Data selection

This lesson covers removing rows, removing columns, reordering columns, removing missing data, and reordering data rows. In the era of big data, you often have too much to look at, so limiting your data to what you need can greatly speed up your work.

Subsetting rows and columns

Example, we will use the iris dataset : measurements of sepal length and width and petal length and width for three species of iris.

```
[82] library("ggplot2")
     summary(iris)
```

```
library("ggplot2")
summary(iris)
```

```
 Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
Median :5.800   Median :3.000   Median :4.350   Median :1.300
Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
      Species
setosa    :50
versicolor:50
virginica :50
```

```
head(iris)
```

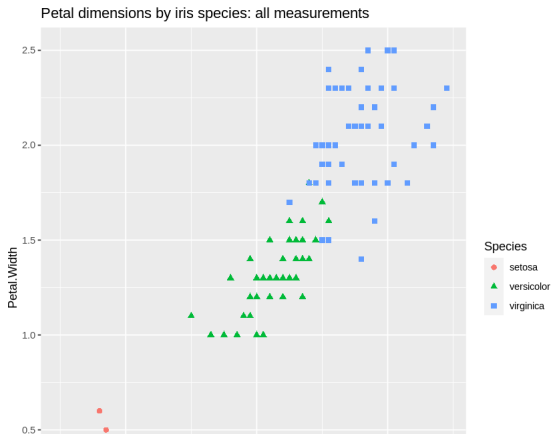
```
head(iris)
```

A data.frame: 6 × 5

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
	<dbl>	<dbl>	<dbl>	<dbl>	<fct>
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5.0	3.6	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa

```
[84] ggplot(iris,
     aes(x = Petal.Length, y = Petal.Width,
     shape = Species, color = Species)) +
     geom_point(size =2 ) +
     ggtitle("Petal dimensions by iris species: all measurements")
```

```
ggplot(iris,
aes(x = Petal.Length, y = Petal.Width,
shape = Species, color = Species)) +
geom_point (size =2 ) +
ggtitle("Petal dimensions by iris species: all measurements")
```



SCENARIO

Suppose we are assigned to generate a report on only petal length and petal width, by iris species, for irises where the petal length is greater than 2. To accomplish this, we need to select a subset of columns (variables) or a subset of rows (instances) from a data frame.

SOLUTION 1: BASE R

```
[85] columns_we_want <- c("Petal.Length", "Petal.Width", "Species")
     rows_we_want <- iris$Petal.Length > 2
```

```
columns_we_want <- c("Petal.Length", "Petal.Width", "Species")
rows_we_want <- iris$Petal.Length > 2
```

```
[86] # before
     head(iris)
```

```
#before
head(iris)
```

A data.frame: 6 × 5

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
	<dbl>	<dbl>	<dbl>	<dbl>	<fct>
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5.0	3.6	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa

```
[87] iris_base <- iris[rows_we_want, columns_we_want, drop = FALSE]
     # after
     head(iris_base)
```

```
iris_base <- iris [rows_we_want, columns_we_want, drop = FALSE]
# after
head(iris_base)
```

A data.frame: 6 × 3

	Petal.Length	Petal.Width	Species
	<dbl>	<dbl>	<fct>
51	4.7	1.4	versicolor
52	4.5	1.5	versicolor

SOLUTION 2: DATA.TABLE

Row and column selection in data.table is performed similarly to base R. data .table uses a very powerful set of index notations. In this case, we use a .. notation to tell data.table that we are using the second index position to specify column names (and not to specify calculations, as we will demonstrate later).

```
[89] # Converts to data.table class
     # to get data.table semantics
     library("data.table")
     iris_data.table <- as.data.table(iris)
     columns_we_want <- c("Petal.Length", "Petal.Width", "Species")
     rows_we_want <- iris_data.table$Petal.Length > 2
     iris_data.table <- iris_data.table[rows_we_want , ..columns_we_want]
     head(iris_data.table)
```

```
# Converts to data.table class
# to get data.table semantics
library("data.table")
iris_data.table <- as.data.table(iris)
columns_we_want <- c("Petal.Length", "Petal.Width", "Species")
rows_we_want <- iris_data.table$Petal.Length > 2
iris_data.table <- iris_data.table [rows_we_want , ..columns_we_want]
head(iris_data.table)
```

