R for Reproducible Scientific Analysis: Data Structures

Supplemental Materials for Software Carpentry, Arizona State University

Martin Frigaard

July 6-8

Course Materials

Course website:

https://annajiat.github.io/2021-07-06-asu-online/

Software Carpentry

These materials accompany day 2, "R for Reproducible Scientific Analysis (Continued)"

Why R?

R is a versatile language for data wrangling, visualization, and modeling

Getting Started

Image credit: R Project

Installing R

Install R from the Comprehensive R Archive Network (CRAN):

https://cran.r-project.org/

CRAN
Mirrors
What's new?
Task Views
Search

About R
R Homepage
The R Journal

Software
R Sources
R Binaries
Packages
Other

Documentation
Manuals
FAQs
Contributed

The Comprehensive R Archive Network

Download and Install R

Precompiled binary distributions of the base system and contributed packages, **Windows and Mac** users most likely want one of these versions of R:

- Download R for Linux
- Download R for (Mac) OS X
- Download R for Windows

R is part of many Linux distributions, you should check with your Linux package management system in addition to the link above.

Source Code for all Platforms

Windows and Mac users most likely want to download the precompiled binaries listed in the upper box, not the source code. The sources have to be compiled before you can use them. If you do not know what this means, you probably do not want to do it!

- The latest release (2020-06-22, Taking Off Again) R-4.0.2.tar.gz, read what's new in the latest version.
- Sources of R alpha and beta releases (daily snapshots, created only in time periods before a planned release).
- Daily snapshots of current patched and development versions are available here. Please read about new features and bug fixes before filing corresponding feature requests or bug reports.
- Source code of older versions of R is available here.
- Contributed extension packages

Questions About R

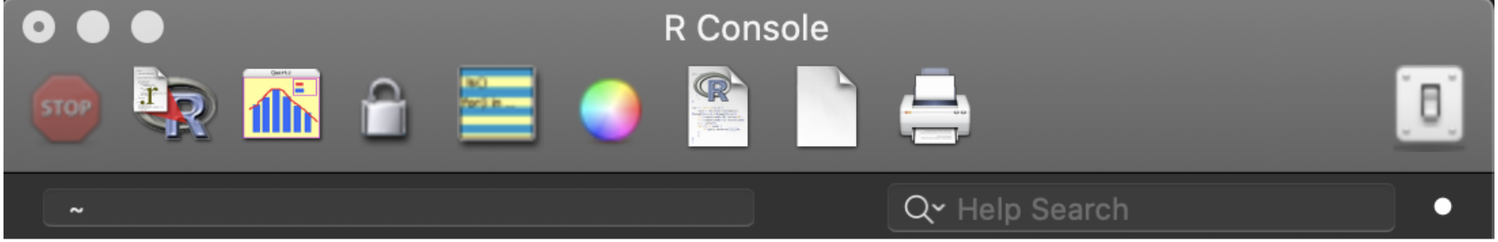
- If you have questions about R like how to download and install the software, or what the license terms are, please read our answers to frequently asked questions before you send an email.

You are recommended to use the RStudio IDE (*but you do not have to*).

