DATA SOCIETY®

Intro to R programming - day 1

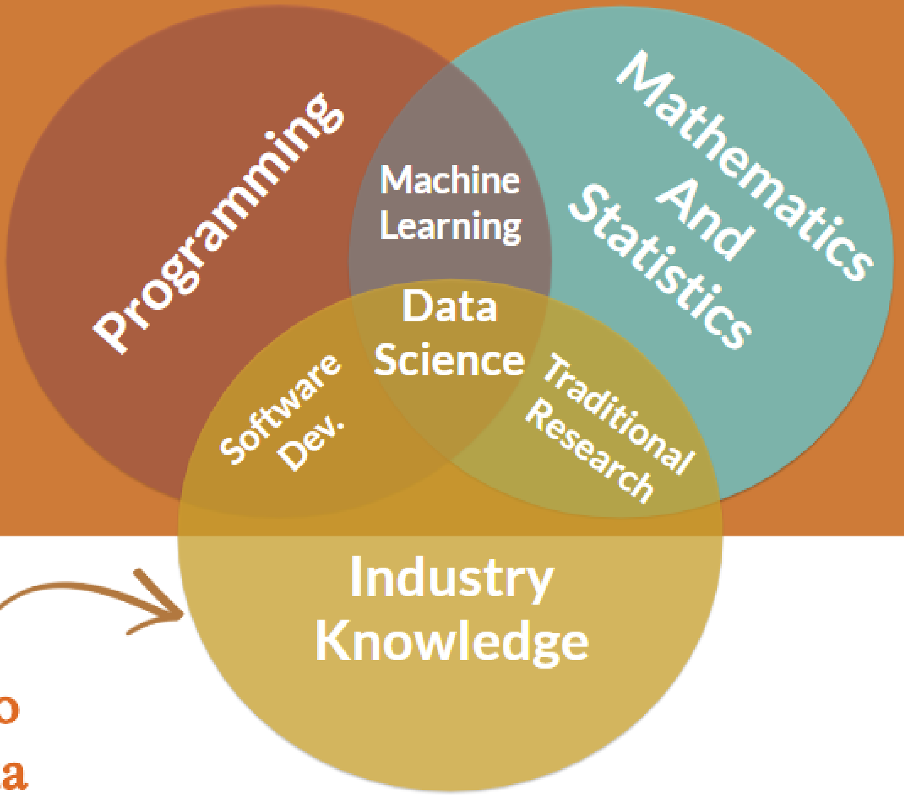
"One should look for what is and not what he thinks should be."
-Albert Einstein.

Module completion checklist

Objective	Complete
Understand and applying basic calculations	
Execute variable functions and identify correct naming conventions for them	
Distinguish data type and class (int, char, float)	
Construct and perform operations using vectors and matrices	
Illustrate and apply the uses of lists and dataframes	
Identify special classes and values in R	

What is data science?

- Data science applies the scientific method to analyzing data
- It lies at the intersection of several disciplines
- It draws on industry knowledge that makes the analysis of Big Data possible

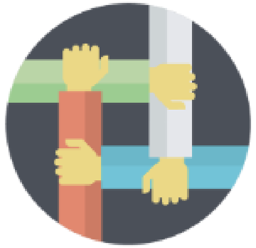


Industry knowledge is essential to knowing what to look for when exploring data

What can data science do?

Data science can be applied across any industry

Empowers management to make better decisions



Increases operational efficiency and investment from staff

Increases accountability and validates decisions

Identifies new opportunities to stay competitive

Topics overview

AFWERX program roadmap

INTRO TO DATA SCIENCE

- What is data science?
- How can you become data-driven?
- What type of problems can data science solve?

R PROGRAMMING

- What is R?
- How is R used as a tool for data science?
- How can you visualize data with R?

POWER BI

- What is Power BI?
- How can you build an effective data dashboard?
- How can you effectively communicate your results?

PYTHON PROGRAMMING

- What is Python?
- How is Python used as a tool for data science?
- How can you visualize data with Python?

SQL

- How can you store structured data in a SQL database?
- How can you query the database to understand the data?

MACHINE LEARNING

- What are the most useful supervised and unsupervised techniques?
- How can you use machine learning to solve problems?

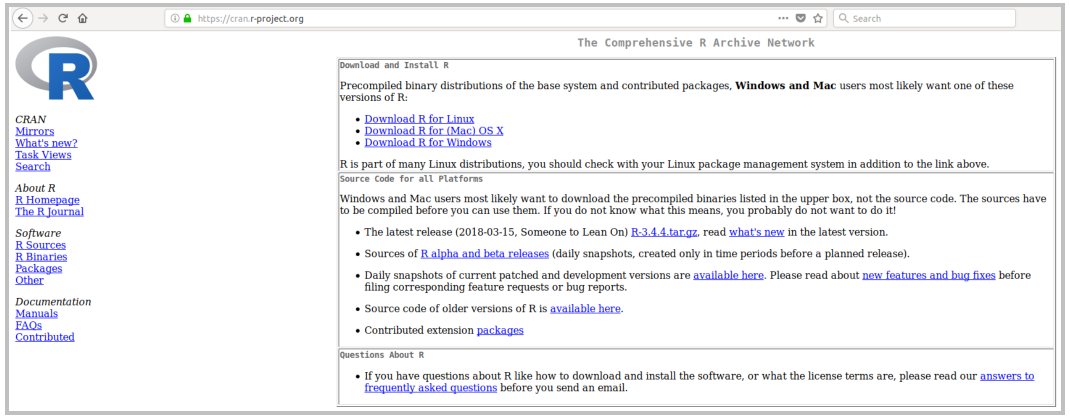
Why use R?

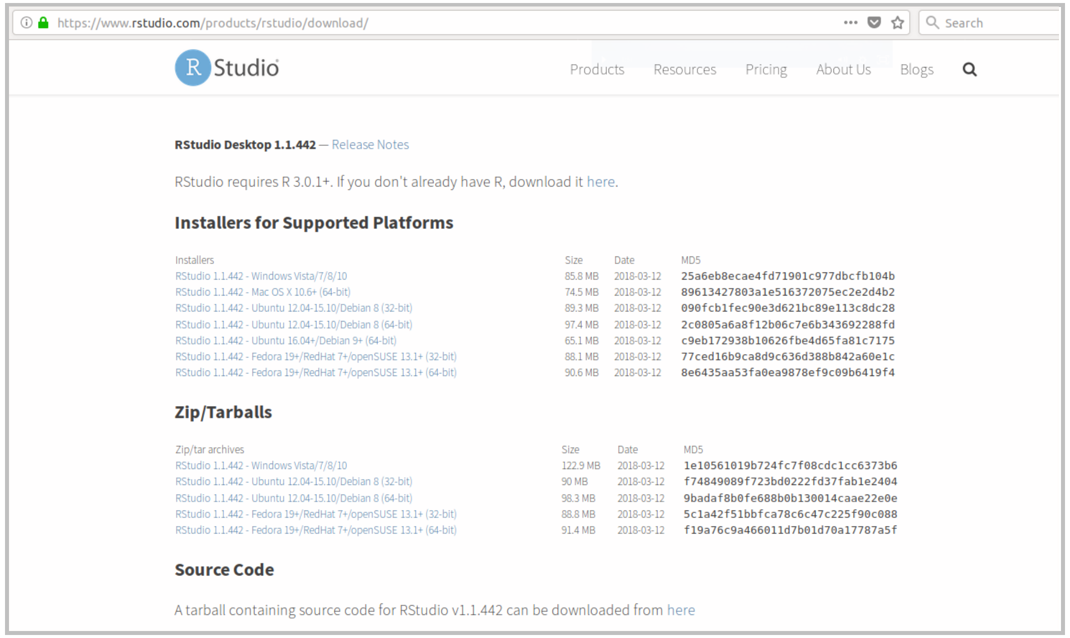
1. De facto standard among professional statisticians
2. Comparable and often superior in power to commercial products (SAS, SPSS, Stata)
3. Available for the Windows, Mac, and Linux operating systems
4. R is a general-purpose programming language, so you can use it to automate analyses
5. Create dynamic graphics and visualization
6. Large community of users, many are prominent scientists: ***www.r-bloggers.com***
7. **Pre-made packages to run data analyses** contributed by user base (over **12,000 packages to date**, and this number is constantly growing)

