Pipe operator in R

This page is based on a document created by Aurélien Nicosia (ULaval) in 2023 called "Pipe Operator." It has been updated.

Since version 4.1.0 of R, released in May 2021, the language has introduced the *pipe* operator `|>`, inspired by what the `magrittr` package was doing. Starting with R 4.3.0, the style guide for the `tidyverse` recommends using the basic *pipe* operator, not the one from `magrittr`.

Keyboard shortcuts

In RStudio, the keyboard shortcut for inserting the *pipe* operator is:

- On Windows: Ctrl + Shift + M
- On macOS: + + M

This operator introduces a way to chain instructions and pass arguments to functions in a more readable way than the traditional method.

To summarize how this operator works, here is how it transforms a few function calls:

- `f(x)` becomes `x |> f()`;
- `f(x, y)` becomes `x |> f(y)`;
- `h(g(f(x)))` becomes `x |> f() |> g() |> h()`.

This operator allows you to highlight the **sequence of actions** rather than the object on which the sequence of actions is performed. This makes the code more readable (and having readable code is a good practice). Reading the instruction `h(g(f(x)))` from left to right, we first see the call to the function `h`, then the call to the function `g`, and finally the call to the function `f`. However, this instruction is evaluated in reverse order. In fact, R will first:

1. evaluate `f(x)`;
2. then pass the result to the function `g` and return the result;
3. which will be passed to the function `h` and the final result will be returned.

If we want to write code that correctly reflects the order of evaluation, we could write:

```
res1 <- f(x)
res2 <- g(res1)
h(res2)
```

However, this code has the disadvantage of creating objects that we do not necessarily want to keep. The `|>` operator does not have this disadvantage! In fact, an instruction written using the `|>` operator allows us to follow the order of evaluations without creating unnecessary objects in memory.

For even greater clarity, it is possible to extend an instruction containing several `|>` operators over several lines so that there is one function per line:

```
x |>
  f() |>
  g() |>
  h()
```

If the argument we want to pass with the `|>` operator is not the first one, we must use `_` as follows with a named argument: `f(y, z = x)` becomes `x |> f(y, z = _)`.

Let's take an example to illustrate the use of the `|>` operator. Suppose we have the following character string:

```
text <- "This is an example"
```

and we want to correct it—replace "example" with "example" and add a period at the end—with the following instruction:

```
paste0(gsub(pattern = "example", replacement = "example", x = text), ".")
```

```
[1] "This is an example."
```

This instruction is a little difficult to read because of the call to the `gsub` function nested within a call to the `paste0` function. We could rewrite it as follows using the `|>` operator:

```
text |>
  gsub(pattern = "example", replacement = "example", x = _) |>
  paste0(".")
```

```
[1] "This is an example."
```

Let's take another numerical example. We want to perform the following calculation:

$$\frac{(2+4)\times 8}{2}.$$

To do this, we need a few mathematical functions.

```r
add <- function(x, y) {
  x + y
}

mul <- function(x, y) {
  x * y
}

div <- function(x, y) {
  x / y
}
```

We can perform the calculation in three different ways:

```r
# By creating different objects
res1 <- add(2, 4)
res2 <- mul(res1, 8)
res3 <- div(res2, 2)
print(res3)
```

```
[1] 24
```

```r
# By nesting functions
res <- div(mul(add(2, 4), 8), 2)
print(res)
```

```
[1] 24
```

```r
# With the pipe operator
res <- 2 |>
  add(4) |>
  mul(8) |>
  div(2)
print(res)
```

```
[1] 24
```