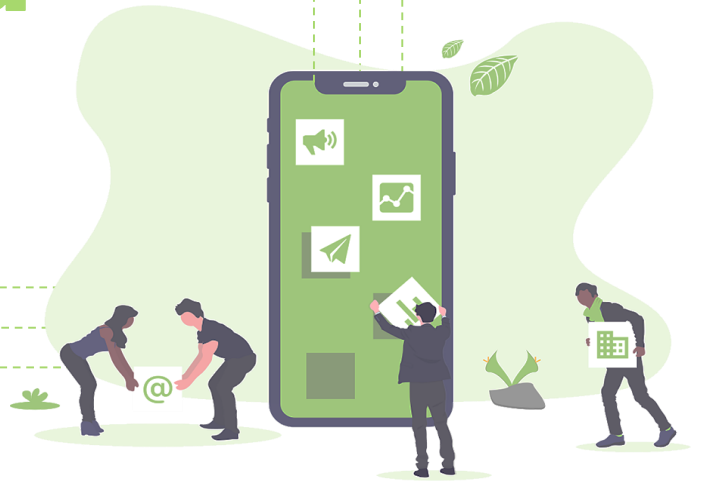


The Data Science Track



Prepared By: R. Daynalo

1

1

8. Data Types and Basic Operations



2

1



Objects



R has five basic or "atomic" classes of objects:

- Character
- Numeric (real numbers)
- Integer
- Complex
- Logical (True/False)

The most basic object is a **vector**.

Prepared By: R. Daynolo

3

3



Vectors



- A vector can only contain objects of the same class.
- BUT: the one exception is a list, which is represented as a vector but can contain objects of different classes
- Empty vectors can be created with the `vector()` function.


```
> vector(mode = "logical", length = 5)
```

- You can also create vectors by using `c()` function, which is short for concatenate (paste together)


Prepared By: R. Daynolo

4


4



Numbers




- Numbers in R are generally treated as numeric objects (i.e. double precision real numbers)
- If you explicitly want an integer, you need to specify the `L` suffix
- Ex: Entering `1` gives you a numeric object; entering `1L` explicitly gives you an integer.
- There is also a special number `Inf` which represents infinity; e.g. `1 / 0`; `Inf` can be used in ordinary calculations; e.g. `1 / Inf` is `0`
- The value `NaN` represents an undefined value ("not a number"); e.g. `0 / 0`; `NaN` can also be thought of as a missing value (more on that later)




Prepared By: R. Daynola

5

5




Attributes



R objects can have attributes

- names, dimnames
- dimensions (e.g. matrices, arrays)
- class
- length
- other user-defined attributes/ metadata

Attributes of an object can be accessed using the `attributes()` function.



Prepared By: R. Daynola

6

6

Entering Input

At the R prompt we type **expressions**. The `<-` symbol is the assignment operator.

```
> x <- 1
> print(x)
[1] 1
> x
[1] 1
> msg <- "hello world"
```

The grammar of the language determines whether an expression is complete or not.

```
> x <- ## incomplete expression
```

Note: The `#` character indicates a comment. Anything to the right of the `#` (including the `#` itself) is ignored)

Prepared By: R. Daynalo

7

7

Evaluation

When a complete expression is entered at the prompt, it is evaluated and the result of the evaluated expression is returned. The result may be auto-printed.

```
> x <- 5 ##nothing printed
> x      ##auto-printing occurs
[1] 5
> print(x) ##explicit printing
[1] 5
```

The `[1]` indicates that `x` is a vector and `5` is the first element.

Prepared By: R. Daynalo

8

8

Printing

```
> x <- 1:20
> x
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14
[15] 15 16 17 18 19 20
```

The `:` operator is used to create integer sequences.

