09 - R Basics II

ST 597 | Spring 2017
University of Alabama

09-rbasics2.pdf

Factor Vectors

Missing and Special Data

Formatting

Recoding

Working with Missing Values

Recoding Numerical Data

General Recoding

Functions

Required Packages

```
library(tidyverse)
library(nycflights13)
# below are part of tidyverse, but not auto loaded
library(stringr)
library(scales)
library(forcats)
```

Factor Vectors

Factor Vectors

Factors are like character vectors, but in addition to the elements of the vector all possible (unique) values (or `levels`) are also stored.

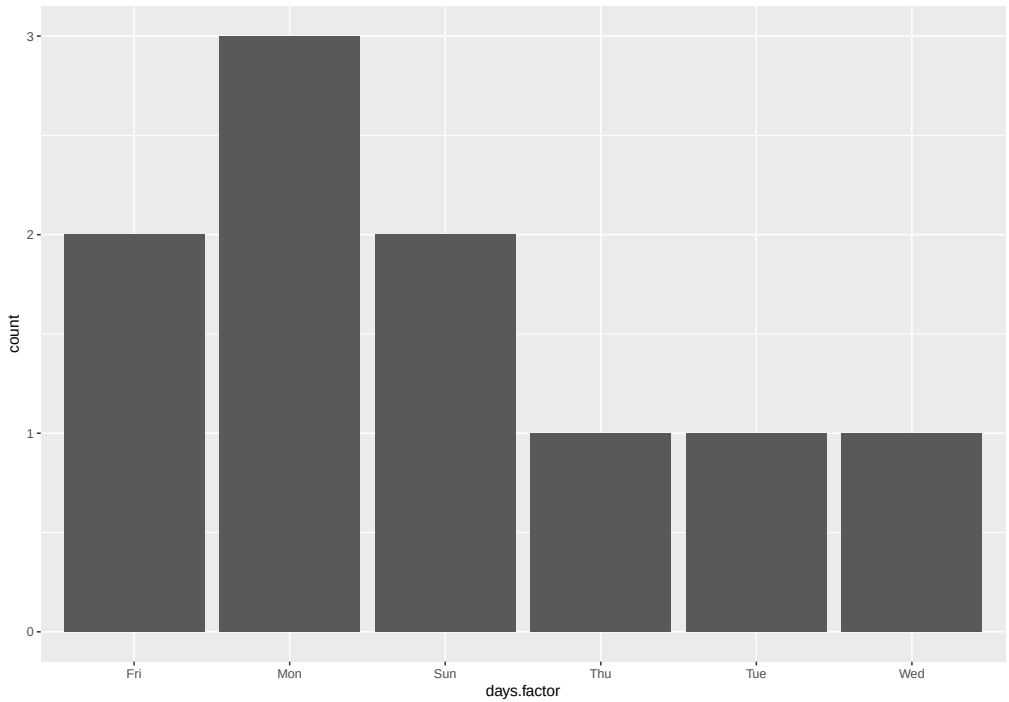
```
days = c("Tue","Sun","Mon","Fri","Thu","Fri","Wed","Mon","Mon","Sun")
days.factor = factor(days)    # convert to factor vector
days.factor                   # defaults to alphabetical order
#>  [1] Tue Sun Mon Fri Thu Fri Wed Mon Mon Sun
#> Levels: Fri Mon Sun Thu Tue Wed
```

Plotting Factors

In ggplot2 the level information determines the order of the factors

```
#- ggplot2 requires data frames/tibbles
df = tibble(days, days.factor)

ggplot(df, aes(days.factor)) + geom_bar()
# equivalent to:
# df %>% count(days.factor) %>% ggplot() + geom_col(aes(days.factor, n
```



Factor Levels

I usually think of factors as a set of outcomes from a categorical random variable, where the `levels` are the sample space.

```
levels(days.factor)
#> [1] "Fri" "Mon" "Sun" "Thu" "Tue" "Wed"
levels(days)  # days is not a factor-it has no levels!
#> NULL
```

