MTH 3270 Notes 4

4 Data Wrangling (Cont'd) (4)

4.11 Chaining Together Actions Using the Pipe Operator %>%

- We'll use the `flights` data set in the `"nycflights13"` package again to illustrate the use of the **pipe operator** %>%:

```
library(nycflights13)
```

Suppose we want to look at the relationship between the `distance` traveled and arrival `delay` for destinations that received more than 20 flights. We could type:

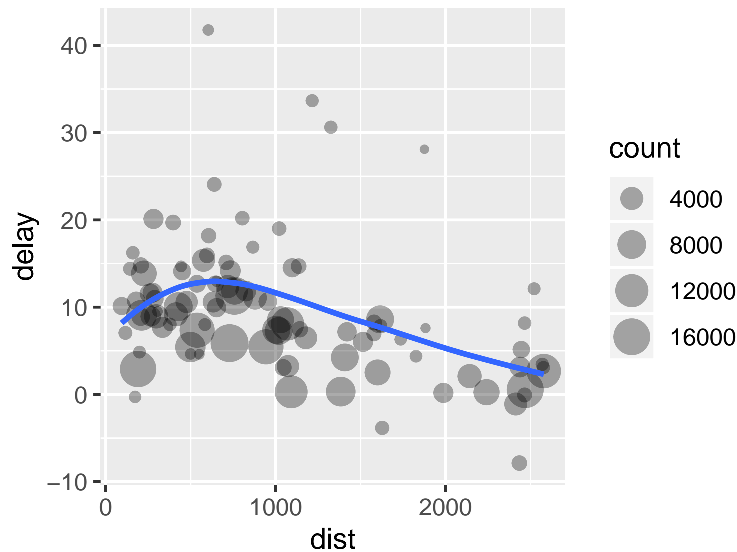
```
by_dest <- group_by(.data = flights, dest)

delay_dist <- summarize(.data = by_dest,
                        count = n(),
                        dist = mean(distance, na.rm = TRUE),
                        delay = mean(arr_delay, na.rm = TRUE))

delay_20_plus <- filter(.data = delay_dist, count > 20, dest != "HNL")
```

Here's a plot of the data:

```
ggplot(data = delay_20_plus, mapping = aes(x = dist, y = delay)) +
  geom_point(aes(size = count), alpha = 1/3) +
  geom_smooth(se = FALSE)
```



(The `alpha` argument to `geom_point()` controls the degree of transparency of points for overplotting.)

It looks like arrival delays increase with distance up to a distance of about 600 miles, then decrease. Perhaps with longer flights, there's more ability to make up for lost time.

- Another way to write the command above is to use the **pipe operator** `%>%`:

```
delay_20_plus <- flights %>%
    group_by(dest) %>%
    summarize(count = n(),
              dist = mean(distance, na.rm = TRUE),
              delay = mean(arr_delay, na.rm = TRUE)) %>%
    filter(count > 20, dest != "HNL")
```

The **pipe operator** `%>%` passes the output data frame from one command as the input (first argument) for the next command.

With the **pipe operator**, there's no need to pick names for intermediate data frames. This can make the code more **readable**.

- In general,
  - `x %>% f()` is equivalent to `f(x)`.
  - `x %>% f(y)` is equivalent to `f(x, y)`.
  - `x %>% f(y) %>% g(z)` is equivalent to `g(f(x, y), z)`.
  - etc.

Section 4.11 Exercises

Exercise 1 This exercise concerns the **pipe operator** %>%.

a) Rewrite the following command using the **pipe operator**:

```
delay <- select(.data = flights, arr_delay)
```

b) Rewrite the following command using the **pipe operator**:

```
dest_delay <- select(.data = flights, dest, arr_delay)
```

c) Rewrite the following command using the **pipe operator**:

```
dest_delay <- select(.data = flights, dest, arr_delay)
sea_den <- filter(.data = dest_delay,
                  dest == "SEA" | dest == "DEN")
```

d) Rewrite the following sequence of commands using the **pipe operator**:

```
dest_delay <- select(.data = flights, dest, arr_delay)
sea_den <- filter(.data = dest_delay,
                  dest == "SEA" | dest == "DEN")
by_dest <- group_by(sea_den, dest)
delay_by_dest <- summarize(by_dest,
                           delay = mean(arr_delay, na.rm = TRUE))
```

Exercise 2 Rewrite the following command using the **pipe operator**:

```
den_delays <- summarize(filter(.data = flights,
                               dest == "DEN"),
                        mean_dep_delay = mean(dep_delay, na.rm = TRUE),
                        mean_arr_delay = mean(arr_delay, na.rm = TRUE))
```

