365 DataScience

R PROGRAMMING

Course notes

R fundamentals | Data types | Vectors | Matrices | Data frames

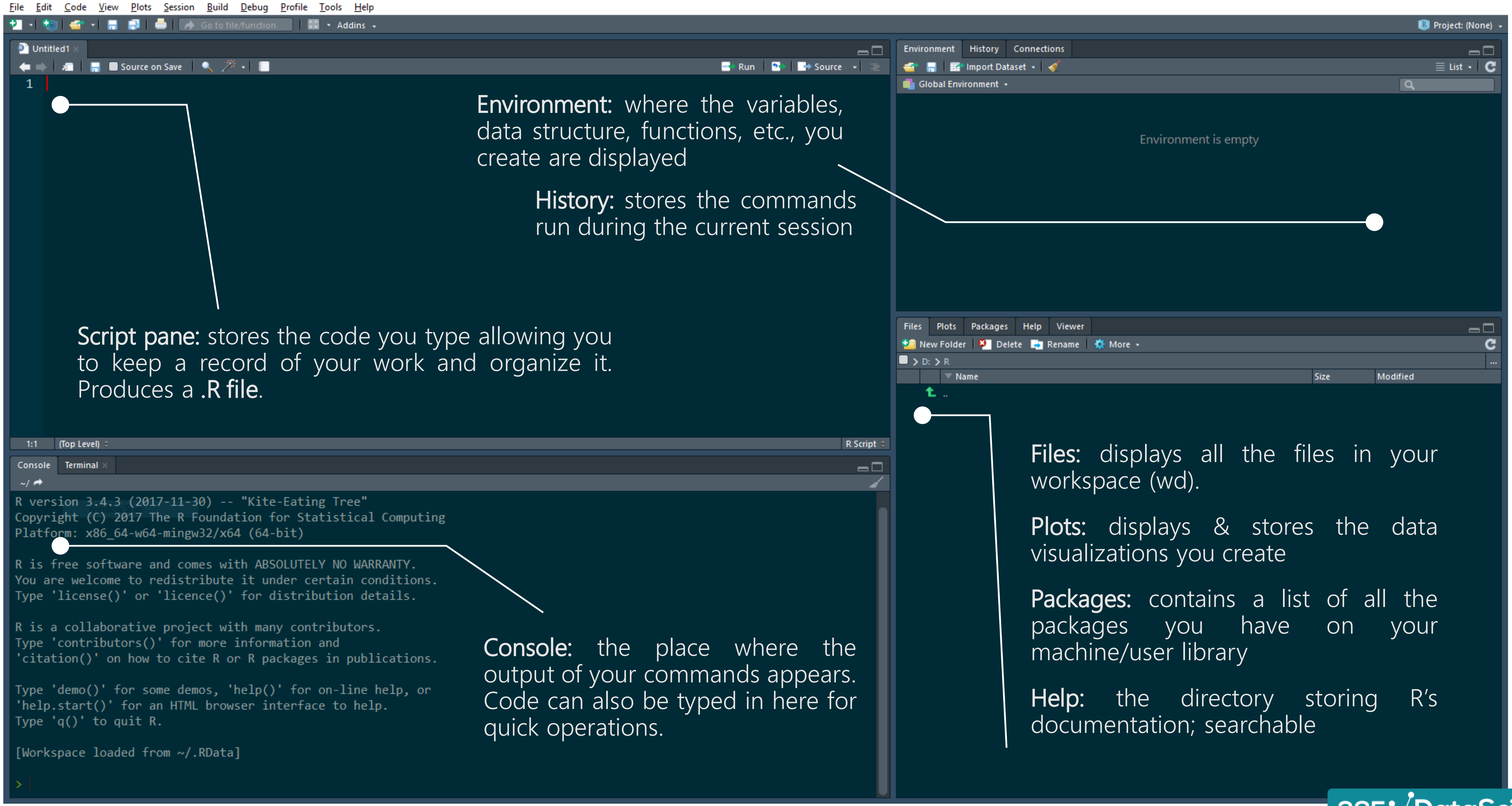
Words of welcome

You are here because you want to work better with data. Becoming comfortable with using a programming language for your data manipulations is a gigantic step forward. This way you will be able to automate, reproduce, and communicate your manipulations faster and better.

