Introduction to Applied Data Science

Lecture 2: Introduction to Programming

Bas Machielsen
Utrecht University
2024-04-22

Lecture 2: Introduction to Programming

- Overview of this class:
  - Lecture 1: Introduction to Data Science & R
  - *This lecture*: Lecture 2: Introduction to Programming
  - Lecture 3: Getting Data, API & Databases
  - Lecture 4: Getting Data, Web Scraping
  - Lecture 5: Transforming and Cleaning Data
  - Lecture 6: Spatial and Network Data
  - Lecture 7: Text Data & Text Mining
  - Lecture 8: Data Science Project

R Basics

Getting Started

- At its basics, R is basically a **calculator on steroids**.
- We can type an arithmetic expression into our script, then source it into the console and receive a result:

```
2+2
```

```
## [1] 4
```

- There is a huge range of **mathematical functions** in R, some of the most useful include `log`, `exp`, and `sqrt`:

```
sqrt(4)
```

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

- It's important to realize that when you run code as we've done above, the result of the code (or value) is **only displayed in the console**.
- This can sometimes be useful, but it is usually much more practical to store the value(s) in a **object.**

Objects

- At the heart of almost everything you will do in R is the concept that everything in R is an **object**.
    - These objects can be almost anything, from a **single number or character string** (like a word) to more complex structures like a plot output or a summary of a statistical model.
- To create an object we simply give the object a **name**.
    - We can then assign a **value** to this object using the assignment operator `←`

```
hello ← 1
```

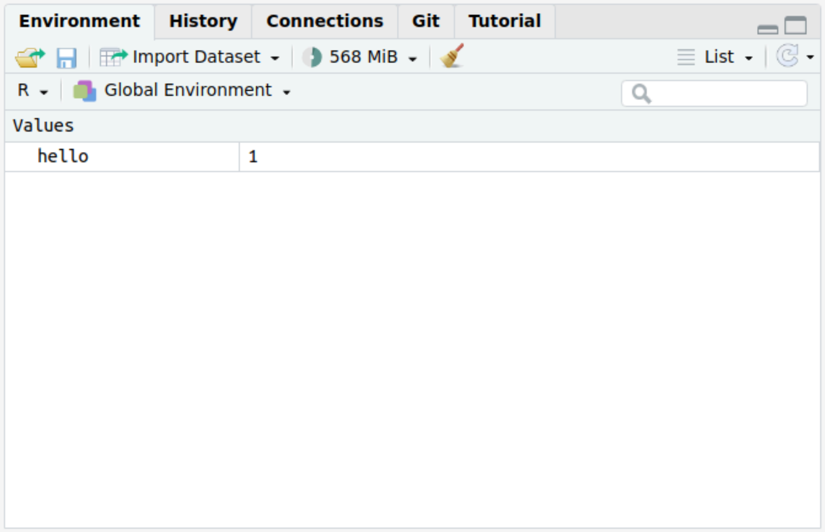
- We refer to 1 as the **value** of the object and to `hello` as the **name** of the object
- To view the value of the object you simply type the name of the object:

```
hello
```

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

Memory

- All of the objects you create will be stored in R's **memory**:
  - You can view all the objects in your workspace in RStudio by clicking on the **Environment** tab in the top right hand pane.



```
Values
hello    1
```

- There are many different types of values that you can assign to an object. For example:

```r
sentence <- "Hello my name is Bas"
```

Looking Up Objects

- Here we have created an object called `sentence` and assigned it a value of "Hello, my name is Bas", which is a character string.
  - Notice that we have **enclosed the string in quotes**. If you forget to use the quotes you will receive an error message:

```
sentence <- Hello
```

```
## Error in eval(expr, envir, enclos): object 'Hello' not found
```

- The reason is that R, like every other programming language, reserves **unquoted names** for objects that may or may not have been stored in memory. Hence, if you type:

```
Hello
```

```
## Error in eval(expr, envir, enclos): object 'Hello' not found
```

in the console, you are telling R: "Go to your memory. Look up what value is given to the object called Hello."