4.12 Combining Multiple Data Frames

- These three functions (from the "dplyr" package) are useful for combining two data frames:

```
inner_join()    # Merge two data frames x and y by matching rows.
                # Returns only rows that have matches in both x and y.
left_join()     # Merge two data frames x and y by matching rows.
                # Returns all rows of x even if they do not have
```

```
                      # a match in y.
full_join()           # Merge two data frames x and y by matching rows.
                      # Returns all rows of x and all rows of y regardless
                      # of whether they have a match.
```

- All three functions append the columns of a data frame `y` to another data frame `x` by matching the rows of the two data frames.
- Here's a data frame with **names** and **ages** of four people:

```
NamesAndAges

##    Name Age
## 1  John  23
## 2 Karen  27
## 3   Ann  19
```

and here's another with their **names** and **weights**:

```
NamesAndWeights

##    Name Weight
## 1  John    155
## 2 Karen    170
## 3   Ann    157
```

To combine the two data frames using `inner_join()`, matching their rows by the `Name` variable, we type:

```
inner_join(x = NamesAndAges, y = NamesAndWeights, by = "Name")

##    Name Age Weight
## 1  John  23    155
## 2 Karen  27    170
## 3   Ann  19    157
```

To combine them using `left_join()`, we type:

```
left_join(x = NamesAndAges, y = NamesAndWeights, by = "Name")

##    Name Age Weight
## 1  John  23    155
## 2 Karen  27    170
## 3   Ann  19    157
```

To combine them using `full_join()`, we type:

```
full_join(x = NamesAndAges, y = NamesAndWeights, by = "Name")

##    Name Age Weight
## 1  John  23    155
## 2 Karen  27    170
## 3   Ann  19    157
```

- Above, `inner_join()`, `left_join()`, and `full_join()` all returned the **same thing**.

  They return **different things** when some rows of `x` and `y` don't match:

  - `inner_join()` returns only the rows that have matches in both `x` and `y`.
  - `left_join()` returns all rows of `x` regardless of whether or not there's a match in `y`. Rows of `x` with no match in `y` will have `NA` values in the new columns.
  - `full_join()` returns all rows of `x` *and* all rows of `y` regardless of whether they have a match in the other data frame. Rows of either data frame that don't have a match in the other will have `NA` values in the new columns.

- For example, consider again the `NamesAndAges` data frame:

```
NamesAndAges

##    Name Age
## 1  John  23
## 2 Karen  27
## 3   Ann  19
```

  and this other data frame containing three `Names`, *only two of which match* the first data frame, and their `Heights`:

```
NamesAndHeights

##    Name Height
## 1 Karen     63
## 2   Ann     65
## 3  Karl     36
```

  Using `inner_join()` gives:

```
inner_join(x = NamesAndAges, y = NamesAndHeights, by = "Name")

##    Name Age Height
## 1 Karen  27     63
## 2   Ann  19     65
```

  (Only the `Names` that are in *both* data frames are returned.)

  Using `left_join()` gives:

```
left_join(x = NamesAndAges, y = NamesAndHeights, by = "Name")

##    Name Age Height
## 1  John  23     NA
## 2 Karen  27     63
## 3   Ann  19     65
```

(All the `Name`s that are in the first data frame are returned, and `NA` is inserted for the missing `Height`. Note that the third `Name` in the second data frame isn't returned.)

Using `full_join()` gives:

```
full_join(x = NamesAndAges, y = NamesAndHeights, by = "Name")

##    Name Age Height
## 1  John  23     NA
## 2 Karen  27     63
## 3   Ann  19     65
## 4  Karl  NA     36
```

(All the `Name`s from *both* data frames are returned, and `NA`s are inserted for the missing `Height` and the missing `Age`.)

- The `Name` variable is common to both data frames, and is used to match rows.

  The variable (such as `Name`) that's common to both data frames and is used to match their rows is called the ***key***.

- The **key** values don't have to be in the same order in the two data frames.

  For example, if the `Names` were in different orders in the two data frames, `inner_join()`, `left_join()`, and `full_join()` would match their orders before combining:

```
NamesAndAges

##    Name Age
## 1  John  23
## 2 Karen  27
## 3   Ann  19

## These Names are in a jumbled order:
JumbledNamesAndWts

##    Name Weight
## 3   Ann    157
## 1  John    155
## 2 Karen    170
```

```
## The rows are put in matching order before combining:
inner_join(NamesAndAges, JumbledNamesAndWts, by = "Name")

##    Name Age Weight
## 1  John  23    155
## 2 Karen  27    170
## 3   Ann  19    157
```

- Sometimes *two* **key** variables needed to distinguish rows in a data set.