A data.table: 6 × 3

Petal.Length	Petal.Width	Species
<dbl>	<dbl>	<fct>
4.7	1.4	versicolor
4.5	1.5	versicolor
4.9	1.5	versicolor
4.0	1.3	versicolor
4.6	1.5	versicolor
4.5	1.3	versicolor

The advantage of data.table is that it is the fastest and most memory efficient solution for data wrangling in R at a wide range of scales. data.table has a very helpful FAQ, and there is a nice cheat sheet:

- https://cran.r-project.org/web/packages/data.table/vignettes/datatable-faq .html
- https://www.datacamp.com/community/tutorials/data-table-cheat-sheet

SOLUTION 3: DPLYR

The dplyr solution is written in terms of select and filter:

- dplyr::select to select desired columns
- dplyr::filter to select desired rows

It is traditional to chain dplyr steps with the magrittr pipe operator %>%, but assigning to temporary variables works just as well. While teaching here, we'll use explicit dot notation, where the data pipeline is written as iris %>% select(., column) instead of the more-common implicit first-argument notation (iris %>% select(column)).

```
[90] library("dplyr")
     iris_dplyr <- iris %>%
     select(.,
     Petal.Length, Petal.Width, Species) %>%
     filter(.,
     Petal.Length > 2)
     head(iris_dplyr)
```

```
library("dplyr")
iris_dplyr <- iris %>%
select(.,
Petal.Length, Petal.Width, Species) %>%
filter(.,
Petal.Length > 2)
head(iris_dplyr)
```

A data.frame: 6 × 3

	Petal.Length	Petal.Width	Species
	<dbl>	<dbl>	<fct>
1	4.7	1.4	versicolor
2	4.5	1.5	versicolor
3	4.9	1.5	versicolor
4	4.0	1.3	versicolor
5	4.6	1.5	versicolor
6	4.5	1.3	versicolor

The advantage of dplyr is the emphasis of data processing as a sequence of operations broken down into a visible pipeline. There is a nice cheat sheet for dplyr available from https://www.rstudio.com/wpcontent/uploads/2015/02/data-wrangling-cheatsheet.pdf. Cheat sheets are always going to be a bit brief, so the sheet will become very useful after you have tried a few examples.

Removing records with incomplete data

Example: the msleep dataset of sleep times of animals with different characteristics. In this dataset, several rows have missing values. An additional goal of this example is to familiarize you with a number of common practice datasets.

```
[92] library("ggplot2")
     data(msleep)
     str(msleep)
```

```
library("ggplot2")
data(msleep)
str(msleep)
```

```
tibble [83 × 11] (S3: tbl_df/tbl/data.frame)
 $ name        : chr [1:83] "Cheetah" "Owl monkey" "Mountain beaver" "Greater short-tailed shrew" ...
 $ genus       : chr [1:83] "Acinonyx" "Aotus" "Aplodontia" "Blarina" ...
 $ vore        : chr [1:83] "carni" "omni" "herbi" "omni" ...
 $ order       : chr [1:83] "Carnivora" "Primates" "Rodentia" "Soricomorpha" ...
 $ conservation: chr [1:83] "lc" NA "nt" "lc" ...
 $ sleep_total : num [1:83] 12.1 17 14.4 14.9 4 14.4 8.7 7 10.1 3 ...
 $ sleep_rem   : num [1:83] NA 1.8 2.4 2.3 0.7 2.2 1.4 NA 2.9 NA ...
 $ sleep_cycle : num [1:83] NA NA NA 0.133 0.667 ...
 $ awake       : num [1:83] 11.9 7 9.6 9.1 20 9.6 15.3 17 13.9 21 ...
 $ brainwt     : num [1:83] NA 0.0155 NA 0.00029 0.423 NA NA NA 0.07 0.0982 ...
 $ bodywt      : num [1:83] 50 0.48 1.35 0.019 600 ...
```

```
[93] summary(msleep)
```

```
summary(msleep)
```

```
      name              genus               vore              order          
 Length:83          Length:83          Length:83          Length:83         
 Class :character   Class :character   Class :character   Class :character  
 Mode  :character   Mode  :character   Mode  :character   Mode  :character  
```

```
 conservation        sleep_total      sleep_rem      sleep_cycle     
 Length:83          Min.   : 1.90   Min.   :0.100   Min.   :0.1167  
 Class :character   1st Qu.: 7.85   1st Qu.:0.900   1st Qu.:0.1833  
 Mode  :character   Median :10.10   Median :1.500   Median :0.3333  
                    Mean   :10.43   Mean   :1.875   Mean   :0.4396  
                    3rd Qu.:13.75   3rd Qu.:2.400   3rd Qu.:0.5792  
                    Max.   :19.90   Max.   :6.600   Max.   :1.5000  
                                    NA's   :22      NA's   :51      
     awake          brainwt            bodywt        
 Min.   : 4.10   Min.   :0.00014   Min.   :   0.005  
 1st Qu.:10.25   1st Qu.:0.00290   1st Qu.:   0.174  
 Median :13.90   Median :0.01240   Median :   1.670  
 Mean   :13.57   Mean   :0.28158   Mean   : 166.136  
 3rd Qu.:16.15   3rd Qu.:0.12550   3rd Qu.:  41.750  
```