R compared to Excel

	R	Excel
Data capacity	R can read files as big as several gigabytes and trillions of data points; only limitation is your RAM	Excel cannot read more than 1,048,576 rows and 16,384 columns (2011 version), files over ~300 megabytes can be very slow to work with
Customization	Can create custom visualizations through code, very flexible	Drop down menus limit the ability to manipulate charts and graphs
Analyzing data	Powerful, pre-built packages that speed up work flow	Less flexible built-in analytic abilities that can be augmented by macros
Modeling	Data analysis and statistical models	Complex financial and accounting models
Seeing data	Built-in spreadsheet viewer	Easy to use spreadsheet interface
Usability	Direct commands similar to Excel if-statements	Keyboard shortcuts and slower point-and-click functionality

R & RStudio: installed



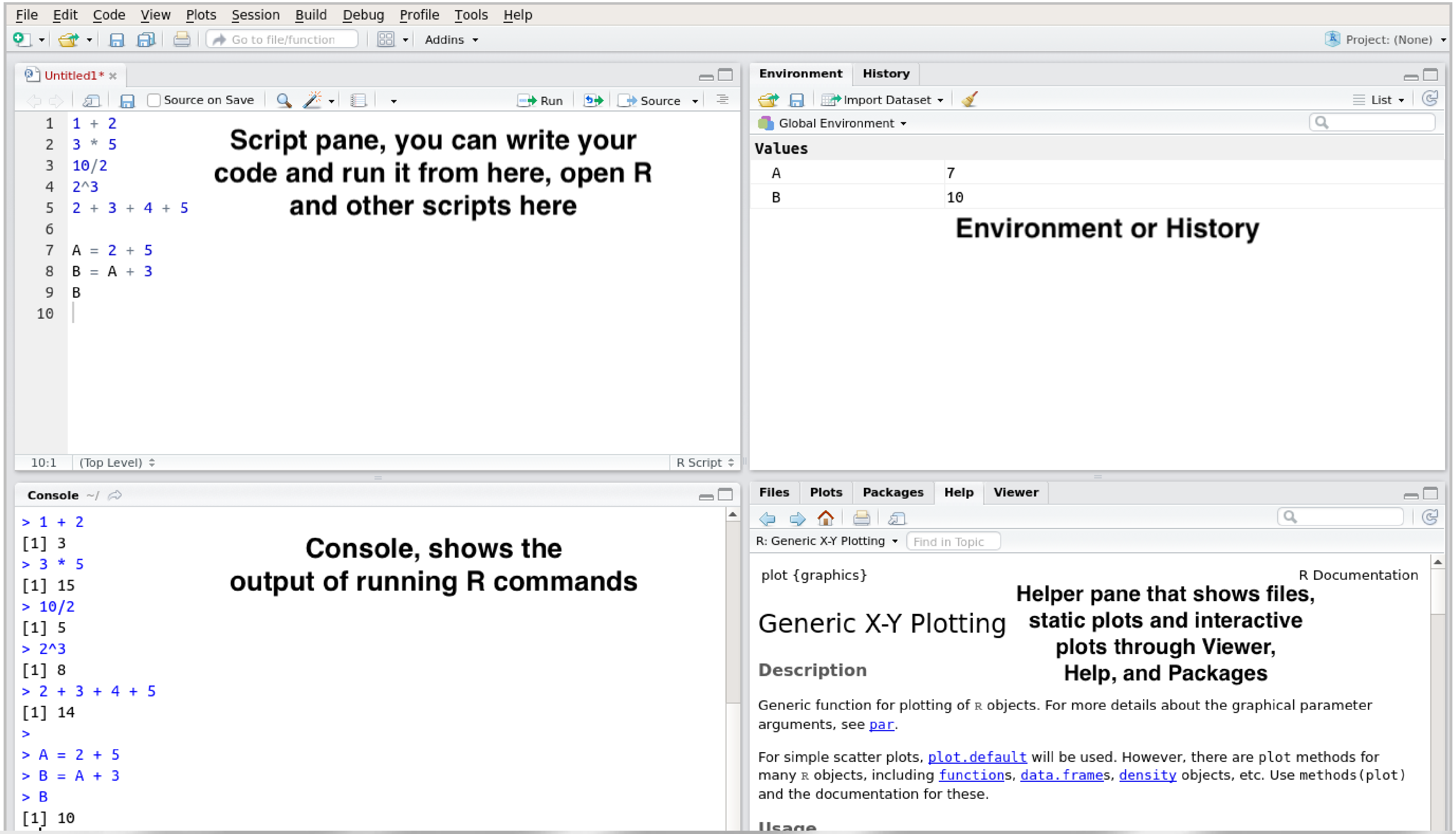


Understanding your panes

A default RStudio layout includes 4 panes:

1. **Top left** pane is used as a `Script` pane, you can write your code and run it from here, open R and other scripts here
2. **Bottom left** pane has a `Console`, which shows the output of running R commands
3. **Top right** is a helper pane that shows your `Environment` or `History`
4. **Bottom right** is another helper pane that shows `Files`, static `Plots` and interactive plots through `Viewer`, `Help`, and `Packages`

Pane overview



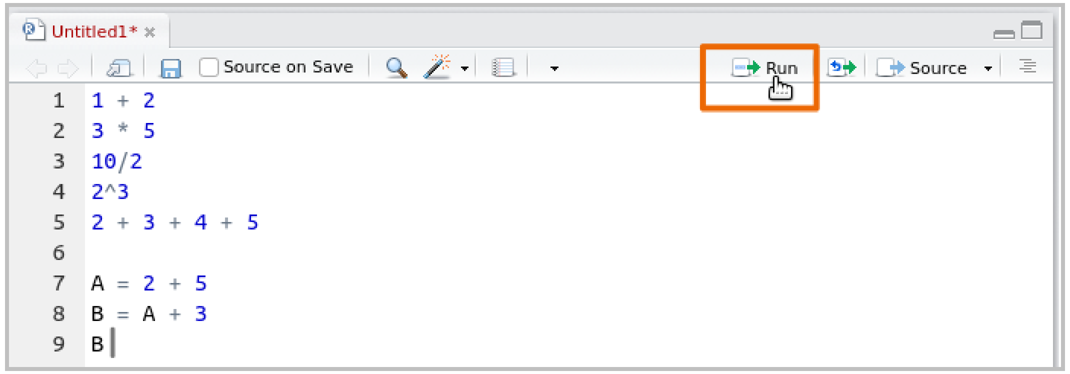
Demystifying code through comments

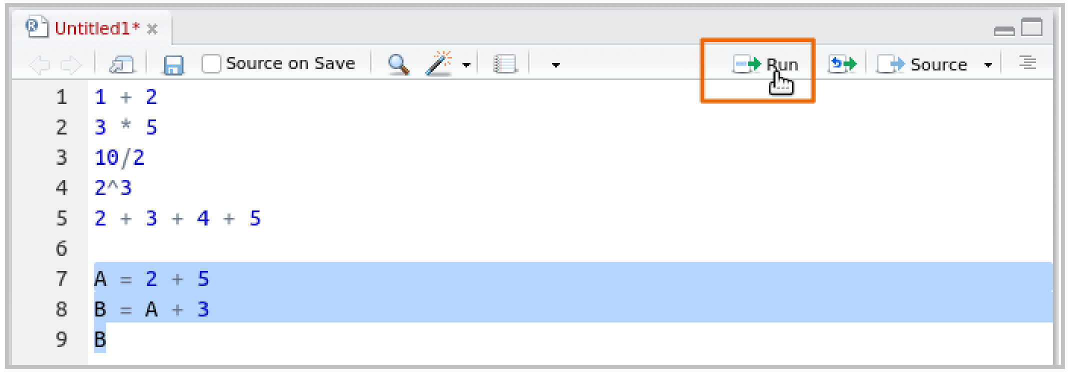
- Use a hashmark to add a comment to your code
- It's good practice to leave descriptive comments in your code if you intend to share it with others. This way others can understand the reasoning behind your code

```
# This is a typical comment in R.
# You don't need a hashmark at
# the end of the line.
# You can add as many as you want,
# just be sure to read them afterwards :)

A = 2 + 5 #<- you can also add comments
B = A + 3 #<- to the end of the code lines
```