  For example, suppose some `Names` were *duplicated* (e.g. there are *two* `"John"`s and *three* `"Ann"`s below), but we had another column `City` that could be used to distinguish between them:

```
NamesDuplicatedAndAges

##    Name      City Age
## 1  John    Denver  23
## 2  John  Longmont  42
## 3 Karen    Salida  27
## 4   Ann   Boulder  19
## 5   Ann    Denver  29
## 6   Ann Leadville  45
## 7  Karl    Denver  36
```

```
NamesDuplicatedAndWts

##    Name      City Weight
## 1  John    Denver    155
## 2  John  Longmont    203
## 3 Karen    Salida    170
## 4   Ann   Boulder    157
## 5   Ann    Denver    161
## 6   Ann Leadville    164
## 7  Karl    Denver    201
```

  In this case, a proper merge of the two data frames would need to be done by the values in *both* columns.

  To to do this, we specify both `Name` *and* `City` in a `"character"` vector passed to `inner_join()` via the `by` argument:

```
inner_join(x = NamesDuplicatedAndAges,
           y = NamesDuplicatedAndWts,
           by = c("Name", "City"))
```

```
##    Name      City Age Weight
## 1  John    Denver  23    155
## 2  John  Longmont  42    203
## 3 Karen    Salida  27    170
## 4   Ann   Boulder  19    157
## 5   Ann    Denver  29    161
## 6   Ann Leadville  45    164
## 7  Karl    Denver  36    201
```

- In fact, by default `inner_join()`, `left_join()`, and `full_join()` merge data frames by whatever column names the two data frames have in common.

  So in all of the examples above, it actually *wasn't necessary* to specify the **key** variable(s) explicitly via the `by` argument.

Section 4.12 Exercises

Exercise 3 Here are two data frames, `df1` and `df2`, containing responses to two survey questions:

```
df1 <- data.frame(Respondent_ID = c(1001, 1002, 1003),
                  Q1_Response = c(55, 62, 39))
df1

##   Respondent_ID Q1_Response
## 1          1001          55
## 2          1002          62
## 3          1003          39
```

```
df2 <- data.frame(Respondent_ID = c(1002, 1003, 1004),
                  Q2_Response = c("yes", "no", "yes"))
df2

##   Respondent_ID Q2_Response
## 1          1002         yes
## 2          1003          no
## 3          1004         yes
```

Notice that the `Respondent_ID`s **differ** across two data frames.

a) Guess what the result of the following command will be, then check your answer and report the result.

```
inner_join(x = df1, y = df2, by = "Respondent_ID")
```

b) Guess what the result of the following command will be, then check your answer and report the result.

```r
left_join(x = df1, y = df2, by = "Respondent_ID")
```

c) Guess what the result of the following command will be, then check your answer and report the result.

```r
full_join(x = df1, y = df2, by = "Respondent_ID")
```

d) If we didn't specify `by = "Respondent_ID"`, what would each of the `*_join()` functions use to match rows? Try it, for example:

```r
full_join(x = df1, y = df2)
```

e) What would happen if `Q1_Response` and `Q2_Response` were *both* named `Response` in the two data frames, and we typed:

```r
full_join(x = df1, y = df2)
```

Try it (after changing both names to `Response`), and report the result:

```r
df1 <- rename(.data = df1, Response = Q1_Response)
df2 <- rename(.data = df2, Response = Q2_Response)
```

f) What would happen if, as in Part *e*), `Q1_Response` and `Q2_Response` were *both* named `Response`, and we typed:

```r
inner_join(x = df1, y = df2)
```

Try it and report the result.

Exercise 4 Here are two data frames containing responses to two survey questions:

```r
df1 <- data.frame(Respondent_ID = c(1000, 1001, 1002, 1003, 1004, 1005, 1006),
                  Q1_Response = c(55, 62, 39, 45, 70, 77, 56))
df1
```

```r
df2 <- data.frame(Respondent_ID = c(1003, 1002, 1000, 1004, 1006, 1001, 1005),
                  Q2_Response = c(12, 17, 23, 24, 19, 30, 20))
df2
```

Note that the `Respondent_ID`s are the same, but in *different orders*.

a) What happens to the ordering of the rows of `df2` when you combine it with `df1` using:

```r
inner_join(x = df1, y = df2, by = "Respondent_ID")
```

b) How would the result differ if you swapped the roles of `df1` and `df2`, e.g.

```
inner_join(x = df2, y = df1, by = "Respondent_ID")
```

Exercise 5 Here are two data frames:

```
dfX <- data.frame(LastName = c("Smith", "Smith", "Jones", "Smith",
                               "Olsen", "Taylor", "Olsen"),
                  FirstName = c("John", "Kim", "John", "Marge", "Bill",
                                "Bill", "Erin"),
                  Gender = c("M", "F", "M", "F", "M", "M", "F"),
                  ExamScore = c(75, 80, 64, 78, 90, 89, 79))
dfX

##   LastName FirstName Gender ExamScore
## 1    Smith      John      M        75
## 2    Smith       Kim      F        80
## 3    Jones      John      M        64
## 4    Smith     Marge      F        78
## 5    Olsen      Bill      M        90
## 6   Taylor      Bill      M        89
## 7    Olsen      Erin      F        79
```

```
dfY <- data.frame(LastName = c("Olsen", "Jones", "Taylor", "Smith",
                               "Olsen", "Smith", "Smith"),
                  FirstName = c("Bill", "John", "Bill", "Kim", "Erin",
                                 "John", "Marge"),
                  Gender = c("M", "M", "M", "F", "F", "M", "F"),
                  Grade = c("A", "D", "B", "B", "C", "C", "C"))
dfY

##   LastName FirstName Gender Grade
## 1    Olsen      Bill      M     A
## 2    Jones      John      M     D
## 3   Taylor      Bill      M     B
## 4    Smith       Kim      F     B
## 5    Olsen      Erin      F     C
## 6    Smith      John      M     C
## 7    Smith     Marge      F     C
```

