Operators you should make more use of in R

Only recently have I discovered the true power of some the operators in R. Here are some tips on some underused operators in R:

The %in% operator

This funny looking operator is very handy. It’s short for testing if several values appear in an object. For instance

```
x = c(2, 6, 4, 4, 6, 8, 10, 14, 2)
```

To grab all the values where x is 2, 4 or 14 we could do

```
x[x == 2 | x == 4 | x == 14]
```

```
## [1]  2  4  4 14  2
```

or we could use `%in%` …

```
x[x %in% c(2, 4, 14)]
```

```
## [1]  2  4  4 14  2
```

This is something I use all the time for filtering data. Imagine you’ve got a tibble of data relating to the world (step up **spData**)

```
library("dplyr")
library("sf")
library("sp")
data(world, package = "spData")

# drop the geometry column because we don't need it
world = world %>%
  st_drop_geometry()
```

Your colleague sends you a list of 50 countries (I’m going to randomly sample the names from the data) and says that they want the average life expectency for each continent group within these 50 countries.

```
colleague_countries = world %>%
  sample_n(50) %>%
  pull(name_long)
head(colleague_countries)
```

```
## [1] "Yemen"          "New Zealand"    "Kyrgyzstan"     "New Caledonia"
## [5] "Morocco"        "Ecuador"
```

We could then ask R to return every row where the column `name_long` matches any value in `colleague_countries` using the `%in%` operator

```
world %>%
  filter(name_long %in% colleague_countries) %>%
  group_by(continent) %>%
  summarise(av_life_exp = mean(lifeExp, na.rm = TRUE))
```

```
## # A tibble: 6 x 2
##   continent     av_life_exp
##
## 1 Africa              63.6
## 2 Asia                72.3
## 3 Europe              79.0
## 4 North America       74.6
## 5 Oceania             80.3
## 6 South America       74.3
```

Did you know?

You can make your own `%%` operators! For instance

```
%add% = function(a, b) a + b
```

```
2 %add% 3
```

```
## [1] 5
```

The && and || operators

If you look on the help page for the logical operators `&` and `|`, you'll find `&&` and `||`. What do they do and hope they actually differ from their single counterparts? Let's look at an example. Take a vector `x`

```
x = c(2, 4, 6, 8)
```

To test for the values in x that are greater than 3 and less than 7 we would write

```
x > 3 & x < 7
```

```
## [1] FALSE  TRUE  TRUE FALSE
```

Then to return these values we would subset using square brackets

```
x[x > 3 & x < 7]
```

```
## [1] 4 6
```

What happens if we repeat these steps with `&&`?

```
x > 3 && x < 7
```

```
## [1] FALSE
```

```
x[x > 3 && x < 7]
```

```
## numeric(0)
```

What is happening here is that the double `&` only evaluates the first element of a vector. So evaluation proceeds only until a result is determined. This has another nice consequence. For example, take the object `a`

```
a = 5
```

In the following test

```
a == 4 & a == 5 & a == 8
```

```
## [1] FALSE
```

All 3 tests are evaluated, even though we know that after the first test, `a == 4`, this test is `FALSE`. Where as in using the double `&&`

```
a == 4 && a == 5 && a == 8
```

```
## [1] FALSE
```

Here we only evaluate the first test as that is all we need to determine the result. This is more efficient as it won't evaluate any test it doesn't need to. To demonstrate this, we'll use two toy functions

```
a = function(){
  print("Hi I am f")
  return(FALSE)
}
b = function(){
  print("Hi I am g")
  return(TRUE)
}

a() & b()
```

```
## [1] "Hi I am f"
## [1] "Hi I am g"
```

```
## [1] FALSE
```

When using the single `&`, R has to evaluate both functions even thought the output of the left hand side is FALSE

```
a() && b()
```

```
## [1] "Hi I am f"
```

```
## [1] FALSE
```

But using `&&`, R only has to evaluate the first function until the result is determined! It's the same rule for `||`…

```
b() | a()
```

```
## [1] "Hi I am g"
## [1] "Hi I am f"
```

```
## [1] TRUE
```

```
b() || a()
```

```
## [1] "Hi I am g"
```

```
## [1] TRUE
```

The xor() function

This last one isn't so much an operator as a function. The `xor()`

function is an exclusive version of the `|`. Take two vector, x and y

```
x = c(1,1,2)
y = c(1,2,2)
```

To get all the elements where either x is 1 or y is 2 we would write

```
x == 1 | y == 2
```

```
## [1] TRUE TRUE TRUE
```

However, this will also return the elements where x = 1 AND y = 2. If we only want elements where only one statement is `TRUE`, we would use xor()

```
xor(x == 1 , y == 2)
```

```
## [1]  TRUE FALSE  TRUE
```

That’s all for this time. Thanks for reading!