Handling massive data

“Processing large dataset with R”

Data Frames

The *data frame* is a key data structure in statistics and in R. The basic structure of a data frame is that there is **one observation per row and each column represents a variable, a measure, feature, or characteristic of that observation**. R has an internal implementation of data frames that is likely the one you will use most often. However, there are packages on CRAN that implement data frames via things like relational databases that allow you to operate on very very large data frames (but we won't discuss them here).

Given the importance of managing data frames, it's important that we have good tools for dealing with them. In previous chapters we have already discussed some tools like the subset() function and the use of [and $ operators to extract subsets of data frames. However, other operations, like **filtering, re-ordering, and collapsing, can often be tedious operations in R** whose syntax is not very intuitive. The dplyr package is designed to mitigate a lot of these problems and to provide a highly optimized set of routines specifically for dealing with data frames.

The dplyr Package

The dplyr package was developed by Hadley Wickham of RStudio and is an optimized and distilled version of his plyr package. **The dplyr package does not provide any “new” functionality to R** per se, in the sense that everything dplyr does could already be done with base R, but it ***greatly*** **simplifies existing functionality** in R.

One important contribution of the dplyr package is that it provides a **“grammar”** (in particular, verbs) for data manipulation and for operating on data frames. With this grammar, you can sensibly communicate what it is that you are doing to a data frame that other people can understand (assuming they also know the grammar). This is useful because it provides an abstraction for data manipulation that previously did not exist. Another useful contribution is that the dplyr functions are **very fast**, as many key operations are coded in C++.

dplyr Grammar

Some of the key “verbs” provided by the dplyr package are

- select: return a subset of the columns of a data frame, using a flexible notation
- filter: extract a subset of rows from a data frame based on logical conditions
- arrange: reorder rows of a data frame
- rename: rename variables in a data frame
- mutate: add new variables/columns or transform existing variables
- summarise / summarize: generate summary statistics of different variables in the data frame, possibly within strata
- %>%: the “pipe” operator is used to connect multiple verb actions together into a pipeline

The dplyr package as a number of its own data types that it takes advantage of. For example, there is a handy print method that prevents you from printing a lot of data to the console. Most of the time, these additional data types are transparent to the user and do not need to be worried about.

Common dplyr Function Properties

All of the functions that we will discuss will have a few common characteristics. In particular,

1. The first argument is a data frame.

2. The subsequent arguments describe what to do with the data frame specified in the first argument, and you can refer to columns in the data frame directly without using the $ operator (just use the column names).

3. The return result of a function is a new data frame

4. Data frames must be properly formatted and annotated for this to all be useful. In particular, the data must be tidy. In short, there should be one observation per row, and each column should represent a feature or characteristic of that observation.

```
> install.packages("dplyr")
> library(dplyr)
```

select()

For the examples we will be using a dataset containing air pollution and temperature data for the city of Chicago in the U.S.

```
> chicago <- read _csv("chicago.rds")
> dim(chicago)
[1] 6940    8
> str(chicago)
'data.frame':   6940 obs. of  8 variables:
 $ city      : chr  "chic" "chic" "chic" "chic" ...
 $ tmpd       : num  31.5 33 33 29 32 40 34.5 29 26.5 32.5 ...
 $ dptp      : num  31.5 29.9 27.4 28.6 28.9 ...
 $ date      : Date, format: "1987-01-01" "1987-01-02" ...
 $ pm25tmean2: num  NA NA NA NA NA NA NA NA NA NA ...
 $ pm10tmean2: num  34 NA 34.2 47 NA ...
 $ o3tmean2  : num  4.25 3.3 3.33 4.38 4.75 ...
 $ no2tmean2 : num  20 23.2 23.8 30.4 30.3 ...
```

You can see some basic characteristics of the dataset with the dim() and str() f unctions.

select()

The select() function can be used to select columns of a data frame that you want to focus on. Often you'll have a large data frame containing "all" of the data, but any *given* analysis might only use a subset of variables or observations.
The select() function allows you to get the few columns you might need.
Suppose we wanted to take the first 3 columns only. There are a few ways to do this. We could for example use numerical indices. But we can also use the names directly.

```
> names(chicago)[1:3]
[1] "city" "tmpd" "dptp"
> subset <- select(chicago, city:dptp)
> head(subset)
  city tmpd   dptp
1 chic 31.5 31.500
2 chic 33.0 29.875
3 chic 33.0 27.375
4 chic 29.0 28.625
5 chic 32.0 28.875
6 chic 40.0 35.125
```

Note that the : normally cannot be used with names or strings, but inside the select() function you can use it to specify a range of variable names.

select()

You can also *omit* variables using the select() function by using the negative sign. With select() you can do

```
> select(chicago, -(city:dptp))
```

which indicates that we should include every variable *except* the variables city through dptp. The equivalent code in base R would be

```
> i <- match("city", names(chicago))
> j <- match("dptp", names(chicago))
> head(chicago[, -(i:j)])
```

Not super intuitive, right?

select()