R is (perhaps) the best language to start with: it is an open-source language that is heavily tailored for handling data: from data wrangling through data visualization to machine learning. R's functionality is huge and, kind of like our universe, R's universe is always expanding.

Packages are the main source of functionality in R: they comprise different families of functions intended for a specific manipulation or field. The majority of R's packages are stored on the CRAN project website https://cran.r-project.org/, but there are CRAN-external sources, too.

The RStudio interface



Environment: where the variables, data structure, functions, etc., you create are displayed

History: stores the commands run during the current session

Script pane: stores the code you type allowing you to keep a record of your work and organize it. Produces a **.R file**.

Console: the place where the output of your commands appears. Code can also be typed in here for quick operations.

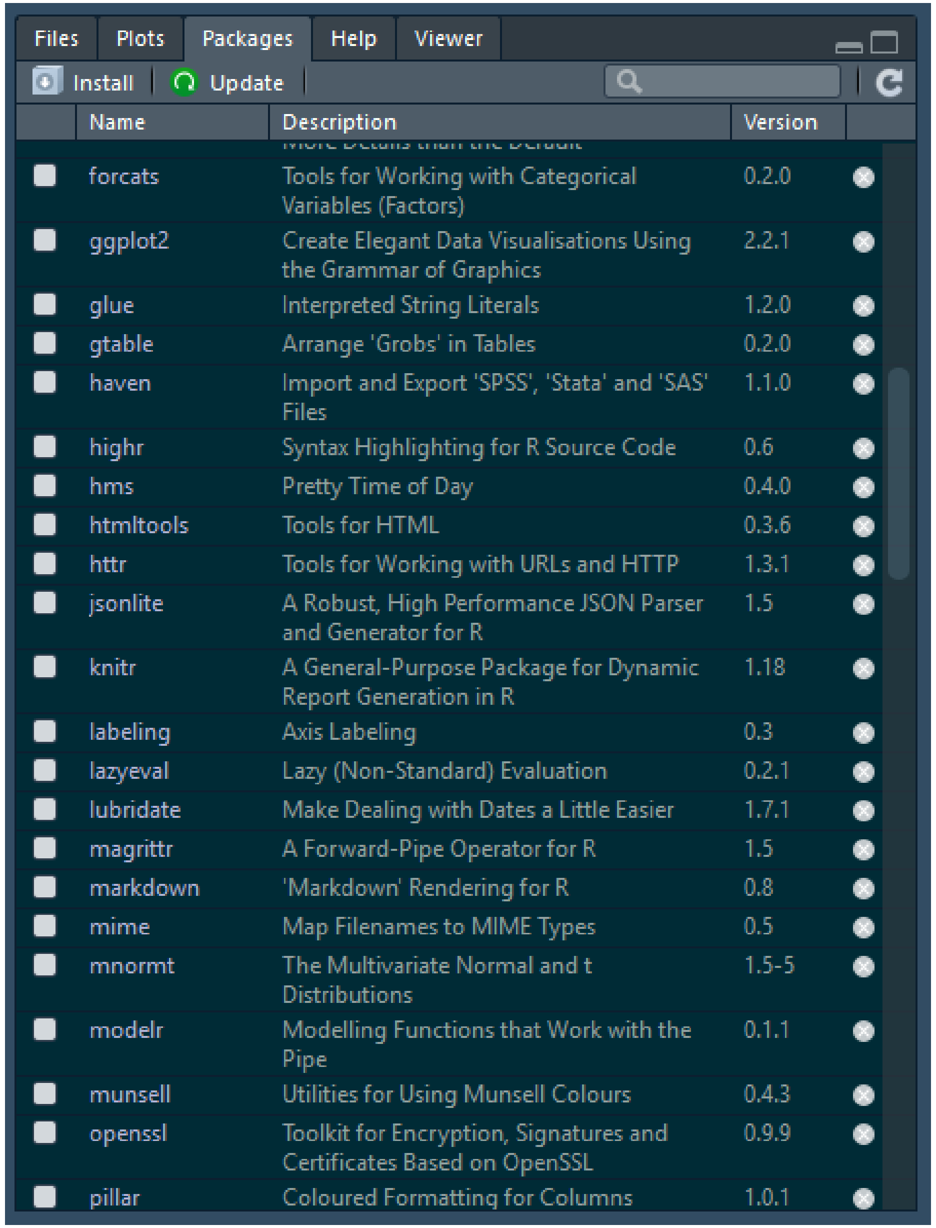
Files: displays all the files in your workspace (wd).

