R Fundamentals Part 2: Subsetting and reshaping

Shinhye Choi, Rochelle Terman, Evan Muzzall, Dillon Niederhut

October 23, 2016

Table of Contents

Learning objectives

1. Day 1 review
2. Loading data from files
3. Subsetting in base R
4. Missing data (NA)
5. Merging data
6. Subsetting with the dplyr R package
7. Tidying/reshaping data with the tidyr R package

1. Day 1 review

1. Set your working directory
2. The assignment operator `<-`
3. Atomic data types: numeric, character, logical
4. Data structures: vector, list, matrix, data frame
5. Save your work: `write.csv()` and `sink()`

2. Loading data from files

Set your working directory

```
getwd()

## [1] "/Users/E/Desktop/R-Fundamentals-master"

setwd("/Users/E/Desktop/R-Fundamentals-master")
```

Install the VIM package and retrieve it into your R instance

```
install.packages("VIM", dependencies=TRUE)
library(VIM)
```

Load the sleep dataset from the VIM package. This dataset looks at sleep patterns in various species of mammals and contains missing (`NA`) values. See the link below for the original research article.

Allison and Chichetti 1976

NOTE: you DO NOT want to load the Student's Sleep Dataset from the "datasets" library. You should have 62 rows and 10 columns.

```
?sleep # Click the "Mammal sleep data" link. You DO NOT want "Student's
Sleep Data".

library(VIM)

## Warning: package 'VIM' was built under R version 3.2.5

## Loading required package: colorspace

## Loading required package: grid

## Loading required package: data.table

## VIM is ready to use.
##  Since version 4.0.0 the GUI is in its own package VIMGUI.
##
##           Please use the package to use the new (and old) GUI.

## Suggestions and bug-reports can be submitted at:
https://github.com/alexkowa/VIM/issues

##
## Attaching package: 'VIM'

## The following object is masked from 'package:datasets':
##
##     sleep

data(sleep)
dim(sleep) # 62 rows, 10 columns

## [1] 62 10

head(sleep)

##    BodyWgt BrainWgt NonD Dream Sleep Span Gest Pred Exp Danger
## 1 6654.000   5712.0   NA    NA   3.3 38.6  645    3   5      3
## 2    1.000      6.6  6.3   2.0   8.3  4.5   42    3   1      3
## 3    3.385     44.5   NA    NA  12.5 14.0   60    1   1      1
## 4    0.920      5.7   NA    NA  16.5   NA   25    5   2      3
## 5 2547.000   4603.0  2.1   1.8   3.9 69.0  624    3   5      4
## 6   10.550    179.5  9.1   0.7   9.8 27.0  180    4   4      4

str(sleep)

## 'data.frame':    62 obs. of  10 variables:
##  $ BodyWgt : num  6654 1 3.38 0.92 2547 ...
##  $ BrainWgt: num  5712 6.6 44.5 5.7 4603 ...
##  $ NonD    : num  NA 6.3 NA NA 2.1 9.1 15.8 5.2 10.9 8.3 ...
##  $ Dream   : num  NA 2 NA NA 1.8 0.7 3.9 1 3.6 1.4 ...
##  $ Sleep   : num  3.3 8.3 12.5 16.5 3.9 9.8 19.7 6.2 14.5 9.7 ...
##  $ Span    : num  38.6 4.5 14 NA 69 27 19 30.4 28 50 ...
##  $ Gest    : num  645 42 60 25 624 180 35 392 63 230 ...
##  $ Pred    : int  3 3 1 5 3 4 1 4 1 1 ...
##  $ Exp     : int  5 1 1 2 5 4 1 5 2 1 ...
##  $ Danger  : int  3 3 1 3 4 4 1 4 1 1 ...
```

We can save this dataframe to a .CSV cile with `write.csv()`. It will save to our working directory:

```
?write.csv
?read.csv

write.csv(sleep, "sleep_VIM.csv", row.names=FALSE)
```

We can load it from the file in our working directory via the `read.csv()` command:

```
sleep <- read.csv("/Users/E/Desktop/R-Fundamentals/sleep_VIM.csv",
header=TRUE, stringsAsFactors=FALSE)
```

Notice that `stringsAsFactors=FALSE`. If set to `TRUE`, R will try to guess which character data vectors should automatically be converted to factors. This is problematic because 1) R is not always good at guessing and 2) R defaults to alphabetical factor level sorting. This might not matter for your data, but we recommend to set `stringsAsFactors=FALSE` and manually convert your desired character vectors to factors. Refer back to the end of Part 1 for these instructions.

When dealing with Microsoft Excel files (.XLSX), you might find it more convenient to save them first as .CSV files in Excel and then import them using `read.csv()`.

Other functions also work to import data from files, such as `load()`. You might also have success with the "xlsx" R package and its `read.xlsx()` command for directly importing Excel files.

Also, the "foreign" R package has commands for loading data from SAS, SPSS, Stata, etc.

2. Loading data from files/ inspecting the data frame

Remember from Part 1 that we can learn a lot about data in R. For dataframes, the following commands are common:

```
str(sleep)     # returns the structure of the dataframe
dim(sleep)     # dataframe dimensions
rownames(sleep)    #row names (they have not been named and default to
character type)
nrow(sleep)    # number of rows
ncol(sleep)    # number of columns
unique(sleep)    # show rows with unique data
```

`names()` and `colnames()` both return column names of the data frame:

```
names(sleep)
```

```
##  [1] "BodyWgt"   "BrainWgt" "NonD"      "Dream"     "Sleep"     "Span"
"Gest"      "Pred"      "Exp"       "Danger"

colnames(sleep)

##  [1] "BodyWgt"   "BrainWgt" "NonD"      "Dream"     "Sleep"     "Span"
"Gest"      "Pred"      "Exp"       "Danger"
```

We can also check which indices are true. Let's convert the "Span" column to a logical vector where missing data is coded as `NA` while present data is marked `TRUE`.

```
?as.logical

new_Span <- as.logical(sleep$Span)
new_Span

##  [1] TRUE TRUE TRUE   NA TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
NA TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
## [24] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE   NA
NA TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
## [47] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
TRUE TRUE TRUE TRUE
```

Now we can see which data are missing (`NA`) and which cells have data present (`TRUE`).

`which()` will return the rows numbers that have data present:

```
?which

which(new_Span)

##  [1]  1  2  3  5  6  7  8  9 10 11 12 14 15 16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31 32 33 34 37 38 39 40 41 42
## [39] 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62
```

3. Subsetting in base R

Efficiently subsetting your data will save you time and stress. Fortunately, there are several different ways to subset data in base R.