Notice that the two data frames contain the *same* seven people, but in different orders. Notice also that both the `LastName` *and* `FirstName` are needed to uniquely identify the people.

a) Write a command involving, say, `full_join()` that combines the two data frames *by person*. You should end up with this:

```
##   LastName FirstName Gender ExamScore Grade
## 1    Smith      John      M        75     C
## 2    Smith       Kim      F        80     B
## 3    Jones      John      M        64     D
## 4    Smith     Marge      F        78     C
## 5    Olsen      Bill      M        90     A
## 6   Taylor      Bill      M        89     B
## 7    Olsen      Erin      F        79     C
```

b) If you don't specify **key** variables to match by via the `by` argument, matching is done by whatever columns the two data frames have in common (`LastName`, `FirstName`, and `Gender`).

What happens with the third variable (`Gender`) when you only specify the other two (`LastName` and `FirstName`) via the `by` argument? Try it:

```
full_join(x = dfX, y = dfY, by = c("LastName", "FirstName"))
```

c) If values in a **key** variable *don't* uniquely identify rows, i.e. if there are multiple matches between rows of two data frames, **all combinations** of the matches are returned.

What would happen if you tried to combine `dfX` and `dfY` *only* specifying `LastName` as the **key** variable? Try it:

```
full_join(x = dfX, y = dfY, by = "LastName")
```

5 Tidy Data and Iteration (5)

5.1 Introduction: The "tidyr" Package

- The the `"tidyr"` package contains several functions for " *tidying*" data and for *iterating* a statistical analysis by *groups*. Type:

```
help(package = tidyr)
```

to see a list of the functions (and data sets) contained in `"tidyr"`.

5.2 Using gather() and spread()

- Sometimes a **single variable** is spread across **multiple columns**. Other times, a single **observation** is scattered across **multiple rows**.

For example, here are two data frames that contain the *same data* (student GPAs), but *arranged differently* (***wide*** in the first case and ***narrow*** in the second):

```
gpaDataWide
```

```
##   StudentID Semester1 Semester2 Semester3
## 1       111      2.54      3.42      3.93
## 2       112      2.90      3.19      3.18
## 3       113      3.99      3.45      2.89
## 4       114      2.99      2.78      3.70
## 5       115      3.67      2.68      2.81
```

```
gpaDataNarrow
```

```
##    StudentID  Semester  GPA
## 1        111 Semester1 2.54
## 2        112 Semester1 2.90
## 3        113 Semester1 3.99
## 4        114 Semester1 2.99
## 5        115 Semester1 3.67
## 6        111 Semester2 3.42
## 7        112 Semester2 3.19
## 8        113 Semester2 3.45
## 9        114 Semester2 2.78
## 10       115 Semester2 2.68
## 11       111 Semester3 3.93
## 12       112 Semester3 3.18
## 13       113 Semester3 2.89
## 14       114 Semester3 3.70
## 15       115 Semester3 2.81
```

In the **wide** format, the **variable** (GPA) is "*spread*" across multiple columns (Semesters 1-3). In the **narrow** format, those columns have been "*gathered*" into a single column.

- The following functions, from the `"tidyr"` package, are useful for converting data back and forth between the **wide** and **narrow** formats:

```
gather()     # Convert from wide to narrow by stacking columns.
spread()     # Convert from narrow to wide by unstacking a column.
```

- To convert a data frame from the **wide** format, like `gpaDataWide`, to the **narrow** format, use `gather()`. For example (using `gpaDataWide` from above):

```
gather(data = gpaDataWide,
       key = Semester,
       value = GPA,
       Semester1, Semester2, Semester3)
```

```
##    StudentID  Semester  GPA
## 1        111 Semester1 2.54
## 2        112 Semester1 2.90
## 3        113 Semester1 3.99
## 4        114 Semester1 2.99
## 5        115 Semester1 3.67
## 6        111 Semester2 3.42
## 7        112 Semester2 3.19
## 8        113 Semester2 3.45
## 9        114 Semester2 2.78
## 10       115 Semester2 2.68
## 11       111 Semester3 3.93
## 12       112 Semester3 3.18
## 13       113 Semester3 2.89
## 14       114 Semester3 3.70
## 15       115 Semester3 2.81
```

You're free to invent any name for the `key` argument. It's used as the name of the **categorical** variable in the **narrow** data frame whose *categories* are the *names* of the columns in the **wide** format (`Semester1`, `Semester2`, `Semester3` above) that get "gathered".

