Learning Objectives

- Understanding the structure of a function and how it is used
- Exposure to a few commonly used base functions in R
- A brief introduction to packages and libraries in R
- Knowing where and how to find help

Functions and their arguments

What are functions?

A key feature of R is functions. Functions are **"self contained" modules of code that accomplish a specific task**. Functions usually take in some sort of data structure (value, vector, dataframe etc.), process it, and return a result.

The general usage for a function is the name of the function followed by parentheses:

```
function_name(input)
```

The input(s) are called **arguments**, which can include:

1. the physical object (any data structure) on which the function carries out a task
2. specifications that alter the way the function operates (e.g. options)

Not all functions take arguments, for example:

```
getwd()
```

However, most functions can take several arguments. If you don't specify a required argument when calling the function, you will either receive an error or the function will fall back on using a *default*.

The **defaults** represent standard values that the author of the function specified as being "good enough in standard cases". An example would be what symbol to use in a plot. However, if you want something specific, simply change the argument yourself with a value of your choice.

Basic functions

We have already used a few examples of basic functions in the previous lessons i.e `getwd()`, `c()`, and `factor()`. These functions are available as part of R's built in capabilities, and we will explore a few more of these base functions below.

Let's revisit a function that we have used previously to combine data `c()` into vectors. The *arguments* it takes is a collection of numbers, characters or strings (separated by a comma). The `c()` function performs the task of combining the numbers or characters into a single vector. You can also use the function to add elements to an existing vector:

```
glengths <- c(glengths, 90) # adding at the end
glengths <- c(30, glengths) # adding at the beginning
```

What happens here is that we take the original vector `glengths` (containing three elements), and we are adding another item to either end. We can do this over and over again to build a vector or a dataset.

Since R is used for statistical computing, many of the base functions involve mathematical operations. If interested, we have linked a detailed guide for performing basic statistical tests in R. One example of a base R mathematical function would be `sqrt()`. The input/argument must be a number, and the the output is the square root of that number. Let's try finding the square root of 81:

```
sqrt(81)
```

Now what would happen if we **called the function** (e.g. ran the function), on a *vector of values* instead of a single value?

```
sqrt(glengths)
```

In this case the task was performed on each individual value of the vector `glengths` and the respective results were displayed.

Let's try another function, this time using one that we can change some of the *options* (arguments that change the behavior of the function), for example `round`:

```
round(3.14159)
```

We can see that we get `3`. That's because the default is to round to the nearest whole number. **What if we want a different number of significant digits?**

Seeking help on arguments for functions

The best way of finding out this information is to use the `?` followed by the name of the function. Doing this will open up the help manual in the bottom right panel of RStudio that will provide a description of the function, usage, arguments, details, and examples:

```
?round
```

Alternatively, if you are familiar with the function but just need to remind yourself of the names of the arguments, you can use:

```
args(round)
```

Even more useful is the `example()` function. This will allow you to run the examples section from the Online Help to see exactly how it works when executing the commands. Let's try that for `round()`:

```
example("round")
```

In our example, we can change the number of digits returned by **adding an argutment**. We can type `digits=2` or however many we may want:

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

> *NOTE:* If you provide the arguments in the exact same order as they are defined (in the help manual) you don't have to name them:

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

> However, it's usually not recommended practice because it's a lot of remembering to do, and if you share your code with others that includes less known functions it makes your code difficult to read. (It's however OK to not include the names of the arguments for basic functions like `mean`, `min`, etc...). Another advantage of naming arguments, is that the order doesn't matter. This is useful when a function has many arguments.

Exercise

Another commonly used base function is `mean()`. Use this function to calculate an average for the `glengths` vector.

Packages and Libraries

Packages are collections of R functions, data, and compiled code in a well-defined format, created to add specific functionality. There are 10,000+ user contributed packages and growing.

