Everything You Need To Know About R For Stats

Comments and Variables

When typing code in R, R ignores anything that has a `#` in front of it. These are called comments. Comments are human-readable text written for other humans. When you write a comment in R it will be in a different color than code--R Studio does this to help you differentiate between comments and code.

```
# Hello! I am a comment
Im_not
```

In the code block above, line two and is a variable because there isn't a `#` in front of it.

```
my_data #comments can exist on the same line as code!
```

Indeed, comments can tell you useful things about the code you wrote to keep track of what you've done.

```
num_cats #this keeps track of how many cats I saw today
```

Variables

The best way to think about a variable is as a special type of box to hold your data. Every variable is a contiguous combination of letters and numbers (i.e. no space in between each character).

```
#These are examples of a single variable per line
onevariable
one_variable
cats
cats2 #try to come up with more helpful variable names so you don't get confused

#R is case sensitive so these two variables are different from each other
Cat
cat

#These are examples of two variables per line
two variables
my_data cats
```

Note: lines 12 and 13 have two variables with no operation to tie them together. This will confuse R because you haven't asked it to do anything to/with those variables.

The Assignment Operator

The assignment operator, `<-`, stores the data to the right of the arrow into the variable sitting on the left. For example, lets store the year Swarthmore was founded in a variable named `Swarthmore`:

```
Swarthmore <- 1864
```

As you can see, you only have to follow the direction of the arrow to see what data is getting stored where.

Data Types

If variables are special containers to hold your data, then a data type is a specific type of container. These containers help R efficiently keep track of your data. Much like you wouldn't put soup on a plate, you should match each data type to its container.

There are 3 data types that you will need to know about for this class:

Data types that deal with words and letters

- *Character* (abbreviated as chr) - The chr data type is any combination of letters and can also include numbers. The data held inside a chr must be enclosed in a string. If your chr isn't enclosed in quotes R will think it's a variable (meaning it's a box holding data instead of the data itself).
    - Examples: `"wug"`, `"problem7"`, `"Stat_21"`

Data types that deal with numbers

- *Integers* (abbreviated as int) - The int data type holds either positive or negative whole numbers.
    - Examples: `6`, `-34`, `0`
- *Double* (abbreviated as dbl) - The dbl data type holds all real numbers. By default R will store your numerical data as a dbl.
    - Examples: `1.6783`, `1`, `-32`, `-15.0000034`

Why Data Types Are Important

R handles most of this under the hood for you, but having a firm grasp of data types is important because you need to communicate to R what sort of data you want to store. See the examples below for why:

Example 1: A chr or dbl?

Look at the following code:

```
x1 <- 7.1
x2 <- "7.1"
```

As you can see, `x1` is of type dbl and `x2` is of type chr because it as quotes around it. Now let's see what happens when we try to multiply both by 2.

```
x1*2
```

> [1] 14.2

```
x2*2
```

> Error in x2 * 2 : non-numeric argument to binary operator

What happened here? Well, `x1*2` is the same as any regular math operation where $x1 * 2 = 14.2 \text{ given that } x1 = 7$ while `x2*2` ran into a data type mismatch. A chr is not a number so R doesn't know how to use it in a mathematical operation. Thus, R threw an error.

Example 2: Which witch is which?

Take a look at the following code:

```
witch <- 100
"witch" <- 100
```

Line 1 looks fine, we're assigning the value of 100 -- the age of our witch -- to the variable `witch`. Line 2 is incorrect because we're assigning the value of 100 to the chr of `"witch"`. That doesn't make a lot of sense because by putting the quotes around *witch* we've turned it into a chr which is not a variable. Variables must not have quotes around them or R will interpret it as chr type data.

Example 3: A chr or dbl?

Take a look at the following code:

```
Mr_Whiskers <- 7 #how old my cat is
cat <- "Mr_Whiskers" #the name of my cat
cat <- Mr_Whiskers
```

So what do you think the variable `cat` holds after line 3?