Plots: displays & stores the data visualizations you create

Packages: contains a list of all the packages you have on your machine/user library

Help: the directory storing R's documentation; searchable

Packages



Preloaded packages: R has a decent functionality before installing new packages. However, R's {base} packages are often outdated, or cumbersome to use.

Installing packages: installing a package saves the package in our User Library for future use. A package needs to be installed only once.

Loading packages: If you want to use the functions in a specific package, you must tell R to load it for you. You only need to load a package once per session. You do that with the `library(package.name)` function.

Distinguishing between functions: some packages have functions whose names overlap. You can recognise which package a function belongs to by checking the `{package.name}` next to it during autocompletion.

Removing packages: you can uninstall a package with the `remove.packages("package.name")` function.

Creating an object

Objects are **named data structures** inside of which you store data. An object can be a single digit, a character, a Boolean value, a sentence, a data frame, a list of data frames, a multi-dimensional structure, and so on. You use functions to create objects.

```
x <- y()
```

the object x

is created from

the function y

The object name:

- Must begin with a small letter.
- Longer names can be created by separating the individual words with a dot (.), an underscore (_), or by capitalizing the first letter of every new word (objectName).
- Once you select a notation, stick with it

Once you store data inside an object, you can use the object name to call that data and do operations with it. An operation will be carried out on each element of the data structure, systematically.

Data types

Integer

An *integer* is a **whole number**; any number that doesn't need anything after the decimal point is an integer.

R doesn't usually jump to creating an integer vector when you pass numbers. Often, you need to **explicitly tell** it to do so.

You do that by passing the letter `L` after each number in the integer object you're creating.

Double

Doubles store **regular numbers**: they can be large, small, positive, negative, with digits after the decimal or without.

Because R is heavily used for data analysis and most operations typically either involve or result in a double, this is **R's default way of saving numerical data**.

Character

Character vectors* store **text data**. A character element can be a single letter, number, or a symbol, or a longer string, like a sentence or even a paragraph.

To create a character string you must pass the value you want stored as a string in **quotation marks**.

Logical

Logical vectors store **Boolean data**; these are TRUE and FALSE values.

TRUE and FALSE and T and F can be used interchangeably as R recognises both. It is better, however, to use TRUE and FALSE because these are the reserved words which cannot be overwritten by the user.

TRUE and FALSE values must be inputted in **capital letters**, and without quotation marks.

`typeof(object.name)` returns the basic data type of the object you pass

`is.integer(object.name)` returns a Boolean signifying whether the object you pass is an integer or not; an analogous command exists for the other data types, too

