Introduction to R

Sean Murphy

4 November 2016

Installing R and Rstudio

First thing's first. We need to install R before we can get started. We'll also be installing Rstudio. While R is the basic programming language we'll be using, the interface for R itself is pretty basic (and, in my opinion, ugly). Rstudio is a third-party interface for R that gives it a nicer look, separates important parts of R into their own windows, and adds dropdown menus for some useful coding functions. After you've installed the two, you'll always be opening Rstudio, which will run R for you.

To install R, you'll need to go to the official R repository here and click the appropriate link for your operating system. Windows users will want to click the 'base' subdirectory after selecting Windows, then click the Download R for Windows link. Mac users will want to download the most recent .pkg file.

To install Rstudio, you'll want to go to their website here and download the most recent version for your operating system.

The basics

Today we'll work through the basic principles of R using examples that you'll run through the RStudio interface. We'll cover basic operations, objects in R, different data types, and vector subsetting.

First thing's first - in your RStudio window, you should have a window called 'Console', with a little `>` sign. That's where you enter commands into R. If you type a command at that prompter and press enter, R will try to execute it.

One thing to note when reading this handout is that only the content in grey boxes is R code that you can copy in to your console and run. Sometimes (like just below) the handout will also show the results you get from running that code. The results will appear below the grey box, and are there to be informative, but you won't be able to paste them into the console and have them work, since they're output, not code.

Now, moving on. The most basic thing you can use R for is a simple calculator. It can perform a large number of mathematical operations, but let's start simple. First, let's take a look at what happens when we use incredibly powerful computing software to add two plus two. Type `2 + 2` into the console (in front of the `>`) and hit enter:

```
2 + 2
```

```
[1] 4
```

Let's deconstruct the output (simply though it may be). First, we can see the answer, four. But before that, we see a [1]. R prints the 1 to show us an index of where the number four falls in our output. Basically it's telling us that it's the *first* thing we asked for. Now, given that we only asked for one thing, that's pretty obvious. But later, when we're asking for 20, 50, or 100 numbers, R will include these indices at the start of each row of output to help us count where we're up to. To see this in action, try running the following command, which tells R to print the numbers from 1 to 100, and notice that the indices match where they should.

```
1:100
```

Alright, back to basic calculations. Symbols R will recognize include, among others, `*` for multiplication, `-` for subtraction, `+` for addition, `/` for division, and `^` for "to the power of."

```
2 * 2 + 4

4 ^ 2
```

Remember the order of operations from high school math? Operations in brackets are evaluated first, then multiplication and division, then addition and subtraction. It works the same in R. If there's a tie in priority (only really relevant for division and subtraction) R will work from left to right. Note the difference in the following two commands:

```
2 / 2*4
```

```
[1] 4
```

```
2 / (2*4)
```

```
[1] 0.25
```

Also note that for most contexts in R where you can insert a space, you can use as many or as few spaces as you want - R doesn't care. So you can crunch your operations up with no spaces or add a lot of spaces between them, and as long as those spaces dont change the meaning of what you're doing, your code will still run.

The first three examples below will run, but not the fourth, because when you add a space between the `3` and the `8` in `38` it changes the meaning of the syntax - R now reads it as two separate numbers - 3 and 8, but without any operator between them to tell it what to do. It gives the error `unexpected constant in "2 + 3 8"` because there was an 8 at the end when it was expecting to be told what to do with the 3 - e.g. whether to add, subtract, or multiply it to the next thing in line.

```
2 * 38

2*38

2   *   38

2   +   3 8
```

Exercise:

1. Use R to find 2 to the power of 6
2. Try dividing something by zero. What output do you get?

Now, one thing that often trips people up early is what's called the continuation prompt. Try typing in the command below and hitting enter, and you'll find that R doesn't spit out any output. Instead, you should see our friendly command prompt `>` replaced with a `+`.

```
2 *
```

R displays the continuation prompt when you have entered something that is not a complete command (it can't possibly be run on its own). You'll find that if you enter any number and hit enter, it will complete the partial command you started (and R will multiply the second number by two).

The most frequent cause for seeing the continuation prompt is that you were trying to write out a long command and forgot something at the end (more on that later). Sometimes you can't figure out what R is waiting for, and if you keep on typing things that don't finish the command you started, it will just keep showing you that `+` symbol. If you find yourself trapped like this, just hit the ESC button, and R will forget you every made that embarrassing mistake and take you back to a fresh command prompt `>`.

Just keep in mind that while R tries to be helpful, it won't try to read your intentions (that would be dangerous!). It will only take you to the continuation prompt if there's no way to run the command you were trying to run. When you hit enter, R will run if what you've typed is a complete, valid command, even if you weren't finished what you were trying to write out. So only hit enter when you're ready to run something.

