R Language Basics

(Part 1)

Barry Grant

UC San Diego

http://thegrantlab.org/teaching/

Background

R is a powerful data programming language that you can use to explore and understand data in an open-ended, highly interactive, iterative way.

Learning R will give you the freedom to experiment and problem solve during data analysis — exactly what we need as bioinformaticians.

Outline

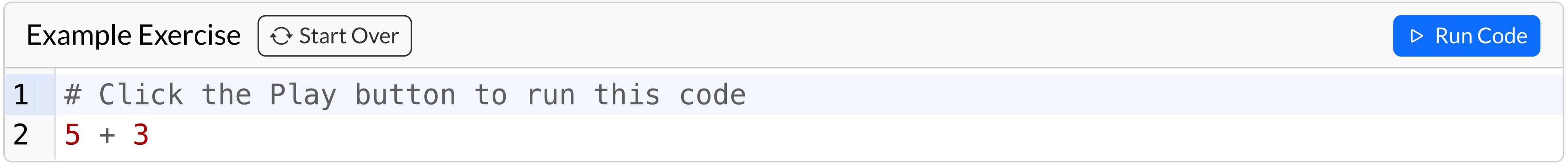
Before delving into working with real data in R, we need to learn some basics of the R language. In this section, we'll learn how to do simple calculations in R, assign values to variables, and call functions.

Then, in part 2, we'll look in more detail at R's vectors and data.frames, along with the critically important topic of vectorization that underpins how we approach many problems in R.

Logistics

Below is a interactive “Example Exercise” **code chunk** that you can edit and run by clicking the > Run Code button.

Give it a try:



```
1 # Click the Play button to run this code
2 5 + 3
```

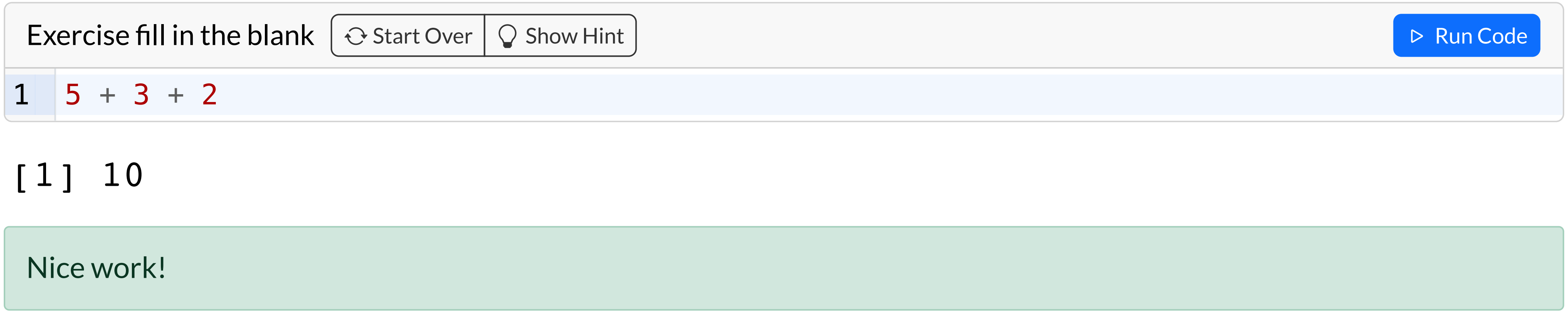
```
[1] 8
```

Note that you can click on line 2 and edit the code if you wish.

Checking your understanding

We will use these types of "code chunks" to ask you to complete some missing code, typically indicated by the blank _____ place holder.

For example, complete the code below so your answer sums to 10:



```r
5 + 3 + 2
```

```
[1] 10
```

Nice work!

Where are we going with this...

OK, so R is a big calculator but it is also so much more than that. For example, we will see soon that R can do complicated statistical analysis on big datasets that you would never want to use a calculator for.

What's more R allows you to do your analysis in a robust and re-usable way so you can automate things later.

At the heart of this is saving your answers as you go along (a.k.a. **object assignment**).

Object assignment - saving your answers

Lets make an assignment and then inspect the object you just created.

Exercise Start Over Run Code

```r
x <- 3 * 4
x
```

```
[1] 12
```

> All R statements where you create objects have this form:
>
> `objectName <- value`
>
> and in my head I hear, e.g., “x gets 12”.