Executing commands in R

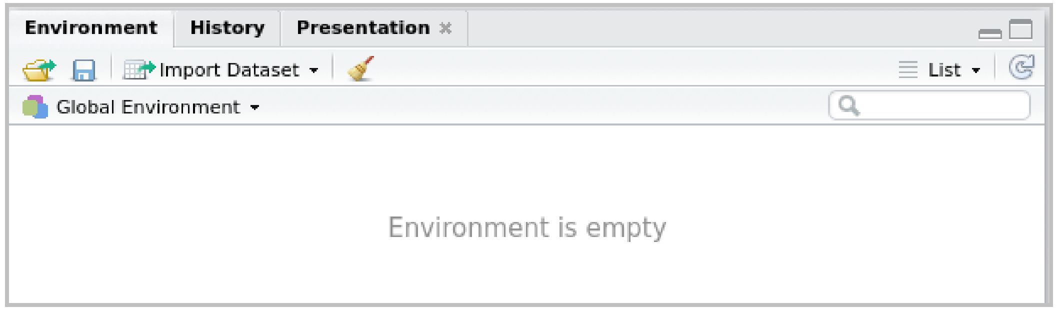
- Code is executed when you press `Run` in the top right hand corner of the script window
- R runs the line of code where your cursor is located
- You can also highlight multiple lines to run at once
- Another equivalent command from your keyboard is `Run` button is a `Ctrl + Enter` (on PC) or `Command + Enter` (on Mac)





Clearing the entire environment

- The clear environment will always show like this in the `Environment` pane



You can clear the environment by clicking on the broom icon at the top of the environment pane

Basic calculations and operations

Function	Operator	Example
Addition	`+`	`1 + 2`
Subtraction	`-`	`10 - 7`
Multiplication	`*`	`1 * 2`
Division	`/`	`9 / 3`
Square root	`sqrt`	`sqrt(100)`
Exponents	`^`	`9 ^ 3`
Remainders	`%%`	`7 %% 3`
Positive and Negative	`- +`	`-7 +7`

Basic calculations and operations

Adding

- Use +

```
# Add whole numbers.
1 + 2
```

```
[1] 3
```

```
# Add numbers with decimals.
3.23 + 4.65
```

```
[1] 7.88
```

Subtracting

- Use –

```
# Subtract whole numbers.
10 - 7
```

```
[1] 3
```

```
# Subtract numbers with decimals.
3.23 - 4.65
```

```
[1] -1.42
```

Basic calculations and operations

Multiplying

- Use *

```
# Multiply whole numbers.
1 * 2
```

```
[1] 2
```

```
# Multiply numbers with decimals.
3.23 * 4.65
```

```
[1] 15.0195
```

Dividing

- Use /

```
# Divide whole numbers.
9 / 3
```

```
[1] 3
```

```
# Divide numbers with decimals.
3.23 / 4.65
```

```
[1] 0.6946237
```

Basic calculations and operations

Square roots

- Use `sqrt`

```
# Take square root of a number.
sqrt(100)
```

```
[1] 10
```

```
# Take square root of an expression.
sqrt(7 * 5)
```

```
[1] 5.91608
```

Exponents

- Use `^` or `**`

```
# Raise number to a power with `^`.
9 ^ 3
```

```
[1] 729
```

```
# Raise number to a power with `**`.
9 ** 3
```

```
[1] 729
```

```
# Raise expression to a power.
(3.23 / 4.65)^2
```

```
[1] 0.482502
```

Basic calculations and operations

Get remainder from division

- Use `%%` (i.e. `modulus`)

```
# Get remainder from division.
7 %% 3
```

```
[1] 1
```

```
# Get remainder from division.
4 %% 2
```

```
[1] 0
```

Perform integer division

- Use `%/%`

```
# Perform integer division.
7 %/% 3
```

```
[1] 2
```

```
# Perform integer division.
4 %/% 2
```

```
[1] 2
```

Knowledge check 1

Exercise 1

Module completion checklist

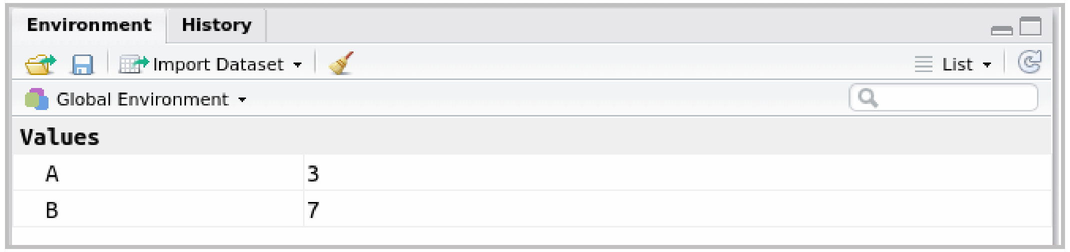
Objective	Complete
Understand and applying basic calculations	✔
Execute variable functions and identify correct naming conventions for them	
Distinguish data type and class (int, char, float)	
Construct and perform operations using vectors and matrices	
Illustrate and apply the uses of lists and dataframes	
Identify special classes and values in R	

Variables and assignment operators

```
# Define a variable using `=`
# as an assignment operator.
B = 2 + 5
B
```

```
[1] 7
```

Notice that you not only can assign numbers to variables, you can assign any expression to a variable!



- You can set variables by setting numbers equal to letters or terms using the assignment operator =
- When a variable is named (instantiated), R stores it in its "environment"
- R session uses the values stored within its environment for all calculations within that session

Reassigning values to variables

```
# 1. Create a variable and assign 67 to it.
this_variable = 67
this_variable
```

```
[1] 67
```

```
# 2. Create another variable and assign -54.
that_variable = -54
that_variable
```

```
[1] -54
```

```
# 3. Calculate their sum.
this_variable + that_variable
```

```
[1] 13
```

```
# 4. Re-assign a value to `this_variable`.
this_variable = 35
this_variable
```

```
[1] 35
```

```
# 5. Add two variables and store the result
#    in `that_variable`.
that_variable = this_variable + that_variable
that_variable
```

```
[1] -19
```

You can re-assign values, variables and expressions to variables you've already used, just be sure to keep track and not to overwrite something you didn't intend to!

Naming variables and functions

Naming rules

- Names of variables and functions can be a combination of letters, digits, periods (`.`), and underscores (_)
- They **must** start with a letter or a period; if it starts with a period, they cannot be followed by a digit
- Reserved words in R **cannot** be used as variable or function names!

Although all of the examples have valid names, not all of them are easy to read and interpret. If you chose one style over the other, stick to it, it will make your coding style more consistent and easy to follow!