Well, line 2 stores the chr "Mr_Whiskers" in the variable `cat`. While line 3 overwrites the actions of line 2 and stores the dbl 7 in the variable `cat`. So the answer is 7. Even though both lines 2 and 3 stored *Mr_Whiskers* in the variable cat, one was chr data (line 2) and the other was a variable that contained a dbl (line 3)!

Data Objects

Data objects are comprised of data types. There are five main data structures in R:

1. *atomic vector* - An atomic vector is an array (think a regular old list) of data where everything inside the array is of the same data type.
    - when we say *atomic* we mean that everything in the vector must be of the same data type (e.g. all chrs or all dbls)
2. *list* - A list is a more robust array that can have elements of mixed data types.
3. *matrix* - A matrix is simply an atomic vector with multiple dimensions. All elements in the matrix must be of the same type.
4. *data frame* - A data frame is a special type of list that has multiple dimensions.
5. *factors* - This is different than the previous four and we will expand on it later.

We will focus on the three data types that you will most commonly encounter: lists, data frames, and factors.

Lists

Lists, also known as vectors, are simple 1D arrays that can have multiple types of data within them. This doesn't mean that they will always have multiple different data types. Rather, R will accept a list that contains `7` and `"seven"` and not throw an error.

The following lines of code make legal lists:

```
L1 <- c(1,"1",2,"2",3,"3") # a mixed type list
L2 <- c("cow", "sheep", "horse") # a list of just chrs
L3 <- c(11.0, 1.2, 77.039, 96.030) # a list of just dbls
```

Data Frames

A data frame is the data type you will deal with the most within R. It is the object that will store any data that you read into R. Think of a data frame as a table. Each column in the table is a different variable and each row in the table is an observational unit.

There are a couple important functions that can be used on data frames:

- `head(my_data)` - shows first 6 rows of the data frame.
- `names(my_data)` - displays the names of the variables within the data frame.
- `summary(my_data)` - displays a summary of the data -- min, max, median, etc. -- for each variable contained within the data set.

Accessing Variables Within a Data Frame

The dollar sign (`$`) allows you to extract variables from within a data set. Let's do a short example with the `mtcars` data set which is built into R.

`mtcars` is stored in a data frame. `mtcars` has a variable named `mpg` (something we can find out by using the `names()` function). The following code extracts the variable `mpg` from the data frame `mtcars`

```
mtcars$mpg
```

> [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2

I modified the output to keep it short, but as you can see the code above returns the contents of the variable `mpg`.

Quick Summary Of The Section So Far

Dimensions	Homogenous	Heterogeneous
1D	atomic vector	list
2D+	matrix	data frame

Factors

Factors (abbreviated as fctr) are used to represent categorical variables in R. Once created, factors can only contain a preset collection of values known as levels. These levels are discretized from the data you inputted when creating the factor.

Often times, your categorical variables will already be imported into the data frame and will be stored as a list. You must use the `factor(cat_varb)` function to turn your categorical variable into a factor. By default, R will store your factor's level in alphabetical order. Later on in the course you will learn how to relevel a factor to change the ordering of the levels.

Advanced note on factors

While factors may seem on their surface like a special type of list, or an atomic chr vector, they are neither. Factors are stored as integers with labels associated with each level. This is why you need to ensure that your categorical variables are stored in a factor object.

Basic Operations

Performing Arithmetic Operations On Data

R can also act as a fancy calculator. Within R you can do the following operations with number data:

- Add two pieces of data together with `+`
- Subtract two pieces of data together with `-`
- Multiply two pieces of data together with `*`
- Divide two pieces of data together with `/`
- Exponentiate data with `^` or `**`

We see these basic operations on the following lines:

```
1  # Set up our x and y variables
2  x <- 10
3  y <- 2
4  # Do some basic math
5  x+y
6  x-y
7  x*y
8  x/y
9  x**y
10 x^y
```

> [Line 5] 12
> [Line 6] 8
> [Line 7] 20
> [Line 8] 5
> [Line 9] 100
> [Line 10] 100

I modified the R output a touch so it is easier to match a line of code with its corresponding output. As you can see, math in R works exactly how math works on paper.