The select() function also allows a special syntax that allows you to specify variable names based on patterns. So, for example, if you wanted to keep every variable that ends with a “2”, we could do

```
> subset <- select(chicago, ends_with("2"))
> str(subset)
'data.frame':   6940 obs. of  4 variables:
 $ pm25tmean2: num  NA NA NA NA NA NA NA NA NA NA ...
 $ pm10tmean2: num  34 NA 34.2 47 NA ...
 $ o3tmean2  : num  4.25 3.3 3.33 4.38 4.75 ...
 $ no2tmean2 : num  20 23.2 23.8 30.4 30.3 ...
```

Or if we wanted to keep every variable that starts with a “d”, we could do

```
> subset <- select(chicago, starts_with("d"))
> str(subset)
'data.frame':   6940 obs. of  2 variables:
 $ dptp: num  31.5 29.9 27.4 28.6 28.9 ...
 $ date: Date, format: "1987-01-01" "1987-01-02" ...
```

filter()

The filter() function is used to extract subsets of rows from a data frame. This function is similar to the existing subset() function in R but is quite a bit faster in my experience.
Suppose we wanted to extract the rows of the chicago data frame where the levels of PM2.5 are greater than 30 (which is a reasonably high level), we could do

```
> chic.f <- filter(chicago, pm25tmean2 > 30)
> str(chic.f)
'data.frame':   194 obs. of  8 variables:
 $ city      : chr  "chic" "chic" "chic" "chic" ...
 $ tmpd       : num  23 28 55 59 57 57 75 61 73 78 ...
 $ dptp      : num  21.9 25.8 51.3 53.7 52 56 65.8 59 60.3 67.1 ...
 $ date      : Date, format: "1998-01-17" "1998-01-23" ...
 $ pm25tmean2: num  38.1 34 39.4 35.4 33.3 ...
 $ pm10tmean2: num  32.5 38.7 34 28.5 35 ...
 $ o3tmean2  : num  3.18 1.75 10.79 14.3 20.66 ...
 $ no2tmean2 : num  25.3 29.4 25.3 31.4 26.8 ...
```

You can see that there are now only 194 rows in the data frame and the distribution of the pm25tmean2 values is.

```
> summary(chic.f$pm25tmean2)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
  30.05   32.12   35.04   36.63   39.53   61.50 
```

filter()

We can place an arbitrarily complex logical sequence inside of filter(), so we could for example extract the rows where PM2.5 is greater than 30 *and* temperature is greater than 80 degrees Fahrenheit.

```
> chic.f <- filter(chicago, pm25tmean2 > 30 & tmpd > 80)
> select(chic.f, date, tmpd, pm25tmean2)
       date tmpd pm25tmean2
1  1998-08-23   81   39.60000
2  1998-09-06   81   31.50000
3  2001-07-20   82   32.30000
4  2001-08-01   84   43.70000
5  2001-08-08   85   38.83750
6  2001-08-09   84   38.20000
7  2002-06-20   82   33.00000
8  2002-06-23   82   42.50000
9  2002-07-08   81   33.10000
10 2002-07-18   82   38.85000
11 2003-06-25   82   33.90000
12 2003-07-04   84   32.90000
13 2005-06-24   86   31.85714
14 2005-06-27   82   51.53750
15 2005-06-28   85   31.20000
16 2005-07-17   84   32.70000
17 2005-08-03   84   37.90000
```

Now there are only 17 observations where both of those conditions are met.

arrange()

The arrange() function is used to reorder rows of a data frame according to one of the variables/columns. Reordering rows of a data frame (while preserving corresponding order of other columns) is normally a pain to do in R.
The arrange() function simplifies the process quite a bit.
Here we can order the rows of the data frame by date, so that the first row is the earliest (oldest) observation and the last row is the latest (most recent) observation.

```
> chicago <- arrange(chicago, date)
```

We can now check the first few rows

```
> head(select(chicago, date, pm25tmean2), 3)
        date pm25tmean2
1 1987-01-01         NA
2 1987-01-02         NA
3 1987-01-03         NA
```

and the last few rows.

```
> tail(select(chicago, date, pm25tmean2), 3)
           date pm25tmean2
6938 2005-12-29    7.45000
6939 2005-12-30   15.05714
6940 2005-12-31   15.00000
```

Columns can be arranged in descending order too by using the special desc() operator.

```
> chicago <- arrange(chicago, desc(date))
```

arrange()

Looking at the first three and last three rows shows the dates in descending order.

```
> head(select(chicago, date, pm25tmean2), 3)
        date pm25tmean2
1 2005-12-31   15.00000
2 2005-12-30   15.05714
3 2005-12-29    7.45000
> tail(select(chicago, date, pm25tmean2), 3)
           date pm25tmean2
6938 1987-01-03         NA
6939 1987-01-02         NA
6940 1987-01-01         NA
```

rename()

Renaming a variable in a data frame in R is surprisingly hard to do! The rename() function is designed to make this process easier.
Here you can see the names of the first five variables in the chicago data frame.

```
> head(chicago[, 1:5], 3)
  city tmpd dptp       date pm25tmean2
1 chic   35 30.1 2005-12-31   15.00000
2 chic   36 31.0 2005-12-30   15.05714
3 chic   35 29.4 2005-12-29    7.45000
```

