Chapter 21: Pipe operators (%>% and others)

lhs	rhs
A value or the magrittr placeholder.	A function call using the magrittr semantics

Pipe operators, available in `magrittr`, `dplyr`, and other R packages, process a data-object using a sequence of operations by passing the result of one step as input for the next step using infix-operators rather than the more typical R method of nested function calls.

Note that the intended aim of pipe operators is to increase human readability of written code. See Remarks section for performance considerations.

Section 21.1: Basic use and chaining

The pipe operator, `%>%`, is used to insert an argument into a function. It is not a base feature of the language and can only be used after attaching a package that provides it, such as `magrittr`. The pipe operator takes the left-hand side (LHS) of the pipe and uses it as the first argument of the function on the right-hand side (RHS) of the pipe. For example:

```
library(magrittr)

1:10 %>% mean
# [1] 5.5

# is equivalent to
mean(1:10)
# [1] 5.5
```

The pipe can be used to replace a sequence of function calls. Multiple pipes allow us to read and write the sequence from left to right, rather than from inside to out. For example, suppose we have `years` defined as a factor but want to convert it to a numeric. To prevent possible information loss, we first convert to character and then to numeric:

```
years <- factor(2008:2012)

# nesting
as.numeric(as.character(years))

# piping
years %>% as.character %>% as.numeric
```

If we don't want the LHS (Left Hand Side) used as the *first* argument on the RHS (Right Hand Side), there are workarounds, such as naming the arguments or using `.` to indicate where the piped input goes.

```
# example with grepl
# its syntax:
# grepl(pattern, x, ignore.case = FALSE, perl = FALSE, fixed = FALSE, useBytes = FALSE)

# note that the `substring` result is the *2nd* argument of grepl
grepl("Wo", substring("Hello World", 7, 11))

# piping while naming other arguments
"Hello World" %>% substring(7, 11) %>% grepl(pattern = "Wo")
```

```r
# piping with .
"Hello World" %>% substring(7, 11) %>% grepl("Wo", .)

# piping with . and curly braces
"Hello World" %>% substring(7, 11) %>% { c(paste('Hi', .)) }
#[1] "Hi World"

#using LHS multiple times in argument with curly braces and .
"Hello World" %>% substring(7, 11) %>% { c(paste(. ,'Hi', .)) }
#[1] "World Hi World"
```

Section 21.2: Functional sequences

Given a sequence of steps we use repeatedly, it's often handy to store it in a function. Pipes allow for saving such functions in a readable format by starting a sequence with a dot as in:

```r
. %>% RHS
```

As an example, suppose we have factor dates and want to extract the year:

```r
library(magrittr) # needed to include the pipe operators
library(lubridate)
read_year <- . %>% as.character %>% as.Date %>% year

# Creating a dataset
df <- data.frame(now = "2015-11-11", before = "2012-01-01")
#          now     before
# 1 2015-11-11 2012-01-01

# Example 1: applying `read_year` to a single character-vector
df$now %>% read_year
# [1] 2015

# Example 2: applying `read_year` to all columns of `df`
df %>% lapply(read_year) %>% as.data.frame  # implicit `lapply(df, read_year)
#    now before
# 1 2015   2012

# Example 3: same as above using `mutate_all`
library(dplyr)
df %>% mutate_all(funs(read_year))
# if an older version of dplyr use `mutate_each`
#    now before
# 1 2015   2012
```

We can review the composition of the function by typing its name or using `functions`:

```r
read_year
# Functional sequence with the following components:
#
#  1. as.character(.)
#  2. as.Date(.)
#  3. year(.)
#
# Use 'functions' to extract the individual functions.
```

We can also access each function by its position in the sequence:

```r
read_year[[2]]
```

```
# function (.)
# as.Date(.)
```

Generally, this approach may be useful when clarity is more important than speed.