SCENARIO

We have been asked to build an extract of the msleep data that has no missing values.

Base R solution

- complete.cases() returns a vector with one entry for each row of the data frame, which is TRUE if and only if the row has no missing entries.
- na.omit() performs the whole task in one step.

```
[94] clean_base_1 <- msleep[complete.cases(msleep), , drop = FALSE]
     summary(clean_base_1)
```

```
clean_base_1 <- msleep[complete.cases(msleep), , drop=FALSE]
summary(clean_base_1)
```

```
      name              genus               vore              order          
 Length:20          Length:20          Length:20          Length:20         
 Class :character   Class :character   Class :character   Class :character  
 Mode  :character   Mode  :character   Mode  :character   Mode  :character  
```

```
 conservation        sleep_total       sleep_rem      sleep_cycle     
 Length:20          Min.   : 2.900   Min.   :0.600   Min.   :0.1167  
 Class :character   1st Qu.: 8.925   1st Qu.:1.300   1st Qu.:0.1792  
 Mode  :character   Median :11.300   Median :2.350   Median :0.2500  
                    Mean   :11.225   Mean   :2.275   Mean   :0.3458  
                    3rd Qu.:13.925   3rd Qu.:3.125   3rd Qu.:0.4167  
                    Max.   :19.700   Max.   :4.900   Max.   :1.0000  
     awake          brainwt            bodywt        
 Min.   : 4.30   Min.   :0.00014   Min.   :  0.0050  
 1st Qu.:10.07   1st Qu.:0.00115   1st Qu.:  0.0945  
 Median :12.70   Median :0.00590   Median :  0.7490  
 Mean   :12.78   Mean   :0.07882   Mean   : 72.1177  
 3rd Qu.:15.07   3rd Qu.:0.03670   3rd Qu.:  6.1250  
 Max.   :21.10   Max.   :0.65500   Max.   :600.0000  
```

```
[95] nrow(clean_base_1)
```

```
nrow(clean_base_1)
```

```
20
```

```
[96] clean_base_2 = na.omit(msleep)
     nrow(clean_base_2)
```

```
clean_base_2 = na.omit (msleep)
```

```
nrow(clean_base_2)
```

```
20
```

```
[97] nrow(clean_base_2)
```

```
nrow(clean_base_2)
```

```
20
```

data.table solution

The complete.cases() solution also works with data.table:

```
[98] library("data.table")
     msleep_data.table <- as.data.table(msleep)
     clean_data.table = msleep_data.table[complete.cases(msleep_data.table), ]
     nrow(clean_data.table)
```

```
 library("data.table")
msleep_data.table <- as.data.table (msleep)
clean_data.table = msleep_data.table [complete.cases(msleep_data.table), ]
nrow(clean_data.table)
```

```
20
```

