Lecture 2 - Key datatypes & operators in R

Contents

Lecture learning objectives:

By the end of this lecture and worksheet 2, students should be able to:

Skip to main content

- Explain how the assignment symbol, `<-` differs from `=` in R
- Create in R, and define and differentiate in English, the below listed key datatypes in R:
  - logical, numeric, character and factor vectors
  - lists
  - data frames and tibbles
- Use R to determine the type and structure of an object
- Explain the distinction between names and values, and when R will copy an object.
- Use the three subsetting operators, `[[`, `[`, and `$`, to subset single and multiple elements from vectors and data frames, lists and matrices
- Compute numeric and boolean values using their respective types and operations

Getting help in R

No one, even experienced, professional programmers remember what every function does, nor do they remember every possible function argument/option. So both experienced and new programmers (like you!) need to look things up, A LOT!

One of the most efficient places to look for help on how a function works is the R help files. Let's say we wanted to pull up the help file for the `max()` function. We can do this by typing a question mark in front of the function we want to know more about.

```
?max
```

At the very top of the file, you will see the function itself and the package it is in (in this case, it is base). Next is a description of what the function does. You'll find that the most helpful sections on this page are "Usage", "Arguments" and "Examples"

Skip to main content

- **Usage** gives you an idea of how you would use the function when coding–what the syntax would be and how the function itself is structured.
- **Arguments** tells you the different parts that can be added to the function to make it more simple or more complicated. Often the "Usage" and "Arguments" sections don't provide you with step by step instructions, because there are so many different ways that a person can incorporate a function into their code. Instead, they provide users with a general understanding as to what the function could do and parts that could be added. At the end of the day, the user must interpret the help file and figure out how best to use the functions and which parts are most important to include for their particular task.
- The **Examples** section is often the most useful part of the help file as it shows how a function could be used with real data. It provides a skeleton code that the users can work off of.

Below is a useful graphical summary of the help docs that might be useful to start getting you oriented to them:

Skip to main content

The name of the function, and the library it is in.

mean {base} R Documentation

Arithmetic Mean

Description

What it does.

Generic function for the (trimmed) arithmetic mean.

Usage

The function's name, and in the parentheses the named arguments it expects, in the order it expects them. If an argument has a default value, it is shown. Arguments without default values (e.g. x) must be provided by you.

```
mean(x, ...)

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

Arguments

More details on each named argument. This will tell you what class of thing each argument has to be—an object, a number, a data frame, a logical value, etc.

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.

The ellipsis allows other arguments to be passed to and from the function.

Value

What the function returns—i.e., the result of whatever operation or calculation it performs. This can be a single number, as here, or a multi-part object such as a list, a data frame, a plot, or a model.

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.

Other related functions

Examples

Skip to main content

```
c(xm, mean(x, trim = 0.10))
```

built-in datasets or other R functions.

[Package *base* version 3.4.3 Index]

Visit the package's Index page to look for Demos and Vignettes detailing how it works.

Source: https://socviz.co/appendix.html#a-little-more-about-r

The assignment symbol, `<-`

- R came from S, S used `<-`
- S was inspired from APL, which also used `<-`
- APL was designed on a specific keyboard, which had a key for `<-`
- At that time there was no `==` for testing equality, it was tested with `=`, so something else need to be used for assignment.

source: https://colinfay.me/r-assignment/

- Nowadays, `=` can also be used for assignment, however there are some things to be aware of...
- stylistically, `<-` is preferred over `=` for readability
- `<-` and `->` are valid in R, the latter can be useful in pipelines (more on this in data wrangling)

Skip to main content

- **we expect you to use `<-` in MDS for object assignment in R**

Assignment readability

Consider this code:

```
c <- 12
d <- 13
```

Which equality is easier to read?

```
e = c == d
```

or

```
e <- c == d
```

Assignment environment

What value does x hold at the end of each of these code chunks?

```
median(x = 1:10)
```

vs

```
median(x <- 1:10)
```

Skip to main content

Here, in the first example where `=` is used to set `x`, `x` only exists in the `median` function call, so we are returned the result from that function call, however, when we call `x` later, it does not exist and so R returns an error.