Prepared By: R. Daynolo

9

9

Creating Vectors

The `c()` function can be used to create vectors of objects.

```
> x <- c(0.5, 0.6)      ## numeric
> x <- c(TRUE, FALSE)   ## logical
> x <- c(T, F)          ## logical
> x <- c("a", "b", "c") ## character
> x <- 9:29             ## integer
> x <- c(1+0i, 2+4i)     ## complex
```

Using the `vector()` function

```
> x <- vector("numeric", length = 10)
> x
[1] 0 0 0 0 0 0 0 0 0 0
```

Prepared By: R. Daynolo

10

10

Mixing Objects

What about the following?

```
> y <- c(1.7 , "a")
> y <- c(TRUE, 2)
> y <- c("a", TRUE)
```

When different objects are mixed in a vector, **coercion** occurs so that every element in the vector is of the same class.

Prepared By: R. Daynolo

11

11

Explicit Coercion

Objects can be **explicitly coerced** from one class to another using the `as.*` functions, if available.

```
> x <- 0:6
> class(x)
[1] "integer"
> as.numeric(x)
[1] 0 1 2 3 4 5 6
> as.logical(x)
[1] FALSE TRUE TRUE TRUE TRUE TRUE TRUE
> as.character(x)
[1] "0" "1" "2" "3" "4" "5" "6"
```

Prepared By: R. Daynolo

12

12

Explicit Coercion

Nonsensical coercion results in NAs.

```
> x <- c("a", "b", "c")
> as.numeric(x)
[1] NA NA NA
Warning message:
NAs introduced by coercion
> as.logical(x)
[1] NA NA NA
> as.complex(x)
[1] NA NA NA
Warning message:
NAs introduced by coercion
```

Prepared By: R. Daynalo

13

13

Matrices

Matrices are vectors with a ***dimension*** attribute. The dimension attribute is itself an integer vector of length 2 (nrow, ncol).

```
> m <- matrix(nrow = 2, ncol = 3)
> m
      [,1] [,2] [,3]
[1,]    NA    NA    NA
[2,]    NA    NA    NA
> dim(m)
[1] 2 3
> attributes(m)
$dim
[1] 2 3
```

Prepared By: R. Daynalo

14

14

Matrices

Matrices are constructed *column-wise*, so entries can be thought of starting in the "upper-left" corner and running down columns.

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

Prepared By: R. Daynalo

15

15

Matrices

Matrices can also be created directly from vectors by adding a dimension attribute.

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

Prepared By: R. Daynalo

16

16

cbind-ing and rbind-ing

Matrices can be created by *column-binding* or *row-binding* with `cbind()` and `rbind()`.

```
> x <- 1:3
> y <- 10:12
> cbind(x,y)
      x y
[1,] 1 10
[2,] 2 11
[3,] 3 12
> rbind(x,y)
      [,1] [,2] [,3]
x         1     2     3
y        10    11    12
```

Prepared By: R. Daynola

17

17

Lists

Lists are a special type of a vector that can contain elements of different classes. Lists are a very important data type in R.

```
> x <- list(1, "a", TRUE, 1+4i)
> x
[[1]]
[1] 1

[[2]]
[1] "a"

[[3]]
[1] TRUE

[[4]]
[1] 1+4i
```

Prepared By: R. Daynola

18

18

Factors

Factors are used to represent categorical data. Factors can be unordered or ordered. One can think of a factor as an integer vector where each integer has a *label*.

- Factors are treated specially by modelling functions like `lm()` and `glm()`
- Using factors with labels is *better* than using integers because factors are self-describing; having a variable that has values "Male" and "Female" is better than a variable that has values 1 and 2.

Prepared By: R. Daynola

19

19

Factors

```
> x <- factor(c("yes", "yes", "no", "yes", "no"))
> x
[1] yes yes no  yes no
Levels: no yes
> table(x)
x
no yes
 2  3
> unclass(x)
[1] 2 2 1 2 1
attr(,"levels")
[1] "no" "yes"
```

Prepared By: R. Daynola

20

20

Factors

The order of the levels can be set using the `levels` argument to `factor()`. This can be important in linear modelling because the first level is used as the baseline level.

```
> x <- factor(c("yes", "yes", "no", "yes", "no"),
  levels = c("yes", "no"))
> x
[1] yes yes no  yes no
Levels: yes no
```

Prepared By: R. Daynola

21

21

Missing Values

Missing values are denoted by `NA` or `NaN` for undefined mathematical operations.