Download RStudio

https://rstudio.com/products/rstudio/download/

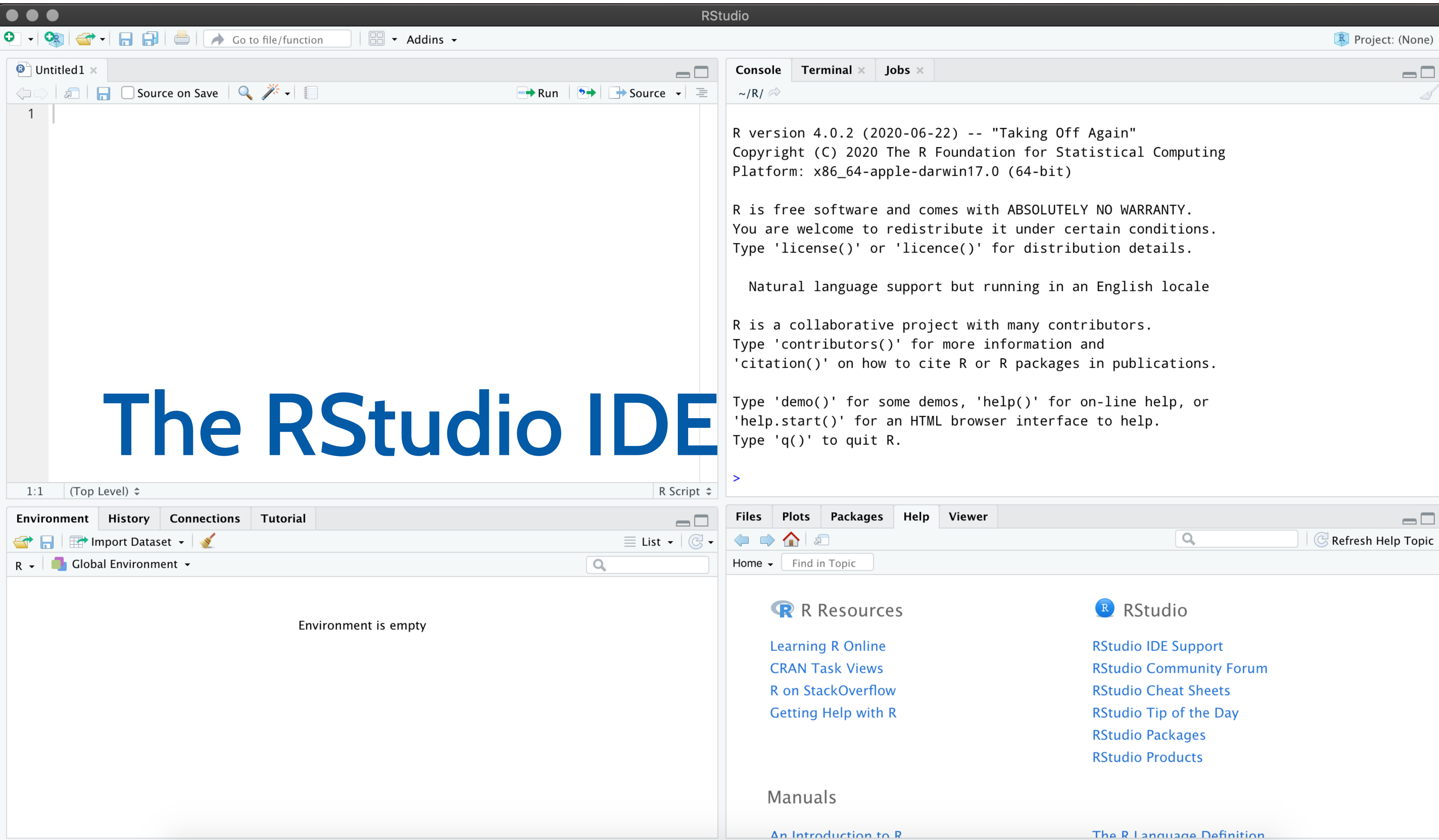
OS	Download	Size	SHA-256
Windows 10/8/7	RStudio-1.3.1093.exe	171.62 MB	62b9e60a
macOS 10.13+	RStudio-1.3.1093.dmg	148.66 MB	bdc4d3a4
Ubuntu 16	rstudio-1.3.1093-amd64.deb	124.33 MB	72f05048
Ubuntu 18/Debian 10	rstudio-1.3.1093-amd64.deb	126.80 MB	ff222177
Fedora 19/Red Hat 7	rstudio-1.3.1093-x86_64.rpm	146.96 MB	ed1f6ef8
Fedora 28/Red Hat 8	rstudio-1.3.1093-x86_64.rpm	151.05 MB	01a978f3
Debian 9	rstudio-1.3.1093-amd64.deb	127.00 MB	a747f9f9
SLES/OpenSUSE 12	rstudio-1.3.1093-x86_64.rpm	119.43 MB	5016cbcf
OpenSUSE 15	rstudio-1.3.1093-x86_64.rpm	128.40 MB	cf47e32d




```
R version 4.0.2 (2020-06-22) -- "Taking Off Again"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin17.0 (64-bit)

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

  Natural language support but running in an English locale

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

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

[R.app GUI 1.72 (7847) x86_64-apple-darwin17.0]

>
```