Examples

```
this_is_a_valid_name = -5
this_is_a_valid_name
```

```
[1] -5
```

```
This.Is.Also.A.Valid.Name = 3
This.Is.Also.A.Valid.Name
```

```
[1] 3
```

```
.another.valid.name3 = -Inf
.another.valid.name3
```

```
[1] -Inf
```

Naming variable and function rules

- Variable and function names are **case sensitive**

```
# R is case sensitive!
X = 35.5 #<- this `X`
X
```

```
[1] 35.5
```

```
x = -9   #<- is not the same as this `x`
x
```

```
[1] -9
```

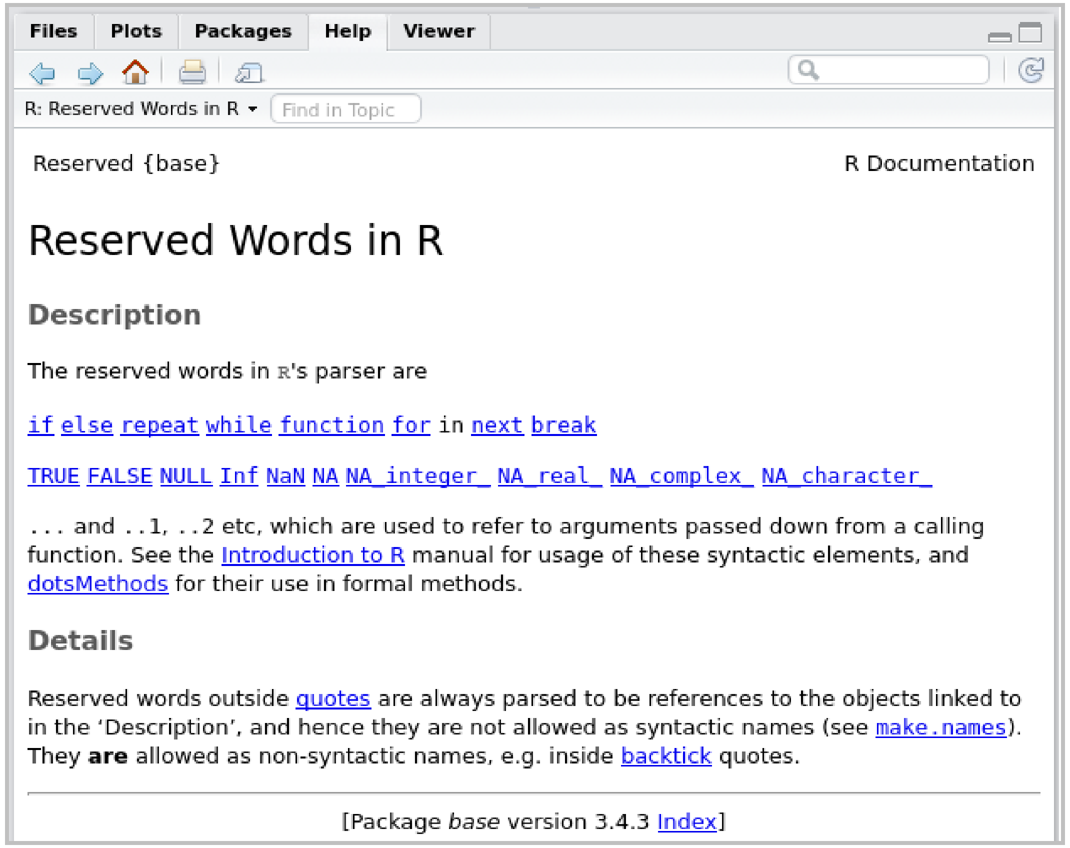
- Reserved names / letters you **cannot** use as variable names

```
# Don't use `T` or `F`, they are reserved
# as a shorthand for `TRUE` and `FALSE`.
T
F

# Don't use `TRUE` and `FALSE` either!
TRUE
FALSE

# Don't use `NULL`,`NA`, `NaN`, `Inf`!
NULL
NA
NaN
Inf
```

Reserved words in R

```
?reserved
```

- To see a full list of reserved words in R, you can run `?reserved` in R and you will find all of the documentation in the `Help` pane of RStudio



Exercise 2

Module completion checklist

Objective	Complete
Understand and applying basic calculations	✔
Execute variable functions and identify correct naming conventions for them	✔
Distinguish data type and class (int, char, float)	
Construct and perform operations using vectors and matrices	
Illustrate and apply the uses of lists and dataframes	
Identify special classes and values in R	

Type

- A **set of values with common characteristics**, from which expressions and functions may be formed. Defines the meaning of data and the way values of that type can be stored. For instance, a **web page** is a type. Any web page has the following basic characteristics:
  - Address
  - Layout (or absence of thereof)
  - Data (or absence of thereof)
  - It can be combined with other web pages into a web site
  - Can be static or can allow people to update its content
  - Web pages are stored on a web server

Class

- A **template that describes how cases of a certain type (or types) can be implemented**. It often carry the same name as the type, if they implement the most basic case of that type. For instance, a **blog page** is a class. Any blog page will be a *special kind of web page* that has:
  - An address
  - Layout
  - Content must have text, pictures and ads
  - Must be a part of a web site called blog
  - Allows people to publish their posts
  - Will be stored on a web server where the blog is stored

Contextualizing type and class

- Types and classes of data are **templates** and **building blocks** of any programming language
- They are both **your map and your legend**
- They are **essential parts** of a key phenomenon in human cognition called **abstraction**

Basic data classes and types

Data class (high level)	Data type (low level)	Example
Integer	Integer	`-1, 5, or 1L, 5L`
Numeric	Double, float	`2.54`
Character	Character	`"Hello"`
Logical	Logical	`TRUE, FALSE`

Basic data classes we will be using

Item	Purpose
Value	Example of class
`typeof()`	Finds the type of the variable
`class()`	Returns the class of the variable
Boolean function	Specific function that checks class and returns TRUE or FALSE
`attributes()`	Checks the metadata/attribute of the variable
`length()`	Checks the length of the object

Create an integer variable

Item	Integer
Value	24, 34L
`typeof()`	integer
`class()`	integer
Boolean function	`is.integer()`
`attributes()`	`NULL`
`length()`	1

```
# Create an integer type variable.
integer_var = 234L

# Check type of variable.
typeof(integer_var)
```

```
[1] "integer"
```

```
# Check if the variable is integer.
is.integer(integer_var)
```

```
[1] TRUE
```

```
# Check length of variable
# (i.e. how many entries).
length(integer_var)
```

```
[1] 1
```

Basic numeric operations

Item	Numeric
Value	24.34
`typeof()`	double
`class()`	numeric
Boolean function	`is.numeric()`
`attributes()`	`NULL`
`length()`	1

```
# Check the type of the variable.
numeric_var = 24.24
typeof(numeric_var)
```

```
[1] "double"
```

Create a character class variable

Item	Character
Value	`"Hello"`
`typeof()`	character
`class()`	character
Boolean function	`is.character()`
`attributes()`	NULL
`length()`	1

```
# Create a character class variable.
character_var = "Hello"
```

```
# Check if the variable is character.
is.character(character_var)
```

```
[1] TRUE
```

```
# Check metadata / attributes of variable.
attributes(character_var)
```

```
NULL
```

```
# Check length of variable
# (i.e. how many entries).
length(character_var)
```

```
[1] 1
```

Some useful character operations

```r
# Create another character class variable.
case_study = "JUmbLEd CaSE"

# Convert a character string to lower case.
tolower(case_study)
```

```
[1] "jumbled case"
```

```r
# Convert a character string to upper case.
toupper(case_study)
```

```
[1] "JUMBLED CASE"
```

```r
# Count number of characters in a string.
nchar(case_study)
```

```
[1] 12
```

```r
# Compare to the output of the `length` command.
length(case_study)
```

```
[1] 1
```

```r
# Get just a part of character string.
substr(case_study, #<- original string
       1,          #<- start index of substring
       7)          #<- end index of substring
```

```
[1] "JUmbLEd"
```