3. Subsetting in base R/ `$`

Remember from Part 1 that the dollar sign operator `$` will extract only a single vector/column within the data frame:

```
?"$" # Remember that we must wrap symbols in quotation marks to view
their help pages

sleep$Dream #Returns only the "Dream" vector from the `sleep` data
frame.
```

```
##  [1]  NA 2.0  NA  NA 1.8 0.7 3.9 1.0 3.6 1.4 1.5 0.7 2.7  NA 2.1 0.0
4.1 1.2 1.3 6.1 0.3 0.5 3.4  NA 1.5  NA 3.4 0.8 0.8
## [30]  NA  NA 1.4 2.0 1.9 2.4 2.8 1.3 2.0 5.6 3.1 1.0 1.8 0.9 1.8 1.9
0.9  NA 2.6 2.4 1.2 0.9 0.5  NA 0.6  NA 2.2 2.3 0.5
## [59] 2.6 0.6 6.6  NA
```

However, you might find subsetting using ***bracket notation*** `[ , ]` along with variable names, positive and negative integers, and/or logical values is easier because you can subset multiple elements at once.

3. Subsetting in base R/ `[,c(Variable Names)]`

You can subset your data by specifying variable names within bracket notation and using the `c()` command to create a column name character vector of names you want to keep.

We can create a new dataframe object `sleep_varnames` that includes only "BodyWgt" and "BrainWgt" variable names from the `sleep` data frame:

```
?"["

sleep_varnames <- sleep[,c("BodyWgt", "BrainWgt")]
str(sleep_varnames)

## 'data.frame':    62 obs. of  2 variables:
##  $ BodyWgt : num  6654 1 3.38 0.92 2547 ...
##  $ BrainWgt: num  5712 6.6 44.5 5.7 4603 ...

head(sleep_varnames)

##    BodyWgt BrainWgt
## 1 6654.000   5712.0
## 2    1.000      6.6
## 3    3.385     44.5
## 4    0.920      5.7
## 5 2547.000   4603.0
## 6   10.550    179.5
```

Notice that the comma is still included within the bracket notation before the vector of column names. This indicates that we want ALL of the rows corresponding to these two columns. This is the same when we only want to subset rows and include ALL columns (see below).

3. Subsetting in base R/ two-dimensional subsetting `[c(x:y), c(x:y)]`

When you subset your data in two dimensions, you subset both the rows and columns.

Remember that in bracket notation `[ , ]` everything ***before*** the comma refers to rows, and everything ***after*** the comma refers to columns!

3. Subsetting in base R/ logical tests

We can also use logical tests to subset our data. For example, what if we want to include only the rows that have a value of 1 for "Exp"? We can use the relational operator ==:

```
?"=="

sleep_logical <- sleep[sleep$Exp == 1,]
sleep_logical

##    BodyWgt BrainWgt NonD Dream Sleep  Span Gest Pred Exp Danger
## 2    1.000     6.60  6.3   2.0   8.3   4.5   42    3   1      3
## 3    3.385    44.50   NA    NA  12.5  14.0   60    1   1      1
## 7    0.023     0.30 15.8   3.9  19.7  19.0   35    1   1      1
## 10  52.160   440.00  8.3   1.4   9.7  50.0  230    1   1      1
## 13   0.550     2.40  7.6   2.7  10.3    NA   NA    2   1      2
## 15   0.075     1.20  6.3   2.1   8.4   3.5   42    1   1      1
## 20  60.000    81.00 12.0   6.1  18.1   7.0   NA    1   1      1
## 23   0.120     1.00 11.0   3.4  14.4   3.9   16    3   1      2
## 26  36.330   119.50   NA    NA  13.0  16.2   63    1   1      1
## 27   0.101     4.00 10.4   3.4  13.8   9.0   28    5   1      3
## 30 100.000   157.00   NA    NA  10.8  22.4  100    1   1      1
## 33   0.010     0.25 17.9   2.0  19.9  24.0   50    1   1      1
## 34  62.000  1320.00  6.1   1.9   8.0 100.0  267    1   1      1
## 35   0.122     3.00  8.2   2.4  10.6    NA   30    2   1      1
## 36   1.350     8.10  8.4   2.8  11.2    NA   45    3   1      3
## 37   0.023     0.40 11.9   1.3  13.2   3.2   19    4   1      3
## 38   0.048     0.33 10.8   2.0  12.8   2.0   30    4   1      3
## 39   1.700     6.30 13.8   5.6  19.4   5.0   12    2   1      1
## 40   3.500    10.80 14.3   3.1  17.4   6.5  120    2   1      1
## 44   1.620    11.40 11.9   1.8  13.7  13.0   17    2   1      2
## 48   0.280     1.90 10.6   2.6  13.2   4.7   21    3   1      3
## 49   4.235    50.40  7.4   2.4   9.8   9.8   52    1   1      1
## 56   0.060     1.00  8.1   2.2  10.3   3.5   NA    3   1      2
## 57   0.900     2.60 11.0   2.3  13.3   4.5   60    2   1      2
## 58   2.000    12.30  4.9   0.5   5.4   7.5  200    3   1      3
## 61   3.500     3.90 12.8   6.6  19.4   3.0   14    2   1      1
## 62   4.050    17.00   NA    NA    NA  13.0   38    3   1      1
```

Only rows with values of 1 for "Exp" are returned!

What if we want to subset the data so it includes only the rows that have a value of 1 ***and*** 3 for Exp and 2 for "Danger"? We would use the "and" & logical operator:

```
?"&"

sleep_logical2 <- sleep[sleep$Exp == c(1,3) & sleep$Danger == 2,]
sleep_logical2
```

```
##    BodyWgt BrainWgt NonD Dream Sleep Span Gest Pred Exp Danger
## 13    0.55      2.4  7.6   2.7  10.3   NA   NA    2   1      2
## 23    0.12      1.0 11.0   3.4  14.4  3.9   16    3   1      2
## 50    6.80    179.0  8.4   1.2   9.6 29.0  164    2   3      2
## 57    0.90      2.6 11.0   2.3  13.3  4.5   60    2   1      2
```

We can subset a dataframe that includes only rows with "Exp" = 2 ***and*** 3 and only "BodyWgt", "BrainWgt", and "Exp" columns.

```
str(sleep)

sleep_2d <- sleep[sleep$Exp == c(2,3), c("BodyWgt", "BrainWgt", "Exp")]
sleep_2d
```

```
##    BodyWgt BrainWgt Exp
## 9    3.300     25.6   2
## 17   0.785      3.5   2
## 19   1.410     17.5   2
## 28   1.040      5.5   3
## 47   4.288     39.2   2
## 50   6.800    179.0   3
## 51   0.750     12.3   2
## 55   1.400     12.5   2
## 59   0.104      2.5   2
## 60   4.190     58.0   3
```

