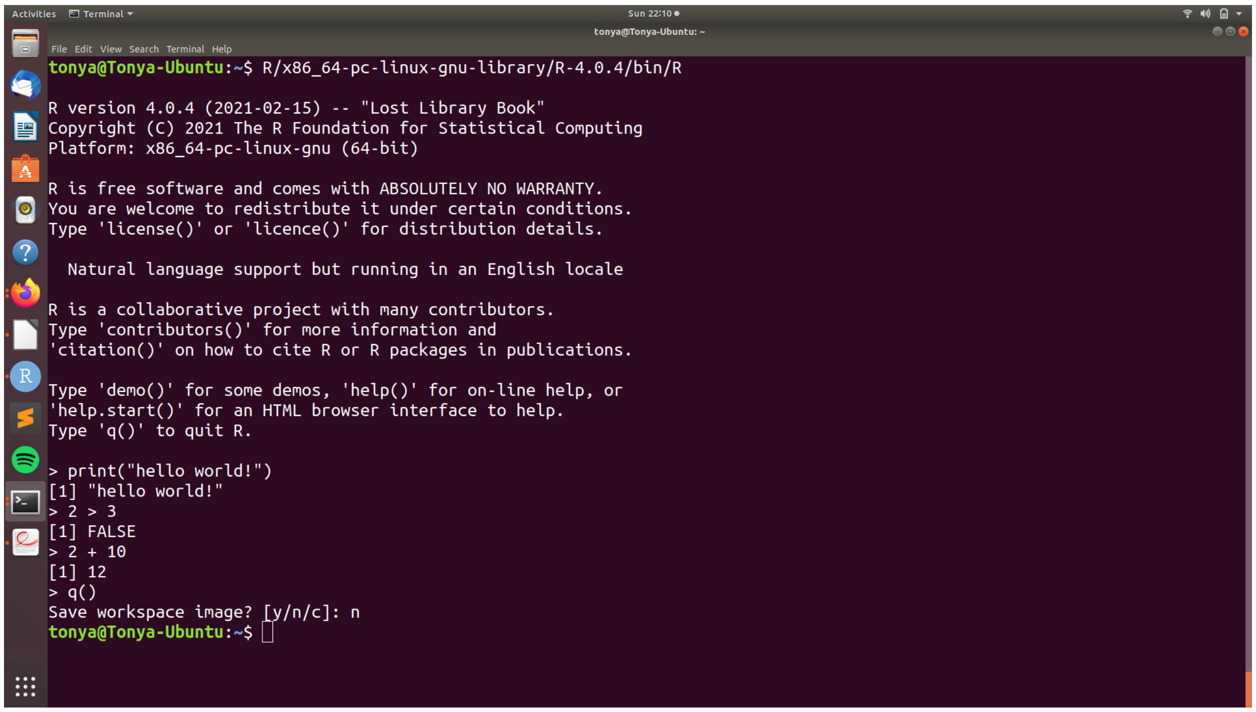
Intro to Rstudio and the R Programming Language

Tonya Brunetti

4/4/2021

Understanding the Rstudio environment

Rstudio is a GUI (graphical user interface) to the R programming language. You will notice, that if you want to use R studio, you must install basic R first. This is because Rstudio is just a visually, user-friendly interface to the R programming language. **Rstudio is NOT a programming language**! If you launched R without Rstudio, your computer would open an interactive R session on the command line. Rstudio hides this command line from the user and offers an enviroment that is much more appealing. If you open basic R on your computer or double click on the basic R icon, it will show you an interactive R session from the command line. For example:



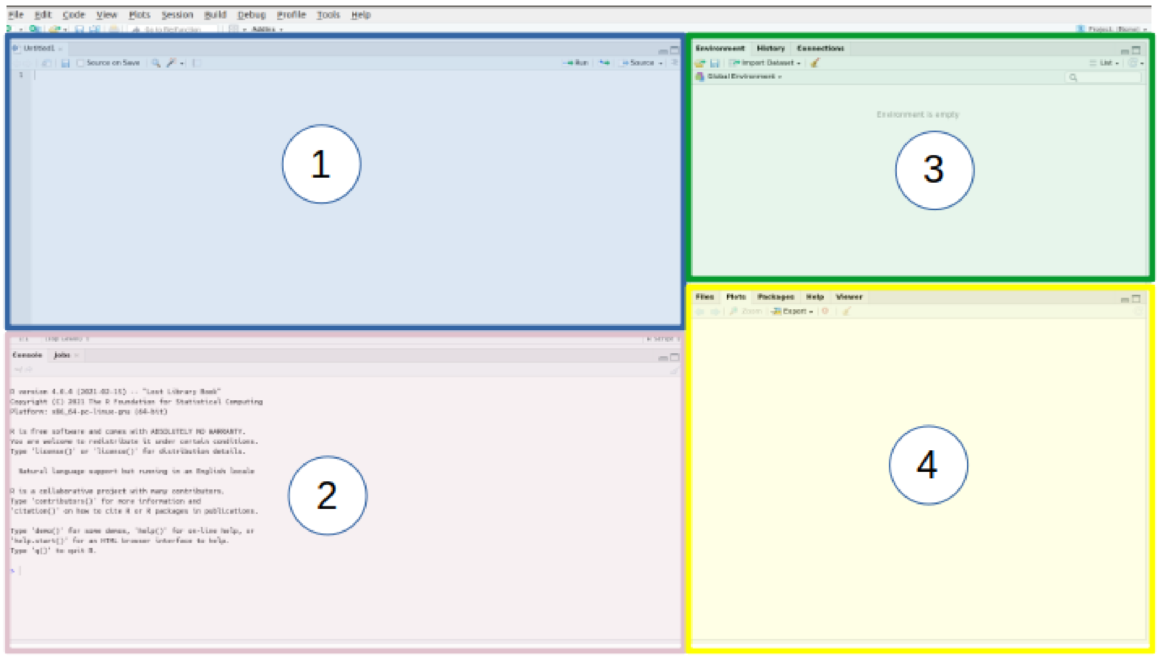
```
tonya@Tonya-Ubuntu:~$ R/x86_64-pc-linux-gnu-library/R-4.0.4/bin/R

R version 4.0.4 (2021-02-15) -- "Lost Library Book"
Copyright (C) 2021 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> print("hello world!")
[1] "hello world!"
> 2 > 3
[1] FALSE
> 2 + 10
[1] 12
> q()
Save workspace image? [y/n/c]: n
tonya@Tonya-Ubuntu:~$
```

Figure 1: Interactive Command Line R

Rstudio utilizes the same programming language except now, a nice visual interactive interface sits on top of it so that the user can use the interface to set up their environment, write code, test code, view environmental variables and paths, and execute codes in a manner that is intutitive. Rstudio takes what you write and runs it within Rstudio and essenetially executes all the code using the same command line environment that you see above, except that you the user never has to open the command line environment – cool!

Rstudio sections

You will notice when you open Rstudio, it is naturally broken down into four different sections. Each section has a particular function within the GUI and helps the user organize code, variables, show code that has been executed, and see what is loaded into the environment.