- `is.na()` is used to test objects if they are `NA`
- `is.nan()` is used to test for `NaN`
- `NA` values have a class also, so there are integer `NA`, character `NA`, etc.
- A `NaN` value is also `NA` but the converse is not true

Prepared By: R. Daynola

22

22

Missing Values

```
> x <- c(1, 2, NA, 10, 3)
> is.na(x)
[1] FALSE FALSE  TRUE FALSE FALSE
> is.nan(x)
[1] FALSE FALSE FALSE FALSE FALSE
> x <- c(1, 2, NaN, NA, 4)
> is.na(x)
[1] FALSE FALSE  TRUE  TRUE FALSE
> is.nan(x)
[1] FALSE FALSE  TRUE FALSE FALSE
```

Prepared By: R. Daynolo

23

23

Data Frames

Data frames are used to store tabular data.

- They are represented as a special type of list where every element of the list has to have the same length
- Each element of the list can be thought of as a column and the length of each element of the list is the number of rows
- Unlike matrices, data frames can store different classes of objects in each column (just like lists); matrices must have every element be the same class
- Data frames also have a special attribute called `row.names`
- Data frames are usually created by calling `read.table()` or `read.csv()`
- Can be converted to a matrix by calling `data.matrix()`

Prepared By: R. Daynolo

24

24

Data Frames

```
> x <- data.frame(foo = 1:4, bar = c(T, T, F, F))
> x
  foo  bar
1   1 TRUE
2   2 TRUE
3   3 FALSE
4   4 FALSE
> nrow(x)
[1] 4
> ncol(x)
[1] 2
```

Prepared By: R. Daynolo

25

25

Data Frames

```
> foo <- 1:4
> bar <- c(T, T, F, F)
> x <- data.frame(foo, bar)
> x
  foo  bar
1   1 TRUE
2   2 TRUE
3   3 FALSE
4   4 FALSE
```

Prepared By: R. Daynolo

26

26



Names



R objects can also have names, which is very useful for writing readable code and self-describing objects.

```
> x <- 1:3
> names(x)
NULL
> names(x) <- c("moment", "generating", "function")
> names(x)
[1] "moment"      "generating"  "function"
```

Prepared By: R. Daynolo

27

27



Names



Lists can also have names.

```
> x <- list(a = 1, b = 2, c = 3)
> x
$a
[1] 1

$b
[1] 2

$c
[1] 3
```

Prepared By: R. Daynolo

28

28



Names

And matrices.

```
> m <- matrix(1:4, nrow = 2, ncol = 2)
> dimnames(m) <- list(c("a", "b"), c("c", "d"))
> m
  c d
a 1 3
b 2 4
```

Prepared By: R. Daynalo


29

29


9. Reading and Writing Data



30




Reading Data



There are a few principal functions reading data into R.


- `read.table()`, `read.csv()`, for reading tabular data
- `readLines()`, for reading lines of a text file
- `source()`, for reading in R code files (inverse of `dump()`)
- `dget()`, for reading in R code files (inverse of `dput()`)
- `load()`, for reading in saved workspaces
- `unserialize()`, for reading single R objects in binary form




Prepared By: R. Daynolo

31

31




Writing Data



There are analogous functions for writing data to files


- `write.table()`
- `writeLines()`
- `dump()`
- `dput()`
- `save()`
- `serialize()`




Prepared By: R. Daynolo

32

32




Reading Data Files with `read.table`



The `read.table()` function is one of the most commonly used functions for reading data. It has a few important arguments:


- `file`, the name of a file, or a connection
- `header`, logical indicating if the file has a header line
- `sep`, a string indicating how the columns are separated
- `colClasses`, a character vector indicating the class of each column in the dataset
- `nrows`, the number of rows in the dataset




Prepared By: R. Daynolo

33

33




Reading Data Files with `read.table`



The `read.table` function is one of the most commonly used functions for reading data. It has a few important arguments:


- `comment.char`, a character string indicating the comment character
- `skip`, the number of lines to skip from the beginning
- `stringsAsFactors`, should character variables be coded as factors?