NOTE: see how we have the column names listed after the comma? When we only want to call rows, we still must include the comma to tell R that we want ALL of the columns! For example, if we want to specify only rows that have values of 2 and 3 for "Exp" and want all of the columns, we still must include the column inside our bracket notation after the specified rows:

```
sleep_rows_only <- sleep[sleep$Exp == c(2,3),]
sleep_rows_only
```

```
##    BodyWgt BrainWgt NonD Dream Sleep Span Gest Pred Exp Danger
## 9    3.300     25.6 10.9   3.6  14.5 28.0   63    1   2      1
## 17   0.785      3.5  6.6   4.1  10.7  6.0   42    2   2      2
## 19   1.410     17.5  4.8   1.3   6.1 34.0   NA    1   2      1
## 28   1.040      5.5  7.4   0.8   8.2  7.6   68    5   3      4
## 47   4.288     39.2   NA    NA  12.5 13.7   63    2   2      2
## 50   6.800    179.0  8.4   1.2   9.6 29.0  164    2   3      2
## 51   0.750     12.3  5.7   0.9   6.6  7.0  225    2   2      2
## 55   1.400     12.5   NA    NA  11.0 12.7   90    2   2      2
## 59   0.104      2.5 13.2   2.6  15.8  2.3   46    3   2      2
## 60   4.190     58.0  9.7   0.6  10.3 24.0  210    4   3      4
```

3. Subsetting in base R/ subsetting with positive integers `[,c(x:y)]`

Subsetting by ***positive*** integers works as well. This will ***include*** only the column numbers specified, rather than typing out their names.

Let's create an object `sleep_posint` that includes only "NonD", "Sleep", and "Exp" columns.

First use `str()` to see which integer values these columns represent. Because we only want the 3rd, 5th, and 9th columns, we type:

```
str(sleep)
sleep_posint <- sleep[,c(3,5,9)]
str(sleep_posint)
head(sleep_posint)
```

3. Subsetting in base R/ subsetting with negative integers `[,-c(x:y)]`

Subsetting by ***negative*** integers will ***exclude*** the specified columns. Notice the - symbol before `c()` inside our bracket notation.

We can create an object called `sleep_negint` that includes everything ***except*** columns 1 and 2 ("BodyWgt" and "BrainWgt"):

```
str(sleep)

## 'data.frame':    62 obs. of  10 variables:
##  $ BodyWgt : num  6654 1 3.38 0.92 2547 ...
##  $ BrainWgt: num  5712 6.6 44.5 5.7 4603 ...
##  $ NonD    : num  NA 6.3 NA NA 2.1 9.1 15.8 5.2 10.9 8.3 ...
##  $ Dream   : num  NA 2 NA NA 1.8 0.7 3.9 1 3.6 1.4 ...
##  $ Sleep   : num  3.3 8.3 12.5 16.5 3.9 9.8 19.7 6.2 14.5 9.7 ...
##  $ Span    : num  38.6 4.5 14 NA 69 27 19 30.4 28 50 ...
##  $ Gest    : num  645 42 60 25 624 180 35 392 63 230 ...
##  $ Pred    : int  3 3 1 5 3 4 1 4 1 1 ...
##  $ Exp     : int  5 1 1 2 5 4 1 5 2 1 ...
##  $ Danger  : int  3 3 1 3 4 4 1 4 1 1 ...

sleep_negint <- sleep[,-c(1,2)]
str(sleep_negint)

## 'data.frame':    62 obs. of  8 variables:
##  $ NonD  : num  NA 6.3 NA NA 2.1 9.1 15.8 5.2 10.9 8.3 ...
##  $ Dream : num  NA 2 NA NA 1.8 0.7 3.9 1 3.6 1.4 ...
##  $ Sleep : num  3.3 8.3 12.5 16.5 3.9 9.8 19.7 6.2 14.5 9.7 ...
##  $ Span  : num  38.6 4.5 14 NA 69 27 19 30.4 28 50 ...
##  $ Gest  : num  645 42 60 25 624 180 35 392 63 230 ...
##  $ Pred  : int  3 3 1 5 3 4 1 4 1 1 ...
##  $ Exp   : int  5 1 1 2 5 4 1 5 2 1 ...
##  $ Danger: int  3 3 1 3 4 4 1 4 1 1 ...

head(sleep_negint)

##   NonD Dream Sleep Span Gest Pred Exp Danger
## 1   NA    NA   3.3 38.6  645    3   5      3
## 2  6.3    2.0    8.3  4.5    42     3    1       3
## 3   NA     NA   12.5 14.0    60     1    1       1
## 4   NA     NA   16.5   NA    25     5    2       3
## 5  2.1    1.8    3.9 69.0   624     3    5       4
## 6  9.1    0.7    9.8 27.0   180     4    4       4
```

3. Subsetting in base R/ lists and double bracket `[[]]` notation

You can also subset lists.

```
?"[["
```

Create an examlpe list:

```
example_list <- list(TRUE, "string data", 5)
example_list

## [[1]]
## [1] TRUE
##
## [[2]]
## [1] "string data"
##
## [[3]]
## [1] 5
```

Single brackets `[]` will return the list container as well as its value:

```
example_list[1]

## [[1]]
## [1] TRUE
```

However, double brackets will return only the value:

```
example_list[[1]]

## [1] TRUE
```

Challenge 1

1. How many ways can you subset the `iris` dataset using column names and positive and negative integers? Type and run `data(iris)` to load the dataset.

4. Missing data (NA)

Identifying missing data can be important for subsetting purposes. R codes missing values as `NA`. Identifying missing data is important because dealing with it might be necessary to run basic tests like `mean()`

```
?NA

mean(sleep$NonD)    # This returns NA because R is unsure how to deal
with NA cells for the `mean()` computation.

## [1] NA
```

However, we can use `na.rm = TRUE` to properly calculate the mean of the NonD column by now excluding the NAs.

```
?mean # Scroll down to `na.rm`

mean(sleep$NonD, na.rm=TRUE) #Now `mean()` returns the mean!

## [1] 8.672917
```

While `na.rm()` nor `str()` will not tell us which data are missing in a convenient way, `is.na()` does. Wrap the name of your data frame in `is.na()` to return logical values. Missing data is coded as `TRUE`, while present data are coded as `FALSE`

```
?is.na
is.na(sleep)
```

Data are coded as missing in many different ways besides `NA`, so don't be surprised if you see some other signifier.

