3
R Command Patterns

Almost everyone who writes computer commands starts by copying and modifying existing commands. To do this, you need to be able to **read** command expressions. Once you can read, you will know enough to identify the patterns you need for any given task and consider what needs to be modified to suit your particular purpose.

As you read this chapter, you will likely get an inkling of what the commands used in examples are intended to do, but that's not what's important now. Instead, focus on

- Distinguishing between *functions* and arguments.
- Distinguishing between *data tables* and the *variables* that are contained in them.
- Recognizing when a function is being used.
- Identifying the arguments to a function.
- Observing how assignment allows values to be stored and referred to by name.
- Discerning when the output of one function becomes the input to another.

You are **not** expected at this point to be able to *write* R expressions. You'll have plenty of opportunity to do that once you've learned to *read* and recognize the several different patterns used in R data wrangling and visualization commands.

3.1 Language Patterns and Syntax

In a human language like English or Chinese, *syntax* is the arrangement of words and phrases to create well-formed sentences. For example, "Four horses pulled the king's carriage," combines noun phrases ("Four horses", "the king's carriage") with a verb.

Consider this pair of English language sentence patterns, a statement and a question:

1. I did go to school .
2. Did I go to school ?

The content of each of the boxes can be replaced by an equivalent object.

- I can be replaced with "you", "we", "he", "Janice", "the President", and so on.
- go to can be replaced with "stay in", "attend", "drive to", "see", ...
- school can be replaced with "the lake", "the movie", "Fred's parents' house", ...

Such replacements produce sentences like these:

- Did we stay in Fred's parents' house?
- Janice did drive to the lake.
- Did the President see the movie?
- Did he attend school?

Each sentence expresses something different, but they all follow the same patterns.

R HAS A FEW PATTERNS THAT SUFFICE for many data wrangling and visualization tasks. Consider these patterns:

- `object_name` <- `function_name` (`arguments`)
  This is called *function application*. The output of the function will be stored under the name to the left of <-.
- `object_name` <-
  `Data_Table` %>%
  `function_name` (`arguments`)
  This is called *chaining syntax*.
- `object_name` <-
  `Data_Table` %>%
  `function_name` (`arguments`) %>%
  `function_name` (`arguments`)
  This is an extended form of chaining syntax. Such chains can be extended indefinitely.

3.2 *Five kinds of objects*

There are five kinds of objects that you will be working with extensively.

1. Data tables
2. Functions
3. Arguments
4. Variables
5. Constants

Just as it helps in English to know what's a noun and what's a verb, etc., by identifying each kind of object in an expression you'll have an easier time understanding what the expression does.

1. DATA TABLES contain tidy data. A data table comprises one or more variables. It's easy to distinguish functions from data tables and variables: the function name will always be followed immediately by an open parenthesis. Data tables appear at the start of a chain, just before the first %>%.

2. FUNCTIONS are the objects that transform an input into an output. They are easy to spot. Functions are used by applying them to arguments, so you will see an opening parenthesis right after the function name. The corresponding closing parenthesis comes after the arguments.

There are a few functions that have a different syntax, e.g. the simple mathematical functions which are given *between* two arguments, e.g. `3 + 2` or `7 * 8`. This is called *infix* notation and is meant to mimic traditional arithmetic notation. Some of the other infix functions you'll encounter are `%in%`, `==`, `>=`, and so on.

3. ARGUMENTS DESCRIBE THE DETAILS of what a function is to do. They appear between the parentheses that follow a function name. One important exception: data tables are typically presented as an input to a function using the chaining notation %>%.

Many functions take *named arguments* where the name of the argument is followed by a = sign and then the value of that argument. For instance, `by = x` gives `x` as the value of the argument named `by`.

4. VARIABLES ARE THE COLUMNS IN A DATA TABLE. In this book, they will *always* be in function arguments, that is, between parentheses.

5. CONSTANTS ARE SINGLE VALUES, most commonly a number or a character string. Character strings will always be in quotation marks, `"like this."` Numerals are the written form of numbers, for

instance `-42`, `1984`, `3.14159`. Sometimes you'll see numerals written in scientific notation, e.g. `6.0221413e+23` or `6.62606957e-34`.