The dptp column is supposed to represent the dew point temperature adn the pm25tmean2 column provides the PM2.5 data. However, these names are pretty obscure or awkward and probably be renamed to something more sensible.

```
> chicago <- rename(chicago, dewpoint = dptp, pm25 =
pm25tmean2)
> head(chicago[, 1:5], 3)
  city tmpd dewpoint       date     pm25
1 chic   35     30.1 2005-12-31 15.00000
2 chic   36     31.0 2005-12-30 15.05714
3 chic   35     29.4 2005-12-29  7.45000
```

The syntax inside the rename() function is to have the new name on the left-hand side of the = sign and the old name on the right-hand side.

mutate()

The mutate() function exists to compute transformations of variables in a data frame. Often, you want to create new variables that are derived from existing variables and mutate() provides a clean interface for doing that.
For example, with air pollution data, we often want to *detrend* the data by subtracting the mean from the data. That way we can look at whether a given day's air pollution level is higher than or less than average (as opposed to looking at its absolute level).
Here we create a pm25detrend variable that subtracts the mean from the pm25 variable.

```
> chicago <- mutate(chicago, pm25detrend = pm25 - mean(pm25, na.rm = TRUE))
> head(chicago)
  city tmpd dewpoint       date     pm25 pm10tmean2  o3tmean2 no2tmean2
1 chic   35     30.1 2005-12-31 15.00000       23.5  2.531250  13.25000
2 chic   36     31.0 2005-12-30 15.05714       19.2  3.034420  22.80556
3 chic   35     29.4 2005-12-29  7.45000       23.5  6.794837  19.97222
4 chic   37     34.5 2005-12-28 17.75000       27.5  3.260417  19.28563
5 chic   40     33.6 2005-12-27 23.56000       27.0  4.468750  23.50000
6 chic   35     29.6 2005-12-26  8.40000        8.5 14.041667  16.81944
pm25detrend
1   -1.230958
2   -1.173815
3   -8.780958
4    1.519042
5    7.329042
6   -7.830958
```

mutate()

There is also the related transmute() function, which does the same thing as mutate() but then *drops all non-transformed variables*.
Here we detrend the PM10 and ozone (O3) variables.

```
> head(transmute(chicago,
+              pm10detrend = pm10tmean2 - mean(pm10tmean2,
na.rm = TRUE),
+              o3detrend = o3tmean2 - mean(o3tmean2, na.rm =
TRUE)))
  pm10detrend  o3detrend
1  -10.395206 -16.904263
2  -14.695206 -16.401093
3  -10.395206 -12.640676
4   -6.395206 -16.175096
5   -6.895206 -14.966763
6  -25.395206  -5.393846
```

Note that there are only two columns in the transmuted data frame.

group_by()

The group_by() function is used to generate summary statistics from the data frame within strata defined by a variable. For example, in this air pollution dataset, you might want to know what the average annual level of PM2.5 is. So the stratum is the year, and that is something we can derive from the date variable. In conjunction with the group_by() function we often use the summarize() function (or summarise() for some parts of the world).
The general operation here is a combination of splitting a data frame into separate pieces defined by a variable or group of variables (group_by()), and then applying a summary function across those subsets (summarize()).

First, we can create a year varible using as.POSIXlt().

```
> chicago <- mutate(chicago, year = as.POSIXlt(date)$year + 1900)
```

Now we can create a separate data frame that splits the original data frame by year.

```
> years <- group_by(chicago, year)
```

group_by()

Finally, we compute summary statistics for each year in the data frame with the summarize() function.

```
> summarize(years, pm25 = mean(pm25, na.rm = TRUE),
+           o3 = max(o3tmean2, na.rm = TRUE),
+           no2 = median(no2tmean2, na.rm = TRUE))
`summarise()` ungrouping output (override with `.groups` argument)
# A tibble: 19 x 4
    year  pm25    o3   no2
   <dbl> <dbl> <dbl> <dbl>
 1  1987 NaN    63.0  23.5
 2  1988 NaN    61.7  24.5
 3  1989 NaN    59.7  26.1
 4  1990 NaN    52.2  22.6
 5  1991 NaN    63.1  21.4
 6  1992 NaN    50.8  24.8
 7  1993 NaN    44.3  25.8
 8  1994 NaN    52.2  28.5
 9  1995 NaN    66.6  27.3
10  1996 NaN    58.4  26.4
11  1997 NaN    56.5  25.5
12  1998  18.3  50.7  24.6
13  1999  18.5  57.5  24.7
14  2000  16.9  55.8  23.5
15  2001  16.9  51.8  25.1
16  2002  15.3  54.9  22.7
17  2003  15.2  56.2  24.6
18  2004  14.6  44.5  23.4
19  2005  16.2  58.8  22.6
```

summarize() returns a data frame
with year as the first column, and then the
annual averages of pm25, o3, and no2.

group_by()

In a slightly more complicated example, we might want to know what are the average levels of ozone (o3) and nitrogen dioxide (no2) within quintiles of pm25. A slicker way to do this would be through a regression model, but we can actually do this quickly with group_by() and summarize().

