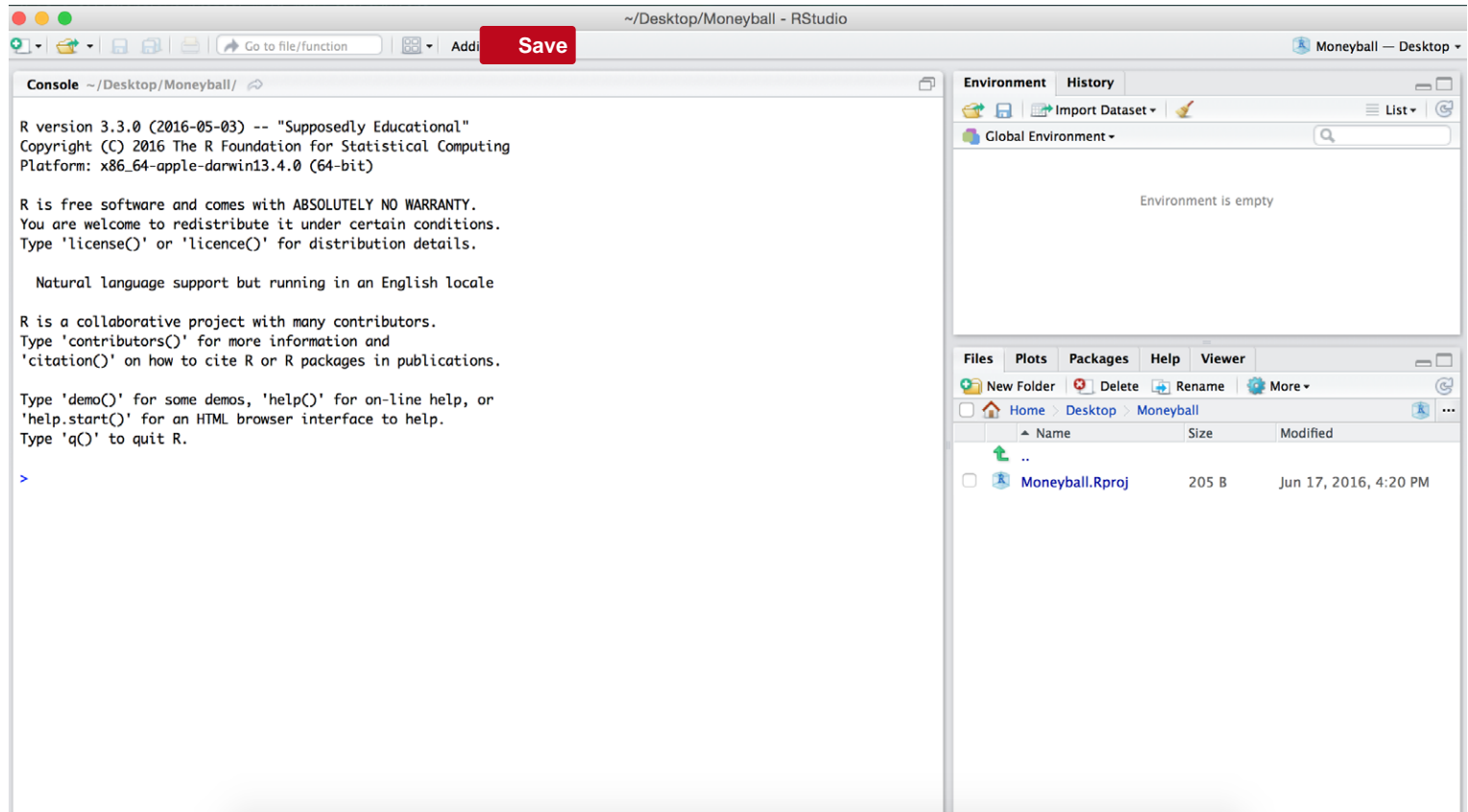


# Lecture 1

## Basic Usage

When you open RStudio for the first time, it should open up a new window that looks something like this:



The RStudio window is divided into three panes. For now, we will focus only on the big pane on the left, the **Console** pane. At the top of the **Console** pane, there is a bit of introductory text that shows the version of R that we are using and the license. Below that, you should see a `>`, which is the *R prompt*. As we mentioned earlier, at its core, R uses a command-line interface which means that you interact with the software by typing commands and hitting **Enter** or **Return**.

The simplest thing we can do is to use R as a calculator. For instance, we can type `2 + 3 * (6 - 4)^5` after the `>` and then hit **Enter**. We see that R returns the value of the expression  $2 + 3 \times (6 - 4)^5$  immediately underneath and the answer is preceded by a `[1]`. For now, don't worry about what this `[1]` means; we'll return to it later.

