R and RStudio Basics

Getting started with R and RStudio

Created by Tania Alarcon, March 2018
Last edited by Kyle Monahan, April 2018

Contents

Skills Covered in this Tutorial Include:

- Using the RStudio IDE
- Installing and loading R packages
- Opening and running scripts
- Using R documentation from the Help Tab
- Creating, viewing, and manipulating common R data structures (atomic vectors, lists, matrices, and data frames)
- Creating and working with factors

1. Introduction

This tutorial is designed to get you started with the statistical programming language R and the RStudio Interface. R is an open-source, fully-featured statistical analysis software. You can work directly in R but we recommend using RStudio, a graphical interface. RStudio is an open-source, integrated development environment (IDE) for R. RStudio combines a powerful code/script editor, special tools for plotting and for viewing R objects and code history, and a code debugger.

In this tutorial, we provide a detailed overview of the RStudio IDE and its functionality. You will learn to navigate and use the Console, Source, Environment, and Files panes. We will guide you through setting a working directory, installing and loading R packages, opening and running scripts, and using R documentation from the Help Tab.

This tutorial also provides an overview of how R stores information. We will create, view, and manipulate the most common types of R data structures (atomic vectors, lists, matrices, and data frames).

This tutorial is suitable for those who have not worked with R/RStudio before. This tutorial may take a few hours to complete.

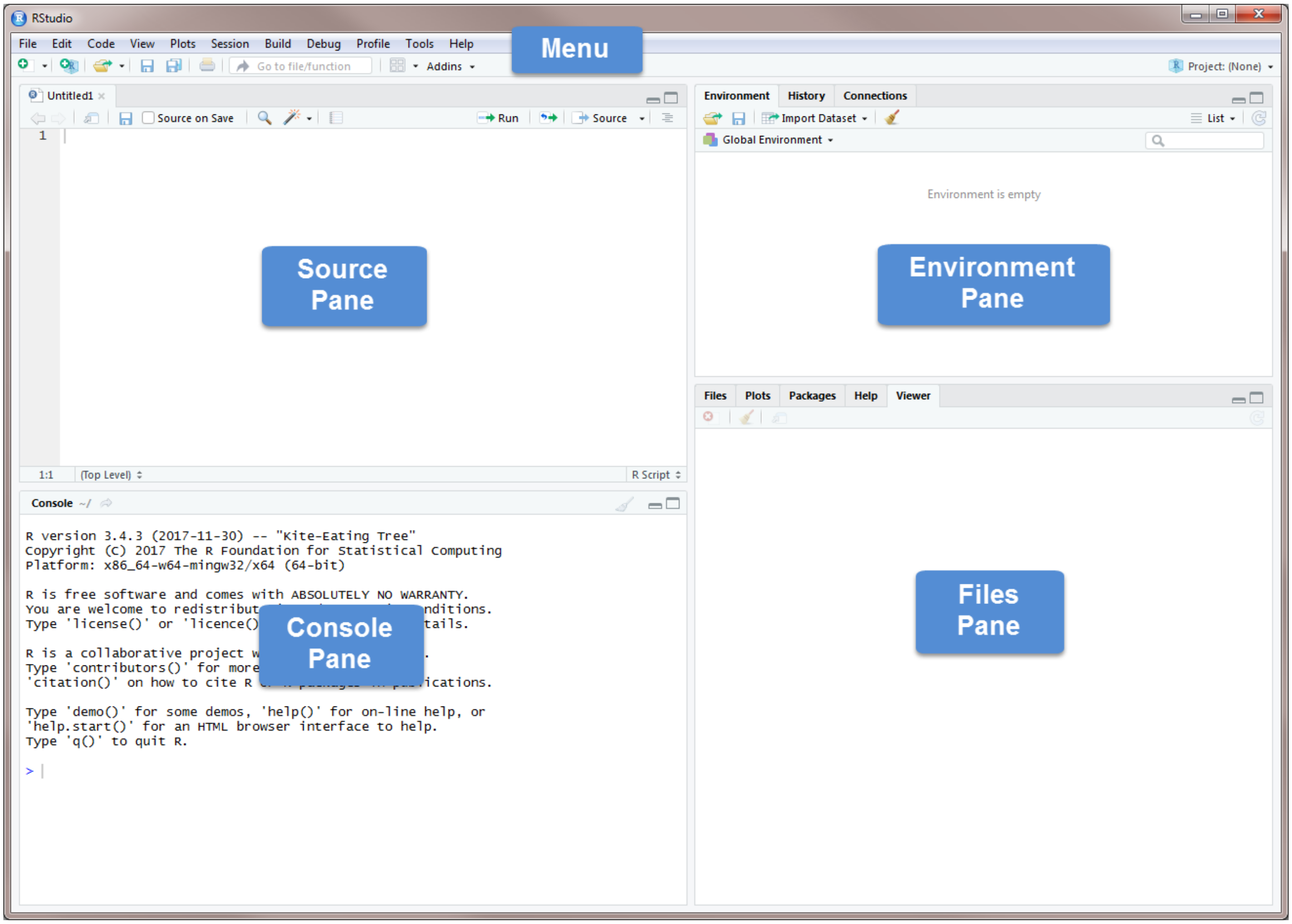
1.1. Accessing the tutorial data

This tutorial uses a file that is available in the S: drive. Create a folder in your H: drive called “IntroR”. Copy the files from **S:\Tutorials & Tip Sheets\Tufts\Tutorial Data\R and RStudio Basics** into that folder. You can also download the file from the link here: https://tufts.box.com/v/WorkshopIntroRStudio

2. Getting Started

2.1. Starting RStudio

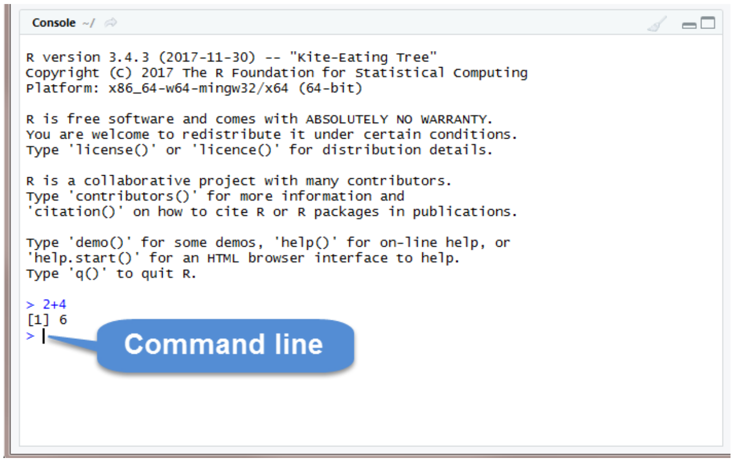
Start RStudio by going to **Start → All Programs → RStudio → RStudio** (note: This might be in a different location in Boston or on the Grafton Campuses. Additionally, on your home computer, RStudio may be under Programs). When you first open RStudio, you will see the Menu, the Console Pane, the Environment Pane, and the Files Pane. To open the Source Pane, click on ⊕ ▾ in the top left corner. From the dropdown menu, select R Script Ctrl+Shift+N. As shown in that dropdown menu, you can also open an R Script by pressing **Ctrl+Shift+N**. You should now see the following window:



```
R version 3.4.3 (2017-11-30) -- "Kite-Eating Tree"
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribut                    nditions.
Type 'license()' or 'licence()                    tails.

R is a collaborative project w                    .
Type 'contributors()' for more
'citation()' on how to cite R                     ications.

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.

> |
```

2.2. The Console Pane

The Console Pane is the interface to R. If you opened R directly instead of opening RStudio, you would see just this console. You can type commands directly in the console. The console displays the results of any command you run. For example, type 2+4 in the command line and press enter. You should see the command you typed, the result of the command, and a new command line.



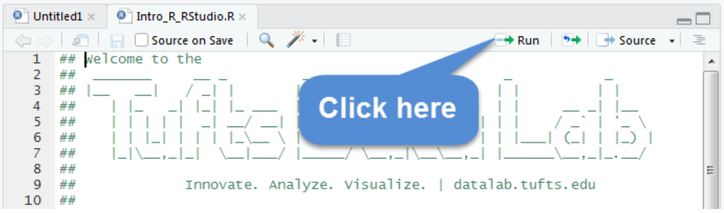
```
R version 3.4.3 (2017-11-30) -- "Kite-Eating Tree"
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

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.

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.

> 2+4
[1] 6
> |
```

To clear the console, you press **Ctrl+L** or type **cat("\014")** in the command line.

2.3. The Source Pane

The Source Pane is a text editor where you can type your code before running it. You can save your code in a text file called a script. Scripts have typically file names with the extension **.R**. To open a script, click on 📂 in the Menu bar or press **Ctrl+O**. Navigate to **H:\IntroR** and open the file called **Intro_to_R_RStudio.R.**

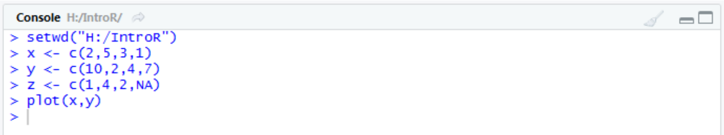
The first thing you should notice is the green text. Any text shown in green is a comment in the script. You write a comment by adding a **#** to an RScript. Anything to the right of a **#** is considered a comment and is thus ignored by R when running code. Place your cursor anywhere on the first few lines of code and click Run. You can also run code by pressing **Ctrl+Enter.**



```
## Welcome to the
##
##
##
##
##
##
##
##             Innovate. Analyze. Visualize. | datalab.tufts.edu
##
```

R will run the line where you placed your cursor. If it is a comment, it will ignore it and run the next line. R will ignore all the comments until it finds a line of code. In this script, the first line of code is in line 23. Your console will show only the code it just ran and not the comments. That first line of code, **setwd("H:/IntroR")**, sets the working directory. We will discuss the working directory in section 2.5.1.