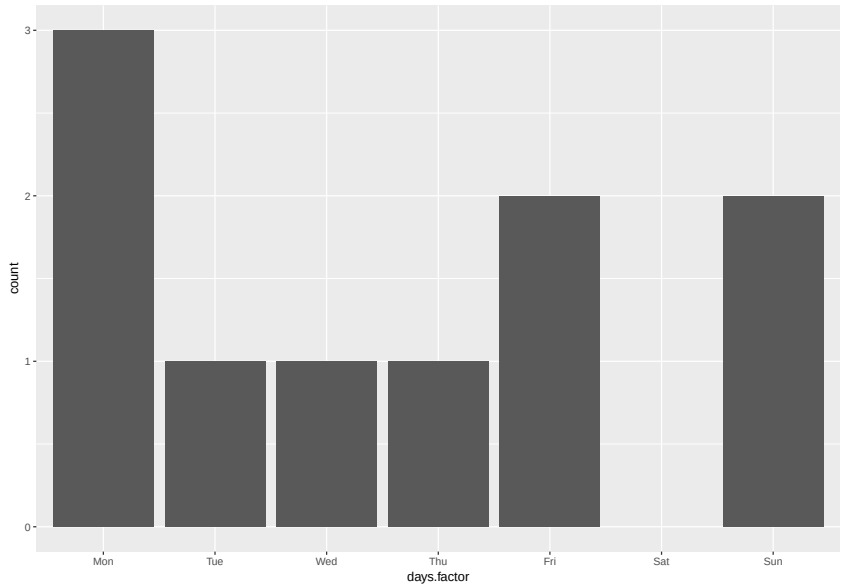
Notice that `days.factor` only has 6 days (missing `Sat`). How is R supposed to know you want days of the week?

Setting Factor Levels

We can explicitly set the `levels`, and their order, when we create the factor

```
dow = c("Mon","Tue","Wed","Thu","Fri","Sat","Sun")
df = mutate(df, days.factor = factor(days, levels = dow))
levels(df$days.factor)
#> [1] "Mon" "Tue" "Wed" "Thu" "Fri" "Sat" "Sun"
```

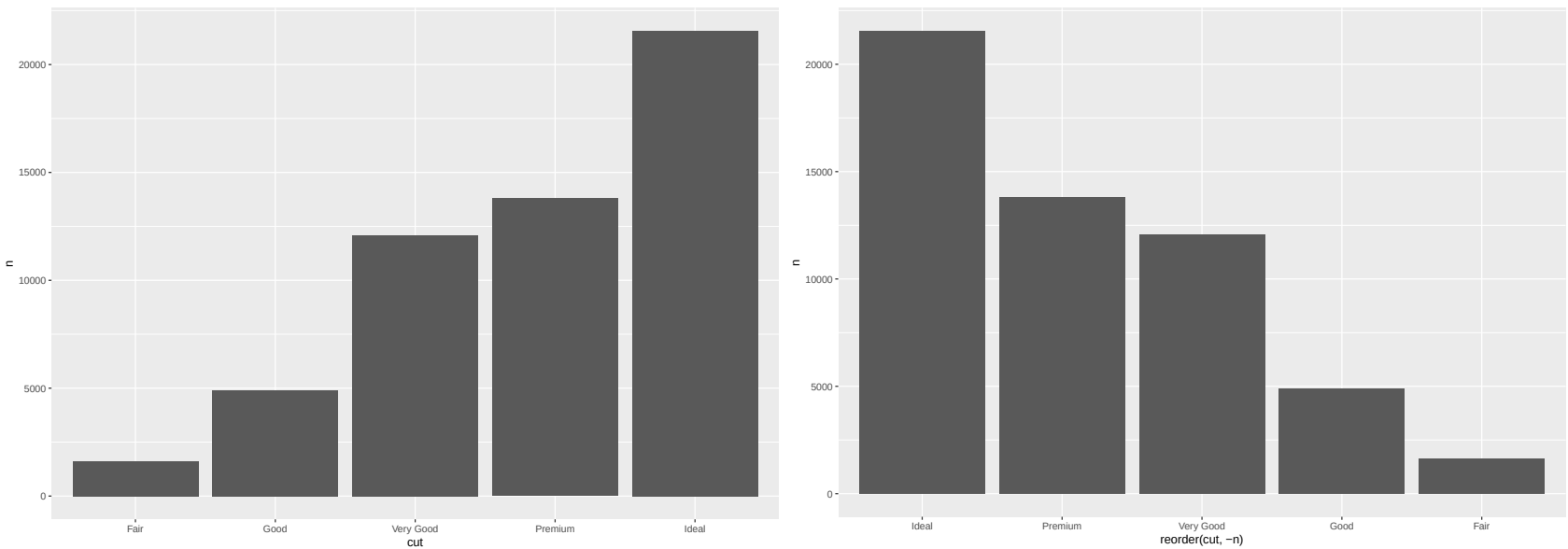
```
ggplot(df, aes(days.factor)) + geom_bar() +
  scale_x_discrete(drop=FALSE)  # this is used to perserve 0 counts
```