A small note on lists

The operations listed above (`+`, `-`, `*`, `/`, `^`) are *element-wise* operators. This means that when you perform these functions on lists, R goes element by element performing the given operation on each corresponding set of elements in each of the lists. Let's look an example, say we have two lists with three elements each. List A contains the numbers $1, 2, 3$ and list B contains the numbers $40, 50, 60$. We would expect $[1, 2, 3] + [40, 50, 60] \rightarrow [1 + 40, 2 + 50, 3 + 60] \rightarrow [41, 52, 63]$. Lets see that in R.

```
# set up our lists
A <- c(1,2,3)
B <- c(40,50,60)
#perform addition
A+B
```

```
[1] 41 52 63
```

Exactly as we expected! We can do the same for `-`, `*`, `/` and `^`.

```
# set up our lists
A <- c(1,2,3)
B <- c(40,50,60)
#perform arithmetic operations
A-B
A*B
A/B
B^A
```

```
[Line 5] -39 -48 -57
[Line 6]  40 100 180
[Line 7] 0.025 0.040 0.050
[Line 8]    40   2500 216000
```

Again, I have slightly modified the R output so that it is easier to match a line of code with its corresponding output.

Order of Operations

R follows the standard PEMDAS (Parentheses, Exponents, Multiplication/Division, Addition/Subtraction) order of operations when evaluating an expression.

```
7+3/2^0
```

```
[1] 10
```

Evaluating Expressions

R evaluates lines of code sequentially. If we were to evaluate the following code below `x1` would end up with the value of 12.

```
x1 <- 3
x1 <- 0
x1 <- 12
```

R also evaluates the right side of the assignment operator (`<-`) before storing it in the left side. After running the code below, `x2` holds the value of 14.

```
x2 <- 7
x2 <- x2*2 # here we do 7*2 = 14 and then store it in x2
```

This has important implications that we will see in the next section.

Moving Data

The Pipe

The pipe (`%>%`) allows you to move your data through a litany of operations. The following two lines of code are equivalent:

```
head(scale(my_data))
my_data %>% scale() %>% head()
```

Especially as you start doing more complex operations in R, using the pipe to move your data from one function to the next makes reading your code -- and understanding what is going on -- much easier.

Please note that "pipes (`%>%`), unlike assignment operators (`<-`), are read from left to right. It's easy enough to keep track of this, just follow the direction of the carrots!" See the example below:

```
my_scaled_data <- my_data %>% scale()
```

In this example, we scaled `my_data` and then saved it in `my_scaled_data`. We can take this one step further. Recalling what we learned in the previous section Evaluating Expressions, we can leverage pipes and how R evaluates lines of code to modify existing data frames.

Say you wanted to remove `year` from your data frame but didn't want to create a new data frame[1]. You could simply do the following:

```
my_data <- my_data %>% select(-year)
```

Instead of saving the modified `my_data` in a new variable, we overwrote the preexisting data frame. Obviously, this is a powerful but dangerous technique and should be used in very limited circumstances.

Packages & Libraries

R comes built in with a whole bunch of neat functionalities. However, people have taken it upon themselves to add even more functionality to R. These come in the form of packages which are just a bundle of extra R functions, data, and complied code. Once these packages are downloaded from the internet, they reside in your computer's local library. Much like a book! In fact, this is quite a useful analogy. Packages, much like ordering a book from a store, only need to be downloaded once. Libraries, much like pulling a book off a shelf, need to be called via a command (`library(package)`) in every file that you want to use something from within a given package.