dplyr solution

dplyr::filter can also be used with complete.cases(). With magrittr pipe notation, a . is taken to mean the item being piped. So we can use . to refer to our data multiple times conveniently, such as telling the dplyr::filter to use the data both as the object to be filtered and as the object to pass to complete .cases().

```
[99] library("dplyr")
     clean_dplyr <- msleep %>%
     filter(., complete.cases(.))
     nrow(clean_dplyr)
```

```
library("dplyr")
clean_dplyr <- msleep %>%
filter(., complete.cases(.))
nrow(clean_dplyr)
```

```
20
```

Ordering rows

In this section, we want to sort or control what order our data rows are in. Perhaps the data came to us unsorted, or sorted for a purpose other than ours.

SCENARIO

We are asked to build a running or cumulative sum of sales by time, but the data came to us out of order:

```
# Uses wrapr::build_frame to type data in directly in
# legible column order
# install.packages("wrapr")
# library("data.table") # data.table before wrapr to avoid := contention
# suppressPackageStartupMessages(library("dplyr"))
library("wrapr")
# library(dplyr)
purchases <- wrapr::build_frame(
"day", "hour", "n_purchase" |
1 ,9 ,5 |
2 ,9 ,3 |
2 , 11 , 5 |
1 , 13 , 1 |
2 , 13 , 3 |
1 , 14 , 1 )
purchases
```

```
# Uses wrapr::build frame to type data in directly in
# legible column order
# install.packages("wrapr")
# library("data.table") # data.table before wrap to avoid : = contention
# suppressPackageStartupMessages(library("dplyr"))
install.packages("wrapr")
library("wrapr")
# library(dplyr)
purchases <- wrapr::build_frame(
"day", "hour", "n_purchase" |
1 ,9 ,5 |
2 ,9 ,3 |
2 , 11 , 5 |
1 , 13 , 1 |
2 , 13 , 3 |
1 , 14 , 1 )
purchases
```

```
Installing package into ‘/usr/local/lib/R/site-library’
(as ‘lib’ is unspecified)
```

A data.frame: 6 × 3

day <dbl>	hour <dbl>	n_purchase <dbl>
1	9	5
2	9	3
2	11	5
1	13	1
2	13	3
1	14	1

PROBLEM

Reorder the rows by day and then hour and compute a running sum.

```
# Base R solution
order_index <- with(purchases, order(day, hour))
purchases_ordered <- purchases[order_index, , drop = FALSE]
purchases_ordered
```

```
# Base R solution
order_index <- with(purchases, order (day, hour))
purchases_ordered <- purchases [order_index, , drop=FALSE]
purchases_ordered
```

A data.frame: 6 × 3

	day	hour	n_purchase
	<dbl>	<dbl>	<dbl>
1	1	9	5
4	1	13	1
6	1	14	1
5	2	13	3

```
purchases_ordered$running_total <- cumsum(purchases_ordered$n_purchase)
purchases_ordered
```

purchases_ordered$running_total <- cumsum(purchases_ordered$n_purchase)
purchases_ordered

A data.frame: 6 × 4

	day	hour	n_purchase	running_total
	<dbl>	<dbl>	<dbl>	<dbl>
1	1	9	5	5
4	1	13	1	6
6	1	14	1	7
2	2	9	3	10
3	2	11	5	15
5	2	13	3	18

```
# data.table solution
library("data.table")
DT_purchases <- as.data.table(purchases)
order_cols <- c("day", "hour")
setorderv(DT_purchases, order_cols)
DT_purchases[ , running_total := cumsum(n_purchase)]
print(DT_purchases)
```

# data. table solution
library("data.table")
DT_purchases <- as.data.table(purchases)
order_cols <- c("day", "hour")
setorderv(DT_purchases, order_cols)
DT_purchases[, running_total := cumsum(n_purchase)]
print(DT_purchases)

```
   day hour n_purchase running_total
1:   1    9          5             5
2:   1   13          1             6
3:   1   14          1             7
4:   2    9          3            10
5:   2   11          5            15
6:   2   13          3            18
```