First, we can create a categorical variable of pm25 divided into quintiles.

```
> qq <- quantile(chicago$pm25, seq(0, 1, 0.2), na.rm = TRUE)
> chicago <- mutate(chicago, pm25.quint = cut(pm25, qq))
```

Now we can group the data frame by the pm25.quint variable.

```
> quint <- group_by(chicago, pm25.quint)
```

Finally, we can compute the mean of o3 and no2 within quintiles of pm25.

```
> summarize(quint, o3 = mean(o3tmean2, na.rm = TRUE),
+         no2 = mean(no2tmean2, na.rm = TRUE))
`summarise()` ungrouping output (override with `.groups` argument)
# A tibble: 6 x 3
  pm25.quint     o3   no2
  <fct>       <dbl> <dbl>
1 (1.7,8.7]    21.7  18.0
2 (8.7,12.4]   20.4  22.1
3 (12.4,16.7]  20.7  24.4
4 (16.7,22.6]  19.9  27.3
5 (22.6,61.5]  20.3  29.6
6 <NA>         18.8  25.8
```

From the table, it seems there isn't a strong relationship between pm25 and o3, but there appears to be a positive correlation between pm25 and no2. More sophisticated statistical modeling can help to provide precise answers to these questions, but a simple application of dplyr functions can often get you most of the way there.

%>%

The pipeline operater %>% is very handy for stringing together multiple dplyr functions in a sequence of operations. Notice above that every time we wanted to apply more than one function, the sequence gets buried in a sequence of nested function calls that is difficult to read, i.e.

```
> third(second(first(x)))
```

This nesting is not a natural way to think about a sequence of operations. The %>% operator allows you to string operations in a left-to-right fashion, i.e.

```
> first(x) %>% second %>% third
```

%>%

Take the example that we just did in the last section where we computed the mean of o3 and no2 within quintiles of pm25. There we had to

1. create a new variable pm25.quint
2. split the data frame by that new variable
3. compute the mean of o3 and no2 in the sub-groups defined by pm25.quint

That can be done with the following sequence in a single R expression.

```
> mutate(chicago, pm25.quint = cut(pm25, qq)) %>%
+       group_by(pm25.quint) %>%
+       summarize(o3 = mean(o3tmean2, na.rm = TRUE),
+               no2 = mean(no2tmean2, na.rm = TRUE))
`summarise()` ungrouping output (override with `.groups`
argument)
# A tibble: 6 x 3
  pm25.quint     o3   no2
  <fct>       <dbl> <dbl>
1 (1.7,8.7]    21.7  18.0
2 (8.7,12.4]   20.4  22.1
3 (12.4,16.7]  20.7  24.4
4 (16.7,22.6]  19.9  27.3
5 (22.6,61.5]  20.3  29.6
6 <NA>         18.8  25.8
```

This way we don’t have to create a set of temporary variables along the way or create a massive nested sequence of function calls.

%>%

Another example might be computing the average pollutant level by month. This could be useful to see if there are any seasonal trends in the data.

Here we can see that o3 tends to be low in the winter months and high in the summer while no2 is higher in the winter and lower in the summer.

```
> mutate(chicago, month = as.POSIXlt(date)$mon + 1) %>%
+       group_by(month) %>%
+       summarize(pm25 = mean(pm25, na.rm = TRUE),
+               o3 = max(o3tmean2, na.rm = TRUE),
+               no2 = median(no2tmean2, na.rm = TRUE))
`summarise()` ungrouping output (override with `.groups`
argument)
# A tibble: 12 x 4
  month  pm25    o3   no2
  <dbl> <dbl> <dbl> <dbl>
 1     1  17.8  28.2  25.4
 2     2  20.4  37.4  26.8
 3     3  17.4  39.0  26.8
 4     4  13.9  47.9  25.0
 5     5  14.1  52.8  24.2
 6     6  15.9  66.6  25.0
 7     7  16.6  59.5  22.4
 8     8  16.9  54.0  23.0
 9     9  15.9  57.5  24.5
10    10  14.2  47.1  24.2
11    11  15.2  29.5  23.6
12    12  17.5  27.7  24.5
```

Summary

The dplyr package provides a concise set of operations for managing data frames. With these functions we can do a number of complex operations in just a few lines of code. In particular, we can often conduct the beginnings of an exploratory analysis with the powerful combination of group_by() and summarize().

Once you learn the dplyr grammar there are a few additional benefits

- dplyr can work with other data frame “backends” such as SQL databases. There is an SQL interface for relational databases via the DBI package

- dplyr can be integrated with the data.table package for large fast tables

The dplyr package is handy way to both simplify and speed up your data frame management code. It’s rare that you get such a combination at the same time!

Control Structures

Control structures in R allow you to control the flow of execution of a series of R expressions. Basically, control structures allow you to put some “logic” into your R code, rather than just always executing the same R code every time. Control structures allow you to respond to inputs or to features of the data and execute different R expressions accordingly.