Vector - a sequence of data elements that are of the **same type**

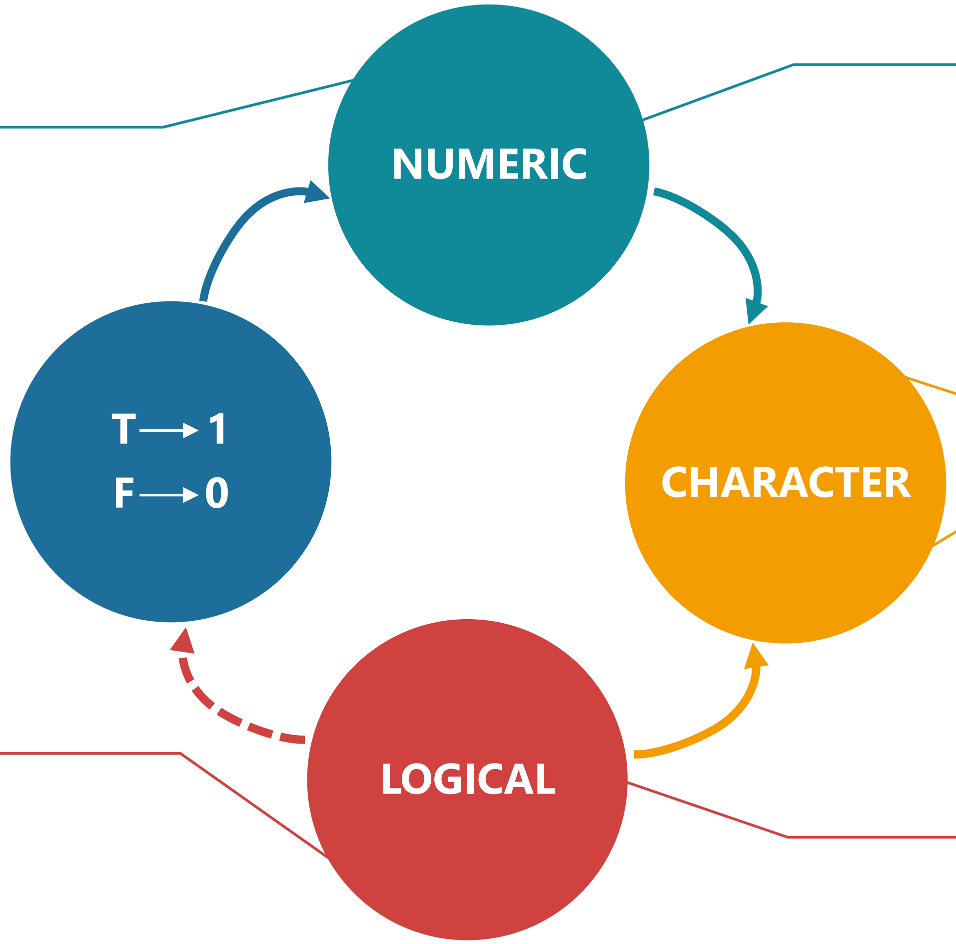
Coercion rules

R has ways to prevent certain mistakes from happening. For example, if you are trying to create an object, and you are passing the wrong type of data as an argument, then R will **convert** the value to the correct type, so you can end up creating your object. The *correct type* is typically the simplest type necessary to represent all the information.

When a **numeric** and a **logical** element are present together, the logical value is converted to a **numerical** one.

TRUE is encoded as 1

FALSE is encoded as 0



When a **numeric** and a **character** value are present together, the numeric value is converted to a **character**.

When a **logical** and a **character** value are present together, the logical element is coerced to a **character**.

Functions and arguments

Think of a function as any other type of object in R. It has a name, stores information (a set of statements organized in a particular way so as to perform a specific task), and can be called on when needed.

```
function.name(argument1 = , argument2 = , … argumentN = NULL)
```

some arguments can have default values (e.g., argument = NULL)

```
function.name <- function(){
# body of code
return(value)
}
```

creates the function

the statements the function executes when called

what the function returns

Basics

- **Functions take arguments** – the arguments of a function can be data, additional instructions on how to carry out the operation, other functions (called **nesting**)
- **To run a function**, call the function and pass in parentheses the data you want the function to operate on; this is an argument
- **To save the result of a function** into an object for further use, use the object-creating formula (`object <- function(data)`)
- **The arguments of a function have an inherent order** when passing values you can either explicitly specify the argument they are for (and thus not follow the inherent order), or you can omit the argument name but keep to the inherent argument order.

Intermediate

- A function is comprised of three parts: the **function name**, the **body of code** (containing the statements), and **arguments**;
- Sometimes a fourth component can be included in the basic structure of a function – the **return value**; it specifies what the function returns once executed and is written last in the body

Vector operations

Vector – a sequence of data elements that are of the same type

Vector operations happen in an element-wise fashion

Vector recycling happens when if the two vectors you are doing operations with are different in length; in that case, R repeats the shorter vector until it matches the length of the longer one.

Example: operation with same-length vectors

v3 <- v1 × v2

Multiplying two vectors happens element-by-element

v1		v2		v3
2	×	1	=	2
4		3		12
6		5		30
8		7		56

Note that the first element of v1 is multiplied by the first element of v2, the element of v2, the second element of v1, by the second element of v2, and so on.

The resulting v3 is the **same length** as v1 and v2.

Example: vector recycling

v3 <- v1 × v2

R repeats the shorter vector (note that **the vector is not then saved this way**!)

v1		v2		v1		v2		v3
2	×	1	=	2	×	1	=	2
4		3		4		3		12
		5		2		5		10
		7		4		7		28

The resulting v3 is the **same length as the longer vector**, v2.

If the shorter vector is not a multiple of the longer one, R will issue a warning, but it will still carry out the operation.