Looking Up Objects

- However, as you can see in your **memory (Environment, upper right window)**, there is no object called Hello.
    - Hence, R will give you an error, saying it cannot find this object in memory.
- Secondly, computer programming languages *cannot* handle spaces well.
    - Therefore, as a rule, always give things names without spaces. So this doesn't work:

```
my object ← "hi"
```

```
## Error: <text>:1:4: unexpected symbol
## 1: my object
##        ^
```

- But this does:

```
my_object ← "hi"
```

Doing Things With Objects

- You can also **overwrite** objects in your memory:

```
my_object ← "hi again"
```

- Once we have created a few objects, we can do things with our objects. For example, the following code performs a simple calculation using objects:

```
numerator ← 6
denominator ← 5

numerator/denominator
```

```
## [1] 1.2
```

Error Messages

- As you first start programming in R, you'll encounter **error messages** frequently. For example:

```
object1 ← "hello"
object2 ← "world!"
object3 ← object1 + object2
```

```
## Error in object1 + object2: non-numeric argument to binary operator
```

- Which means you are doing something illegal or **unexpected** with your objects: The error message is essentially telling you that either one or both of the objects aren't numbers and therefore can't be added

Error Messages

- Another error message that you'll get quite a lot when you first start using R is Error: object 'X' not found. For example:

```
new_object <- c(object1, object3)
```

```
## Error in eval(expr, envir, enclos): object 'object3' not found
```

- R returns an error message because we **haven't created** the object `object1` yet. Another clue that there's a problem with this code is that, if you check your environment, you'll see that object `object1` has not been created.

Functions

- Up until now we've been creating **simple objects** by directly assigning a single value to an object.
    - We want to create **more complicated objects** for potentially more complex tasks
- The first function we will learn about is the `c()` function.
- `c()` is short for concatenate and can be used to store a **series of values** in a data structure called a **vector**

```
object ← c(1,2,3,4,5)
object
```

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

Functions

- When you use a function in R, the **function name** is always followed by a pair of round brackets even if there's nothing contained between the brackets.
- The argument(s) of a function are placed inside the **round brackets** and are separated by **commas**. You can think of an argument as way of customizing the use or behavior of a function.
    - In the example on the previous slide, the arguments are the numbers we want to concatenate.
- How do you know which function to use for what task?
    - Each function will always have a **help document** associated with it which will explain how to use the function: try typing `?c` in the console
    - Google and ChatGPT can also help out

Examples of Functions

- Some example of functions we might use in the future are:

```
mean(object)
```

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

```
var(object)
```

```
## [1] 2.5
```

```
median(object)
```

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

```
length(object)
```

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

- Can you guess what these do?

More Useful Functions

- Sometimes it can be useful to create a **vector** that contains a regular sequence of values in steps of one. We can do that in the following, special way:

```
1:10
```

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

- Other useful functions for generating vectors of sequences include the `seq()` and `rep()` functions.

```
rep(2, 10)
```

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

```
seq(from = 1,
    to = 10,
    by = 1.5)
```

```
## [1]  1.0  2.5  4.0  5.5  7.0  8.5 10.0
```

Indexing

- To extract one or more values from a vector we use the `[ ]` notation.

```r
new_object ← c(4, 23, 1)

new_object[2]
```

```
## [1] 23
```

```r
new_object[c(1, 3)]
```

```
## [1] 4 1
```

```r
new_object > 1
```

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

```r
new_object[new_object > 1]
```

```
## [1]  4 23
```

Vectorization

- One of the cool things about R functions is that most of them are **vectorized**.
    - This means that the function will operate on **all elements of a vector** without needing to apply the function on each element separately.

```
new_object * 3
```

```
## [1] 12 69  3
```

```
new_object + 3
```

```
## [1]  7 26  4
```

Data Structures

Data Types

- R has a couple basic types of data; numeric, integer, logical, and character.

- We have already seen a couple of them.

    - **Numeric** data are numbers that contain a decimal. Actually they can also be whole numbers but we'll gloss over that.
    - **Integers** are whole numbers (those numbers without a decimal point).
    - **Logical** data take on the value of either TRUE or FALSE. There's also another special type of logical called NA to represent missing values.
    - **Character** data are used to represent string values. You can think of character strings as something like a word (or multiple words).