Commonly used control structures are

- if and else: testing a condition and acting on it
- for: execute a loop a fixed number of times
- while: execute a loop *while* a condition is true
- repeat: execute an infinite loop (must break out of it to stop)
- break: break the execution of a loop
- next: skip an iteration of a loop

Most control structures are not used in interactive sessions, but rather when writing functions or longer expressions. However, these constructs do not have to be used in functions and it’s a good idea to become familiar with them before we delve into functions.

if-else

The if-else combination is probably the most commonly used control structure in R (or perhaps any language). This structure allows you to test a condition and act on it depending on whether it’s true or false.

```
if(<condition>) {
      ## do something
}
## Continue with rest of code
```

You can have a series of tests by following the initial if with any number of else ifs.

```
if(<condition1>) {
      ## do something
} else if(<condition2>)  {
      ## do something different
} else {
      ## do something different
}
```

If – else: example

```
## Generate a uniform random number
x <- runif(1, 0, 10)
if(x > 3) {
        y <- 10
} else {
        y <- 0
}
```

```
y <- if(x > 3) {
        10
} else {
        0
}
```

The value of `y` is set depending on whether `x > 3` or not. This expression can also be written a different, but equivalent, way in R.

for Loops

For loops are pretty much the only looping construct that you will need in R. While you may occasionally find a need for other types of loops, in my experience doing data analysis, I’ve found very few situations where a for loop wasn’t sufficient. In R, for loops take an interator variable and assign it successive values from a sequence or vector. For loops are most commonly used for iterating over the elements of an object (list, vector, etc.)

```
> for(i in 1:10) {
+       print(i)
+ }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

This loop takes the i variable and in each iteration of the loop gives it values 1, 2, 3, …, 10, executes the code within the curly braces, and then the loop exits.

for Loops

```
> x <- c("a", "b", "c", "d")
>
> for(i in 1:4) {
+       ## Print out each element of 'x'
+       print(x[i])
+ }
[1] "a"
[1] "b"
[1] "c"
[1] "d"
```

The seq_along() function is commonly used in conjunction with for loops in order to generate an integer sequence based on the length of an object (in this case, the object x).

```
> ## Generate a sequence based on length of 'x'
> for(i in seq_along(x)) {
+       print(x[i])
+ }
[1] "a"
[1] "b"
[1] "c"
[1] "d"
```

It is not necessary to use an index-type variable.

```
> for(letter in x) {
+       print(letter)
+ }
[1] "a"
[1] "b"
[1] "c"
[1] "d"
```

For one line loops, the curly braces are not strictly necessary.

```
> for(i in 1:4) print(x[i])
[1] "a"
[1] "b"
[1] "c"
[1] "d"
```

Nested for loops

Nested loops are commonly needed for multidimensional or hierarchical data structures (e.g. matrices, lists). Be careful with nesting though. Nesting beyond 2 to 3 levels often makes it difficult to read/understand the code. If you find yourself in need of a large number of nested loops, you may want to break up the loops by using functions (discussed later).

```
x <- matrix(1:6, 2, 3)

for(i in seq_len(nrow(x))) {
        for(j in seq_len(ncol(x))) {
                print(x[i, j])
        }
}
```

while Loops

While loops begin by testing a condition. If it is true, then they execute the loop body. Once the loop body is executed, the condition is tested again, and so forth, until the condition is false, after which the loop exits.

```
> count <- 0
> while(count < 10) {
+       print(count)
+       count <- count + 1
+ }
[1] 0
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
```

While loops can potentially result in infinite loops if not written properly. Use with care!

while Loops

Sometimes there will be more than one condition in the test.

```
> z <- 5
> set.seed(1)
>
> while(z >= 3 && z <= 10) {
+       coin <- rbinom(1, 1, 0.5)
+
+       if(coin == 1) {  ## random walk
+             z <- z + 1
+       } else {
+             z <- z - 1
+       }
+ }
> print(z)
[1] 2
```

Conditions are always evaluated from left to right. For example, in the above code, if `z` were less than 3, the second test would not have been evaluated.

repeat Loops

repeat initiates an infinite loop right from the start. These are not commonly used in statistical or data analysis applications but they do have their uses. The only way to exit a repeat loop is to call break.
One possible paradigm might be in an iterative algorithm where you may be searching for a solution and you don't want to stop until you're close enough to the solution. In this kind of situation, you often don't know in advance how many iterations it's going to take to get "close enough" to the solution.

```
x0 <- 1
tol <- 1e-8

repeat {
    x1 <- computeEstimate()

    if(abs(x1 - x0) < tol) {  ## Close enough?
        break
    } else {
        x0 <- x1
    }
}
```

Note that the above code will not run if the computeEstimate() function is not defined (I just made it up for the purposes of this demonstration).
The loop above is a bit dangerous because there's no guarantee it will stop. You could get in a situation where the values of x0 and x1 oscillate back and forth and never converge. Better to set a hard limit on the number of iterations by using a for loop and then report whether convergence was achieved or not.

next, break

next is used to skip an iteration of a loop.

```
for(i in 1:100) {
        if(i <= 20) {
                ## Skip the first 20 iterations
                next
        }
        ## Do something here
}
```

break is used to exit a loop immediately, regardless of what iteration the loop may be on.