Read the comments shown in the script and continue clicking run until you reach the end of the Example (line 35). Your console should look like this:

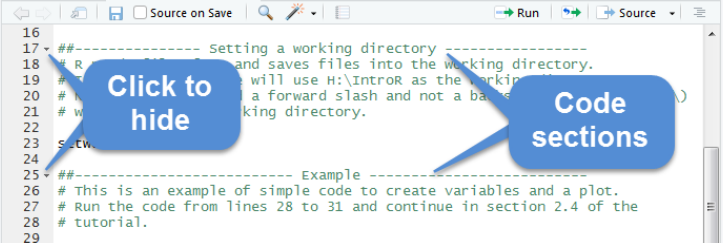


```
> setwd("H:/IntroR")
> x <- c(2,5,3,1)
> y <- c(10,2,4,7)
> z <- c(1,4,2,NA)
> plot(x,y)
> |
```

The Example in the script shows simple lines of code to create variables and a plot. We will discuss creating variables in sections 3 and 4. We will not discuss creating plots in this tutorial.

2.3.1. Code Sections

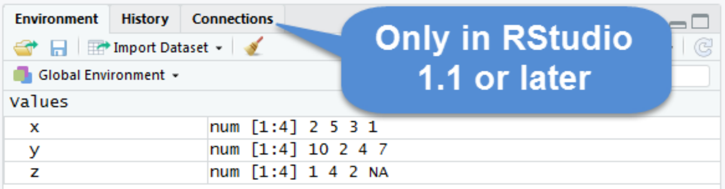
Code sections allow you to break a script into a set of discrete regions. To create a new code section, include at least four dashes, equal signs, or pound signs (-, =, or #) at the end of a comment. You can easily hide and show code sections by clicking in the arrow next to the code section line.



```
16
17 ##--------------- Setting a working directory -----------------
18 # R                and saves files into the working directory.
19 #                 will use H:\IntroR as the workin
20 #                 a forward slash and not a ba               \)
21 # w              king directory.
22
23 s tw
24
25 ##------------------------- Example ---------------------------
26 # This is an example of simple code to create variables and a plot.
27 # Run the code from lines 28 to 31 and continue in section 2.4 of the
28 # tutorial.
29
```

2.4. The Environment Pane

The Environment Pane includes an Environment and a History tab. If you are using RStudio 1.1 or a later version, you will also see a Connections tab. The Connections tab makes it easy to connect to any data source on your system. You will not see this tab on previous versions of RStudio.



```
Values
  x                  num [1:4] 2 5 3 1
  y                  num [1:4] 10 2 4 7
  z                  num [1:4] 1 4 2 NA
```

2.4.1. The Environment Tab

The Environment tab displays any objects that you have created during your R session. As part of the Example code section, we created three variables: x, y, and z. R stored those variables as objects, and you can see them in the Environment pane. We will discuss R objects in more detail in section 3. If you want to see a list of all objects in the current session, type **ls()** in the command line. You can remove an individual object from the environment with the **rm(…)** command. For example, remove **x** by typing **rm(x)** in the command line. You can remove all objects from the environment by clicking 🧹 or typing **rm(list=ls())** in the command line.

2.4.2. The History Tab

The History tab keeps a record of all the commands you have run. To copy a command from the history into the console, select the command and press **Enter** or click To Console. If you want to copy the command into the script, select the command and press **Shift+Enter** or click To Source. You can clear your history by clicking 🧹.

2.5. The Files Pane

The Files Pane includes several tabs that provide useful information.

2.5.1. The Files Tab

The Files tab displays the contents of your working directory. R reads files from and saves files into the working directory. You can find out which directory R is using by typing **getwd()** in the command line. For this tutorial, you should specify **H:\IntroR** as your working directory. To change the working directory, type **setwd("H:/IntroR")** in the command line. Notice that you need a forward slash and not a backslash (/ instead or \) when setting the working directory.

If you do not see the contents of **H:\IntroR** displayed in the Files tab, click on More ▾ and then on Go To Working Directory .

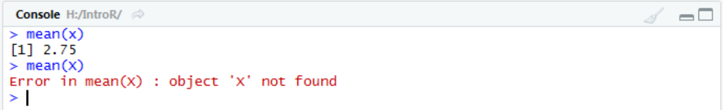
2.5.2. The Plots Tab

The Plot tab shows all graphs that you have created. If you have multiple plots, you can navigate through them by clicking ⇦ and ⇨ . As part of the Example code section, we created a plot that should be visible in the plots tab. Click Zoom to open the plot in another window. Click Export ▾ to export your plot as an image file or a pdf. To remove a single plot, click ✖ . To remove all plots, click 🧹 or type **dev.off()** in the command line.

2.5.3. The Packages Tab

An R package typically includes code, data, documentation for the package and functions inside, and tests to check everything works as it should. R packages make it easy to share your work with others. The variety of R packages is one of the reasons R is so powerful. As of February 2017, there were over 12,000 packages available in the official R repository (https://cloud.r-project.org/). Packages allow you to quickly perform tasks without having to write extensive code. For example, the **base** package, which loads automatically when you start RStudio, has a function that allows you to calculate a mean without having to type the formula for mean. We will calculate the mean of the variable **x**. Make sure the variable **x** is shown in the Environment tab. If it is not, run again the code in the script that defines **x** (line 30). Calculate the mean of **x** by typing **mean(x)** in the command line.

Important: R is case sensitive. If you type mean(X), you will get an error. X is not found because the variable is called x and not X.

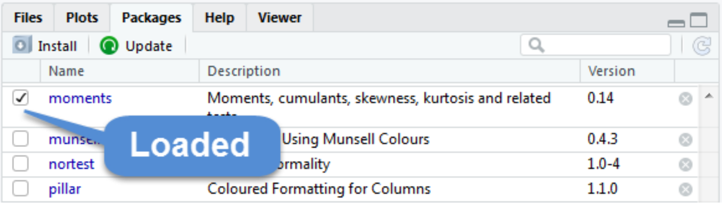


```
> mean(x)
[1] 2.75
> mean(X)
Error in mean(X) : object 'X' not found
> |
```

The Packages tab displays the R packages that you have installed in your System Library. Check to see if the package **moments** has been installed. If you cannot find it, you need to install it by using the command **install.packages("moments")**. Once you have installed the package, you need to load it using the command **library(moments).** You can type those commands in the command line or go to the script and run the Packages code section (lines 36 to 51).

Once a package is installed, you do not need to reinstall the package again unless you install a new version of R. If you want to use a package, you have to load it every time you start a new RStudio session.

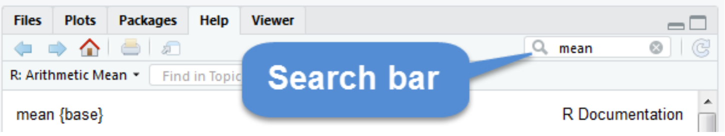
The Packages tab should now show the moments package. Packages that are loaded in the current R session have a check mark next to their name.



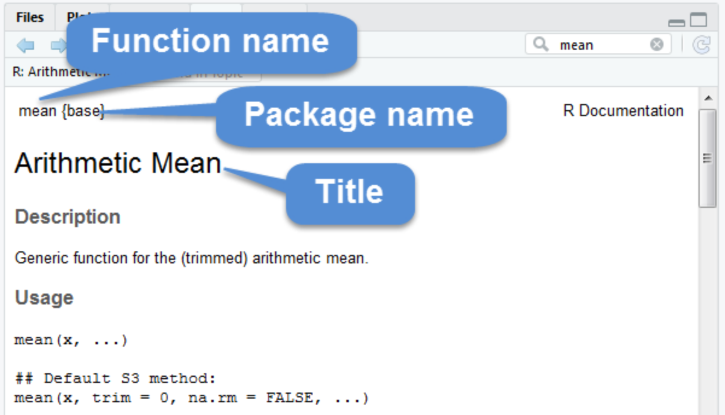
	Name	Description	Version
☑	moments	Moments, cumulants, skewness, kurtosis and related tests	0.14
□	munse	Using Munsell Colours	0.4.3
□	nortest	ormality	1.0-4
□	pillar	Coloured Formatting for Columns	1.1.0

2.5.4. The Help Tab

The help tab has built-in documentation for packages and functions in R. The help is automatically available for any loaded packages. We will access the help file for the **mean** function. You can access that file by typing **help(mean)** or **?mean** in the command line. You can also use the search bar in the help tab.



An R documentation file always has a header that includes information on the name of the function, the name of the package, a title, a description of the function, and usage information.



mean {base} R Documentation

Arithmetic Mean

Description

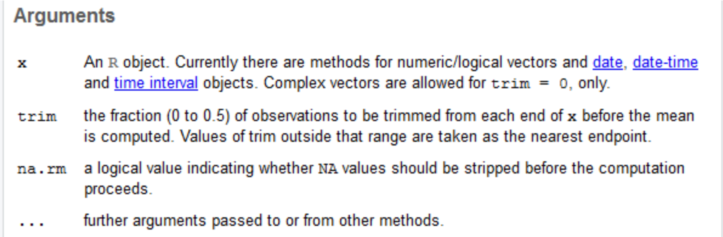
Generic function for the (trimmed) arithmetic mean.

Usage

```
mean(x, ...)

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

The usage section shows the function and the variables that need to be specified. In the case of the mean function, you must specify variable **x**. There are two other possible arguments listed for this function: **trim** and **na.rm**. Any value specified after an argument is its default value. For example, the default value for **na.rm** is **FALSE**.

The arguments section of the R documentation provides a description of the function's arguments.



Arguments

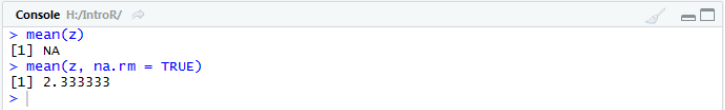
`x` An `R` object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects. Complex vectors are allowed for `trim = 0`, only.

`trim` the fraction (0 to 0.5) of observations to be trimmed from each end of `x` before the mean is computed. Values of trim outside that range are taken as the nearest endpoint.

`na.rm` a logical value indicating whether `NA` values should be stripped before the computation proceeds.

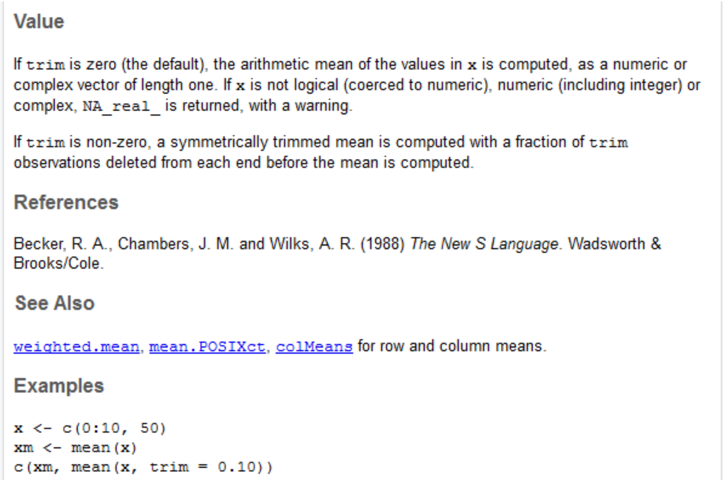
`...` further arguments passed to or from other methods.

Reading this section, we see that **na.rm** tells R if missing values, coded as NA, should be removed before calculating the mean. Let's test this argument. Calculate the mean of variable **z**. Make sure the variable **z** is shown in the Environment tab. If it is not, run again the code in the script that defines **z** (line 32). Calculate the mean of **z** by typing **mean(z)** in the command line. The result is not a number but an **NA**. R could not calculate the mean of the variable z because that variable has missing values. We need to tell R to first remove those missing values and then calculate the mean. Type **mean(z, na.rm = TRUE)** in the command line.