4. Missing data (NA)/ recoding missing data

Let's recode NA values in place to say "NONE":

```
sleep[is.na(sleep)] <- "NONE"

sleep
```

They now say "NONE".

However, for R to handle them correctly, we want to recode them to say `NA`. We can do this with a combination of the name of our data set, bracket notation, our relational operator `==` and our old friend the assignment operator `<-` !

```
sleep[sleep == "NONE"] <- NA

sleep
```

NOTE: here <NA> and NA are synonymous and R will treat them both as missing. <NA> with less than/greater than symbols is handy because it will let you know which values you have manually recoded to missing.

We can also subset only rows without any missing data using bracket notation. `complete.cases()` will find rows with no missing values.

```
?complete.cases
```

```r
sleep_no_NA <- sleep[complete.cases(sleep),]
```

Remember to include the comma here to tell R you want ALL of the columns for these rows :)

```r
sleep_no_NA
```

Then, test it to see if it contains missing values. All cells are `FALSE`

```r
is.na(sleep_no_NA)

# Conversely, we can subset the sleep data to include only rows with
missing data by adding the logical bash operator `!` (not).
?"!"

sleep_NA <- sleep[!complete.cases(sleep),]

sleep_NA # All rows have at least one cell with missing data
is.na(sleep_NA) # Now we see TRUE values where data is missing
```

Challenge 2

1. How many different ways can you subset the `sleep` dataset using logical tests for `NA` data?

5. Merging data

Merging data is useful when we want to combine two different dataframes that share a vector/column.

We will now create a new data frame called `sleep_ratios` in which we will compute ***three*** ratios from data in `sleep`:

1) Body to Brain weight ratio ("*Body_Brain*")

2) Body Weight to Gestation Period ratio ("*Body_Gest*")

3) Brain Weight to Gestation Period ratio ("*Brain_Gest*")

First, we will create and subset our new object `sleep_ratios` to contain the "BodyWgt" and "BrainWgt" columns from the `sleep` data frame.

```r
sleep_ratios <- sleep[,c("BodyWgt", "BrainWgt")]
str(sleep_ratios) # This data frame only contains "BodyWgt" and
"BrainWgt"

## 'data.frame':    62 obs. of  2 variables:
##  $ BodyWgt : num  6654 1 3.38 0.92 2547 ...
##  $ BrainWgt: num  5712 6.6 44.5 5.7 4603 ...
```

Then, we will add to `sleep_ratios` three columns that contain the computations.

1) Add the *Body_Brain* ratio:

```
sleep_ratios$Body_Brain <- sleep$BodyWgt/sleep$BrainWgt
head(sleep_ratios)

##    BodyWgt BrainWgt Body_Brain
## 1 6654.000   5712.0 1.16491597
## 2    1.000      6.6 0.15151515
## 3    3.385     44.5 0.07606742
## 4    0.920      5.7 0.16140351
## 5 2547.000   4603.0 0.55333478
## 6   10.550    179.5 0.05877437

str(sleep_ratios)

## 'data.frame':    62 obs. of  3 variables:
##  $ BodyWgt   : num  6654 1 3.38 0.92 2547 ...
##  $ BrainWgt  : num  5712 6.6 44.5 5.7 4603 ...
##  $ Body_Brain: num  1.1649 0.1515 0.0761 0.1614 0.5533 ...
```

2) Add the *Body_Gest* ratio:

```
sleep_ratios$Body_Gest <- sleep$BodyWgt/as.numeric(sleep$Gest) # note
that to perform division here, we coerce "Gest" to `as.numeric()` type
head(sleep_ratios)

##    BodyWgt BrainWgt Body_Brain   Body_Gest
## 1 6654.000   5712.0 1.16491597 10.31627907
## 2    1.000      6.6 0.15151515  0.02380952
## 3    3.385     44.5 0.07606742  0.05641667
## 4    0.920      5.7 0.16140351  0.03680000
## 5 2547.000   4603.0 0.55333478  4.08173077
## 6   10.550    179.5 0.05877437  0.05861111

str(sleep_ratios)

## 'data.frame':    62 obs. of  4 variables:
##  $ BodyWgt   : num  6654 1 3.38 0.92 2547 ...
##  $ BrainWgt  : num  5712 6.6 44.5 5.7 4603 ...
##  $ Body_Brain: num  1.1649 0.1515 0.0761 0.1614 0.5533 ...
##  $ Body_Gest : num  10.3163 0.0238 0.0564 0.0368 4.0817 ...
```

3) Add the *Brain_Gest* ratio:

```
sleep_ratios$Brain_Gest <- sleep$BrainWgt/as.numeric(sleep$Gest) # note
that to perform division here, we parse "Gest" `as.numeric()` type
head(sleep_ratios)

##    BodyWgt BrainWgt Body_Brain   Body_Gest Brain_Gest
## 1 6654.000   5712.0 1.16491597 10.31627907  8.8558140
## 2    1.000      6.6 0.15151515  0.02380952  0.1571429
## 3    3.385     44.5 0.07606742  0.05641667  0.7416667
## 4    0.920      5.7 0.16140351  0.03680000  0.2280000
## 5 2547.000   4603.0 0.55333478  4.08173077  7.3766026
## 6   10.550    179.5 0.05877437  0.05861111  0.9972222
```

```
str(sleep_ratios)
```

```
## 'data.frame':    62 obs. of  5 variables:
##  $ BodyWgt   : num  6654 1 3.38 0.92 2547 ...
##  $ BrainWgt  : num  5712 6.6 44.5 5.7 4603 ...
##  $ Body_Brain: num  1.1649 0.1515 0.0761 0.1614 0.5533 ...
##  $ Body_Gest : num  10.3163 0.0238 0.0564 0.0368 4.0817 ...
##  $ Brain_Gest: num  8.856 0.157 0.742 0.228 7.377 ...
```

Finally, we can merge these new columns to our `sleep` data frame by matching the "BodyWgt" and "BrainWgt" columns with the ones from the `sleep_ratios` data frame via the `merge()` function:

```
?merge #Click the "Merge two data frames" link
```