Functions

In order to do calculations more complicated than basic math, we can use functions. Generally speaking, functions take some input (which goes in brackets just after the function name, as you can see below), perform their specific function (thus the name!) and output the result. Functions are what you'll be using to do most of your data management and analysis, and can be extremely powerful – in R, functions do everything from T tests up to and beyond fitting multilevel models. Try out a few basic functions – `sqrt()` gives the square root of its input, and `log()` gives (you guessed it!) the log of its input.

```
sqrt(4)

log(20)

log (4 + 6)

sqrt(log(10))
```

The final example above, humble though it is, provides an example of one of the most powerful properties of R for data analysis. We can see that the square root function happily accepts `log(10)` as input, even though log(10) is itself a function. This line of code is telling R "take the square root of the log of 10" – which is exactly what it does. This is possible because of how R treats functions. Functions in R are treated as objects, similar to numbers, that are evaluated (solved, or turned into their output) when neccesary. So, when it needs to, R will treat a function such as `log(10)` as if it is the result of that function (in this case, 2.3025851).

This has far-reaching consequences when it comes to things like data analysis. For instance, as we'll see when we start running regressions, the regression function will accept `scale(variableA)` as a predictor (or IV) just as happily as it would take `variableA`. Since the `scale()` function mean-centers variables, we can use it in a regression when we want to test an interaction (which requires both predictors to be centered). Gone are the days of SPSS files with dozens of separate, centered versions of variables - you can center them right at the point of analysis!

Now, you may not share my excitement for centering variables, but the take-home for now is that that we can nest functions within functions and R will handle them as we'd intuitively expect. We'll see more of the impact of this when we dig in to data analysis and data management.

Assigning variables

Since most of the time we're interested in making things a bit more permanent, we can assign our numbers to variables. To be clear, these are the programming type of variables, conceptually distinct from psychological variables (of the independent and dependent kind). Variables are containers that we can name (almost) anything we like. They're used to 'park' numbers, and will remember their value. In fact, for all intents and purposes, once we've assigned a number to a variable, R treats the variable as if it *is* the number. Thus, we can perform calculations using variables as if we were using the numbers we'd assigned to those variables. To assign something to a variable, we use the `<-` command, like so:

```
a <- 4
b <- 2

a + b
a * b
```

The `<-` operator can be a little counterintuitive at first, since we read left to right but it's placing a number from the right into a variable on the left. Think of it as reading something like 'is now'. So `a <- 4` would read 'a is now 4.' Hopefully that makes sense.

You might have noticed when you ran the code above that an `a` and a `b` appeared in the "Environment" window of Rstudio, up in the top right of the screen. The environment in R is kind of like your current workspace - it contains every object that you've created so far this session (the environment gets cleared when you restart R, unless you save it). Because variables remain in R until you overwrite them or remove them, R will remember the values of `a` and `b`, and will print them if you type their names into the console and hit enter. Try it now:

```
a
```

```
[1] 4
```

If you no longer wanted `a` around, you could get rid of it using the `rm()` function. Type `rm(a)` and watch the variable vanish into the aether, never to be seen again. To check that `a` really no longer exists, you can try typing its name in to the console again, and watch R get lost trying to find it. If you want to clear all the variables you've created so far, you can hit the little broom button at the top of the Environment tap in Rstudio. There's no undo button in R, so if you remove something, it's gone (until you create it again).

```
a
```

One very important thing to remember in R is that variables (and pretty much everything else) are case sensitive. Try typing in a capital `A`, to see what happens. This can take a little while to get used to, and you'll certainly get your cases wrong occasionally (I still do) but just try to keep it in mind.

If R gives you an error and you can't figure out what you did wrong, there's a halfway decent chance you capitalized something it didn't expect you to (or vice versa!). Because programming can be very precise, R won't guess that you meant `a` when you typed `A`, even if there's no `A` around, because that could lead to a lot of unexpected errors when the programs you're writing get more complex than this.

Variables can also become input for functions - R treats them just like the numbers they represent.

```
sqrt(a)
```

```
[1] 2
```

We can also assign the result of a calculation to a variable. R will perform the calculation (e.g. `1 + 1`), and assign the answer `2` to the variable we name:

```
c <- 2 * 5
c
```

```
[1] 10
```

And just as a reminder, you can name variables whatever you want – don't think variables can only be single letters! That wouldn't be very handy later on, when we're trying to use them to store real data (the psychological kind of variables). Actually, variable naming does follow a few sensible conventions, some of which might be familiar from SPSS. For instance, a variable can't have a space in it, and it can't have specific characters like `?`. But you can use any combinations of upper and lower case letters, numbers, `_`, and `.`, though you should always start with a letter. When you're working with real data, you want your variable names to be compact (you might be typing them a lot in your analyses) but informative. The following are a few possible styles for naming variables. I generally stay in all lower case (which reduces cases of forgetting a capital) with underscores, but you can pick your preference.

```
really_cool_number <- 10
really.cool.number <- 10
Really.Cool.Number <- 10
```

Vectors

Just as we're not usually interested in a single point of data, we don't usually work too much with single numbers in R. Instead, when cleaning and analysing data we're usually dealing with vectors - long strings of data, like the scores of all of your participants on a single variable. Technically speaking, all numbers in R are vectors - some are just vectors with a length of 1 (so they only contain one number, like 4). But from our point of view, vectors containing more than 1 number have a lot more to offer.