```
> mean(z)
[1] NA
> mean(z, na.rm = TRUE)
[1] 2.333333
>
```

The value section specifies what is returned by the function. R documentation may also include references, a list of similar functions, and examples.



Value

If `trim` is zero (the default), the arithmetic mean of the values in `x` is computed, as a numeric or complex vector of length one. If `x` is not logical (coerced to numeric), numeric (including integer) or complex, `NA_real_` is returned, with a warning.

If `trim` is non-zero, a symmetrically trimmed mean is computed with a fraction of `trim` observations deleted from each end before the mean is computed.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`weighted.mean`, `mean.POSIXct`, `colMeans` for row and column means.

Examples

```
x <- c(0:10, 50)
xm <- mean(x)
c(xm, mean(x, trim = 0.10))
```

Besides the R help, there are many resources online for seeking answers to R questions. Stack Overflow is a very useful site for help and discussion about programming, including R.

2.5.5. The Viewer Tab

The Viewer tab displays HTML output. R packages such as R Markdown and Shiny create HTML outputs that you can view in the Viewer tab.

2.6. The Menu

The Menu includes drop-down menus as well as buttons for opening, saving, and viewing or editing datasets. Here we list a few of the things you can do from each drop-down menu:

- File: Create new file, save data, open previously saved file.
- Edit: Search for text, clear console, undo/redo.
- Code: Insert code sections, run code.
- View: Move around tabs without clicking.
- Plots: Save Plots, Navigate through Plots.
- Session: Restart R session, set working directory, save session.
- Build/Debug/Profile: Advanced tools for programming
- Tools: Install packages, get information on version of RStudio
- Help: Help files, cheat sheets, and links to documentation.

3. R Objects

R stores information as objects. Objects can be variables, functions, or more general structures built from those components. You assign data to objects using the assign operator **<-**. In the example shown in the script (lines 25-35), we assigned some values to the variables **x**, **y**, and **z**. Those variables are stored as objects in R.

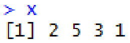
The assignment statement will usually have the form:

object_name <- expression

The object name can have letters, numbers, underscores, and dots. All the following names are valid in R: **var_name**, **VarName**, **var.name**, **var2**. Names can never start with a number, so **2var** is not a valid name. We suggest selecting one naming style and using it consistently. In this tutorial, we will use the underscore-separated naming style, e.g. var_name.

The expression is the information that will be stored in that object. For the variable **x**, for example, the information stored in that object are the numbers 2, 5, 3, and 1. Since we were assigning multiple numbers to one object, we used the combine **c()** function: **x <- c(2, 5, 3, 1)**.

To display the contents of an object, type the object name in the command line. For example, type **x** in the command line.



```
> x
[1] 2 5 3 1
```

4. Data Structures

In this section, we will discuss the ways that R stores and organizes data. R has five basic data structures: atomic vectors, matrices, arrays, lists, and data frames. These structures have specific requirements in terms of their dimension.

- One dimension: Atomic vectors and lists (section 4.1)
- Two dimensions: Matrices and data frames (section 4.2)
- N dimensions: Arrays (section 4.3)

Data structures also have specific requirements in terms of their contents. Homogeneous structures have contents that are all of the same type. Heterogeneous structures can have contents of different types.

- Homogeneous: Atomic vectors, matrices, and arrays
- Heterogeneous: Lists and data frames

Dimensions	Data Structure	
	Homogeneous	Heterogeneous
1	Atomic vector	List
2	Matrix	Data frame
n	Array	

4.1. Vectors

Vectors are the basic data structure in R. There are two types of vectors: atomic vectors and lists. They have three common properties: type, length, and attribute. Type, as the name implies, describes the type of vector. Length describes the number of elements the vector contains. Attributes provide additional information, or metadata. Each of those properties can be checked using the following commands: **typeof(), length(), attributes()**. We will provide examples for using those commands in the following subsections.

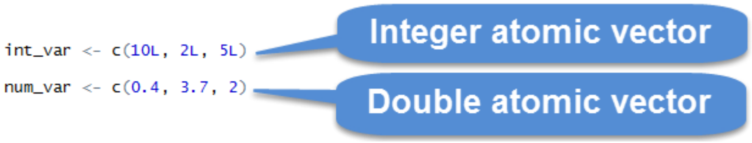
4.1.1. Atomic Vectors

Atomic vectors can only have elements of the same type. There are four common types of atomic vectors: integer, double (often called numeric), character, and logical.

Type	Example
Integer	2, 356
Double	3.0, 0.125
Character	“High”, “a”

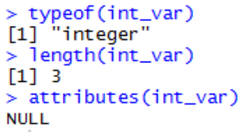
Logical	TRUE, FALSE

Go to the R script and run lines 53 to 68 to create atomic vectors of each type. Notice that to create an integer atomic vector, we had to include the letter L after each number. If you do not include an L after a number, R will consider that number to be of type double rather than integer.



```
int_var <- c(10L, 2L, 5L)
num_var <- c(0.4, 3.7, 2)
```

Run lines 69 to 78 in the script to get the type, length, and attributes of the integer atomic vector. You can also type **typeof(int_var)**, **length(int_var)**, or **attributes(int_var)** in the command line. Your console should show that the variable called **int_var** is an integer atomic vector with three elements and no attributes.



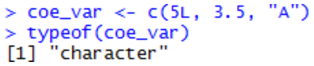
```
> typeof(int_var)
[1] "integer"
> length(int_var)
[1] 3
> attributes(int_var)
NULL
```

Now check the type, length, and attributes for the other three variables: **num_var**, **car_var**, and **log_var**. You will notice that all the variables have no special attributes. We will look at vectors with special attributes in section 4.1.3.

If your vector has missing values, the missing value is specified as **NA**. The variable **z** we created in line 32 of the script has a missing value. The type of vector in that case will be specified by the other elements in the vector. In the case of the **z** variable, all other elements were numbers, and thus the vector type is double. Confirm the vector type for z by typing **typeof(z)** in the console.

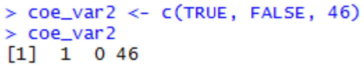
Remember that all elements of an atomic vector must be of the same type. When you try to combine elements of different types, R will coerce the vector to the most flexible type. Types from least to most flexible are: logical, integer, double, and character. Run lines 79 to 86 in the script for an example.

The atomic vector called **coe_var** has elements that are characters and numbers (both integer and double). The most flexible type is character, thus the vector type is character.



```
> coe_var <- c(5L, 3.5, "A")
> typeof(coe_var)
[1] "character"
```

The atomic vector called **coe_var2** has elements that are logical and double. The most flexible type is double, thus the vector type is double. Notice that when the logical values were coerced into double, **TRUE** became a **1** and **FALSE** a **0**. You can check by looking in the Environment pane or by typing **coe_var2** in the command line.



```
> coe_var2 <- c(TRUE, FALSE, 46)
> coe_var2
[1]  1  0 46
```

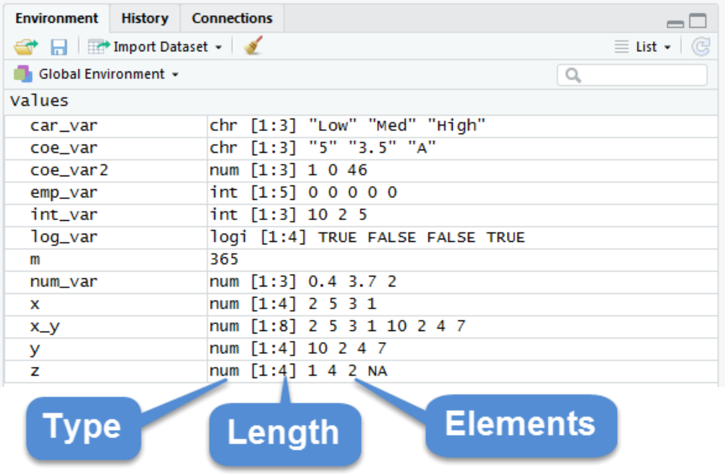
Creating Atomic Vectors

Individual elements in R are saved as atomic vectors of length 1. For example, **m <- 365** (line 88 in the script) assigns the value of 365 to the variable **m**. The variable **m** is a double type vector of length 1. You can check by running the **typeof(m)** and **length(m)** commands (lines 89 to 90 in the script).

You can create an empty vector using the **vector()** command. The command **vector("integer" , 5),** shown in line 93 in the script, creates an empty integer vector with 5 elements.

As mentioned in section 3, to create atomic vectors with more than one number, we need to use the combine function **c()**. We used that function to create all these variables in this section: **int_var**, **num_var**, **car_var**, **log_var, coe_var**, and **coe_var2**. You can also use the **c()** function to combine multiple vectors into one. See lines 95 to 98 in the script.

You can see all atomic vectors that you have created so far in the Environment pane under **Values**.

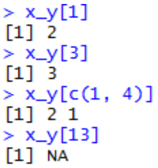