The first two arguments in `merge()` are the names of the two data frames, followed by `by` where we tell which column names we want to match:

```
sleep_and_sleep_ratios <- merge(sleep, sleep_ratios, by=c("BodyWgt",
"BrainWgt"))
head(sleep_and_sleep_ratios)
```

```
##   BodyWgt BrainWgt NonD Dream Sleep Span Gest Pred Exp Danger
Body_Brain     Body_Gest   Brain_Gest
## 1   0.005     0.14  7.7   1.4   9.1  2.6 21.5    5   2      4
0.03571429 0.0002325581 0.006511628
## 2   0.010     0.25 17.9     2  19.9   24   50    1   1      1
0.04000000 0.0002000000 0.005000000
## 3   0.023     0.30 15.8   3.9  19.7   19   35    1   1      1
0.07666667 0.0006571429 0.008571429
## 4   0.023     0.40 11.9   1.3  13.2  3.2   19    4   1      3
0.05750000 0.0012105263 0.021052632
## 5   0.048     0.33 10.8     2  12.8    2   30    4   1      3
0.14545455 0.0016000000 0.011000000
## 6   0.060     1.00  8.1   2.2  10.3  3.5 <NA>    3   1      2
0.06000000           NA          NA
```

```
str(sleep_and_sleep_ratios)
```

```
## 'data.frame':    62 obs. of  13 variables:
##  $ BodyWgt   : num  0.005 0.01 0.023 0.023 0.048 0.06 0.075 0.101
0.104 0.12 ...
##  $ BrainWgt  : num  0.14 0.25 0.3 0.4 0.33 1 1.2 4 2.5 1 ...
##  $ NonD      : chr  "7.7" "17.9" "15.8" "11.9" ...
##  $ Dream     : chr  "1.4" "2" "3.9" "1.3" ...
##  $ Sleep     : chr  "9.1" "19.9" "19.7" "13.2" ...
##  $ Span      : chr  "2.6" "24" "19" "3.2" ...
##  $ Gest      : chr  "21.5" "50" "35" "19" ...
##  $ Pred      : int  5 1 1 4 4 3 1 5 3 3 ...
##  $ Exp       : int  2 1 1 1 1 1 1 1 2 1 ...
##  $ Danger    : int  4 1 1 3 3 2 1 3 2 2 ...
##  $ Body_Brain: num  0.0357 0.04 0.0767 0.0575 0.1455 ...
##  $ Body_Gest : num  0.000233 0.0002 0.000657 0.001211 0.0016 ...
##  $ Brain_Gest: num  0.00651 0.005 0.00857 0.02105 0.011 ...

#What happened here?
```

5. Merging data/ `cbind()` and `rbind()`

Other useful functions include `cbind()` and `rbind()`.

`cbind()` will bind two data frames by their columns and will simply add all of the columns in the `sleep_ratios` data frame to the end of the `sleep` data frame.

```
?cbind
?rbind # Click the "Combine R Objects by Rows or Columns" link

cbind_sleep <- cbind(sleep, sleep_ratios)
head(cbind_sleep)

##    BodyWgt BrainWgt NonD Dream Sleep Span Gest Pred Exp Danger
BodyWgt BrainWgt Body_Brain   Body_Gest Brain_Gest
## 1 6654.000   5712.0 <NA>  <NA>   3.3 38.6  645    3   5      3
6654.000   5712.0 1.16491597 10.31627907  8.8558140
## 2    1.000      6.6  6.3     2   8.3  4.5   42    3   1      3
1.000      6.6 0.15151515  0.02380952  0.1571429
## 3    3.385     44.5 <NA>  <NA>  12.5   14   60    1   1      1
3.385     44.5 0.07606742  0.05641667  0.7416667
## 4    0.920      5.7 <NA>  <NA>  16.5 <NA>   25    5   2      3
0.920      5.7 0.16140351  0.03680000  0.2280000
## 5 2547.000   4603.0  2.1   1.8   3.9   69  624    3   5      4
2547.000   4603.0 0.55333478  4.08173077  7.3766026
## 6   10.550    179.5  9.1   0.7   9.8   27  180    4   4      4
10.550    179.5 0.05877437  0.05861111  0.9972222

str(cbind_sleep)

## 'data.frame':    62 obs. of  15 variables:
##  $ BodyWgt   : num  6654 1 3.38 0.92 2547 ...
##  $ BrainWgt  : num  5712 6.6 44.5 5.7 4603 ...
##  $ NonD      : chr  NA "6.3" NA NA ...
##  $ Dream     : chr  NA "2" NA NA ...
##  $ Sleep     : chr  "3.3" "8.3" "12.5" "16.5" ...
##  $ Span      : chr  "38.6" "4.5" "14" NA ...
##  $ Gest      : chr  "645" "42" "60" "25" ...
##  $ Pred      : int  3 3 1 5 3 4 1 4 1 1 ...
##  $ Exp       : int  5 1 1 2 5 4 1 5 2 1 ...
##  $ Danger    : int  3 3 1 3 4 4 1 4 1 1 ...
##  $ BodyWgt   : num  6654 1 3.38 0.92 2547 ...
##  $ BrainWgt  : num  5712 6.6 44.5 5.7 4603 ...
##  $ Body_Brain: num  1.1649 0.1515 0.0761 0.1614 0.5533 ...
##  $ Body_Gest : num  10.3163 0.0238 0.0564 0.0368 4.0817 ...
##  $ Brain_Gest: num  8.856 0.157 0.742 0.228 7.377 ...
```

We now have duplicate column names for BodyWgt and BrainWgt! This is bad and we recommend making sure your names are unique.

`rbind()` will add more rows to the `sleep` dataframe. Let's start by creating a new row. Create a vector that contains 10 elements to be added to the `sleep` data frame (remember that the `sleep` data frame contains 10 columns).

```
?rbind # Click "Combine R objects by rows or colums" link

ncol(sleep)

## [1] 10

rbind_for_sleep <- c("This", "is", "how", "rbind", "works", "This",
"is", "how", "rbind", "works")
rbind_for_sleep

##  [1] "This"  "is"    "how"   "rbind" "works" "This"  "is"    "how"
"rbind" "works"

#Now, `rbind()` it to the `sleep` data frame
sleep_rbind <- rbind(sleep, rbind_for_sleep)

sleep_rbind #We have successfully added another row!
```

Challenge 3

1. Load your `animals` dataset from Day 1 using `read.csv()`.
2. Create a subsetted data frame called `cats_dogs` that contains only cats and dogs.
3. Create a subsetted data frame that only contains healthy pigs!

6. Subsetting with the dplyr R package

The "dplyr" R package uses a different syntax to subset your data in perhaps a more efficient way than base R. dplyr's strength is specifically in its subsetting functions. It uses the pipe symbol `%>%` to pass the output of a function into the input of another.

Also, you do not need to include quotation marks " " when specifying column names. Furthermore, the pipe symbol saves you from having to write lots of nested parentheses. You might even find this code easier to read!

