Data Analysis for the Social Sciences with R

Basic Programming in R

Prof. Kevin Koehler
kevin.koehler@santannapisa.it

Programming in R

Sant'Anna
School of Advanced Studies – Pisa

Programming in R

Learning some basic programming allows you to

1. Write user-defined functions
2. Automate repetitive tasks
3. Integrate with other programming environments (e.g., Python)

Today's class

Task: On the GitHub page, you will find a file with the names of about 13,000 Turkish MPs. We want to create a new column with a transliterated version of their names.

Today's class

1. Basic programming notions

Sant'Anna
School of Advanced Studies – Pisa

Today's class

1. Basic programming notions
2. User-defined functions

Today's class

1. Basic programming notions
2. User-defined functions
3. `for` loops

Sant'Anna
School of Advanced Studies – Pisa

Basic programming notions

Sant'Anna
School of Advanced Studies – Pisa

Data classes and structures

There are three data classes in R

1. Numeric
2. Character
3. Factor

You can convert objects between classes with `as.numeric()`, `as.character()`, and `as.factor()`.

Why `factors` are weird

`factors` have **levels** and **labels**. Internally, R represents factors as ordered sequences. This can lead to odd behavior such as:

```
factor <- as.factor(c(1,2,4,2,5,99))
factor
```

```
[1] 1  2  4  2  5  99
Levels: 1 2 4 5 99
```

```
as.numeric(factor)
```

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

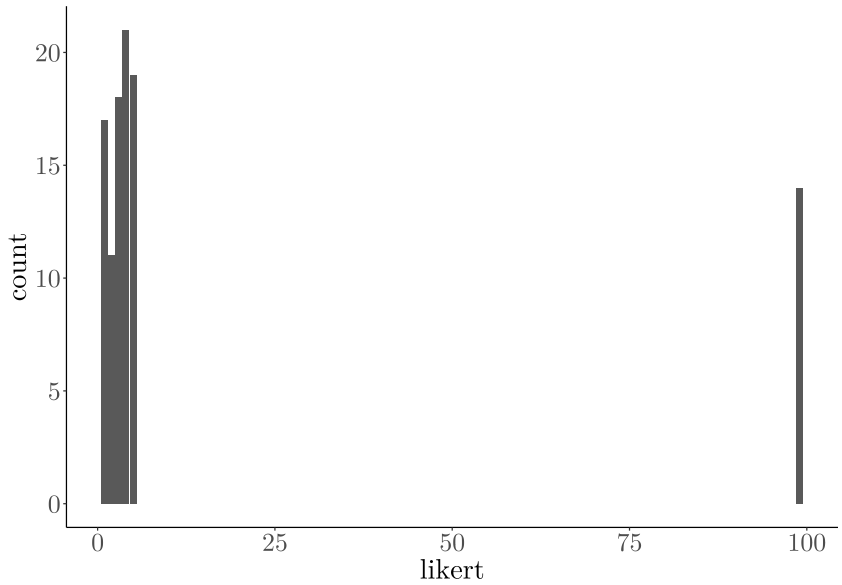
What R does here is to order the elements of the factor (in this case by size since the elements are numerical–if we had entered characters they would be ordered alphabetically). It then returns the position in this order as the numerical value.

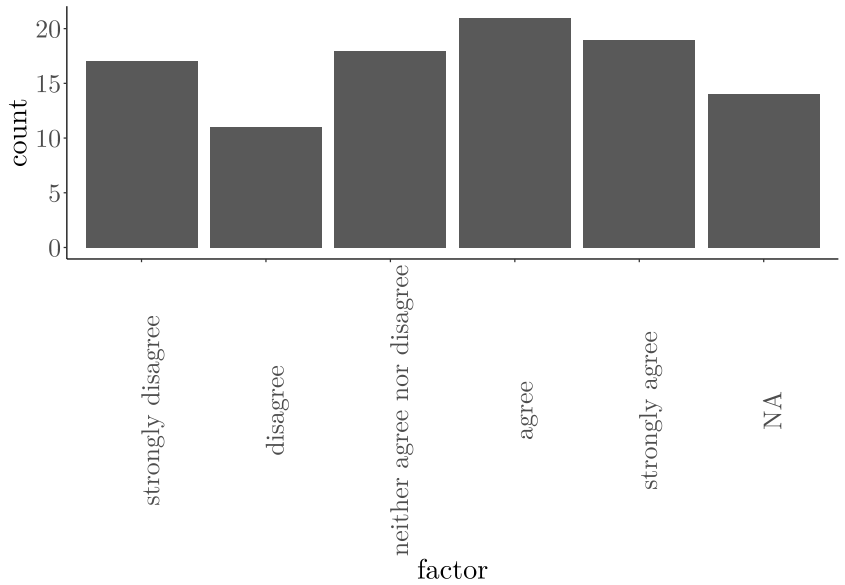
Why `factors` are still usefull

Despite this odd behavior, factors can be useful for plotting. Say we have a Likert scale response such as in the Tunisia survey. Plotting this variable as a vector allows use to control the labels:

```
likert <- sample(c(1:5, 99), size = 100, replace = TRUE)
factor <- factor(likert,
                 levels=c(1,2,3,4,5,99),
                 labels=c("strongly disagree",
                          "disagree",
                          "neither agree nor disagree",
                          "agree",
                          "strongly agree",
                          "NA"))
example <- data.frame(likert=likert,
                      factor=factor)
```