:= AND [] Operations that alter data in place (such as :=) annotate the result to suppress printing. This is important, as often you are working with large structures and do not want intermediate data to print. [] is a no-operation that as a side effect restores printing.

setorderv() reorders data in place and takes a list of ordering column names to specify the order. This is much more convenient than the base R solution that takes multiple ordering columns as multiple arguments. wrapr::orderv() tries to bridge this gap by allowing the user to specify ordering constraints with a list of columns (column values, not column names).

dplyr solution

dplyr uses the word arrange to order data, and mutate to add a new column:

```
library("dplyr")
res <- purchases %>%
arrange(., day, hour) %>%
mutate(., running_total = cumsum(n_purchase))
print(res)
```

```
library("dplyr")
res <- purchases %>%
arrange(., day, hour) %>%
mutate(., running_total = cumsum(n_purchase))
print (res)
```

```
  day hour n_purchase running_total
1   1    9          5             5
2   1   13          1             6
3   1   14          1             7
4   2    9          3            10
5   2   11          5            15
6   2   13          3            18
```

ADVANCED USE OF ORDERING

For our advanced example, suppose we want the cumulative sum of sales to be perday—that is, to reset the sum at the start of each day. **Base R solution**

This easiest base R solution is a split and recombine strategy:

```
# First sorts the data
order_index <- with(purchases, order(day, hour))
purchases_ordered <- purchases[order_index, , drop = FALSE]
# Now splits the data into a list of groups
data_list <- split(purchases_ordered, purchases_ordered$day)
# Applies the cumsum to each group
data_list <- lapply(
data_list,
function(di) {
di$running_total <- cumsum(di$n_purchase)
di
})
```

```
# First sorts the data
order_index <- with(purchases, order (day, hour))
purchases_ordered <- purchases [order_index, , drop = FALSE]
# Now splits the data into a list of groups
data_list <- split(purchases_ordered, purchases_ordered$day)
# Applies the cumsum to each group
data_list <- lapply(
data_list,
function(di) {
di$running_total <- cumsum(di$n_purchase)
di
})
```

```
# Puts the results back
# together into a single
# data.frame
purchases_ordered <- do.call(base::rbind, data_list)
# R often keeps annotations in rownames().
# In this case, it is storing the original row
# numbers of the pieces we are assembling.
# This can confuse users when printing, so it
# is good practice to remove these
# annotations, as we do here.
rownames(purchases_ordered) <- NULL
purchases_ordered
```

```
# Puts the results back
# together into a single
# data.frame
purchases_ordered <- do.call(base::rbind, data_list)
```

```
# R often keeps annotations in rownames ().
# In this case, it is storing the original row
# numbers of the pieces we are assembling.
# This can confuse users when printing, so it
# is good practice to remove these
# annotations, as we do here.
rownames(purchases_ordered)
purchases_ordered
```

'1.1' · '1.4' · '1.6' · '2.2' · '2.3' · '2.5'

A data.frame: 6 × 4

	day	hour	n_purchase	running_total
	<dbl>	<dbl>	<dbl>	<dbl>
1.1	1	9	5	5
1.4	1	13	1	6
1.6	1	14	1	7
2.2	2	9	3	3
2.3	2	11	5	8
2.5	2	13	3	11

data.table solution

The data.table solution is particularly concise. We order the data and then tell data.table to calculate the new running sum per-group with the by argument. The idea that the grouping is a property of the calculation, and not a property of the data, is similar to SQL and helps minimize errors