Fun Fact: You might have encountered pipes before in the Unix shell. In R, a pipe symbol is `%>%` while in the shell it is `|`. But the concept is the same!

NOTE: remember that | in R specifies an "or" logical operator.

```
?"|"
```

Data frames in dplyr are called "tibbles". All you have to do is "pipe in" functions to your dataset.

```
install.packages('dplyr', dependencies=TRUE)
library(dplyr)
```

First, we will use `sample()` to create some toy data containing some various gross domestic product information for North America. By not specifying `stringsAsFactors=FALSE`, R will automatically convert "Country" and "Region" to factor data types. This is fine for this example.

Set your seed to "1" so that we all get the sample resampled data:

```
set.seed(1)
gdp <- data.frame(Country=sample(c("Canada", "Mexico", "USA"), 50, 
replace=TRUE),
                  Region = sample(c("coastal", "inland", "mountain", 
"riverine"), 50, replace=TRUE),
                  Year = sample(2011:2015, 50, replace=TRUE),
                  Pop = sample(1000:50000, 50, replace=FALSE),
                  GDP = sample(4000:100000, 50, replace=FALSE),
                  Poverty = sample(1:10, 50, replace=TRUE))
head(gdp)

##   Country   Region Year   Pop   GDP Poverty
## 1  Canada   inland 2014 31118 29681       8
## 2  Mexico riverine 2012 28300 24989      10
## 3  Mexico   inland 2012 17109 53611       5
## 4     USA  coastal 2015 23202 29818       7
## 5  Canada  coastal 2014 25520 21391       5
## 6     USA  coastal 2012  9861 53781       2

gdp
```

`glimpse()` is dplyr's version of base R's `str()`:

```
library(dplyr)

## Warning: package 'dplyr' was built under R version 3.2.5

## ---------------------------------------------------------------------
---------------------------------------------------

## data.table + dplyr code now lives in dtplyr.
## Please library(dtplyr)!
## --------------------------------------------------------------------
--------------------------------------------------

##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:data.table':
##
##     between, last

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
?glimpse #Click the "Get a glimpse of your data" link

library(dplyr)
glimpse(gdp)
```

```
## Observations: 50
## Variables: 6
## $ Country <fctr> Canada, Mexico, Mexico, USA, Canada, USA, USA,
Mexico, Mexico, Canada, Canada, Canada, USA, Mexico...
## $ Region  <fctr> inland, riverine, inland, coastal, coastal,
coastal, inland, mountain, mountain, inland, riverine,...
## $ Year    <int> 2014, 2012, 2012, 2015, 2014, 2012, 2011, 2013,
2015, 2013, 2015, 2014, 2012, 2013, 2011, 2011, 201...
## $ Pop     <int> 31118, 28300, 17109, 23202, 25520, 9861, 26949,
4688, 14608, 11420, 14952, 44850, 22860, 39209, 441...
## $ GDP     <int> 29681, 24989, 53611, 29818, 21391, 53781, 58024,
16398, 28609, 72916, 96286, 13612, 77260, 94993, 8...
## $ Poverty <int> 8, 10, 5, 7, 5, 2, 3, 5, 4, 10, 6, 4, 3, 8, 8, 2, 1,
8, 7, 2, 1, 2, 4, 2, 3, 2, 3, 2, 5, 8, 1, 6, 9...
```

Now, sort the table alphabetically by Country and descending by Year (most recent Year first) using base R's `order()` command. Note that descending order for the Year column is specified by the - symbol.

Also, we are overwriting our gdp data frame instead of creating a new object.

```
?order #Click the "Fast row reordering of a data.table by reference"
link

gdp <- gdp[order(gdp$Country, -gdp$Year),]
head(gdp)
```

```
##    Country   Region Year   Pop   GDP Poverty
## 11  Canada riverine 2015 14952 96286       6
## 1    Canada   inland 2014 31118 29681       8
## 5    Canada  coastal 2014 25520 21391       5
## 12   Canada   inland 2014 44850 13612       4
## 25   Canada   inland 2014 19635 92679       3
## 34   Canada   inland 2014 25647 64596       4

gdp
```

6. Subsetting with the dplyr R package/ `select()` and `filter()`

The dataset `gdp` is the first item to go into the definition for our subsetted data frame called `country_year` followed by pipe `%>%`. All of our other functions will be "piped in" after it.

`select()` then chooses the columns we want to include in our subsetting operation. If we want to include only Country, Region, and Year, we would type:

```
?select #Click the "Select/rename variables by name" link

country_year <- gdp %>% select(Country, Region, Year)
head(country_year)

##    Country   Region Year
## 11  Canada riverine 2015
## 1   Canada   inland 2014
## 5   Canada  coastal 2014
## 12  Canada   inland 2014
## 25  Canada   inland 2014
## 34  Canada   inland 2014

glimpse(country_year)

## Observations: 50
## Variables: 3
## $ Country <fctr> Canada, Canada, Canada, Canada, Canada, Canada,
Canada, Canada, Canada, Canada, Canada, Canada, Me...
## $ Region  <fctr> riverine, inland, coastal, inland, inland, inland,
inland, riverine, riverine, coastal, inland, in...
## $ Year    <int> 2015, 2014, 2014, 2014, 2014, 2014, 2013, 2013,
2013, 2013, 2011, 2011, 2015, 2014, 2014, 2014, 201...

country_year

##    Country   Region Year
## 11  Canada riverine 2015
## 1   Canada   inland 2014
## 5   Canada  coastal 2014
## 12  Canada   inland 2014
## 25  Canada   inland 2014
## 34  Canada   inland 2014
## 10  Canada   inland 2013
## 22  Canada riverine 2013
## 27  Canada riverine 2013
## 38  Canada  coastal 2013
## 24  Canada   inland 2011
## 47  Canada   inland 2011
## 9   Mexico mountain 2015
## 42  Mexico  coastal 2014
## 45  Mexico riverine 2014
## 48  Mexico   inland 2014
## 8   Mexico mountain 2013
## 14  Mexico   inland 2013
## 19  Mexico  coastal 2013
## 23  Mexico   inland 2013
## 26  Mexico riverine 2013
## 30  Mexico riverine 2013
## 31  Mexico   inland 2013
## 40  Mexico  coastal 2013
## 2   Mexico riverine 2012
## 3   Mexico   inland 2012
## 28  Mexico   inland 2012
## 44  Mexico riverine 2012
## 16  Mexico   inland 2011
## 32  Mexico mountain 2011
## 33  Mexico   inland 2011
## 4      USA  coastal 2015
## 21     USA   inland 2015
## 35     USA riverine 2015
## 39     USA  coastal 2015
## 50     USA mountain 2015
## 17     USA   inland 2014
## 20     USA riverine 2014
## 41     USA  coastal 2014
## 36     USA  coastal 2013
## 37     USA mountain 2013
## 46     USA riverine 2013
## 6      USA  coastal 2012
## 13     USA   inland 2012
## 29     USA riverine 2012
## 43     USA mountain 2012
## 7      USA   inland 2011
## 15     USA mountain 2011
## 18     USA riverine 2011
## 49     USA riverine 2011
```

