Module 3: R

Data in R

Instructor: Anjali Silva, PhD

Data Sciences Institute, University of Toronto

2022

Course Documents

- Visit: https://github.com/anjalisilva/IntroductionToR
- All course material will be available via IntroductionToR GitHub repository (https://github.com/anjalisilva/IntroductionToR). Folder structure is as follows:
  - Lessons - All files: This folder contains all files.
  - Lessons - Data only: This folder contains data only.
  - Lessons - Lesson Plans only: This folder contains lesson plans only.
  - Lessons - PDF only: This folder contains slide PDFs only.
  - README - README file
  - .gitignore - Files to ignore specified by instructor

Course Contacts

- Instructor: Anjali Silva Email: a.silva@utoronto.ca (Must use the subject line DSI-IntroR. E.g., DSI-IntroR: Inquiry about Lecture I.)
- TA: Tia Harrison Email: tia.harrison@mail.utoronto.ca

Overview

- Vectors (Wickham and Grolemund, 2017 Chapter 20)
- Tibbles (Wickham and Grolemund, 2017 Chapter 10)
- Strings (Wickham and Grolemund, 2017, Chapter 14)
- Factors (Wickham and Grolemund, 2017, Chapter 15)
- Dates and times (Wickham and Grolemund, 2017, Chapter 16)
- Missing values (Wickham and Grolemund, 2017 Chapter 5)

Data types and structures

Atomic types in R

1. *Character* have quotes around them. `"welcome"`, `'hello world'`, and `"2"` are all of type character.

- May also be referred to as a string in some contexts

1. *Logical* is either `TRUE` or `FALSE`
2. *Double* is a number. `3.1`, `-73`, and `2700` are all doubles.
3. Integer ex. 100
4. Complex ex. 10+3i
5. Raw (byte representation)

You likely will only need to know the first four.

Atomic types in R

```r
typeof("welcome")
```

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

```r
typeof(FALSE)
```

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

```r
typeof(3.14)
```

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

```r
typeof(100L)
```

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

```r
typeof(10+3i)
```

```
## [1] "complex"
```

Vectors

Atomic vectors are made using the `c()` function.

We can build vectors of data out of all atomic data types. All the data types in an atomic vector need to match.

Lists, which are sometimes called recursive vectors, can contain other lists. They can also be heterogeneous, containing multiple types.

Logical Vectors

Possible values are TRUE, FALSE, and NA

Often created with comparison operators

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

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

```
length(logical_vector)
```

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

```
#which numbers between 1 and 5 are divisible by 2?
compare_vector <- 1:5 %% 2 == 0
typeof(compare_vector)
```

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

```
length(compare_vector)
```

```
## [1] 5
```

Numeric Vectors

Integer and double vectors together are called numeric. Numbers in R are doubles by default, so you need to specify L to make an integer value.

```
double_vector <- c(3.1, -73, 2700)
typeof(double_vector)
```

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

```
length(double_vector)
```

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

```
integer_vector <- c(3L, -73L, 2700L)
typeof(integer_vector)
```

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

```
length(integer_vector)
```

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

Numeric Vectors

Differences:

1. Doubles are approximations, because floating point numbers cannot always be represented with a fixed amount of memory.
2. Special values:

- Integers have NA
- Doubles have NA, NaN, Inf, and -Inf

We can check for special values in general with is.infinite, is.na, and is.nan:

```
special_values <- c(-1, 0, 1, NA) / 0
special_values
```

```
## [1] -Inf  NaN  Inf   NA
```

```
is.finite(special_values)
```

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

```
is.infinite(special_values)
```

Character Vectors

```
character_vector <- c("hello", "world", "2,000")
typeof(character_vector)
```

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

```
length(character_vector)
```

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

Augmented Vectors

Augmented vectors, which add metadata in the form of attributes to vectors, are the basis of many data types in R.

- Factors are made from integer vectors
- Dates and times are made from numeric vectors
- Data frames and tibbles are made from lists.