Vector attributes

Attributes are *additional information* about an object stored in the object; they do not affect the values of the object, instead they provide us with extra functionality if a function is designed to take into account whether or not an object has a specific attribute

`attributes(object.name)` checks whether the object has any attributes; an output `NULL` means the attributes object is empty

Names

You can give each element in a vector a name value.
When printed, R will display the values with their names above them.

```
George   John   Paul  Ringo
    23     26     24     26
>
```

Assign names by creating a character vector and using the `names(object) <- c("name.1", "name.2" … "name.k")` command

Use the same command to **change the names** if needed

Remove names by setting the names values to null
`names(object) <- NULL`

Dimensions

Changing the dimensions of a vector enables you to transform your 1-dimensional vector into an as-many-dimensional object you want.

You can create a 2-dimensional matrix from a vector by specifying row and column values in the dim() attribute.

```
dim(object) <- c(3, 4)
```

The **row value comes first**, followed by the columns value.

Example:

```
> x
 [1]  1  2  3  4  5  6  7  8  9 10 11 12
> dim(x) <- c(3, 4)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
>
```

Notice that R fills out the matrix **column by column**

365√DataScience

Vector slicing and recycling

- Indexing refers to selecting and extracting a single element from a structure (in this case, a vector)
- Slicing is to select and extract a sequence of elements
- R's notation for selecting a value is `object[i]`, where `i` is the index to which your value corresponds

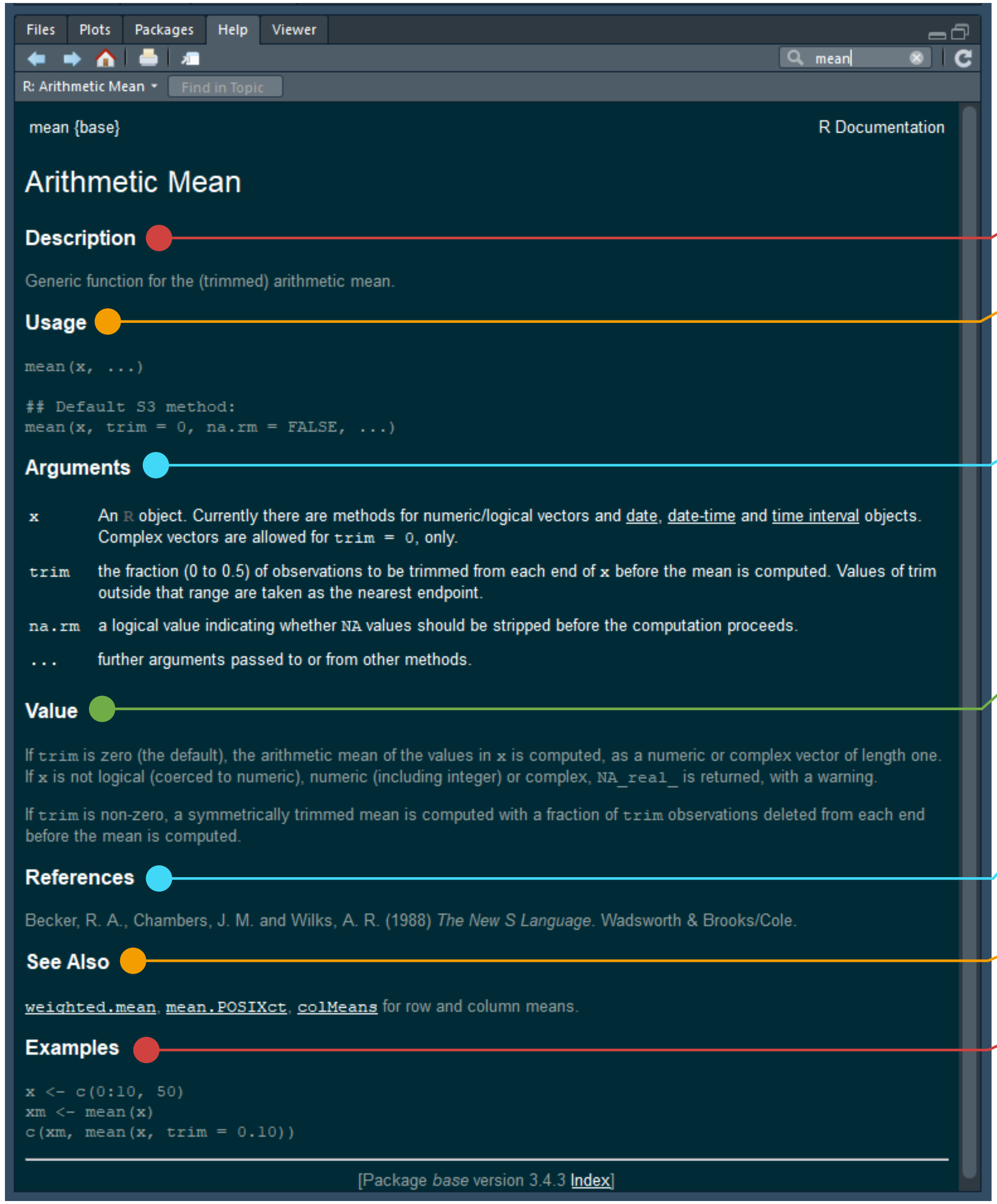
name:	a	b	c	d	e	f	g	h	i
	11	**13**	**15**	**17**	**19**	**21**	**23**	**25**	**27**
i:	1	2	3	4	5	6	7	8	9