- You can check which data type an object is by using the `class()` function:

```r
class(new_object)
```

```
## [1] "numeric"
```

- You can also change variables from one class to another with `as.character()`, `as.numeric()`, etc.

More Complicated Data Structures

- R also has more complicated data structures:
- The next data structure we will quickly take a look at is a **list**.

```r
my_list ← list(name = "John", age = 30, city = "New York")
my_list
```

```
## $name
## [1] "John"
##
## $age
## [1] 30
##
## $city
## [1] "New York"
```

Data Structures: Lists

- Here's a slightly more complicated list:

```r
hello ← list(a=10, b="6", 5, d = list(a=11, f=1:20, "My name is Bas"))
hello
```

```
## $a
## [1] 10
##
## $b
## [1] "6"
##
## [[3]]
## [1] 5
##
## $d
## $d$a
## [1] 11
##
## $d$f
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
##
## $d[[3]]
## [1] "My name is Bas"
```

Selecting From A List

- Selecting from a list is done with the help of `[[]]` syntax
- For example:

```
hello[['a']]
```

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

.. will give you access to the *value* of the element `a`.

```
hello[['d']]
```

```
## $a
## [1] 11
##
## $f
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
##
## [[3]]
## [1] "My name is Bas"
```

Indexing a List

- As you can see, some objects inside our `list` do not have a name.
- Those can be selected on the basis of **position**:

```r
hello[[3]]
```

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

```r
hello[['d']][[3]]
```

```
## [1] "My name is Bas"
```

- In fact, this also works for elements that *do* have a name:

```r
hello[[1]] #the same as hello[['a']]
```

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

Data Structures: data.frames

- By far the most **commonly used data structure** to store data in is the `data.frame`.
- Typically, in a data frame each row corresponds to an individual observation and each column corresponds to a different measured or recorded variable.
- A useful way to think about data frames is that they are essentially made up of **several vectors** (columns) with each vector containing its own data type but the data type can be different between vectors.

```r
my_df <- data.frame(name = c("John", "Alice", "Bob"),
                    age = c(30, 25, 35),
                    city = c("New York", "Los Angeles", "Chicago"))

my_df
```

```
##    name age        city
## 1  John  30    New York
## 2 Alice  25 Los Angeles
## 3   Bob  35     Chicago
```

Indexing data.frames

- Selecting from a `data.frame` works similarly, but with a slightly different syntax.
    - We select the columns of a data.frame by the `data.frame[i]` syntax
    - For example:

```
my_df[2]
```

```
##   age
## 1  30
## 2  25
## 3  35
```

- We select the rows of a data.frame by the `data.frame[i,]` syntax
- For example:

```
my_df[2,]
```

```
##    name age        city
## 2 Alice  25 Los Angeles
```

- Both ways take a data.frame as an input and as an output (check this with the `class` function)

Indexing data.frames

- It is also possible to index data.frames by column names:

```
my_df['name']
```

```
##    name
## 1  John
## 2 Alice
## 3   Bob
```

- Or to take a slice of a data.frame by position:

```
my_df[1:2, c(1,3)]
```

```
##    name        city
## 1  John    New York
## 2 Alice Los Angeles
```

- Or a combination of both:

```
my_df[1, c('name', 'city')]
```

```
##   name     city
## 1 John New York
```

- Another way to select columns of a data.frame is this:

```
my_df$name
```

```
## [1] "John"  "Alice" "Bob"
```

Tibbles Instead of Data.Frames

- I recommend that you use the function `tibble` from the `tidyverse` packages instead of `data.frame`
    - This function is nearly identical, but just slightly more flexible
    - Compare this:

```
data.frame(x = 1:3, y = x + 1)
```

```
## Error in data.frame(x = 1:3, y = x + 1): object 'x' not found
```

- To this:

```
library(tidyverse)
tibble(x = 1:3, y = x + 1)
```

```
## # A tibble: 3 × 2
##       x     y
##   <int> <dbl>
## 1     1     2
## 2     2     3
## 3     3     4
```