The R Console

The RStudio IDE

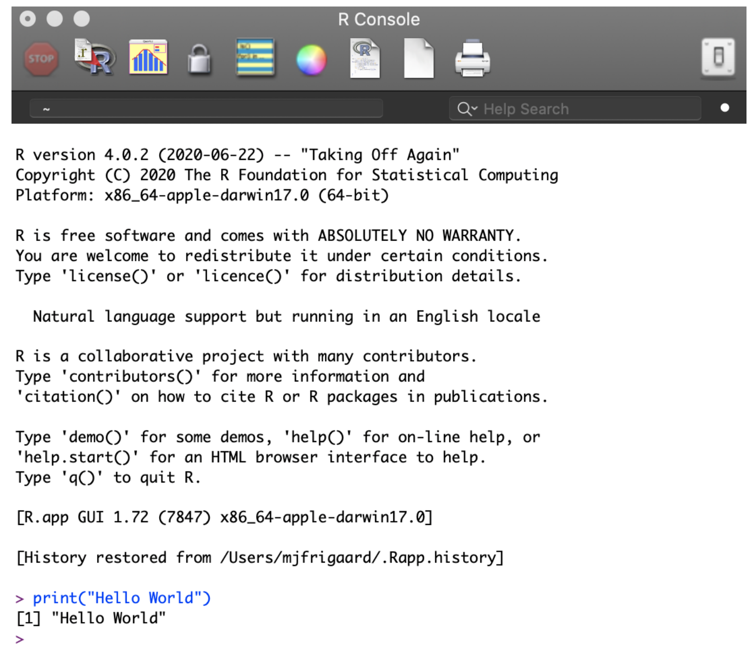


Running R Commands

You can run R commands in the Console by entering them after the > operator (see example in R below)

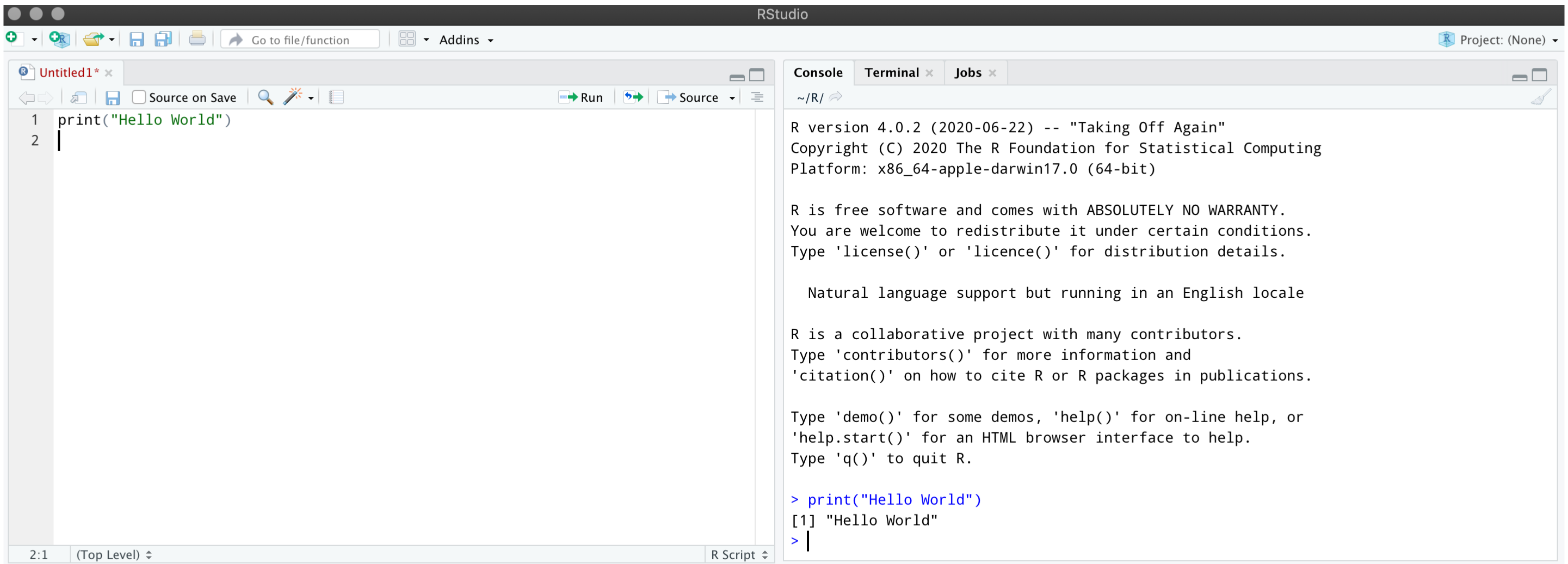
```r
print("Hello World")
```

```
## [1] "Hello World"
```



Running R Commands

You can also run them in R scripts (see example in RStudio below)



Objects

Objects

R is typically referred to as an "*object-oriented, functional programming*" language

Most things in R are either functions or objects (and "*functions do things to objects*")

Types of objects in R

- **Vectors**
  - atomic (logical, integer, double, and character)
  - S3 (factors, dates, date-times, durations)
- **Matrices**
  - two dimensional objects
- **Arrays**
  - multidimensional objects
- **Data frames & tibbles**
  - rectangular objects
- **Lists**
  - recursive objects

Atomic Vectors

Atomic Vectors

Vectors are the fundamental data type in R.

Many of R's functions are *vectorised*, which means they're designed for performing operations on vectors.