```
Values
  car_var          chr [1:3] "Low" "Med" "High"
  coe_var          chr [1:3] "5" "3.5" "A"
  coe_var2         num [1:3] 1 0 46
  emp_var          int [1:5] 0 0 0 0 0
  int_var          int [1:3] 10 2 5
  log_var          logi [1:4] TRUE FALSE FALSE TRUE
  m                365
  num_var          num [1:3] 0.4 3.7 2
  x                num [1:4] 2 5 3 1
  x_y              num [1:8] 2 5 3 1 10 2 4 7
  y                num [1:4] 10 2 4 7
  z                num [1:4] 1 4 2 NA
```

Accessing Elements of Atomic Vectors

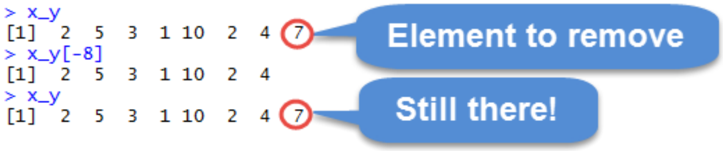
You can access elements of an atomic vector by declaring an index inside a square bracket **[]**. See lines 99 to 106 in the script for examples.

The variable **x_y** has 8 elements. Check using **length(x_y)**. To access the first element of that vector, use the command **x_y[1]**. To access the first and fourth elements of that vector, you also need to use the **c()** command: **x_y[c(1, 4)]**. If you try to access an element that does not exist, for example **x_y[13]**, R will return NA.



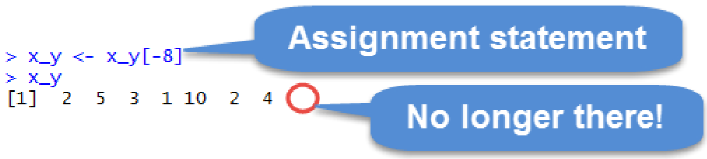
```
> x_y[1]
[1] 2
> x_y[3]
[1] 3
> x_y[c(1, 4)]
[1] 2 1
> x_y[13]
[1] NA
```

You can remove elements of an atomic vector by using a negative number. The code **x_y[-8]** removes the last element of that vector (lines 107 to 111 in the script). Remember that for R to save a variable you need to assign it. It is not enough to just run it.



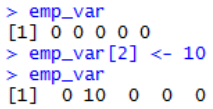
```
> x_y
[1]  2  5  3  1 10  2  4  7
> x_y[-8]
[1]  2  5  3  1 10  2  4
> x_y
[1]  2  5  3  1 10  2  4  7
```

If we want to remove the 8th element from variable x_y, we have to tell that to R. From section 3, recall that the assignment statement will usually have the form **object_name <- expression**. In the code shown above, we only have expressions, for example **x_y[-8]**. R will run those expressions and show the result in the console, but it will not store that information. We need to use an assignment statement for R to save information. The assignment statement tells R to remove the 8th element in the vector **x_y** and save that information as object **x_y**.



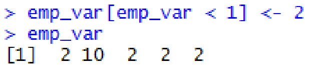
```
> x_y <- x_y[-8]
> x_y
[1]  2  5  3  1 10  2  4
```

You can replace specific elements using an assignment statement. For example, the command **emp_var[2] <- 10** replaces the second element of the variable **emp_var** with the number **10** (lines 112 to 115 in the script).



```
> emp_var
[1] 0 0 0 0 0
> emp_var[2] <- 10
> emp_var
[1]  0 10  0  0  0
```

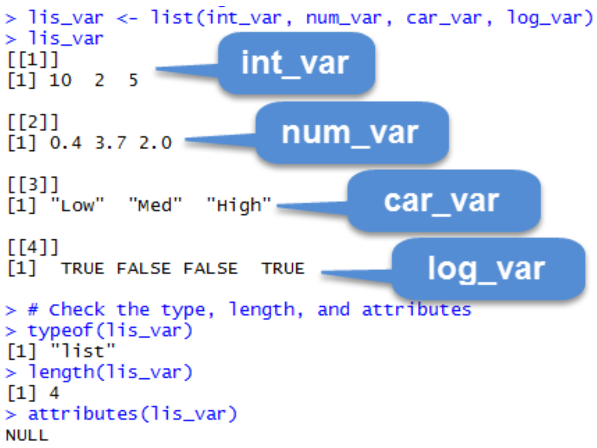
You can also use conditional statements to replace variables. The command **emp_var[emp_var < 1] <- 2** tells R to find all values in **emp_var** that are less than **1** and replace them with the number **2** (lines 116 to 119 in the script).



```
> emp_var[emp_var < 1] <- 2
> emp_var
[1]  2 10  2  2  2
```

4.1.2. Lists

Lists, unlike atomic vectors, can have elements of any type. Go to the R script and run lines 120 to 136 to create two lists and check their type, length, and attributes. The variable called **lis_var** is a list with 4 components and no attributes. The variable called **lis_xyz** is a list with 3 components and no attributes. Notice that **length()** now provides the number of components in a list. For atomic vectors, **length()** provides the number of elements in that atomic vector.



```
> lis_var <- list(int_var, num_var, car_var, log_var)
> lis_var
[[1]]
[1] 10  2  5

[[2]]
[1] 0.4 3.7 2.0

[[3]]
[1] "Low"  "Med"  "High"

[[4]]
[1]  TRUE FALSE FALSE  TRUE

> # Check the type, length, and attributes
> typeof(lis_var)
[1] "list"
> length(lis_var)
[1] 4
> attributes(lis_var)
NULL
```

Creating Lists

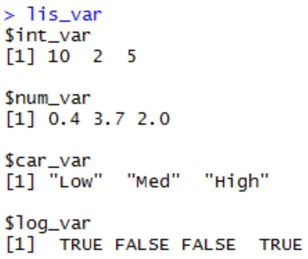
You create lists using the **list()** function. We used that function to create two lists in this section: **lis_var** and **lis_xyz**.

We can assign a name to each component of a list. This is good practice so that you do not forget what each component represents. Run lines 137 to 140 in the script to assign names to the components of **lis_var** and **lis_xyz**. The command to assign names to list components has the form **names(list_name) <- c(component_names).**

For example, the variable called **lis_var** is a list with atomic vectors **int_var**, **num_var**, **car_var**, **log_var**. We assigned the corresponding atomic vector name to each component of **lis_var** using the command:

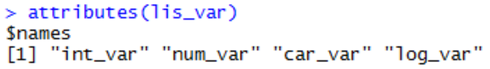
names(lis_var) <- c("int_var", "num_var", "car_var", "log_var").

Type **lis_var** in the command line. You will see the names you assigned to the components of the list.



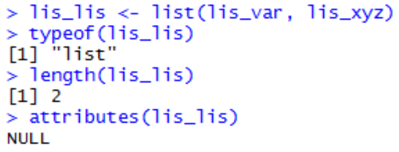
```
> lis_var
$int_var
[1] 10  2  5

$num_var
[1] 0.4 3.7 2.0

$car_var
[1] "Low"  "Med"  "High"

$log_var
[1]  TRUE FALSE FALSE  TRUE
```

Now check the attributes for the list **lis_var**. That list now has an attribute called **names**. R stores additional information about an object as **attributes**.



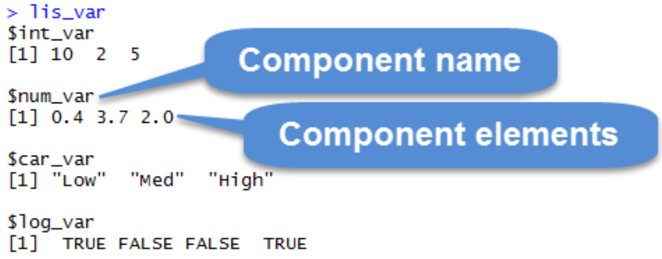
```
> attributes(lis_var)
$names
[1] "int_var" "num_var" "car_var" "log_var"
```

A list allows you to group objects, any object, in an ordered way. For example, you can create a list of lists. Create a list of the two lists, **lis_var** and **lis_xyz**, and check type, length, and attributes of that new list using lines 141 to 148 in the script. The variable **lis_lis** is a list with 2 components and no attributes. This list has no attributes, even if the individual lists, **lis_var** and **lis_xyz**, do have the names attribute.



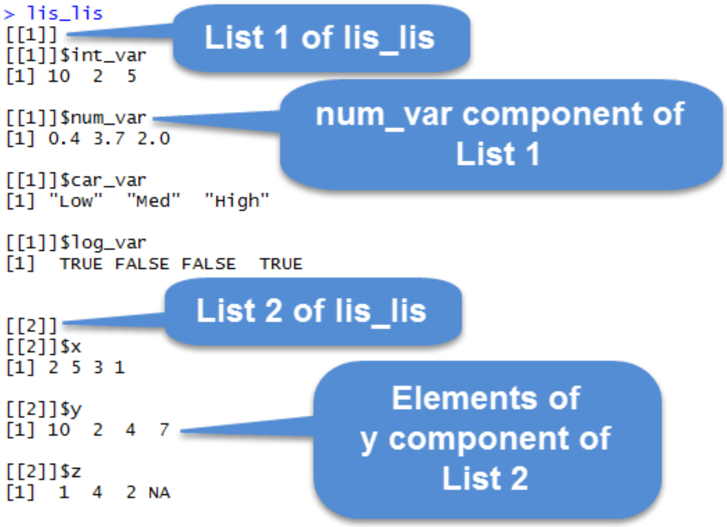
```
> lis_lis <- list(lis_var, lis_xyz)
> typeof(lis_lis)
[1] "list"
> length(lis_lis)
[1] 2
> attributes(lis_lis)
NULL
```

When you print a list that has only atomic vectors, such as **lis_var,** you see the elements of each component in that list.



```
> lis_var
$int_var
[1] 10  2  5

$num_var
[1] 0.4 3.7 2.0

$car_var
[1] "Low"  "Med"  "High"

$log_var
[1]  TRUE FALSE FALSE  TRUE
```

When you print a list that has other lists, such as **lis_lis,** you first see the components of that list. In the case of **lis_lis**, the components are two lists. You can check this using **length(lis_lis)**. You can also see the components of each of the two lists. List 1 of **lis_lis** has 4 components, and list 2 has 3 components. Notice that R kept the names we assigned to the components of each list.