Create a logical class variable

```
# Create a logical class variable.
logical_var = TRUE

# Check type of variable.
typeof(logical_var)
```

```
[1] "logical"
```

Item	Logical
Value	`TRUE` or `FALSE`
`typeof()`	logical
`class()`	logical
Boolean function	`is.logical()`
`attributes()`	`NULL`
`length()`	1

Summary & conversion of basic data classes

Item	Integer	Numeric	Character	Logical
Value	`24, 34L`	`24.34`	Hello	`TRUE` or `FALSE`
`typeof()`	integer	double	character	logical
`class()`	integer	numeric	character	logical
Boolean function	`is.integer()`	`is.numeric()`	`is.character()`	`is.logical()`
`attributes()`	`NULL`	`NULL`	`NULL`	`NULL`
`length()`	1	1	1	1
To convert a variable to this type	`as.integer()`	`as.numeric()`	`as.character()`	`as.logical()`

Knowledge check 2

Exercise 3

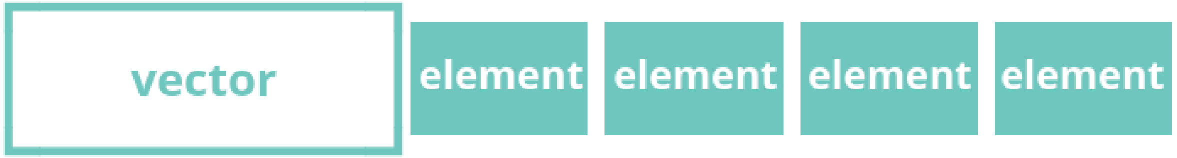
Module completion checklist

Objective	Complete
Understand and applying basic calculations	✔
Execute variable functions and identify correct naming conventions for them	✔
Distinguish data type and class (int, char, float)	✔
Construct and perform operations using vectors and matrices	
Illustrate and apply the uses of lists and dataframes	
Identify special classes and values in R	

Basic data structures

Data structure	Number of dimensions	Single data type	Multiple data types
Vector (atomic vector)	1 (entries)	✔	✘
Vector (list)	1 (entries)	✔	✔
Matrix	2 (rows and columns)	✔	✘
Dataframe	2 (rows and columns)	✔	✔

Atomic vectors



- A `vector` is a collection of elements that has a single dimension of entries
  - A single dimension of entries is also known as *array*
- Vectors are considered the simplest and most common data structure throughout nearly all programming languages
- Vectors contain elements of a single class or type
- Mode of vector refers to the types of elements it contains
  - Most common `modes` of vectors are: `character`, `logical`, `numeric`

Your computer's memory is one giant single-dimensional array!

Creating atomic vectors

```
# To make an empty vector in R,
# you have a few options:
# Option 1: use `vector()` command.
# The default in R is an empty vector of
# `logical` mode!
vector()
```

```
logical(0)
```

```
# Option 2: use `c()` command
# (`c` stands for concatenate).
# The default empty vector produced by `c()`
# has a single entry `NULL`!
c()
```

```
NULL
```

An empty vector will always be of length 0, since it has no entries in it!

- To make a vector out of a given set of character strings, you can wrap them into `c` and separate by commas

```
# Make a vector from a set of char. strings.
c("My", "name", "is", "Vector")
```

```
[1] "My"     "name"   "is"     "Vector"
```

- To make a vector out of a given set of numbers, you can wrap them into `c` and separate by commas

```
# Make a vector out of given set
# of numbers.
c(1, 2, 3, 765, -986, 0.5)
```

```
[1]    1.0    2.0    3.0  765.0 -986.0    0.5
```

Working with vectors

```
# Create a vector of mode `character` from
# pre-defined set of character strings.
character_vec = c("My", "name", "is", "Vector")
character_vec
```

```
[1] "My"     "name"   "is"     "Vector"
```

```
# Check if the variable is character.
is.character(character_vec)
```

```
[1] TRUE
```

```
# Check metadata/attributes of variable.
attributes(character_vec)
```

```
NULL
```

Item	Vector
Value	`character_vec`
`typeof()`	character
`class()`	character
Boolean function	`is.character()`
`attributes()`	`NULL`
`length()`	4

```
# Check length of variable
# (i.e. how many entries).
length(character_vec)
```

```
[1] 4
```

Access vectors values

```
# To access an element inside of the
# vector, use `[]` and the index of the element.
character_vec[1]
```

```
[1] "My"
```

```
# To access multiple elements inside of
# a vector, use the start and end indices
# with `:` in-between.
character_vec[1:3]
```

```
[1] "My"   "name" "is"
```

Notice, all data structures in R including vectors start at index 1!

```
# A special form of a vector in R
# is a sequence.
number_seq = seq(from = 1, to = 5, by = 1)
number_seq
```

```
[1] 1 2 3 4 5
```

```
# Check class.
class(number_seq)
```

```
[1] "numeric"
```

```
# Subset the first 3 elements.
number_seq[1:3]
```

```
[1] 1 2 3
```

Using operations on vectors

```
# Let's take our vector.
number_seq
```

```
[1] 1 2 3 4 5
```

```
# Add a number to every entry.
number_seq + 5
```

```
[1]  6  7  8  9 10
```

```
# Subtract a number from every entry.
number_seq - 5
```

```
[1] -4 -3 -2 -1  0
```

```
# Multiply every entry by a number.
number_seq * 2
```

```
[1]  2  4  6  8 10
```

All arithmetic operations in R are element-wise!

```
# To sum all elements, use `sum`.
sum(number_seq)
```

```
[1] 15
```

```
# To multiply all elements, use `prod`.
prod(number_seq)
```

```
[1] 120
```

```
# To get the mean of all vector
# values, use `mean`.
mean(number_seq)
```

```
[1] 3
```

```
# To get the smallest value
# in a vector, use `min`.
min(number_seq)
```

```
[1] 1
```

Appending & naming

```
# To name each entry in a vector, use `names`.
names(number_seq) = c("First", "Second",
                      "Third", "Fourth",
                      "Fifth")

# Check the attributes of vector.
attributes(number_seq)
```

```
$names
[1] "First"  "Second" "Third"  "Fourth" "Fifth"
```

```
# Check the length of vector.
length(number_seq)
```

```
[1] 5
```

Item	Vector
Value	`number_seq`
`typeof()`	double
`class()`	numeric
Boolean function	`is.numeric()`
`attributes()`	`names`
`length()`	5

```
# To append elements to a vector, just
# wrap the vector and additional element(s)
# into `c` again!
character_vec = c(character_vec, "!")
character_vec
```

```
[1] "My"     "name"   "is"     "Vector" "!"
```

Why are these vectors called `atomic?`

- What happens if you mix different types of data inside an atomic vector?

```
# Try mixing different types of data inside an atomic vector
atomic_vec = c(333, "some text", TRUE, NULL)
atomic_vec
```

```
[1] "333"       "some text" "TRUE"
```

