A Short Introduction to the `sprintf()` Function

R Reading Group, Department of Linguistics, University of Alberta

Isabell Hubert

Wednesday, July 22, 2015

`sprintf()` was one of the functions that we talked about during our last R Reading Group meeting on July 20. As it is very useful to either create strings from variables, or to change the name of a range of objects into uniform format, we thought it might be useful to have a closer look at it.

`sprintf()` is originally a function in the C programming language that has been wrapped into `R`. If you are a more proficient programmer or user of `R`, or if you know and use `R`'s `paste()` function already, this introduction might not be so useful for you. While `paste()` and `sprintf()` do virtually the same thing, you do have more control over the ouput with `sprintf()`, and if you are planning to pad strings and numbers to a certain length, `paste()` might not be able to do this at all - unless you use it in conjuction with the `formatC()` function. A nice comparison (in plain English, not the `R` Documentation style) with examples of the two functions can be found on *R Tutorial* for simple examples, and in the *Cookbook for R* for more elaborate ones. Stackoverflow has a comparison on how to achieve padding in `sprintf()` vs `paste()`.

Note that `sprintf()` is not pronounced *sprint-f*, but rather *s-print-f* as it is a printing function.

Basic Syntax

The basic syntax of `sprintf()` involves calling the function with two arguments, the first one of which is a character string where you want certain things inserted, and the second argument is what you want inserted in the string. You signal the position where the second argument should be inserted by `%s` for strings, and `%d` for integers.

In the following example, we store the string *Alberta* in the variable `a`, and then insert this in the sentence that is passed to the `sprintf()` function as the first argument:

```
a <- "Alberta"
sprintf("The University of %s is relatively old.", a)
```

```
## [1] "The University of Alberta is relatively old."
```

Manipulating Character Strings

You can use `sprintf()` to add things to names. Maybe the items in a vector are all part of the same dataset, let's call it *set 2*, and you want to add this information to each of their names. To do this, you can instruct `sprintf()` to add a fixed character string to an item or a range of items, and then save this in a new vector/data frame/whatever you need:

```
items <- c("apple", "ball", "cake", "daisy", "elk")
items.set <- sprintf("set2-%s", items)
items.set
```

```
## [1] "set2-apple" "set2-ball"  "set2-cake"  "set2-daisy" "set2-elk"
```

Note that we have used `%s` and not `%d` as we are dealing with strings here. To add something to the end of a string, simply put the `%s` at the end of the first argument you pass to the function.

You can access items in a vector just like you usually would:

```
ilike <- sprintf("I like my %s.", items[1])
ilike
```

```
## [1] "I like my apple."
```

And you can loop over them as well:

```
like.items <- vector()

for (i in items) {
    like.loop <- sprintf("I like my %s.", i)
    like.items <- c(like.items, like.loop)
}

cat(like.items,sep="\n")
```

```
## I like my apple.
## I like my ball.
## I like my cake.
## I like my daisy.
## I like my elk.
```

Working with Integers

Making batch editing of character strings a lot easier, `sprintf()` really shines when padding character strings or integers to a certain length. We can create a vector that stores an integer to test things out:

```
number <- 4
```

We can use `sprintf()` to pad this number with characters, for example with as many spaces as we want. This is especially useful for batch naming items, which we'll get to soon. Here, we will add two spaces in front of the number (note how we use `%d` now instead of `%s`):

```
sprintf("%3d", number)
```

```
## [1] "  4"
```

We can do the same with zeroes, which might also be easier to make out than spaces - simply insert a zero between the % sign and the *d*:

```
sprintf("%03d", number)
```

```
## [1] "004"
```

Note that the number you specify after the % sign and before the `d` always means the *total* number of characters in the resulting string, *including* the number or string that you want padded (the second argument passed to the function). This is why `sprintf()` is so great for padding - you tell it how many characters or digits you want, and it'll take care of the rest.

If we create a vector with numbers of different lengths and then run `sprintf()`, this becomes even clearer:

```
numbers <- c(1, 15, 332, 4848)
numbers.padded <- sprintf("%04d", numbers)
numbers.padded
```

```
## [1] "0001" "0015" "0332" "4848"
```

We have padded all numbers in this vector to a length of four digits, which is the highest number stored in this vector. For sorting opertions, it might make sense to pad the shorter numbers up with zeroes until you reach the length of the highest number. Now you would, for example, have no troubles sorting the items (we all know how in certain file systems `item13.txt` will magically be sorted between `item1.txt` and `item2.txt`, which is at least one problem you'renot going to have anymore!)

A version of this came in very handy for the first assignment in the online R Programming class. There, you were asked to create a function where users could specify the ID of a weather observation post whose reports they wanted to have summed up. The function was then supposed to pull the data corresponding to those ID's from a folder that stored 332 csv files, one for each of 332 weather observation posts. The files were stored in the format 001.csv, 002.csv, ..., 025. csv, ..., 113.csv, and so on and so forth, up to 332.csv. So, if the user specifies that they would like to see data from weather observation post 5, how would you use this integer to access the respective csv file in the folder?

The answer is very simple - so let's assume the user specified `id=5` in the function call. We can then simply pad this ID input to three numeric digits, and then append `.csv` to have it match the csv file name:

```
id=5
filename <- sprintf("%03d.csv", id)
filename
```

```
## [1] "005.csv"
```

Then, we can simply move on by looping through the `filename` vector and reading in the respective files. This works for more than one ID, too - so say the user specified ID's 5, 115, 239, and 331:

```
id=c(5,115,239,331)
filenames <- sprintf("%03d.csv", id)
filenames
```

```
## [1] "005.csv" "115.csv" "239.csv" "331.csv"
```

You can then simply pass the resulting `filenames` vector to your `read.csv()` function, et voila! No need to load 332 csv files when you can instead just load the ones that are requested or required.

This can be very handy when you are working with large datasets, either storing eye-tracking or ERP data, or large corpora stored in separate files.