Coercion

You can coerce one type of vector to another explicitly:

```
character_vector <- c("1", "0", "1")
typeof(character_vector)
```

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

```
numeric_vector <- as.numeric(character_vector)
typeof(numeric_vector)
```

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

```
double_vector <- as.double(character_vector)
typeof(double_vector)
```

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

```
logical_vector <- as.logical(character_vector)
typeof(logical_vector)
```

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

Implicit Coercion

```r
numeric_vector <- 1:10

# which are greater than 4?
logical_vector <- numeric_vector > 4
logical_vector
```

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

```r
# how many are greater than 4?
sum(logical_vector)
```

```
## [1] 6
```

```r
# what proportion are greater than 4?
mean(logical_vector)
```

```
## [1] 0.6
```

Mixing Types

If you mix types in a vector, all types will be coerced to match the "most complex" type.

```
typeof(c(TRUE, FALSE, 10L))
```

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

```
typeof(c(1L, 4L, 1.5))
```

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

```
typeof(c(1.5, -3.2, "a"))
```

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

Checking type with tidyverse functions

```
is_logical(c(TRUE, FALSE))
```

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

```
is_integer(c(1L, 2L))
```

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

```
is_double(c(1.2, 1.3))
```

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

```
is_character(c("hello", "world"))
```

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

```
is_atomic(c(1,2,3))
```

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

```
is_list(c(list(1,2,3)))
```

Vector Recyling

If an operation requires a longer vector than provided, R will recycle the vector to get to the required length:

```
1:5 + 1:10
```

```
##  [1]  2  4  6  8 10  7  9 11 13 15
```

It will also warn you if the recycled vector isn't a complete multiple of the smaller vector:

```
1:5 + 1:13
```

```
## Warning in 1:5 + 1:13: longer object length is not a multiple of
## shorter object length
```

```
##  [1]  2  4  6  8 10  7  9 11 13 15 12 14 16
```

Naming Vectors

```
named_vector <- c(a = 100, b = 90, c = 80, d = 70, e = 60)
named_vector
```

```
##   a   b   c   d   e 
## 100  90  80  70  60
```

Named vectors are good if you want to subset.

Subsetting

You can subset with a numeric vector containing only integers:

```
named_vector[3]
```

```
##  c
## 80
```

```
named_vector[c(3,3,4)]
```

```
##  c  c  d
## 80 80 70
```

```
named_vector[c(-1,-2,-5)]
```

```
##  c  d
## 80 70
```

Subsetting

You can subset with a logical vector:

```
named_vector[c(TRUE, TRUE, FALSE, TRUE, FALSE)]
```

```
##   a   b   d
## 100  90  70
```

```
named_vector[named_vector %% 20 == 0]
```

```
##   a   c   e
## 100  80  60
```

Subsetting

You can subset with a character vector:

```
named_vector[c("a", "c")]
```

```
##   a   c
## 100  80
```

Lists

Because a list can contain other lists, they can represent hierarchical structures.

```
mylist <- list(7, "abc", FALSE)
mylist
```

```
## [[1]]
## [1] 7
##
## [[2]]
## [1] "abc"
##
## [[3]]
## [1] FALSE
```

```
str(mylist)
```

```
## List of 3
##  $ : num 7
##  $ : chr "abc"
##  $ : logi FALSE
```

Subsetting lists

```
mylist <- list(a = 1:4, b = "zyx", c = list(-1, -5))

mylist[1:2]
```

```
## $a
## [1] 1 2 3 4
##
## $b
## [1] "zyx"
```

Extracting items

```r
mylist[[2]]
```

```
## [1] "zyx"
```

```r
mylist[["b"]]
```

```
## [1] "zyx"
```

```r
mylist$b
```

```
## [1] "zyx"
```

Additional Attributes

- Names
- Dimensions (vector behaves like a matrix or array)
- Class

Tibbles

R has data.frames for storing columns and rows of data, but in tidyverse we have tibbles instead.

Tibbles are augmented lists. TAll elements of the tibble must be vectors with the same length. The same applies to data.frames.

You can create a new tibble as follows:

```
mytibble <- tibble(x = 1:5,
       y = 1,
       z = x ^ 2 + y)
mytibble
```

Coercing to tibble

```
data("iris")
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```
iris_tibble <- as_tibble(iris)
iris_tibble
```

```
## # A tibble: 150 × 5
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##           <dbl>       <dbl>        <dbl>       <dbl> <fct>
##  1          5.1         3.5          1.4         0.2 setosa
##  2          4.9         3            1.4         0.2 setosa
##  3          4.7         3.2          1.3         0.2 setosa
##  4          4.6         3.1          1.5         0.2 setosa
##  5          5           3.6          1.4         0.2 setosa
##  6          5.4         3.9          1.7         0.4 setosa
```