You're also free to invent a name for the `value` argument. It's the name of the variable in the **narrow** data frame that contains the *values* from the "gathered" columns (GPAs above).

- We can use the "helper" functions from `select()` (i.e. `starts_with()`, `ends_with()`, `contains()`, and `num_range()`) to specify columns in `gather()`. For example, to use `num_range()` to obtain the same result as the above, type:

```
gather(data = gpaDataWide,
       key = Semester,
       value = GPA,
       num_range("Semester", 1:3))
```

```
##    StudentID  Semester  GPA
## 1        111 Semester1 2.54
## 2        112 Semester1 2.90
## 3        113 Semester1 3.99
## 4        114 Semester1 2.99
## 5        115 Semester1 3.67
## 6        111 Semester2 3.42
## 7        112 Semester2 3.19
## 8        113 Semester2 3.45
## 9        114 Semester2 2.78
## 10       115 Semester2 2.68
## 11       111 Semester3 3.93
## 12       112 Semester3 3.18
## 13       113 Semester3 2.89
## 14       114 Semester3 3.70
```

```
## 15        115 Semester3 2.81
```

- To covert from the **narrow** format, like `gpaDataNarrow`, to **wide**, use `spread()`:

```
spread(data = gpaDataNarrow,
      key = Semester,
      value = GPA)

##   StudentID Semester1 Semester2 Semester3
## 1       111      2.54      3.42      3.93
## 2       112      2.90      3.19      3.18
## 3       113      3.99      3.45      2.89
## 4       114      2.99      2.78      3.70
## 5       115      3.67      2.68      2.81
```

- The `StudentID` variable in `gpaDataNarrow` is needed to match `GPA`s for a given student across `Semester`s to compose a row in `gpaDataWide`. Without the `StudentID` variable in `gpaDataNarrow`, `spread()` would return an error message.

Section 5.2 Exercises

Exercise 6 Here's a data frame containing responses for four individuals in each of three treatment groups in an experiment:

```
xWide <- data.frame(GrpA = c(1, 4, 2, 3),
                    GrpB = c(7, 5, 8, 6),
                    GrpC = c(9, 9, 8, 7))
xWide

##   GrpA GrpB GrpC
## 1    1    7    9
## 2    4    5    9
## 3    2    8    8
## 4    3    6    7
```

Write a command involving `gather()` that converts `xWide` to **narrow** format. Name the columns `Grp` and `Y`. You should end up with this:

```
xNarrow

##     Grp Y
## 1  GrpA 1
## 2  GrpA 4
## 3  GrpA 2
## 4  GrpA 3
## 5  GrpB 7
## 6  GrpB 5
## 7  GrpB 8
## 8  GrpB 6
## 9  GrpC 9
## 10 GrpC 9
## 11 GrpC 8
## 12 GrpC 7
```

Exercise 7 Here's are data from a study in which a variable `Y` was recorded on each of five subjects *before* and *after* an intervention:

```
xNarrow <- data.frame(Subject = c(1:5, 1:5),
                      Period = c("Before", "Before", "Before", "Before",
                                 "Before", "After", "After", "After",
                                 "After", "After"),
                      Y = c(22, 45, 32, 45, 30, 60, 44, 24, 56, 59),
                      stringsAsFactors = FALSE)
```

```
xNarrow

##    Subject Period  Y
## 1        1 Before 22
## 2        2 Before 45
## 3        3 Before 32
## 4        4 Before 45
## 5        5 Before 30
## 6        1  After 60
## 7        2  After 44
## 8        3  After 24
## 9        4  After 56
## 10       5  After 59
```

a) Write a command involving `spread()` that converts `xNarrow` to a **wide** format. You should end up with this:

```
xWide
```

```
##   Subject After Before
## 1       1    60     22
## 2       2    44     45
## 3       3    24     32
## 4       4    56     45
## 5       5    59     30
```

b) The `Subject` variable in `xNarrow` is needed to match `Y` values for a given subject across `Period`s to compose their row in `xWide`. What would happen if the `Subject` variable was missing? Try it:

```
xNarrowNoSubject <- data.frame(Period = c("Before", "Before", "Before",
                               "Before", "Before", "After", "After",
                               "After", "After", "After"),
                         Y = c(22, 45, 32, 45, 30, 60, 44, 24, 56, 59),
                         stringsAsFactors = FALSE)
```

```
xNarrowNoSubject
```

```
##    Period  Y
## 1  Before 22
## 2  Before 45
## 3  Before 32
## 4  Before 45
## 5  Before 30
## 6   After 60
## 7   After 44
## 8   After 24
## 9   After 56
## 10  After 59
```

Exercise 8 This exercise involves using the "helper" functions (from `select()`) in `gather()`.

