Programming Basics

Alejandro Schuler, adapted from Steve Bagley and based on R for Data Science by Hadley Wickham
2019

Learning Goals:

- save values to variables
- find and call R functions with multiple arguments by position and name
- recognize vectors and vectorized functions
- recognize and inspect data frames
- issue commands to R using the Rstudio script pane

Programming Basics

- We've seen code like

```
genes = read_csv("https://raw.githubusercontent.com/alejandroschuler/r4ds-courses/advance-2020/data/lupusGenes.csv")
```

- We know this reads a .csv from a file and creates something called a “data frame”
- We've been using this data frame in code like

```
ggplot(genes) +
  geom_bar(aes(x = ancestry, fill = phenotype))
```

- But what does this syntax really mean? Is it useful outside of making plots?

Assignment

Assignment

- To do complex computations, we need to be able to give names to things.

```
mpg = read_csv("https://raw.githubusercontent.com/alejandroschuler/r4ds-courses/advance-2020/data/mpg.csv")
```

- This code *assigns* the result of running `read_csv("https://raw.githubusercontent.com/alejandroschuler/r4ds-courses/advance-2020/data/mpg.csv")` to the name `mpg`
- You can do this with any values and/or functions

```
x = (13 + 7)/2
```

- R prints no result from this assignment, but what you entered causes a side effect: R has stored the association between x and the result of this expression (look at the Environment pane.)

Using the value of a variable

```
x
[1] 10
x / 5
[1] 2
```

- When R sees the name of a variable, it uses the stored value of that variable in the calculation.
- Here R uses the value of x, which is 10.
- We can break complex calculations into named parts. This is a simple, but very useful kind of abstraction.

Two ways to assign

In R, there are (unfortunately) two assignment operators. They have subtly different meanings (more details later).

- <– requires that you type two characters. Don't put a space between < and –. (What would happen?)
- RStudio hint: Use "`Option –`" (Mac) or "`Alt –`" (PC) to type this using one key combination.
- = is easier to type.
- You will see both used throughout R and user code.

```
x <- 10
x
[1] 10
x = 20
x
[1] 20
```

Assignment has no undo

```
x = 10
x
[1] 10
x = x + 1
x
[1] 11
```

- If you assign to a name with an existing value, that value is overwritten.
- There is no way to undo an assignment, so be careful in reusing variable names.

Naming variables

- It is important to pick meaningful variable names.
- Names can be too short, so don't use `x` and `y` everywhere.
- Names can be too long (`Main.database.first.object.header.length`).
- Avoid silly names.
- Pick names that will make sense to someone else (including the person you will be in six months).
- ADVANCED: See `?make.names` for the complete rules on what can be a name.

Case matters for names in R

```
a = 1
A # this causes an error because A does not have a value
```

```
Error: object 'A' not found
```

- R cares about upper and lower case in names.
- We also see that some error messages in R are a bit obscure.

More about naming

There are different conventions for constructing compound names. Warning: disputes over the right way to do this can get heated.

```
stringlength
string.length
StringLength
stringLength
string_length (underbar)
string-length (hyphen)
```

- To be consistent with the packages we will use, I recommend snake_case where you separate lowercase words with _
- Note that R itself uses several of these conventions.
- One of these won't work. Which one and why?

R saves some names for itself

```
for = 7 # this causes an error
```

- `for` is a reserved word in R. (It is used in loop control.)
- ADVANCED: see `?Reserved` for the complete rules.

Exercise: birth year

- Make a variable that represents the age you will be at the end of this year
- Make a variable that represents the current year
- Use them to compute the year of your birth and save that as a variable
- Print the value of that variable

Functions

Calling built-in functions

- To call a function, type the function name, then the argument or arguments in parentheses. (Use a comma to separate the arguments, if more than one.)

```
sqrt(2)
[1] 1.414214
```

Functions and variable assignment

```
x = 4
sqrt(x)
[1] 2
x
[1] 4
y = sqrt(x)
y
[1] 2
x = 10
y
[1] 2
```

- What do you observe?