File Edit Code View Plots Session Build Debug Profile Tools Help
Untitled1
Source on Save
Run
Source
Environment History Connections
Import Dataset
Global Environment
Environment is empty
Console Jobs
Files Plots Packages Help Viewer
Zoom
Export
1
2
3
4

```
R version 4.0.4 (2021-02-15) -- "Lost Library Book"
Copyright (C) 2021 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

Figure 2: Sections of Rstudio

1. Scripts

   A script is a collection of commands that is saved to a file so that you can run the program (script) again using the same commands. This is useful if you want to save your code and you want to document what you have done, share/send the code you ran with a collaborator, or if you think you want to run your code in a different location on another computer. A script does not generally require that a person needs to manually write each of your written commands line-by-line. They can just execute your file and it will automatically call all the commands you have written sequentially**.

   ** I say sequentially, because if you have functions or parallelism or concurrency, this does not apply in terms of code execution order. For the purpose of this mini-workshop, all of you code will be run sequentially.

2. R Console

   This is essentially the same as running your code in the R interactive command line. In Rstudio, if you press the **Run** button in your scripts section, it will automatically copy that command into the R Console section here, and execute the code and show you the result. You can choose to write code here, however, beware, any code you write directly into the Console is not saved in your Rscript, unless you use the **Run** button from your Rscript to populate the Console. This is good for testing what a command or section of code does or what the result of executed code returns before you add it to your script.

3. Environmental Variables & Log History

   This section is primarily used for one of two things: show you all the environmental variable and objects currenly loaded in your Rstudio session and to record a log or history of all the commands you have run.

   The **Environment tab** is probably the most used tab in this section. It is very convenient because it shows you all the variables, files, objects you have loaded into your current Rstudio session and what the value and type are for each variable.

   The **History tab** shows you all the commands you have run in the Console from most recent to least recent. One cool feature of the **History tab** is it has two buttons associated with it: **To Console** button and **To Source** button. If you highlight a section of code in the history/log line and press **To Console**, it will automatically populate in the R console so you don't have to re-type it. If you highlight a section of code in the history/log line and press **To Source**, it will automatically populate in your script section so that you don't have to re-type it and you can save it to a file.

4. OS File Viewer, Plots, Packages, & Help Section

   This section is a hodge-podge of other useful windows. The first tab, **Files tab**, lists files and directories located on your computer. You can actually navigate through your computer through this panel.

   The **Plots tab** is where all of the graphs and figures you generate in R are located. This tab has zoom capabilities and the ability to export your graph as an image saved to your computer. Also, one cool feature of the plots tab is as long as you are in the same Rstudio session, it keeps a library of all of your plots for the duration you have Rstudio open. You can use the left and right arrows in the upper left corner of the plots tab to move between graphs.

   The **Packages tab** is one I rarely, if ever, use but is shows you R packages that are installed. If there is a check mark in the left box, it means you have that library loaded into your current session. If it is unchecked, it means the library is installed but not loaded into your current session. By checking the box, the R Console will populate and call the library command followed by the package you want loaded into your current environment. One nice thing, is it does list the package version, which is useful for publication purposes and reproducibility of analyses.

   The **Help tab** is useful for looking at documentation for a command or R package. In the R Consolse, if you type `?` followed by a library name or R command it will populate the **Help tab** with documentation on how to use it and other suggested linked resources.

Understanding Basic R Data Types

There are a few basic R data types that you should familiarize yourself with. The three that I am going to discuss in this section are characters, numerics, and booleans.

- Characters
  Characters in are typically alphabet strings surrounded by single and/or double quotation marks. One or more characters is equal to a string and this value is to be taken literally as written. If a number surrounded with double or single quotes, that is also considered a character. It does not have a numeric value associated with it and is to be interpreted as a word and literal. You do not perform any mathematical operations on characters. Examples of characters are the following:

```
'Hello!'
"This is a character type!"
"42"
"12.0"
"32 + 3"
```

- Numerics
  Numerics are numbers, whether they be fractions, floats/doubles/decimals, integers, etc… They **never have single or double quotations around them** (otherwise they would be characters!). Numeric operations can be applied to this type. Examples include:

  ```
  42
  0
  1.023
  -3
  1/2
  ```

- Booleans Booleans are binary values that return either `TRUE` or `FALSE`. `TRUE` and `FALSE` are keywords in the R language so do not use these values as variable names, otherwise, it will get very confusing. These are some examples of expressions that will return a boolean:

```
1394 > 0
```

```
## [1] TRUE
```

```
"3" == "1"
```

```
## [1] FALSE
```

```
1/2 < 3/4
```

```
## [1] TRUE
```

Variables

Variables are a names that you give to values/objects so that you can store/save a value/object and also reference it by name without having to hard-code or update your code over and over. Variables are set by assigning a name to a value to a character, boolean, numeric, categories (factors), lists & vectors, objects (such as tibbles and dataframes), etc… There are 3 ways to assign a value to a variable. One way is to use a single `=`. Examples:

```
i = 2
val = "hello"
boolVal = FALSE
```

In the case above, you will notice all the names of the value **must be on the left-hand side of the** `=`. The value of that variable is located to the right of the `=`. Everytime I call or type `i` into my code, it will replace `i` with 2. Similarly everywhere I call or code `val` it will replace `val` with the string "hello" and the same for `boolVal`.

Another way to assign variable is to use the arrows symbols: `<-` and `->`. The direction of the arrow dictates where the name of the variable will be located. For example:

```
n <- "this is a value"
height <- 60.4
8012 -> numOfSteps
```

In the example above, you will see both variables, `n` and `height` are located on the left-hand side of the value. The arrow shows you the variable name will be located to the left. However, for the variable `numOfSteps` the value is on the left and the variable name is on the right. The arrow is pointing to the right, therefore, that must be the variable name.

Table of Basic Operators in R

Below is a list of some basic operators in R and what they return. This is by no means an exhaustive list, just the most commonly used ones. You will have a chance to experiment with these operators in the excersices at the end of this section. If you open someone else's R code, you will see many of these in use. By getting familiar with these operators, not only can you write your own code, but also you will be able to read others' code as well!

operators	explanation	return type
>	greater than	boolean
<	less than	boolean
==	is equal to	boolean
>=	greater than or equal to	boolean
<=	less than or equal to	boolean
&	compound and	boolean
|	compound or	boolean
! (-)	negate or not	boolean
!=	not equal to	boolean
=	assign variable value	assignment
<-	assign variable (left)	assignment
->	assign variable (right)	assignment
""	assign value as string	string
*	multiply operator	numeric
**	exponential operator	numeric
+	addition operator	numeric
-	subtraction operator	numeric
/	division operator	numeric

Exercises:

1. Execute the following code. What do you expect it to return? If you look in your Rstudio Environment section, what are the types for a and b? Hint: `typeof(a)`, `typeof(b)`.

```
a = 3
b = 2
a + b
```

2. Execute the following code. What do you expect it to return? If you look in your Rstudio Environment section, what are the types for a and b? Hint: `typeof(a)`, `typeof(b)`.

```
a = "3"
b = "2"
a + b
```

3. Execute the following code. What do you expect it to return?

```
a = "3"
b = "2"
a == b
```

4. Execute the following code. What do you expect it to return?

```
a = 3
b = 2
a == b
```

5. What is the difference between `a == b` and `a = b`? Try running the following:

```
a = 10
b = -3
a == b
print(a)
print(b)
a = b
print(a)
print(b)
```

6. What do expect the variable `myValue` to return? Try running the following:

```
myFirstNumber = 200.3
mySecondNumber = 200.32
myValue <- myFirstNumber > mySecondNumber
print(myValue)
```

7. What do you expect the following to return?

```
x = 20
y = 21
x != y
```

8. Let's try something a bit trickier. What will the following return? How does this change if the `x` variable and `y` variable are not now strings instead of numbers? How do you explain each of the differences?

```
x = 20.0
y = 20
first <- x != y
print(first)

x = "20.0"
y = "20"
second <- x != y
print(second)

x = "hello"
y = "Hello"
thrid <- x == y
print(thrid)

x = "I am coding!"
y = "I am Coding!"
fourth <- x == y
print(fourth)

x = "coding rules"
y = "coding rules"
fifth <- x == y
print(fifth)

x = "hello world!"
y = " hello world!"
```

```
sixth <- x == y
print(sixth)
```

9. Look familiar? You may notice that we are calculating distance or displacement from physics class. What do you expect the variable `myDistanceCalculation` to return? Try running the following:

```
acceleration <- 9.8
2.0 -> initialVelocity
time = 3.2
myDistanceCalculation = (1/2)*acceleration*(time**2) + initialVelocity*time
print(myDistanceCalculation)
```

A few notes to take-away from the exercises

I am hope the above exercises help to clarify what most of the operators in the table are used for and what they return. There are a few things you may have noticed by executing the code for each exercise.

- A single `=` means you are changing the value/assigning a new value of a variable while double `==` means you are asking whether the value/variable to the left is the same as the value/variable to the right. (See Exercise 5)
- The double `==` can be applied to both strings and numeric types (See Exercise 3 & 4)
- You cannot add variables that are type string or non-numeric (See Exercise 2)
  Hint: If you want to concatenate strings you can use the `paste()` *function in R. For example:*