Summarizing data.frames

Data.frames as Spreadsheets

- Usually (and in this course), your data.frame is like a spreadsheet.
    - Pretty much every data file you import (a `.csv`, a `.xlsx`, etc.) will be converted into a data.frame
- It contains **useful information** that you might want to have a closer look at
- There exist various ways of **summarizing** data.frames.
- In this course, we'll first use the apparatus we have just learned to look more closely at data.frames
- Afterwards, we'll have a close look at the **tidyverse**, a set of packages making working with data.frames a little bit easier

Dealing with `NA` Observations

- Sometimes your data contain `NA` observations
- This can be problematic because of the following:

```
data_na <- tibble(a = c(1:2, NA, 4), b = 5:8)
data_na
```

```
## # A tibble: 4 × 2
##         a     b
##     <dbl> <int>
## 1       1     5
## 2       2     6
## 3      NA     7
## 4       4     8
```

```
mean(data_na['a'])
```

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

Dealing with `NA` Observations

- There are a couple of solutions for this:
    - Just as we did with vectors, we can also **extract data** from our data frame based on a logical test.
    - We can use all of the logical operators that we used for our vector examples:

- Consider this logical test:

```r
!is.na(data_na['a'])
```

```
##          a
## [1,]  TRUE
## [2,]  TRUE
## [3,] FALSE
## [4,]  TRUE
```

- We select only the rows that aren't NA in column a

```r
data_na[!is.na(data_na['a']), ]
```

```
## # A tibble: 3 × 2
##       a     b
##   <dbl> <int>
## 1     1     5
## 2     2     6
## 3     4     8
```

Filtering Out Observations

- This can also be used in a more general way, when wanting to zoom in on particular parts of a data.frame:

```
data_na
```

```
## # A tibble: 4 × 2
##       a     b
##   <dbl> <int>
## 1     1     5
## 2     2     6
## 3    NA     7
## 4     4     8
```

```
data_na['b'] == 5 | data_na['a'] == 2
```

```
##           b
## [1,]   TRUE
## [2,]   TRUE
## [3,]     NA
## [4,] FALSE
```

```
data_na[data_na['b'] == 5 | data_na['a
```

```
## # A tibble: 3 × 2
##       a     b
##   <dbl> <int>
## 1     1     5
## 2     2     6
## 3    NA    NA
```

Summarizing the Data

- Given that you have **selected the part of the data you care about**, you can use functions we've already seen to summarize the data:

```r
mean(data_na[data_na['b'] == 5 | data_na['b'] == 6, ]$a)
```

```
## [1] 1.5
```

```r
data_na[!is.na(data_na['a']), ]
```

```
## # A tibble: 3 × 2
##         a     b
##     <dbl> <int>
## 1       1     5
## 2       2     6
## 3       4     8
```

```r
median(data_na[!is.na(data_na['a']), ]$b)
```

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

- This is a quite complicated way to compute relatively simple statistics.

The `tidyverse` package

Introduction to `tidyverse`

- The **tidyverse** is a collection of R packages designed for data science. It is based on the principles of tidy data and provides a consistent set of tools for data manipulation, visualization, and analysis.
- Notably, it contains `dplyr`: a package for data manipulation, providing functions for filtering, selecting, mutating, summarizing, and arranging data, and `tidyr`: a package for tidying data, providing functions for reshaping data into tidy formats.
- It can be installed using the `install.packages()` function
- And loaded by:

```r
library(tidyverse)
```

Understanding the Pipe Operator in R

- The pipe operator |> is a powerful tool in the R programming language that simplifies and enhances the readability of code, especially in data analysis workflows.
- It takes the output from one function and uses it as the first argument of the next function in the chain.
- It enables a more natural, left-to-right style of coding, akin to how we read and interpret information.
- Example (more will follow):

```r
# Example without pipe operator
result ← sqrt(mean(c(1, 4, 9, 16)))
```

```r
# Example with pipe operator
result ← c(1, 4, 9, 16) ▷
            mean() ▷
            sqrt()
```

Dealing with NA's and Filtering

- In the `tidyverse` package, dealing with `NA` observations is very easy: we can use the `drop_na` function:
- In addition, we can use the `filter` function, allowing us to filter on a specific variable with respect to `NA` observations:

```
data_na ▷
  drop_na()
```

```
## # A tibble: 3 × 2
##         a     b
##     <dbl> <int>
## 1       1     5
## 2       2     6
## 3       4     8
```

```
data_na ▷
  filter(!is.na(a))
```

```
## # A tibble: 3 × 2
##         a     b
##     <dbl> <int>
## 1       1     5
## 2       2     6
## 3       4     8
```