Now, when you start loading in your own data, R will create a vector to store each variable, and tie them all together into a dataset (more on that later). But R also has a lot of ways to create vectors from scratch. We already saw one way near the start of the handout, where we used 1:100 to create a vector with the numbers from one to one hundred. We can also use the `c()` function, which stands for combine. `c()` will take all of the numbers that you put into it, separated by commas, and put them together into a vector. Then there's `rep()`, which repeats the first number you enter an amount of times equal to the second number you enter.

```
a <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
b <- rep(1, 10)
```

You will have noticed that the two new functions we introduced above, `c()` and `rep()` were different from the previous functions we looked at – they took more than one number as input. This is because these functions accept more than one 'argument'. In fact, in the case of `rep()`, it requires at least two arguments – one to tell it which number to repeat, and one to tell it how many times to repeat the first number.

The term argument sounds aggressive, but in programming it just stands for an input to a function that tells it how to procede. Technically, all functions take arguments - when we type `sqrt(4)`, `4` is an argument that says "this is the number you should square root." In the case of `rep()`, the first argument, (`1`), says "this is the data to replicate"", and the second argument, (`10`), says"this is how many times to replicate the first argument".

One important thing about arguments is that each of them has a name. In the case of `rep()`, the first argument is called "x", which is generally used when some form of data is expected as primary input (the only argument of `sqrt()` is also called "x"). The second argument is called "times", which is the number of times to repeat x. You don't always need to call arguments by name, which is why the example we just ran worked. The arguments are arranged in a logical order, and if you don't name them, R assumes that you're inputting arguments in the default order it expects, and match your input up accordingly. However, by using names, you can call arguments in any order you want. See the below examples, all of which give the same output:

```
rep(2, 5)
rep(x = 2, times = 5)
rep(times = 5, x = 2)
rep(times = 5, 2)
rep(5, x = 2)
```

Named arguments are matched first, and then unnamed arguments are matched to available arguments in order. This is why the second last example works even though we don't specify that `2` is meant to match the argument "x". R matches the input 5 to the argument "times" and then when it encounters the input 2, it starts looking through valid arguments from the beginning. For the same reason, the last example works, even though 5 comes first, where it would usually be matched to "x" (and thus be the number that is repeated). Because the argument "x" is matched to 2 by name, that argument is already taken, and so 5 is slotted into the next available spot – "times".

There are a few more things to know about arguments. One is that many arguments have default values. If you don't input a value for the argument into R, it will fill in the default. For `rep()`, there is no default value for "x", so the function will give us an error if we don't give it any input at all. But "times" does have a default value – 1. If we just put in "x" and leave out anything else, R will assume we want to repeat "x" one time.

```
rep(2)
```

```
[1] 2
```

In fact, this is the main reason to have named arguments in the first place. Many functions, especially advanced statistical functions that fit complex models, have a lot of arguments with default settings, like which assumptions to make or what results to test. If we don't specify anything, the function will run with its default settings, but if we want to, we can name an argument and specify the setting we want to use, tweaking the specifics of the model being fitted. Named arguments allow us to control the value of some setting far down the list without having to input a value for every argument before that.

You may be thinking that remembering the arguments for each function would be an awful lot to ask of a person. Thankfully, each function in R has a help file that describes what that function does, and what its arguments are. `help()` and `?` both take you to the help file for a given function. Try them out, and take a quick look at the help file for `rep`. The details at the top of the help file can be a little dense if you're not familiar with the internal workings of R, but a little further down you'll see a section labelled 'Arguments'. That section will list all of the different arguments that can be used to modify the way that `rep()` functions.

```
help(rep)
?rep
```

If you scroll to the bottom of the help page you'll see an 'Examples' section, which contains code examples you can try running yourself to get a feel for the different options available. If you run the first example, you'll see that `rep()` doesn't just work on single numbers. It can be used to repeat a series of numbers as well. Another thing you should take note of in the `rep()` help file is the "each" argument. Using "each" instead of "times" changes the way the rep works.

```
rep(1:4, times = 2)
rep(1:4, each = 2)
```

Now, before we continue working with `rep()`, let's double back quickly to `c()`. As we saw before, `c()` can be used to tie together a bunch of numbers into a single vector. In fact, when we used `c()` before, we put in all the numbers one through ten, which seems like a lot of arguments to match, doesn't it? In fact, the `c()` functon is a bit of a special case. The only argument it recognizes is the rather enigmatic "...", which is sort of like R's way of saying "put in as much or as little as you like." Basically, the `c()` function will take any number of arguments you enter, and match them all to its catch-all "..." category. The function then binds all of these arguments into a single vector, and that vector is its output. It's not really important to understand "..." in any real detail. Just remember that most functions have a specific set of arguments they'll take, while `c()` is deliberately flexible, since its job is to create a vector out of whatever you throw at it. More on that later.