```
for(i in 1:100) {
      print(i)

      if(i > 20) {
              ## Stop loop after 20 iterations
              break
      }
}
```

exercise

Random vector

1. Generate a random normal vector of size 100
2. Compute its mean with for/repeat loop
3. Compute its variance withfor/repeat loop

treating missing values

1. Use the airquality dataset from base
2. Compute the percentage p_na of missing values in a column
3. If p_na > 0,5 delete the column
4. If p_na <= 0,5 replace the missing values by 0 or by the mean of the column, depending on a variable "type_na"

Loop Functions

Writing for and while loops is useful when programming but not particularly easy when working interactively on the command line. Multi-line expressions with curly braces are just not that easy to sort through when working on the command line. R has some functions which implement looping in a compact form to make your life easier.

- **lapply(): Loop over a list and evaluate a function on each element**
- **sapply(): Same as lapply but try to simplify the result**
- **apply(): Apply a function over the margins of an array**
- **tapply(): Apply a function over subsets of a vector**
- **mapply(): Multivariate version of lapply**

An auxiliary function split is also useful, particularly in conjunction with lapply.

lapply()

The lapply() function does the following simple series of operations:

1. it loops over a list, iterating over each element in that list
2. it applies a *function* to each element of the list (a function that you specify)
3. and returns a list (the l is for "list").

This function takes three arguments: (1) a list X; (2) a function (or the name of a function) FUN; (3) other arguments via its ... argument. If X is not a list, it will be coerced to a list using as.list().

The body of the lapply() function can be seen here.

```
> lapply
function (X, FUN, ...)
{
   FUN <- match.fun(FUN)
   if (!is.vector(X) || is.object(X))
      X <- as.list(X)
   .Internal(lapply(X, FUN))
}
<bytecode: 0x7ff2f68331c0>
<environment: namespace:base>
```

Note that the actual looping is done internally in C code for efficiency reasons. It's important to remember that lapply() always returns a list, regardless of the class of the input.

lapply()

Here's an example of applying the mean() function to all elements of a list. If the original list has names, the the names will be preserved in the output.

```
> x <- list(a = 1:5, b = rnorm(10))
> lapply(x, mean)
$a
[1] 3

$b
[1] 0.1322028
```

Notice that here we are passing the mean() function as an argument to the lapply() function. Functions in R can be used this way and can be passed back and forth as arguments just like any other object. When you pass a function to another function, you do not need to include the open and closed parentheses () like you do when you are *calling* a function.

lapply()

Here is another example of using lapply().

```
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d =
rnorm(100, 5))
> lapply(x, mean)
$a
[1] 2.5

$b
[1] 0.248845

$c
[1] 0.9935285

$d
[1] 5.051388
```

You can use lapply() to evaluate a function multiple times each with a different argument.

lapply()

Below, is an example where I call the runif() function (to generate uniformly distributed random variables) four times, each time generating a different number of random numbers.

```
> x <- 1:4
> lapply(x, runif)
[[1]]
[1] 0.02778712

[[2]]
[1] 0.5273108 0.8803191

[[3]]
[1] 0.37306337 0.04795913 0.13862825

[[4]]
[1] 0.3214921 0.1548316 0.1322282 0.2213059
```

lapply()

When you pass a function to lapply(), lapply() takes elements of the list and passes them as the *first argument* of the function you are applying. In the above example, the first argument of runif() is n, and so the elements of the sequence 1:4 all got passed to the n argument of runif().

Functions that you pass to lapply() may have other arguments. For example, the runif() function has a min and max argument too. In the example above I used the default values for min and max. How would you be able to specify different values for that in the context of lapply()?

Here is where the ... argument to lapply() comes into play. Any arguments that you place in the ... argument will get passed down to the function being applied to the elements of the list.

Here, the min = 0 and max = 10 arguments are passed down to runif() every time it gets called.

```
> x <- 1:4
> lapply(x, runif, min = 0, max = 10)
[[1]]
[1] 2.263808

[[2]]
[1] 1.314165 9.815635

[[3]]
[1] 3.270137 5.069395 6.814425

[[4]]
[1] 0.9916910 1.1890256 0.5043966 9.2925392
```

lapply()

The lapply() function and its friends make heavy use of *anonymous* functions. Anonymous functions are like members of Project Mayhem—they have no names. These are functions are generated "on the fly" as you are using lapply(). Once the call to lapply() is finished, the function disappears and does not appear in the workspace.
Here I am creating a list that contains two matrices.

```
> x <- list(a = matrix(1:4, 2, 2), b = matrix(1:6, 3, 2))
> x
$a
     [,1] [,2]
[1,]    1    3
[2,]    2    4

$b
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

lapply()

Suppose I wanted to extract the first column of each matrix in the list. I could write an anonymous function for extracting the first column of each matrix.

```
> lapply(x, function(elt) { elt[,1] })
$a
[1] 1 2
$b
[1] 1 2 3
```

Notice that I put the function() definition right in the call to lapply(). This is perfectly legal and acceptable. You can put an arbitrarily complicated function definition inside lapply(), but if it's going to be more complicated, it's probably a better idea to define the function separately.
For example, I could have done the following.

```
> f <- function(elt) {
+       elt[, 1]
+ }
> lapply(x, f)
$a
[1] 1 2
$b
[1] 1 2 3
```

Now the function is no longer anonymous; it's name is f. Whether you use an anonymous function or you define a function first depends on your context. If you think the function f is something you're going to need a lot in other parts of your code, you might want to define it separately. But if you're just going to use it for this call to lapply(), then it's probably simpler to use an anonymous function.

