Introduction to R
LING 518, Márton Sóskuthy

This document outlines some basic features of R. It is meant as a complement to the work that we do in class, not as a replacement. As such, it is purposely concise: it is meant simply to help you remember key concepts but does not provide a comprehensive description.

Objects

Objects in R store some form of data: a single number, a piece of text, a sequence of numbers, a data table, etc. Think about them as a sort of a drawer with a label on it: there can be pretty much anything inside the drawer, and you can put any label on it, regardless of the contents. In order to create (or overwrite) and object in R, we use an operation called *assignment*, which assigns some data to the object. Here's an example:

```
a <- 1
```

This assigns the number 1 to object `a`. From this point onwards, `a` will have the value 1 until it is overwritten or until you close your R session. If you type the name of the object, R will display its contents for you. The shortcut for <– is alt/option + "-".

Object names

Almost anything goes as the name of an object, but there are some rules:

- it can only contain alphanumeric characters (letters and numbers) or the . and _ characters
- it must start either with a letter or the . character followed by a letter
- it cannot contain whitespace characters (space, tab, newline, etc.)

So the following are OK:

`a`, `A`, `abc`, `thisIsAnObject`, `this.is.an.object`, `.this.too`, `t1`, `t_1`

But the following are not:

`3`, `1t`, `.1t`, `_t`, `hey!`, `a+b`, `this is an object`

My recommendation: use brief but transparent object names, with _ characters to separate things. For instance: `rt_data` (for reaction time data), `n_of_rows` (number of rows), etc.

Object types

Objects in R have types (or classes), which determines what you can do with them. For instance, you can add two numbers:

```
1 + 2
```

But you can't add two character objects (i.e. bits of text)

```
"not" + "possible"
```

The two most common simple object types are numeric and character, as exemplified above. In general, numbers are simply written as numbers (`1`, `0.1`, `.1`), while character values must be enclosed between 's or "s (see above).

If you have an object `a`, you can check its type using the function `class(a)`.

Another common basic data type is logical, i.e. just `TRUE` or `FALSE`. There are some special operations that you can only perform on logical data, like negation (using the `!` symbol). You'll often come across logical data when you need to perform some sort of comparison, such as checking each cell in a column with respect to some condition (e.g. if they are numbers, are they all lower than 1000?).

Object conversion

You can convert between object types: `as.numeric("1")` will return `1`, while `as.character(1)` will return `"1"`. Note that

```
as.numeric("1") + as.numeric("2")
```

is OK, but

```
as.character(1) + as.character(2)
```

is not (and will throw an error message).

Vectors

R is all about numbers in the plural. A sequence of numeric or character data in R is called a vector. In fact, even a single number or character is treated as a vector in R (in this case, a vector with only a single entry). Vectors are useful because they allow you to do stuff repeatedly (e.g. to perform the same operation on multiple observations in a data set). There are many ways to create vectors, but the most explicit one is to use the `c()` function (`c` for concatenate):

```
my_vec <- c(1, 2, 3)
```

Vectors in R must have entries of the same type. Thus, the following command will actually work, but only because it converts the numbers to characters:

```
my_vec2 <- c(1, 2, "no")
```

You can verify this by checking the types of `my_vec` and `my_vec2` using the `class()` function.

Vectors are neat because they allow you to perform operations repeatedly. For instance:

```
my_vec + 10
```

will add 10 to each entry in my_vec. You can do similar stuff with character vectors too:

```
my_ch_vec <- c("Ho", "Ho", "Hey", "Ho")
tolower(my_ch_vec)
```

And you can perform operations on multiple vectors too, e.g.

```
c(1, 2, 3) + c(4, 5, 6)
```

Functions

Functions are your "verbs" in R: they allow you to do stuff to your objects (and, like verbs, they also have arguments). For example:

```
mean(my_vec)
sd(my_vec)
```

will return the mean and standard deviation of `my_vec` (we'll talk about means and standard deviations later in the course!). The arguments are between the parentheses, and there can be more than one of them (depending on the function). For example:

```
rep(1, 10)
```

repeats the number one 10 times. The arguments here are so-called positional arguments, in that their role is specified by their position: the first argument is the number to be repeated, and the second one the number of repetitions.

When there are multiple arguments, some of them are often so-called keyword arguments, i.e. their role is specified not by their position, but by a label:

```
sample(my_vec, 10, replace=TRUE)
```

where the first two positional arguments tell R to sample randomly from `my_vec` 10 times, and the third keyword argument tells it that it should sample with replacement (i.e. each entry in the vector could be drawn multiple times). In general, the same arguments in R can be specified both as positional or keyword (though keyword arguments are far easier to work with when you have more than 2-3 arguments). Thus, the following two functions have the same effect:

```
sample(my_vec, 10, TRUE)
sample(x=my_vec, size=10, replace=TRUE)
```

What you'll often see is that the first one or two arguments are given as positional, and the rest as keyword, like I did with `sample()` originally.

Libraries

Libraries extend R functionality in various ways. They are absolutely essential for working with R, and we'll use many libraries in this course. To install a library, use `install.packages()`, e.g.

```
install.packages("swirl")
```

To load a package, use

```
library(swirl)
```

(Note that you need the "s for `install.packages()`, but not for `library()`).

Data frames

Data frames are basically data tables, with each row corresponding to an observation and each column to a variable (more on this later!). In this course, we'll use a type of data frame called "tibble", which is provided by the `tidyverse` package. You'll see a lot of tibbles!

To load a tibble, you must first import the `tidyverse` (after installing it, if you haven't already done so):

```
library(tidyverse)
```

Our main way of interacting with tibbles will be via the tidyverse commands that we'll learn about very soon. But there are some simple operations that are useful to be aware of. To manually create a tibble, use the `tibble()` command:

```
my_tib <- tibble(size=c(5,6,3,2), weight=c(84,100,85,53))
```

To retrieve a column, you can use the $ operator.

```
my_tib$size
```

You can also use this to modify a column:

```
my_tib$size <- my_tib$size + 5
```

It's also possible to use "indexing" to get specific columns / rows / cells:

```
my_tib[1, ]  # first row
my_tib[ ,2]  # second column
my_tib[3,2]  # third cell in second column
```

Assuming that your input file is a csv, you can import tibbles from an external file using the `read_csv()` command from the `tidyverse`.