Prepared By: R. Daynolo

34

34



read.table




For small to moderately sized datasets, you can usually call `read.table()` without specifying any other arguments

```
> data <- read.table("foo.txt")
```

R will automatically


- skip lines that begin with a #
- figure out how many rows there are (and how much memory needs to be allocated)
- figure what type of variable is in each column of the table telling R all these things directly makes R run faster and more efficiently.
- `read.csv()` is identical to `read.table()` except that the default separator is a comma.




Prepared By: R. Daynola

35

35




Reading in Larger Datasets with read.table



With much larger datasets, doing the following things will make your life easier and will prevent R from choking.

- Read the help page for `read.table()`, which contains many hints
- Make a rough calculation of the memory required to store your dataset. If the dataset is larger than the amount of RAM on your computer, you can probably stop right here.
- Set `comment.char = ""` if there are no commented lines in your file.



Prepared By: R. Daynola

36

36

Reading in Larger Datasets with `read.table`

Use the `colClasses` argument. Specifying this option instead of using the default can make 'read.table' run MUCH faster, often twice as fast. In order to use this option, you have to know the class of each column in your data frame. If all of the columns are "numeric", for example, then you can just set `colClasses = "numeric"`. A quick and dirty way to figure out the classes of each column is the following:

```
> initial <- read.table("datatable.txt", nrows=100)
> classes <- apply(initial,class)
> tabAll <- read.table("datatable.txt", colClasses = classes)
```

- Set `nrows`. This doesn't make R run faster but it helps with memory usage.

Prepared By: R. Daynola

37

37

Know Thy System


In general, when using R with larger datasets, it's useful to know a few things about your system.

- How much memory is available?
- What other applications are in use?
- Are there other users logged into the same system?
- What operating system?
- Is the OS 32 or 64 bit?


Prepared By: R. Daynola

38


38



Calculating Memory Requirements




I have a data frame with 1,500,000 rows and 120 columns, all of which are numeric data. Roughly, how much memory is required to store this data frame?




Prepared By: R. Daynalo

39


39



Text Formats



- dumping and dputing are useful because the resulting textual format is edit-able, and in the case of corruption, potentially recoverable.
- Unlike writing out a table or csv file, `dump()` and `dput()` preserve the metadata (sacrificing some readability), so that another user doesn't have to specify it all over again.
- Textual formats can work much better with version control programs like **subversion** or **git** which can track changes meaningfully in text files
- Textual formats can be longer-lived; if there is corruption somewhere in the file, it can be easier to fix the problem



Prepared By: R. Daynalo

40

40

dput-ing R Objects

Another way to pass data around is by deparsing the R object with `dput` and reading it back in (parsing it) using `dget()`.

```
> y <- data.frame(a = 1, b = "a")
> dput(y)
structure(list(a = 1, b = structure(1L, .Label = "a", class = "factor")),
  class = "data.frame", row.names = c(NA,
-1L))
> dput(y, file = "y.R")
> new.y <- dget("y.R")
> new.y
  a b
1 1 a
```

Prepared By: R. Daynola

41

41

Dumping R Objects

Multiple objects can be deparsed using the `dump()` function and read back in using `source()`.

```
> x <- "foo"
> y <- data.frame(a = 1, b = "a" )
> dump(c("x", "y"), file = "data.R")
> rm(x, y)
> source("data.R")
> y
  a b
1 1 a
> x
[1] "foo"
```

Prepared By: R. Daynola

42

42

Interfaces to the Outside World

Data are read in using *connection* interfaces. Connections can be made to files (most common) or to other things.

