Functions and Iteration with Park Visitation Data

Katie Willi

2025-06-09

Lesson Objectives

In this lesson we will work with National Park Service (NPS) visitation data to create our own function and use `map()` for iteration. We'll be utilizing NPS-wide visitation data that has been provided as a CSV file.

Loading Data and Packages

We'll work directly with a CSV file containing park visitation data. Let's load our packages and read in our data. No new packages needed for this lesson!

```r
library(tidyverse)
```

We have one CSV file to work with: - `nps_parkwide_visitation.csv`: Contains monthly visitation data across all NPS park units for multiple years

Let's read in this data file:

```r
# Read in the parkwide visitation data
parkwide_data <- read_csv("data/nps_parkwide_visitation.csv")

# Take a look at the structure of our data
glimpse(parkwide_data)
```

Functions

You may find yourself writing repetitive code when working with data. Instead of copying and pasting many coding steps over and over again and tweaking just a tiny portion of it, we can write *functions* that combine many coding steps into just one command. The benefits of reducing redundant code in this way are threefold. As Grolemund & Wickham describe in their book, *R for Data Science*:

1. It's easier to see the intent of your code, because your eyes are drawn to what's different, not what stays the same.
2. It's easier to respond to changes in requirements. As your needs change, you only need to make changes in one place, rather than remembering to change every place that you copied-and-pasted the code.
3. You're likely to have fewer bugs because each line of code is used in more places.

Functions provide the option of changing just a minor part of the code base from one run to the next. To develop a function requires specific formatting:

```r
<NAME> <- function(<ARGUMENTS>){

  <ACTIONS>

    return(<OUTPUT>)
}
```

… where NAME is what we want to name the function; ARGUMENTS are the variables in the code that get "tweaked"; ACTIONS are the lines of code we want the function to perform (which includes our ARGUMENTS); and the OUTPUT is the object we want as the final outcome of running the function.

Creating a Function

Let's create a simple function that filters our parkwide visitor data to a user-specified year.

```r
filter_year_function <- function(year){

  parkwide_data %>%
    filter(Year == year)

}
```

Our function name is `filter_year_function`. Our argument, or the thing we want to be able to change, is the `year`. We perform the action of filtering our parkwide visitor data set within the function. Now that this function is in our environment, we can use the function like any other:

```r
filter_year_function(year = 2018)
```

Creating a Decadal Summary Function

Let's try making a more sophisticated function called `get_decadal_visitation()` that summarizes the parkwide visitation data for a decade of choice. For `get_decadal_visitation()`, we will use filter, group_by, and summarise code as the basis for our function:

```r
get_decadal_visitation <- function(decade) {

  decade_end <- decade + 9

  parkwide_data %>%
    # filter to years in the decade
    filter(Year >= decade, Year <= decade_end) %>%
    group_by(Month) %>%
    summarise(Decade = paste0(decade, "s"),
              MonthlyTotal = sum(RecreationVisitors, na.rm = TRUE),
              AvgVisitation = mean(RecreationVisitors, na.rm = TRUE),
              # for identifying decades with some missing years
              NumberYears = n_distinct(Year))

}
```

Now we can pull visitation data for any decade with just one line of code!

```r
pull_1970s <- get_decadal_visitation(decade = 1970)
pull_1980s <- get_decadal_visitation(decade = 1980)
pull_1990s <- get_decadal_visitation(decade = 1990)
pull_2010s <- get_decadal_visitation(decade = 2010)
pull_2020s <- get_decadal_visitation(decade = 2020)
```

Function Defaults

When developing functions, there is an option for setting default values for arguments so that you don't necessarily have to write all of them out every time you run it in the future. But, the option still exists within the function to make changes when necessary. For example, let's tweak our `get_decadal_visitation()` function to have the default decade be 1990:

```
get_decadal_visitation <- function(decade = 1990) {

  decade_end <- decade + 9

  parkwide_data %>%
    filter(Year >= decade, Year <= decade_end) %>%
    group_by(Month) %>%
    summarise(Decade = paste0(decade, "s"),
              MonthlyTotal = sum(RecreationVisitors, na.rm = TRUE),
              AvgVisitation = mean(RecreationVisitors, na.rm = TRUE),
              NumberYears = n_distinct(Year))
}

get_decadal_visitation()
```

Because the default decade is 1990, you don't have to write it out explicitly in the function (so long as that's the decade you're interested in).

```
get_decadal_visitation()
```

But, you still have the option of changing the decade to something else:

```
get_decadal_visitation(decade = 1970)
```

Mapping with {purrr}

Sometimes we want to apply the same function to multiple inputs. For example, above, we still ended up calling the same function multiple times. The {tidyverse}'s {purrr} package has an iteration function called `map()` that takes a vector and applies a single function across it, then automatically stores all of the results into a list.

For example, if we want to get parkwide decadal data for multiple decades, we can use `map()`:

```
years <- c(1970, 1980, 1990, 2000, 2010, 2020)

output_map <- years %>%
  map(~ get_decadal_visitation(decade = .x))
```

... where `~` indicates that we want to perform `get_decadal_visitation()` across all decades, and `.x` indicates that we want to use our piped vector, `years`, as the input to the `decade` argument.

We now have a list containing five data frames: one for each decade of visitation data:

```
summary(output_map)
```

Because each year's output is structured identically, we can confidently combine each decade's data frame into a single data frame using `dplyr::bind_rows()`:

```
multi_years <- bind_rows(output_map)
```

Want more practice with functions and mapping?

Review “07-More_on-Functions_and_Iteration.Rmd”, adapted from the RStudio Cloud Primers under their CC BY-SA 4.0 license.