Hide

```
> 2 + 3 * (6 - 4)^5
[1] 98
```

Like any scientific calculator, R comes with a number of built-in functions

Hide

```
> sqrt(4)
[1] 2
> log(10)
[1] 2.302585
> cos(pi)
[1] -1
```

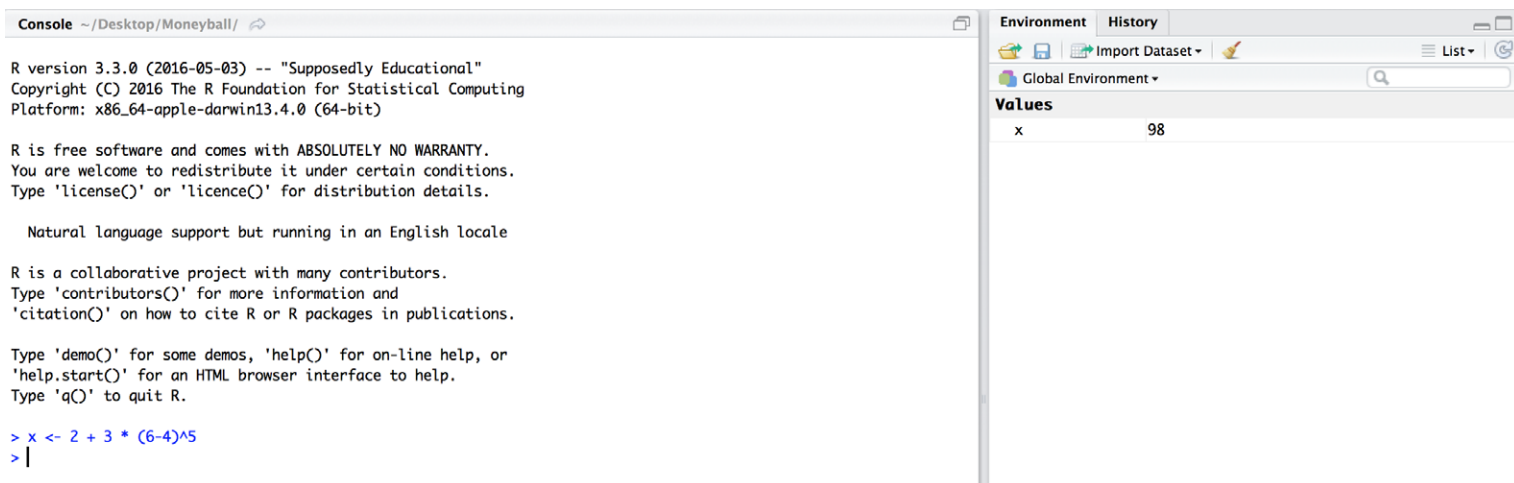
## Assignment

When you enter expressions like those above, R evaluates them, prints them, and then immediately discards them. Oftentimes, however, you'll want to store a value as a named variable and use it in subsequent calculations. For instance, let's say that we want to store the value of  $2 + 3 \times (6 - 4)^5$  as  $x$  and then compute the following:  $1/x$ ,  $x + 1$ , and  $\sqrt{x}$ . To *assign* the value of the expression `2 + 3*(6 - 4)^5` to the variable `x` we use the *assignment operator* `<-`. The assignment operator evaluates the expression immediately to the right of it and stores that value in an object whose name is whatever text came to the left of the operator.

Hide

```
> x <- 2 + 3 * (6 - 4)^5
```

Now when we execute this expression, R does not automatically print anything like it did in an earlier example. However, if you look closely at the **Environment** pane (top right-hand side of the RStudio window), you'll see that it now lists  $x$  and its value 98.



The screenshot shows the RStudio interface. The console on the left displays the R version (3.3.0) and the assignment of  $x$  to the value 98. The Environment pane on the right shows the variable  $x$  with the value 98.