Plot both `likert` and `factor` to see the difference





Data structures

1. *Vectors* (numerical or character, only one class): `id <- c(1,2,3,4,5)`, `gender <- c("m","f","m","f","m")`

Data structures

1. *Vectors* (numerical or character, only one class): `id <- c(1,2,3,4,5)`, `gender <- c("m","f","m","f","m")`
2. *Matrices* (two dimensional, only one class): `matrix1 <- matrix(c(1,2,3,4,5,23,34,19,46,38), ncol = 2)`

Data structures

1. *Vectors* (numerical or character, only one class): `id <- c(1,2,3,4,5)`, `gender <- c("m","f","m","f","m")`
2. *Matrices* (two dimensional, only one class): `matrix1 <- matrix(c(1,2,3,4,5,23,34,19,46,38), ncol = 2)`
3. *Lists* (collection of elements of different classes): `list1 <- list(id, gender, matrix1)`

Data structures

1. *Vectors* (numerical or character, only one class): `id <- c(1,2,3,4,5)`, `gender <- c("m","f","m","f","m")`
2. *Matrices* (two dimensional, only one class): `matrix1 <- matrix(c(1,2,3,4,5,23,34,19,46,38), ncol = 2)`
3. *Lists* (collection of elements of different classes): `list1 <- list(id, gender, matrix1)`
4. *Data frames* (two dimensional structure, different classes): `df1 <- data.frame(id = c(1,2,3,4,5), gender = c("m","f","m","f","m"))`

Data structures

R saves the output of many more complex functions as lists. The following code creates random `x` and `y` variables and regresses y on x:

```
y <- runif(100, min = 0, max = 1)
x <- runif(100, min = 10, max = 20)
model1 <- lm(y ~ x)
names(model1)
```

```
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "xlevels"       "call"          "terms"         "model"
```

Data structures

```
summary(model1)
```

```
Call:
lm(formula = y ~ x)

Residuals:
     Min       1Q   Median       3Q      Max
-0.54588 -0.26046  0.00935  0.27056  0.52087

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.74395    0.16719   4.450 2.28e-05 ***
x           -0.01480    0.01114  -1.329    0.187
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.2993 on 98 degrees of freedom
Multiple R-squared:  0.0177,    Adjusted R-squared:  0.007678
F-statistic: 1.766 on 1 and 98 DF,  p-value: 0.187
```

Operators

1. **Relational** operators
   1. `<` less than
   2. `>` greater than
   3. `<=` less than or equal to
   4. `>=` greater than or equal to
   5. `==` equal (identical) to
   6. `!=` not equal to
2. **Logical** operators
   1. `!` not ($\neg$ in propositional logic)
   2. `&` and ($\wedge$ in propositional logic)
   3. `|` or ($\vee$ in propositional logic)

Operators return an object of the class "logical" which can have the values `TRUE` or `FALSE`.

Operators

What will this evaluate to and why?

- `1 < 2 & "left"=="right"` — ?
- `1 < 2 | "left"=="right"` — ?
- `1 < 2 | !"left"=="right"` — ?
- `!(1 < 2 | "left"=="right")` — ?

Operators

What will this evaluate to and why?

- `1 < 2 & "left"=="right"` — `FALSE`
- `1 < 2 | "left"=="right"` — `TRUE`
- `1 < 2 | !"left"=="right"` — `TRUE`
- `!(1 < 2 | "left"=="right")` — `FALSE`

Conditional statements

All programming languages have a way of making conditional statements and R is no exception:

```
if ([test]) {
  [Action if test is passed]
} else {
  [Action if test is not passed]
}
```

Try writing code which returns “positive” if a number is positive and “negative or zero” otherwise

Conditional statements

```
x <- 0
if (x>0) {
  print("positive")
} else {
  print("negative or zero")
}
```

```
[1] "negative or zero"
```

Conditional statements

You can nest if/else statements:

```
x <- 0
if (x>0) {
  print("positive")
} else if (x==0) {
  print("zero")
} else {
  print("negative")
}
```

```
[1] "zero"
```

Let's return to the issue of transliterating Turkish names

Transliterating the names of Turkish MPs

Read the CSV file with the names of Turkish MPs from GitHub.

```
mps <- read_csv("turkish_mps.csv")
head(mps,5)
```

```
# A tibble: 5 x 3
  name                    constituency          session
  <chr>                   <chr>                   <dbl>
1 Abdülgafur Efendi       Karesi (Balıkesir)          1
2 Abdülgani Ensari        Siverek                     1
3 Abdülgani Ertan         Muş                         1
4 Abdülhak Adnan Adıvar   İstanbul                    1
5 Abdülhak Tevfik Gençtürk Dersim (Tunceli)           1
```

Make sure to either download the file and save it in the correct directory, or to replace the file name with the URL to the raw version.

