want to make sure the specific sample does not change anything important), but in other can be tricky. For example, I used this to generate the 10% sample of airline data but don't want to accidentally change the sample if I run the code again. To do this, run `set.seed` with some constant value first:

```
set.seed(1)
sample_frac(bikes, 0.1)
```

```
## # A tibble: 73 × 10
##    season  year workingday weather      temp
##     <int> <int>      <int>   <int>     <dbl>
## 1       3     0          1       1 28.934978
## 2       4     0          1       1 24.700022
## 3       1     1          1       1 10.124978
## 4       4     1          1       1 19.915022
## 5       2     0          1       1 28.990022
## 6       4     1          0       1 18.430022
## 7       4     1          1       2  5.230022
```

```
## 8       2     1          1       2  5.230022
## 9       2     1          1       2  8.420000
## 10      1     0          1       1 11.390000
## # ... with 63 more rows, and 5 more
## #   variables: humidity <dbl>,
## #   windspeed <dbl>, registered <int>,
## #   count <int>, season_name <chr>
```

This sample will not change if I run it again (as both lines).

Advanced summarizing

I wrote the function `group_summarize` because I found that MATH 209 students struggled using the raw summarizing commands early in the semester. You may find that you need to do some time of summarization that we did not cover, so here are some notes on how to do it.

We have to use the function `group_by` and the function summarize on the dataset. The first tells R which variables to summarize by, but the second tells it which new variables to create:

```
summarize(group_by(bikes, season_name), min_temp = min(temp),
          max_temp = max(temp))
```

```
## # A tibble: 4 × 3
##   season_name   min_temp max_temp
##         <chr>      <dbl>    <dbl>
## 1        Fall  -1.425022 27.39500
## 2      Spring   0.700838 37.34998
## 3      Summer  14.965022 40.87002
## 4      Winter -12.097394 21.78500
```

Each of the new variables, however, must be described explicitly.[1] Here we are able to compute the minimum and maximum

Group by can also be combined with the `mutate` function to append summary statistics to a group of variables. For example, if we wanted to add the average temperature of each season to every row of the dataset, we would do this:

[1] Even more advanced functions `summarize_at`, `summarize_each`, `summarize_if`, and `summarize_all` allow for writing generic formulas for describing patterns of functions. I use these in the group summarize function.

```
temp <- mutate(group_by(bikes, season_name), avg_temp = mean(temp))
```

I find this command to be very helpful with the sports data.

Gathering data

In some cases we have datasets where multiple columns could be treated as individual observations. What does that mean? Think of

the cancer dataset we used earlier in the semester, taking off just the cancer incidence rates:

```
cancer <- read_csv("https://statsmaths.github.io/stat_data/cancer_inc_data.csv")
cancer <- select(cancer, breast, colorectal, prostate, lung, melanoma)
cancer
```

```
## # A tibble: 1,961 × 5
##    breast colorectal prostate  lung melanoma
##     <dbl>      <dbl>    <dbl> <dbl>    <dbl>
## 1   127.8       40.9    113.0  61.5     14.5
## 2   133.2       39.2     82.1  58.1     13.1
## 3   101.6       36.9    117.9  35.1     13.7
## 4   127.4       41.0     96.5  64.9     12.3
## 5   119.8       37.5    121.4  69.7     15.0
## 6   150.8       46.2    110.6  74.9     25.4
## 7   117.5       41.5    121.3  66.9     30.9
## 8   132.6       49.6    159.8  74.6     12.3
## 9   158.1       44.5    116.7  86.4     15.6
## 10  110.2       41.1     95.6  69.7     27.1
## # ... with 1,951 more rows
```

Several students at the time asked how we could plot all of the cancer types on the same plot. The canonical way of doing this would be to make a new dataset where each row, instead of being a single county, is then just one incidence rate. In other words each county will have five rows associated with it. To do this we use the `gather` function

```
gather(cancer)
```

```
## # A tibble: 9,805 × 2
##       key value
##     <chr> <dbl>
## 1  breast 127.8
## 2  breast 133.2
## 3  breast 101.6
## 4  breast 127.4
## 5  breast 119.8
## 6  breast 150.8
## 7  breast 117.5
## 8  breast 132.6
## 9  breast 158.1
## 10 breast 110.2
## # ... with 9,795 more rows
```

If we want to give the name of the `key' andvalue', those are given as the next two parameters to gather:

```
gather(cancer, type, incidence)
```

```
## # A tibble: 9,805 × 2
##      type incidence
##     <chr>     <dbl>
## 1  breast     127.8
## 2  breast     133.2
## 3  breast     101.6
## 4  breast     127.4
## 5  breast     119.8
## 6  breast     150.8
## 7  breast     117.5
## 8  breast     132.6
## 9  breast     158.1
## 10 breast     110.2
## # ... with 9,795 more rows
```