Exercise:

Now you've done a lot of reading, so it's time to stretch your programming muscles a little. Try the following exercises:

1. Modify the first example in the `rep()` help page to create the vector `(1, 2, 3, 1, 2, 3, 1, 2, 3)`
2. Change the second argument in your solution above to `each` - you should get the vector `(1, 1, 1, 2, 2, 2, 3, 3, 3)`

3. Remembering default arguments, what do you think "times" is doing when you specify a value for "each" instead? What happens when you use both arguments at once?
4. Instead of using the `:`, try using a series of numbers constructed with `c()` as input to `rep()`. Check that it still works if you save the numbers as a variable first and run `rep()` on the variable.

Functions on Vectors

We can use functions to perform operations on vectors, just like we can create them. Some functions summarise vectors, giving us a single output that describes them in some way. For instance, try the following - you can probably guess what the `mean` and `median` functions do:

```
a <- 1:10
mean(a)
median(a)
```

Other functions work on each element in a vector (i.e. each number stored in it). R is set up to perform what are called vector-wise operations. This means that if you use a function like `sqrt` on a vector, that function will be performed on every element in the vector at the same time. This isn't always the case with programming languages, but it should feel fairly intuitive as psychologists - after all, if we want to log-transform a variable, we certainly expect that the result will be a new variable that contains the log of each data point in that variable. Try it now:

```
log(a)
sqrt(a)
```

Similarly, if you try to add or multiply two vectors, R will perform the chosen operation on each element of the vector at the same time. This is easiest to demonstrate in practice:

```
a <- 1:10
b <- 1:10

a + b
a * b
```

Exercise:

1. Using `c()` and `mean()`, find the mean of the vector containing 2, 6, 10, and 24.
2. Create a variable (choose your own name) containing the numbers 40, 31, and 72. Then create a new variable containing the median of the first variable.
3. Can you guess what the following line of code will give as output? What if we used the `sum()` function, which calculates the total sum of its arguments?

```
mean(rep(2:4, 2))
```

Random numbers

Before we move on, let's start to play with data a little. To do this, we'll use a new function. `rnorm()` generates random data from the normal distribution. It takes three arguments:

1. `n`: the number of observations to generate
2. `mean`: the mean of the normal distribution to draw from. The default is set to 0.
3. `sd`: the standard deviation of the normal distribution to use. The default is set to 1.

Try the following commands to get a feel for the kind of data rnorm generates. Then try playing with the data it generates a little. Use the `mean()` and `sd()` functions to check how close the data you generated is to the arguments you specified. How does the accuracy of the `mean()` command change as you increase the standard deviation of the distribution, or increase or decrease the number of observations?

```
rnorm(n = 10)
rnorm(n = 10, mean = 10)
rnorm(10, mean = 10)
rnorm(100, mean = 0, sd = 5)
```

You can also try using `cor.test()`, which is an R function to test the correlation between two variables, on two vectors of the same length generated with `rnorm()`. There should be no correlation between two randomly generated variables, but how many times do you have to run the code below before you get a p value below .05 – a type I error? Does it seem like the type I error rate changes with N?

```
cor.test(rnorm(100), rnorm(100))
```

These are simple questions, but functions like `rnorm()` are great for getting a tangible feel for the answers, even the ones you already know. As we'll see later, being able to simulate data under different conditions will end up being very powerful later to check assumptions, run power analyses, and more.

Subsetting

One of the most powerful tools in R is subsetting. Subsetting (like it sounds) lets us select a subset of our data. Later on, it will allow us to do things like select only participants in our data who are female, or only trials where reaction time is less than 3 standard deviations from the mean.

For now, we'll practice simple subsetting on vectors. To subset, we use the square brackets, '[' and ']', to indicate which part of our vector we'd like to look at. We can subset using a single number to select, say, the fourth element of a vector, by using the subset code `[4]`. Or we can subset using a series of numbers put together by `:` or `c()`. We can even subset a vector with a variable, provided the variable contains values that point to valid locations in the vector. Test it out to see how it works in practice.

```
a <- 1:10

a[4]

a[1:2]

b <- 5
a[b]
```

You can also use negative subsetting. Negative subsetting asks for every value in a vector *except* the ones you specify. For instance, a subset of [-4] would return all but the fourth element of a vector. You can string together negative subsets the same as positive ones. One major limitation is that you can't mix positive subsets and negative subsets – mostly because it doesn't make a sense to ask for the fourth element while also asking for "everything except the sixth element" (`[c(4, -6)]`). So the last example below will give an error.

```
a[-4]
a[c(-1, -5)]
a[c(4, -6)]
```

Exercise:

Why does the first example below give an error, while the second one doesn't? Hint: Try running just `-1:4` see what that gives you. Do you understand why they give different results?

```
a[-1:4]

a[-(1:4)]
```

Character vectors

So far, we've just been using numbers, but of course we often have data that isn't numeric. We might have the names of conditions, participant gender coded with the letters 'M' and 'F', or even written responses from participants in our datasets. Accordingly, vectors in R can contain all kinds of data. Essentially, every data type from SPSS has a corresponding vector type in R. For now, we'll look at the 'character' type. Character variables contain strings of text, and correspond roughly to the string variable in SPSS.