`filter()` chooses the rows you want to include. What if we are only interested in the Canada data? We can use `filter()` to select only rows with data for Canada.

```
?filter #Click "Return rows with matching conditions" link
```

```
canada <- gdp %>%
  filter(Country == "Canada") %>%
  select(Region, Year, Pop, GDP, Poverty)
head(canada)

##     Region Year   Pop   GDP Poverty
## 1 riverine 2015 14952 96286       6
## 2   inland 2014 31118 29681       8
## 3  coastal 2014 25520 21391       5
## 4   inland 2014 44850 13612       4
## 5   inland 2014 19635 92679       3
## 6   inland 2014 25647 64596       4

canada

##      Region Year   Pop   GDP Poverty
## 1  riverine 2015 14952 96286       6
## 2    inland 2014 31118 29681       8
## 3   coastal 2014 25520 21391       5
## 4    inland 2014 44850 13612       4
## 5    inland 2014 19635 92679       3
## 6    inland 2014 25647 64596       4
## 7    inland 2013 11420 72916      10
## 8  riverine 2013 42173 19880       2
## 9  riverine 2013 32555 28725       3
## 10  coastal 2013 36474 57250       2
## 11   inland 2011 20167 52960       2
## 12   inland 2011  6402 45140       1
```

Note: The order of operations is very important in this case. If we used `select()` first, `filter()` would not be able to find the Country variable since we would have removed it in the previous step.

6. Subsetting with the dplyr R package/ apply-split-combine

`group_by()`

Apply/split/combine saves us trouble when we want to add a new column to our existing dataframe.

We want to ***split*** our data into groups (in this case countries), ***apply*** some calculations on that group, then ***combine*** the results together afterwards.

More helpful, however, is the `group_by()` function, which will essentially use every unique criteria that we could have used in `filter()`. `group_by()` even allows us to pass in multiple arguments!

We can create a nice summary table using a combination of the `group_by()` and `summarize()` functions.

Let's say we want to take the mean and standard deviations of GDP for each country, and then add those values into a new column in a new data frame called `gdp_by_country`.

```
?group_by
?summarize

gdp_by_country <- gdp %>%
  group_by(Country, Region, Year) %>%
  summarize(MeanGDP = mean(GDP),
            sdGDP = sd(GDP))
head(gdp_by_country)

## Source: local data frame [6 x 5]
## Groups: Country, Region [3]
##
##   Country   Region  Year MeanGDP     sdGDP
##    <fctr>   <fctr> <int>   <dbl>     <dbl>
## 1  Canada  coastal  2013 57250.0        NA
## 2  Canada  coastal  2014 21391.0        NA
## 3  Canada   inland  2011 49050.0  5529.575
## 4  Canada   inland  2013 72916.0        NA
## 5  Canada   inland  2014 50142.0 35456.113
## 6  Canada riverine  2013 24302.5  6254.359

#Why do you think some rows have `NA` for the `sdGDP` column? (hint:
because they have only 1 observation! For any sort of variance
computation to be calculated, it must have at least 2 entries).
```

Notice how dplyr only prints out the columns that fit in your console and a truncated number of rows.

```
gdp
```

This can be changed in the options settings:

```
options(dplyr.print_max=99999)

gdp
```

6. Subsetting with the dplyr R package/ apply-split-combine

`mutate()`

We can use `mutate()` to add a new column to our original `gdp` dataframe. `mutate()` is similar to `summarize()` except you do not need to create a new object. Let's also add a new column "GDP_Pop", which contains "GDP" divided by "Pop".

```
?mutate

gdp <- gdp %>%
  group_by(Country, Year) %>%
  mutate(MeanGDP = mean(GDP),
         sdGDP = sd(GDP),
         GDP_Pop = GDP/Pop)
head(gdp)
```

```
## Source: local data frame [6 x 9]
## Groups: Country, Year [2]
##
##   Country   Region  Year   Pop   GDP Poverty MeanGDP    sdGDP
GDP_Pop
##    <fctr>   <fctr> <int> <int> <int>   <int>   <dbl>    <dbl>
<dbl>
## 1  Canada riverine  2015 14952 96286       6 96286.0       NA
6.4396736
## 2  Canada   inland  2014 31118 29681       8 44391.8 33289.28
0.9538209
## 3  Canada  coastal  2014 25520 21391       5 44391.8 33289.28
0.8382053
## 4  Canada   inland  2014 44850 13612       4 44391.8 33289.28
0.3035006
## 5  Canada   inland  2014 19635 92679       3 44391.8 33289.28
4.7200917
## 6  Canada   inland  2014 25647 64596       4 44391.8 33289.28
2.5186572
```

6. Subsetting with the dplyr R package/ `arrange()`

We can also arrange our data frame with `arrange()`. This is similar to `order()` in base R, or sort in MS Excel.

Let's sort our tibble alphabetically (by default) by "Region". All we have to do is pipe in `arrange()` from our previous example.

```
?arrange

gdp <- gdp %>%
  group_by(Country, Year) %>%
  mutate(MeanGDP = mean(GDP),
         sdGDP = sd(GDP),
         GDP_Pop = GDP/Pop) %>%
  arrange(Country, -Year, Region)
head(gdp)

## Source: local data frame [6 x 9]
## Groups: Country, Year [2]
##
##   Country   Region  Year   Pop   GDP Poverty MeanGDP    sdGDP
GDP_Pop
##    <fctr>   <fctr> <int> <int> <int>   <int>   <dbl>    <dbl>
<dbl>
## 1  Canada riverine  2015 14952 96286       6 96286.0       NA
6.4396736
## 2  Canada  coastal  2014 25520 21391       5 44391.8 33289.28
0.8382053
```

```
## 3  Canada   inland  2014 31118 29681       8 44391.8 33289.28
0.9538209
## 4  Canada   inland  2014 44850 13612       4 44391.8 33289.28
0.3035006
## 5  Canada   inland  2014 19635 92679       3 44391.8 33289.28
4.7200917
## 6  Canada   inland  2014 25647 64596       4 44391.8 33289.28
2.5186572

gdp
```

Challenge 4

1. Use dplyr to add the medians of BodyWgt and BrainWgt to the `sleep` data frame. You have not yet seen how to calculate the median. How do you think you might find out how to do so?

