Coding Best Practices

Quantitative Applications for Data Analysis

Alexey Fedoseev

February 3, 2026

Best practices

What are coding best practices?

- These are coding practices which have been discovered, over the course of decades, to produce code which is easy to write, read, debug, test, modify, share and use.
- Best practices are a set of rules which affect how you
  - design the code;
  - implement the code.
- Broadly speaking, coding best practices can be summed up thus:
  - write code which is modular;
  - don't write code which has already been written, and
  - write code which is easy to read and understand.

We've discussed a number of these points during previous lectures, but today we will discuss them in detail.

Modularity

What do I mean when I say that I'm writing modular code? I'm writing code which:

- is separated into individual functions and procedures and each of these performs a single, specific task;
- is separated into files, if the code is big enough, which contain related functionality;
- is written so as to enforce boundaries between sections of code, and
- includes testing routines, to test the functions against known correct answers.

Modularity

Why does modularity matter?

- Scientific software can be large, complex and subtle;
- If each section of code uses the internal details of other sections you must understand the entire code at once to understand what the code in a particular section is doing;
- This makes finding bugs (mistakes) extremely difficult, and
- It also makes writing testing routines for your code extremely difficult.

Developing the code

Let us continue with the car project. So far we have the data directory and the file car_search.R

```
user@scinet cars $ cat car_search.R
args <- commandArgs(trailingOnly = TRUE)
cat("The command line arguments are:", args, "\n")

# Concatenates the vector using commas between the elements
vectorToString <- function(vector) {
  return(paste(vector, collapse = ", "))
}
```

Developing the code

We would like to view the available models based on the specific car manufacturer.

```r
# car_search.R

# Read the dataset
cars <- read.csv("./data/cars.csv")

# Store the first argument in a separate variable
car.make <- args[1]

# Select the entries with the specified car make
cars.make.data <- cars[cars$Make == car.make, ]

# View the first entries of the data frame
head(cars.make.data)
```

Developing the code

```
user@scinet cars $ Rscript car_search.R Lamborghini
The command line arguments are: Lamborghini
     City.mpg      Classification       Driveline
440        12 Manual transmission Rear-wheel drive
450        12 Manual transmission Four-wheel drive
451        12 Manual transmission Four-wheel drive
5073       12 Manual transmission  All-wheel drive
5074       12 Manual transmission  All-wheel drive
...
```

This is a lot of output and could be confusing for someone who runs this script. Let us extract the important information and present it in a tidy way.

Developing the code

Replace the output of the whole data frame with the following code.

```
# Select the entries with the specified car make
cars.make.data <- cars[cars$Make == car.make, ]

# Display year and model only
cat("Available years:", vectorToString(cars.make.data[, "Model.Year"]),"\n")
```

```
user@scinet cars $ Rscript car_search.R Lamborghini
Available models: 2010 Lamborghini Gallardo Coupe,
2011 Lamborghini Gallardo Coupe, 2011 Lamborghini Gallardo Coupe,
2012 Lamborghini Gallardo Coup, 2012 Lamborghini Gallardo Spyder
```

Notice that I removed the output of the command line arguments.

Developing the code

Our program does not handle well the situation when no arguments were given.

```
user@scinet cars $ Rscript car_search.R
Available models: NA, NA, NA, NA ...
```

To fix this let us display a usage example.

```r
# args[1] must be a car make
args <- commandArgs(trailingOnly = TRUE)

if (length(args) == 0) {
  cat("usage: Rscript car_search.R Lamborghini\n")
  quit()
}
```

```
user@scinet cars $ Rscript car_search.R
usage: Rscript car_search.R Lamborghini
```

Improving the program

Let us add a feature to our program when user can view the information about models available in a certain year.

```
user@scinet cars $ Rscript car_search.R Lamborghini
Available models: 2010 Lamborghini Gallardo Coupe,
2011 Lamborghini Gallardo Coupe, 2011 Lamborghini Gallardo Coupe,
2012 Lamborghini Gallardo Coup, 2012 Lamborghini Gallardo Spyder

user@scinet cars $ Rscript car_search.R Lamborghini 2011
Available models in 2011: 2011 Lamborghini Gallardo Coupe,
2011 Lamborghini Gallardo Coupe
```

We can use the second command line argument to specify the year of the model. In order to do that we need to reorganize our code.

Restructuring the code

```r
if (length(args) == 0) {
  cat("usage: Rscript car_search.R Lamborghini\n")
  quit()
}
if (length(args) == 1) {
  # Read the dataset
  cars <- read.csv("./data/cars.csv")
  # Store the first argument in a separate variable
  car.make <- args[1]
  # Select the entries with the specified car make
  cars.make.data <- cars[cars$Make == car.make, ]
  # Show all available models of the specified car make
  cat("Available models:",vectorToString(cars.make.data[,"Model.Year"]),"\n")
} else if (length(args) == 2) {
  cat("In progress...\n")
}
```

Adding a feature

```r
...
} else if (length(args) == 2) {
  cars <- read.csv("./data/cars.csv") # Read the dataset
  # Store the arguments in separate variables
  car.make <- args[1]
  car.year <- as.numeric(args[2])
  # Select the entries with the specified car make
  cars.by.make <- (cars$Make == car.make)
  # Select the cars with the specified year of production
  cars.by.year <- (cars$Year == car.year)
  # Select the entries that are of the specified make and year
  cars.by.make.and.year <- cars[cars.by.make & cars.by.year, ]
  # Display the models
  cat("Available models:",
    vectorToString(cars.by.make.and.year[, "Model.Year"]), "\n")
}
```

Modularity

Have you noticed how we had to restructure our code? While developing your code you are going to do this many times. Using functions helps us to separate the logic in the code.

