Before we start

Data Carpentry contributors

Learning Objectives

- Articulating motivations for this lesson
- Introduce participants to the RStudio interface
- Set up participants to have a working directory with a `data/` folder inside
- Introduce R syntax
- Point to relevant information on how to get help, and understand how to ask well formulated questions

---

Presentation of RStudio

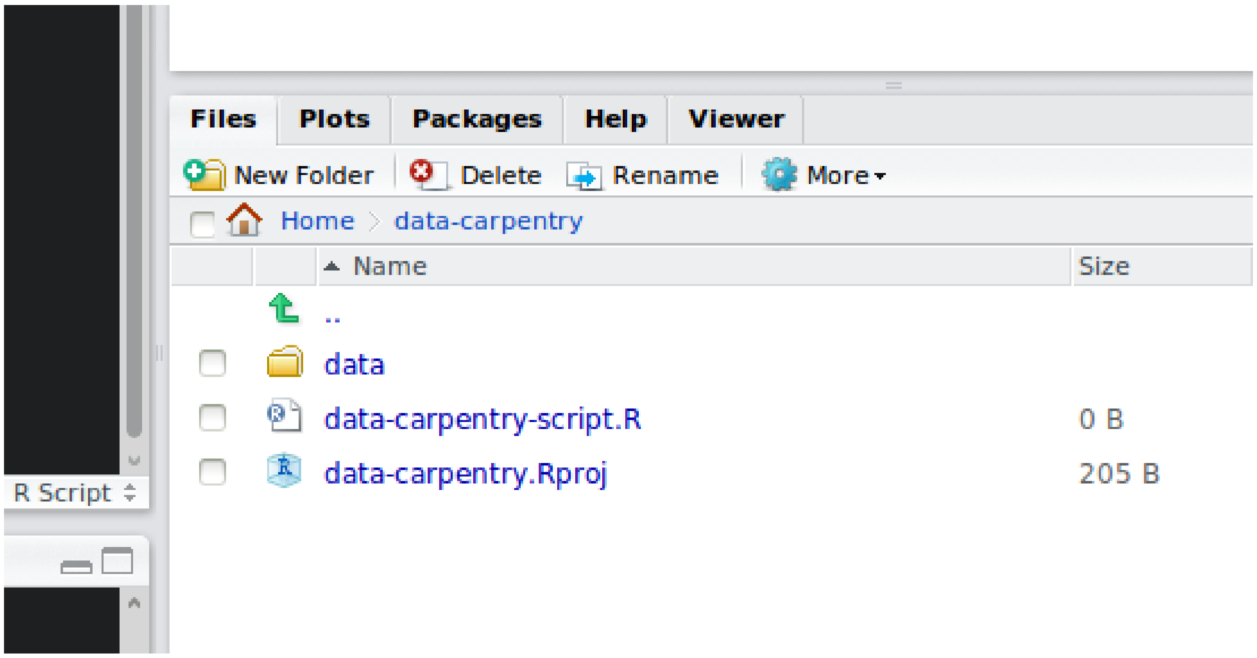
Let's start by learning about our tool.

- Console, Scripts, Environments, Plots
- Code and workflow are more reproducible if we can document everything that we do.
- Our end goal is not just to "do stuff" but to do it in a way that anyone can easily and exactly replicate our workflow and results.

Before we get started

- Under the `File` menu, click on `New project`, choose `New directory`, then `Empty project`
- Enter a name for this new folder, and choose a convenient location for it. This will be your **working directory** for the rest of the day (e.g., `~/data-carpentry`)
- Click on "Create project"
- Under the `Files` tab on the right of the screen, click on `New Folder` and create a folder named `data` within your newly created working directory. (e.g., `~/data-carpentry/data`)
- Create a new R script (File > New File > R script) and save it in your working directory (e.g. `ds-workshop-script.R`)

Your working directory should now look like this:



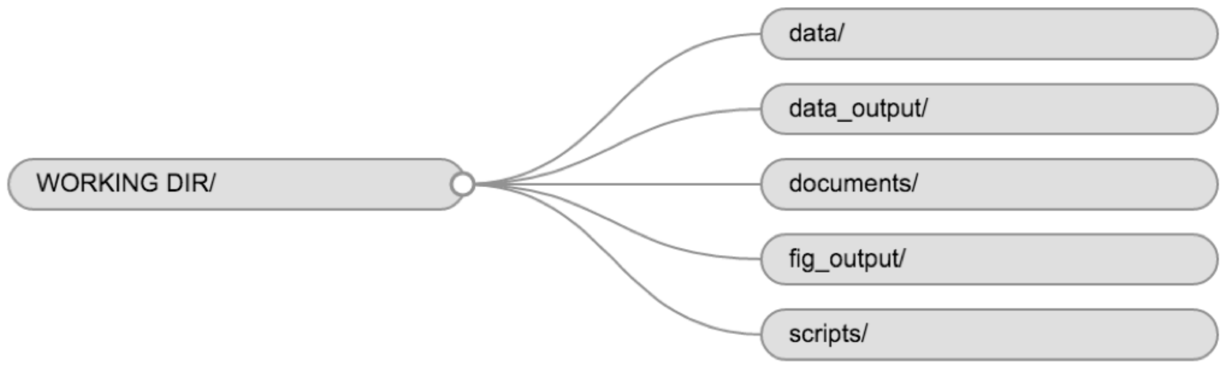
Name	Size
data	
data-carpentry-script.R	0 B
data-carpentry.Rproj	205 B

Organizing your working directory

Using the same folder structure across your projects (working directory) will help keep things organized, and will also make it easy find/file things in the future. This can be especially helpful when you have multiple projects. In general, you may create directories (folders) for **scripts**, **data**, and **documents**.

- `data/` You should separate the original data (raw data) from intermediate datasets that you may create for the need of a particular analysis. For instance, you may want to create a `data/` directory within your working directory that stores the raw data, and have a `data_output/` directory for intermediate datasets and a `figure_output/` directory for the plots you will generate.
- `documents/` This would be a place to keep outlines, drafts, text.
- `scripts/` This would be the location to keep your R scripts for different analyses or plotting, and potentially a separate folder for your functions (more on that later!)

You may want additional directories or subdirectories depending on your project needs, but these should form the backbone of your working directory. For this workshop, you only need a `data/` folder.



Interacting with R

The basis of programming is that we write down instructions for the computer to follow, and then we tell the computer to follow those instructions. We write, or *code*, instructions in R because it is a common language that both the computer and we can understand. We call the instructions *commands* and we tell the computer to follow the instructions by *executing* (also called *running*) those commands.

There are two main ways of interacting with R: using the console or by using script files (plain text files that contain your code).

The console window (in RStudio, the bottom left panel) is the place where R is waiting for you to tell it what to do, and where it will show the results of a command that has been executed. You can type commands directly into the console and press `Enter` to execute those commands, but they will be forgotten when you close the session. It is better to enter the commands in the script editor, and save the script. This way, you have a complete record of what you did, you can easily show others how you did it and you can do it again later on if needed. Rstudio allows you to execute commands directly from the script editor by using the `Ctrl-Enter` shortcut. The command on the current line in the script or all of the commands in the currently selected text will be sent to the console and executed when you press `Ctrl-Enter`.

At some point in your analysis you may want to check the content of variable or the structure of an object, without necessarily keep a record of it in your script. You can type these commands and execute them directly in the console. RStudio provides the `Ctrl-1` and `Ctrl-2` shortcuts allow you to jump between the script and the console windows.

If R is ready to accept commands, the R console shows a `>` prompt. If it receives a command (by typing, copy-pasting or sent from the script editor using `Ctrl-Enter`), R will try to execute it, and when ready, show the results and come back with a new `>`-prompt to wait for new commands.

If R is still waiting for you to enter more data because it isn't complete yet, the console will show a `+` prompt. It means that you haven't finished entering a complete command. This is because you have not 'closed' a parenthesis or quotation, i.e. you don't have the same number of left-parentheses as right-parentheses, or the same number of opening and closing quotation marks. If you're in Rstudio and this happens, click inside the console window and press `Esc`; this will cancel the incomplete command and return you to the `>` prompt.

Basics of R

By this time should be covered in intro presentation to the lesson R is a versatile, open source programming/scripting language that's useful both for statistics but also data science. Inspired by the programming language S.