Transliterating the names of Turkish MPs

Even though Turkey adopted the Latin alphabet in 1928 as part of the reforms promoted by Mustafa Kemal ("Atatürk"), the Turkish alphabet contains some non-standard letters. Here is a list:

Turkish Small Letter	Latin Small Equivalent	Turkish Capital Letter	Latin Capital Equivalent
ç	c	Ç	C
ş	s	Ş	S
ğ	g	Ğ	G
		İ	I
ı	i		
ö	o	Ö	O
ü	u	Ü	U

User-defined functions

We would like to write a function which returns the Latin equivalent if a special Turkish character is provided as input. See how far you can get by following these steps

1. Create a vector which maps the special characters to their Latin equivalents (for your convenience, I posted a table on the GitHub page so that you can copy/paste the special characters if you like)
2. Write a function which accepts a letter as input and returns the Latin equivalent if the input is a special Turkish character. If the input is not a special Turkish character, the function should return the original input.

Step 1: The mapping vector

```
turkish_letters <- c(
    "ç" = "c","ğ" = "g","ı" = "i","İ" = "I",
    "ş" = "s","ö" = "o","ü" = "u","Ç" = "C",
    "Ğ" = "G","Ş" = "S","Ö" = "O", "Ü" = "U"
  )
```

This creates a character vector with the Turkish letters as names and the Latin equivalents as entries.

Step 1: The mapping vector

You can verify the result like this:

```
turkish_letters
```

```
  ç   ğ   ı   İ   ş   ö   ü   Ç   Ğ   Ş   Ö   Ü
"c" "g" "i" "I" "s" "o" "u" "C" "G" "S" "O" "U"
```

```
names(turkish_letters)
```

```
 [1] "ç" "ğ" "ı" "İ" "ş" "ö" "ü" "Ç" "Ğ" "Ş" "Ö" "Ü"
```

```
cat(turkish_letters)
```

```
c g i I s o u C G S O U
```

Step 2: The function

Now we want to create a function which maps the Turkish letters to the Latin equivalent.

Note that our mapping vector can return the equivalent value like this:

```
turkish_letters["ç"]
```

```
  ç
"c"
```

This returns the Latin character at the vector position named ç

Step 2: The function

Generally, R functions are defined in the following way:

```
[name of the function] <- function([input]) {
  output <- [some transformation of the input]
  return(output)
}
```

Write a basic function which takes a special character as input and returns the Latin equivalent based on our mapping vactor.

Step 2: The function

```
tr_to_lat <- function(x) {
  turkish_letters <- c(
    "ç" = "c","ğ" = "g","ı" = "i","İ" = "I",
    "ş" = "s","ö" = "o","ü" = "u","Ç" = "C",
    "Ğ" = "G","Ş" = "S","Ö" = "O", "Ü" = "U"
  )
  latin_letter <- turkish_letters[x]
  return(latin_letter)
}
```

What happens to this function if we enter a non-special character?

Step 2: The function

```
tr_to_lat("d")
```

```
<NA>
  NA
```

How can we avoid this?

Step 2: The function

```
tr_to_lat("d")
```

```
<NA>
  NA
```

How can we avoid this?

Add an if/else statement to the function to return the Latin equivalent if the input is indeed a special Turkish character, and the original input otherwise.

Hint: you can check whether an element is in a vector with `%in%`, e.g. `"ç" %in% names(turkish_letters)`.

Step 2: The function

```
tr_to_lat <- function(x) {
  turkish_letters <- c(
    "ç" = "c","ğ" = "g","ı" = "i","İ" = "I",
    "ş" = "s","ö" = "o","ü" = "u","Ç" = "C",
    "Ğ" = "G","Ş" = "S","Ö" = "O", "Ü" = "U"
  )
  if (x %in% names(turkish_letters)) {
    latin_letter <- as.character(turkish_letters[x])
  } else {
    latin_letter <- x
  }
  return(latin_letter)
}
```

Does it work?

```
tr_to_lat("ç")
```

```
[1] "c"
```

```
tr_to_lat("d")
```

```
[1] "d"
```

Sant'Anna
School of Advanced Studies – Pisa

This works for single letters, how can we make it work for entire words?