Plotting order with `reorder()`

Recall that we can use the `reorder()` function to create a factor with levels ordered by another variables

```
y = count(diamonds, cut)
ggplot(y, aes(cut, n)) + geom_col()
ggplot(y, aes(reorder(cut, -n), n)) + geom_col()
```



More Factors Help

The tidyverse package `forcats` provides many nice conveniences for working with factors.

Missing and Special Data

Missing Data

It is common to find missing elements in data. R uses `NA` ("Not Available") to represent missing data.

```
m = c(1,2,NA,1,5,5,NA)    # NA doesn't change 'type' of vector
class(m)
#> [1] "numeric"
m
#> [1]  1  2 NA  1  5  5 NA
```

Just use the letters `NA` since it is a "reserved" word in R (don't use quotations).

```
m2 = c(1,2,"NA",1,5,5,"NA") # oops, "NA" is a character.
class(m2)
#> [1] "character"
m2
#> [1] "1"  "2"  "NA" "1"  "5"  "5"  "NA"
```

That converts everything to a character!

Special Data

You may run across some other special words:

- `NULL` stands for "nothing"

```
c(1,2,NULL,3,4,NULL,5)
#> [1] 1 2 3 4 5
```

- `NaN` (Not a Number)
- `Inf` and `-Inf` for infinite values

Try these:

```
5/0
0/5
0/0
log(0)
log(-1)
```

Formatting

Formatting numbers

Useful for

- labels in plots
- results in tables
- getting more information on screen

Formatting numbers

The function `round()` will round numeric data.

```
x = c(1.1, 2.22, 3.333, 4.4444)

round(x, digits=2)
#> [1] 1.10 2.22 3.33 4.44
round(x, digits=0)
#> [1] 1 2 3 4
```

Use the `scales` package to convert to percentage, dollars, scientific:

```
library(scales)     # required package
percent(x)
#> [1] "110%" "222%" "333%" "444%"
dollar(x)
#> [1] "$1.10" "$2.22" "$3.33" "$4.44"
scientific(x*100)
#> [1] "1.10e+02" "2.22e+02" "3.33e+02" "4.44e+02"
```

Formatting Numbers

The `paste()` function is helpful for adding general units

```
## paste("\u20ac",x, sep='')  # euros
paste(x, 'cm')
#> [1] "1.1 cm"    "2.22 cm"   "3.333 cm"  "4.4444 cm"
```

And `paste()` can also be used to make a character string using the `collapse=` argument

```
paste(x, collapse=',')
#> [1] "1.1,2.22,3.333,4.4444"
```