The "atomic" in atomic vectors means, "*of or forming a single irreducible unit or component in a larger system.*"

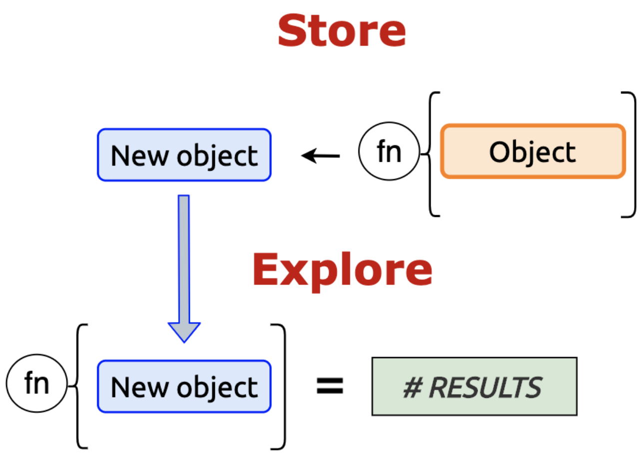
Atomic vectors can be logical, integer, double, or character (strings).

We will build each of these vectors using the previously covered assignment operator (`<-`) and `c()` function (*which stands for 'combine'*).

Store and Explore

A common practice in R is to create an object, perform an operation on that object with a function, and store the results in new object.

We then explore the contents of the new object with another function.



Many of the functions in R are written with this *store and explore* process in mind.

Atomic vectors: numeric

The two atomic numeric vectors are integer and double.

Integer vectors are created with a number and capital letter `L` (i.e. `1L`, `10L`)

```
vec_integer <- c(1L, 10L, 100L)
```

Double vectors can be entered as decimals, but they can also be created in scientific notation (`2.46e8`), or values determined by the floating point standard (`Inf`, `-Inf` and `NaN`).

```
vec_double <- c(0.1, 1.0, 10.01)
```

Atomic vectors: numeric

We will use the `typeof()` and `is.numeric()` functions to explore the contents of `vec_integer` and `vec_double`.

```
typeof(vec_integer)
```

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

```
is.numeric(vec_integer)
```

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

`typeof()` tells us that this is an `"integer"` vector, and `is.numeric()` tests to see if it is numeric (which is `TRUE`).

Atomic vectors: logical vectors

Logical vectors can be `TRUE` or `FALSE` (or `T` or `F` for short). Below we use `typeof()` and `is.logical()` to explore the contents of `vec_logical`.

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

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

```
is.logical(vec_logical)
```

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

Atomic vectors: logical vectors

Logical vectors are handy because when we add them together, and the total number tells us how many `TRUE` values there are.

```
TRUE + TRUE + FALSE + TRUE
```

```
## [1] 3
```

Logical vectors can be useful for subsetting (a way of extracting certain elements from a particular object) based on a set of conditions.

How many elements in `vec_integer` are greater than `5`?

```
vec_integer > 5
```

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

Atomic vectors: character vectors

Character vectors store text data (note the double quotes). We'll *store and explore* again.

```
vec_character <- c("A", "B", "C")
typeof(vec_character)
```

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

```
is.character(vec_character)
```

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

Character vectors typically store text information that we need to include in a calculation, visualization, or model. In these cases, we'll need to convert them into `factor`s. We'll cover those next.

S3 Vectors

S3 Vectors

S3 Vectors can be factors, dates, date-times, and difftimes.

Vectors with additional attributes:

- `"levels"`
- `"tzone"`
- `"names"`

S3 Vectors: factors

Factors are categorical vectors with a given set of responses. Below we create a factor with three levels: `low`, `medium`, and `high`

```
vec_factor <- factor(x = c("low", "medium", "high"))
class(vec_factor)
```

```
## [1] "factor"
```

Factors are not character variables, though. They get stored with an integer indicator for each character level.

```
typeof(vec_factor)
```

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

S3 Vectors: factor attributes

Factors are integer vectors with two additional attributes: `class` is set to `factor`, and `levels` for each unique response.

We can check this with `unique()` and `attributes()` functions.

```
unique(vec_factor)
```

```
## [1] low    medium high
## Levels: high low medium
```

```
attributes(vec_factor)
```

```
## $levels
## [1] "high"   "low"    "medium"
##
## $class
## [1] "factor"
```

S3 Vectors: factor attributes

Levels are assigned alphabetically, but we can manually assign the order of factor levels with the `levels` argument in `factor()`.

```
vec_factor <- factor(x = c("medium", "high", "low"),
                     levels = c("low", "medium", "high"))
```

We can check the levels with `levels()` or `unclass()`

```
levels(vec_factor)
```

```
## [1] "low"    "medium" "high"
```

```
unclass(vec_factor)
```

```
## [1] 2 3 1
## attr(,"levels")
## [1] "low"    "medium" "high"
```

S3 Vectors: date

Dates are stored as `double` vectors with a `class` attribute set to `Date`.

