

Lecture 2: Data frames, functions, packages, and the tidyverse

Stats 32: Introduction to R for Undergraduates

Harrison Li

April 4, 2024

Agenda

1 Data frames

2 Functions

3 Packages

4 The tidyverse

Reading: Sections 1.3, 4.2, 4.4

Data frames

Data frames

Recall a data frame is a 2-D array of named columns, where each column is a vector.

Under the hood, a data frame is a named list, whose elements are the columns of the data frame (which are themselves vectors).

Data frames

The built-in `iris` data frame gives measurements for 50 flowers from each of 3 different species of irises. We can view the first 6 rows of `iris` using `head()`:

```
head(iris)
```

##	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
## 1	5.1	3.5	1.4	0.2	setosa
## 2	4.9	3.0	1.4	0.2	setosa
## 3	4.7	3.2	1.3	0.2	setosa
## 4	4.6	3.1	1.5	0.2	setosa
## 5	5.0	3.6	1.4	0.2	setosa
## 6	5.4	3.9	1.7	0.4	setosa

There are 5 named columns. What are their names and data types?

Data frame indexing

```
head(iris)
```

##	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
## 1	5.1	3.5	1.4	0.2	setosa
## 2	4.9	3.0	1.4	0.2	setosa
## 3	4.7	3.2	1.3	0.2	setosa
## 4	4.6	3.1	1.5	0.2	setosa
## 5	5.0	3.6	1.4	0.2	setosa
## 6	5.4	3.9	1.7	0.4	setosa

One way to index into a data frame is to treat it as a matrix:

```
iris[4,2]
```

```
## [1] 3.1
```

Data frame indexing

```
head(iris)
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
## 1	5.1	3.5	1.4	0.2	setosa
## 2	4.9	3.0	1.4	0.2	setosa
## 3	4.7	3.2	1.3	0.2	setosa
## 4	4.6	3.1	1.5	0.2	setosa
## 5	5.0	3.6	1.4	0.2	setosa
## 6	5.4	3.9	1.7	0.4	setosa

Alternatively, we recall that data frames are named lists, and use the list and vector indexing syntax from Lecture 1:

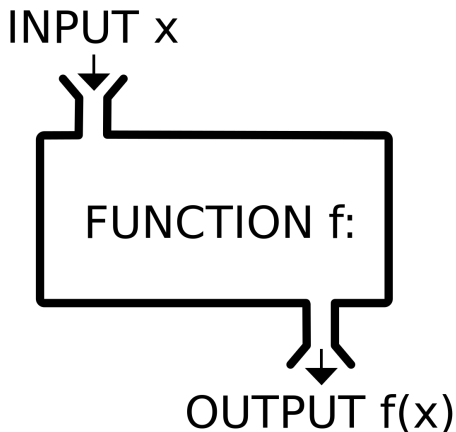
```
a_vector <- iris$Sepal.Width  
a_vector[4]
```

```
## [1] 3.1
```

Functions

Functions

Recall that in math, a **function** takes one or more inputs and returns an output:



Functions

Similarly, in programming, a function takes in some inputs (formally called **arguments** in R), runs some code based on those inputs, and returns an output (or **value**).

The main reason to use a function is to avoid having to copy and paste similar code multiple times to execute a common task many times.

To use a function with specific inputs is to **call** it.

Functions

For example, suppose we want to compute the average of numbers in a vector.

R has a built in function, `mean()`, that does this for you, taking in a single argument (the vector), and spitting out the mean:

```
v <- c(3, -1, -1, 1)
mean(v)
```

```
## [1] 0.5
```

The help page

The R help page for a built-in function can be reached by typing a question mark followed by the name of the function, e.g. `?mean`:

mean (base) R Documentation

Arithmetic Mean

Description

Generic function for the (trimmed) arithmetic mean.

Usage

```
mean(x, ...)
```

```
## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

Arguments

x An R object. Currently there are methods for numeric/logical vectors and [date](#), [date-time](#) and [time interval](#) objects. Complex vectors are allowed for `trim = 0`, only.

trim the fraction (0 to 0.5) of observations to be trimmed from each end of `x` before the mean is computed. Values of `trim` outside that range are taken as the nearest endpoint.

na.rm a logical value indicating whether NA values should be stripped before the computation proceeds.

... further arguments passed to or from other methods.

Value

If `trim` is zero (the default), the arithmetic mean of the values in `x` is computed, as a numeric or complex vector of length one. If `x` is not logical (coerced to numeric), numeric (including integer) or complex, `NA_real_` is returned, with a warning.

If `trim` is non-zero, a symmetrically trimmed mean is computed with a fraction of `trim` observations deleted from each end before the mean is computed.

The help page

The help page provides the following:

- A brief description of what the function does
- The list of arguments that the function takes in
- Details about how the function computes the value
- (Usually) several examples

Write your own function

Here's an example of a user-defined function:

```
myFunction <- function(myFirstArg, mySecondArg){  
  myVariable <- myFirstArg+mySecondArg  
  return(myVariable)  
}
```

The above code defines a function called myFunction

- Note myFunction is an object, just like the simpler objects we saw in Lecture 1.
- myFunction has two arguments: myFirstArg and mySecondArg. Functions can have any number of arguments.
- The function defines a new variable called myVariable as the sum of the two arguments.
- myFunction returns myVariable as its value.