```
median(x = 1:10)
x
```

```
5.5
Error in eval(expr, envir, enclos): object 'x' not found
Traceback:
```

Here, in the second example where `<-` is used to set `x`, `x` exists in the median function call, **and** in the global environment (outside the `median` function call). So when we call `x` later, it **does** exist and so R returns the value that the name `x` is bound to.

```
median(x <- 1:10)
x
```

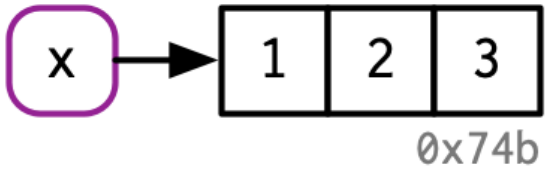
```
5.5
1 2 3 4 5 6 7 8 9 10
```

What does assignment do in R?

When you type this into R: `x <- c(1, 2, 3)`

This is what R does:

Skip to main content



Source: Advanced R by Hadley Wickham

What does this mean? It means that even if you don't bind a name to an object in R using `<-`, it still exists somewhere in memory during the R session it was created in. This is typically not a problem unless your data sets are very large.

A note on names

Rules for syntactic names:

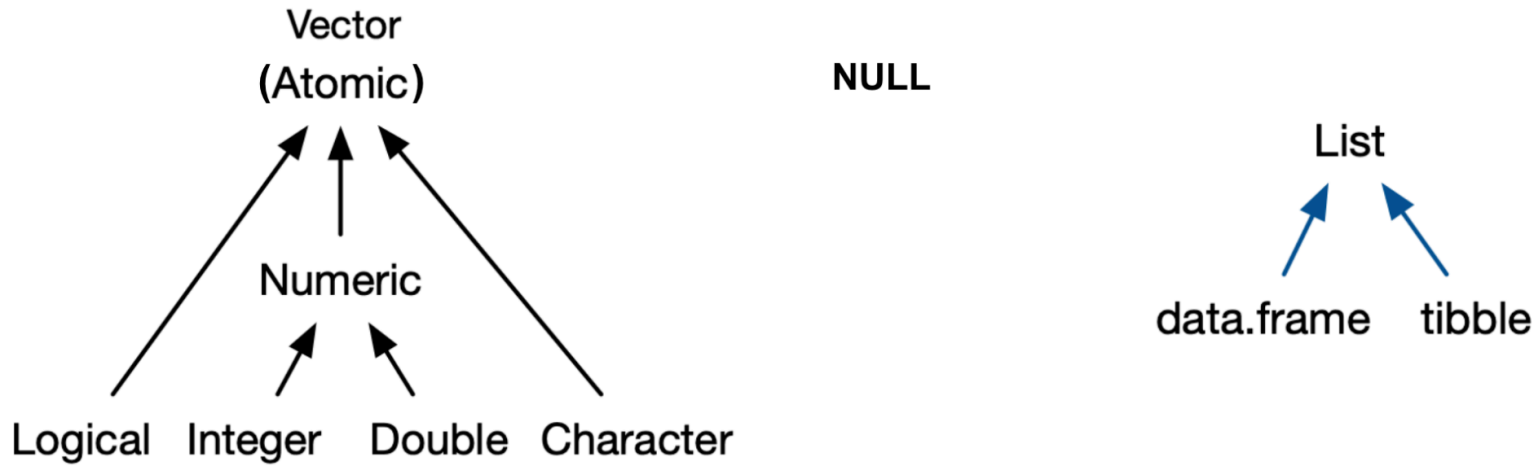
- May use: letters, digits, `.` and `_`
- Cannot begin with `_` or a digit
- Cannot use reserved words (e.g., `for`, `if`, `return`)

How to manage non-syntactic names

- Usually come across these when reading in someone else's data
- Backticks, `` ` ``, can be used manage these cases (e.g., `` `_abc` <- 1 ``)
- If your data contains these, use R to rename things to make them syntactic (for your future sanity)

Skip to main content

Key datatypes in R



note - There are no scalars in R, they are represented by vectors of length 1.

Source: Advanced R by Hadley Wickham

- `NULL` is not a vector, but related and frequently functions in the role of a generic zero length vector.

What is a data frame?

From a data perspective, it is a rectangle where the rows are the observations and the columns are variables:

https://github.com/UBC-DSCI/introduction-to-datascience/blob/main/img/wrangling/data_frame_slides_cdn.004.jpeg?raw=true

Skip to main content

What is a data frame?

From a computer programming perspective, in R, a data frame is a special subtype of a list object whose elements (columns) are vectors.


https://github.com/UBC-DSCI/introduction-to-datascience/blob/main/img/wrangling/data_frame_slides_cdn.005.jpeg?raw=true

Question: What do you notice about the elements of each of the vectors in this data frame?

What is a vector?

- objects that can contain 1 or more elements
- elements are ordered
- must all be of the same type (e.g., double, integer, character, logical)


https://github.com/UBC-DSCI/introduction-to-datascience/blob/main/img/wrangling/data_frame_slides_cdn.007.jpeg?raw=true

How are vectors different from a list?


https://github.com/UBC-DSCI/introduction-to-datascience/blob/main/img/wrangling/data_frame_slides_cdn.008.jpeg?raw=true

Reminder: what do lists have to do with data frames?

Skip to main content

A bit more about Vectors

Your closest and most important friend in R

Creating vectors and vector types