In many cases, there are other variables that we want to be duplicated along with the other keys. For example, take the speed skating dataset, selecting off a few variables to make it a bit more tractable:

```
speed <- read_csv("https://statsmaths.github.io/stat_data/speed_skate.csv")
speed <- select(speed, num_skater, nationality, time_lap1, time_lap2, time_lap3,
                time_lap4, time_lap5)
```

We can indicate the variables that should not be gathered by including them after the key and value terms with minus signs:

```
gather(speed, skater, value, -num_skater, -nationality)
```

```
## # A tibble: 23,520 × 4
##    num_skater nationality    skater value
##         <int>       <chr>     <chr> <dbl>
## 1         175         RUS time_lap1  6.91
## 2         100         CAN time_lap1  6.69
## 3         138         ITA time_lap1  6.87
## 4         182         TUR time_lap1  7.30
## 5          92         AUS time_lap1  7.00
## 6         190         USA time_lap1  6.69
## 7         180         TPE time_lap1  6.80
## 8         149         KAZ time_lap1  7.20
## 9         183         UKR time_lap1  7.03
## 10        120         GBR time_lap1  6.93
## # ... with 23,510 more rows
```

Or, we can not use the minus sign and include on those variables that should be gathered:

```
gather(speed, skater, value, time_lap1, time_lap2, time_lap3,
               time_lap4, time_lap5)
```

```
## # A tibble: 23,520 × 4
##    num_skater nationality    skater value
##         <int>       <chr>     <chr> <dbl>
## 1         175         RUS time_lap1  6.91
## 2         100         CAN time_lap1  6.69
## 3         138         ITA time_lap1  6.87
## 4         182         TUR time_lap1  7.30
## 5          92         AUS time_lap1  7.00
## 6         190         USA time_lap1  6.69
## 7         180         TPE time_lap1  6.80
## 8         149         KAZ time_lap1  7.20
## 9         183         UKR time_lap1  7.03
## 10        120         GBR time_lap1  6.93
## # ... with 23,510 more rows
```

The results are the same; in most cases one or the other will lead to less typing. There is also a complement to gathering called spreading, available through the function `spread`. You should not need that in this class because every dataset is maximally spread already.[2]

[2] It is also a lot more difficult to use because you have to be careful about implicit missing values and what to do with them.

Faceting

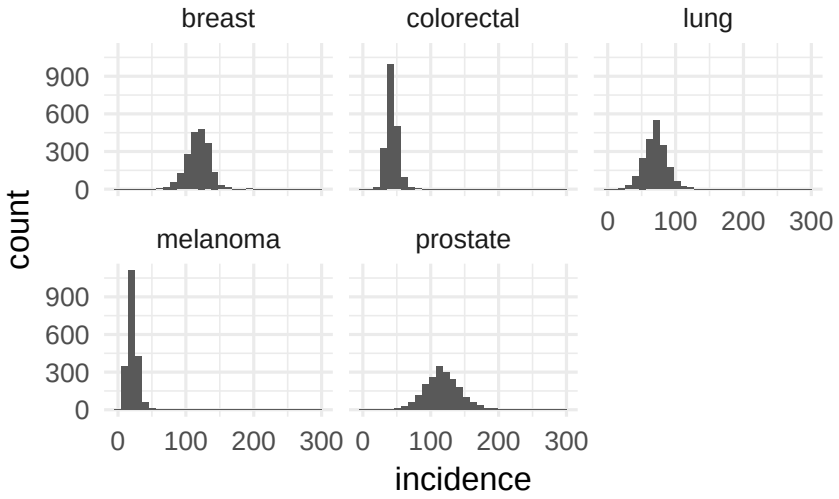
I have seen a lot of you following this pattern:

- Create subsets of the data based on a category you are interested in.
- Replicating plots for each of these groups and trying to compare them.

This pattern often makes sense, but there is a better way to do this all in one step with facets. They are also really easy to use!