```
?paste # this will bring up the help page for how to use the command paste()

a = "3"
b = "2"
paste(a, b, sep = "")

a = "hello"
b = "world!"
paste(a, b, sep = " ")
```

- R drops non-significant digits in numbers (See Exercise 8, first code chunk for variable `first`)
- R is case-sensitive. Capitalization and spelling matter! (See Exercise 8, for variable `third`, `fourth`, `fifth`)
- Whitespaces are not your friend – try not to use them in code! (See Exercise 8, last code chunk for variable `sixth`)
- When assigning a value to a variable, all three of these can be used for variable assignment: `<-`, `=`, `->`. (See Exercise 9)
  – The main difference is the arrows tell you which direction the variable name is located. If you use an `=` then the variable name always has to go to the left. `=` and `<-` are thus equivalent. `->` means the value is to the left and the variable name is to the right.

The Main Data Object in R: the dataframe

The main data object in R is the dataframe and is a tabular way to store data, similar to Excel. It contains rows and columns of data that you can subset, query, apply operations, and many more. For the purpose of today, we are going to use a dataset that is already built into the R language called `mtcars`. This data set contains information about different cars and for the purpose of learning dataframe functions, will work out well. First, let's make sure everyone can see the `mtcars` dataset. To print the full dataset to your R console, you should just type the following:

```
mtcars
```

```
##                      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
## Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
## Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
## Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
## Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
## Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
## Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
## Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
## AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
## Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
## Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
## Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
## Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
## Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
## Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
## Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
## Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

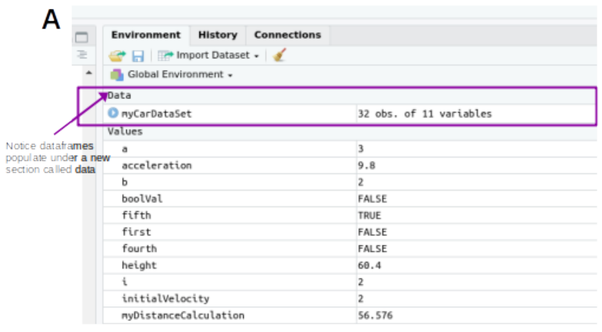
This data set has a lot of column abbreviations without much information. In order to get a full explanation of the dataset, we can use the `?` that we learned about at the beginning of the tutorial.

```
?mtcars
```

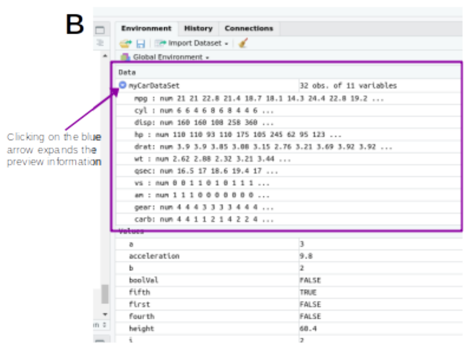
You will quickly realize that printing the full data frame to your console is not at all helpful! So let's go ahead and store the dataframe as a variable:

```
myCarDataSet <- mtcars
```