`x[2]` `x["b"]` `x[3:5]` `x[c(1,7,9)]` `x[-c(1:5,7,9)]` `x[x>13 & x<21]`

Note that indexing with a **negative** integer works best if you are keeping most of the values and only want to get rid of some. You can also subset using TRUE and FALSE values, but that isn't very useful at this stage of learning. For the curious ones, it looks like this:

```
x <- 1:5          x[c(TRUE, FALSE, TRUE, TRUE, FALSE)]          ## 1 3 4
```

Getting HELP with R



Description: gives you a brief overview of what a function does

Usage: an example of how the function is written; if the function has many arguments, here you will see the inherent argument order

Arguments: gives you the complete list of arguments a function can take, and the kind of information R needs for each. It also tells you what that information will be used for

Value: an explanation of what the function will return when you execute it; this is essentially what your output will be

Details/References: with more complex functions this is where the author of the function may decide to draw your attention to some specifics of the function you may want to know for highly specific uses

See also: A useful pointer to related functions

Examples: Example code of the function in action; **guaranteed to work**

Use `?function` or `help(function)` to call help on a function
Use `??keyword` to call general help on something you want to do

Creating a matrix

```r
matrix(seq(2, 70, 2), nrow = 5, byrow = TRUE,
       dimnames = list(c("r1", "r2", "r3", "r4", "r5"),
                       c("a", "b", "c", "d", "e", "f", "g")))
```

If `nrow =` is specified, R infers what `ncol =` should be, and vice versa

	a	b	c	d	e	f	g
r1	2	4	6	8	10	12	14
r2	16	18	20	22	24	26	28
r3	30	32	34	36	38	40	42
r4	44	46	48	50	52	54	56
r5	58	60	62	64	66	68	70

named columns

named rows

`byrow = TRUE` fills out the matrix row-by-row

- Matrices are a natural extension to vectors: while vectors are 1-dimensional collections of data, matrices are **two-dimensional arrays**
- Matrices have a **fixed number of rows and columns**
- Matrices can contain only **one basic data type**
- You can **create** a matrix in the following ways:
  - ✓ `dim(x) <- c(i, j)`, where `i` and `j` are the values for rows and columns, respectively
  - ✓ `array(data, dim = c(i, j))`
  - ✓ `matrix(data, nrow =, ncol = , byrow = , dimnames = )`
    the `byrow =` argument specifies whether the matrix ought to be filled out column-by-column or row-by-row
- **Naming the dimensions** of a matrix happens in two ways:
  - ✓ With the `rownames()` and `colnames()` functions
  - ✓ By defining the `dimnames =` argument in the `matrix()` function

Matrix operations

Just like with vectors, matrix operations happen in **element-wise** fashion

Scaling is when you do an arithmetic operation on a matrix with a single value; it happens on all the values in the matrix and effectively standardizes it

To do **arithmetic operations** with matrices, they must be of the **same size** (rows x columns)

Example: operation with same-length matrices

```
m3 <- m1 + m2
```

Adding two matrices together happens element-by-element

m1

2	4	6	8
16	18	20	22
30	32	34	36
44	46	48	50

+

m2

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

=

m3

3	9	15	21
18	24	30	36
33	39	45	51
48	54	60	66

Note that to do inner and outer matrix multiplication (linear algebra), you need to specify this to R