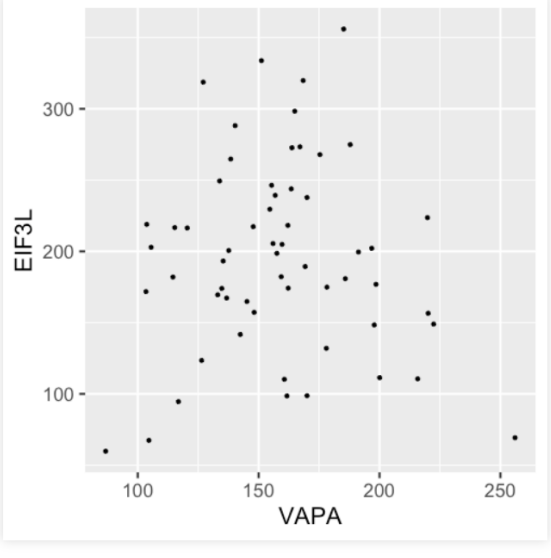
Functions and variable assignment

- functions generally do not affect the variables you pass to them (`x` remains the same after `sqrt(x)`)
- The results of a function call will simply be printed out if you do not save the result to a variable
- Saving the result to a variable lets you use it later, like any other variable you define manually
- Once a variable has been assigned (`y`), it keeps its value until updated, even if you change other variables (`x`) that went into the original assignment of that variable

Arguments by position vs. name

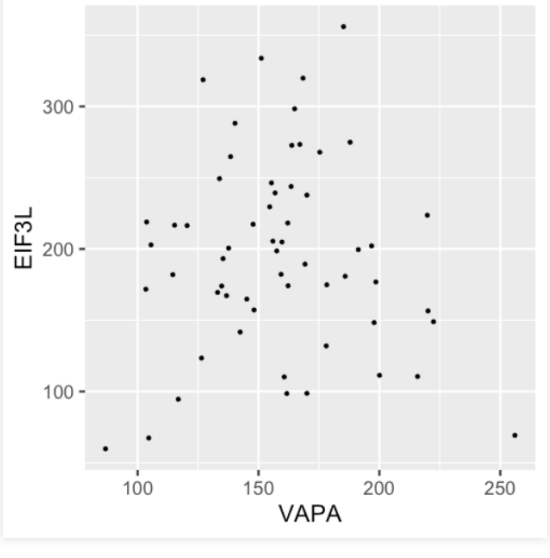
- Arguments can be specified by position, with one supplied argument for each name in the function parameter list, and in the same order

```
ggplot(genes) +
  geom_point(aes(VAPA, EIF3L))
```



- Sometimes, arguments can be supplied by name using the syntax, variable = value.
- When using names, the order of the named arguments does not matter.

```
ggplot(data=genes) +
  geom_point(mapping=aes(y=EIF3L,
x=VAPA))
```



Optional arguments

- Many R functions have arguments that you don't always have to specify. For example:

```
file_name = "https://raw.githubusercontent.com/alejandroschuler/r4ds-courses/advance-2020/data/lupusGenes.csv"
genes_10 = read_csv(file_name, n_max=10) # only read in 10 rows
genes = read_csv(file_name)
```

- `n_max` tells `read_csv()` to only read the first 10 rows of the dataset.
- If you don't specify it, it defaults to infinity (i.e. R reads until there are no more lines in the file).

Why?

- What are the benefits/drawbacks of using positional vs. named arguments?

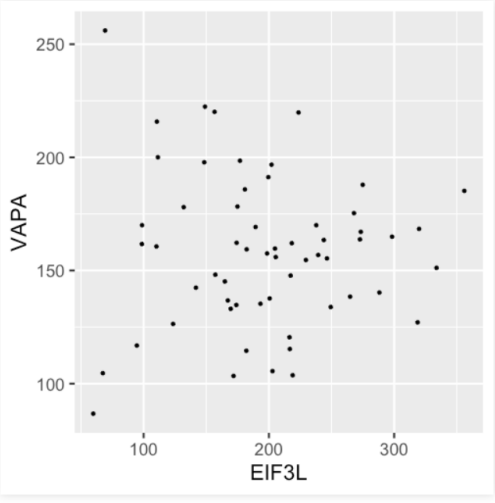
Exercise

Why does this code generate errors?

```
Warning: Ignoring unknown aesthetics: y_axis, x_axis
Error in FUN(X[[i]], ...): object 'EIF3L' not found
```

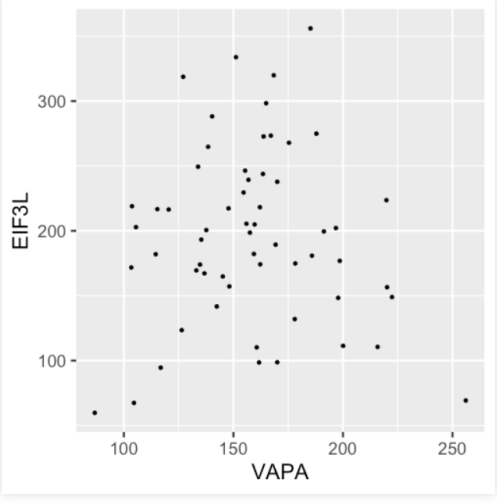
Exercise

I'm trying to generate this plot:



But when I use this code, I get:

```
ggplot(data=genes) +
  geom_point(aes(VAPA, EIF3L))
```



What am I doing wrong?