To help distinguish *data tables* from the *variables* in them, these notes will use a simple naming convention.

- The names of data tables will start with a CAPITAL letter. For instance: `WorldCities`, `NCI60`, `BabyNames`, and so on.
- The names of variables within data tables will start with a **lowercase** letter. For instance: `latitude`, `country`, `population`, `date`, `sex`, `count`, `countryRegion`, `population_density`.

These are *conventions*, not rules enforced by the R language. As you create your own data tables and variables, it is up to you to follow the convention. And, apologies, but even in this book you will encounter data or variables that fail to follow this convention. With time, such situations will be identified and fixed. But they can't be fixed everywhere, since sometimes you rely on resources developed by people or institutions who don't follow the conventions.

EXAMPLE: CLASSIFYING OBJECTS IN AN EXPRESSION. Consider this command:

```
BabyNames %>%
  filter(name == "Arjun") %>%
  summarise(total = sum(count))
```

The statement involves a data table (hints: starts with a capital letter, not followed by a opening parenthesis), three functions (hint: look for the names followed by an opening parenthesis), a named argument `total` (hint: inside the parentheses and followed by a single equal sign). The string `"Arjun"` is a constant. The remaining names — `name` and `count` — are variables. (Hint: they are involved in the arguments to functions).

The `filter()` function is being given the argument `name == "Arjun"`. This can be confusing. Although == somewhat resembles =, the single = is always just punctuation in a named argument. The double == is a function — one of those few infix functions that don't involve parentheses.

By the way, the overall effect of the command is to calculate the total number of individuals named Arjun represented in the `BabyNames` data. The result of the calculation is 5,578. Try it yourself!

NOTE TO EXPERIENCED PROGRAMMERS. You may be wondering where common programming constructs like looping and conditional flow, indexing, lists, and function definition fit in with this book. This book uses a couple of domain-specific, sub-languages, particularly `dplyr` and `ggplot2`. A "sub-language" is a part of a computer language that can be used almost like a language of it's own. The functions in `dplyr` and `ggplot2` already contain within them those programming constructs, so there is no need to use them explicitly. This is analogous to driving a car. The sub-language is the use of

the steering wheel, brake, and accelerator. You can use these to accomplish your task without having to know about how the engine or suspension work.

3.3 *Example Expressions*

These are expressions that you will use frequently, each written in the chaining style. To remind you, `BabyNames` is a data table.

Expressions that give a quick glance at a data table

These functions are generally used interactively, in the R console, to help you when constructing expressions for data wrangling or visualization.

```
BabyNames %>% nrow()     # how many cases in the table
```

```
[1] 1792091
```

```
BabyNames %>% names()    # the names of the variables
```

```
[1] "name"  "sex"   "count" "year"
```

```
BabyNames %>% head(3)    # the first three cases
```

name	sex	count	year
Mary	F	7065	1880
Anna	F	2604	1880
Emma	F	2003	1880

```
BabyNames %>% str()      # another view of the first few cases
```

```
'data.frame':   1792091 obs. of  4 variables:
 $ name : chr  "Mary" "Anna" "Emma" "Elizabeth" ...
 $ sex  : chr  "F" "F" "F" "F" ...
 $ count: int  7065 2604 2003 1939 1746 1578 1472 1414 1320 1288 ...
 $ year : int  1880 1880 1880 1880 1880 1880 1880 1880 1880 1880 ...
```

```
BabyNames %>% glimpse() # just like str()
```

```
Observations: 1792091
Variables:
$ name  (chr) "Mary", "Anna", "Emma", "E...
$ sex   (chr) "F", "F", "F", "F", "F", "...
$ count (int) 7065, 2604, 2003, 1939, 17...
$ year  (int) 1880, 1880, 1880, 1880, 18...
```

Modifying a data table

Sometimes, you will want to modify a data table and store the result under the original name. To illustrate:

```
BabyNames <-
  BabyNames %>%
  filter(year == 1998)
```