`m1 %*% m2`
creates the product of inner multiplication

`m1 %o% m2`
creates the product of outer multiplication

Use `t()` to transpose a matrix if needed

Recycling with matrix operations

Example: creating a matrix from a shorter vector

m1 <- v1

$$\underset{v1}{\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}} \longrightarrow \underset{m1}{\begin{bmatrix} 1 & 4 & 3 & 2 \\ 2 & 1 & 4 & 3 \\ 3 & 2 & 1 & 4 \end{bmatrix}}$$

```
matrix(1:4, nrow = 3, ncol = 4)
```

Creating a matrix – you can create a matrix from a vector that does not have all the values needed to fill out the dimensions you specify, because R will recycle the vector to match the desired length

Vector × matrix operations – you can do operations with a matrix and a vector, and if the vector has fewer values than the matrix, R will again recycle

Example: multiplying a matrix with a vector

m2 <- m1 × v1

$$\underset{v1}{\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}} \times \underset{m1}{\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}} = \underset{v1}{\begin{bmatrix} 1 & 4 & 3 \\ 2 & 1 & 4 \\ 3 & 2 & 1 \end{bmatrix}} \times \underset{m1}{\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}} = \underset{m2}{\begin{bmatrix} 1 & 16 & 21 \\ 4 & 5 & 32 \\ 9 & 12 & 9 \end{bmatrix}}$$

Relational and Logical Operators

Relational

Relational operators (comparison operators), are for **evaluating R objects in relation to one another**.

- “==“ equal to;
- “!=“ not equal to;
- “<“ less than;
- “>“ more than;
- “<=“ less than or equal to;
- “>=“ more than or equal to.

Can be used with all data types and larger structures like matrices and data frames.

Logical

Logical operators, also called Boolean operators, are useful when you want to **combine the results of two or more comparisons**.

- AND “&” – if *all* comparison outcomes are TRUE, the result is TRUE (an exclusive operator);
- OR “|” – if there is a TRUE value *anywhere* in the expression, R returns TRUE (an inclusive operator);
- NOT “!” – flips the result of a logical test.

Keep in mind!

- A single equals sign “=“ is not a relational operator; this is used to **assign information** to an object;
- **%in%** operator – tests if an object is in a group of objects;
- double AND “&&” – examines only the first element of each vector; instead of doing a logical test on all available data, it does a single logical test, and outputs a single value.

Comparing a single value to a vector: R compares vectors in an element-by-element fashion; if you pass a vector in your command, the output is a vector of logical values that has the length of the vector you’ve passed.

```
> 12 > c(11, 11, 13, 14)
[1]  TRUE  TRUE FALSE FALSE
> "catch" == c("catch", 22, "is", "fantastic")
[1]  TRUE FALSE FALSE FALSE
```

Comparing two vectors: R compares the first element of the vector on the right with the first element of the vector on the left, then it proceeds to the second elements, and so on.

```
> c(11, 12, 13) >= c(10, 12, 14)
[1]  TRUE  TRUE FALSE
```

If and Else Statements

The general syntax of an **if else** statement:

```
if(A){
  Scenario 1
} else {
  Scenario 2
}
```

condition to be checked

command to be executed if the condition is met

instruction what to do if the condition is not met

command to be executed if the condition is not met

```r
v <- 8
if(v < 0){
  v <- v*-1
  print("I have made your object positive. Look:")
  print(v)
} else {
  print("Your object was already positive or zero, so I did nothing")
}
```

- If else statements build up on logical tests and allow the programmer to instruct a program to take action based on the outcome of a test (whether it is TRUE or FALSE). An if statement is an instruction to R to do something if a condition is met. An else statement is an instruction to R to do something if a condition is **not** met.
- An **if statement** begins with **if**, followed by the **condition** R should check in parentheses, and curly brackets that hold the **code** R must execute if the condition in the parentheses is TRUE;
- If an if statement's condition **does not** evaluate to TRUE, the program doesn't run anything and terminates, unless an else statement is defined;
- An **else statement** begins with else, followed by the code R must execute in curly brackets, in case the if condition is not met;

Else If Statements

The general syntax of an **else if** statement:

```
if(A){
  Scenario 1
} else if(B){
  Scenario 2
} else if(C){
  Scenario 3
} else{
  Scenario 4
}
```

condition to be checked

command to be executed if the condition is met

new condition to be checked in case the previous condition isn't met

command to be executed if the condition is met

new condition to be checked in case the previous condition isn't met

command to be executed if the condition is met

- Else if statements tell R to check if a different condition is met in case the previous one is not; typically used when there are more than two mutually exclusive cases or when you want to specify a special case in which R does something;
- There is no limit on the number of else if statements you can string.