7. Tidying/reshaping data with the tidyr R package

For our final example, we are going quickly create some data in "wide" format so that we can convert it to "long" and then to "medium" formats.

"Wide" format generally refers to data where values (e.g., GDP, Pop) are spread out across columns. You might also hear this referred to as "multivariate" format.

"Long" format refers to data that has one column for the values, and the other columns are ID variables. You might also hear this referred to as "univariate" format".

"Medium" format is somewhere in between!

In R, some functions are explicitly written for long format data, and others for wide format data so it is useful to know how to tidy your data.

The two most important properties of tidy data are: 1) Each column is a variable. 2) Each row is an observation.

First, let's install and call the 'tidyr' package.

```
install.packages("tidyr", dependencies=TRUE)
library(tidyr)
```

Now let's create some "wide" format toy data for medal counts from the Olympics.

```
set.seed(1)
medals_wide <- data.frame(
  country = c("Canada",  "Mexico",  "USA" ),
  gold_2012 = sample(1:5, 3, replace=TRUE),
  silver_2012 = sample(6:10, 3, replace=TRUE),
  bronze_2012 = sample(11:15, 3, replace=TRUE),
  gold_2016 = sample(1:5, 3, replace=TRUE),
  silver_2016 = sample(6:10, 3, replace=TRUE),
  bronze_2016 = sample(11:15, 3, replace=TRUE)
  )
medals_wide

##   country gold_2012 silver_2012 bronze_2012 gold_2016 silver_2016
bronze_2016
## 1  Canada         2          10          15         1           9
13
## 2  Mexico         2           7          14         2           7
14
## 3     USA         3          10          14         1           9
15
```

7. Tidying/reshaping data with the tidyr R package/ `gather()`

We can use `gather()` to combine the observation variables (gold, silver, bronze) into a single variable by year. This is similar to "melting" your data from wide to long format in the "reshape2" R package.

```
?gather

library(tidyr)

## Warning: package 'tidyr' was built under R version 3.2.5

glimpse(medals_wide)

## Observations: 3
## Variables: 7
## $ country     <fctr> Canada, Mexico, USA
## $ gold_2012   <int> 2, 2, 3
## $ silver_2012 <int> 10, 7, 10
## $ bronze_2012 <int> 15, 14, 14
## $ gold_2016   <int> 1, 2, 1
## $ silver_2016 <int> 9, 7, 9
## $ bronze_2016 <int> 13, 14, 15

medals_long <- medals_wide %>%
  gather(obstype_year, obs_values, 2:7)
medals_long

##    country obstype_year obs_values
## 1   Canada    gold_2012          2
## 2   Mexico    gold_2012          2
## 3      USA    gold_2012          3
## 4   Canada  silver_2012         10
## 5   Mexico  silver_2012          7
## 6      USA  silver_2012         10
## 7   Canada  bronze_2012         15
## 8   Mexico  bronze_2012         14
## 9      USA  bronze_2012         14
## 10  Canada    gold_2016          1
## 11  Mexico    gold_2016          2
## 12     USA    gold_2016          1
## 13  Canada  silver_2016          9
## 14  Mexico  silver_2016          7
## 15     USA  silver_2016          9
## 16  Canada  bronze_2016         13
## 17  Mexico  bronze_2016         14
## 18     USA  bronze_2016         15
```

Notice that we put 3 arguments into the `gather()` function: 1. The name the new column for the new ID variable (`obstype_year`), 2. The name for the new amalgamated observation variable (`obs_value`), 3. The indices of the old observation variables (`2:7`, signalling columns 2 through 7) that we want to gather into one variable (medal types and years). Notice that we don't want to melt down column 1 (country), as this is the "ID" variable.

7. Tidying/reshaping data with the tidyr R package/ `separate()`

You will also notice that in our "long" dataset, "obstype" actually contains 2 pieces of information: "medal type" (gold, silver, or bronze) and "year".

`separate()` can be used to split "obstype" (gold, silver, or bronze) back into medal type and year columns. We want to separate it at the underscore _:

```
?separate

medals_long_sep <- medals_long %>%
  separate(obstype_year, into = c("obs_type", "year"), sep = "_") %>%
  mutate(year = as.integer(year))
medals_long_sep
```

```
##    country obs_type year obs_values
## 1   Canada     gold 2012          2
## 2   Mexico     gold 2012          2
## 3      USA     gold 2012          3
## 4   Canada   silver 2012         10
## 5   Mexico   silver 2012          7
## 6      USA   silver 2012         10
## 7   Canada   bronze 2012         15
## 8   Mexico   bronze 2012         14
## 9      USA   bronze 2012         14
## 10  Canada     gold 2016          1
## 11  Mexico     gold 2016          2
## 12     USA     gold 2016          1
## 13  Canada   silver 2016          9
## 14  Mexico   silver 2016          7
## 15      USA   silver 2016          9
## 16   Canada   bronze 2016         13
## 17   Mexico   bronze 2016         14
## 18      USA   bronze 2016         15

glimpse(medals_long_sep) #We have successfully separated "obs_type" and
"year"! :)

## Observations: 18
## Variables: 4
## $ country    <fctr> Canada, Mexico, USA, Canada, Mexico, USA,
Canada, Mexico, USA, Canada, Mexico, USA, Canada, Mex...
## $ obs_type   <chr> "gold", "gold", "gold", "silver", "silver",
"silver", "bronze", "bronze", "bronze", "gold", "gol...
## $ year       <int> 2012, 2012, 2012, 2012, 2012, 2012, 2012, 2012,
2012, 2016, 2016, 2016, 2016, 2016, 2016, 2016, ...
## $ obs_values <int> 2, 2, 3, 10, 7, 10, 15, 14, 14, 1, 2, 1, 9, 7, 9,
13, 14, 15
```

7. Tidying/reshaping data with the tidyr R package/ `spread()`

You can spread this dataset into 'medium' format using `spread()`

```
?spread

medals_medium <- medals_long_sep %>%
  spread(obs_type, obs_values)
medals_medium

##   country year bronze gold silver
## 1  Canada 2012     15    2     10
## 2  Canada 2016     13    1      9
## 3  Mexico 2012     14    2      7
## 4  Mexico 2016     14    2      7
## 5     USA 2012     14    3     10
## 6     USA 2016     15    1      9
```

Acknowledgements

Wickham H, Grolemund G. 2016. R for Data Science