

## 6.1: Reading CSV files

Perhaps the simplest format for exchanging data among computer systems is the *de facto* standard *comma separated values*, or *csv*, file. R provides a function to directly read data from a csv file and assign it to a data frame:

```
> processors <- read.csv("all-data.csv")
```

The name between the quotes is the name of the csv-formatted file to be read. Each file line corresponds to one data record. Commas separate the individual data fields in each record. This function assigns each data record to a new row in the data frame, and assigns each data field to the corresponding column. When this function completes, the variable `processors` contains all the data from the file `all-data.csv` nicely organized into rows and columns in a data frame.

If you type `processors` to see what is stored in the data frame, you will get a long, confusing list of data. Typing

```
> head(processors)
```

will show a list of column headings and the values of the first few rows of data. From this list, we can determine which columns to extract for our model development. Although this is conceptually a simple problem, the execution can be rather messy, depending on how the data was collected and organized in the file.

As with any programming language, R lets you define your own functions. This feature is useful when you must perform a sequence of operations multiple times on different data pieces, for instance. The format for defining a function is:

```
function-name <- function(a1, a2, ...) { R expressions
  return(object)
}
```

where `function-name` is the function name you choose and `a1, a2, ...` is the list of arguments in your function. The R system evaluates the expressions in the body of the definition when the function is called. A function can return any type of data object using the `return()` statement.

We will define a new function called `extract_data` to extract all the rows that have a result for the given benchmark program from the `processors` data frame. For instance, calling the function as follows:

```
> int92.dat <- extract_data("Int1992")
> fp92.dat <- extract_data("Fp1992")
> int95.dat <- extract_data("Int1995")
> fp95.dat <- extract_data("Fp1995")
> int00.dat <- extract_data("Int2000")
> fp00.dat <- extract_data("Fp2000")
> int06.dat <- extract_data("Int2006")
> fp06.dat <- extract_data("Fp2006")
```

extracts every row that has a result for the given benchmark program and assigns it to the corresponding data frame, `int92.dat`, `fp92.dat`, and so on.

We define the `extract_data` function as follows:

```
extract_data <- function(benchmark) {

  temp <- paste(paste("Spec",benchmark,sep=""), "..average.base.", sep="")

  perf <- get_column(benchmark,temp)

  max_perf <- max(perf)
  min_perf <- min(perf)
  range <- max_perf - min_perf
  nperf <- 100 * (perf - min_perf) / range

  clock <- get_column(benchmark,"Processor.Clock..MHz.")
  threads <- get_column(benchmark,"Threads.core")
  cores <- get_column(benchmark,"Cores")
  TDP <- get_column(benchmark,"TDP")
  transistors <- get_column(benchmark,"Transistors..millions.")
  dieSize <- get_column(benchmark,"Die.size..mm.2.")
  voltage <- get_column(benchmark,"Voltage..low.")
  featureSize <- get_column(benchmark,"Feature.Size..microns.")
  channel <- get_column(benchmark,"Channel.length..microns.")
  F04delay <- get_column(benchmark,"F04.Delay..ps.")
  L1icache <- get_column(benchmark,"L1..instruction...on.chip.")
  L1dcache <- get_column(benchmark,"L1..data...on.chip.")
  L2cache <- get_column(benchmark,"L2..on.chip.")
  L3cache <- get_column(benchmark,"L3..on.chip.")

  return(data.frame(nperf, perf, clock, threads, cores, TDP, transistors, dieSize,
    voltage, featureSize, channel, F04delay, L1icache, L1dcache, L2cache, L3cache))
}
```

The first line with the `paste` functions looks rather complicated. However, it simply forms the name of the column with the given benchmark results. For example, when `extract_data` is called with `Int2000` as the argument, the nested `paste` functions simply concatenate the strings "Spec", "Int2000", and "..average.base.". The final string corresponds to the name of the column in the `processors` data frame that contains the performance results for the `Int2000` benchmark, "SpecInt2000..average.base.".

The next line calls the function `get_column`, which selects all the rows with the desired column name. In this case, that column contains the actual performance result reported for the given benchmark program, `perf`. The next four lines compute the normalized performance value, `nperf`, from the `perf` value we obtained from the data frame. The following sequence of calls to `get_column` extracts the data for each of the predictors we intend to use in developing the regression model. Note that the second parameter in each case, such as "Processor.Clock..MHz.", is the name of a column in the `processors` data frame. Finally, the `data.frame()` function is a predefined R function that assembles all its arguments into a single data frame. The new function we have just defined, `extract_data()`, returns this new data frame.

Next, we define the `get_column()` function to return all the data in a given column for which the given benchmark program has been defined:

```
get_column <- function(x,y) {

  benchmark <- paste(paste("Spec",x,sep=""), "..average.base.", sep="")
  ix <- !is.na(processors[,benchmark]) return(processors[ix,y])
}
```

The argument `x` is a string with the name of the benchmark program, and `y` is a string with the name of the desired column. The nested `paste()` functions produce the same result as the `extract_data()` function. The `is.na()` function performs the interesting work. This function returns a vector with “1” values corresponding to the row numbers in the `processors` data frame that have `NA` values in the column selected by the `benchmark` index. If there is a value in that location, `is.na()` will return a corresponding value that is a `0`. Thus, `is.na` indicates which rows are missing performance results for the benchmark of interest. Inserting the exclamation point in front of this function complements its output. As a result, the variable `ix` will contain a vector that identifies every row that contains performance results for the indicated benchmark program. The function then extracts the selected rows from the `processors` data frame and returns them.

These types of data extraction functions can be somewhat tricky to write, because they depend so much on the specific format of your input file. The functions presented in this chapter are a guide to writing your own data extraction functions.

---

This page titled [6.1: Reading CSV files](#) is shared under a [CC BY-NC 4.0](#) license and was authored, remixed, and/or curated by [David Lilja \(University of Minnesota Libraries Publishing\)](#) via [source content](#) that was edited to the style and standards of the LibreTexts platform.