Differences between data.frames and tibbles

- Tibbles print more nicely and are easier to read in the console
- Subsetting works differently

Subsetting data.frames

```r
iris$Species
```

```r
iris[["Species"]]
```

```
##   [1] setosa     setosa     setosa     setosa     setosa
##   [6] setosa     setosa     setosa     setosa     setosa
##  [11] setosa     setosa     setosa     setosa     setosa
##  [16] setosa     setosa     setosa     setosa     setosa
##  [21] setosa     setosa     setosa     setosa     setosa
##  [26] setosa     setosa     setosa     setosa     setosa
##  [31] setosa     setosa     setosa     setosa     setosa
##  [36] setosa     setosa     setosa     setosa     setosa
##  [41] setosa     setosa     setosa     setosa     setosa
##  [46] setosa     setosa     setosa     setosa     setosa
##  [51] versicolor versicolor versicolor versicolor versicolor
##  [56] versicolor versicolor versicolor versicolor versicolor
##  [61] versicolor versicolor versicolor versicolor versicolor
##  [66] versicolor versicolor versicolor versicolor versicolor
##  [71] versicolor versicolor versicolor versicolor versicolor
##  [76] versicolor versicolor versicolor versicolor versicolor
##  [81] versicolor versicolor versicolor versicolor versicolor
##  [86] versicolor versicolor versicolor versicolor versicolor
##  [91] versicolor versicolor versicolor versicolor versicolor
```

Strings

```
library(stringr) # part of the tidyverse
```

Strings are contained between single '' or double "" quotes.

```
"This is a string"
'6' # this is ALSO a string
```

Check the length

```
str_length("This is a string")
```

```
## [1] 16
```

Strings

Combine

```
str_c("This is a string", "6")
```

```
## [1] "This is a string6"
```

Take a subset

```
str_sub("This is a string", 7, 12)
```

```
## [1] "s a st"
```

Strings

Change capitalization

```
str_to_lower("UPPER case")
```

```
## [1] "upper case"
```

```
str_to_upper("LOWER case")
```

```
## [1] "LOWER CASE"
```

```
str_to_title("no capitalization")
```

```
## [1] "No Capitalization"
```

Matching patterns

```
mystring <- c("apple", "banana", "clementime", "dragonfruit")

str_view(mystring, "an")
```

```
apple
banana
clementime
dragonfruit
```

Regular expressions

A period matches any character.

```
str_view(mystring, ".a.")
```

```
apple
banana
clementime
dragonfruit
```

Regex Anchors

^ matches to the start of a string

```
str_view(mystring, "^a")
```

```
apple
banana
clementime
dragonfruit
```

Classes

- "\d" matches any digit. (Remember that it will need an additional escape character.)
- "\s" matches any whitespace. (Remember that it will need an additional escape character.)
- [xyz] matches x, y, or z
- [^xyz] matches anything except x, y, or z

Amounts

- ? matches 0 or 1
-
    - matches 1 or more
-
    - matches 0 or more

```
mystring <- "abcccdeee"

str_view(mystring, "cc?")
```

abcccdeee

Disambiguating

We can use parentheses in complex expressions to make multiple requirements. For example, finding repeated pairs:

```r
mystring <- c("abab", "cdcd", "efgh")
str_view(mystring, "(..)\\1", match = T)
```

```
abab
cdcd
```

Using regex

```r
mystring <- c("banana", "dodo", "apple")

str_detect(mystring, "(..)\\1")
```

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

```r
str_subset(mystring, "(..)\\1")
```

```
## [1] "banana" "dodo"
```

```r
str_count(mystring, "(..)\\1")
```

```
## [1] 1 1 0
```

Factors

In R, factors are for working with categorical variables, where there is a fixed and known set of possible values.

```
library(forcats) # part of the tidyverse
```

Let's say we have a variable storing the months of our data.

```
months <- c("Dec", "Apr", "Jan", "Mar")
months
```

```
## [1] "Dec" "Apr" "Jan" "Mar"
```

Factors

With a factor, you can restrict the number of possible values and order those values.

```r
month_levels <- c(
  "Jan", "Feb", "Mar", "Apr", "May", "Jun",
  "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
)
```

```r
month_fix <- factor(months, month_levels)
month_fix
```

```
## [1] Dec Apr Jan Mar
## Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

Recoding factors

If we wanted all the levels to be full month names instead, we could recode the levels:

```
fct_recode(month_fix, "December" = "Dec")
```

```
## [1] December Apr      Jan      Mar
## 12 Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct ... December
```

Dates

- Dates in R are numeric vectors that represent number of days since January 1, 1970.
- Tibbles print this as .

