University of Zurich^UZH

Department of Geography

GEO 812
Getting started with R for Spatial Analysis

Session 3: Data types and functions

Peter Ranacher, Nico Neureiter

September 2023

Learning objectives

You are able to

- name and explain R’s most important data structures
- explain loops and understand why not to use them in R
- work with functionals
- write your own functions, test and debug them.

Vectors

Vectors are the most basic objects in R.

Atomic vectors are sequences of data elements of the same type.

Lists are recursive vectors. They can contain atomic vectors and other lists.

All vectors have a type and a length.

Some vectors have attributes.

Find out more: Vectors | Advanced R

The four (most important) atomic vector types

The four Dalton brothers Joe, William, Jack and Averell are 120.5, 150.4, 170.3 and 190.0 cm tall. William and Jack have committed 10 crimes, Averell 5 and Joe 136. They are smart, except for Averell.

character	`names_daltons <- c("Joe", "William", "Jack", "Averell")`
double	`height_daltons <- c(120.5, 150.4, 170.3, 190.0)`
integer	`crimes_daltons <- c(136L, 10L, 10L, 5L)`
logical	`smart_daltons <- c(TRUE, TRUE, TRUE, FALSE)`

Inspecting vectors

Which type is it?

```
typeof(names_daltons)
class(names_daltons)
```

Check if a vector is of a specific type

```
is.character(names_daltons)
is.logical(smart_daltons)
is.double(height_daltons)
is.integer(height_daltons)
is.numeric(height_daltons)
```

Subsetting and indexing

Get the third of the Dalton's names.

```
names_daltons[3]
```

Get the second to fourth of the Dalton's names.

```
names_daltons[2:4]
```

Get the second and fourth of the Dalton's names

```
names_daltons[c(1, 4)]
```

Get the names of the Daltons taller than 125 cm.

```
index <- height_daltons > 125
names_daltons[index]
```

Find out more: Subsetting | Advanced R

Useful but annoying I: Coercion

All elements of an atomic vector must be of the same type.

If not they will be forced into the most flexible type.

```
a <- c(TRUE, "hello", 1)
typeof(a)
```

Find out more: What is Coercion in R?

Useful but annoying II: Recycling

When adding (subtracting, multiplying, …) two vectors, they need not have the same length. If not, the shorter vector is repeated to the same length as the longer vector.

Add 10 cm to the Daltons height.

```
height_daltons + c(10, 10, 10, 10)
height_daltons + 10
height_daltons + c(10, 10)
```

Find out more: Vector Recycling in R

Useful but annoying III: Wildcard imports

The `library(...)` command imports all functions and variables from the package.

Unfortunate consequence: **name collisions**.

E.g. the packages `raster` and `dplyr` both have a `select` function. If you import both, the first one will be overwritten.

Possible solution: specify the namespace of a function using two colons. E.g. use

`raster::select` and `dplyr::select`

for the different select functions.

Find out more:

Lists

Lists are an **ordered collections** of elements.

Lists can combine different objects and even lists.

Collect the names and heights of the Daltons in a list.

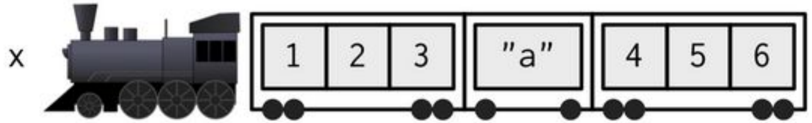
```
daltons_list <- list(names_daltons, height_daltons)
```

Put that list in a list together with the crimes.

```
daltons_list2 <- list(daltons_list, crimes_daltons)
```

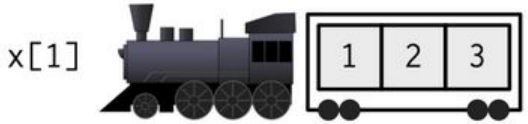
Find out more: Lists | Advanced R

Subsetting lists



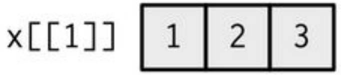
Extract the first and second elements of the Dalton’s list and return as list.

```
daltons_list[2]
daltons_list[1:2]
```



Extract the second element of the Dalton’s list and return.

```
daltons_list[[2]]
```



Find out more: Subsetting | Advanced R

Matrices

Matrices are vectors of dimension 2.
Matrices consist of rows and columns.