Important

- All conditions must evaluate to a **single logical value**; it cannot be a vector of TRUEs and FALSEs. If it is a vector, R will only look at the first instance in the vector and execute accordingly;
- An if statement needs only a **single condition** to evaluate to TRUE in order to **stop** its search and execute the code for that condition. This is especially relevant for situations in which two or more conditions are not mutually exclusive and there could be more than one TRUE condition.

Creating a data frame

```r
name <- c("Flipper", "Bromley", "Nox", "Orion", "Dagger", "Zizi", "Carrie")
mo <- c(53, 19, 34, 41, 84, 140, 109)
size <- c("medium", "small", "medium", "large", "small", "extra small", "large")
weight <- c(21, 8, 4, 6, 7, 2, 36)
breed <- c("dog", "dog", "cat", "cat", "dog", "cat", "dog")

pets <- data.frame("Months old" = mo, "Size" = size, "Weight" = weight, "Breed" = breed,
                   stringsAsFactors = FALSE)
rownames(pets) <- name
```

	Months old	Size	Weight	Breed
Flipper	53	medium	21	dog
Bromley	19	small	8	dog
Nox	34	medium	4	cat
Orion	41	large	6	cat
Dagger	84	small	7	dog
Zizi	140	extra small	2	cat
Carrie	109	large	36	dog

one data type

named rows and columns

multiple data types

- Data frames are list structures that can store **variables of different basic types**, like numeric and string values
- **Data types** can differ **between**, but not within, columns
- A **row** can be comprised of cells with different data types
- The cells of a **column** (variable) can only be of a single data type
- You can **create** a data frame with the `data.frame()` function:
  - ✓ `my.df <- data.frame(var.a, var.b, var.c, stringsAsFactors = FALSE)`
  - ✓ the `stringsAsFactor =` argument specifies whether the character variables in your data should be coded as factors or not; it is often better to set it to FALSE, and encode the variables which are factors manually with the `as.factor()` function.
- **Naming the columns** of a data frame:
  - ✓ With the `names()` functions: this will name your **columns**
  - ✓ **`rownames()`** will name your rows if needed
  - ✓ By defining the `names` as you create your data frame
  - ✓ `my.df <- data.frame("A" = var.a, "B" = var.b)`

Importing and exporting data frames

Importing data is the primary way you will be using to get your data into R. You can import data from text files (CSV, tab-delimited), Excel, SAS, SPSS, Stata, and so on.

`read.table("file/path", sep = "", header = TRUE, …)` is the general-purpose family of functions for reading text data into R

Importing data

Importing a text file with **Comma Separated Values (CSV)** is done with the `read.csv()` function from the `read.table()` family

```
my.data <- read.csv("file.name", stringsAsFactors = FALSE)

my.data <- read.csv2("file.name", stringsAsFactors = FALSE)
```

read.csv2() imports data saved as a CSV but where the decimals are denoted with a comma (π = 3,14) instead of a dot (π = 3.14)

```
my.data <- read.delim("file.name", stringsAsFactors = FALSE)

my.data <- read.delim2("file.name", stringsAsFactors = FALSE)
```

`read.delim()` imports data in which the values are separated by a tab; there is a `read.delim2()` version of the call, too

Exporting data

Exporting a data frame out of R to share it with others is done with the `write.table()` function in a much similar way to importing data.

`write.csv()` exports a data frame as a CSV text file, with a `write.csv2()` for European users

```
write.csv(my.data, file = "file.name", row.names = FALSE)

write.csv2(my.data, file = "file.name", row.names = FALSE)

write.table(my.data, file = "file.name", sep = "\t", row.names = FALSE)
```

`write.table()` with a `sep = "\t"` argument exports a data frame as a tab-delimited text file

The `row.names =` argument is best set to `FALSE`. The default `TRUE` creates a redundant column of numbers in the beginning of your data frame, if your rows aren't already named.