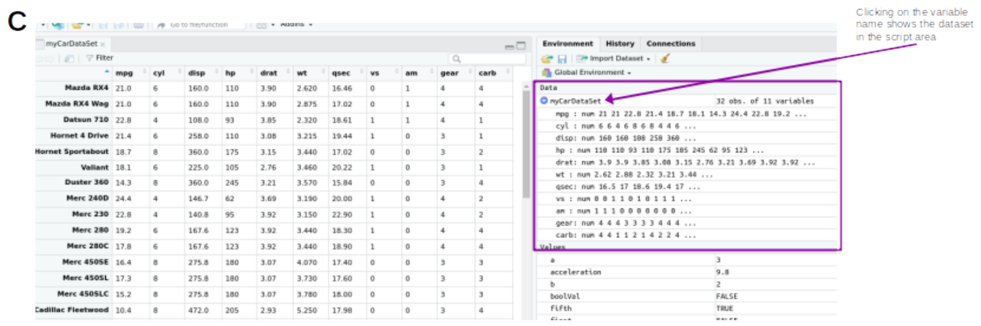
As soon as your set it to a variable, the variable will appear in your environmental variables section in Rstudio under the data section. It has a little blue arrow which means you can expand it by clicking on it and it will show you a preview of the dataset.



```
Data
myCarDataSet              32 obs. of 11 variables
Values
  a                       3
  acceleration            9.8
  b                       2
  boolVal                 FALSE
  fifth                   TRUE
  first                   FALSE
  fourth                  FALSE
  height                  60.4
  i                       2
  initialVelocity         2
  myDistanceCalculation   56.576
```



```
Data
myCarDataSet              32 obs. of 11 variables
  mpg : num 21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
  cyl : num 6 6 4 6 8 6 8 4 4 6 ...
  disp: num 160 160 108 258 360 ...
  hp  : num 110 110 93 110 175 105 245 62 95 123 ...
  drat: num 3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
  wt  : num 2.62 2.88 2.32 3.21 3.44 ...
  qsec: num 16.5 17 18.6 19.4 17 ...
  vs  : num 0 0 1 1 0 1 0 1 1 1 ...
  am  : num 1 1 1 0 0 0 0 0 0 0 ...
  gear: num 4 4 4 3 3 3 3 4 4 4 ...
  carb: num 4 4 1 1 2 1 4 2 2 4 ...
Values
  a                       3
  acceleration            9.8
  b                       2
  boolVal                 FALSE
  fifth                   TRUE
  first                   FALSE
  fourth                  FALSE
  height                  60.4
  i                       2
```



	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4
Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3
Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3	3
Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3
Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4

```
Data
myCarDataSet              32 obs. of 11 variables
  mpg : num 21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
  cyl : num 6 6 4 6 8 6 8 4 4 6 ...
  disp: num 160 160 108 258 360 ...
  hp  : num 110 110 93 110 175 105 245 62 95 123 ...
  drat: num 3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
  wt  : num 2.62 2.88 2.32 3.21 3.44 ...
  qsec: num 16.5 17 18.6 19.4 17 ...
  vs  : num 0 0 1 1 0 1 0 1 1 1 ...
  am  : num 1 1 1 0 0 0 0 0 0 0 ...
  gear: num 4 4 4 3 3 3 3 4 4 4 ...
  carb: num 4 4 1 1 2 1 4 2 2 4 ...
Values
  a                       3
  acceleration            9.8
  b                       2
  boolVal                 FALSE
  fifth                   TRUE
```

Getting Size and Names in Dataframes

There are a few commands that are helpful in getting infomration about a dataframe, especially when they get very large!

```
dim(myCarDataSet)
```

```
## [1] 32 11
```

```
colnames(myCarDataSet)
```

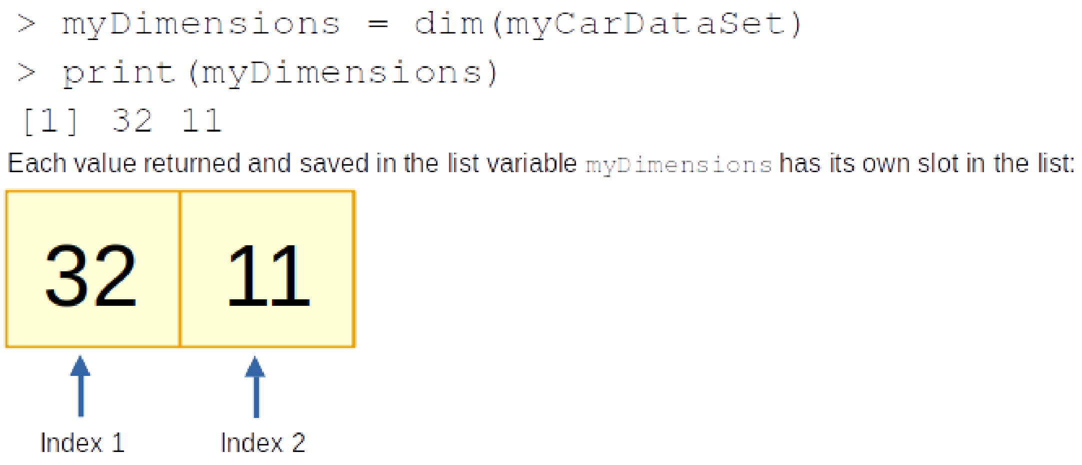
```
##  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
## [11] "carb"
```

```
rownames(myCarDataSet)
```

```
##  [1] "Mazda RX4"           "Mazda RX4 Wag"       "Datsun 710"
##  [4] "Hornet 4 Drive"      "Hornet Sportabout"   "Valiant"
##  [7] "Duster 360"          "Merc 240D"           "Merc 230"
## [10] "Merc 280"            "Merc 280C"           "Merc 450SE"
## [13] "Merc 450SL"          "Merc 450SLC"         "Cadillac Fleetwood"
## [16] "Lincoln Continental" "Chrysler Imperial"   "Fiat 128"
## [19] "Honda Civic"         "Toyota Corolla"      "Toyota Corona"
## [22] "Dodge Challenger"    "AMC Javelin"         "Camaro Z28"
## [25] "Pontiac Firebird"    "Fiat X1-9"           "Porsche 914-2"
## [28] "Lotus Europa"        "Ford Pantera L"      "Ferrari Dino"
## [31] "Maserati Bora"       "Volvo 142E"
```

