Intro R piping

Roberto Molowny-Horas

April 24-25, 2023

Piping. What is it?

The Wikipedia (**https://en.wikipedia.org/wiki/Pipeline_(software) (https://en.wikipedia.org/wiki/Pipeline_(software))**) provides a useful definition for the use of pipelines in programming:

> "*A pipeline consists of a chain of processing elements (processes, threads, coroutines, functions, etc.), arranged so that the output of each element is the input of the next*"

The pipe operator is a recent addition to the base-R universe (that is, it is built into the R code). Previously, the package **magrittr** offered an implementation of a pipe operator with syntax **%>%**. However, starting from R version 4.1.0, piping can be done natively in R with the operator **|>**, which avoids loading external packages. Their functionalities are very similar, so it should be hassle-free to make changes to your codes in case you have used **magrittr** before.

Yes, but… what is it, really?

The concept of "piping" comes from Unix. In simple words, by means of a pipeline we give or direct one data structure as input to a function or process. That data structure, in turn, may be an output from another process. This is usually read as a left-to-right expression, like e.g.:

```
c(4, 5) |> mean()
```

In this line we have already introduced **|>**, the pipe operator in base-R. If you have installed the **magrittr** R package in your computer, you may try like this (although I would recommend sticking with the base-R implementation):

```
# library(magrittr)
c(4, 5) %>% mean()
```

Or, if you want to avoid loading the **magrittr** R package in advance:

```
`%>%` <- magrittr::`%>%`
c(4, 5) %>% mean()
```

However, the implementation of the pipe operator in base-R is simpler and more convenient, without unnecessary dependences on external packages.

Advantages

- We can easily read expressions from left to right.
- We can also avoid long expressions with multiple nested functions and parentheses.

- We can quickly add further steps (i.e. function calls) to an expression.

One example:

```
x <- runif(10)
y1 <- cumsum(exp(diff(log(sort(x)))))
y2 <- x |> sort() |> log() |> diff() |> exp() |> cumsum()
```

Another example:

```
breaks <- seq(-10, 10, by = .1)
x <- rnorm(10000)
h1 <- mean(hist(x, breaks = breaks, plot=F)$density)
h2 <- x |> hist(, breaks = breaks, plot=F) |> getElement("density") |> mean()
```

Sometimes we may want to pipe into two or more functions. In R it is very quick and easy to create an anonymous function that is called only when needed, and it is destroyed/eliminated afterwards:

```
h1 <- x |> hist(, breaks = breaks, plot=F) |> getElement("density") |>
  (function(z) {c(mean(z), sd(z))})()
h2 <- x |> hist(, breaks = breaks, plot=F) |> getElement("density") |>
  (\(z) {c(mean(z), sd(z))})()
```

A more complex use of pipelines in R would involve the **lm()** linear regression function. An example:

```
data("iris")
r1 <- lm(Sepal.Width ~ Sepal.Length, data = iris)
r2 <- iris |> with(lm(Sepal.Width ~ Sepal.Length))
r3 <- iris |> (function(dat, foo) {lm(foo, dat)}) (Sepal.Width ~ Sepal.Length)
r4 <- iris |> (\(dat, foo) {lm(foo, dat)}) (Sepal.Width ~ Sepal.Length)
# Does not work! r4 <- iris |> (\(dat, foo) {lm(dat, foo)}) (Sepal.Width ~ Sepal.Leng
th)
# Neither does this! r4 <- iris |> (\(foo, dat) {lm(foo, dat)}) (Sepal.Width ~ Sepal.
Length)
```

Sometimes the use of pipelines may not be advantageous in terms of code comprehensibility or debugging. However, once we start working within the *tidy* universe with *tidy* data, pipes come in handy and thus become a very useful tool.