```
> lis_lis
[[1]]
[[1]]$int_var
[1] 10  2  5

[[1]]$num_var
[1] 0.4 3.7 2.0

[[1]]$car_var
[1] "Low"  "Med"  "High"

[[1]]$log_var
[1]  TRUE FALSE FALSE  TRUE

[[2]]
[[2]]$x
[1] 2 5 3 1

[[2]]$y
[1] 10  2  4  7

[[2]]$z
[1]  1  4  2 NA
```

List 1 of lis_lis
num_var component of List 1
List 2 of lis_lis
Elements of y component of List 2

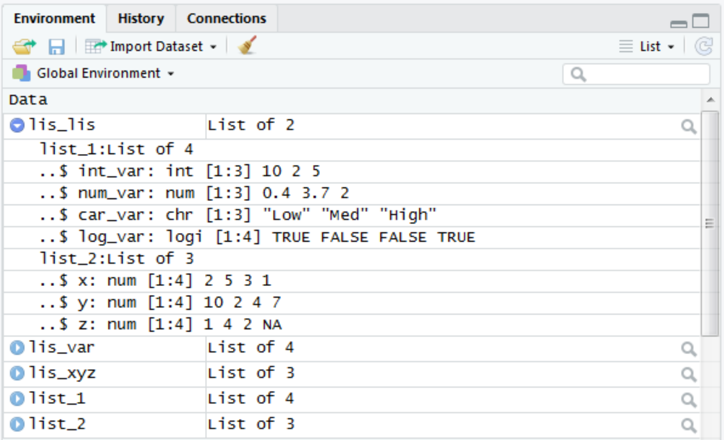
You can extract all components from a *named* list and saved them as individual objects in R using the function **list2env()**. To use this function, we first need to add names to the **lis_lis** components. Run lines 149 to 152 in the script to add names to the components of **lis_lis** and then create individual objects from that list.

Notice the structure of the command we used to save objects:

list2env(lis_lis, .GlobalEnv)

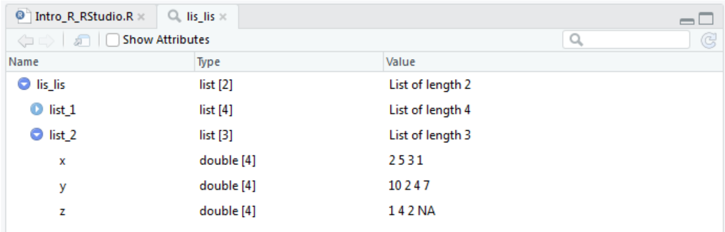
We had to specify the name of the list, **lis_lis**, and where to save the objects, **.GlobalEnv**. The global environment **.GlobalEnv**, is the user's workspace. You can see all objects we have created in the global environment under the Environment pane. You can see that we created two lists using the list2env function, **list_1** and **list_2**. All lists that you have created so far are shown in the environment pane under **Data**.

If you click on the blue arrow next to a list name, you can see the components of that list.



```
Data
lis_lis          List of 2
  list_1:List of 4
  ..$ int_var: int [1:3] 10 2 5
  ..$ num_var: num [1:3] 0.4 3.7 2
  ..$ car_var: chr [1:3] "Low" "Med" "High"
  ..$ log_var: logi [1:4] TRUE FALSE FALSE TRUE
  list_2:List of 3
  ..$ x: num [1:4] 2 5 3 1
  ..$ y: num [1:4] 10 2 4 7
  ..$ z: num [1:4] 1 4 2 NA
lis_var          List of 4
lis_xyz          List of 3
list_1           List of 4
list_2           List of 3
```

If you click on the name of the list, a window will open in the Source pane that will show you the contents of that list. That window also has blue arrows that you can click to see the contents of a list.

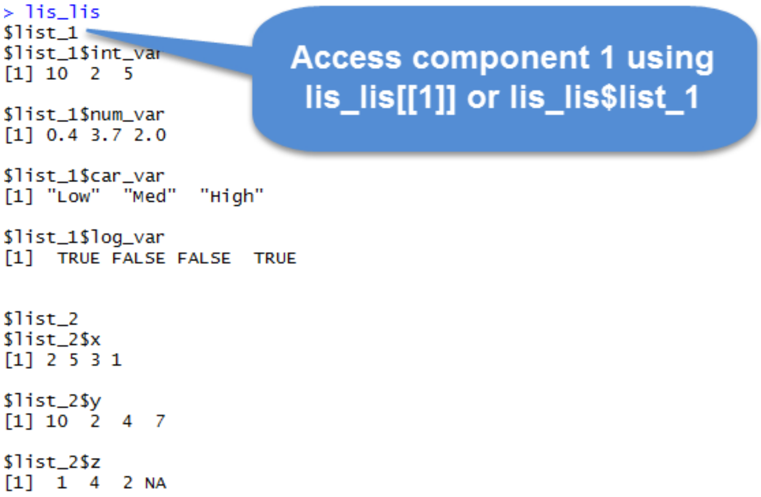


Name	Type	Value
lis_lis	list [2]	List of length 2
list_1	list [4]	List of length 4
list_2	list [3]	List of length 3
x	double [4]	2 5 3 1
y	double [4]	10 2 4 7
z	double [4]	1 4 2 NA

Accessing Elements of Lists

A list can have multiple components and elements, just like **lis_lis**. You can access a component of a list using the name or the numbered position of that component. To access a component by its numbered position, we need to declare an index inside a double square bracket **[[]]**.To access a component by its name, we need to use the **$** sign. For programing, you may want to access elements of a list using a single square braket **[]**. We will not cover programing in this tutorial. If you want to learn more, see Advanced R - Subsetting.

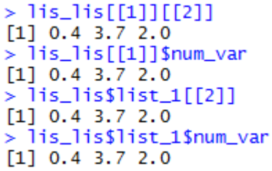
To access component 1 of **lis_lis** by its numbered position, we use the following command: **lis_lis[[1]].** To access component 1 of **lis_lis** by its name, we use the following command: **lis_lis$list_1.** See script lines 153 to 159.



```
> lis_lis
$list_1
$list_1$int_var
[1] 10  2  5

$list_1$num_var
[1] 0.4 3.7 2.0

$list_1$car_var
[1] "Low"  "Med"  "High"

$list_1$log_var
[1]  TRUE FALSE FALSE  TRUE


$list_2
$list_2$x
[1] 2 5 3 1

$list_2$y
[1] 10  2  4  7

$list_2$z
[1]  1  4  2 NA
```

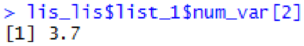
Access component 1 using lis_lis[[1]] or lis_lis$list_1

You can use the same logic to access components of specific lists within a list. All the commands shown below, give you component 2 of **list_1** (lines 160 to 165 in the script).



```
> lis_lis[[1]][[2]]
[1] 0.4 3.7 2.0
> lis_lis[[1]]$num_var
[1] 0.4 3.7 2.0
> lis_lis$list_1[[2]]
[1] 0.4 3.7 2.0
> lis_lis$list_1$num_var
[1] 0.4 3.7 2.0
```

To access specific elements of a vector within a list, you need to use a single square braket **[]**. Run line 167 in the script to access the second element of the **num_var** component of **list_1**.

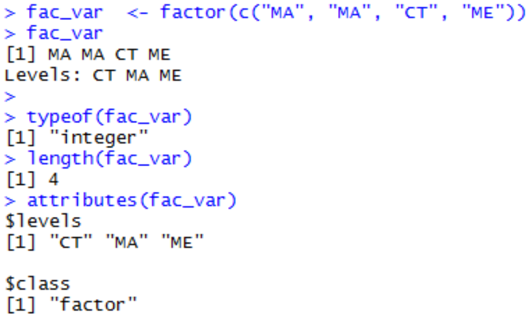


```
> lis_lis$list_1$num_var[2]
[1] 3.7
```

4.1.3. Factors

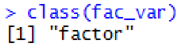
A factor is a vector used to store categorical variables. It is important to tell R that a variable is categorical, rather than character or numeric.

A common categorical variable is state (e.g., Massachusetts). Run lines 169 to 180 in the script to create variable **fac_var** and check its type, length and attributes. The variable **fac_var** is an integer vector with four elements and two attributes: class = factor and levels = CT, MA, and ME.



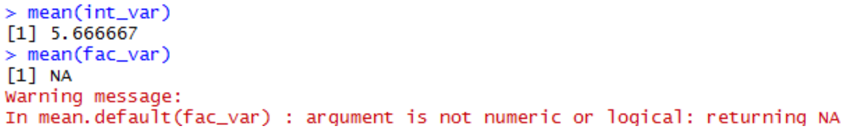
```
> fac_var  <- factor(c("MA", "MA", "CT", "ME"))
> fac_var
[1] MA MA CT ME
Levels: CT MA ME
>
> typeof(fac_var)
[1] "integer"
> length(fac_var)
[1] 4
> attributes(fac_var)
$levels
[1] "CT" "MA" "ME"

$class
[1] "factor"
```

A factor in R is an integer atomic vector that has two attributes, class and levels. You can check the class attribute using the **class()** function (lines 181 to 183) .



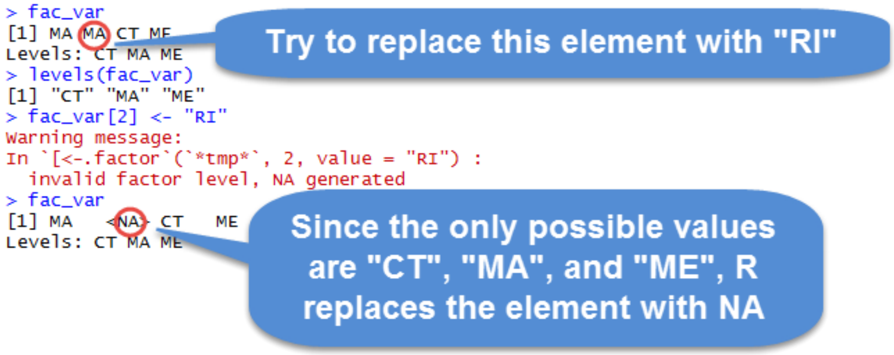
```
> class(fac_var)
[1] "factor"
```

Class specifies that the variable is a factor and it should be treated as such and not as an integer atomic vector. For example, you can calculate the mean of the elements of an integer atomic vector but not of the elements of a factor. Try this using lines 184 to 190 in the script. The **fac_var** contains state names, and you cannot calculate the mean of the state names.