Recoding

Recoding Definition

Recoding is the process of changing the values of elements in a vector/column

- converting data types
- replacing NA (missing) with values
- replacing values with NA (missing)
- binning numerical data
- general replacement

Data

We will make use of these data

```
data(flights)   # flights that departed NYC (i.e. JFK, LGA or EWR) in
data(economics) # US economic time series
data(midwest)   # Midwest demographics
# select a few variables from midwest data
mw = select(midwest, PID:popdensity, percollege, inmetro:category)
```

Converting data types

Use the `as.<datatype>()` functions

```
data(midwest)
mw %>% mutate(PID = as.character(PID), inmetro=as.logical(inmetro))
```

Extracing date components

Recall the `format.Date()` to extract date elements

```
economics %>% mutate(day = format(date, '%a'),
                     month=format(date, '%b'),
                     year=format(date, '%Y'))
```

Working with Missing Values

Working with missing values (NA's)

- In R, `NA` represents missing.
- In practice, you should know why data are missing. It can have big implications for your analysis.
- In general, there are two approaches to handling missing values:
  - Remove observations with missing values
  - replace missing values (*statistical imputation*)
  - both of these are dangerous!
  - if you run into this problem, try a few options to examine the *sensitivity* of the results

Removing NA's

Remove observations with missing values using `!is.na()`

```
flights %>% filter(!is.na(arr_delay)) %>% nrow()  # not cancelled flig
#> [1] 327346
```

Ignoring missing values with `na.rm=TRUE` argument

```
mean(flights$arr_delay, na.rm=TRUE)
#> [1] 6.895377
```

Replacing NA's with `coalesce()`

`NA` elements in a vector can be replaced with `coalesce()`

```
(x = c(NA, 1:4, NA))
#> [1] NA  1  2  3  4 NA
coalesce(x, 0L)          # replace NA with 0L (must be integer)
#> [1] 0 1 2 3 4 0
```

`coalesce()` requires the replacement to be the same data type as x!

```
coalesce(as.numeric(x), mean(x, na.rm=TRUE))
#> [1] 2.5 1.0 2.0 3.0 4.0 2.5
```

Replaceing all NA's in a data frame

We can replace all `NA`s in a data frame with the command

```
(df = tibble(x=c(NA, 1:4, NA), y=c('a','b',NA,NA,'c','d' )))
#> # A tibble: 6 × 2
#>       x     y
#>   <int> <chr>
#> 1    NA     a
#> 2     1     b
#> 3     2  <NA>
#> 4     3  <NA>
#> 5     4     c
#> 6    NA     d
df[is.na(df)] = 0
df
#> # A tibble: 6 × 2
#>       x     y
#>   <dbl> <chr>
#> 1     0     a
#> 2     1     b
#> 3     2     0
#> 4     3     0
#> 5     4     c
#> 6     0     d
```

Replace values with NA

Sometimes data imported from other places using something other than `NA` to indicate missing values

```
#- missing values are coded as -999
(x = c(1:4, -999))
#> [1]    1    2    3    4 -999
na_if(x, -999)
#> [1]  1  2  3  4 NA
```

We will discuss better ways of doing this when we cover data import.