Finding the names of a function's arguments

`read_csv()` takes a number of optional named arguments. What are some of them?

Calling functions from a package

- Sometimes packages introduce name conflicts, which is when the pacakge loads a function that is named the same thing as a function that's already in the environment
- Typically, the package being loaded will take precedence over what is already loaded.
- For instance:

```
?filter # returns documentation for a function called filter in the stats package
library(dplyr)
?filter # now returns documentation for a function called filter in the dplyr
package!
```

- You can tell R which function you want by specifying the package name and then `::` before the function name

```
?stats::filter
?dplyr::filter
```

Vectors

Repetitive calculations

```
x1 = 1
x2 = 2
x3 = 3
```

Let's say I have these variables and I want to add 1 to all of them and save the result.

```
y1 = 1 + x1
y2 = 2 + x2
y3 = 3 + x3
```

This does the trick but it's a lot of copy-paste

Vectors

- Vectors solve the problem

```
x = c(1,2,3)
y = x + 1
y
[1] 2 3 4
```

- A vector is a one-dimensional sequence of zero or more values
- Vectors are created by wrapping the values separated by commas with the `c()` function, which is short for "combine"
- Many R functions and operators (like +) automatically work with multi-element vector arguments.

Ranges

```
1:50
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
[26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
```

- The colon `:` is a handy shortcut to create a vector that is a sequence of integers from the first number to the second number (inclusive).
- Long vectors wrap around. (Your screen may have a different width than what is shown here.)
- Look at the `[ ]` notation. The second output line starts with 23, which is the 24^{th} element of the vector.
- This notation will help you figure out where you are in a long vector.

Elementwise operations on a vector

- This multiplies each element of `1:10` by the corresponding element of `1:10`, that is, it squares each element.

```
(1:10)*(1:10)
 [1]   1   4   9  16  25  36  49  64  81 100
```

- Equivalently, we could use exponentiation:

```
(1:10)^2
 [1]   1   4   9  16  25  36  49  64  81 100
```

- Many basic R functions operate on multi-element vectors as easily as on vectors containing a single number.

```
sqrt(0:10)
 [1] 0.000000 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751
 [9] 2.828427 3.000000 3.162278
```

Some functions operate on vectors and give back a single number

```
numbers <- c(9, 12, 6, 10, 10, 16, 8, 4)
numbers
[1]  9 12  6 10 10 16  8  4
sum(numbers)
[1] 75
sum(numbers)/length(numbers)
[1] 9.375
mean(numbers)
[1] 9.375
```

Vectors of other types

- text data in R is called a "string"

```
my_string = "hello"
```

- when using data that is text in R, you have to refer to it using quotation marks (why?)

```
my_string = hello # what does this code do?
```

- you can have a vector of strings, and functions can operate on these too:

```
words = c("hello", "how", "are", "you", "?")
paste(words, collapse=" ")
[1] "hello how are you ?"
```

Exercise: subtract the mean

```
x = c(7, 3, 1, 9)
```

- Subtract the mean of `x` from `x`, and then `sum` the result.

Exercise: a vector of variables

- Predict the output of the following code:

```
a = 1
b = 2
x = c(a,b)

a = 3
print(x)
```

Data Frames

Making data frames

- use `tibble()` to make your own data frames from scratch in R

```
my_data = tibble(
  person = c("carlos", "nathalie", "christina", "alejandro"),
  age = c(33, 48, 8, 29)
)
my_data
# A tibble: 4 x 2
  person      age
  <chr>     <dbl>
1 carlos       33
2 nathalie     48
3 christina     8
4 alejandro    29
```

Data frame properties

- `dim()` gives the dimensions of the data frame. `ncol()` and `nrow()` give you the number of columns and the number of rows, respectively.

```
dim(my_data)
[1] 4 2
ncol(my_data)
[1] 2
nrow(my_data)
[1] 4
```