We can construct character vectors in much the same way as we did numeric vectors, as you can see below.

```
words <- c('apple', 'pear', 'orange', 'nachos')
numbers <- c(1, 2, 3, 4)

class(words)
class(numbers)
```

The `class()` function tells us which type of data is in a vector. In the above example, it reminds us that the vector with string data is a "character" type, and tells us that the vectors we've been working with up until now are "numeric" type.

One important thing to keep in mind when building character vectors is that you need quotation marks to tell R that it should treat the word you're giving it as a string of letters, rather than as a variable. If you forget the quotation marks, R will go looking for a variable named `apple` and will panic when it doesn't find it. See below.

```
words <- c(apple, pear, orange, nachos)
```

```
Error in eval(expr, envir, enclos): object 'apple' not found
```

In fact, R takes the quotation marks very literally, and will represent anything in quotation marks into a character variable, as you can see if you run this example.

```
numbers <- c('1', '2', '3', '4')
class(numbers)
```

Logical vectors

Now that we've looked at numerical and character data, the third important type of data in R is logical. Logical data is special in that it can only take on two values: `TRUE` or `FALSE`. Remember that capitalization is important, especially here. Only `TRUE` has special meaning, for instance, not `True`, or `true`.

Logical data is also different to the previous types of data we've looked at in that we usually don't construct logical data using functions like `c()`, though that's possible. Instead, logical data is usually the answer to a question We ask R to evaluate a test, like whether a participant's age is less than 50, and it gives us logical data in response. "TRUE" means that the our test was passed – so the participants age was indeed less than 50. "FALSE", of course, means the opposite.

There are a handful of operators we use to test logical propositions. The primary one is `==`. This operator is used to test whether something is exactly equal to something else - like whether 5 * 5 is 25. Take careful note of the double equals sign. A single equals sign means something else in R, so we use the double equal sign to test logical propositions. If you ever get an error that mentions 'left-hand side to assignment' you probably used a single `=` where you should have used a `==`. Try these examples:

```
10 == 10
10 == 5 + 5
1 == 2
```

Apart from testing whether two things are exactly equal, we can test the usual mathematical questions - is one number greater than (`>`), less than (`<`), greater than or equal to (`>=`), and less than or equal to (`<=`) another number. We can also test the opposite of the `==` operator with `!=`, which tests whether two things are NOT equal.

```
25 < 50
20 != 20
20 != 21
19 >= 19
```

You can also store the results of a logical test in a variable – this is the main way we create logical variables, and, soon, logical vectors.

```
result <- 25 < 50
result
```

```
[1] TRUE
```

Now, on to logical vectors. Remember in the 'Functions on Vectors' section, when I mentioned that R is set up to do vector-wise operations, applying a function to every element of a vector at the same time? The same thing is true for logical operations. If I have a vector full of numbers, and I ask R whether that vector is less than 50, it won't total up the numbers in the vector and tell me whether that is less than 50. Instead, it will give me a logical vector of `TRUE` and `FALSE` values, the result of testing whether each individual number in the vector is less than 50. This will be super handy later - for instance, that operation could be used to create a categorical variable called `age_over_50` if you needed it. Try the examples below, and play with a few of your own.

```
a <- 1:10
a > 5
a == 2
a != 9
```

Logical operations don't just work on numbers, we can also operate on text. This is mainly useful with the `==` operator, of course, which tells us whether two strings of text are the same. When applied to a vector, this can act kind of like a search. For instance, in the vector "pets", containing the choice of pet of each of seven hypothetical participants, which elements contain the word "cat"?

Just be careful using the `>` and `<` symbols on text. R will allow it, but it interprets these as asking whether one word comes after another word alphabetically, or vice versa (try it on the pets vector). There are a few cases of this in R where we might expect it to give an error message, but it will instead give its best-guess of what you wanted. These are the most dangerous cases from a data analysis perspective because you might not notice that an error you made had resulted in a completely nonsensical command (that ran anyway). More on that later.

```
pets <- c('cat', 'dog', 'fish', 'cat', 'cat', 'dog', 'turtle')
pets == 'cat'
```

There are a few more operators that extend the powers of logical tests in R. Theses will by familiar to anyone who has ever used Qualtrics, or another 'flow logic' system. These operators are AND (`&`) and OR (`|`). As

they sound, they allow us to test more complex propositions, like 'which people own either cats or turtles?', or 'which people own both cats and dogs?' Though of course, since everyone in our pets vector only had one pet, noone owns both a cat and a dog. How sad for them.