🔑 Key point:

You will make lots of assignments and the little arrow operator `<-` is a pain to type. **Don't be lazy and use `=`**, although it would work, because it will just sow confusion later when we learn about functions.

> In *RStudio* folks will often utilize the keyboard shortcut: `Alt` + `-` (that is pushing the `Alt` and `minus` keys together or the `Option` and `minus` keys if you are on a Mac).

Side note about object names...

Object names cannot start with a digit and can not contain certain other “special” characters such as a comma or a space.

You will be wise to adopt a convention for demarcating words in names so as to avoid spaces, for example:

```
i_use_snake_case
other.people.use.periods
evenOthersUseCamelCase
only-crazy-folks-use-kabab-case
```

Typing fedility matters

Let’s make another assignment

R Code Start Over Run Code

```r
r_rocks <- 2 ^ 3
```

and then try to inspect this object:

Exercise Start Over Run Code

```r
str(r_rocks)
```

```
 num 8
```

What happened? What about the 2nd line? Did you run both chunks? Can you fix this?

🔑 Key point:

When we program we enter into an implicit contract with the computer. The computer will do tedious computation for you. In return, you will be completely precise in your instructions.

Typos matter. Case matters. The order that you do things matters.

Therefore get better at typing!

Functions

Being productive in R is all about using functions - we call functions when we read a dataset, perform a statistical analysis, make a fancy visualization or do just about anything else.

All functions are accessed like so:

```
functionName(arg1 = val1, arg2 = val2, and so on)
```

Notice that we use the = sign here for setting function inputs (a.k.a. setting function “arguments”) and that each one is separated by a comma.

Calling functions

Let’s try using the `seq()` function, which makes regular sequences of numbers:

Exercise Start Over Run Code

```r
seq(1,10)
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

You can always access the help/documentation of a particular function by using the `help(myFunction)` function, e.g. `help(seq)`. Try it out above.

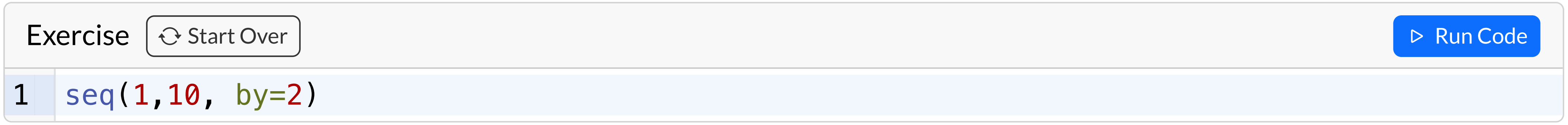
🔑 Key point:

The previous `seq(1, 10)` example above also demonstrates something about how R resolves function arguments.

You can always specify in `name = value` form e.g. `seq(from=1, to=10, by=1)`. But if you do not, R attempts to resolve by position (1st argument, 2nd argument, etc.).

So in `seq(1, 10)` it is assumed that we want a sequence `from = 1` that goes `to = 10`. Since we didn’t specify step size, the default value of `by` in the function definition is used, which ends up being 1 in this case.

What does the following command do?



```r
seq(1,10, by=2)
```

```
[1] 1 3 5 7 9
```

Run it and see. Could you of figured out this result from the documentation accessed via the command `help(seq)`?

Side note

One very useful part of a functions documentation is the **Examples** section (typically found at the very end of the documentation entry).

You can copy and paste these into your session or execute them all in one go with the `examples()` function, e.g.

R Code Start Over Run Code

```r
example(seq)
```

```
seq> seq(0, 1, length.out = 11)
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0

seq> seq(stats::rnorm(20)) # effectively 'along'
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20

seq> seq(1, 9, by = 2)     # matches 'end'
[1] 1 3 5 7 9
```

🔑 Key point:

R has built in help accessible with the question mark or `help()` function.

However, it is rather heavy on nerd-speak and takes some getting used to.

> Remember Google and ChatGPT are your friends. Asking Google and ChatGPT is often the quickest way to get the help you need before you are full on nerd-speak fluent.

A brief note about vectors

We will go into more details of vectors later 😉

Vectors

Vectors are the most fundamental data structures in R. Their job is to hold a contiguous collection of “*elements*” of the same “*type*”.

For example a set of numbers (what we call **numeric** type), words (**character** type) or True and False vales (**logical** type).

To create vectors we combine values with the function `c()`:

Exercise Start Over Run Code