sapply()

The sapply() function behaves similarly to lapply(); the only real difference is in the return value. sapply() will try to simplify the result of lapply() if possible. Essentially, sapply() calls lapply() on its input and then applies the following algorithm:

- If the result is a list where every element is length 1, then a vector is returned
- If the result is a list where every element is a vector of the same length (> 1), a matrix is returned.
- If it can't figure things out, a list is returned

sapply()

Here’s the result of calling lapply()

```
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d =
rnorm(100, 5))
> lapply(x, mean)
$a
[1] 2.5

$b
[1] -0.251483

$c
[1] 1.481246

$d
[1] 4.968715
```

sapply()

Here's the result of calling lapply()

```
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d =
rnorm(100, 5))
> lapply(x, mean)
$a
[1] 2.5

$b
[1] -0.251483

$c
[1] 1.481246

$d
[1] 4.968715
```

Here's the result of calling sapply() on the same list.

```
> sapply(x, mean)
       a         b         c         d
 2.500000 -0.251483  1.481246  4.968715
```

Because the result of lapply() was a list where each element had length 1, sapply() collapsed the output into a numeric vector, which is often more useful than a list.

split()

The split() function takes a vector or other objects and splits it into groups determined by a factor or list of factors. The arguments to split() are

```
> str(split)
function (x, f, drop = FALSE, ...)
```

where

- x is a vector (or list) or data frame
- f is a factor (or coerced to one) or a list of factors
- drop indicates whether empty factors levels should be dropped

The combination of split() and a function like lapply() or sapply() is a common paradigm in R. The basic idea is that you can take a data structure, split it into subsets defined by another variable, and apply a function over those subsets. The results of applying tha function over the subsets are then collated and returned as an object. This sequence of operations is sometimes referred to as “map-reduce” in other contexts.

split()

Here we simulate some data and split it according to a factor variable. Note that we use the gl() function to “generate levels” in a factor variable.

```
> x <- c(rnorm(10), runif(10), rnorm(10, 1))
> f <- gl(3, 10)
> split(x, f)
$`1`
 [1]  0.3981302 -0.4075286  1.3242586 -0.7012317 -0.5806143 -1.0010722
 [7] -0.6681786  0.9451850  0.4337021  1.0051592

$`2`
 [1] 0.34822440 0.94893818 0.64667919 0.03527777 0.59644846 0.41531800
 [7] 0.07689704 0.52804888 0.96233331 0.70874005

$`3`
 [1]  1.13444766  1.76559900  1.95513668  0.94943430  0.69418458  1.89367370
 [7] -0.04729815  2.97133739  0.61636789  2.65414530
```

split()

A common idiom is split followed by an lapply.

```
> lapply(split(x, f), mean)
$`1`
[1] 0.07478098

$`2`
[1] 0.5266905

$`3`
[1] 1.458703
```

tapply

tapply() is used to apply a function over subsets of a vector. It can be thought of as a combination of split() and sapply() for vectors only. I’ve been told that the “t” in tapply() refers to “table”, but that is unconfirmed.

```
> str(tapply)
function (X, INDEX, FUN = NULL, ..., default = NA, simplify = TRUE)
```

the arguments to tapply() are as follows:

- X is a vector
- INDEX is a factor or a list of factors (or else they are coerced to factors)
- FUN is a function to be applied
- … contains other arguments to be passed FUN
- simplify, should we simplify the result?

tapply

Given a vector of numbers, one simple operation is to take group means.

```
> ## Simulate some data
> x <- c(rnorm(10), runif(10), rnorm(10, 1))
> ## Define some groups with a factor variable
> f <- gl(3, 10)
> f
 [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3
Levels: 1 2 3
> tapply(x, f, mean)
        1         2         3
0.1896235 0.5336667 0.9568236
```

tapply

We can also take the group means without simplifying the result, which will give us a list. For functions that return a single value, usually, this is not what we want, but it can be done.

```
> tapply(x, f, mean, simplify = FALSE)
$`1`
[1] 0.1896235
$`2`
[1] 0.5336667
$`3`
[1] 0.9568236
```

We can also apply functions that return more than a single value. In this case, tapply() will not simplify the result and will return a list. Here's an example of finding the range of each sub-group.

```
> tapply(x, f, range)
$`1`
[1] -1.869789  1.497041
$`2`
[1] 0.09515213 0.86723879
$`3`
[1] -0.5690822  2.3644349
```

apply()

The apply() function is used to a evaluate a function (often an anonymous one) over the margins of an array. It is most often used to apply a function to the rows or columns of a matrix (which is just a 2-dimensional array). However, it can be used with general arrays, for example, to take the average of an array of matrices. Using apply() is not really faster than writing a loop, but it works in one line and is highly compact.

```
> str(apply)
function (X, MARGIN, FUN, ...)
```

The arguments to apply() are

- X is an array
- MARGIN is an integer vector indicating which margins should be "retained".
- FUN is a function to be applied
- ... is for other arguments to be passed to FUN

apply()