By the way, the | should be above your enter key on your keyboard.

```
pets == 'cat' | pets == 'turtle'
pets == 'cat' & pets == 'dog'
```

Now, one thing that can trip you up is that the & and | operators need to test two separate and complete logical propositions. So, to write the first proposition above formally, it's testing "is the pet a cat, or is the pet a dog?". Imagine two people, Bob and Sally. Bob gets asked 'is the pet a cat', while Sally gets asked 'is the pet a dog'. They go off and check their own answer, come back together, and if either of them got the answer TRUE, then they put forward a collective TRUE answer. If they were using the AND operator instead of OR, then they'd answer FALSE if either of them got a FALSE answer. This metaphor (sort of) represents how machines test logical propositions at the lowest level, using separate logic gates for each question.

If you try to convert the human English question 'is the pet a cat or a dog', directly into code, you get an error, as you can see below. This is because, to continue our metaphor, Bob got sent off with the question "is the pet a cat", while Sally got sent off with the question "dog", which isn't a logical test in any language. Hopefully that visual helps keep the structure of AND and OR operations clear in your head.

```
pets == 'cat' | 'dog'
```

Now, we've seen one way that logical vectors could be useful, in that they could be used to create categorical variables. But another key use of logical data is for subsetting. Before, we used numbers to subset, specifying the index of the elements we wanted. Logical subsetting is similar, but it's like an on-off switch. Any element that is subset with a `TRUE` value is retrieved, while any element subset with a `FALSE` value is not. To demonstrate, let's create another vector to go with our "pets" vector, this time containing the names of the pet owners.

```
owners <- c('Jason', 'Rob', 'Sandy', 'Elliot', 'Lilly', 'Sophia', 'Chloe')
```

Now, we've already seen that if we test whether the pets vector contains a "cat", we get a logical vector of `TRUE` and `FALSE` values as a result. But that's not super informative. Using logical subsetting, we can find out the names of the cat owners. Play around with this to get confortable with the idea, because we're going to end up using it quite a bit. Try retrieving the names of pet owners with either a fish or a dog.

```
pets
```

```
[1] "cat"    "dog"    "fish"   "cat"    "cat"    "dog"    "turtle"
```

```
owners
```

```
[1] "Jason"  "Rob"    "Sandy"  "Elliot" "Lilly"  "Sophia" "Chloe"
```

```
pets == 'cat'
```

```
[1]  TRUE FALSE FALSE  TRUE  TRUE FALSE FALSE
```

```
owners[pets == 'cat']
```

```
[1] "Jason"  "Elliot" "Lilly"
```

```
owners[c(1, 4, 5)]
```

```
[1] "Jason"  "Elliot" "Lilly"
```

Now, a few final words about logical values before we move on. First, if you try to use logical values in a computation (like if you tried to multiply a numeric vector by a logical vector) it treats `TRUE` as `1` and `FALSE` as 0. This is useful in a lot of ways. For instance, if we wanted to count the number of cat owners in our previous example, we could use `sum()` as shown in the code below. It comes in very handy when you want to count how many participants in your dataset are english-speaking and male, or how many trials in your experiment take longer than 400 milliseconds.

```
sum(pets == 'cat')
```

Finally, the `!` value acts as a logical reverser. You can read it as saying "is not." We saw it in this role in the `!=` operator, but it can act on its own. The brackets below aren't strictly neccesary, but I find it helps to make things clear (and when you start writing more complicated code, unneccesary brackets can help you read what's going on).

```
pets == 'cat'
!(pets == 'cat')

# Here we find the pet owners who don't have cats.
owners[!(pets == 'cat')]
```

Exercise:

Try playing around with the `TRUE` and `FALSE` operators directly to get a better feel for them. For instance, try to intuitively guess whether each example below will give a `TRUE` or a `FALSE` result, then check whether you were right.

```
!TRUE
TRUE | FALSE
TRUE & FALSE
(TRUE & FALSE) | TRUE
FALSE & (TRUE == FALSE)
(FALSE & TRUE) == FALSE
FALSE & (FALSE | TRUE)
(TRUE == FALSE) == (TRUE & FALSE)
```

Vector coercion

So, we've heard that a vector can only contain one type of data, and we've now seen the three key classes of data that R can represent: numerical, character, and logical. So what happens when we try to put them together – do we get an error?

```
a <- c('alpha', 'beta', 2, 4, 'gamma')
class(a)
a
```

In fact, no we don't. See, as I mentioned before, R does its best to be helpful. So when you try to create a vector with more than one class of data, it forces (or "coerces", thus the name) all of the elements of that vector to be the class with the lowest common denominator. Basically, the number 2 can either be a number (numeric), or it can be a string "2" (character). Since there's no way to convert "alpha" to be a number, R instead converts the numeric data in the vector to be character data. Thus, harmony and balance within the vector is maintained. In this fashion, if you combine logical and numeric data, you get a numeric vector, with all the `TRUE`s turned into 1s and all the `FALSE`s turned into 0s.