```
> mean(int_var)
[1] 5.666667
> mean(fac_var)
[1] NA
Warning message:
In mean.default(fac_var) : argument is not numeric or logical: returning NA
```

The levels attribute tells R the possible values that can exist in that factor. Using the **levels()** function, you can see that that **fac_var** has three possible values: CT, MA, and ME. All elements in this factor can only have those values. Try to replace the second element of **fac_var** with RI, for Road Island (lines 191 to 198 in the script). You access elements of a factor just as you did for atomic vectors, using a single square bracket. Since RI is not a possible value for **fac_var**, R gives you a warning, and replaces the specified element with **NA**.

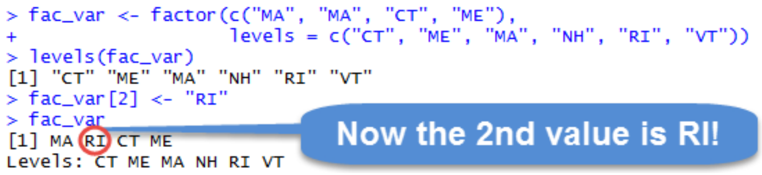


```
> fac_var
[1] MA MA CT ME
Levels: CT MA ME
> levels(fac_var)
[1] "CT" "MA" "ME"
> fac_var[2] <- "RI"
Warning message:
In `[<-.factor`(`*tmp*`, 2, value = "RI") :
  invalid factor level, NA generated
> fac_var
[1] MA   <NA> CT   ME
Levels: CT MA ME
```

Try to replace this element with "RI"

Since the only possible values are "CT", "MA", and "ME", R replaces the element with NA

You can specify additional levels when you create the factor. Create **fac_var** again, but this time specify that the variable can have the names of any New England state. Now you can replace the second element of **fac_var** with RI (see lines 199 to 207 in the script).

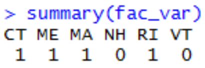


```
> fac_var <- factor(c("MA", "MA", "CT", "ME"),
+                   levels = c("CT", "ME", "MA", "NH", "RI", "VT"))
> levels(fac_var)
[1] "CT" "ME" "MA" "NH" "RI" "VT"
> fac_var[2] <- "RI"
> fac_var
[1] MA RI CT ME
Levels: CT ME MA NH RI VT
```

Now the 2nd value is RI!

The variable **fac_var** now has six possible values rather than three. Notice the order. R kept the order that we provided in the statement **levels = c("CT", "ME", "MA", "NH", "RI", "VT")**. Changing levels does not change any other information in that factor. Confirm this by checking type, length, and attributes.

The function **summary()** in R gives you a quick overview of the contents of a variable. For a factor, it will give you the number of responses in each category. Type **summary(fac_var)** in the command line or run line 209 in the script.



```
> summary(fac_var)
CT ME MA NH RI VT 
 1  1  1  0  1  0 
```

4.2. Matrices and Arrays

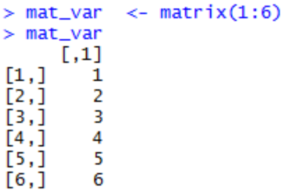
Matrices and arrays are homogeneous data structures, just like atomic vectors, that can have more than one dimension, unlike vectors.

Dimensions	Data Structure Homogeneous
1	Atomic vector
2	Matrix
n	Array

R stores matrices and arrays in a similar manner as vectors, but with the attribute called dimension. A matrix is an array that has two dimensions. Data in a matrix are organized into rows and columns. Matrices are commonly used while arrays are rare. We will focus on matrices for the rest of this section. If you want to learn more about arrays, see Advanced R - Data structures.

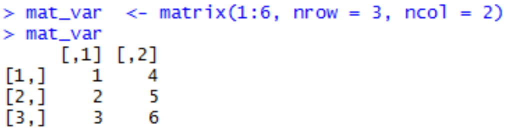
Creating Matrices

You create a matrix using the **matrix()** function. Run lines 211 to 218 in the script to create a matrix. If you do not specify a number of rows and columns, R will create a matrix with one column and multiple rows. The variable **mat_var** is a matrix of 1 column and 6 rows.



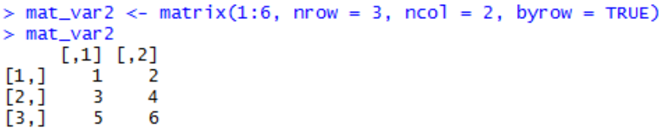
```
> mat_var  <- matrix(1:6)
> mat_var
     [,1]
[1,]    1
[2,]    2
[3,]    3
[4,]    4
[5,]    5
[6,]    6
```

If you want that matrix to have a different number of columns and rows, you have to use the **nrow** and **ncol** attributes. Run lines 219 to 222 in the script. Now the variable **mat_var** has 3 rows and 2 columns.



```
> mat_var  <- matrix(1:6, nrow = 3, ncol = 2)
> mat_var
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

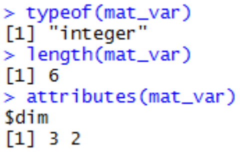
Matrices are constructed column wise. If you want to construct a matrix row wise, set the argument **byrow = TRUE** (lines 223 to 227 in the script).



```
> mat_var2 <- matrix(1:6, nrow = 3, ncol = 2, byrow = TRUE)
> mat_var2
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
```

Variables **mat_var** and **mat_var2** have 2 columns and 3 rows, and elements that are a sequence of numbers from 1 to 6. In **mat_var**, the numbers 1, 2, and 3 are in the first column and 4, 5, and 6 in the second column. In **mat_var2**, the numbers 1 and 2 are in the first row, 3 and 4 in the second row, and 5 and 6 in the third row. The difference arises from variable **mat_var2** being built row wise rather than column wise.

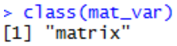
Check the type, length and attributes of variables **mat_var** and **mat_var2**. Since matrices are stored in R as vectors with the attribute dimension, **typeof()** and **length()** will give you the data type of the elements and the total number of elements. Lines 228 to 231 in the script.



```
> typeof(mat_var)
[1] "integer"
> length(mat_var)
[1] 6
> attributes(mat_var)
$dim
[1] 3 2
```

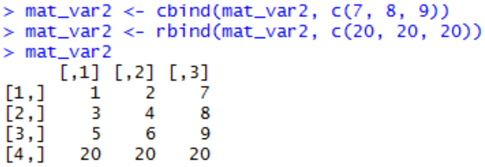
You can see that both variables have six elements that are integers. The attribute dim shows that these variables have 3 rows and 2 columns.

You can confirm that the variable is a matrix using the **class()** function (lines 232 to 234 in the script).



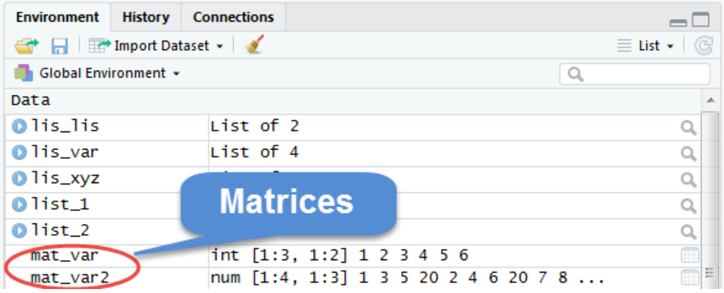
```
> class(mat_var)
[1] "matrix"
```

You can add rows or columns to a matrix using the **rbind()** and **cbind()** functions. Run lines 235 to 239 in the script to add a row and a column to variable **mat_var2**. Remember that for R to save new information, you need to assign it to the variable.



```
> mat_var2 <- cbind(mat_var2, c(7, 8, 9))
> mat_var2 <- rbind(mat_var2, c(20, 20, 20))
> mat_var2
     [,1] [,2] [,3]
[1,]    1    2    7
[2,]    3    4    8
[3,]    5    6    9
[4,]   20   20   20
```

You can see all matrices that you have created so far in the Environment pane under **Data**.

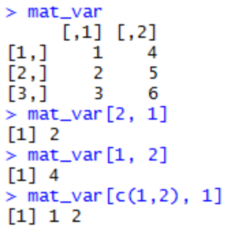


```
Data
lis_lis     List of 2
lis_var     List of 4
lis_xyz
list_1
list_2
mat_var     int [1:3, 1:2] 1 2 3 4 5 6
mat_var2    num [1:4, 1:3] 1 3 5 20 2 4 6 20 7 8 ...
```

Accessing Elements of Matrices

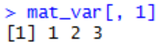
You can access elements of a matrix by declaring a row and a column index inside a square bracket **[row_index, column_index]**. See lines 240 to 245 in the script for examples.

Recall that the variable **mat_var** has six elements in 3 rows and 2 columns. To access the element in the second row and the first column of that matrix, use the command **mat_var[2, 1]**. To access the element in the first row and the second column of that matrix, use the command **mat_var[1, 2]**. To access multiple elements, you need to use the **c()** function.



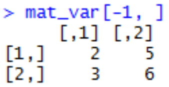
```
> mat_var
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> mat_var[2, 1]
[1] 2
> mat_var[1, 2]
[1] 4
> mat_var[c(1,2), 1]
[1] 1 2
```

If a field inside the bracket is empty, R will select all those elements. For example, **mat_var[, 1]** selects all elements in the first column of the matrix (lines 246 to 248 in the script).



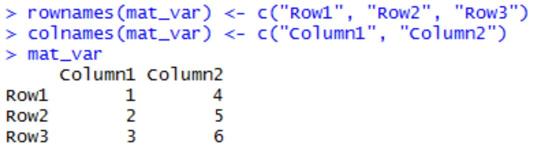
```
> mat_var[, 1]
[1] 1 2 3
```

You can use negative numbers to specify rows or columns to be excluded. For example, **mat_var[-1,]** selects all rows of the matrix except the first one (lines 249 to 251 in the script).



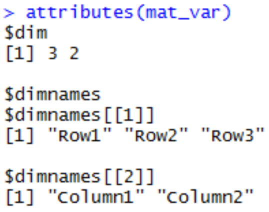
```
> mat_var[-1, ]
     [,1] [,2]
[1,]    2    5
[2,]    3    6
```

You can also select elements of a matrix by the names of the columns and rows. To assign names to columns and rows of a matrix, use the **rownames()** and **colnames()** functions. Run lines 252 to 257 to assign names to the rows and columns of matrix **mat_var**.