This pane will show every variable that we have defined. As we start creating more and more variables, this list will be really helpful to keep track of what we've defined. Now that we have created the variable  $x$ , we can use the symbol `x` in more expressions. For example, we can compute  $1/x$ ,  $x + 1$ , and  $\sqrt{x}$  as follows:

Hide

```
> x
[1] 98
> 1/x
[1] 0.01020408
> x + 1
[1] 99
> sqrt(x)
[1] 9.899495
> round(3.14159, digits = 2)
```

```
[1] 3.14
> round(3.14159, digits = 4)
[1] 3.1416
> round(sqrt(x), digits = 4)
[1] 9.8995
```

In the last three examples, we used the function `round()`, which takes 2 arguments (entered inside the parantheses). The first argument is the number that we want to round and the second argument (following the comma) is the number of digits to which we want to round the first argument. This is our first example of a multi-argument function and we'll be seeing a lot more of them later on. In the last example above, we didn't give `round` an explicit number to round. Instead, R first evaluated `sqrt(x)` and then rounded it. R follows the conventional order of operations, in that it evaluates the inner-most expression first and then works its way out.

If we try to evaluate an expression using the name of a variable that we haven't defined, R will throw an error because we have not yet created a variable named `y`. Additionally, if you try to create a variable whose name starts with a number (like `2a`), R will throw an error.

Hide

```
> y + 5
[1] 2 10 7 10 14 12
```

Hide

```
> 2a <- 5
Error: <text>:1:2: unexpected symbol
1: 2a
  ^
```

## Exercises

1. Save the value of `8/5 + 3^3` as a new variable `y`.
2. Save the values of `y/x` as a new variable `z`.
3. Compute the square root of `x`, `y` and `z`.
4. Round these values to 3 decimal points.

## Data Types

R operates on several *data types* but we will focus on the following three:

- numeric: decimal numbers like 1.0, 3.4, -1.245. By far, these will be the most common.
- character: strings of letters and numbers, which are always enclosed in double quotes.
- logical: `TRUE` and `FALSE`, often created by logical operators. For instance, if we ask R whether 2 is less than 3, it returns the value `TRUE`. We will frequently use logicals when we are exploring large data sets and want to extract only some entries from a larger data set (e.g. finding all NBA players who took more than 100 3 point shots or all shots in the NHL taken from beyond 15 feet from the net).

Hide

```
> 2 < 3
[1] TRUE
> x <- 2 < 3
> x
```

```
[1] TRUE
```

# Vectors

In an earlier exercise, we asked you to compute the square root of four different numbers. Imagine if we had instead asked you to compute the square root of four hundred different numbers. Typing out each of those expressions would get really tedious and it would rather difficult to print out the results in one go. Luckily, many of R's built-in functions act on **vectors**, which allow us to do the same calculation on a lot of different values at once. At a very high level, a vector is simply a one-dimensional array that can store numeric data, character data, or logical data.

## Creating vectors

To create a vector, we use the combine function `c()` and enter the elements of that vector within the parantheses separated by commas. In the example below, we create a vector `v` with four elements: 1,2, 3, and 5.

Hide

```
> v <- c(1, 2, 3, 5)
> v
[1] 1 2 3 5
```

Once we have created a vector, we can evaluate arithmetic expressions of element of the vector with very straightforward syntax.

Hide

```
> v + 5
[1] 6 7 8 10
> v/5
[1] 0.2 0.4 0.6 1.0
> v^5
[1] 1 32 243 3125
> sqrt(v)
[1] 1.000000 1.414214 1.732051 2.236068
```

In each of these examples, we see that arithment proceeds *elements-wise*: `v + 5` added 5 to every element of `v` while `sqrt(v)` took the square root of every element in `v`. We can also do elements-wise arithment with multiple vectors.

Hide

```
> w <- c(-1, -3, -4, 5)
> v + w
[1] 0 -1 -1 10
> v/w
[1] -1.000000 -0.6666667 -0.750000 1.000000
```

To evaluate `v + w`, R took the first element of `v` and added it othe first element of `w` and so on. In order for this to work, `v` and `w` have to be the same length. Technically, R is able to handle element-wise arithmetic when the vectors are not of the same length (this is known as recycling) but we will not get into it. **For the purposes of this class, whenever you want to do element-wise arithmetic with two vectors, they need to be of the same length.** To determine the length of a vector we can use the function `length()` as follows:

Hide

```
> length(v)
[1] 4
> length(w)
[1] 4
> length(c(1, 2, 3))
[1] 3
```

We can also create character vectors. Below, we create a vector `mvp.winners` which contains the names of the last three NBA players to win the Most Valuable Player award.

Hide

```
> mvp.winners <- c("Stephen Curry", "Kevin Durant", "LeBron James")
> mvp.winners
[1] "Stephen Curry" "Kevin Durant"  "LeBron James"
```

## Special Vectors

Later in the course, we will need to use a few special vectors, like a vector whose elements are all the same or a vector containing a sequence of equally spaced numbers. To create such vectors, it would be really tedious to have to use the `c()` function and enter the elements manually. Luckily there are a few shortcuts for creating special vectors.