`for` loops

One option is to split the word into single letters and apply the function to all letters. We can do this with a loop by following these steps:

1. Use the `strisplit()` function to split the name into separate letters
2. Use a `for` loop to pass each letter to our function
3. Paste the resulting letters together and return the new spelling of the name

Step 1: strisplit()

The `strisplit()` function splits an input string (a word) into components, based on a specified split delimiter.

```
strsplit([input],split="[split delimiter]")
```

For example:

```
strsplit("name","m")
```

```
[[1]]
[1] "na" "e"
```

or:

```
strsplit("name",character())
```

```
[[1]]
[1] "n" "a" "m" "e"
```

Step 1: `strisplit()`

We want each letter individually, so we write:

```
letters <- strsplit("name",character())[[1]]
```

The `[[1]]` element in the end of the expression extracts only the letters from the `list` returned by the `strsplit()` function.

Step 2: The `for` loop

```
for (i in 1:10) {
  print(i)
}
```

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

```
words <- c("this","is",
           "a","for",
           "loop")
for (w in words) {
  print(w)
}
```

```
[1] "this"
[1] "is"
[1] "a"
[1] "for"
[1] "loop"
```

Step 2: The `for` loop

Now we can build a new function which takes a name as input, splits it into letters, and transliterates the letters if necessary:

```
transl_names <- function(name) {
  letters <- strsplit(name, character())[[1]]
  converted_letters <- NULL
  for (l in letters) {
    converted_letters <- c(converted_letters,tr_to_lat(l))
  }
  return(converted_letters)
}
```

Step 2: The `for` loop

Let's try the new function with the first name:

```
transl_names(mps$name[1])
```

```
 [1] "A" "b" "d" "u" "l" "g" "a" "f" "u" "r" " " " " "E" "f" "e" "n" "d" "i
```

That's not quite what we were looking for. We still need to paste the letters together.

Step 3: Pasting the new name together

```
transl_names <- function(name) {
  letters <- strsplit(name, character())[[1]]
  converted_letters <- NULL
  for (l in letters) {
    converted_letters <- c(converted_letters,tr_to_lat(l))
  }
  new_name <- paste(converted_letters, collapse = "")
  return(new_name)
}
transl_names(mps$name[1])
```

```
[1] "Abdulgafur Efendi"
```

Applying the function

Now we can apply our function and create a `name_clear` variable in the `mps` data frame:

```
test <- mps %>%
  mutate(name_clean=transl_names(name)) %>%
  select(name, name_clean)
head(test)
```

```
# A tibble: 6 x 2
  name                      name_clean
  <chr>                     <chr>
1 Abdülgafur Efendi         Abdulgafur Efendi
2 Abdülgani Ensari          Abdulgafur Efendi
3 Abdülgani Ertan           Abdulgafur Efendi
4 Abdülhak Adnan Adıvar     Abdulgafur Efendi
5 Abdülhak Tevfik Gençtürk Abdulgafur Efendi
6 Abdülhalim Çelebi         Abdulgafur Efendi
```

Why does this not work as expected?

Another loop

Our function accepts only a single name as input, but we provide a `vector` of names (namely the 12,992 names in `mps$name`).

We need to adapt the function to accept a `vector` instead. To do this, we can include another `for` loop which loops over the entries in a `vector` of names.

Give it a try!

Another loop

```
transl_names <- function(name_vec) {
  output <- character(length(name_vec))

  for (i in seq_along(name_vec)) {
    name <- name_vec[i]
    letters <- strsplit(name, NULL)[[1]]
    converted_letters <- NULL

    for (l in letters) {
      converted_letters <- c(converted_letters,
                             tr_to_lat(l))
    }
    output[i] <- paste(converted_letters, collapse = "")
  }
  return(output)
}
```

Applying the function (once more)

```r
test <- mps %>%
  mutate(name_clean=transl_names(name)) %>%
  select(name, name_clean)
head(test)
```

```
# A tibble: 6 x 2
  name                      name_clean
  <chr>                     <chr>
1 Abdülgafur Efendi         Abdulgafur Efendi
2 Abdülgani Ensari          Abdulgani Ensari
3 Abdülgani Ertan           Abdulgani Ertan
4 Abdülhak Adnan Adıvar     Abdulhak Adnan Adivar
5 Abdülhak Tevfik Gençtürk Abdulhak Tevfik Gencturk
6 Abdülhalim Çelebi         Abdulhalim Celebi
```

Exercises

1. Look at how the Likert-scale responses are coded in the Tunisia survey. Write a function to recode them so that higher value reflect more agreement.
2. Write a `for` loop to apply the function from the previous exercise to a list of variables.
3. For loops work, but they are not necessarily the most efficient way of doing things. Paste your previous solution into ChatGPT and ask it to provide a solution based on `lapply`.