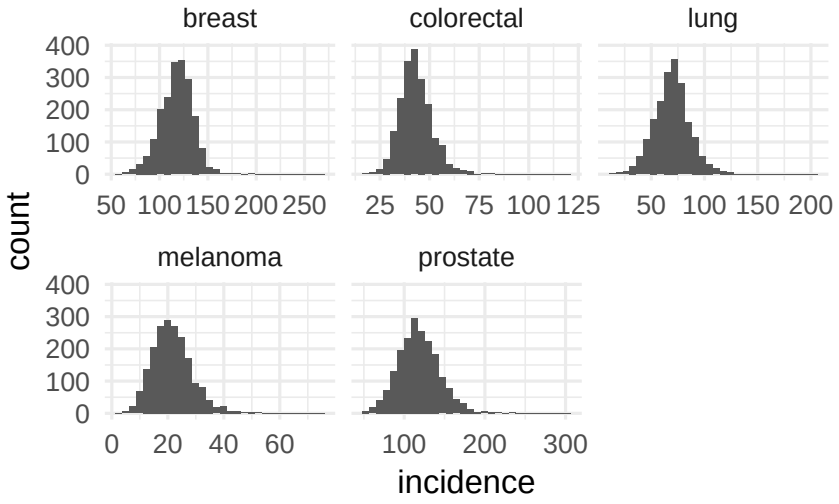
Whenever you have a plot and you want that plot to be replicated separately over every value of a variable, simply add `facet_wrap(~variable)` to the plot like this:

```
temp <- gather(cancer, type, incidence)
qplot(incidence, data = temp) + facet_wrap(~type)
```



A useful option is to add `scales = "free_x"` to allow the x-scale scale of each plot to vary:

```
qplot(incidence, data = temp) + facet_wrap(~type, scale = "free_x")
```



You may also set `free_y` to vary the y-axis and `free` to vary both. Notice how you can cleverly use gather and facet together.

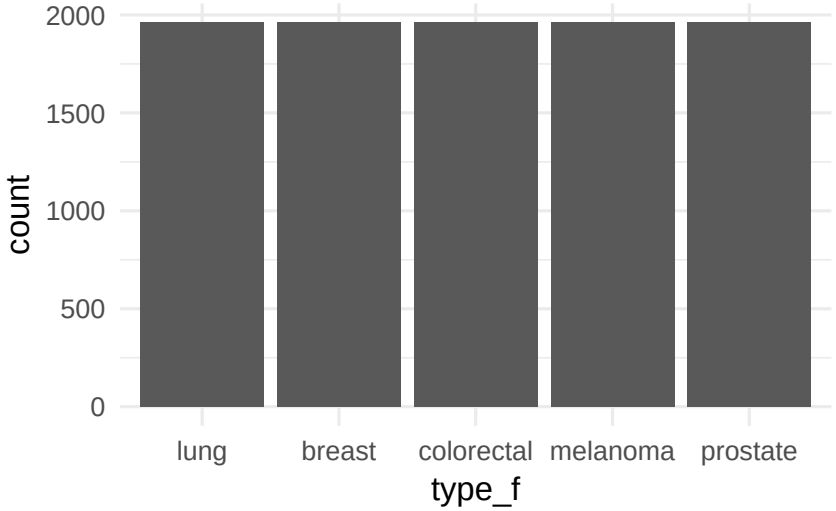
Releveling factors

Finally, as promised, here is how to relevel a factor value to any order you would like. First, specify that the variable you want is a factor, or create a new one that is:

```
temp <- gather(cancer, type, incidence)
temp <- mutate(temp, type_f = factor(type))
```

And then use `fct_relevel` with the new list of the levels **in quotes**:

```
temp <- mutate(temp, type_f = fct_relevel(type, "lung", "breast",
               "colorectal", "melanoma", "prostate"))
qplot(type_f, data = temp)
```



Which, from the plot you can has been properly ordered. Of course, this plot is uninteresting because every type has exactly the same number of observations.

Testing set containment

We have seen many times how to test whether a variable is equal to a specific value using == or, in the case of numeric variables, to see if it is greater than smaller than some fixed cut-off. In the case of categorical variables we sometimes may want to see whether a variable is equal to a set of values. For example, take the speed skating dataset:

```
speed <- read_csv("https://statsmaths.github.io/stat_data/speed_skate.csv")
```

How would we construct a subset of only those skaters from the United States, Canada, and Germany? Of course, this would work for any specific category:

```
temp <- filter(speed, nationality == "CAN")
```

But to allow for country being in a set of values we need to combine the operator `%in%` and function `c` as follows:

```
temp <- filter(speed, nationality %in% c("CAN", "USA", "GER"))
```

And we can see it worked by tabulating the results:

```
select(group_summarize(temp, nationality), nationality, n)
```

```
## # A tibble: 3 × 2
##   nationality     n
##         <chr> <int>
## 1         CAN   403
## 2         GER   148
## 3         USA   266
```

Notice that only skaters from these three countries remain.