Recall that `num_range("x", 1:3)` matches `x1`, `x2`, `x3`.

Here's a **wide** data frame in which a variable was recorded on each of three `Subject`s at four different time points:

```
xWide <- data.frame(Subject = c(1001, 1002, 1003),
                    t1 = c(22, 45, 32),
                    t2 = c(45, 30, 60),
                    t3 = c(44, 24, 56),
                    t4 = c(55, 27, 53))
xWide

##   Subject t1 t2 t3 t4
## 1    1001 22 45 44 55
## 2    1002 45 30 24 27
## 3    1003 32 60 56 53
```

Write a command involving `gather()` and the "helper" function `num_range()` that converts `xWide` to **narrow** format. You should end up with this:

```
xNarrow

##    Subject Time  Y
## 1     1001   t1 22
## 2     1002   t1 45
## 3     1003   t1 32
## 4     1001   t2 45
## 5     1002   t2 30
## 6     1003   t2 60
## 7     1001   t3 44
## 8     1002   t3 24
## 9     1003   t3 56
## 10    1001   t4 55
## 11    1002   t4 27
## 12    1003   t4 53
```

Exercise 9 Here's the **wide** data frame from Exercise 8, but this time it includes each `Subject`'s `Gender`:

```
xWide <- data.frame(Subject = c(1001, 1002, 1003),
                    Gender = c("m", "f", "f"),
                    t1 = c(22, 45, 32),
                    t2 = c(45, 30, 60),
                    t3 = c(44, 24, 56),
                    t4 = c(55, 27, 53))
xWide

##   Subject Gender t1 t2 t3 t4
## 1    1001      m 22 45 44 55
## 2    1002      f 45 30 24 27
## 3    1003      f 32 60 56 53
```

The `Gender` of a `Subject` is *constant* (i.e. doesn't change over the four time points). Thus we'd want the `Gender` column to be duplicated four times in the **narrow** format

just as the `Subject` column was in Exercise 8.

What happens to the `Gender` column when you convert `xWide` to **narrow** format, e.g. by typing:

```
xNarrow <- gather(data = xWide, key = Time, value = Y, num_range("t", 1:4))
```

5.3 Separating and Uniting Columns Using `separate()` and `unite()`

- Sometimes a **single column** contains **multiple variables**. Other times, a **single variable** is split across **multiple columns**.

  The following functions (from `"tidyr"`) are useful for separating and uniting columns

```
separate()     # Separate a column that contains multiple variables.
unite()        # Unite multiple columns across which a single variable
               # is spread (the reverse of separate()).
```

- Here's an example in which **two variables**, GPA and letter grade, are in a **single column**:

```
gpaDataWide

##   StudentID GPAandGrade
## 1       111      2.54/C
## 2       112       2.9/B
## 3       113      3.99/A
## 4       114      2.99/B
## 5       115      3.67/A
```

- To split the `GPAandGrade` column into two, using `separate()`, type:

```
separate(data = gpaDataWide,
        col = GPAandGrade,
        into = c("GPA", "Grade"),
        sep = "/")

##   StudentID  GPA Grade
## 1       111 2.54     C
## 2       112  2.9     B
## 3       113 3.99     A
## 4       114 2.99     B
## 5       115 3.67     A
```

We use the argument `col` to specify the name of the column to be separated, `into` to specify the names of the new columns, and `sep` to specify the `"character"` *separator* between columns.

For more info, look at the help file by typing:

```
? separate
```

- The `unite()` function does the reverse of `separate()`: It forms a single column from multiple columns across which a single variable is spread.

  For more info, look at the help file by typing:

```
? unite
```

Section 5.3 Exercises

Exercise 10 Here's a data frame containing the `Rate` of occurrences a rare disease (number of cases divided by population) and the year for three countries:

```
diseases <- data.frame(country = c("Afghanistan", "Afghanistan",
                                   "Brazil", "Brazil", "China", "China"),
                                year = c(1999, 2000, 1999, 2000, 1999, 2000),
                                rate = c("745/19987071", "2666/20595360",
                                    "37737/172006362", "80488/174504898",
                                    "212258/1272915272", "213766/1280428583"))
diseases

##       country year              rate
## 1 Afghanistan 1999      745/19987071
## 2 Afghanistan 2000     2666/20595360
## 3      Brazil 1999   37737/172006362
## 4      Brazil 2000   80488/174504898
## 5       China 1999 212258/1272915272
## 6       China 2000 213766/1280428583
```

Write a command involving `separate()` that separates the `rate` column into two columns named `cases` and `population`.