```
today()
```

```
## [1] "2022-10-31"
```

Time

- time within a day: tibbles print this as .

Datetime

- date-time: a date plus a time that uniquely identifies an instant in time.
- Numeric vectors that represent the number of seconds since January 1, 1970.
- Tibbles print this as .
- Elsewhere in R these are called POSIXct.

```
now()
```

```
## [1] "2022-10-31 17:00:24 EDT"
```

Managing dates using tidyverse

You will primarily use the library lubridate, and not see POSIXcts very frequently.

```r
library(lubridate)

lubridate::as_datetime(<POSIXct item>)
```

Parsing dates from strings and numbers

```r
ymd("2017-01-31")
```

```
## [1] "2017-01-31"
```

```r
ymd(20170131) # the most concise way
```

```
## [1] "2017-01-31"
```

```r
mdy("January 31st, 2017")
```

```
## [1] "2017-01-31"
```

```r
dmy("31-Jan-2017")
```

```
## [1] "2017-01-31"
```

Switching between date and datetime

```
today()
```

```
## [1] "2022-10-31"
```

```
as_datetime(today())
```

```
## [1] "2022-10-31 UTC"
```

```
now()
```

```
## [1] "2022-10-31 17:00:24 EDT"
```

```
as_date(now())
```

```
## [1] "2022-10-31"
```

Components

- We can extract year, month, day of the month, day of the year, day of the week, hour, minute, and second:

```r
datetime <- ymd_hms("2016-07-08 12:34:56")

year(datetime)
```

```
## [1] 2016
```

```r
month(datetime)
```

```
## [1] 7
```

```r
mday(datetime)
```

```
## [1] 8
```

```r
yday(datetime)
```

```
## [1] 190
```

Time spans

```
today() - ymd(20000101)
```

```
## Time difference of 8339 days
```

```
as.duration(today() - ymd(20000101))
```

```
## [1] "720489600s (~22.83 years)"
```

```
dseconds(120)
```

```
## [1] "120s (~2 minutes)"
```

```
dminutes(60)
```

```
## [1] "3600s (~1 hours)"
```

```
dhours(c(12, 24))
```

```
## [1] "43200s (~12 hours)" "86400s (~1 days)"
```

```
ddays(0:7)
```

Periods

Periods are time spans that don't have fixed length in seconds, so they work more like you might anticipate.

```
today() + years(1)
```

```
## [1] "2023-10-31"
```

```
today() + months(1)
```

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

```
today() + days(1)
```

```
## [1] "2022-11-01"
```

```
today() + hours(1)
```

```
## [1] "2022-10-31 01:00:00 UTC"
```

```
today() + minutes(1)
```

Time zones

```
ymd_hms("2021-01-01 12:00:00", tz = "America/New_York")
```

```
## [1] "2021-01-01 12:00:00 EST"
```

```
ymd_hms("2021-01-01 18:00:00", tz = "Europe/Copenhagen")
```

```
## [1] "2021-01-01 18:00:00 CET"
```

```
ymd_hms("2021-01-01 04:00:00", tz = "Pacific/Auckland")
```

```
## [1] "2021-01-01 04:00:00 NZDT"
```

Missing data

Comparisons do not work as expected with missing values.

```
NA > 5
```

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

```
NA == 10
```

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

```
NA + 5
```

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

```
NA == NA
```

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

Detect missing values with:

```
is.na(NA)
```

Exercises

Exercises

1. Make a tibble where the vectors do not have equal length. What happens?
2. In the following tibble, extract variable:

```
mytibble <- tibble(
  A = 1:10,
  B = A * 2)
```

1. Try using functions paste() and paste0(). Compare them to str_c. How do they work differently?
2. Look up function str_trim() and demonstrate application.
3. Match the sequence "'\ with regex
4. Match words that start with x with regex.
5. Match words that are 3 letters long with regex.
6. Match words that only contain consonants with regex.
7. What does ^.*$ match?
8. What happens if you parse a string with invalid dates?

Any questions?