Basic Syntax

The most basic use of R is as a simple calculator:

```
5 + 4
```

```
## [1] 9
```

```
1 - 3
```

```
## [1] -2
```

```
4 * -2
```

```
## [1] -8
```

```
5 / 6
```

```
## [1] 0.8333333
```

A function in R follows the syntax `function_name(argument1, argument2)`. Functions perform operations on their arguments and return a result. The most basic function is the `print()` statement.

```
print('hello world!')
```

```
## [1] "hello world!"
```

R also gives us access to more complex mathematical funtions. For instance, `log()` gives us the natural log of a number, and `exp()` does the inverse.

```
log(10)
```

```
## [1] 2.302585
```

```
exp(log(10))
```

```
## [1] 10
```

But do we know all of this just from the top of our head? Of course not, there's lots of useful documentation to leaf through. Importantly, you'll never stop looking through the docs - it's a fact of life. We can look up what a function does by prefixing it with an `?` or hovering over it with our cursor in RStudio and pressing `F1`.

```
?log()
```

```
log                    package:base                    R Documentation

Logarithms and Exponentials

Description:

     'log' computes logarithms, by default natural logarithms, 'log10'
     computes common (i.e., base 10) logarithms, and 'log2' computes
     binary (i.e., base 2) logarithms.  The general form 'log(x, base)'
     computes logarithms with base 'base'.

     'log1p(x)' computes log(1+x) accurately also for |x| << 1.

     'exp' computes the exponential function.
     'expm1(x)' computes exp(x) - 1 accurately also for |x| << 1.

Usage:

     log(x, base = exp(1))
     logb(x, base = exp(1))
     log10(x)
     log2(x)

# further output omitted for brevity
```

Importantly, the bottom of the help pages always contain executable examples. You can *always* copy and paste them into your R console and they will work. (A package won't build properly if the examples don't work!) So now that we know where to go for help, we know all we need to know about R, right? ;)

Variables and assignment

We'll probably want to save our answers at some point. We do this by assigning a variable. A variable is a name for a saved piece of data - let's show some examples:

Assign the value 5 to the variable "a":

```
a <- 5
a
```

```
## [1] 5
```

Note that `a` and it's value can be used interchangeably:

```
a * 7
```

```
## [1] 35
```

You may have noticed that we used the `<-` sign to assign a variable earlier. This is different than most languages, which use the = sign. We can actually still use the = for assignment in R if we want to.

```
b = 10
b
```

```
## [1] 10
```

Note that using = is frowned upon in R. You'll hear that phrase a lot in this lesson… "such and such is frowned upon", "so and so is better". Over the course of this lesson, we're going to actually try to explain why we do things each way. In this particular case, both methods are the same in all but one rarely-used edge case (inline variable assignment). As such, you are free to use whichever operator you choose, but by convention, R programmers will typically use `<-` (RStudio hotkey: `Alt + -`). If you're interested in doing things according to the "official style", you can open up Tools -> Global Options -> Code -> Diagnostics and tell it to check code style.

One important thing to note is when variables are modified. Let's demonstrate this via example.

```
weight_kg <- 55
weight_lb <- weight_kg * 2.2
weight_lb
```

```
## [1] 121
```

Ok, everyone should be with me so far. But what about if we modify the value of `weight_kg`, does `weight_lb` change as well?

```
weight_kg <- 9000
weight_lb
```

```
## [1] 121
```

It does not. Variables only update when we explicitly assign them with `<-`.

```
weight_lb <- weight_kg * 2.2
weight_lb
```

```
## [1] 19800
```

We can contrast with how variable assignment works in Python and other languages: where there is a distinction between objects and primitives and assigning a new copy of an object just creates a link to the original set of data. R does not treat any variables differently from others. In R, whenever you assign a variable under a new name, it creates a copy of the orignal.

Copy-on-modify

Or, that's almost how it works. R uses a "copy-on-modify" behavior. Essentially whenever you assign a variable under a new name, it points at the old variable, and does not take up any extra space. However, as soon as you modify *anything* in the new variable, R will create a new copy.

Let's demonstrate this by example. If you want to follow along for this part, you will need to install the `pryr` package (`install.packages("pryr")`). Also, this will introduce a new function `library()` - this is used to load extra bits of functionality that is not included in the base R programming language.

```
var1 <- 10
var2 <- var1
```

Alright, we've assigned two variables, `var1` and `var2`. `var2` is a copy of `var1`. Is there a way to check if I'm telling the truth about the copy-on-modify behavior? Are these the same object?

Fortunately, the `pryr` package lets us take a closer look at R's internal workings. The `address` function can be used to see the memory address of an R object.

```
library(pryr)
address(var1)
```

```
## [1] "0x55d47f806318"
```

```
address(var2)
```

```
## [1] "0x55d47f806318"
```

Without going into too much detail on how memory addresses and allocation works (it's not important for writing R code), we can see that both `var1` and `var2` have the same address: they are stored in the same spot in your computer.

What happens if we modify `var2`?

```
var2 <- var1 + 1
address(var1)
```

```
## [1] "0x55d47f806318"
```

```
address(var2)
```

```
## [1] "0x55d48074f028"
```

The address has changed. After modifying `var2`, R realized that it could no longer store both variables in the same spot and made a new copy to store `var2` in.

This copy-on-modify behavior has important implications for how we write R code. Don't reassign variable names just because you want a new name for it. Every time you reassign a variable and modify it (even slightly), you force R to make a new copy, doubling it's memory use.

A smarter way of doing things (especially if you just need to modify a variable), is to do things “in-place”, or simply overwrite the original variable with its modified value.

Good example

```
var1 <- 10
var1 <- var1 + 1
```

Bad example

```
var1 <- 10
var1_modified <- var1 + 1
```

Next Section