5.4 Iteration Using `for()` and `"dplyr"`'s `do()`

- Sometimes we need to repeatedly execute, i.e. ***iterate***, a set of R commands, each time changing one or more of the values used in the commands. ***Looping*** is a way of **iterating** the commands.
- Loops are usually implemented using:

```
for()      # Iterate a set of statements a specified number of times
```

- A special case of **iteration** is applying the same function repeatedly, each time on a different **group** within a **grouped** data frame (as returned by `"dplyr"`'s `group_by()`).

  The following function (from the `"dplyr"` package) is useful in this regard.

```
do()       # Apply a function repeatedly, each time on a different
           # group within a grouped data frame as returned by
           # group_by().
```

5.4.1 Iteration Using a `for()` Loop

- As a simple (but not useful) example, the following sequence of **five commands** prints the numbers $1^2, 2^2, ..., 5^2$ to the console (output not shown):

```
print(1^2)
print(2^2)
print(3^2)
print(4^2)
print(5^2)
```

  We can achieve the **same result** more succinctly using a `for()` loop by typing:

```
for(i in 1:5) {            # i takes the values 1, 2, 3, 4, 5 in succession.
  print(i^2)               # The print() statement is executed 5 times,
}                          # once for each value of i.

## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25
```

  Above, `i` takes the values 1, 2, 3, 4, 5 in succession, and the command `print(i^2)` is executed **five times**, once for each value of `i`.

- The general form of a `for()` loop is:

```
for(var in seq) {
    statement1
    statement2
        .
```

```
        .
        .
    statementq
}
```

where `seq` is a vector, usually of the form `1:n`, `var` (whose name you're free to change) takes values `1`, `2`, ..., `n` sequentially, each time triggering another iteration of the loop during which `statements 1` through `q` are executed.

The `statements` usually involve the variable `var`.

More generally, `seq` can be *any* vector, and `var` takes the values `seq[1]`, `seq[2]`, ..., `seq[length(seq)]`.

- For single-statement loops, we can omit the curly brackets `{ }` as long as we put the entire `for()` loop on one line, like this:

```
for(var in seq) statement1
```

Section 5.4 Exercises

Exercise 11 Guess how many times `"Good Sport"` will be printed to the screen in the following set of commands. Then check your answer.

```
for(i in 1:5) {
    print("Good Sport")
  }
```

Exercise 12 The sequence of values we iterate over doesn't have to be of the form `1:n`. Guess what will be printed to the screen in the following set of commands. Then check your answer.

```
x <- c(2, 4, 6, 8)

for(i in x) {
    print(i^2)
  }
```

Exercise 13 The sum of squares

$$\sum_{i=1}^{10} i^2 = 1^2 + 2^2 + \cdots + 10^2$$

can be computed using a `for()` loop by typing:

```
sum.sq <- 0

for(i in 1:10) {
    sum.sq <- sum.sq + i^2
}
```

Why is it necessary to make the assignment `sum.sq <- 0` *before* entering the loop? What would happen if `sum.sq <- 0` wasn't there? Try it (but remove `sum.sq` from your Workspace if it's there):

```
rm(sum.sq)

for(i in 1:10) {
    sum.sq <- sum.sq + i^2
}
```

Hint: Notice that `sum.sq` appears on *both* sides of the assignment statement in the loop, and R attempts to evaluate the *right* side *before* making the assignment.

Exercise 14

a) What does the following loop do?

```
num.sq <- rep(NA, 10)      # Pre-allocate a 10-element vector

for(i in 1:10) {
    num.sq[i] <- i^2
}
```

b) Loops are relatively **slow** to execute in R. It's advisable to *avoid* using loops whenever possible, and instead use the *vectorized* property of R's built-in functions or one of the `apply()` functions (`apply()`, `sapply()`, etc.).

   Can you think of a way to create the `num.sq` vector *without* using a loop? **Hint**: The exponentiation operator `^` is *vectorized*. Report your R command(s).

5.4.2 Iteration Over Groups Using "dplyr"'s do()

- We'll work with the `sleepstudy` data set from the `"lme4"` package.

Data Set: sleepstudy

The `sleepstudy` data set (in the `"lme4"` package) contains data on the average reaction time per day for subjects in a sleep deprivation study (Belenky et al. 2003). On day 0 the subjects had their normal amount of sleep. Starting that

night they were restricted to 3 hours of sleep per night. The response variable, `Reaction`, represents average reaction times in milliseconds (ms) on a series of tests given each `Day` to each `Subject`.

The three variables are:

`Reaction`	Average reaction time (milliseconds).
`Days`	Days into the study (0-9)
`Subject`	Subject ID number.