Tables

We did briefly mention this at one point, but if you want to quickly see just the count of one or more categorical variables (or numeric ones that can be converted to distinct categories, like year/month/day of the week) we may just directly use the function `table`. For example the last code snippet could have been more quickly arrived at by:

```
table(temp$nationality)
```

```
##
## CAN GER USA
## 403 148 266
```

Or, we could see where each of these skaters competed with a two-way table:

```
table(temp$nationality, temp$country)
```

```
##
##       CAN CHN GER HUN ITA JPN KOR NED RUS
##   CAN 123  52  43   9  18  38  28   8  58
##   GER  25  19  18   3   9  10  15  12  24
##   USA  80  38  25   6  12  19  25   6  32
##
##       SWE TUR USA
##   CAN   0  11  15
##   GER   9   2   2
##   USA   0   3  20
```

Of course, we could also do this with group summarize, but tables get to the point much faster.

Filtering by another table

Now that we know about the `%in%` command and tables, we can use them to do some more advanced filtering. Let's find the names of the most common skaters:

```
tab <- table(speed$name)
skater_name <- names(tab[tab > 60])
skater_name
```

```
##  [1] "AN Victor"
##  [2] "BREEUWSMA Daan"
##  [3] "Dmitry MIGUNOV"
##  [4] "ELISTRATOV Semen"
##  [5] "Freek van der WART"
##  [6] "HAMELIN Charles"
##  [7] "Jon ELEY"
##  [8] "KNEGT Sjinkie"
##  [9] "KNOCH Viktor"
## [10] "LEPAPE Sebastien"
## [11] "LIU Shaolin Sandor"
## [12] "PARK Se Yeong"
## [13] "Paul STANLEY"
## [14] "SHOEBRIDGE Richard"
## [15] "Vladimir GRIGOREV"
## [16] "WU Dajing"
```

We can subset the skater data to include only these skaters now by:

```
temp <- filter(speed, name %in% skater_name)
dim(temp)
```

```
## [1] 1334   27
```

This would be useful, for example, in filtering out the most common players in the NBA dataset or filtering out the most common dropoff locations in the taxi dataset.

Flipping the coordinates

I've noticed that many of you have successfully made use of the facet and `ifelse` commands to build nicer and more readable plots. One thing you may find is that if categories have names that are too long the plots may become difficult to read. For example, let's take the reduced dataset that has only the most frequent skaters and produce a bar plot:

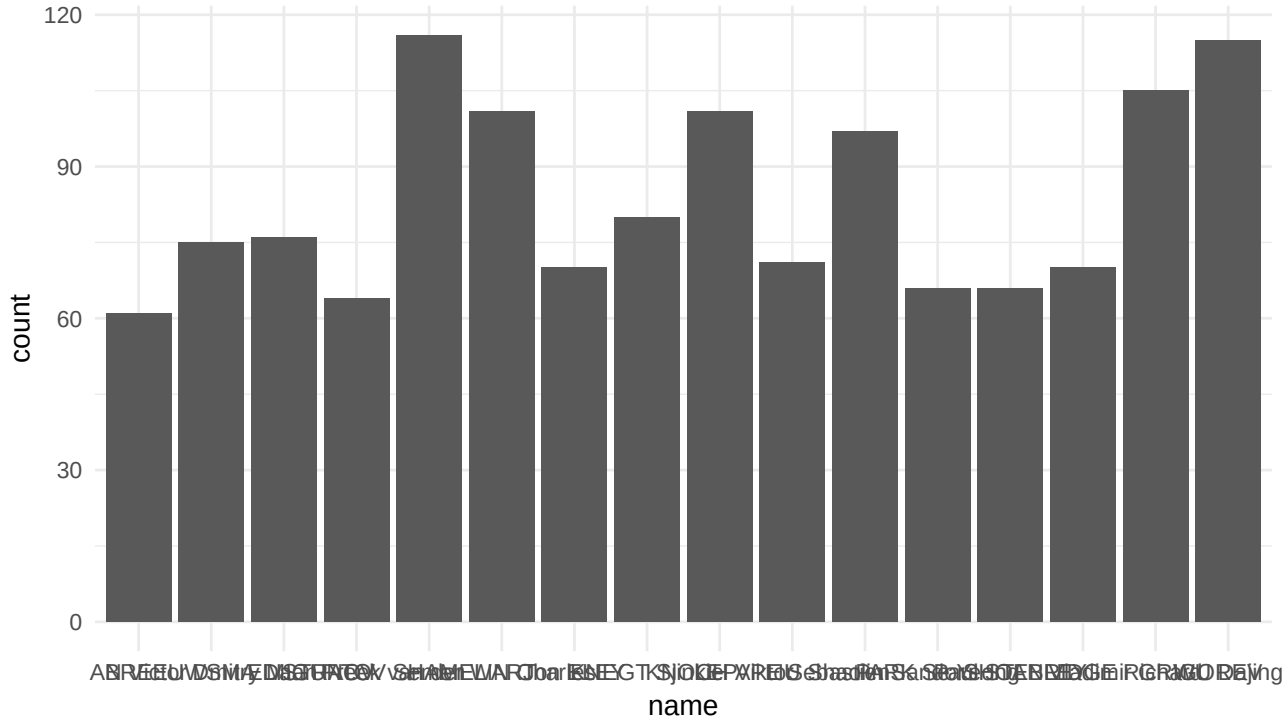
```
qplot(name, data = temp)
```

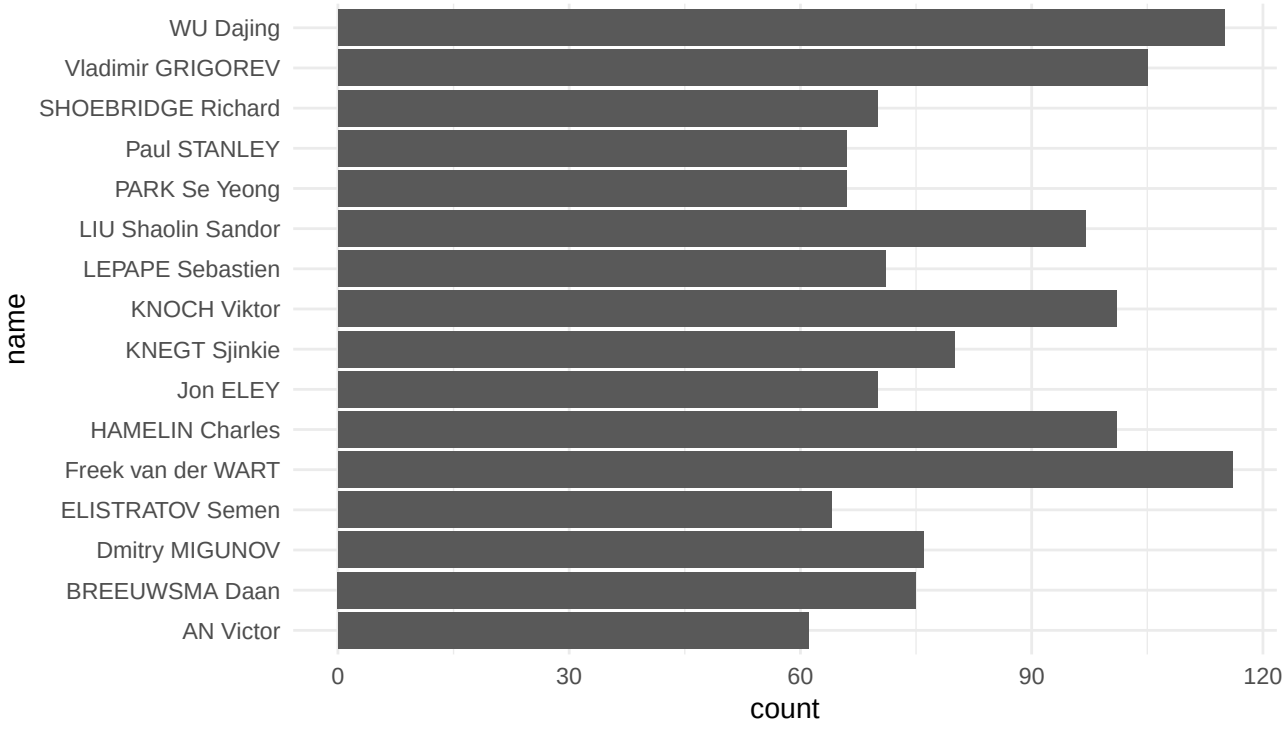
Adding the `coord_flip` command flips the x and y-axes to make this more readable:

```
qplot(name, data = temp) + coord_flip()
```

Of course, this is useful only when doing barplots and boxplots.[3] Otherwise we could just flip them manually as with a scatter plot.

[3] The first has only one input and the second requires the categorical one to be on the x-axis, until we flip it at least.


120
90
60
30
0
count
name


WU Dajing
Vladimir GRIGOREV
SHOEBRIDGE Richard
Paul STANLEY
PARK Se Yeong
LIU Shaolin Sandor
LEPAPE Sebastien
KNOCH Viktor
KNEGT Sjinkie
Jon ELEY
HAMELIN Charles
Freek van der WART
ELISTRATOV Semen
Dmitry MIGUNOV
BREEUWSMA Daan
AN Victor
name
0
30
60
90
120
count