R has a function for getting today's date, `Sys.Date()`. We'll create a `vec_date` using `Sys.Date()` and adding `1` and `2` to this value.

```
vec_date <- c(Sys.Date(), Sys.Date() + 1, Sys.Date() + 2)
vec_date
```

```
## [1] "2021-07-07" "2021-07-08" "2021-07-09"
```

We can see adding units to the `Sys.Date()` added days to today's date.

The `attributes()` function tells us this vector has it's own class.

```
attributes(vec_date)
```

```
## $class
## [1] "Date"
```

S3 Vectors: date calculations

Dates are stored as a number because they represent the amount of days since January 1, 1970, which is referred to as the UNIX Epoch.

`unclass()` tells us what the actual number is.

```
unclass(vec_date)
```

```
## [1] 18815 18816 18817
```

S3 Vectors: date-time

Date-times contain a bit more information than dates. The function to create a datetime vector is `as.POSIXct()`.

We'll convert `vec_date` to a date-time and store it in `vec_datetime_ct`. View the results below.

```
vec_date
```

```
## [1] "2021-07-07" "2021-07-08" "2021-07-09"
```

```
vec_datetime_ct <- as.POSIXct(x = vec_date)
vec_datetime_ct
```

```
## [1] "2021-07-06 17:00:00 MST" "2021-07-07 17:00:00 MST"
## [3] "2021-07-08 17:00:00 MST"
```

We can see `vec_datetime_ct` stores some additional information.

S3 Vectors: date-time attributes

`vec_datetime_ct` is a `double` vector with an additional attribute of `class` set to `"POSIXct" "POSIXt"`.

```
typeof(vec_datetime_ct)
```

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

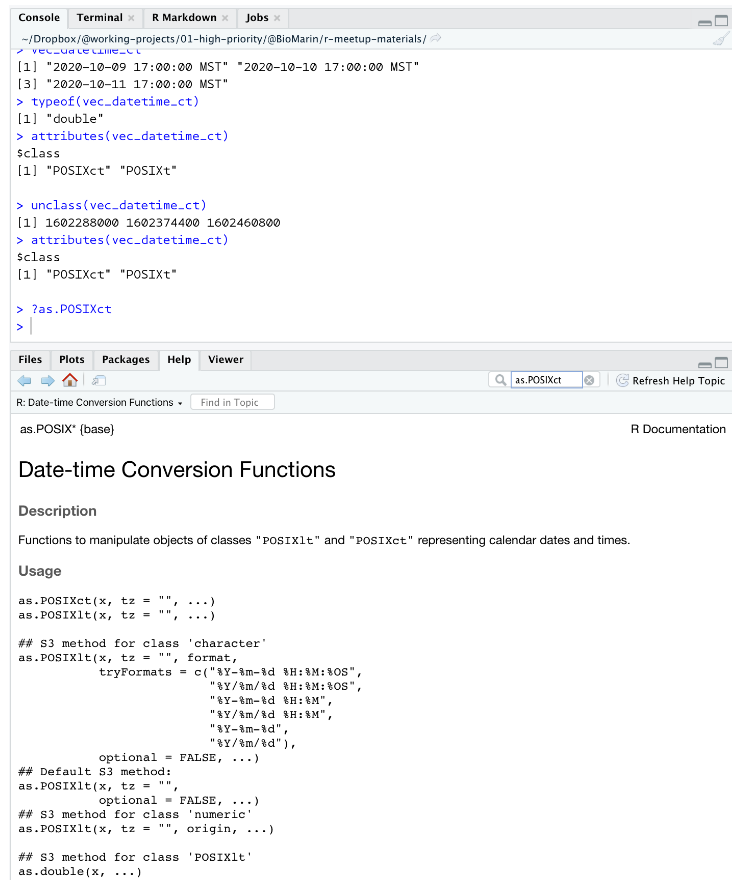
```
attributes(vec_datetime_ct)
```

```
## $class
## [1] "POSIXct" "POSIXt"
```

S3 Vectors: date-time help

Read more about date-times by entering the `as.POSIXct` function into the console preceded by a question mark.

```
?as.POSIXct
```



S3 Vectors: difftime

Difftimes are durations, so we need to supply two dates, which we will create with `time_01` and `time_02`.

```
time_01 <- Sys.Date()
time_02 <- Sys.Date() + 10
vec_difftime <- difftime(time_01, time_02, units = "days")
vec_difftime
```

```
## Time difference of -10 days
```