```
char_vec <- c("joy", "peace", "help", "fun", "sharing")
char_vec
typeof(char_vec)
```

Skip to main content

'joy' · 'peace' · 'help' · 'fun' · 'sharing'

'character'

```
log_vec <- c(TRUE, TRUE, FALSE, FALSE, TRUE)
log_vec
typeof(log_vec)
```

TRUE · TRUE · FALSE · FALSE · TRUE

'logical'

```
double_vec <- c(1, 2, 3, 4, 5)
double_vec
typeof(double_vec)
```

1 · 2 · 3 · 4 · 5

'double'

```
int_vec <- c(1L, 2L, 3L, 4L, 5L)
int_vec
typeof(int_vec)
```

1 · 2 · 3 · 4 · 5

'integer'

`str` is a useful command to get even more information about an object:

```
str(int_vec)
```

Skip to main content

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

What happens to vectors of mixed type?

```
mixed_vec <- c("joy", 5.6, TRUE, 1L, "sharing")
typeof(mixed_vec)
```

'character'

Hierarchy for coercion:

character → double → integer → logical

Useful functions for testing type and forcing coercion:

- `is.logical()`, `is.integer()`, `is.double()`, and `is.character()` returns `TRUE` or `FALSE`, depending on type of object and function used.
- `as.logical()`, `as.integer()`, `as.double()`, or `as.character()` coerce vector to type specified by function name.

Skip to main content

How to subset and modify vectors

Subsetting

- R counts from 1!!!

```
name <- c("T", "i", "f", "f", "a", "n", "y")
```

What letter will I get in R? What would I get in Python?

Skip to main content

```
name[2]
```

'i'

What letters will I get in R? What would I get in Python?

```
name <- c("T", "i", "f", "f", "a", "n", "y")
```

```
name[2:4]
```

'i' · 'f' · 'f'

What letter will I get in R? What would I get in Python?

```
name <- c("T", "i", "f", "f", "a", "n", "y")
```

```
name[-1]
```

'i' · 'f' · 'f' · 'a' · 'n' · 'y'

How do I get the last element in a vector in R?

```
name <- c("T", "i", "f", "f", "a", "n", "y")
```

```
name[length(name)]
```

Skip to main content

'y'

Modifing vectors

We can combine the assignment symbol and subsetting to modify vectors:

```
name <- c("T", "i", "f", "f", "a", "n", "y")
```

```
name[1] <- "t"
name
```

't' · 'i' · 'f' · 'f' · 'a' · 'n' · 'y'

This can be done for more than one element:

```
name[1:3] <- c("T", "I", "F")
name
```

'T' · 'I' · 'F' · 'f' · 'a' · 'n' · 'y'

What if you ask for elements that are not there?