- Free/Libre/Open Source Software under the GPL.
- Superior (if not just comparable) to commercial alternatives. R has over 7,000 user contributed packages at this time. It's widely used both in academia and industry.
- Available on all platforms.
- Not just for statistics, but also general purpose programming.
- For people who have experience in programmming: R is both an object-oriented and a so-called functional language.
- Large and growing community of peers.

Commenting

Use `#` signs to comment. Comment liberally in your R scripts. Anything to the right of a `#` is ignored by R, meaning it won't be executed.

Assignment operator

`<-` is the assignment operator. It assigns values on the right to objects on the left. So, after executing `x <- 3`, the value of `x` is `3`. The arrow can be read as 3 **goes into** `x`. For historical reasons, you can also use `=` for assignments, but not in every context. Because of the slight differences in syntax, it is good practice to use always `<-` for assignments, except when specifying the values of arguments in functions, when only `=` should be used, see below.

In RStudio, typing Alt + - (push Alt at the same time as the - key) will write `<-` in a single keystroke.

Functions and their arguments

Functions are "canned scripts" that automate something complicated or convenient or both. Many functions are predefined, or can be made available by importing R *packages* (more on that later). A function usually gets one or more inputs called *arguments*. Functions often (but not always) return a *value*. A typical example would be the function `sqrt()`. The input (the argument) must be a number, and the return value (in fact, the output) is the square root of that number. Executing a function ('running it') is called *calling* the function. An example of a function call is:

```
b <- sqrt(a)
```

Here, the value of `a` is given to the `sqrt()` function, the `sqrt()` function calculates the square root, and returns the value which is then assigned to variable `b`. This function is very simple, because it takes just one argument.

The return 'value' of a function need not be numerical (like that of `sqrt()`), and it also does not need to be a single item: it can be a set of things, or even a data set. We'll see that when we read data files in to R.

Arguments can be anything, not only numbers or filenames, but also other objects. Exactly what each argument means differs per function, and must be looked up in the documentation (see below). Some functions take arguments which may either be specified by the user, or if left out, take on a *default* value: these are called *options*. Options are typically used to alter the way the function operates, such as whether it ignores 'bad values', or what symbol to use in a plot. However, if you want something specific, you can specify a value of your choice which will be used instead of the default.

Let's try a function that can take multiple arguments: `round`.

```
round(3.14159)
```

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

Here, we've called `round` with just one argument, `3.14159`, and it has returned the value `3`. That's because the default is to round to the nearest whole number. If we want more digits we can see how to do that by getting information about the `round` function. We can use `args(round)` or look at the help for this function using `?round`.

```
args(round)
```

```
## function (x, digits = 0)
## NULL
```

```
?round
```

We see that if we want a different number of digits, we can type `digits=2` or however many we want.

```
round(3.14159, digits=2)
```

```
## [1] 3.14
```

If you provide the arguments in the exact same order as they are defined you don't have to name them:

```
round(3.14159, 2)
```

```
## [1] 3.14
```

And if you do name the arguments, you can switch their order:

```
round(digits=2, x=3.14159)
```

```
## [1] 3.14
```

It's good practice to put the non-optional arguments (like the number you're rounding) first in your function call, and to specify the names of all optional arguments. If you don't, someone reading your code might have to look up definition of a function with unfamiliar arguments to understand what you're doing.

Seeking help

I know the name of the function I want to use, but I'm not sure how to use it

If you need help with a specific function, let's say `barplot()`, you can type:

```r
?barplot
```

If you just need to remind yourself of the names of the arguments, you can use:

```r
args(lm)
```

I want to use a function that does X, there must be a function for it but I don't know which one...

If you are looking for a function to do a particular task, you can use `help.search()` function, which is called by the double question mark `??`. However, this only looks through the installed packages for help pages with a match to your search request

```r
??kruskal
```

If you can't find what you are looking for, you can use the rdocumention.org website that search through the help files across all packages available.

I am stuck... I get an error message that I don't understand

Start by googling the error message. However, this doesn't always work very well because often, package developers rely on the error catching provided by R. You end up with general error messages that might not be very helpful to diagnose a problem (e.g. "subscript out of bounds"). If the message is very generic, you might also include the name of the function or package you're using in your query.