Difftimes are stored as a `double` vector.

```
typeof(vec_difftime)
```

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

S3 Vectors: difftime attributes

Difftimes are their own `class` and have a `units` attribute set to whatever we've specified in the `units` argument.

```
attributes(vec_difftime)
```

```
## $class
## [1] "difftime"
##
## $units
## [1] "days"
```

We can see the actual number stored in the vector with `unclass()`

```
unclass(vec_difftime)
```

```
## [1] -10
## attr(,"units")
## [1] "days"
```

Matrices

Matrices

A matrix is several vectors stored together into two a two-dimensional object.

```
mat_data <- matrix(data = c(vec_double, vec_integer),
                   nrow = 3, ncol = 2, byrow = FALSE)
mat_data
```

```
##       [,1] [,2]
## [1,]  0.10    1
## [2,]  1.00   10
## [3,] 10.01  100
```

This is a three-column, two-row matrix.

We can check the dimensions of `mat_data` with `dim()`.

```
dim(mat_data)
```

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

Matrix positions

The output in the console tells us where each element is located in `mat_data`.

For example, if I want to get the `10` that's stored in `vec_integer`, I can use look at the output and use the indexes.

```
mat_data
```

```
##       [,1] [,2]
## [1,]  0.10    1
## [2,]  1.00   10
## [3,] 10.01  100
```

```
mat_data[2, 2]
```

```
## [1] 10
```

By placing the index (`[2, 2]`) next to the object, I am telling R, "*only return the value in this position*".

Arrays

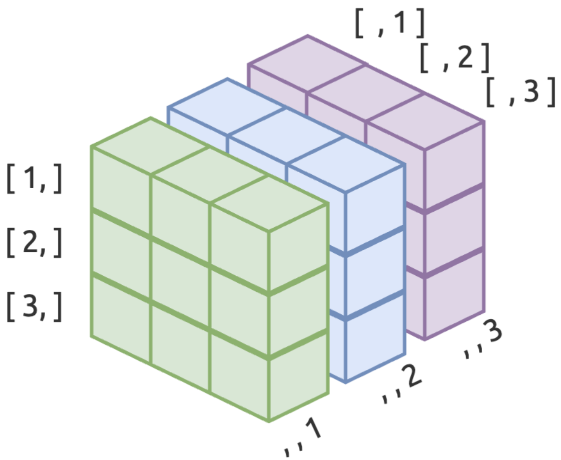
Arrays

Arrays are like matrices, but they can have more dimensions. Below we create a 3x3x3 array using the `seq()` function.

```
array_dat <- array(
  data = c(
    seq(0.3, 2.7, by = 0.3),
    seq(0.5, 4.5, by = 0.5),
    seq(3, 27, by = 3)
  ),
  dim = c(3, 3, 3)
)
```

Array layers

`array_dat` contains numbers in three columns and three rows, stacked in three *layers*.




```r
array_dat
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]  0.3  1.2  2.1
## [2,]  0.6  1.5  2.4
## [3,]  0.9  1.8  2.7
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]  0.5  2.0  3.5
## [2,]  1.0  2.5  4.0
## [3,]  1.5  3.0  4.5
##
## , , 3
##
##      [,1] [,2] [,3]
## [1,]    3   12   21
## [2,]    6   15   24
## [3,]    9   18   27
```

Arrays vs. matrices

Matrices are arrays, but arrays are not matrices.

```
is.matrix(array_dat)
```

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

```
is.array(mat_data)
```

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

```
class(array_dat)
```

```
## [1] "array"
```

```
class(mat_data)
```

```
## [1] "matrix" "array"
```

Data Frames

Data Frames

Data frames are rectangular data with rows and columns (or observations and variables).

```
DataFrame <- data.frame(character = c("A", "B", "C"),
                        integer = c(0.1, 1.0, 10.01),
                        logical = c(TRUE, FALSE, TRUE),
                         stringsAsFactors = FALSE)
DataFrame
```

```
##   character integer logical
## 1         A    0.10    TRUE
## 2         B    1.00   FALSE
## 3         C   10.01    TRUE
```

NOTE: `stringsAsFactors = FALSE` is not required as of R version 4.0.0.