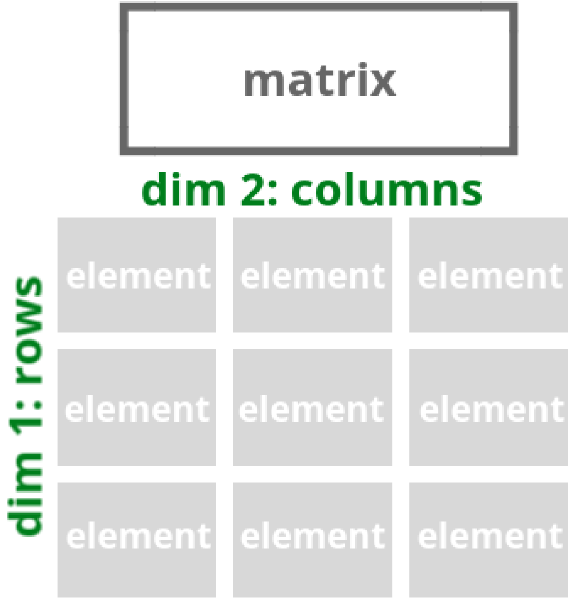
```
# Check class of the resulting vector.
class(atomic_vec)
```

```
[1] "character"
```

- R will **cast** (i.e. coerce) all elements of that vector to a type/class that can most easily accommodate all elements it contains!
- This is why this type of data structure is called `atomic`, which, in the computer science world, is equivalent to homogeneous or unsplittable (although we all know we can split the `atom` ☢)

Matrices

- A matrix is a 2D `vector`
- A `matrix` is also an array of elements, but instead of having 1 dimension, it has 2
- Since a `matrix` is a 2-dimensional version of an atomic vector, it only allows elements of the same `type`
- Working with matrices is very similar to working with 1D `vectors`



Making matrices

```
# Create a matrix with 3 rows and 3 columns.
sample_matrix1 = matrix(nrow = 3, #<- n rows
                        ncol = 3) #<- m cols
sample_matrix1
```

```
     [,1] [,2] [,3]
[1,]   NA   NA   NA
[2,]   NA   NA   NA
[3,]   NA   NA   NA
```

```
# Notice that by default an empty matrix
# will be filled with NAs.

# Check matrix dimensions.
dim(sample_matrix1)
```

```
[1] 3 3
```

```
# Notice that the `length` command will produce
# the total number elements in the matrix
# (length = n rows x m cols).
length(sample_matrix1)
```

```
[1] 9
```

```
# Another way to create a matrix is to make
# it out of a vector of numbers.
sample_matrix2 = 1:9 #<- another way to make
                     #   a sequence of numbers!

# Assign dimensions to matrix:
# 1st number is for rows, 2nd is for columns.
dim(sample_matrix2) = c(3, #<- n rows
                        3) #<- m cols

sample_matrix2
```

```
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

```
# Check matrix dimensions.
dim(sample_matrix1)
```

```
[1] 3 3
```

Making matrices

- The shorthand version of the previous 2 commands looks like this

```
# Create a matrix from a sequence of numbers
# with 3 rows & 3 columns.
sample_matrix3 = matrix(1:9,       #<- entries
                        nrow = 3,  #<- n rows
                        ncol = 3)  #<- m cols
sample_matrix3
```

```
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

Notice that `matrix` command arranges the values by `column` by default!

- Create the same matrix but with values arranged by **rows**

```
# Create a matrix from a sequence of numbers
# with 3 rows & 3 columns arranged by row.
sample_matrix4 = matrix(1:9,
                        nrow = 3,
                        ncol = 3,
                        byrow = TRUE)
sample_matrix4
```

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

Checking type and class of matrix

```
# Check type of variable.
typeof(sample_matrix4)
```

```
[1] "integer"
```

```
# Check class of variable.
class(sample_matrix4)
```

```
[1] "matrix"
```

```
# Check if the variable of type `integer`.
is.integer(sample_matrix4)
```

```
[1] TRUE
```

```
# Check metadata/attributes of variable.
attributes(sample_matrix4)
```

```
$dim
[1] 3 3
```

Using rbind and cbind

```
# To append rows to a matrix, use `rbind`.
new_matrix1 = rbind(sample_matrix4,
                    10:12)
new_matrix1
```

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
[4,]   10   11   12
```

```
# To append columns to a matrix, use `cbind`.
new_matrix2 = cbind(sample_matrix3,
                    10:12)
new_matrix2
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

```
# To access an element of a matrix, use
# the row and column indices separated
# by a comma inside of `[]`.
new_matrix1[1, 2] #<- element in row 1, col 2
```

```
[1] 2
```

```
# To access a row, leave the space in
# column index empty.
new_matrix1[1 , ]
```

```
[1] 1 2 3
```

```
# To access a column, leave the space in
# row index empty.
new_matrix1[ , 2]
```

```
[1]  2  5  8 11
```

Operations on matrices

```r
# Let's take a sample matrix.
sample_matrix2
```

```
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

```r
# Add a number to every entry.
sample_matrix2 + 5
```

```
     [,1] [,2] [,3]
[1,]    6    9   12
[2,]    7   10   13
[3,]    8   11   14
```

```r
# Multiply every entry by a number.
sample_matrix2 * 2
```

```
     [,1] [,2] [,3]
[1,]    2    8   14
[2,]    4   10   16
[3,]    6   12   18
```

```r
# To sum all elements, use `sum`.
sum(sample_matrix2)
```

```
[1] 45
```

```r
# To multiply all elements, use `prod`.
prod(sample_matrix2)
```

```
[1] 362880
```

```r
# To get the mean of all matrix
# values, use `mean`.
mean(sample_matrix2)
```

```
[1] 5
```

```r
# To get the smallest value
# in a matrix, use `min`.
min(sample_matrix2)
```

```
[1] 1
```

Names & attributes

```
# To name columns of a matrix, use `colnames`.
colnames(sample_matrix2) = c("Col1", "Col2",
"Col3")

# To name rows of a matrix, use `rownames`.
rownames(sample_matrix2) = c("Row1", "Row2",
"Row3")
sample_matrix2
```

```
     Col1 Col2 Col3
Row1    1    4    7
Row2    2    5    8
Row3    3    6    9
```

```
# Check the attributes of a matrix.
attributes(sample_matrix2)
```

```
$dim
[1] 3 3

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

$dimnames[[2]]
[1] "Col1" "Col2" "Col3"
```

Item	Matrix
To create	`matrix()`
Value	`sample_matrix2`
`typeof()`	integer
`class()`	matrix
Boolean function	`is.matrix()`
`attributes()`	dim, `dimnames[[1]]`, `dimnames[[2]]`
`length()`	9

Knowledge check 3

Exercise 4

Module completion checklist

Objective	Complete
Understand and applying basic calculations	✔
Execute variable functions and identify correct naming conventions for them	✔
Distinguish data type and class (int, char, float)	✔
Construct and perform operations using vectors and matrices	✔
Illustrate and apply the uses of lists and dataframes	
Identify special classes and values in R	

Lists

- `List` is a collection of entries that act as **containers**
- List has a **single dimension** at its top level
- List is also known as a **generic vector** because the elements can be of the same or of different types
- Lists can be **nested** i.e. `lists` can contain elements that are also `lists`

If you have ever worked with `JSON` files, they can be translated naturally into the `list` data structure

Creating a lists

- Creating a `list`

```
# To make an empty list in R,
# you have a few options:
# Option 1: use `list()` command.
list()
```

```
list()
```

- How is this different from a vector?