There are a set of **standard (or base) packages** which are considered part of the R source code and automatically available as part of your R installation. Base packages contain the **basic functions** that allow R to work, and enable standard statistical and graphical functions on datasets; for example, all of the functions that we have been using so far in our examples.

The directories in R where the packages are stored are called the **libraries**. The terms *package* and *library* are sometimes used synonomously and there has been discussion amongst the community to resolve this. It is

somewhat counter-intuitive to *load a package* using the `library()` function and so you can see how confusion can arise.

You can check what packages are loaded in your R session by typing into the console:

```
sessionInfo()
```

In this workshop we will mostly be using functions from the standard base packages. However, the more you work with R you will come to realize that there is a cornucopia of R packages that offer a wide variety of functionality. To use additional packages will require installation. Many packages can be installed from the CRAN or Bioconductor repositories.

Package installation from CRAN

CRAN is a repository where the latest downloads of R (and legacy versions) are found in addition to source code for thousands of different user contributed R packages.

Packages for R can be installed from the CRAN package repository using the `install.packages` function. This function will download the source code from on the CRAN mirrors and install the package (and any dependencies) locally on your computer.

An example is given below for the `ggplot2` package that will be required for some images we will create later on. Run this code to install `ggplot2`.

```
install.packages('ggplot2')
```

Package installation from source

R packages can also be installed from source. This is useful when you do not have an internet connection (and have the source files locally), since the other two methods are retrieving the source files from remote sites.

To install from source, we use the same `install.packages` function but we have additional arguments that provide *specifications* to *change from defaults*:

```
install.packages('ggplot2_1.0.1.tar.gz', type="source", repos=NULL)
```

Loading libraries

Once you have the package installed, you can load it into your R session for use. Any of the functions that are specific to that package will be available for you to use by simply calling the function as you would for any of the base functions. *Note that quotations are not required here.*

```
library(ggplot2)
```

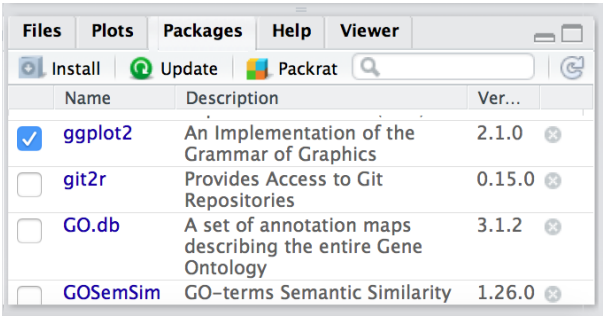
You can also check what is loaded in your current environment by using `sessionInfo()` and you you should see your package listed as:

```
other attached packages:
[1] ggplot2_2.0.0
```

In this case there are several other packages that were also loaded along with `ggplot2`.

Finding functions specific to a package

This is your first time using `ggplot2`, how do you know where to start and what functions are available to you? One way to do this, is by using the `Package` tab in RStudio. If you click on the tab, you will see listed all packages that you have installed. For those *libraries that you have loaded*, you will see a blue checkmark in the box next to it. Scroll down to `ggplot2` in your list:



If your library is successfully loaded you will see the box checked, as in the screenshot above. Now, if you click on `ggplot2` RStudio will open up the help pages and you can scroll through.

An alternative is to find the help manual online, which can be less technical and sometimes easier to follow. For example, this website is much more comprehensive for ggplot2 and is the result of a Google search. Many of the Bioconductor packages also have very helpful vignettes that include comprehensive tutorials with mock data that you can work with.

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.

Cryptic error messages

It is very likley that someone else has encountered this same problem already!

- 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").
- 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

Asking for help

The key to getting 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.

(1) 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.**

(2) **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.

```
sessionInfo()
```

(3) If possible, **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.

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: 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.
- The Bioconductor support site. This is very useful and if you tag your post, there is a high likelihood of getting an answer from the developer.
- 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.

More resources

- The Posting Guide for the R mailing lists.
- How to ask for R help useful guidelines
- 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.