Data Frames

Check the structure of the `data.frame` with `str()`

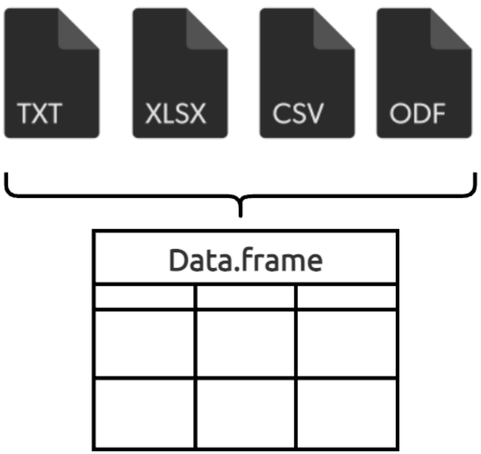
```
str(DataFrame)
```

```
## 'data.frame':    3 obs. of  3 variables:
##  $ character: chr  "A" "B" "C"
##  $ integer  : num  0.1 1 10
##  $ logical  : logi  TRUE FALSE TRUE
```

`str()` gives us a transposed view of the `DataFrame` object, and tells us the dimensions of the object.

Data Frames

If you're importing spreadsheets, most of the work you'll do in R will be with rectangular data objects (i.e. `data.frame`s).



These are the common rectangular data storage object for tabular data in R

Data Frames

What type of object is a `data.frame`?

If we check `DataFrame` with `dput()`...

```
DataFrame
```

```
##   character integer logical
## 1         A    0.10    TRUE
## 2         B    1.00   FALSE
## 3         C   10.01    TRUE
```

```
dput(DataFrame)
```

```
## structure(list(
##  character = c("A", "B", "C"),
##  integer = c(0.1, 1, 10.01),
##  logical = c(TRUE, FALSE, TRUE)),
##  class = "data.frame",
##  row.names = c(NA, -3L))
```

...we see they are `lists`

Data Frames

`data.frame`s are lists with their own class

```
typeof(DataFrame)
```

```
## [1] "list"
```

```
class(DataFrame)
```

```
## [1] "data.frame"
```

...so we can think of `data.frame`s as a special kind of *rectangular* lists, made with different types of vectors, with each vector being of equal length.

Lists

Lists

Lists are special objects because they can contain all other objects (including other lists).

```
dat_list <- list("integer" = vec_integer,
                 "array" = array_dat,
                 "matrix" = mat_data,
                 "data.frame" = DataFrame)
```

Lists have a `names` attribute, which we've defined above in double quotes.

```
attributes(dat_list)
```

```
## $names
## [1] "integer"    "array"      "matrix"     "data.frame"
```

List structure

If we check the `str()` of `dat_list`, we see the structure of list, and the structure of the elements in the list.

```
str(dat_list)
```

```
## List of 4
##  $ integer   : int [1:3] 1 10 100
##  $ array     : num [1:3, 1:3, 1:3] 0.3 0.6 0.9 1.2 1.5 1.8 2.1 2.4 2.7 0.5
##  $ matrix    : num [1:3, 1:2] 0.1 1 10 1 10 ...
##  $ data.frame:'data.frame':    3 obs. of  3 variables:
##   ..$ character: chr [1:3] "A" "B" "C"
##   ..$ integer  : num [1:3] 0.1 1 10
##   ..$ logical  : logi [1:3] TRUE FALSE TRUE
```

Subsetting

Subsetting Vectors

We can subset vectors using brackets [ ]

```
# single item
vec_character[1]
```

```
## [1] "A"
```

```
# range of items
vec_character[1:3]
```

```
## [1] "A" "B" "C"
```

```
# vector of items
vec_character[c(1, 3)]
```

```
## [1] "A" "C"
```

Subsetting Matrices

Matrices are two-dimensional, so we need to use a comma to separate each position in the brackets `[ , ]`

```r
# review
mat_data
```

```
##        [,1] [,2]
## [1,]  0.10    1
## [2,]  1.00   10
## [3,] 10.01  100
```

```r
mat_data[ , 2]
```

```
## [1]   1  10 100
```

```r
mat_data[3,  ]
```

```
## [1]  10.01 100.00
```

```r
mat_data[1, 2]
```

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

Subsetting Arrays

Arrays contain a collection of equal-dimension matrices, so we need to use a comma to separate each position in the bracket `[ , ]`

```r
# review
dim(array_dat)
```

```
## [1] 3 3 3
```

```r
array_dat[3, , 2]
```

```
## [1] 1.5 3.0 4.5
```

```r
array_dat[1, c(3, 2), 2]
```

```
## [1] 3.5 2.0
```

```r
array_dat[1, , ]
```

```
##      [,1] [,2] [,3]
## [1,]  0.3  0.5    3
## [2,]  1.2  2.0   12
## [3,]  2.1  3.5   21
```

Subsetting Arrays

If we only supply a single row `array_dat[1, , ]`, we will see R returns the rows as a column in a single matrix. They are also arranged by columns, not rows.