- `names()` gives you the names of the columns (a vector)

```
names(my_data)
[1] "person" "age"
```

Data frame properties

- `glimpse()` shows you a lot of information, `head()` returns the first `n` rows

```
glimpse(my_data)
Rows: 4
Columns: 2
$ person <chr> "carlos", "nathalie", "christina", "alejandro"
$ age    <dbl> 33, 48, 8, 29
head(my_data, n=2)
# A tibble: 2 x 2
  person     age
  <chr>    <dbl>
1 carlos      33
2 nathalie    48
```

Missing Data

NA

- R has a special value that represents missing data- it's called `NA`

```
c(1,2,NA,4)
[1]  1  2 NA  4
```

- NA can appear anywhere that R would expect some other kind of data
- NA usually ruins computations:

```
1 + NA + 3
[1] NA
```

- The result makes sense because if I don't know what I'm adding together, I don't know the result either
- some functions have options to ignore the missing values in vectors:

```
mean(c(1,2,NA,4), na.rm=TRUE)
[1] 2.333333
```

Scripts

Using the script pane

- Writing a series of expressions in the console rapidly gets messy and confusing.
- The console window gets reset when you restart RStudio.
- It is better (and easier) to write expressions and functions in the script pane (upper left), building up your analysis.
- There, you can enter expressions, evaluate them, and save the contents to a .R file for later use.
- Look at the RStudio “Code" menu for some useful keyboard commands.

Script pane example

- Create a script pane: File > New File > R Script
- Put your cursor in the script pane.
- Type: `factorial(1:10)`
- Then hit `Command-RETURN` (Mac), or `Ctrl-ENTER` (Windows).
- That line is copied to the console pane and evaluated.
- You can save the script to a file.
- Explore the RStudio Code menu for other commands.

Adding comments

```
## This is a comment
1 + 2 # add some numbers
[1] 3
```

- Use a # to start a comment.
- A comment extends to the end of the line and is ignored by R.

Exercise: Plotting a parabola

Write an R script that starts with:

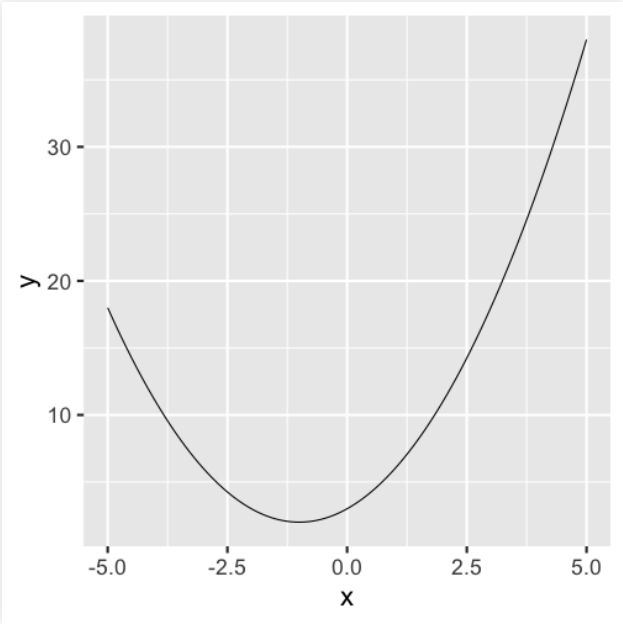
```
A = 1
B = 2
C = 3
```

In the rest of the script, do the following:

- generate an evenly-spaced sequence of 100 values between -5 and 5 (find an R function that does this). Call this `x`
- generate the corresponding y-values `y` by computing the formula $y = Ax^2 + Bx + C$
- create a data frame with `x` and `y` as columns
- use ggplot to create a line plot of `x` vs `y`

Run your script to see the generated plot. Try changing the values of `A`, `B`, and `C` at the top of the script and re-running to see how the plot changes.

Your result should look like:



RStudio Pro-tip: multicursors

- RStudio's script pane supports multi-cursors! Hold `alt` and drag your mouse up and down
- You can also set a keyboard shortcut for `quick add next`
- These features make it much easier to rename variables, etc.
- You should also be aware of `cmd-<arrow>` and `alt-<arrow>` for moving the cursor (by line and by word)
- and `cmd-shift-<arrow>` and `alt-shift-<arrow>` for selecting text (by line and by word)