Recoding Numerical Data

Simple Binning

The `ifelse(test, yes, no)` function can create simple bins

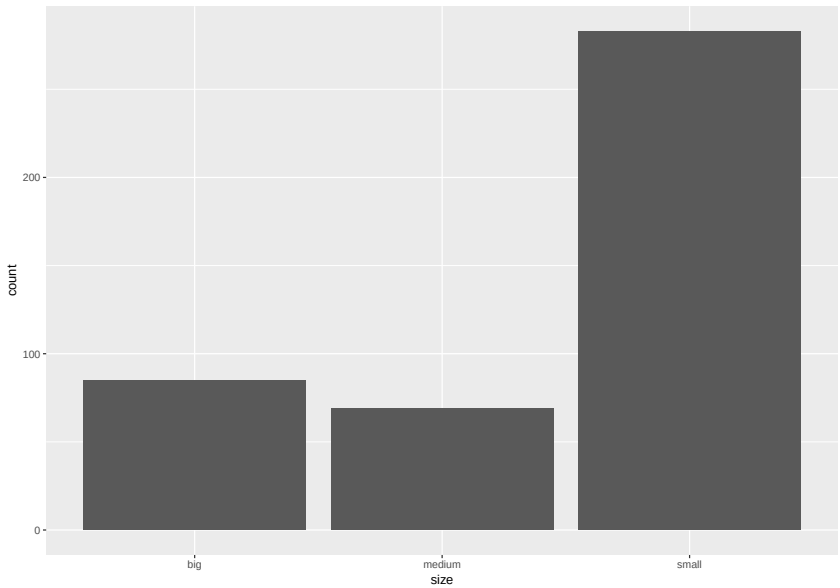
```
#- county is 'big' if population over 100,000
mw %>% transmute(county, poptotal,
                 size = ifelse(poptotal>100000, "big", "small"))
```

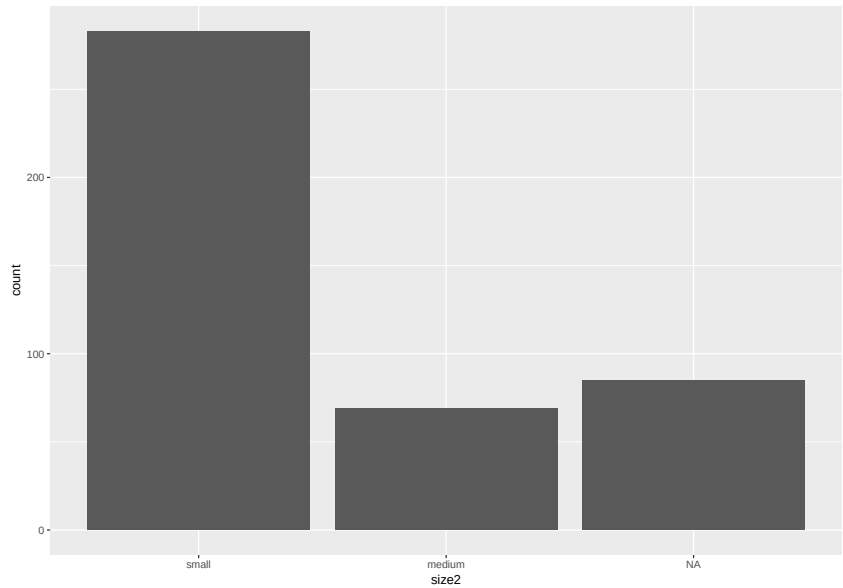
Or nest `ifelse()` to get three levels

```
mw.size = mw %>%
  transmute(county, poptotal,
  size = ifelse(poptotal>100000, "big",
                ifelse(poptotal>50000, "medium", "small")))
count(mw.size, size)
#> # A tibble: 3 × 2
#>     size     n
#>    <chr> <int>
#> 1    big    85
#> 2 medium    69
#> 3  small   283
```

Remember the Factors Levels

```
mw.size = mutate(mw.size,
                 size2=factor(size,
                              levels=c('small', 'medium', 'large')))
ggplot(mw.size, aes(size)) + geom_bar()     # left
ggplot(mw.size, aes(size2)) +geom_bar()     # right
```





Binning Numerical Data

It is *sometimes* useful to convert numerical data to categorical. The numeric data would usually be converted to an *ordered factor*.

- Histograms use bins to count similar valued observations
  - but remember, `geom_density()` isn't sensitive to bin origin
- analysis can be simplified by reducing the complexity of a variable
  - summary tables
- quantile based colors for choropleth maps
- but, you lose data!