We'll talk more about vector coercion when we start reading in real data. It becomes espescially relevant when data entry errors put different data types in the same variable (for instance, when someone enters their age as 'twenty-six' instead of '26').

Data frames

Now, there's a bit more to cover with vectors, but we've gotten far enough that we know the kind of variables that they can represent and most of how they work. And since we're all interested in real-world data, we're going to move on to the R object that holds what we're looking for: the dataset.

When we're analyzing data, we have a lot of variables – participant ID numbers, demographic variables, and vectors of our IVs and DVs - all tied together. R handles this with objects called data.frames, which are essentially identical to what you're probably used to working with in SPSS - they have rows and columns, and the columns can represent all sorts of different variables, which usually have variable names. Whereas vectors in R can only contain one type of data each (numeric, character, logical, etc), data.frames can tie together vectors of all different types, each representing a variable of interest.

Most of the time, we'll create data.frames in R by reading in our data from outside sources (like excel spreadsheets or SPSS data files) using functions for that purpose. We'll get to those functions later. We can also build data.frames from scratch in R similar to what we did with vectors (this is often how we'll run simulations).

In the meantime, however, R contains a number of example datasets in data.frame format out of the box, which can be accessed just by typing their names. Typing `data()` runs a special function that will open a separate window describing these example datasets. For now, we'll get to know the data.frame structure using the `trees` dataset, which lists the heights, girths, and volume of 31 trees (very relevant to us as psychologists, I know. If it helps, you can think of the trees as participants). Try taking a look at the trees object - R will print datasets just like vectors if you type their name.

```
trees
```

Since most datasets can be a little big to print all at once, R has various functions to examine parts of the dataset to get an idea of what it looks like. For instance, `head()` and `tail()` will print the first or last 6 rows of a dataset, and `str()` and `summary()` will give you information on the variables in that dataset. Try them now.

```
head(trees)
tail(trees)
str(trees)
summary(trees)
```

There are also R functions which are designed specifically to operate on datasets. For instance, `colMeans` will show you the mean for each variable in the dataset. We'll explore more functions that work on datasets like this a little later.

```
colMeans(trees)
```

Subsetting datasets

Subsetting with datasets is one step more complicated than when using simple vectors, because datasets have two dimensions, not one. When subsetting datasets, instead of having a single index `[4]`, which refers to the fourth item in the vector, we need to identify both the row(s) and the column(s) we're interested in. To do this, we add a comma to the subsetting syntax, and the index before the comma identifies the row(s) we're interested in, while the index after the comma identifies the column(s). For instance, to look at the fourth row and first column of the trees dataset, we would use the following code:

```
trees[4, 1]
```

```
[1] 10.5
```

Take another look with `head(trees)` and you'll see that we got the right answer. But what if we wanted the entire first column of the trees dataset - if we want to look at the 'Girth' variable alone? With dataset subsetting, if you leave one of the indexes blank, that is taken to mean 'everything'. So, for example, the first line of code below would give us every row with only the first column, and the second line of code would give us the fourth row with every column.

```
trees[, 1]

trees[4, ]
```

Of course, just like we can subset vectors using multiple numbers, we can do the same with datasets.

```
trees[1:10, ]

trees[1:4, c(1, 2)]
```

Now, with larger datasets, it's probably easier to identify variables by their names, rather than by their position in the dataset. With that in mind, datasets have two ways to select specific variables using their names. For instance, if we wanted to look at the 'Height' variable in the trees dataset, we could use either of the following commands – try them now:

```
trees[ , 'Height']

trees$Height
```

Now, using the dollar sign method (`trees$Height`) is faster if we just want to look at one variable. But the benefit of the first method is that we can extract more than one variable at a time if we subset using a vector of variable names, like so:

```
trees[ , c('Girth', 'Height')]
```

Alright, we know how to subset variables by name. Let's try logical subsetting. First, take a look at what happens when we ask R to test whether each tree is taller than 75 (we'll assume that's in feet).

```
trees$Height > 75
```

As you might expect after learning about logical operations, we get a vector that tells us whether the tree in each row is taller than 75 feet. Now, as we saw with vectors, we can use this to subset, or select, the part of the dataset that only contains trees taller than 75 feet.

Now, this is the slightly confusing part of dataset subsetting. We're testing our logical value using the Height variable, which is a column, so your intuition might be that this logical variable needs to go after the comma – where we select which columns we want.