Here is the original arrangement of the first rows:

```
, , 1

     [,1] [,2] [,3]
[1,]  0.3  1.2  2.1
[2,]  0.6  1.5  2.4
[3,]  0.9  1.8  2.7

, , 2

     [,1] [,2] [,3]
[1,]  0.5  2.0  3.5
[2,]  1.0  2.5  4.0
[3,]  1.5  3.0  4.5

, , 3

     [,1] [,2] [,3]
[1,]    3   12   21
[2,]    6   15   24
[3,]    9   18   27
```

And here is the returned matrix, presented as columns:

```
     [,1] [,2] [,3]
[1,]  0.3  0.5    3
[2,]  1.2  2.0   12
[3,]  2.1  3.5   21
```

Subsetting Data frames

There are multiple ways to subset `data.frame`'s

```r
# subset named vectors
DataFrame$character
```

```
## [1] "A" "B" "C"
```

```r
# row 1, column 2
DataFrame[1, 2]
```

```
## [1] 0.1
```

```r
# using c() & []
DataFrame[c(1, 3), "logical"]
```

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

```r
# using $ and []
DataFrame$character[2]
```

```
## [1] "B"
```

```r
# using $ & ==
DataFrame$integer == 1
```

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

Subsetting Data frames (advanced)

We can combine all three (and more!)

Notice these return `data.frame`s

```
# using [], $ and >=
DataFrame[DataFrame$integer >= 1, ]
```

```
##   character integer logical
## 2         B    1.00   FALSE
## 3         C   10.01    TRUE
```

```
# using [], $, %in% and c()
DataFrame[DataFrame$character %in% c("A", "C"), ]
```

```
##   character integer logical
## 1         A    0.10    TRUE
## 3         C   10.01    TRUE
```

Subsetting Data frames (warning!)

The class of the return object depends on the brackets

using `[]` vs. `[[]]`

```
DataFrame["character"]
```

```
##   character
## 1         A
## 2         B
## 3         C
```

```
DataFrame[["character"]]
```

```
## [1] "A" "B" "C"
```

Note the `class()`

```
# column as data frame
class(DataFrame["character"])
```

```
## [1] "data.frame"
```

```
# column as character vector
class(DataFrame[["character"]])
```

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

Subsetting Lists (single brackets)

Singe bracket `data.frame`

```r
# numeric position returns
# a data.frame
DataFrame[3]
```

```
##   logical
## 1    TRUE
## 2   FALSE
## 3    TRUE
```

```r
# name returns a vector
DataFrame[ , "logical"]
```

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

Single bracket `list`

```r
# numeric position
dat_list[3]
```

```
## $matrix
##       [,1] [,2]
## [1,]  0.10    1
## [2,]  1.00   10
## [3,] 10.01  100
```

```r
# name
dat_list["matrix"]
```

```
## $matrix
##       [,1] [,2]
## [1,]  0.10    1
## [2,]  1.00   10
## [3,] 10.01  100
```

Subsetting Lists (double brackets)

Double bracket (`data.frame`)

```
# numeric position
DataFrame[[3]]
```

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

```
# name
DataFrame[["logical"]]
```

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

Double bracket (`list`)

```
# numeric position
dat_list[[3]]
```

```
##       [,1] [,2]
## [1,]  0.10    1
## [2,]  1.00   10
## [3,] 10.01  100
```

```
# name
dat_list[["matrix"]]
```

```
##       [,1] [,2]
## [1,]  0.10    1
## [2,]  1.00   10
## [3,] 10.01  100
```

Subsetting Lists

We can subset any object in a list using the methods above

```
# single $
dat_list$integer
```

```
## [1]   1  10 100
```

```
# $ & []
dat_list$array[3, , 2]
```

```
## [1] 1.5 3.0 4.5
```

```
# $, [] and >=
dat_list$data.frame[dat_list$data.frame$integer < 1, ]
```

```
##   character integer logical
## 1         A     0.1    TRUE
```

Recap

Recap

The most common R object is a vector

- Atomic vectors: *logical, integer, double, or character (strings)*
- S3 Vectors: *factors, dates, date-times, and difftimes*

More complicated data structures: matrices and arrays

- Matrix: *two-dimensional object*
- Array: *multidimensional object*

Rectangular data structures:

- *`data.frame`s & `tibble`s are special kinds of rectangular lists, which can hold different types of vectors, with each vector being of equal length*

Catch-all data structures:

- *lists can contain all other objects (including other lists)*

More resources

Learn more about R objects in the help files or the following online texts:

1. R for Data Science
2. Advanced R
3. Hands on Programming with R
4. R Language Definition

THANK YOU!

Feedback

@mjfrigaard on Twitter and Github

mjfrigaard@pm.me