:= VERSUS = In data.table, := means “assign in place”—it is used to alter or create a column in the incoming data.table. Conversely, = is used to mean “create in new data.table,” and we wrap these sorts of assignments in a .() notation so that column names are not confused with arguments to data.table.

```
library("data.table")
# new copy for result solution
DT_purchases <- as.data.table(purchases)[order(day, hour),
.(hour = hour,
n_purchase = n_purchase,
running_total = cumsum(n_purchase)),
by = "day"] #Adding the by keyword converts the calculation into
# a per-group calculation.

# First solution: result is a second copy of
# the data .(=) notation. Only columns used
# in the calculation (such as day) and those
# explicitly assigned to are in the result.
print(DT_purchases)
```

```
library("data.table")
# new copy for result solution
DT_purchases <- as.data.table(purchases)[order(day, hour),
.(hour = hour,
n_purchase = n_purchase,
running_total = cumsum(n_purchase)),
by = "day"] #Adding the by keyword converts the calculation into
# a per-group calculation.
# First solution: result is a second copy of
# the data (= notation. Only columns used
# in the calculation (such as day) and those
# explicitly assigned to are in the result.
print(DT_purchases)
```

```
   day hour n_purchase running_total
1:   1    9          5             5
2:   1   13          1             6
3:   1   14          1             7
4:   2    9          3             3
5:   2   11          5             8
6:   2   13          3            11
```

Second solution: result is computed in place by ordering the table before the grouped calculation.

```
# in-place solution
DT_purchases <- as.data.table(purchases)
order_cols <- c("day", "hour")
setorderv(DT_purchases, order_cols)
DT_purchases[ , running_total := cumsum(n_purchase), by = day]
# print(DT_purchases)
```

```
# in-place solution
DT_purchases <- as.data.table(purchases)
order_cols <- c("day", "hour")
setorderv(DT_purchases, order_cols)
DT_purchases[ , running_total:= cumsum(n_purchase), by = day]
# print(DT_purchases)
print(DT_purchases)
```

```
        day hour n_purchase running_total
    1:   1    9          5             5
    2:   1   13          1             6
    3:   1   14          1             7
    4:   2    9          3             3
    5:   2   11          5             8
    6:   2   13          3            11
```

Third solution: result is in the same order as the original table, but the cumulative sum is computed as if we sorted the table, computed the grouped running sum, and then returned the table to the original order.

```
# don't reorder the actual data variation!
DT_purchases <- as.data.table(purchases)
DT_purchases[order(day, hour),
`:=`(hour = hour,
n_purchase = n_purchase,
running_total = cumsum(n_purchase)),
by = "day"]
# print(DT_purchases)
```

```
# don't reorder the actual data variation!
DT_purchases <- as.data.table(purchases)
DT_purchases[order(day, hour),
`:=`(hour = hour,
n_purchase = n_purchase,
running_total = cumsum(n_purchase)),
by="day"]
# print(DT_ purchases)
print(DT_purchases)
```

```
        day hour n_purchase running_total
    1:   1    9          5             5
    2:   2    9          3             3
    3:   2   11          5             8
    4:   1   13          1             6
    5:   2   13          3            11
    6:   1   14          1             7
```

SEQUENCING DATA.TABLE OPERATIONS

Sequencing data.table operations is achieved either by writing in-place operations one after the other (as we did in these examples) or by starting a new open-[right after a close-] for operations that create new copies (this is called method chaining and is equivalent to using a pipe operator).

dplyr solution

The dplyr solution works because the command mutate() (which we will discuss in the next section) works per-group if the data is grouped. We can make the data grouped by using the group_by() command :

```
library("dplyr")
res <- purchases %>%
arrange(., day, hour) %>%
group_by(., day) %>%
mutate(., running_total = cumsum(n_purchase)) %>%
ungroup(.)
print(res)
```


```
library("dplyr")
res <- purchases %>%
arrange(., day, hour) %>%
group_by(., day) %>%
mutate(., running_total = cumsum(n_purchase)) %>%
ungroup(.)
print(res)
```


```
# A tibble: 6 × 4
    day  hour n_purchase running_total
  <dbl> <dbl>      <dbl>         <dbl>
1     1     9          5             5
2     1    13          1             6
3     1    14          1             7
4     2     9          3             3
5     2    11          5             8
6     2    13          3            11
```

UNGROUP()

In dplyr it is important to always ungroup your data when you are done performing per-group operations. This is because the presence of a dplyr grouping annotation can cause many downstream steps to calculate unexpected and incorrect results. We advise doing this even after a summarize() step, as summarize() removes one key from the grouping, leaving it unclear to a code reader if the data remains grouped or not.

✓ 0s completed at 10:58 PM