For two years, the Daltons repeatedly robbed cities in Colorado and Kansas. In the first year, they robbed two cities in Colorado and four in Kansas; in the second year, they robbed five in Colorado and three in Kansas.

```
robberies_daltons <- matrix(data = c(2, 4, 5, 3),
                            ncol = 2, nrow = 2, byrow = T)
```

Find out more: Matrices | Advanced R

Subsetting matrices

Get all robberies in the first year and second state.

```
robberies_daltons[1, 2]
```

Get all robberies in the second year.

```
robberies_daltons[2, ]
```

Get all robberies in the first state.

```
robberies_daltons[, 1]
```

Zeilen zuerst, Spalten später
(first the rows, then the columns)!

Find out more: Subsetting | Advanced R

Dataframes (tibbles)

Dataframes are **lists** where all elements have the same length.

Dataframes share properties of a **matrix** and a **list**.

Combine the Dalton's height and names in a dataframe.

```
daltons_df <- data.frame(height = height_daltons,
                         names = names_daltons)
```

Subset the height column.

```
daltons_df[, 1]
daltons_df$height
```

Find out more: Data frames | Advanced R

Factors

Factors are vectors whose elements take a fixed set of possible values.

Factors represent **categorical** data.

Lucky Luke categorizes the Daltons into three levels of danger: 'deadly, 'dangerous' and 'harmless'. Joe is deadly, Jack, and William are dangerous, Averell is harmless.

```
danger_daltons <- factor(c("deadly","dangerous",
                           "dangerous","harmless"),
                           levels = c("deadly","dangerous","harmless"))
```

possible values

Find out more: Factors

Verbose code is bad

Write code that is clear, concise and easy to follow and avoid copy-pasting.

- easier to see the intent of your code
- easier to respond to changing requirements □ only change code in one place
- fewer bugs, since each line of code is used in more than one place

Never copy-and-paste more than twice!

Imagine the following task

Compute the mean for every column in `msleep`, but only if the column contains numeric data.

`is.numeric(msleep$name)`	column is not numeric
`is.numeric(msleep$genus)`	column is not numeric
…	
`is.numeric(msleep$sleep_total)`	column is numeric
`mean(msleep$sleep_total)`	compute the mean
…	
`is.numeric(msleep$bodywt)`	columns is numeric
`mean(msleep$bodywt)`	compute the mean

very naive approach.

Generalise the task

1. Iterate through all columns in `msleep`.

loop

2. If the column is numeric, compute the mean.
   If not, don't.

conditional statement

Loops

`for` loops

run a code block a certain number of times,

e.g compute the square root for all integers from 1 to 100

`while` loops

run a code block until a certain condition is met

e.g. compute the sum of all integers until it exceeds 100

Find out more: Control flow | Advanced R

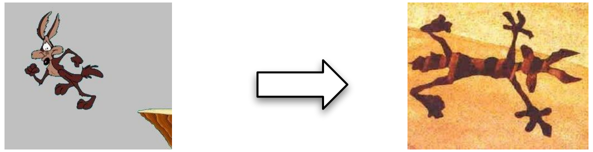
Conditional statements

A **condition** is a logical expression that is either TRUE or FALSE.

When the condition is TRUE, the **consequence** is executed.

When the condition is FALSE, the **alternative consequence** is executed.

```
rock_below <- FALSE
if (!rock_below) {
  coyote_falls <- TRUE
} else {
  coyote_falls <- FALSE
}
```



Find out more: Control flow | Advanced R

To loop or not loop?

Loops offer a good view on what is supposed to happen.