Default arguments

Functions can have defaults for any or all arguments, so that you can (optionally) omit those arguments in a function call.

We redefine `myFunction` so that the second argument, `mySecondArg`, has a default value of 2:

```
myFunction <- function(myFirstArg, mySecondArg=2){  
  myVariable <- myFirstArg+mySecondArg  
  return(myVariable)  
}
```

Default arguments

```
myFunction(1, 3)
```

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

```
myFunction(1)
```

```
## [1] 3
```


Named arguments

All function arguments have names. For `myFunction()`, the argument names are `myFirstArg` and `mySecondArg`.

When calling a function, if you don't specify the argument names then R will just assume the arguments correspond to the arguments in the function definition, in order.

For clarity, it's best practice to explicitly specify the argument names. In fact, this is necessary if you have arguments with defaults sandwiched between arguments that don't have defaults:

Named arguments

```
myThreeArgFunction <- function(x, y=2, z){  
  return(x+y+z)  
}  
myThreeArgFunction(x=2, z=1)
```

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

Trying to run `myThreeArgFunction(2, 1)` will raise an error. R will complain you haven't specified `z`.

Packages

Packages

A **package** is a collection of functions and/or data written by somebody in the R community and made freely available (yay open source) to make some tasks easier.

There are over 20,000 packages available (<https://cran.r-project.org/web/packages/>?) and thus they are not all loaded into R by default! Instead, they must be *installed* first.

Installing packages

To install a package, simply use the `install.packages()` function built into R and follow the instructions. You only need to install a package once (on each machine). This will place its contents onto your computer's hard drive.

```
install.packages("earth")
```

Loading packages

In order to use any functions or data from a package, you must first load it by calling the `library()` function.

```
library(earth)
```

```
## Loading required package: Formula
```

```
## Loading required package: plotmo
```

```
## Loading required package: plotrix
```

```
## Loading required package: TeachingDemos
```

Note there are no quotes around the name of the package when calling `library()`.

The tidyverse

The tidyverse

The **tidyverse** is a large collection of packages, primarily developed by Hadley Wickham. These packages are extremely popular due to their ability to facilitate data wrangling and visualization in a powerful and consistent way.

Although the tidyverse is not a single package, you can install and load it as if it were, using `install.packages("tidyverse")` and `library(tidyverse)`, respectively.

Tibbles

A **tibble** is a special type of data frame which only exists in the tidyverse. Indeed, it is a data frame under the hood, but has certain behaviors that tend to make tibbles a bit more pleasant to work with than ordinary data frames. For more information, see <https://r4ds.had.co.nz/tibbles.html>.

Tidy data

Hadley Wickham established the following 3 principles for “tidy” data:

- 1 Each variable forms a column
- 2 Each observation forms a row
- 3 Each type of observational unit forms a table

Here the term “variable” is used in the scientific sense, not in the sense of an R variable.

Tidy data

Here's an example of non-tidy data:

```
library(tidyverse)

dat <- tibble(Date=c("2024-03-31", "2024-04-01"),
              SFO_high_temp=c("64", "70"),
              SJC_high_temp=c("68", "73"))

dat
```

```
## # A tibble: 2 x 3
##   Date          SFO_high_temp SJC_high_temp
##   <chr>         <chr>         <chr>
## 1 2024-03-31  64             68
## 2 2024-04-01 70             73
```

The problem is that we have one variable, the high temperature, split across 2 columns. People often describe such data as **wide**.

Tidy data

Here's the same data in tidy format:

```
tidy_dat <- tibble(Data=c("2022-03-31", "2022-04-01", "2022-03-31", "2022-04-01"),  
                  Airport=c("SFO", "SFO", "SJC", "SJC"),  
                  High_temp=c("64", "70", "68", "73"))  
tidy_dat
```

```
## # A tibble: 4 x 3  
##   Data      Airport High_temp  
##   <chr>      <chr>   <chr>  
## 1 2022-03-31 SFO      64  
## 2 2022-04-01 SFO      70  
## 3 2022-03-31 SJC      68  
## 4 2022-04-01 SJC      73
```

Tidy data

We can automatically convert from “wide” to “tidy” format using `pivot_longer()` from the `tidyr` package in the tidyverse:

```
auto_tidy_dat <- pivot_longer(data=dat,  
                             names_to="Airport",  
                             values_to="High_temp",  
                             cols=c("SFO_high_temp", "SJC_high_temp"))  
  
auto_tidy_dat
```

```
## # A tibble: 4 x 3  
##   Date      Airport      High_temp  
##   <chr>      <chr>      <chr>  
## 1 2024-03-31 SFO_high_temp 64  
## 2 2024-03-31 SJC_high_temp 68  
## 3 2024-04-01 SFO_high_temp 70  
## 4 2024-04-01 SJC_high_temp 73
```

Conversely, you can use `pivot_wider()` to convert from “tidy” to “wide” format.

Look at the help page for details!