Section 21.3: Assignment with %<>%

The `magrittr` package contains a compound assignment infix-operator, `%<>%`, that updates a value by first piping it into one or more `rhs` expressions and then assigning the result. This eliminates the need to type an object name twice (once on each side of the assignment operator `<-`). `%<>%` must be the first infix-operator in a chain:

```
library(magrittr)
library(dplyr)

df <- mtcars
```

Instead of writing

```
df <- df %>% select(1:3) %>% filter(mpg > 20, cyl == 6)
```

or

```
df %>% select(1:3) %>% filter(mpg > 20, cyl == 6) -> df
```

The compound assignment operator will both pipe and reassign `df`:

```
df %<>% select(1:3) %>% filter(mpg > 20, cyl == 6)
```

Section 21.4: Exposing contents with %$%

The exposition pipe operator, `%$%`, exposes the column names as R symbols within the left-hand side object to the right-hand side expression. This operator is handy when piping into functions that do not have a **`data`** argument (unlike, say, `lm`) and that don't take a data.frame and column names as arguments (most of the main dplyr functions).

The exposition pipe operator `%$%` allows a user to avoid breaking a pipeline when needing to refer to column names. For instance, say you want to filter a data.frame and then run a correlation test on two columns with **`cor.test`**:

```
library(magrittr)
library(dplyr)
mtcars %>%
  filter(wt > 2) %$%
  cor.test(hp, mpg)

#>
#>  Pearson's product-moment correlation
#>
#> data:  hp and mpg
#> t = -5.9546, df = 26, p-value = 2.768e-06
#> alternative hypothesis: true correlation is not equal to 0
#> 95 percent confidence interval:
#>  -0.8825498 -0.5393217
#> sample estimates:
#>       cor
```

```
#> -0.7595673
```

Here the standard `%>%` pipe passes the data.frame through to `filter()`, while the `%$%` pipe exposes the column names to `cor.test()`.

The exposition pipe works like a pipe-able version of the base R `with()` functions, and the same left-hand side objects are accepted as inputs.

Section 21.5: Creating side effects with %T>%

Some functions in R produce a side effect (i.e. saving, printing, plotting, etc) and do not always return a meaningful or desired value.

`%T>%` (tee operator) allows you to forward a value into a side-effect-producing function while keeping the original `lhs` value intact. In other words: the tee operator works like `%>%`, except the return values is `lhs` itself, and not the result of the `rhs` function/expression.

Example: Create, pipe, write, and return an object. If `%>%` were used in place of `%T>%` in this example, then the variable `all_letters` would contain `NULL` rather than the value of the sorted object.

```
all_letters <- c(letters, LETTERS) %>%
    sort %T>%
    write.csv(file = "all_letters.csv")

read.csv("all_letters.csv") %>% head()
#   x
# 1 a
# 2 A
# 3 b
# 4 B
# 5 c
# 6 C
```

Warning: Piping an unnamed object to `save()` will produce an object named `.` when loaded into the workspace with `load()`. However, a workaround using a helper function is possible (which can also be written inline as an anonymous function).

```
all_letters <- c(letters, LETTERS) %>%
    sort %T>%
    save(file = "all_letters.RData")

load("all_letters.RData", e <- new.env())

get("all_letters", envir = e)
# Error in get("all_letters", envir = e) : object 'all_letters' not found

get(".", envir = e)
#  [1] "a" "A" "b" "B" "c" "C" "d" "D" "e" "E" "f" "F" "g" "G" "h" "H" "i" "I" "j" "J"
# [21] "k" "K" "l" "L" "m" "M" "n" "N" "o" "O" "p" "P" "q" "Q" "r" "R" "s" "S" "t" "T"
# [41] "u" "U" "v" "V" "w" "W" "x" "X" "y" "Y" "z" "Z"

# Work-around
save2 <- function(. = ., name, file = stop("'file' must be specified")) {
  assign(name, .)
  call_save <- call("save", ... = name, file = file)
  eval(call_save)
}
```

```
all_letters <- c(letters, LETTERS) %>%
    sort %T>%
    save2("all_letters", "all_letters.RData")
```

Section 21.6: Using the pipe with dplyr and ggplot2

The %>% operator can also be used to pipe the dplyr output into ggplot. This creates a unified exploratory data analysis (EDA) pipeline that is easily customizable. This method is faster than doing the aggregations internally in ggplot and has the added benefit of avoiding unnecessary intermediate variables.

```
library(dplyr)
library(ggplot)


diamonds %>%
    filter(depth > 60) %>%
    group_by(cut) %>%
    summarize(mean_price = mean(price)) %>%
    ggplot(aes(x = cut, y = mean_price)) +
        geom_bar(stat = "identity")
```