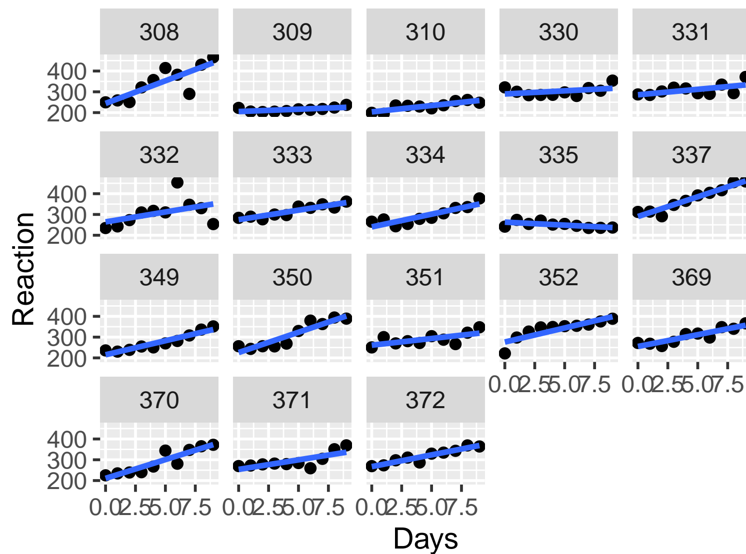
- Here are the data:

```
library(lme4)
head(sleepstudy)

##   Reaction Days Subject
## 1 249.5600    0     308
## 2 258.7047    1     308
## 3 250.8006    2     308
## 4 321.4398    3     308
## 5 356.8519    4     308
## 6 414.6901    5     308
```

- Here's a **faceted plot** of the data by `Subject`:

```
ggplot(data = sleepstudy, mapping = aes(x = Days, y = Reaction)) +
  facet_wrap(facets = ~ Subject) +
  geom_point() +
  geom_smooth(method = lm, se = FALSE)
```



(The specification `method = lm` in `geom_smooth()` fits a a "**linear model**", i.e. **straight line**, to the data.)

- We can fit a "**linear model**" to a set of data using the built-in, `base` R `lm()` function, which also reports the **equation** of the fitted line.

```
lm()          # Carry out a linear regression analysis by fitting a
              # linear model to a data set.
summary()     # Summarize the results of the regression analysis.
```

To obtain the **equations** of the lines shown in the **faceted plot** above, we need to apply `lm()` separately to each `Subject`'s data.

We can do this using `"dplyr"`'s `do()` function with the `sleepstudy` data, **grouped** by `Subject`, by typing:

```
by_subject <- group_by(.data = sleepstudy, Subject)
```

```
models <- do(.data = by_subject, mod = lm(Reaction ~ Days, data = .))
models

## Source: local data frame [18 x 2]
## Groups: <by row>
##
## # A tibble: 18 x 2
##    Subject mod
##  * <fct>   <list>
##  1 308     <lm>
```

```
##  2 309     <lm>
##  3 310     <lm>
##  4 330     <lm>
##  5 331     <lm>
##  6 332     <lm>
##  7 333     <lm>
##  8 334     <lm>
##  9 335     <lm>
## 10 337     <lm>
## 11 349     <lm>
## 12 350     <lm>
## 13 351     <lm>
## 14 352     <lm>
## 15 369     <lm>
## 16 370     <lm>
## 17 371     <lm>
## 18 372     <lm>
```

`do()` applies a function (`lm()` above) separately to each **group** in a *grouped* data frame (such as `by_subject` above).

In `lm()`, the expression `Reaction ~ Days` (an R ***formula***) indicates that `Reaction` is the ***y*** **variable** and `Days` is the ***x*** **variable**.

The '`data = .`' means "use the current group's data" as `do()` iterates over the **groups** (`Subjects` above).

`do()` returns a data frame. The first column will be the **group** labels, the second will be a list-column whose elements are the returned values of the function that's applied to the **groups** (`lm()` above).

We can look at the **equation** of the fitted line for, say, `Subject 330` (the 4th subject in the study) by typing:

```
models$mod[[4]]       # Gets the line for the 4th Subject.

##
## Call:
## lm(formula = Reaction ~ Days, data = .)
##
## Coefficients:
## (Intercept)         Days
##     289.685        3.008
```

The ***y-intercept*** is `289.685` and the ***slope*** is `3.008`, so the ***equation*** of the line is

$$Y = 289.685 + 3.008X.$$

- More examples on fitting linear models to each **group** in a grouped data frame can be found in the help file for `do()`:

```
? do
```

Section 5.4 Exercises

Exercise 15 Using the `sleepstudy` data (from the `"lme4"` package), use `do()` with `lm()` to fit lines separately to each `Subject`, with `Days` as the x variable and `Response` as the y variable by typing:

```
library(lme4)          # Contains the sleepstudy data set.

by_subject <- group_by(.data = sleepstudy, Subject)
```

```
models <- do(.data = by_subject, mod = lm(Reaction ~ Days, data = .))
models
```

What's the equation of the fitted line for `Subject 371` (the 17th subject in the study)?