```r
x <- c(1, 5, 3, 4, 2)
y <- c("G","A","C","A")
z <- c(TRUE, FALSE, TRUE, TRUE)
x + 10
```

```
[1] 11 15 13 14 12
```

Vectorization

Vectors are the basis of one of R’s most important features: **vectorization**. Vectorization allows us to loop over all elements in a vector without the need to write an explicit loop.

For example, R’s arithmetic operators (and tones of other functions) are all vectorized:

Exercise Start Over Run Code

```r
x <- c(1, 10, 5)
x + 10
```

```
[1] 11 20 15
```

Exercise Start Over Run Code

```r
x <- c(1, 10, 5)
log(x)/2 + 1
```

```
[1] 1.000000 2.151293 1.804719
```

🔑 Key point:

Unlike other languages, R allows us to completely forgo explicitly looping over vectors with a for loop.

> Later on, we'll see other methods used for more explicit looping. For now I want you to appreciate that the *vectorized approach* is not only more **clear** and **readable** but it's also computationally **faster**.

Indexing - getting to subsets of your data

An index is a number (or logical) vector that specifies which element, or subset of elements, in a vector you want to retrieve.

Basically, we use **indexing** to get to certain elements of a vector:

Exercise Start Over Run Code

```r
# Access the 2nd element of x
x <- c(56, 95.3, 0.4)
x[2]
```

```
[1] 95.3
```

Overwritting vector elements

We can change specific vector elements by combining indexing and assignment. For example:

Exercise Start Over Run Code

```
x <- c(56, 95.3, 0.4)
x[3] <- 0.5
x
```

```
[1] 56.0 95.3  0.5
```

Our next session will cover more on **vectors**, and the other major R data structures (**data.frames** and **lists**).

Data.frames combine vectors into tables

A data.frame is a table (or two-dimensional array-like structure) where columns can contain different types of data (numeric, character, logical, etc.).

Data.frames are ideal for managing and analyzing datasets with rows and columns like we find in spreadsheets.

Exercise Start Over Run Code

```r
df <- data.frame(column1 = c("A", "B", "C"), column2 = c(10, 200, 3))
df
```

```
  column1 column2
1       A      10
2       B     200
3       C       3
```

Indexing a data.frame

We can use numeric or logical index values for subseting a data.frame, similar to those we used for vectors but with one number for rows and the second for columns.

Exercise Start Over Run Code

```r
df <- data.frame(column1 = c("A", "B", "C"), column2 = c(10, 200, 3))
df[1,]  # return all row one values
df[1,2] # return row one column two value
```

```
  column1 column2
1       A      10
```

```
[1] 10
```

You can also access specific columns via their column name:

Exercise Start Over Run Code

```r
df <- data.frame(column1 = c("A", "B", "C"), column2 = c(10, 200, 3))
df$column1
```

```
[1] "A" "B" "C"
```

Errors, warnings, and messages

R will show red text in the console pane in three different situations:

- **Errors**: When the red text is a legitimate error, it will be prefaced with "Error in…" and will try to explain what went wrong. Generally when there's an error, the code will not run.
- **Warnings**: When the red text is a warning, it will be prefaced with "Warning:' and R will try to explain why there's a warning. Generally your code will still work, but with some caveats.
- **Messages**: When the red text doesn't start with either "Error" or "Warning", it's just a friendly message. You'll see these messages when you load R packages for example.

🔑 Key point:

R reports errors, warnings, and messages in a glaring red font, which makes it seem like R is scolding you.

Don’t panic, it doesn’t necessarily mean anything is wrong and indeed it is not always a bad thing. However, you should read this text to make sure things are as you expect.

Over time we will learn more “R speak” and get more comfortable generally working in the console. Again Google and ChatGPT can help you interpret these important messages.

End of Part 1

Well done for getting this far !

Feel free to close this now 😀

R Language Basics

Hands-on Lab Session

Barry Grant

UC San Diego

http://thegrantlab.org