First of all we already have an improved version of the function `vectorToString` from the lecture 6 on scripts and we can use it in our project.

Let us create a directory `lib` in our project and put the file `vectorUtils.R` that we have developed.

```
user@scinet cars $ mkdir lib
```

Modularity

```
user@scinet cars $ cat lib/vectorUtils.R
# vectorUtils.R

# Concatenates the vector using commas between the elements
vectorToString <- function(vec, last.and = FALSE) {
  if (last.and & length(vec) > 1) {
    # Concatenate all elements except the last one using commas
    str.commas <- paste(vec[-length(vec)], collapse = ", ")
    # Concatenate the last element using "and"
    str.and <- paste(str.commas, "and", vec[length(vec)])
    return(str.and)
  } else
    # Concatenate all elements using commas
    return(paste(vec, collapse = ", "))
}
```

Modularity

To load the `vectorUtils.R` use the command `source`.

```
args <- commandArgs(trailingOnly = TRUE)
source("lib/vectorUtils.R")
...
```

We need to verify that after the modifications our program still works as intended.

```
user@scinet cars $ Rscript car_search.R
usage: Rscript car_search.R Lamborghini
user@scinet cars $ Rscript car_search.R Lamborghini
Available models: 2010 Lamborghini Gallardo Coupe,
2011 Lamborghini Gallardo Coupe, 2011 Lamborghini Gallardo Coupe,
2012 Lamborghini Gallardo Coup, 2012 Lamborghini Gallardo Spyder
user@scinet cars $ Rscript car_search.R Lamborghini 2011
Available models: 2011 Lamborghini Gallardo Coupe,
2011 Lamborghini Gallardo Coupe
```

Split the logic

Our program has logic:

- Handle the specified arguments
  - if no parameters were given - show the usage example
  - if 1 parameter was given - treat it as a car make
  - if 2 parameters were given - treat the first argument as a car make and the second one as a model year
- Load the dataset into a data frame
- Slice the data frame according to the arguments provided
- Display the resulting sliced data frame

We can split the logic between the functions.

Driver script

```
# car_search.R

# args[1] must be a car make
# args[2] must be a year of a car model (if specified)
args <- commandArgs(trailingOnly = TRUE)

# import function vectorToString
source("lib/vectorUtils.R")

# import function checkArgs and parseArgs
source("lib/parseArgs.R")

# import function displayModels
source("lib/displayCars.R")
```

Driver script (continued)

```r
# Stop the script if no arguments were specified
if (length(args) == 0) {
  cat("usage: Rscript car_search.R Lamborghini\n")
  quit()
}

# Validate the command line arguments
args.filtered <- checkArgs(args)
# Read the dataset
cars <- read.csv("./data/cars.csv")
# Slice the data frame using the arguments
selected.cars <- parseArgs(args.filtered, cars)
# View the resulting data frame
displayModels(selected.cars)
```

Utility files - parseArgs.R

```r
# parseArgs.R

checkArgs <- function(args) {
  # Discard arguments # 3,4,...
  args.copy <- args[1:2]
  # Check whether the second argument is a number
  # as.numerics produces NA with the warning if it cannot convert the value
  # suppress the warning about NA
  if (suppressWarnings(is.na(as.numeric(args.copy[2]))))
    args.copy <- args[1] # Discard the second argument
  return(args.copy)
}
```

Utility files - parseArgs.R (continued)

```r
parseArgs <- function(args, cars) {
  # Store the first argument in a separate variable
  car.make <- args[1]
  # Select the entries with the specified car make
  cars.make.data <- cars[cars$Make == car.make, ]
  if (length(args) == 1) {
    return(cars.make.data)
  }
  if (length(args) == 2) {
    # Convert the second argument to numeric value and store in a variable
    car.year <- as.numeric(args[2])
    # Select the entries that are of the specified make and year
    cars.by.make.and.year <- cars.make.data[cars.make.data$Year==car.year,]
    return(cars.by.make.and.year)
  }
}
```

Utility files - displayCars.R

```r
# displayCars.R

displayModels <- function(cars.data) {
  # If the data frame is empty
  if (nrow(cars.data) == 0) {
    # Display the message and stop the script
    cat("There are no car models available\n")
    quit()
  }

  # Display the models
  cat("Available models:", vectorToString(cars.data[, "Model.Year"]),"\n")
}
```

Further improvements

Additional features

- Partial search: “Lambor” should match “Lamborghini”
- Case independent search: “ferrari” or “fERRAri” should match “Ferrari”
- More user interactions: if data frame is empty, display an example of the arguments for a non-empty output

Code improvements

- Defensive programming - check that function arguments, or script command-line arguments meet certain criteria, for example, handle the situation when the dataset does not exist.
- More detailed comments

While you are modifying the part of your code, another part could stop working as expected. If you do not notice it right away, it might lead to a chain of errors that is hard to trace to the beginning. To prevent this behavior it is useful to create functions that will test the resulting value of your functions.