```
name[8:12]
```

NA · NA · NA · NA · NA

This syntax also lets you add additional elements:

Skip to main content

```
name[8:12] <- c("-", "A", "n", "n", "e")
name
```

'T' · 'l' · 'F' · 'f' · 'a' · 'n' · 'y' · '-' · 'A' · 'n' · 'n' · 'e'

What happens when you modify a vector in R?

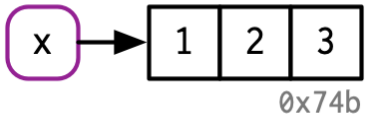
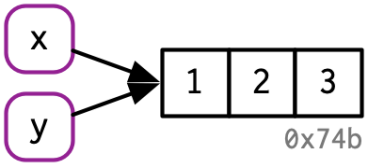
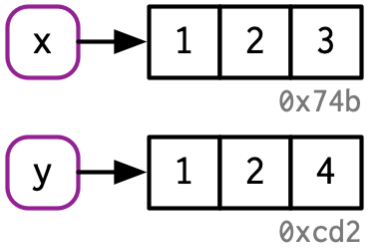
Consider:

```
x <- c(1, 2, 3)
y <- x

y[3] <- 4
y
#> [1] 1 2 4
```

What is happening in R's memory for each line of code?

Skip to main content

Code	R's memory representation
`x <- c(1, 2, 3)`	
`y <- x`	
`y[[3]] <- 4`	

This is called "copy-on-modify".

Source: Advanced R by Hadley Wickham

Why copy-on-modify

- Since there are no scalars in R, vectors are essentially immutable
- If you change one element of the vector, you have to copy the whole thing to update it

Skip to main content

Why do we care about knowing this?

- Given that data frames are built on-top of vectors, this has implications for speed when working with large data frames

Why vectors?

Vectorized operations!

```
c(1, 2, 3, 4) + c(1, 1, 1, 1)
```

2 · 3 · 4 · 5

But watch out for vector recycling in R!

This makes sense:

```
c(1, 2, 3, 4) + c(1)
```

2 · 3 · 4 · 5

but this does not!

```
c(1, 2, 3, 4) + c(1, 2)
```

2 · 4 · 4 · 6

Skip to main content

One to watch out for, logical and (`&`) and or (`|`) operators come in both an elementwise and first element comparison form, for example:

```
# compares each elements of each vector by position
c(TRUE, TRUE, TRUE) & c(FALSE, TRUE, TRUE)
```

FALSE · TRUE · TRUE

```
# compares only the first elements of each vector
c(TRUE, TRUE, TRUE) && c(FALSE, TRUE, TRUE)
```

```
Error in c(TRUE, TRUE, TRUE) && c(FALSE, TRUE, TRUE): 'length = 3' in coercion to 'logical(1)'
Traceback:
```

Skip to main content

Extending our knowledge to data frames

Vector	Vector	Vector
region	year	population
Toronto	2016	2235145
Vancouver	2016	1027613
Montreal	2016	1823281
Calgary	2016	544870
Ottawa	2016	571146
Winnipeg	2016	321484
Hamilton	2016	306034
Edmonton	2016	537634
Halifax	2016	187478
London	2016	220452
Victoria	2016	172559
St. John's	2016	92353
Saskatoon	2016	124766

Getting to know a data frame

Skip to main content

```
head(mtcars)
```

A data.frame: 6 × 11

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225	105	2.76	3.460	20.22	1	0	3	1

```
str(mtcars)
```

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

Skip to main content

Subsetting and modifying data frames

There are 3 operators that can be used when subsetting data frames: `[`, `$` and `[[`

Operator	Example use	What it returns
`[`	`mtcars[1:10, 2:4]`	rows 1-10 for columns 2-4 of the data frame, as a data frame
`[`	`mtcars[1:10, ]`	rows 1-10 for all columns of the data frame, as a data frame
`[`	`mtcars[1]`	the first column of the data frame, as a data frame
`[[`	`mtcars[[1]]`	the first column of the data frame, as a vector
`$`	`mtcars$cyl`	the column the corresponds to the name that follows the `$`, as a vector

Note that `$` and `[[` remove a level of structure from the data frame object (this happens with lists too).

Other R objects

We are focusing on vectors and data frames in this lecture because these are the objects you will encounter most frequently in R for data science. These subsetting (and modification) syntax also work on other objects in R, in the same way.

Examples that you will encounter in the worksheet and lab are matrices and lists.

Skip to main content

Logical indexing of data frames

We can also use logical statements to filter for rows containing certain values, or values above or below a threshold. For example, if we want to filter for rows where the cylinder value in the `cyl` column is 6 in the mtcars data frame shown below:

```
options(repr.matrix.max.rows = 10) # limit number of rows that are output
mtcars
```

A data.frame: 32 × 11

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.440	17.02	0	0	3	2
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.9	1	1	5	2
Ford Pantera L	15.8	8	351.0	264	4.22	3.170	14.5	0	1	5	4
Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.5	0	1	5	6
Maserati Bora	15.0	8	301.0	335	3.54	3.570	14.6	0	1	5	8
Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.6	1	1	4	2

Skip to main content

```
mtcars[mtcars$cyl == 6, ]
```

A data.frame: 7 × 11

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4
Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.50	0	1	5	6

Another example:

```
mtcars[mtcars$hp > 200, ]
```

Skip to main content

A data.frame: 7 × 11

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
Duster 360	14.3	8	360	245	3.21	3.570	15.84	0	0	3	4
Cadillac Fleetwood	10.4	8	472	205	2.93	5.250	17.98	0	0	3	4
Lincoln Continental	10.4	8	460	215	3.00	5.424	17.82	0	0	3	4
Chrysler Imperial	14.7	8	440	230	3.23	5.345	17.42	0	0	3	4
Camaro Z28	13.3	8	350	245	3.73	3.840	15.41	0	0	3	4
Ford Pantera L	15.8	8	351	264	4.22	3.170	14.50	0	1	5	4
Maserati Bora	15.0	8	301	335	3.54	3.570	14.60	0	1	5	8

Modifing data frames

Similar to vectors, we can combine the assignment symbol and subsetting to modify data frames.

For example, here we create a new column called `kml`:

```
mtcars$kml <- mtcars$mpg / 2.3521458
head(mtcars)
```

Skip to main content

A data.frame: 6 × 12

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb	kml
	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1	4	4	8.928018
Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1	4	4	8.928018
Datsun 710	22.8	4	108	93	3.85	2.320	18.61	1	1	4	1	9.693277
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1	9.098075
Hornet Sportabout	18.7	8	360	175	3.15	3.440	17.02	0	0	3	2	7.950187
Valiant	18.1	6	225	105	2.76	3.460	20.22	1	0	3	1	7.695101

The same syntax works to overwrite an existing column.

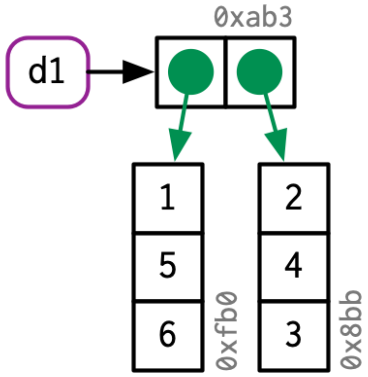
What happens when we modify an entire column? or a row?

To answer this we need to look at how data frames are represented in R's memory.

How R represents data frames:

- Remember that data frames are lists of vectors
- As such, they don't store the values themselves, they store references to them:

Skip to main content

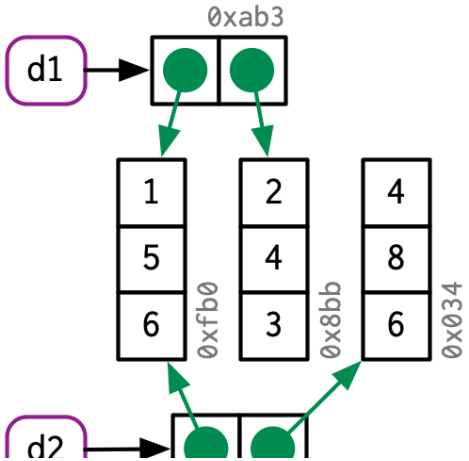


Source: Advanced R by Hadley Wickham

How R represents data frames:

If you modify a column, only that column needs to be modified; the others will still point to their original references:

```
d2 <- d1
d2[, 2] <- d2[, 2] * 2
```



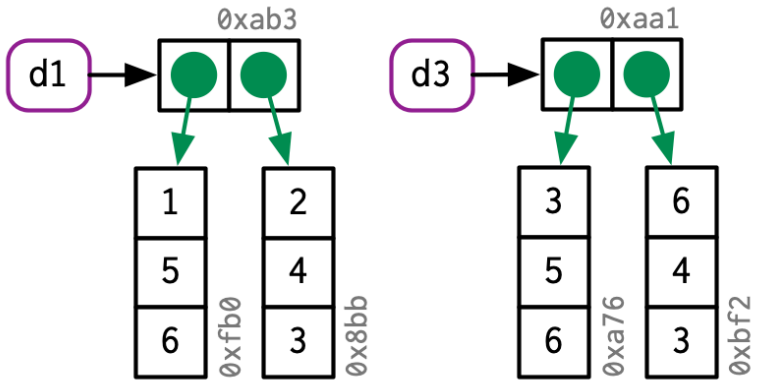
Skip to main content

Source: Advanced R by Hadley Wickham

How R represents data frames:

However, if you modify a row, every column is modified, which means every column must be copied:

```
d3 <- d1
d3[1, ] <- d3[1, ] * 3
```



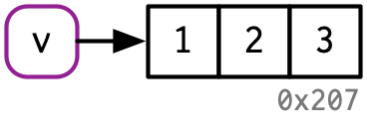
Source: Advanced R by Hadley Wickham

An exception to copy-on-modify

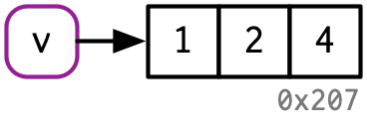
If an object has a single name bound to it, R will modify it in place:

```
v <- c(1, 2, 3)
```

Skip to main content




```
v[[3]] <- 4
```



- Hence, modify in place can be a useful optimization for speeding up code.
- However, there are some complications that make predicting exactly when R applies this optimisation challenging (see here for details)
- There is one other time R will do this, we will cover this when we get to environments.

Source: Advanced R by Hadley Wickham

Writing readable R code

- WriTing AND reading (code) TaKes cognitive RESOURCES, & We only hAvE so MUCh!
- To help free up cognitive capacity, we will follow the tidyverse style guide

Skip to main content

Sample code **not** in tidyverse style

Can we spot what's wrong?

```
library(tidyverse)
us.2015.econ=read_csv( "data/state_property_data.csv")
us.2016.vote=read_csv( "data/2016_presidential_election_state_vote.csv")
stateData=left_join (us.2015.econ,us.2016.vote) %>%
    filter(party!="Not Applicable") %>%
    mutate(meanCommuteHours=mean_commute_minutes/60)
ggplot(stateData, aes (x=mean_commute_minutes, y=med_prop_val, color=party)) +
    geom_point()+
    xlab( "Income (USD)" )+
    ylab("Median property value (USD)")+
    scale_colour_manual (values = c("blue","red"))+
    scale_x_continuous (labels = scales::dollar_format())+
    scale_y_continuous (labels = scales::dollar_format())
```

Skip to main content

Sample code in tidyverse style

```
library(tidyverse, quietly = TRUE)
us_2015_econ <- read_csv("data/state_property_data.csv")
us_2016_vote <- read_csv("data/2016_presidential_election_state_vote.csv")
state_data <- left_join(us_2015_econ, us_2016_vote) %>%
    filter(party != "Not Applicable") %>%
    mutate(mean_commute_hours = mean_commute_minutes / 60)
ggplot(state_data, aes(x = med_income, y = med_prop_val, color = party)) +
    geom_point() +
    xlab("Income (USD)") +
    ylab("Median property value (USD)") +
    scale_colour_manual(values = c("blue", "red")) +
    scale_x_continuous(labels = scales::dollar_format()) +
    scale_y_continuous(labels = scales::dollar_format())
```

What did we learn today?

- How to get help in R
- How the `<-` differs from `=` in R
- Base R syntax for subsetting and modifying R objects
- Some aspects of tidyverse code style

Additional resources:

- RStudio base R cheat sheet
- R Operators cheat sheet

Skip to main content

Attribution:

- Advanced R by Hadley Wickham
- Why do we use arrow as an assignment operator? by Colin Fay

Previous
Lecture 1 - Introduction to R via the tidyverse

Next
Lecture 3 - dates & times, strings, as well as factors