Here I create a 20 by 10 matrix of Normal random numbers. I then compute the mean of each column.

```
> x <- matrix(rnorm(200), 20, 10)
> apply(x, 2, mean)  ## Take the mean of each column
 [1]  0.02218266 -0.15932850  0.09021391  0.14723035
-0.22431309 -0.49657847
 [7]  0.30095015  0.07703985 -0.20818099  0.06809774
```

I can also compute the sum of each row.

```
> apply(x, 1, sum)   ## Take the mean of each row
 [1] -0.48483448  5.33222301 -3.33862932 -1.39998450
2.37859098  0.01082604
 [7] -6.29457190 -0.26287700  0.71133578 -3.38125293
-4.67522818  3.01900232
[13] -2.39466347 -2.16004389  5.33063755 -2.92024635
3.52026401 -1.84880901
[19] -4.10213912  5.30667310
```

Note that in both calls to apply(), the return value was a vector of numbers.

apply()

You’ve probably noticed that the second argument is either a 1 or a 2, depending on whether we want row statistics or column statistics. What exactly *is* the second argument to apply()?
The MARGIN argument essentially indicates to apply() which dimension of the array you want to preserve or retain. So when taking the mean of each column, I specify

```
> apply(x, 2, mean)
```

because I want to collapse the first dimension (the rows) by taking the mean and I want to preserve the number of columns. Similarly, when I want the row sums, I run

```
> apply(x, 1, mean)
```

because I want to collapse the columns (the second dimension) and preserve the number of rows (the first dimension).

Col/Row Sums and Means

For the special case of column/row sums and column/row means of matrices, we have some useful shortcuts.

- rowSums = apply(x, 1, sum)
- rowMeans = apply(x, 1, mean)
- colSums = apply(x, 2, sum)
- colMeans = apply(x, 2, mean)

The shortcut functions are heavily optimized and hence are *much* faster, but you probably won't notice unless you're using a large matrix. Another nice aspect of these functions is that they are a bit more descriptive. It's arguably more clear to write colMeans(x) in your code than apply(x, 2, mean).

mapply()

The mapply() function is a multivariate apply of sorts which applies a function in parallel over a set of arguments. Recall that lapply() and friends only iterate over a single R object. What if you want to iterate over multiple R objects in parallel? This is what mapply() is for.

```
> str(mapply)
function (FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)
```

The arguments to mapply() are

- FUN is a function to apply
- ... contains R objects to apply over
- MoreArgs is a list of other arguments to FUN.
- SIMPLIFY indicates whether the result should be simplified

The mapply() function has a different argument order from lapply() because the function to apply comes first rather than the object to iterate over. The R objects over which we apply the function are given in the ... argument because we can apply over an arbitrary number of R objects.

mapply()

For example, the following is tedious to type
list(rep(1, 4), rep(2, 3), rep(3, 2), rep(4, 1))
With mapply(), instead we can do

```
>  mapply(rep, 1:4, 4:1)
[[1]]
[1] 1 1 1 1

[[2]]
[1] 2 2 2

[[3]]
[1] 3 3

[[4]]
[1] 4
```

This passes the sequence 1:4 to the first argument of rep() and the sequence 4:1 to the second argument.

mapply()

Here’s another example for simulating random Normal variables.

```
> noise <- function(n, mean, sd) {
+       rnorm(n, mean, sd)
+ }
> ## Simulate 5 randon numbers
> noise(5, 1, 2)
[1] -0.5196913  3.2979182 -0.6849525  1.7828267
2.7827545
>
> ## This only simulates 1 set of numbers, not 5
> noise(1:5, 1:5, 2)
[1] -1.670517  2.796247  2.776826  5.351488  3.422804
```

mapply()

Here's another example for simulating random Normal variables.

```
> noise <- function(n, mean, sd) {
+       rnorm(n, mean, sd)
+ }
> ## Simulate 5 randon numbers
> noise(5, 1, 2)
[1] -0.5196913  3.2979182 -0.6849525  1.7828267
2.7827545
>
> ## This only simulates 1 set of numbers, not 5
> noise(1:5, 1:5, 2)
[1] -1.670517  2.796247  2.776826  5.351488  3.422804
```

we can use mapply() to pass the sequence 1:5 separately to the noise() function so that we can get 5 sets of random numbers, each with a different length and mean.

```
> mapply(noise, 1:5, 1:5, 2)
[[1]]
[1] 0.8260273

[[2]]
[1] 4.764568 2.336980

[[3]]
[1] 4.6463819 2.5582108 0.9412167

[[4]]
[1]  3.978149  1.550018 -1.192223  6.338245

[[5]]
[1] 2.826182 1.347834 6.990564 4.976276 3.800743
```

exercise

mean and standard deviation over the columns

1. Compute the mean of all columns of iris dataset
2. Compute their standard deviation

Summary

- The loop functions in R are very powerful because they allow you to conduct a series of operations on data using a compact form

- The operation of a loop function involves iterating over an R object (e.g. a list or vector or matrix), applying a function to each element of the object, and the collating the results and returning the collated results.

- Loop functions make heavy use of anonymous functions, which exist for the life of the loop function but are not stored anywhere

- The split() function can be used to divide an R object in to subsets determined by another variable which can subsequently be looped over using loop functions.