- `file`, opens a connection to a file
- `url`, opens a connection to a webpage
- `gzipfile`, opens a connection to a file compressed with gzip
- `bzfile`, opens a connection to a file compressed with bzip2

Prepared By: R. Daynolo

43

43

File Connections

```
> str(file)
function (description = "", open = "", blocking = TRUE,
        encoding = getOption("encoding"), raw = FALSE,
        method = getOption("url.method", "default"))
```

- `description` is the name of the file
- `open` is a code indicating
 - "r" read only
 - "w" writing (and initializing a new file)
 - "a" appending
 - "rb", "wb", "ab" reading, writing, or appending in binary mode (Windows)

Prepared By: R. Daynolo

44

44

Connections

In general, connections are powerful tools that let you navigate files or other external objects. In practice, we often don't need to deal with the connection interface directly.

```
> con <- file("foo.txt", "r")
> data <- read.csv(con)
> close(con)
```

is the same as

```
> data <- read.csv("foo.txt")
```

Prepared By: R. Daynalo

45

45

Reading Lines of a Text File

```
> con <- gzfile("words.gz")
> x <- readLines(con, 10)
> x
[1] "1080"      "10-point"  "10th"      "11-point"
[5] "12-point"  "16-point"  "18-point"  "1st"
[9] "2"         "20-point"
```

Prepared By: R. Daynalo

46

46

Reading Lines of a Text File

`readLines()` can be useful for reading in lines of webpages

```
> con <- url("https://www.upv.edu.ph/", "r")
> x <- readLines(con)
Warning message:
In readLines(con) :
  incomplete final line found on 'https://www.upv.edu.ph/'
> head(x)
[1] " "

[2] "<!DOCTYPE HTML> "

[3] "<html lang=\"en-gb\" dir=\"ltr\" data-config='{\"twitter\":0,
\"plusone\":0,\"facebook\":0,\"style\": \"default\"}'> "
[4] ""

[5] "<head> "

[6] "<meta charset=\"utf-8\">"
```

Prepared By: R. Daynold

47

47

10. Subsetting and Removing NA Values



48

Subsetting

There are a number of operators that can be used to extract subsets of R objects.

- `[]` always returns an object of the same class as the original; can be used to select more than one element (there is one exception)
- `[[` is used to extract elements of a list or a data frame; it can only be used to extract a single element and the class of the returned object will not necessarily be a list or data frame
- `$` is used to extract elements of a list or data frame by name; semantics are similar to that of `[]`.

Prepared By: R. Daynola

49

49

Subsetting

```
> x <- c("a", "b", "c", "c", "d", "a")
> x[1]
[1] "a"
> x[2]
[1] "b"
> x[1:4]
[1] "a" "b" "c" "c"
> x[x > "a"]
[1] "b" "c" "c" "d"
> u <- x > "a"
> u
[1] FALSE TRUE TRUE TRUE TRUE FALSE
> x[u]
[1] "b" "c" "c" "d"
```

Prepared By: R. Daynola

50

50

Subsetting a Matrix

Matrices can be subsetting in the usual way with (i,j) type indices.

```
> x <- matrix(1:6, 2, 3)
> x[1,2]
[1] 3
> x[2,1]
[1] 2
```

Indices can also be missing.

```
> x[1, ]
[1] 1 3 5
> x[, 2]
[1] 3 4
```

Prepared By: R. Daynola

51

51

Subsetting a Matrix

By default, when a single element of a matrix is retrieved, it is returned as a vector of length 1 rather than a 1×1 matrix. This behavior can be turned off by setting `drop = FALSE`.

```
> x <- matrix(1:6, 2, 3)
> x[1,2]
[1] 3
> x[1,2, drop = FALSE]
      [,1]
[1,]     3
```

Prepared By: R. Daynola

52

52

Subsetting a Matrix

Similarly, subsetting a single column or a single row will give you a vector, not a matrix (by default).

```
> x <- matrix(1:6, 2, 3)
> x[1, ]
[1] 1 3 5
> x[1, ,drop = FALSE]
      [,1] [,2] [,3]
[1,]    1    3    5
```

Prepared By: R. Daynola

53

53

Subsetting Lists

```
> x <- list(foo = 1:4, bar = 0.6)
> x[1]
$foo
[1] 1 2 3 4

> x[[1]]
[1] 1 2 3 4
> x$bar
[1] 0.6
> x[["bar"]]
[1] 0.6
> x[["bar"]]
$bar
[1] 0.6
```