command	explanation	return type
dim()	dimensions of dataframe	list
colnames()	get all column names	list
rownames()	get all row names	list
length()	get the legnth of a list	integer
c()	create a new list	list

If we go ahead and store the values of each of these commands as a variable, R will save it as a list with each value stored in its own index in the list.



```
> myDimensions = dim(myCarDataSet)
> print(myDimensions)
[1] 32 11
```

Each value returned and saved in the list variable `myDimensions` has its own slot in the list:

Figure 3: Lists in R

Querying and Accessing Values in a Dataframe

Now that we have a dataframe and we know the column names and row names, lets see if you can pull out some data. If I wanted to only look at the total miles per gallon (mpg) for every car in the datafarme; you can pull out a single column using the `$` symbol. This will store all the values in the same order as the rownames for each car in a list:

```
mpgForEachCar <- myCarDataSet$mpg
print(mpgForEachCar)
```

```
##  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4
## [16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7
## [31] 15.0 21.4
```

However, there is another way to do it that returns a dataframe and allows you to select multiple columns. R can also be subset using `[,]`. This format follows the following convention: `[row(s), columns(s)]`. If we wanted to only select the `mpg` and `cyl` (number of cylinders the car has) columns for all cars we could do the following: **(Note: you will notice I use the new list command `c()`. Anytime you want to select more than one row or column in a dataframe, you have to give it to the brackets as a list.)**

```r
    mpgCylData <- myCarDataSet[,c("mpg", "cyl")]
    print(mpgCylData)
```

```
##                      mpg cyl
## Mazda RX4           21.0   6
## Mazda RX4 Wag       21.0   6
## Datsun 710          22.8   4
## Hornet 4 Drive      21.4   6
## Hornet Sportabout   18.7   8
## Valiant             18.1   6
## Duster 360          14.3   8
## Merc 240D           24.4   4
## Merc 230            22.8   4
## Merc 280            19.2   6
## Merc 280C           17.8   6
## Merc 450SE          16.4   8
## Merc 450SL          17.3   8
## Merc 450SLC         15.2   8
## Cadillac Fleetwood  10.4   8
## Lincoln Continental 10.4   8
## Chrysler Imperial   14.7   8
## Fiat 128            32.4   4
## Honda Civic         30.4   4
## Toyota Corolla      33.9   4
## Toyota Corona       21.5   4
## Dodge Challenger    15.5   8
## AMC Javelin         15.2   8
## Camaro Z28          13.3   8
## Pontiac Firebird    19.2   8
## Fiat X1-9           27.3   4
## Porsche 914-2       26.0   4
## Lotus Europa        30.4   4
## Ford Pantera L      15.8   8
## Ferrari Dino        19.7   6
## Maserati Bora       15.0   8
## Volvo 142E          21.4   4
```

Notice, if I leave the rows blank in the `[,]` it grabs every row. This is also the same for the columns. For example, let's select the first 3 rows of cars in the dataframe, but all the columns:

```r
   firstThreeCars <- myCarDataSet[1:3,]
   print(firstThreeCars)
```

```
##                mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4     21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710    22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
```

Exercises:

1. Without using the `dim()` command, write a way to get the total number columns in the `myCarDataSet`. Hint: use the `length()` command as one of your commands.