Hide

```
> rep(x = 1, times = 10)
[1] 1 1 1 1 1 1 1 1 1 1
> seq(from = 0, to = 1, by = 0.2)
[1] 0.0 0.2 0.4 0.6 0.8 1.0
> 1:20
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

In the first example above, we used the `rep` function to create a vector of 10 1's. Between the parentheses, we had to specify 2 arguments: `x` gives the value of we want to repeat and `times` gives the number of times we want to repeat the value of `x`. In the second example we created a sequence of equally spaced numbers from 0 to 1 with a spacing of 0.2. The third example created a vector of consecutive integers from 1 to 20.

## Useful Vector Functions

Up to this point, we have seen how to perform element-wise arithmetic with vectors. But sometimes, there are times when we want to compute a function of all of the data contained in the vector. For instance, we may want to compute the arithmetic mean of the elements of a vector or the maximum of all of the elements. There are several functions that operate on the entire vector and not element-wise. As an exercise, see if you can describe what the following functions do, based on their name and output.

Hide

```
> sum(v)
[1] 11
> mean(v)
[1] 2.75
> sd(v)
[1] 1.707825
```

```
> min(v)
[1] 1
> max(v)
[1] 5
> median(v)
[1] 2.5
> range(v)
[1] 1 5
> length(v)
[1] 4
```

## Naming the elements of a vectors

Finally, it is possible to *name* the elements of a vector. To do this, we use the `names()` function.

[Hide](#)

```
> mvp.wins <- c(2, 1, 2)
> mvp.wins
[1] 2 1 2
> names(mvp.wins) <- mvp.winners
> mvp.wins
Stephen Curry   Kevin Durant   LeBron James
           2             1             2
```

When we use `names()`, we put the vector whose elements we want to name in the parantheses and on the right-hand side of the `<-` we put a character vector of the same length as the vector we want to name containing the element names. In the example above, we created a numeric vector `mvp.wins` containing the number of times Stephen Curry, Kevin Durant, and LeBron James have won the MVP award.

## Excercise: NBA shooting percentages

The table below lists the number of field goal attempts and makes, three point attempts and makes, and free throw attempts and makes for several players in the 2015-16 NBA season. We will try to find the best shooters from this list.

PLAYER	FGM	FGA	TPM	TPA	FTM	FTA
Stephen Curry	805	1597	402	887	363	400
John Wall	572	1349	115	327	272	344
Jimmy Butler	470	1034	64	206	395	475
James Harden	710	1617	236	657	720	837
Kevin Durant	698	1381	186	480	447	498
LeBron James	737	1416	87	282	359	491
Kristaps Porzingis	498	1112	126	342	250	280

Dirk Nowitzki	373	886	81	243	201	240
Tim Duncan	215	442	0	2	92	131
Andre Drummond	552	1061	2	6	208	586

## Exercises

5. Create a vector `players` containing the names of the players from the table above.
6. Create vectors `fgm`, `fga`, `tptm`, `tpta`, `ftm` and `fta` which contains the number of field goals made and attempted, three pointers made and attempted, and free throws made and attempted for each player. Be sure to name the elements of these vectors using `players`.
7. Field goal percentage (FGP) is perhaps the simplest way to assess how good of a shooter a player is. To compute FGP, you simply divide the number of field goals made by the number of field goals attempted. Create a vector `fgp` which computes the the field goal percentages of these players. Who had the highest field goal percentage?
8. Create vectors `tpp` and `ftp` which contain each player's 3-point percentage and free throw percentage. Be sure to name the elements of these vectors. Which players have the highest three point percentages and free throw percentages?
9. If a player's field goal percentage, three point percentage, and free throw percentage exceed 50%, 40%, and 90%, respectively, they are said to belong to the [50-40-90 club](#). Did any of the players in the table above qualify for the club?
10. One problem with FGP is that it treats 2-point shots the same as 3-point shots. Typically the league-leader in FGP is a center like DeAndre Jordan, whose shots mostly come from near the rim. [effective Field Goal Percentage](#) is a statistics that adjusts FGP to account for the fact that a made 3-point shots is worth 50% more than a made 2-point shot. The formula for eFGP is

$$\text{eFGP} = \frac{\text{FGM} + 0.5 \times \text{TPM}}{\text{FGA}}.$$

Create a vector `efgp` which contains the effective field goal percentage of the players listed in the table. Be sure to name the elements of this vector! Which player has the largest difference between their FGP and eFGP?

## Indexing

Up to this point, we have created and manipulated vectors. In the exercises above, we took advantage R's ability to perform element-wise arithmetic on vectors. But what if we had a vector `v` and only wanted to add 3 to the first element? We could execute `v + 3` and look at the first element, but this is rather inefficient since we're asking R to add 3 to **every** element of `v` in the process. Moreover, what if we wanted to take the first element of a vector `v`, add 3 to it, and then save that value as a new variable `a`? As a more concrete example, what if we wanted to compute the difference in Stephen Curry's field goal percentage and Dirk Nowitzki's field goal percentage? To do this, we need a way to *extract* specific elements of a vector. This is known as **subsetting** and is an **EXTREMELY** important aspect of R since it allows us to extract and modify specific elements of an object (e.g. change the third element of `v` from 3 to -3). To subset vectors, we will use the single-bracket notation `[ ]`. Between the brackets we will specify which elements we want to extract. So to extract the first element we would execute `v[1]` and to extract the third element we would execute `v[3]`:

Hide