In this class we will be using several packages. Most commonly, we will be using the packages `tidyverse` and `ggplot2`. As such, near the top of every R or R markdown file you write make sure the following lines of code appear:

```
library("tidyverse")
library("ggplot2")
```

Common Mistakes With Data Types

Consider the code below to plot a graph. (Don't worry if the code doesn't mean anything to you. Your professor will go over how to plot graphs in class)

```
# broken code
mtcars #built in data set with the following varbs: mpg, cyl disp, hp, drat, wt, qsec, vs, am, gear, carb
ggplot(mtcars, aes(x="wt", y=mpg)) +
  geom_point(shape=18, color=blue)+
  geom_smooth(method=lm, se=FALSE, linetype="dashed")
```

On line 3 `"wt"` shouldn't be a chr because it's a variable in `mtcars`, not a chr. On line 3, `blue` should be a chr because it's a chr parameter passed into the `geom_point` function not a variable you created.

```
# fixed code
mtcars #built in data set with the following varbs: mpg, cyl disp, hp, drat, wt, qsec, vs, am, gear, carb
ggplot(mtcars, aes(x=wt, y=mpg)) +
  geom_point(shape=18, color="blue")+
  geom_smooth(method=lm, se=FALSE, linetype="dashed")
```

Helpful Functions and What They Do

Getting info on data sets

- `head(my_data)` - displays the first six lines of a data set
- `names(my_data)` - displays the names of the variables within a data set
- `summarize(mydata)` - summarizes data into single row of values
- `count()` - lets you count the unique values of one or more variables within each group of a categorical variable
    - ex. `my_data %>% count(large_cats)`

Manipulating data sets

- `filter(my_data, condition)` - keeps the data that matches the condition
    - ex. Assuming you have a data set `my_data` that has a variable named `num_cats` you could filter out all rows that saw no cats with the following code `filter(my_data, num_cats>0)`

- `select(my_data, vars)` - can return a specified subset of the data or remove a set of variables from the data
    - ex. `select(my_data, A, B, C)` - selects the variables `A`, `B`, `C` from `my_data` data set
    - ex. `select(my_data, -B)` - removes variable `C` from `my_data` data set
- `mutate()` - appends a new column of data to the original data set
    - ex. Assuming you have a data set `my_data` that had a variable named `num_cats` you could add a variable named `cats_per_hour` with the following code: `my_data %>% mutate(cats_per_hour = (num_cats/60))`
- `mutate_if(if_statement, mutation)` - affects only the columns that satisfy the if-statement
- `mutate_at(c(col1, col2, col3), mutation)` - edits the specific columns with the mutation
- `group_by(my_data, var)` - groups the data according to different "levels" of a categorical variable of your choice.
- `relevel(fct_vect, new-reference-level)` - the levels of a factor are re-ordered so that the level specified by `new-reference-level` is first and the others are moved down
- `fct_infreq(cat_varb)` - orders factors so that the category with the most observational units is the reference variable
- `fct_rev(cat_varb)` - reverses the order of the factor levels
- `fct_recode(cat_varb)` - manually chooses what you want the reference level to be
    - ex. from R documentation:
    - ```
      x <- factor(c("apple", "bear", "banana", "dear"))
      fct_recode(x, fruit = "apple", fruit = "banana")
      ```
- `factor(cat_varb)` - factor levels are arranged in alphabetical order

Standardizing Data

Uses the `mutate_at` and `scale` function to standardize the data

```
my_data_standard <- my_data %>%
                     mutate_at(vars("var1", "var2", "var3", "var4"),
funs(scale))
```

Miscellaneous

- `cor(data_object$xvariable, data_object$yvariable)` - calculates the correlation (strength of the linear relationship) between two quantitative variables
    - Recall correlation is the standardized (is on the domain of [-1,1]) version of the covariance
- `dim(my_data)` - spits out the dimensionality --- i.e. how big -- a table or dataset is

Works Referenced

Adler, J. (2012). *R in a Nutshell*. O'Reilly.
```

John Blischak, Daniel Chen, Harriet Dashnow, and Denis Haine (eds): "Software Carpentry: Programming with R". Version 2016.06.

R Documentation and manuals: www.rdocumentation.org

Thornton, S. (2020). *Class 5 Part 2* [Stat 021 F'2020 Class Slides]. Retrieved from Professor Thornton's Website

---

1. It is strongly recommended that you create a new data frame every time you modify it. ↩