```
# Make a list with different entries.
sample_list = list(1, "am", TRUE)
sample_list
```

```
[[1]]
[1] 1

[[2]]
[1] "am"

[[3]]
[1] TRUE
```

Naming list elements

- Lists can have *attributes* such as `names`
- You can name list elements when you **create** a `list`

```
# Create a named list.
sample_list_named = list(One = 1,
                         Two = "am",
                         Three = TRUE)
sample_list_named
```

```
$One
[1] 1

$Two
[1] "am"

$Three
[1] TRUE
```

```
attributes(sample_list_named)
```

```
$names
[1] "One"    "Two"    "Three"
```

- You can also set element names **after** it has been created

```
# Name existing list.
names(sample_list) = c("One", "Two", "Three")
sample_list
```

```
$One
[1] 1

$Two
[1] "am"

$Three
[1] TRUE
```

```
attributes(sample_list)
```

```
$names
[1] "One"    "Two"    "Three"
```

Introducing structure

```
?str

str(object) #<- either a list, a matrix, or
dataframe
```

str {utils} R Documentation

Compactly Display the Structure of an Arbitrary R Object

Description

Compactly display the internal **str**ucture of an R object, a diagnostic function and an alternative to summary (and to some extent, dput). Ideally, only one line for each 'basic' structure is displayed. It is especially well suited to compactly display the (abbreviated) contents of (possibly nested) lists. The idea is to give reasonable output for **any** R object. It calls args for (non-primitive) function objects.

`strOptions()` is a convenience function for setting `options(str = .)`, see the examples.

Usage

```
str(object, ...)

## S3 method for class 'data.frame'
str(object, ...)

## Default S3 method:
str(object, max.level = NA,
    vec.len  = strO$vec.len, digits.d = strO$digits.d,
    nchar.max = 128, give.attr = TRUE,
    drop.deparse.attr = strO$drop.deparse.attr,
    give.head = TRUE, give.length = give.head,
    width = getOption("width"), nest.lev = 0,
    indent.str = paste(rep.int(" ", max(0, nest.lev + 1)),
                       collapse = ".."),
    comp.str = "$ ", no.list = FALSE, envir = baseenv(),
    strict.width = strO$strict.width,
```

List structure

```
# Inspect the list's structure.
str(sample_list)
```

```
List of 3
 $ One  : num 1
 $ Two  : chr "am"
 $ Three: logi TRUE
```

- Command `str` lets you inspect the object's structure - it provides:
  - the `class` of the object (e.g. `List`)
  - the `length` of the object (e.g. `3`)
  - snippet of each entry and its `type` (e.g. `One: num 1`, `Two: chr "am"`, `Three: logi TRUE`)

Accessing data within lists

- To access an element of a list, you can use its **index**

```
# Access an element of a list.
sample_list[[2]]
```

```
[1] "am"
```

```
# Access a sub-list with its element(s).
sample_list[2]
```

```
$Two
[1] "am"
```

```
# Access a sub-list with its element(s).
sample_list[2:3]
```

```
$Two
[1] "am"

$Three
[1] TRUE
```

- You can also reference an element by its **name**, using the $ operator (as seen in the output of the `str` command)

```
# Access named list elements.
sample_list$One
```

```
[1] 1
```

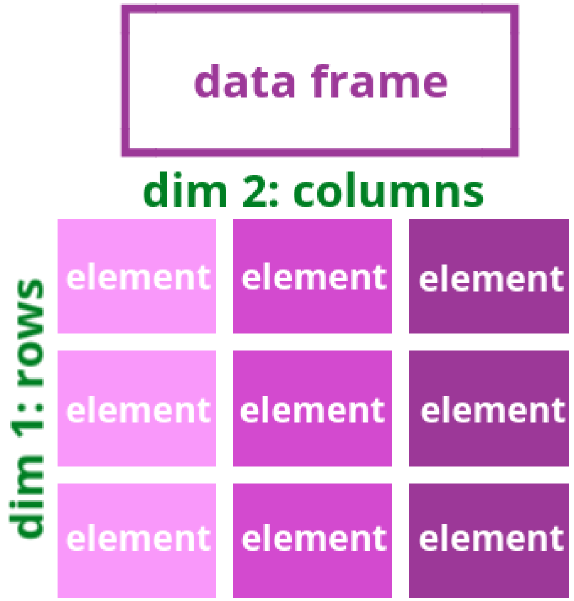
```
sample_list$Two
```

```
[1] "am"
```

```
sample_list$Three
```

```
[1] TRUE
```

Dataframes



If you have ever worked with relational databases, you can think of a dataframe as a table in a relational database!

- `data.frame` is a special kind of `list`, that is limited to a **2D structure**
- Each entry in a `list` is a `column`
- Each `column` has the same number of *entries*
- Columns can be of **different types** (e.g. `character`, `numeric`, `logical`)
- But within each column the entries are always of the **same type**, which makes each column of a `data.frame` an `atomic vector`
- It combines properties of both `lists` and `atomic vectors`, which makes it a *de facto* standard data structure for use in data analysis

Making dataframes

```
# To make an empty dataframe in R,
# use `data.frame()` command.
data.frame()
```

```
data frame with 0 columns and 0 rows
```

```
# To make a dataframe with several
# columns, pass column values
# to `data.frame()` command just like
# you would do with lists.
data.frame(1:5, 6:10)
```

```
  X1.5 X6.10
1    1     6
2    2     7
3    3     8
4    4     9
5    5    10
```

- As with vector, matrices, & lists, a `data.frame` can be created empty
- Column values can be passed to dataframes when created as you would with lists
- You can also combine pre-existing vectors

Without defined column names, `data.frame` auto-generates them. Column names in R cannot have numbers as the first character, which is why R appends X to them!

Naming columns

- Name (rename) columns after `data.frame` is created

```
# Dataframe with unnamed columns.
unnamed_df = data.frame(1:3, 4:6)
unnamed_df
```

```
  X1.3 X4.6
1    1    4
2    2    5
3    3    6
```

```
# Name columns of a dataframe.
colnames(unnamed_df) = c("col1", "col2")
unnamed_df
```

```
  col1 col2
1    1    4
2    2    5
3    3    6
```

- Name columns at the time of creation of the `data.frame`

```
# Pass column names and values to
# `data.frame` command just like you
# would do with named lists.
named_df = data.frame(col1 = 1:3, col2 = 4:6)
named_df
```

```
  col1 col2
1    1    4
2    2    5
3    3    6
```

Naming rows

- You can **rename** row names of any dataframe

```
# View dataframe.
named_df
```

```
  col1 col2
1    1    4
2    2    5
3    3    6
```

```
# Rename dataframe rows.
rownames(named_df) = c(7:9)
named_df
```

```
  col1 col2
7    1    4
8    2    5
9    3    6
```

- You can also create a dataframe *and* **define** row names at the time of its creation

```
# Define row names explicitly,
# use a `row.names` argument.
data.frame(col1 = 1:3,
           col2 = 4:6,
           row.names = 7:9)
```

```
  col1 col2
7    1    4
8    2    5
9    3    6
```

Converting a matrix

```
# Make a dataframe from matrix.
sample_df1 = as.data.frame(sample_matrix1)
sample_df1
```

```
  V1 V2 V3
1 NA NA NA
2 NA NA NA
3 NA NA NA
```

```
# Make a dataframe from matrix with
# named columns and rows.
sample_df2 = as.data.frame(sample_matrix2)
sample_df2
```

```
     Col1 Col2 Col3
Row1    1    4    7
Row2    2    5    8
Row3    3    6    9
```

- We can make a dataframe from a matrix by casting a `matrix` into a `data.frame` with `as.data.frame` command

Converting a matrix

Row and column names of a matrix

```
# Check attributes of a dataframe.
attributes(sample_df1)
```

```
$names
[1] "V1" "V2" "V3"

$class
[1] "data.frame"

$row.names
[1] 1 2 3
```

- **Unnamed** `matrix` **column names** will default to `V1, V2, ..., Vm`, where `m = num columns` of a matrix
- **Unnamed** `matrix` **row names** will default to `1, 2, ..., n`, where `n = num rows` of a matrix

```
# Check the attributes of dataframe.
attributes(sample_df2)
```

```
$names
[1] "Col1" "Col2" "Col3"

$class
[1] "data.frame"

$row.names
[1] "Row1" "Row2" "Row3"
```

- **Named** `matrix` **column names** will become `data.frame` column names
- **Named** `matrix` **row names** will become `data.frame` row names

Selecting columns

- Let's explore the ways we can select a column of a `data.frame`
  - Use `$column_name`
  - Use `[, "column name"]`
  - Use `[, column_index]`