But remember, we're trying to subset rows – remember, we're looking to select a subset of only specific trees, and each row in the dataset is a tree. So even though our logical test is on a column, we need to place it before the comma. Essentially, this reads "give us every row for which the Height variable is greater than 75".

```
trees[trees$Height > 75, ]
```

And, of course, we can combine logical operations to be more picky in what we look for in our dataset.

```
trees[trees$Height > 75 & trees$Girth < 12, ]
```

Exercise:

1. Try selecting a subset of trees with either a Girth below 13, or a Height above 80
2. Try selecting a subset of trees where their Height is greater than their Volume plus 40
3. Using `sum()`, count how many trees have a volume below 15

Lists

Now we've looked at both vectors and data.frames. These are two of the four key types of objects that R can use to represent data. The other two are lists and matrices.

Let's start with lists, because they're the older cousin of data.frames. Like data.frames, lists aren't constrained by the rules of vectors – each element of a list can contain a different type of data. Actually, under the surface, data.frames *are* lists, with a special rule that each element of a data.frame is a vector of the same length as all the others, which is what makes data.frames so well-suited to hold our data.

Lists are under no such compunctions. Each element of a list can be any length you want it to be. In fact, lists can contain almost *anything* in R, including data.frames, matrices, and even other lists! That last part can get pretty trippy after a while, but just think of lists as our catchall container for everything, not caring what we store in them. For this reason, lists will come in very handy later when we want to read in and store a lot of datasets at once, like when we have a separate excel spreadsheet with the output from each participant's run of our experiment.

We'll come back to lists later, but let's put one together and take a look at it really quick.

```
shopping <- list('coffee', 'hairspray', 'apples', 'bananas')
shopping
```

Notice that the indices of the list look different to other objects we've looked at. At the top level, lists use double square brackets `[[2]]` to indicate the second element of the list. This is neccesary because, as I mentioned, lists can contain anything, including objects that can have their own indices. Below the `[[2]]` in our list of shopping, we can see a `[1]` before hairspray. That's indicating that hairspray is the first item in the second list element. We can see this in action if we put more than one item in each list element.

```
shopping <- list(c('coffee', 'hairspray'), c('apples', 'bananas'))
shopping
```

List subsetting can get pretty complicated, but essentially it works from left to right in a sort of tunnelling search. For instance, if you wanted to get the first item of the second list element above ('apples'), you'd specify [[2]] to go to the second list element, and then [1] to get the first element of the vector in that list element.

```
shopping[[2]][1]
```

If we name the elements of our list, we can also retrieve them using the `$` subsetting syntax we learned for data.frames. We assign names to objects by assigning a vector of the right length to the output of the `names()` function.

```
names(shopping) <- c('Sean', 'Michael')
shopping
```

The `$` operator will now work in the place of the double square bracket (`[[]]`). So you can use it in the tunnelling search subsetting as well. For instance, we can retrieve the shopping list for Sean, and by adding the `[1]` we can get the first item of that list.

```
shopping$Sean
shopping$Sean[1]
```

Don't worry is list subsetting seems a bit much right now. Lists are complicated beasts, and we won't be coming back to them for a while. We'll be sticking mostly to their smaller cousin, the data.set.

Matrices

Matrices are the last, and probably the least interesting of the four object types when we're actually interacting with R. However, they are very important under the surface for a lot of what we want to do with R. See, matrices are like datasets in that they have a two-dimensional structure, but they're like vectors in that they can only contain a single type of data. Usually, that data is numeric. Because matrices are simpler than data.frames, R can operate on them faster, and so matrices are used heavily in statistical computation to do the actual number crunching of our analyses (think factoring covariance matrices, estimating parameters, and minimizing least squares).

We can create a matrix with the `matrix()` command. Whatever we input will be formed into a matrix with a number of rows specified in `nrow`, starting at the top left and working down column-wise.

```
Bob <- matrix(1:25, nrow = 5)
class(Bob)
Bob
```

Matrices can have column names like data.frames, and they work with almost the same subsetting rules as data.frames, except that the `$` operator doesn't work.

```
colnames(Bob) <- c('A', 'B', 'C', 'D', 'E')
Bob[, 'A']
Bob$A
```

That's because the `$` subsetting syntax is specific to lists and data.frames. And while matrices look more like data.frames, if you scratch the surface, they're actually long vectors with a dimension attribute which tells R how many rows and columns to arrange the vector in to form the matrix.

Because Bob the matrix is mostly a vector in disguise, he can be operated on much like a vector. Many operations will apply to each element of a matrix, just as they would to each element of a vector.

```
Bob + 2
Bob * 3
Bob + Bob
```

Some functions will actually treat a matrix as one long vector. For example, the `cumsum()` function takes the cumulative sum of its arguments, which means it adds each argument to the sum of the previous arguments as it goes along a vector. We can see that it works its way through the matrix column by column, starting from the top and working its way down.

```
cumsum(c(1, 2, 3, 4))

cumsum(Bob)
```

If you've reached the end of this handout, then you can get started on the exercises.R file for today, delving more into subsetting.