Binning

The functions `cut_width()`, `cut_interval()`, and `cut_number()`, are handy to discretize numeric data.

```
x=1:100

#- cut_interval with n makes n groups with equal range
cut_interval(x, n=5)        # 5 groups

#- cut_interval with length makes group
cut_interval(x, length=25) # intervals of width 25

#- cut_width allows more generic width
cut_width(x, width=25, boundary=1) # intervals of width 25

#- cut_number puts about equal observations in each bin
cut_number(x, n=4) # quartiles
```

Quantile Binning

Quantile binning attempts to group an equal number of observations in each bin.

```
x = c(1,1, 2,2, 3,3, 4,4)
cut_number(x, 3)     # creates breaks based on values
#> [1] [1,2] [1,2] [1,2] [1,2] (2,3] (2,3] (3,4] (3,4]
#> Levels: [1,2] (2,3] (3,4]
ntile(x, 3)          # doesn't consider ties
#> [1] 1 1 1 2 2 2 3 3
```

`ntile()` splits data exactly into equal bins, but doesn't respect ties!

The `cut()` function

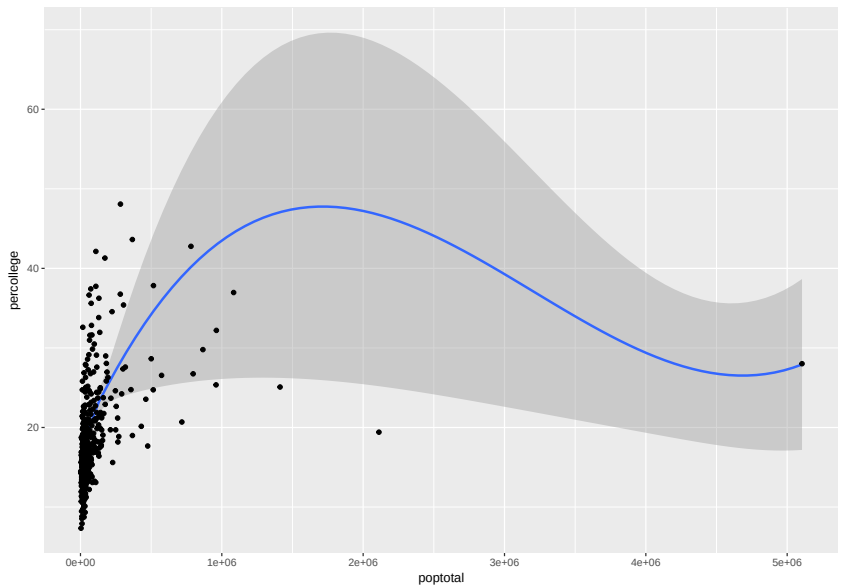
Learn the `cut()` function to have full control

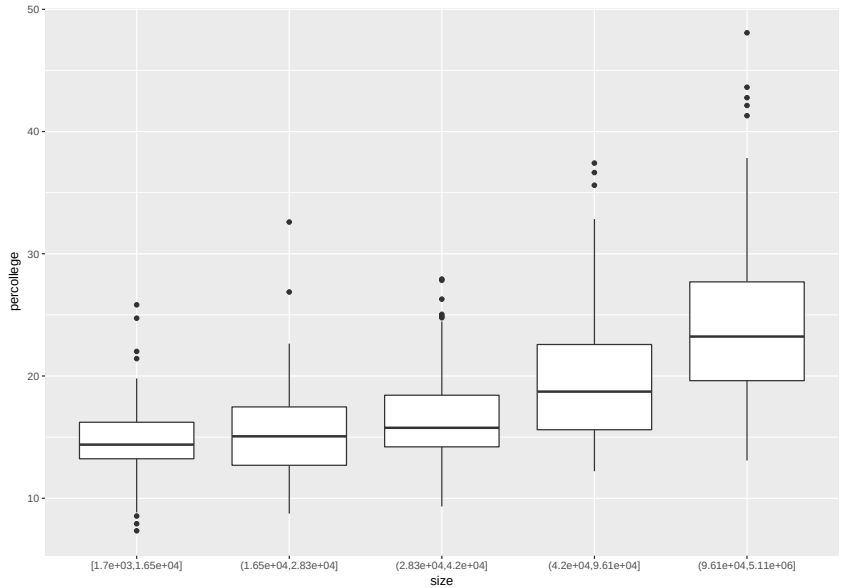
```
bks = c(0, quantile(mw$poptotal, probs=c(.25, .50, .75)), Inf)
mw %>%
  transmute(county, poptotal,
            size = cut(poptotal, breaks=bks,
               labels=c('small', 'med_small', 'med_large', 'large')))
#> # A tibble: 437 × 3
#>       county poptotal      size
#>        <chr>    <int>    <fctr>
#> 1      ADAMS    66090 med_large
#> 2  ALEXANDER    10626     small
#> 3       BOND    14991     small
#> 4      BOONE    30806 med_small
#> 5      BROWN     5836     small
#> 6     BUREAU    35688 med_large
#> 7    CALHOUN     5322     small
#> 8    CARROLL    16805     small
#> 9       CASS    13437     small
#> 10 CHAMPAIGN   173025     large
#> # ... with 427 more rows
```