Prepared By: R. Daynola

54

54

Subsetting Lists

```
> x <- list(foo = 1:4, bar = 0.6, baz = "hello")
> x[c(1,3)]
$foo
[1] 1 2 3 4

$baz
[1] "hello"
```

Prepared By: R. Daynolo

55

55

Subsetting Lists

The `[[` operator can be used with *computed* indices; `$` can only be used with literal names.

```
> x <- list(foo = 1:4, bar = 0.6, baz = "hello")
> name <- "foo"
> x[[name]]      ## computed index for 'foo'
[1] 1 2 3 4
> x$name         ## element 'name' doesn't exist!
NULL
> x$foo          ## element 'foo' does exist
[1] 1 2 3 4
```

Prepared By: R. Daynolo

56

56

Subsetting Nested Elements of a List

The `[]` operator can take a integer sequence.

```
> x <- list(a = list(10, 12, 14), b = c(3.14, 2.81))
> x[[c(1, 3)]]
[1] 14
> x[[1]][[3]]
[1] 14
> x[[c(2, 1)]]
[1] 3.14
```

Prepared By: R. Daynolo

57

57

Partial Matching

Partial matching of names is allowed with `[]` and `$`.

```
> x <- list(aardvark = 1:5)
> x$a
[1] 1 2 3 4 5
> x[["a"]]
NULL
> x[["a", exact = FALSE]]
[1] 1 2 3 4 5
```

Prepared By: R. Daynolo

58

58

Removing NA Values

A common task is to remove missing values (NAs).

```
> x <- c(1, 2, NA, 4, NA, 5)
> bad <- is.na(x)
> x[!bad]
[1] 1 2 4 5
```

Prepared By: R. Daynalo

59

59

Removing NA Values

What if there are multiple things and you want to take the subset with no missing values?

```
> x <- c(1, 2, NA, 4, NA, 5)
> y <- c("a", "b", NA, "d", NA, "f")
> good <- complete.cases(x, y)
> good
[1] TRUE TRUE FALSE TRUE FALSE TRUE
> x[good]
[1] 1 2 4 5
> y[good]
[1] "a" "b" "d" "f"
> x1 <- x[good]
> x1
[1] 1 2 4 5
```

Prepared By: R. Daynalo

60

60

Removing NA Values

```
> airquality[1:6, ]
  Ozone Solar.R Wind Temp Month Day
1   41    190  7.4   67     5    1
2   36    118  8.0   72     5    2
3   12    149 12.6   74     5    3
4   18    313 11.5   62     5    4
5   NA     NA 14.3   56     5    5
6   28     NA 14.9   66     5    6

> good <- complete.cases(airquality)
> airquality[good, ][1:6, ]
  Ozone Solar.R Wind Temp Month Day
1   41    190  7.4   67     5    1
2   36    118  8.0   72     5    2
3   12    149 12.6   74     5    3
4   18    313 11.5   62     5    4
7   23    299  8.6   65     5    7
8   19     99 13.8   59     5    8
```

Prepared By: R. Daynalo

61

61

11. Vectorized Operations



62

Vectorized Operations

Many operations in R are *vectorized* making code more efficient, concise, and easier to read.

```
> x <- 1:4; y <- 6:9
> x + y
[1] 7 9 11 13
> x > 2
[1] FALSE FALSE TRUE TRUE
> x >= 2
[1] FALSE TRUE TRUE TRUE
> y == 8
[1] FALSE FALSE TRUE FALSE
> x * y
[1] 6 14 24 36
> x / y
[1] 0.1666667 0.2857143 0.3750000 0.4444444
```

Prepared By: R. Daynola

63

63

Vectorized Matrix Operations

```
> x <- matrix(1:4, 2, 2); y <- matrix(rep(10, 4), 2, 2)
> x * y ## element-wise multiplication
      [,1] [,2]
[1,]   10   30
[2,]   20   40
> x / y
      [,1] [,2]
[1,]  0.1  0.3
[2,]  0.2  0.4
> x %*% y ## true matrix multiplication
      [,1] [,2]
[1,]   40   40
[2,]   60   60
```

Prepared By: R. Daynola

64

64