```
> v[1]
[1] 1
> v[3]
```

```
[1] 3
> x <- v[1] + 3
> x
[1] 4
> v[3] <- -3
> v
[1] 1 2 -3 5
```

To get multiple elements at one time, we put a vector between the brackets:

Hide

```
> v[c(1, 2)]
[1] 1 2
> v[1:2]
[1] 1 2
> players[c(1, 3)]
[1] "Stephen Curry" "Jimmy Butler"
```

We can also use names to subset vectors. Instead of putting a number (or vector of numbers) between the bracket, we now put a character string (or vector of character strings) specifying the names of vector elements we want to subset.

Hide

```
> fgp["Stephen Curry"]
Stephen Curry
0.5040701
> fgp[c("Stephen Curry", "Dirk Nowitzki")]
Stephen Curry Dirk Nowitzki
0.5040701 0.4209932
> fgp["Stephen Curry"] - fgp["Dirk Nowitzki"]
Stephen Curry
0.0830769
```

In the last example, we actually used a vector to subset another vector. The resulting output is itself a vector (of length 2)

## Logical Comparison

What if we wanted to extract all elements of a vector which were greater than 3? Or, as a more concrete example, what if we wanted to find the three point percentages of all players who took at least 250 three pointers?

We have the following logical comparison operators at our disposal

- `==` for “equal to”
- `!=` for “not equal to”
- `<` and `<=` for “less than” and “less than or equal to”
- `>` and `>=` for “greater than” and “greater than or equal to”

We can use these comparison operators directly in R:

Hide

```
> tpa >= 250
Stephen Curry      John Wall      Jimmy Butler
      TRUE          TRUE          FALSE
```

James Harden	Kevin Durant	LeBron James
TRUE	TRUE	TRUE
Kristaps Porzingis	Dirk Nowitzki	Tim Duncan
TRUE	FALSE	FALSE
Andre Drummond		
FALSE		

This returns a logical vector indicating whether each element of `tpm` is greater than or equal to 250. Since the vector `tpm` was named, the resulting logical vector was named as well. While this is informative, to determine which players actually attempted more than 120 three pointers, we would have to examine the results. This can be tedious (if not outright impossible) if our vector contained, say, the number of three pointers attempted by every NBA player since the 1996-97 season! To determine **which** elements of `tpm` are at least 120, we use R's `which()` function. Between the parantheses, we type in a comparison expression like `tpa >= 120`. The function first evaluates the comparison expression and then reports which elements of the resulting logical vectors are `TRUE`:

Hide

```
> which(tpa >= 250)
Stephen Curry      John Wall      James Harden
               1               2               4
Kevin Durant      LeBron James  Kristaps Porzingis
               5               6               7
```

We can use the results of the above expression to subset other vectors to find, for instance, the field goal percentages of players who attempted at least 250 three point shots. First, we will create a vector `my.index` which contains the results of `which(tpa >= 250)`. Then we will use `my.index` to subset the vector `fgp`.

Hide

```
> my.index <- which(tpa >= 250)
> fgp[my.index]
Stephen Curry      John Wall      James Harden
    0.5040701      0.4240178      0.4390847
Kevin Durant      LeBron James  Kristaps Porzingis
    0.5054308      0.5204802      0.4478417
```

Finally, there are two special functions that you should know about: `which.min()` and `which.max()`. Sometimes, we want to know the index of the the largest or smallest element of a vector and not the actual value of the largest or smallest. These two functions return exactly that:

Hide

```
> which.min(fgp)
Dirk Nowitzki
               8
> which.max(fgp)
LeBron James
               6
```