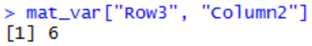
```
> rownames(mat_var) <- c("Row1", "Row2", "Row3")
> colnames(mat_var) <- c("Column1", "Column2")
> mat_var
     Column1 Column2
Row1       1       4
Row2       2       5
Row3       3       6
```

R stores those names as attributes. Check by typing **attributes(mat_var)** in the command line.



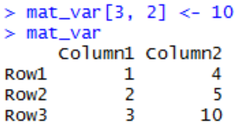
```
> attributes(mat_var)
$dim
[1] 3 2

$dimnames
$dimnames[[1]]
[1] "Row1" "Row2" "Row3"

$dimnames[[2]]
[1] "Column1" "Column2"
```

You can now select elements of the matrix by name. For example, **mat_var["Row3", "Column2"]** selects the element in Row3 and Column2 of the matrix (lines 258 to 261 in the script).



```
> mat_var["Row3", "Column2"]
[1] 6
```

Just like with vectors, you can replace specific elements of a matrix with an assignment statement. For example, the command **mat_var [3, 2] <- 10** replaces the element in row 3 and column 2 of the variable **mat_var** with the number **10** (lines 262 to 266 in the script).



```
> mat_var[3, 2] <- 10
> mat_var
     Column1 Column2
Row1       1       4
Row2       2       5
Row3       3      10
```

4.3. Data Frames

A data frame is the most common way of storing data in R. A data frame shares properties with both lists and matrices. Like a matrix, a data frame has 2 dimensions and data are organized into rows and columns. Each column represents a variable and each row represents an observation. You can also think of each column as a vector. All columns must have the same length, but they can be of different data types. A data frame is therefore a list of equal-length vectors and can thus be heterogeneous.

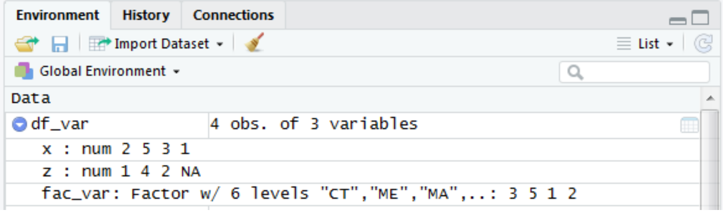
Dimensions	Data Structure	
	Homogeneous	Heterogeneous
1	Atomic vector	List
2	Matrix	Data frame

Creating Data Frames

You create a data frame using the **data.frame()** function. Run lines 267 to 272 in the script to create a data frame that includes the atomic vectors **x** and **z,** and the factor **fac_var**.

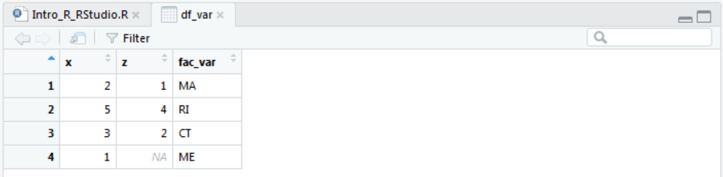
```
> df_var <- data.frame(x, z, fac_var)
> df_var
  x  z fac_var
1 2  1      MA
2 5  4      RI
3 3  2      CT
4 1 NA      ME
```

Each vector is a column of the data frame. Every column of a data frame must have the same length, so we used three vectors of the same length (4 elements). Notice that not all vectors included are of the same type, **x** and **z** are double vectors while **fac_var** is a factor. The data frame maintains the data structure of each column. You can check the structure of each column by going to the environment pane and finding the variable under **Data**. If you click on the blue arrow next to a data frame, you can see the columns of that data frame. Notice that the **fac_var** column is still saved as a factor and the x and z variables are numeric (or double).



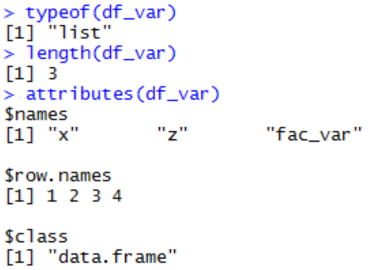
```
Data
df_var          4 obs. of 3 variables
  x : num 2 5 3 1
  z : num 1 4 2 NA
  fac_var: Factor w/ 6 levels "CT","ME","MA",..: 3 5 1 2
```

If you click on the name of the data frame, a window will open in the Source pane that will show you the contents of that data frame.



	x	z	fac_var
1	2	1	MA
2	5	4	RI
3	3	2	CT
4	1	NA	ME

Check the type, length, and attributes of variable **df_var** (lines 273 to 276 in the script). The **df_var** variable is stored as a list of 3 components (or columns), and it has three attributes: names, row.names, and class.

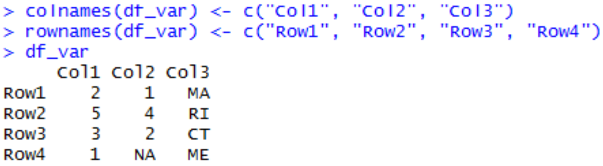


```
> typeof(df_var)
[1] "list"
> length(df_var)
[1] 3
> attributes(df_var)
$names
[1] "x"       "z"       "fac_var"

$row.names
[1] 1 2 3 4

$class
[1] "data.frame"
```

You can see that R stores data frames as lists. Just like with lists, length does not provide the number of elements in the variable, but the number of components, or columns, in the data frame. In this case, **df_var** has 3 columns. You can also check the number of columns with the **ncols()** function, and the number of rows with the **nrows()** function (lines 277 to 280 in the script).

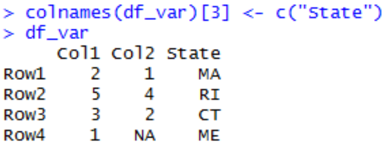
The attribute **class** specifies that the variable is a data frame and it should be treated as such and not as a list. You can confirm that the variable is a matrix using the **class()** function (lines 281 to 283 in the script).

The attribute **names** includes the column names, and the **row.names**, as the name implies, includes the row names. The default row names are the numbers of each row. You can change column and row names using the functions **colnames()** and **rownames()** just as you did with matrices. Run lines 284 to 289 to change the column names and row names of **df_var**.



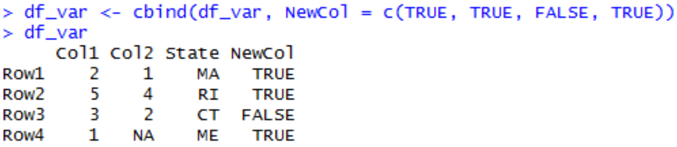
```
> colnames(df_var) <- c("Col1", "Col2", "Col3")
> rownames(df_var) <- c("Row1", "Row2", "Row3", "Row4")
> df_var
     Col1 Col2 Col3
Row1    2    1   MA
Row2    5    4   RI
Row3    3    2   CT
Row4    1   NA   ME
```

To change the name of column 3 to "State", access element 3 of the colnames vector using **colnames(df_var)[3],** and then assign to that element the new value (lines 290 to 293 in the script).



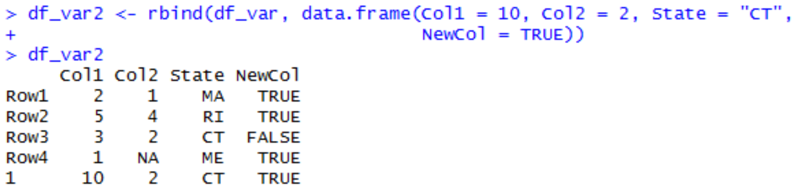
```
> colnames(df_var)[3] <- c("State")
> df_var
     Col1 Col2 State
Row1    2    1    MA
Row2    5    4    RI
Row3    3    2    CT
Row4    1   NA    ME
```

Just like with matrices, you can add rows or columns to a data frame using the **rbind()** and **cbind()** functions. When adding data column wise, the number of rows must match, and you must specify the name of the new column. Run lines 294 to 297 in the script.



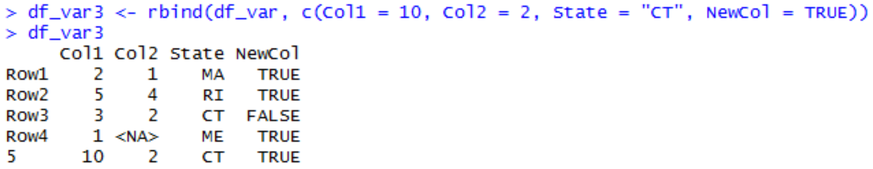
```
> df_var <- cbind(df_var, NewCol = c(TRUE, TRUE, FALSE, TRUE))
> df_var
     Col1 Col2 State NewCol
Row1    2    1    MA   TRUE
Row2    5    4    RI   TRUE
Row3    3    2    CT  FALSE
Row4    1   NA    ME   TRUE
```

When adding data row wise, the number and the names of columns must match. Run lines 298 to 303 in the script.



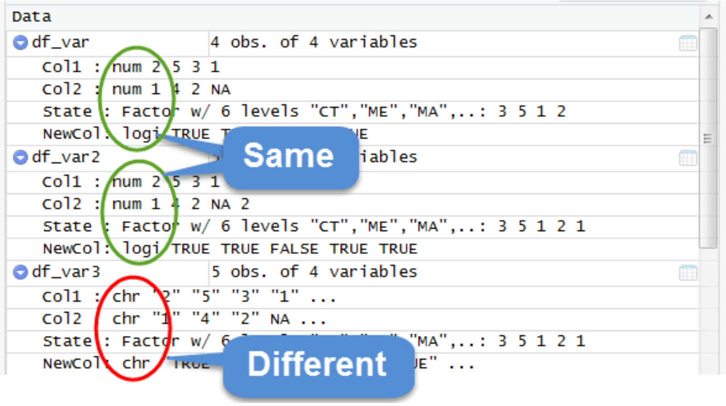
```
> df_var2 <- rbind(df_var, data.frame(Col1 = 10, Col2 = 2, State = "CT",
+                                      NewCol = TRUE))
> df_var2
     Col1 Col2 State NewCol
Row1    2    1    MA   TRUE
Row2    5    4    RI   TRUE
Row3    3    2    CT  FALSE
Row4    1   NA    ME   TRUE
1      10    2    CT   TRUE
```

We used the **data.frame()** command rather than the **c()** command to specify the elements of the new row. Remember that the **c()** command creates an atomic vector, and all elements of that vector must be of the same type. If you use the **c()** command to bind a row to a data frame, R will coerce the data to the most flexible type (see Section 4.1.1). Run lines 304 to 308 in the script.