They require an understanding of the data and the process to carry out,

but they are usually **verbose**, and **inefficient**.

Know your loops, but **get rid** of them whenever possible using

- vectorisation
- functions
- functionals

Vectorisation

Add `a` and `b`.

```
a <- c(1, 2, 4, 1)
b <- c(2, 1, 5, 1)
```

You could loop over each element in `a` and add it to the corresponding element in `b`.
Don’t.

Use vector addition instead.

```
a + b
```

Find out more: Vectorization in R: Why?

Functions

Find the average age of passengers on the Titanic.

You could iterate over each row in the dataframe, sum the values in the age column, and then divide by the number of rows.
Don’t.

Use a function that understands vectors instead.

```
mean(titanic_survival$age, na.rm = TRUE)
```

Find out more: Functions

Functionals

Functionals are higher-order functions that operate on functions.

`lapply()`

- takes a list (or a vector) and a function as input
- calls the function for each element of the vector
- returns a list where each element contains to the result of applying the function to the corresponding element of the input list

`apply()`

- takes a matrix and a function as input
- `MARGIN` indicates whether the function is applied to the rows (`MARGIN = 1`) or columns (`MARGIN = 2`)
- returns results as a matrix or if possible, in more compact form

Find out more: Functionals | Advanced R

Applying `lapply()`

Compute the mean for every column in `msleep`, but only if the column contains numeric data.

You could follow the approach on slide 16.
Don't.

Use functionals instead.

```
lapply(msleep, function(x)
  if (is.numeric(x)) mean(x, na.rm = TRUE) else NULL)
```

Writing your own functions

Write a function to calculate the cube root of a number.

function name

argument(s)

```r
cube_root <- function(x) {
  result <- x^(1/3)
  return(result)
}
```

What does the function do?

What does the function return?

Find out more: Functions

Calling the cube root function

```
cube_root(1000)
```

OK

```
cube_root(-1000)
```

NaN

```
cube_root("Busta Rhymes")
```

Verifying the input data

Ensure that the data given to the function are of the right type and make it handle different types of input data.

Verify that the input is numeric

```
cube_root <- function(x) {
  if (!is.numeric(x)) stop("x must be a number")
  result <- ifelse(x >= 0, x^(1/3), -(-x)^(1/3))
  return(result)
}
```

Treat negative and positive input differently

Debugging a function

When your function does not work or produces the wrong results you can debug it.

Inspect the cube root function step by step and examine the values of the variables when using 'Busta Rhymes' as input. Once the debugging opens in the console, you can enter a variable and show its current value.

```
debug(cube_root)
cube_root("Busta Rhymes")
```

Stop entering the debug mode.

```
undebug(cube_root)
```

Find out more: Debugging | Advanced R

Learning objectives revisited

You are able to

- name and explain R’s most important data structures
- explain loops and understand why not to use them in R
- work with functionals
- write your own functions, test and debug them.

And now for something completely different: Where did you go on your summer vacation?

Please enter the destination of your last vacation in this spreadsheet:

http://bit.ly/460UOBk

We will use this as data for the following exercise.

Exercises

1. Download the holiday_destinations table as a CSV file and load it into a tibble.

2. Load the `tidygeocoder` package and use it to add geolocations for each destination in the tibble.

3. Write a function d_great_circle that computes the great-circle distance (d) between two points on earth. This is the formula for d:

$$d = 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \cos(\varphi_1)\cos(\varphi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right)$$

   It takes as input the latitude φ and longitude λ of two locations on the Earth surface and the Earth radius *r*. You can set *r* to 6371 km.
   - Perform data checking (is the input numeric, is it a valid latitude/longitude?)
   - In R, cos()and sin() take radians as input! This function helps you with the conversion:

```
deg2rad <- function(deg) {(deg * pi) / (180)}
```

4. Compute the distance from Zurich (φ = 47.3686498, λ = 8.5391825) to your holiday locations.
   - Use mutate() rather than apply(). Why?