However, you should check StackOverflow. Search using the `[r]` tag. Most questions have already been answered, but the challenge is to use the right words in the search to find the answers: http://stackoverflow.com/questions/tagged/r

The Introduction to R can also be dense for people with little programming experience but it is a good place to understand the underpinnings of the R language.

The R FAQ is dense and technical but it is full of useful information.

Asking for help

The key to get help from someone is for them to grasp your problem rapidly. You should make it as easy as possible to pinpoint where the issue might be.

Try to use the correct words to describe your problem. For instance, a package is not the same thing as a library. Most people will understand what you meant, but others have really strong feelings about the difference in meaning. The key point is that it can make things confusing for people trying to help you. Be as precise as possible when describing your problem.

If possible, try to reduce what doesn't work to a simple reproducible example. If you can reproduce the problem using a very small `data.frame` instead of your 50,000 rows and 10,000 columns one, provide the small one with the description of your problem. When appropriate, try to generalize what you are doing so even people who are not in your field can understand the question.

To share an object with someone else, if it's relatively small, you can use the function `dput()`. It will output R code that can be used to recreate the exact same object as the one in memory:

```r
dput(head(iris)) # iris is an example data.frame that comes with R
```

```
## structure(list(Sepal.Length = c(5.1, 4.9, 4.7, 4.6, 5, 5.4),
##     Sepal.Width = c(3.5, 3, 3.2, 3.1, 3.6, 3.9), Petal.Length = c(1.4,
##     1.4, 1.3, 1.5, 1.4, 1.7), Petal.Width = c(0.2, 0.2, 0.2,
##     0.2, 0.2, 0.4), Species = structure(c(1L, 1L, 1L, 1L, 1L,
##     1L), .Label = c("setosa", "versicolor", "virginica"), class = "factor")), .Names = c("Sepal.Leng
## "Sepal.Width", "Petal.Length", "Petal.Width", "Species"), row.names = c(NA,
## 6L), class = "data.frame")
```

If the object is larger, provide either the raw file (i.e., your CSV file) with your script up to the point of the error (and after removing everything that is not relevant to your issue). Alternatively, in particular if your questions is not related to a `data.frame`, you can save any R object to a file:

```r
saveRDS(iris, file="/tmp/iris.rds")
```

The content of this file is however not human readable and cannot be posted directly on stackoverflow. It can however be sent to someone by email who can read it with this command:

```r
some_data <- readRDS(file="~/Downloads/iris.rds")
```

Last, but certainly not least, **always include the output of `sessionInfo()`** as it provides critical information about your platform, the versions of R and the packages that you are using, and other information that can be very helpful to understand your problem.

```r
sessionInfo()
```

```
## R version 3.2.3 (2015-12-10)
## Platform: x86_64-apple-darwin13.4.0 (64-bit)
## Running under: OS X 10.11.3 (El Capitan)
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## loaded via a namespace (and not attached):
##  [1] magrittr_1.5    formatR_1.2.1   tools_3.2.3     htmltools_0.2.6
##  [5] yaml_2.1.13     stringi_1.0-1   rmarkdown_0.9.2 knitr_1.11
##  [9] stringr_1.0.0   digest_0.6.8    evaluate_0.8
```

Where to ask for help?

- Your friendly colleagues: if you know someone with more experience than you, they might be able and willing to help you.
- StackOverflow: if your question hasn't been answered before and is well crafted, chances are you will get an answer in less than 5 min.

- The R-help mailing list: it is read by a lot of people (including most of the R core team), a lot of people post to it, but the tone can be pretty dry, and it is not always very welcoming to new users. If your question is valid, you are likely to get an answer very fast but don't expect that it will come with smiley faces. Also, here more than everywhere else, be sure to use correct vocabulary (otherwise you might get an answer pointing to the misuse of your words rather than answering your question). You will also have more success if your question is about a base function rather than a specific package.
- If your question is about a specific package, see if there is a mailing list for it. Usually it's included in the DESCRIPTION file of the package that can be accessed using `packageDescription("name-of-package")`. You may also want to try to email the author of the package directly.
- There are also some topic-specific mailing lists (GIS, phylogenetics, etc…), the complete list is here.

More resources

- The Posting Guide for the R mailing lists.
- How to ask for R help useful guidelines
- This blog post by Jon Skeet has quite comprehensive advice on how to ask programming questions.