2. Without using the `dim()` command, write a way to get the total number of rows in the `myCarDataSet`. Hint: use the `length()` command as one of your commands.

3. Create a variable called `allWeights` which is a list of all the weights (wt) for every car.

4. Using R code only, write a way to get the horse power (hp) of the Fiat 128.

5. Create a new dataframe named `mySubset` where the data is subset to the first 4 columns of data only containing rows from the Maserati Bora and Volvo 142E.

Advanced Querying

Using the `which()` command

Complex queries can be addressed by using the `which()` command. The `which()` command returns indices/positions within a list where the questions or expressions you write are true given the input data. We can illustrate this concept with an example. Recall in the beginning of the tutorial, where using mathematical operators such as `>`, `<`, `==`, `>=`, and `<=` return boolean values. As a refresher:

```
i = 9
j = 0
i > j
```

```
## [1] TRUE
```

```
i == j
```

```
## [1] FALSE
```

You can use this same concept to apply to columns of a dataframe. Try to run the following code and see what it returns:

```
myCarDataSet$mpg > 18.0
```

```
##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE FALSE FALSE
## [13] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
## [25]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE
```

This returns a list of booleans. Each value in the list corresponds to the row number/name (in the same order as the dataframe), and a `TRUE` is listed if the `mpg` for that row is `TRUE` and if it is not true, it is listed as `FALSE`.

Now, what happens if we make the same call and wrap the `which()` command around it.

```
which(myCarDataSet$mpg > 18.0)
```

```
##  [1]  1  2  3  4  5  6  8  9 10 18 19 20 21 25 26 27 28 30 32
```

The list of numbers `which()` returns are all the rows/positions where `myCarDataSet$mpg > 18.0` is `TRUE`. This is very convenient because now we can subset the dataframe by supplying the row values the same list of numbers as what is returned from the `which()` command. A new dataset can now be made containing only cars that get more than 18.0 miles per gallon.

```
carsGoodmpg <- myCarDataSet[which(myCarDataSet$mpg > 18.0),]
print(carsGoodmpg)
```

```
##                    mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Merc 240D         24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230          22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 280          19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## Fiat 128          32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic       30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla    33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## Toyota Corona     21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
## Pontiac Firebird  19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
## Fiat X1-9         27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## Porsche 914-2     26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
## Lotus Europa      30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
## Ferrari Dino      19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
## Volvo 142E        21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

The `&` (and) and `|` (or) operators

You may have noticed that when we went over the R basic data types section and operators there were two operators we never discussed: `&` and `|`. These two operators can be used to query datasets with more complex compound questions. Lets say now we are interested in cars that not only get more than 18.0 miles per gallon, but are also 4 cylinder vehicles.

```
carsGoodmpgCyl <- myCarDataSet[which((myCarDataSet$mpg > 18.0) & (myCarDataSet$cyl == 4)),]
print(carsGoodmpgCyl)
```

```
##                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Merc 240D      24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230       22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Fiat 128       32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic    30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## Toyota Corona  21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
## Fiat X1-9      27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
## Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
## Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

Similary, how would we get all the cars that get at least 18.0 miles per gallon or have an automatic engine:

```
carsGoodmpgCyl <- myCarDataSet[which((myCarDataSet$mpg >= 18.0) | (myCarDataSet$am == 0)),]
print(carsGoodmpgCyl)
```

```
##                      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
## Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
## Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
## Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
## Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
## Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
## Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
## Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
## AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
## Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
## Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
## Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
## Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
## Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
## Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

Exercises:

1. Create a new dataframe that lists only cars that have a V-shaped engine (vs) and have at least 2 carburetors (carb).

2. Create a new dataframe where the cars have only 8 cylinders (cyl) or have 4 carburetors (carb). Of these cars, make it so that only the Transmission column (am) is part of the dataframe.