Using binning

```
ggplot(mw, aes(poptotal, percollege)) +
  geom_smooth() + geom_point()
#> `geom_smooth()` using method = 'loess'

mw %>% mutate(size=cut_number(poptotal, 5)) %>%
  ggplot(aes(size, percollege)) + geom_boxplot()
```





Using binning

```
mw %>% mutate(size=cut_number(poptotal, 5)) %>%
  group_by(size) %>% summarize(n=n(),
                               mean=mean(percollege),
                               sd=sd(percollege))
#> # A tibble: 5 × 4
#>                    size     n     mean       sd
#>                  <fctr> <int>    <dbl>    <dbl>
#> 1  [1.7e+03,1.65e+04]    88 14.70660 3.179170
#> 2 (1.65e+04,2.83e+04]    87 15.47246 3.924847
#> 3  (2.83e+04,4.2e+04]    87 16.59473 3.863858
#> 4  (4.2e+04,9.61e+04]    87 19.82750 5.867889
#> 5 (9.61e+04,5.11e+06]    88 24.72916 7.367284
```

General Recoding

Merging Factor Levels

Factors with many levels can be annoying. Sometimes we can get away with *lumping* together the infrequent values into an "other" category

```
count(mw, category, sort=TRUE)  # has 16 categories
```

```
library(forcats)
mutate(mw, cat2=fct_lump(category,prop=.05)) %>% count(cat2)
```

Or, keep/collapse certain levels

```
#- keep levels that begin with A
library(stringr)
levs = unique(mw$category)
Alevs = levs[str_sub(levs, 1, 1) == 'A']
mutate(mw, catA=fct_other(category, keep=Alevs)) %>% count(catA)
```

Recode from Lookup Tables

We can used a *named* vector to match new values

```
x = c("DL","DL","AA", "DL", "AA")
lut = c("AA"="American", "DL"="Delta")
lut[x]
#>         DL         DL         AA         DL         AA
#>    "Delta"    "Delta" "American"    "Delta" "American"
```

```
#- An alternative way to get lut:
lut=c("American", "Delta"); names(lut) = c("AA", "DL")
```

Explicitly enter old/new pairs with `recode()`

```
recode(x, DL="Delta", AA="American")
#> [1] "Delta"    "Delta"    "American" "Delta"    "American"
```

Use `match()` for vectors (or data frame) with old/new values

```
old = c('DL', 'AA')
new = c('Delta', 'American')
new[match(x, old)]
#> [1] "Delta"    "Delta"    "American" "Delta"    "American"
```

Using joins

We will get to joins next week, but here is a preview of how this can be used for recoding (and more).