Because assignment is being used to capture the value being created by `filter()`, it might look like nothing is being done. But, behind the scenes, `BabyNames` has changed.

```
BabyNames %>% nrow()
```

```
[1] 27890
```

```
BabyNames %>% head(4)
```

Of course, it's not necessarily to use the original name to store the result of the modification. You can use any name that you think appropriate.

name	sex	count	year
Emily	F	26177	1998
Hannah	F	21368	1998
Samantha	F	20191	1998
Ashley	F	19868	1998

Table 3.1: The first 4 rows in the data table produced by filtering `BabyNames` to include only those babies born in 1998.

Named arguments and functions in arguments

The previous example, `BabyNames %>% head(4)` involved an argument to a function; `head()` is given the number 4 to specify how many cases to show. Sometimes the arguments will be the name of a variable or a function applied to a variable. Here's an example:

```
BabyNames %>%
  group_by(sex) %>%
  summarise(total = sum(count))
```

sex	total
F	1765766
M	1910081

The `group_by()` and `summarise()` functions will be described in Chapter 4. They are called *data verbs* because they each take a data table as an input and produce a data table as output.

Taking apart the above expression, you can see three functions, `group_by()`, `summarise()` and `sum()`. The name of functions is always followed by an open parenthesis. Inside those parentheses are the arguments. The argument to `group_by()` is `sex`, a variable. How do you know? It's evidently not a data table — it's not capitalized. It's neither a character string nor a numerical constant. And it's not a function — `sex` isn't followed by an opening parenthesis. By the process of elimination, this suggests that `sex` is a variable. The argument to `summarise()` is the expression `total = sum(count)`.

The argument to `summarise()` is in named-argument form. Note that `sum()` is a function. You can tell this from the expression: `sum` is followed immediately by an open parenthesis.

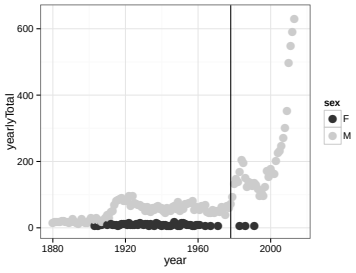

Figure 3.1: The number of babies born each year given the name "Prince".

EXAMPLE: A GENUINE DATA WRANGLING COMPUTATION. What do realistic data wrangling and graphics expressions look like? Figure 3.1 depicts the popularity of the name "Prince" as it varies over the years. (Don't worry for now about what the various functions are doing. You will get to that later.)

```
Princes <-
  BabyNames %>%
  filter(name == "Prince") %>%
  group_by(year, sex) %>%
  summarise(yearlyTotal = sum(count))
# Now graph it!
Princes %>%
  ggplot(aes(x = year, y = yearlyTotal)) +
  geom_point(aes(color = sex)) +
  geom_vline(xintercept = 1978) +
  ylim(0,640) + xlim(1880,2015)
```

Judging from Figure 3.1, the name "Prince" has been increasing in popularity over the last 40 years. One possible explanation is the popularity of the musician, Prince. The vertical line in the graph marks the year that Prince's first album was released: 1978.

3.4 *Constant objects*

Sometimes you will use assignment to store a constant. For instance:

```
data_file_name <- "tiny.cc/mosaic/engines.csv"
age_cutoff <- 21
```

Reminder: The two kinds of constants we will use are quoted character strings and numerals.

Naming constants in this way can help to make your data wrangling expressions more readable. For instance, by using `age_cutoff` in your expressions, you make it easily to update your expressions if you decide to change the age cutoff; just change the 21 to whatever the new value is to be.

3.5 *Chaining syntax*

If you have experience with R, you may never have seen the chaining operator %>% before this book.

Chapter 2 showed several different examples of function application, `function_name ( arguments )`, for instance `sqrt(2)` and `help(CPS85, package="mosaic")`

Chaining syntax is merely another form of function application. The following two patterns accomplish exactly the same thing:

`Data_Table` %>% `function_name` (`arguments`) ..... Chaining pattern.