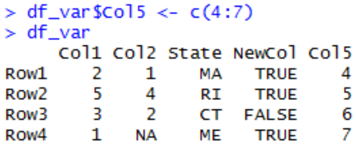
```
> df_var3 <- rbind(df_var, c(Col1 = 10, Col2 = 2, State = "CT", NewCol = TRUE))
> df_var3
     Col1 Col2 State NewCol
Row1    2    1    MA   TRUE
Row2    5    4    RI   TRUE
Row3    3    2    CT  FALSE
Row4    1 <NA>    ME   TRUE
5      10    2    CT   TRUE
```

Notice that **df_var2** and **df_var3** are almost the same. Both were created by adding a row to **df_var**. The difference is that **df_var2** maintained the original data type, while **df_var3** did not. Look in the Environment Pane:



```
Data
df_var            4 obs. of 4 variables
  Col1 : num 2 5 3 1
  Col2 : num 1 4 2 NA
  State : Factor w/ 6 levels "CT","ME","MA",..: 3 5 1 2
  NewCol: logi TRUE T                E
df_var2                  iables
  Col1 : num 2 5 3 1
  Col2 : num 1 4 2 NA 2
  State : Factor w/ 6 levels "CT","ME","MA",..: 3 5 1 2 1
  NewCol: logi TRUE TRUE FALSE TRUE TRUE
df_var3           5 obs. of 4 variables
  Col1 : chr "2" "5" "3" "1" ...
  Col2   chr "1" "4" "2" NA ...
  State : Factor w/ 6              "MA",..: 3 5 1 2 1
  NewCol: chr                      E" ...
```

Since a data frame also shares properties with a list, you can also add columns to a data frame using the **$** sign. See lines 309 to 312 in the script.



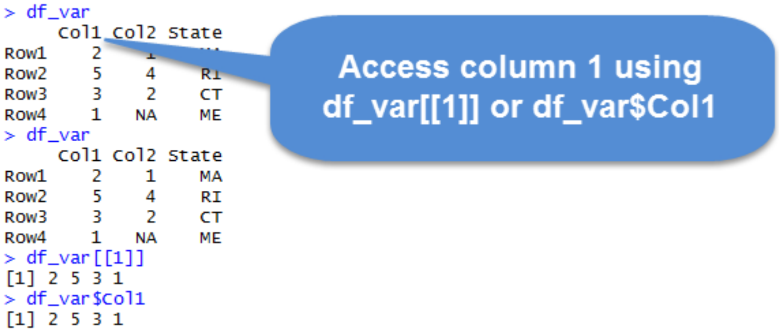
```
> df_var$Col5 <- c(4:7)
> df_var
     Col1 Col2 State NewCol Col5
Row1    2    1    MA   TRUE    4
Row2    5    4    RI   TRUE    5
Row3    3    2    CT  FALSE    6
Row4    1   NA    ME   TRUE    7
```

If you want to combine two data frames that have an unequal number of rows or columns, you can use functions from the **dplyr** package. We will not cover the **dplyr** package in this tutorial. If you want more information about this package, see R for Data Science - Data transformation.

Accessing Elements of Data Frames

Since a data frame is a list of equal-length vectors, you can access those vectors using the same commands you used for lists. Remember that the vectors form the columns of the data frame. Just like a list, declare an index inside a double square bracket **[[]]** to access a column using its numbered position, and a use a **$** sign to access a column by its name. For programing, you may want to access columns of a data frame using an index inside a single square braket **[]**. We will not cover programing in this tutorial. If you want to learn more, see Advanced R - Subsetting.

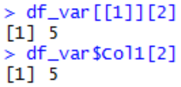
To access the first column of **df_var** by its numbered position, we use the following command: **df_var[[1]].** To access the first column of **df_var** by its name, we use the following command: **df_var $Col1.** See script lines 313 to 320.



```
> df_var
     Col1 Col2 State
Row1    2    1    MA
Row2    5    4    RI
Row3    3    2    CT
Row4    1   NA    ME
> df_var
     Col1 Col2 State
Row1    2    1    MA
Row2    5    4    RI
Row3    3    2    CT
Row4    1   NA    ME
> df_var[[1]]
[1] 2 5 3 1
> df_var$Col1
[1] 2 5 3 1
```

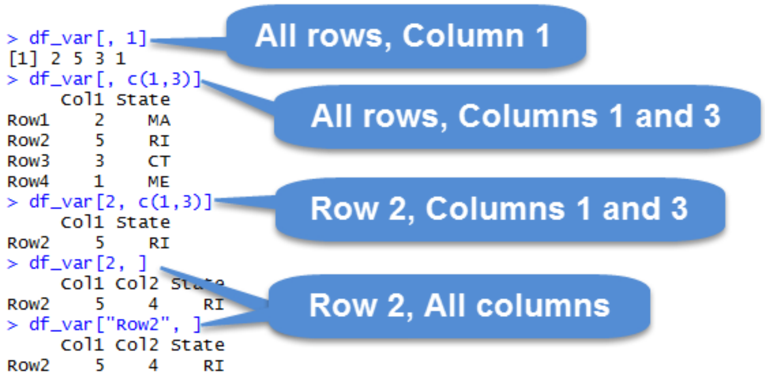
Access column 1 using
df_var[[1]] or df_var$Col1

To access specific elements of a column within a data frame, you need to use a single square braket **[]** after accessing that column. Run lines 321 to 324 in the script to access the second element of the **Col1** column of **df_var**.



```
> df_var[[1]][2]
[1] 5
> df_var$Col1[2]
[1] 5
```

Remember that a data frame shares properties with both lists and matrices. Just like a matrix, you can access elements of a data frame by declaring a row and a column index inside a square bracket **[row_index, column_index]** or by using the names of the rows and columns **[row_name, column_name]**. See lines 325 to 332 in the script for examples.



```
> df_var[, 1]
[1] 2 5 3 1
> df_var[, c(1,3)]
     Col1 State
Row1    2    MA
Row2    5    RI
Row3    3    CT
Row4    1    ME
> df_var[2, c(1,3)]
     Col1 State
Row2    5    RI
> df_var[2, ]
     Col1 Col2 State
Row2    5    4    RI
> df_var["Row2", ]
     Col1 Col2 State
Row2    5    4    RI
```

All rows, Column 1

All rows, Columns 1 and 3

Row 2, Columns 1 and 3

Row 2, All columns

4.4. Summary of Differences between Data Structures

The table below shows a summary of important data structure characteristics that we discussed in this section.

Data Structures	Dimensions	Data type	Create data structure	typeof()	length()	attributes()	Accessing information examples	
							Components	Elements
Atomic vector	1	Homogeneous	c()	integer double character logical	Number of elements			x_y[1]
Factor	1	Homogeneous	factor()	integer	Number of elements	class = "factor" levels		fac_var[2]
List	1	Heterogeneous	list()	list	Number of components	names	lis_lis[[1]] lis_lis$list_1	lis_lis$list_1$num_var[2]
Matrix	2	Homogeneous	matrix()	integer double character logical	Number of elements	dim dimnames[[1]] dimnames[[2]]		mat_var[, 1] mat_var["Row3", "Column2"]
Data frame	2	Heterogeneous	data.frame()	list	Number of components	class = "data.frame" names row.names	df_var[[1]] df_var$Col1	df_var$Col1[2] df_var[, 1] df_var["Row2",]

5. Steps forward

Now we have learned the basics of RStudio and the R together. To keep moving forward with learning R, you can use the following resources:

- **Lynda Campus**: As a Tufts student, staff or faculty, we can access the Lynda.com instructional materials. These include R programming for Data Science, Logistic Regression in R, Interactive Visualizations in R, and more. There are over 678 video tutorials on R available at Lynda.com. For more information, see Lynda Campus.
- **Tufts Data Lab R Resources**: The staff at the Tufts Data Lab create tutorials, workshops (with video recordings) and curated instructional materials on R. You can access these here.

6. Functions used in this tutorial

Package	Function	Description
base	attributes()	Access the attributes of an object
	c()	Combine arguments to form an atomic vector
	cat("\014")	Clear the console
	cbind()	Add columns to a matrix or data frame
	class()	Check the class attribute
	colnames()	Get or set column names for a matrix or data frame
	data.frame()	Create a data frame
	factor()	Create a factor
	getwd()	Get the current working directory
	length()	Get the number of elements or components of an object
	levels()	Get the levels attribute of a factor
	library()	Load a package
	list()	Create a list
	list2env()	Extract all components from a named list and saved them as individual objects
	ls()	Get a list of all objects in the current session
	matrix()	Create a matrix
	mean()	Calculate the arithmetic mean of a variable
	names()	Get or set the names of the components of a list
	ncol()	Get the number of columns in a data frame
	nrow()	Get the number of rows in a data frame
	rbind()	Add rows to a matrix or data frame

	rm()	Remove objects from the environment
	rownames()	Get or set row names for a matrix or data frame
	setwd()	Set the working directory
	summary()	Get a quick overview of the contents of an object
	typeof()	Get the storage mode of an object
	vector()	Create an empty vector
graphics	plot()	Create plots
grDevices	dev.off()	Remove all plots from the plots tab
utils	help()	Access help files
	install.packages()	Download and install packages

7. References

A Gentle Introduction to R, Tufts University Data Lab Workshop, Kyle Monahan
Working with the RStudio IDE (part 1), DataCamp, Garrett Grolemund
Introduction to R, DataCamp, Jonathan Cornelissen
R Tutorial, Kelly Black
Introduction to R Seminar, UCLA Institute for Digital Research and Education
Introductory R workshop for Python Programmers, Ramnath Vaidyanathan
Learn R Programming, Datamentor
Advanced R - Data structures, Hadley Wickham
Advanced R - Subsetting, Hadley Wickham