Check out the wikipedia page on airline codes https://en.wikipedia.org/wiki/List_of_airline_codes.

```
# read in the table
library(rvest)
url = "https://en.wikipedia.org/wiki/List_of_airline_codes"
tab = url %>% read_html() %>%
  html_node("table.wikitable") %>%
  html_table(fill=TRUE) %>%
  select(-NA)                              # remove extra column
```

```
#- Join the table with x values
tibble(x) %>%                              # make x into tibble
  left_join(tab, by=c("x"="IATA"))         # join with tab
#> Error in x[needs_ticks] <- paste0("`", gsub("`", "\\\\`", x[needs_t
```

Functions

Function Help

We have already used many built-in R functions. E.g.,

```
seq(0, 1, by=0.25)
#> [1] 0.00 0.25 0.50 0.75 1.00
```

Let's examine this function in more detail. To get the help page,

```
?seq
```

(or you could type: `help(seq)`)

If you don't know the function name, try

```
??sequence
help.search("sequence")  # this is same as ??sequence
```

or web search.

Function Defaults

Some functions have default values that will be used unless you specify an alternative.

Notice the text

```
## Default S3 method:
seq(from = 1, to = 1, by = ((to - from)/(length.out
    length.out = NULL, along.with = NULL, ...)
```

It tells you that the default value of `from=1` and `to=1`, etc. So if I pass in no arguments, the function will return the vector `1:1`

```
seq()
#> [1] 1
```

Function Defaults

If you specify some of the arguments, the others stay at their default values:

```
seq(to=8)
#> [1] 1 2 3 4 5 6 7 8
seq(to=2, by=0.25)
#> [1] 1.00 1.25 1.50 1.75 2.00
seq(from=0, by=0.10)
#>  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

From help page:

```
## Default S3 method:
seq(from = 1, to = 1, by = ((to - from)/(length.out - 1)),
    length.out = NULL, along.with = NULL, ...)
```

Function Arguments: by name

It is probably best practice to assign arguments by name:

```
seq(from=0, to=1, length.out=11)
#>  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
seq(from=0, to=1, by=0.1)
#>  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

- But it is common for first two arguments to be assigned by position only.

Notice that sometimes only one argument can be used at a time:

```
seq(from=0, to=1, by=0.1, length.out=11)
#> Error in seq.default(from = 0, to = 1, by = 0.1, length.out = 11):
```

Function Arguments: by name

Names can be determined by partial string match:

```
seq(from=0, to=1, length.out=11)  # full name
#>  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
seq(from=0, to=1, length=11)      # partial name
#>  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
seq(from=0, to=1, len=11)         # partial name
#>  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

And named in any order

```
seq(len=11, to=1, from=0)
#>  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

Function Arguments: by position

For the function `seq()`, the first argument is `from`, the second is `to`, etc.

Instead of writing the names, we can just enter the values in the correct order

```
seq(0, 1, 0.1)        # seq(from=0, to=1, by=0.1)
#>  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

Function Arguments: dot-dot-dot

Sometimes you will see an ellipsis (. . .) as a function argument.

This means that the function may call *another* function which can get additional arguments.

We won't be too concerned with this, but here is a good example:

```
?plot
plot(1:10, 1:10, typ='l')
plot(1:10, 1:10, typ='p',
     xlab="The x-axis label",
     ylab="The y-axis label",
     main="plot title here")
```

Your Turn #1 : Functions

Use the following data to answer the questions

```
scores = c(98,100,NA,78,92,88,NA,NA,91,89,97,88,99)
```

1. What is the mean of `scores`? (Hint: `?mean` and argument `na.rm`)
2. What is trimmed mean of `scores`? Trim 20% of values.
3. What is the median of `scores`?
4. Find a function to sort `scores` from smallest to largest.
5. What does the function `summary` do?
   `summary(scores)`
6. What does this do: `quantile(scores, probs=c(0,.1,.5,.99), na.rm=TRUE)`
7. Compare `min(scores, na.rm=TRUE)` to `pmin(scores, 90)`