`function_name` (`Data_Table` , `arguments`) ..... Non-chaining pattern.

In chaining syntax, the value on the left side of %>% becomes the *first argument* to the function on the right side. Note also that %>% is never at the start of a line — it should be placed at the end of any line which is to be followed by another step in a command sequence.

The chaining syntax is a help to the human reading and writing computer commands. Chaining makes more prominent the functions at each step. This is particularly helpful when there are many steps in a data wrangling or visualization task. To illustrate, here's a command sequence used previously in this chapter:

```
Princes <-
  BabyNames %>%
  filter(name == "Prince") %>%
  group_by(year, sex) %>%
  summarise(yearlyTotal = sum(count))
```

This expression can also be written in a non-chaining syntax, for instance:

```
Princes <-
  summarise(
    group_by(
      filter(BabyNames, name == "Prince"),
      year, sex),
    yearlyTotal = sum(count))
```

The chaining syntax takes advantage of the nature of data verbs. Each step in a data wrangling sequence takes a data table along with some other arguments as input and produces a data table as output. The chaining sequence brings the other arguments much closer to the data verb itself, so that you can see at a glance which arguments belong to which data verbs.

3.6 *Exercises*

Problem 3.1
Using the object name `fireplace`, write different expressions with enough context to be able to identify the name as belonging to

1. a data frame
2. a function
3. the name of a named argument
4. a variable

Write one expression for each of the above.

Problem 3.2
Explain why the following sentence is illegitimate:

```
Result <- %>% filter(BabyNames, name=="Prince")
```

Problem 3.3
What's wrong with this statement?

```
help(NHANES, package <- "NHANES")
```

Problem 3.4
Consider these R expressions. (You don't have to know what the various functions do to solve this problem.)

```
Princes <-
  BabyNames %>%
  filter(name == "Prince") %>%
  group_by(year, sex) %>%
  summarise(yearlyTotal = sum(count))
# Now graph it!
Princes %>%
  ggplot(aes(x = year, y = yearlyTotal)) +
  geom_point(aes(color = sex)) +
  geom_vline(xintercept = 1978)
```

There are several kinds of named objects in the above expressions.

a. function name
b. data table name
c. variable name
d. name of a named argument

Using the naming convention and position rules, identify what kind of object each of the following name is used for. That is, assign one of the types (a) through (d) to each name.

1) `BabyNames` 2) `filter` 3) `name` 4) `==`
2) `group_by` 6) `year` 7) `sex` 8) `summarise`
3) `yearlyTotal` 10) `sum` 11) `count` 12) `ggplot`
4) `aes` 14) `x` 15) `y` 16) `geom_point`
5) `color` 18) `geom_vline` 19) `xintercept`

Problem 3.5
There are several small, example data tables in the `ggplot2` package. Look at the `msleep` data table by using the `View()` function with the name of the object as an argument.

- What is the meaning of the `brainwt` variable?
- How many cases are there?
- What is the real-world meaning of a case?
- What are the levels of the `vore` variable?

Problem 3.6
The data verb functions all take a data table as their first argument and return a data table as their output. The chaining syntax lets the output of one function become the input to the following function, so you don't have to repeat the name of the data frame. An alternative syntax is to assign the output of one function to a named object, then use the object as the first argument to the next function in the computation.

Each of these statements, but one, will accomplish the same calculation. Identify the statement that does not match the others.

a)
```
BabyNames %>%
  group_by( year, sex ) %>%
  summarise( totalBirths=sum(count))
```

b)
```
group_by( BabyNames, year, sex) %>%
  summarise( totalBirths=sum(count) )
```

c)
```
group_by( BabyNames, year, sex ) %>%
  summarise( totalBirths=mean(count) )
```

d)
```
Tmp <- group_by(BabyNames, year, sex)
summarise( Tmp, totalBirths=sum(count) )
```

Problem 3.7
Which characters can be used in an object name?

Problem 3.8
The `date()` function returns an indication of the current time and date.

- What arguments does `date()` take? Use `help()` to find out.
- What *kind* of object is the result from `date()`.