```
# To access a column of a dataframe
# Option 1: Use a `$column_name`.
named_df$col1
```

```
[1] 1 2 3
```

```
# To access a column of a dataframe
# Option 2: Use a `[, "column_name"]`.
named_df[,"col1"]
```

```
[1] 1 2 3
```

```
# To access a column of a dataframe
# Option 3: Use a `[, column_index]`.
named_df[, 1]
```

```
[1] 1 2 3
```

Subsetting rows

- Let's explore the ways we can select a row of a `data.frame`
  - Use `[row_index, ]`
  - Use `["row_name", ]`

```
# To access a row of a dataframe
# Option 1: use `[row_index, ]`.
sample_df2[1, ]
```

```
     Col1 Col2 Col3
Row1    1    4    7
```

```
# To access a row of a dataframe
# Option 2: use `["row_name", ]`.
sample_df2["Row1", ]
```

```
     Col1 Col2 Col3
Row1    1    4    7
```

Accessing individual values

```
# Option 1:
# `data_frame$column_name[row_index]`

sample_df2$Col2[1]
```

```
[1] 4
```

```
# Option 2:
# `["row_name", "column_name"]`

sample_df2["Row1", "Col2"]
```

```
[1] 4
```

```
# Option 3:
# `[row_index, column_index]`

sample_df2[1, 2]
```

```
[1] 4
```

```
# Option 4:
# `data_frame[[column_index]][row_index]`

sample_df2[[2]][1]
```

```
[1] 4
```

Adding new columns

```
# To add a new column to a dataframe
# Option 1: use `$new_column_name`.
sample_df2$Col4 = "New column"
sample_df2
```

```
     Col1 Col2 Col3        Col4
Row1    1    4    7 New column
Row2    2    5    8 New column
Row3    3    6    9 New column
```

```
# To add new column(s) to a dataframe
# Option 2: use `cbind`.
sample_df2 = cbind(sample_df2,
                   Col5 = c("Yet another",
                            "new",
                            "column"))
```

Using operations in our dataframe

```
# Let's take our sample dataframe.
str(sample_df2)
```

```
'data.frame':	3 obs. of  5 variables:
 $ Col1: int  1 2 3
 $ Col2: int  4 5 6
 $ Col3: int  7 8 9
 $ Col4: chr  "New column" "New column" "New column"
 $ Col5: Factor w/ 3 levels "column","new",..: 3 2 1
```

```
# Add a number to each value in a column.
sample_df2$Col1 + 2
```

```
[1] 3 4 5
```

```
# Add a number to each value in a row.
sample_df2[1:3, ] + 2
```

```
Error in FUN(left, right) : non-numeric argument to binary operator
```

We get an error message because you can't add characters in a dataframe!

Module completion checklist

Objective	Complete
Understand and applying basic calculations	✔
Execute variable functions and identify correct naming conventions for them	✔
Distinguish data type and class (int, char, float)	✔
Construct and perform operations using vectors and matrices	✔
Illustrate and apply the uses of lists and dataframes	✔
Identify special classes and values in R	

Factors

```
# Let's take a look at the structure of the dataframe.
str(sample_df2)
```

```
'data.frame':	3 obs. of  5 variables:
 $ Col1: int  1 2 3
 $ Col2: int  4 5 6
 $ Col3: int  7 8 9
 $ Col4: chr  "New column" "New column" "New column"
 $ Col5: Factor w/ 3 levels "column","new",..: 3 2 1
```

- A talk about data classes, types, and data structures in R is not complete without a special class `factor`
- A `factor` is a class of variable that is used to **quantify categorical** data
- Every `factor` variable has `Levels`, which are unique instances of the values in the column (e.g. `Col5` has 3 unique values, hence the 3 levels)
- Use levels() to find the number of unique values of a factor

Dates

```
# Let's make a dataframe.
special_data = data.frame(date_col1 = c("2018-01-01", #<- make a column with character strings
                                        "2018-02-01", #    in the format of date (YYYY-MM-DD)
                                        "2018-03-01"),
                          stringsAsFactors = FALSE)   #<- this option allows us to tell R
                                                      #    to NOT interpret strings as `factors`
special_data
```

```
   date_col1
1 2018-01-01
2 2018-02-01
3 2018-03-01
```

```
# Take a look at the structure.
# Notice both columns appear as `character`
# and not as `factor`.
str(special_data)
```

```
'data.frame':	3 obs. of  1 variable:
 $ date_col1: chr  "2018-01-01" "2018-02-01" "2018-03-01"
```

Dates and basic formats

- Given a character string of a particular format, we can convert to a `Date` using `as.Date` function (e.g. `YYYY-MM-DD` format will be automatically detected by R)

```
# Let's make another vector with dates, but in
# a different format.
new_dates = c("January 1, 2018",
              "February 1, 2018",
              "March 1, 2018")

# Let's add another column to the dataframe
# and save it as a Date with a special format.
special_data$date_col2 = as.Date(new_dates,
                         format = "%B %d, %Y")
special_data
```

```
   date_col1  date_col2
1 2018-01-01 2018-01-01
2 2018-02-01 2018-02-01
3 2018-03-01 2018-03-01
```

- R now recognizes these values as a `Date` object, and thus can be operated upon (summed, subtracted, etc.)

Code	Value
`%d`	Day of the month (number)
`%m`	Month (number)
`%b`	Month (abbreviated name)
`%B`	Month (full name)
`%y`	Year (2 digit)
`%Y`	Year (4 digit)

Identifying `NA`s

- `is.na` helps identify `NA` values
- We will illustrate this now:

```
# Let's add a column with a numeric vector.
special_data$num_col1 = c(1, 555, 3)

# Let's make the 2nd element in that column `NA`.
special_data$num_col1[2] = NA

# To check for NAs, we use `is.na`.
is.na(special_data$num_col1[2])
```

```
[1] TRUE
```

```
# We can also use it to check the whole column/vector.
# The result will be a vector of `TRUE` or `FALSE`
# values corresponding to each element of the vector.
is.na(special_data$num_col1)
```

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

Identifying `NULL`

- Another special value in R is `NULL`. This value makes an object or a part of the object to be `NULL`ified, i.e. removed or cleared.

```
# To get rid of a column in a `data.frame`, all
# you have to do is set it to `NULL`.
special_data$num_col3 = NULL
special_data
```

```
   date_col1  date_col2 num_col1
1 2018-01-01 2018-01-01        1
2 2018-02-01 2018-02-01       NA
3 2018-03-01 2018-03-01        3
```

```
# To check for NULLs, use `is.null`.
is.null(special_data$num_col3)
```

```
[1] TRUE
```

```
# To check for NULLs, use `is.null`.
is.null(special_data$num_col2)
```

```
[1] TRUE
```

Knowledge check 4

Exercise 5

Module completion checklist

Objective	Complete
Understand and applying basic calculations	✔
Execute variable functions and identify correct naming conventions for them	✔
Distinguish data type and class (int, char, float)	✔
Construct and perform operations using vectors and matrices	✔
Illustrate and apply the uses of lists and dataframes	✔
Identify special classes and values in R	✔

Workshop!

- **Today will be your first *after class* workshop**
- Workshops are to be completed outside of class and emailed to the instructor by the beginning of class tomorrow
- Make sure to comment your code so that it is easy for others to understand what you are doing
- This is an exploratory exercise to get you comfortable with the content we discussed today
- Workshop objectives:
  - Identify and explore more operations with basic data types in R
  - Explore with vectors, matrices and lists in R
  - Read new dataframe and perform various operations on a dataframe

This completes our module

Congratulations!