Filtering

- More generally, we can use `filter` in the same way as we did when devising logical conditions to select variables:

```
data_na |>
  filter(a > 1 | b < 6)
```

```
## # A tibble: 3 × 2
##         a     b
##     <dbl> <int>
## 1       1     5
## 2       2     6
## 3       4     8
```

```
data_na |>
  filter(b == 8)
```

```
## # A tibble: 1 × 2
##         a     b
##     <dbl> <int>
## 1       4     8
```

Finding Summary Characteristics

- Finding summary characteristics can be done with the `summarize` function.
- In addition to the data, the summarize function takes arguments in the form of `name = expression`, where name is the name of the column to be created in the output and expression is the computation to be applied.

```
data_na ▷
  filter(!is.na(a)) ▷
  summarize(mean_a = mean(a), median_b = median(b))
```

```
## # A tibble: 1 × 2
##   mean_a median_b
##    <dbl>    <int>
## 1   2.33        6
```

Grouping By

- It is often necessary to **perform operations on groups** within your data.
- The `group_by` function allows you to group data by one or more variables
- It is often used in conjunction with `summarize`
- Consider this dataset:

```
mtcars ▷ as_tibble()
```

```
## # A tibble: 32 × 11
##      mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
##    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
##  1  21       6  160    110  3.9   2.62  16.5     0     1     4     4
##  2  21       6  160    110  3.9   2.88  17.0     0     1     4     4
##  3  22.8     4  108     93  3.85  2.32  18.6     1     1     4     1
##  4  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1
##  5  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2
##  6  18.1     6  225    105  2.76  3.46  20.2     1     0     3     1
##  7  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4
##  8  24.4     4  147.    62  3.69  3.19  20       1     0     4     2
##  9  22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2
## 10  19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4
## # ℹ 22 more rows
```

Grouping By

- And these summary statistics:

```
mtcars |>
  group_by(cyl) |>
  summarize(max_usage = max(mpg), mean_gears = mean(gear), median_hp = median(hp))
```

```
## # A tibble: 3 × 4
##     cyl max_usage mean_gears median_hp
##   <dbl>     <dbl>      <dbl>     <dbl>
## 1     4      33.9       4.09        91
## 2     6      21.4       3.86       110
## 3     8      19.2       3.29       192.
```

Slicing

- `slice_max` selects rows with the maximum values of a specified variable.
- It is useful when you want to **identify the rows** that correspond to the highest values in a particular variable.
- You can specify the variable based on which you want to find the maximum using the order_by argument.
- By default, `slice_max` selects only the first row with the maximum value. You can specify the number of rows to select using the `n` argument.

```r
slice_max(mtcars, order_by=mpg, n=2)
```

```
##                 mpg cyl disp hp drat    wt  qsec vs am gear carb
## Toyota Corolla 33.9   4 71.1 65 4.22 1.835 19.90  1  1    4    1
## Fiat 128       32.4   4 78.7 66 4.08 2.200 19.47  1  1    4    1
```

- `slice_min` works in the same way

Adding or Changing Variables

- It is also possible to **change variables**, or add new variables to your dataset
- This is done by the `mutate` function
  - Sometimes, the `if_else` function also comes in handy
- The `mutate` function takes the input data frame .data and creates a modified version of it by adding or modifying columns based on the specified transformations.
- You specify the transformations using the `new_column = expression` syntax, where `new_column` is the name of the new column you want to create or the name of an existing column you want to modify, and expression is the R expression that defines how the new column should be calculated.
- The `if_else` function is used to perform vectorized conditional operations on data frames.
  - It is particularly useful when you need to **create or modify** columns based on specific conditions.

Adding or Changing Variables

- The syntax of the `if_else` function is:

```
if_else(condition, true_value, false_value)
```

- Oftentimes, you overwrite your data.frame with the new, `mutate`d data.frame
- For example:

```
mtcars ← mtcars ▷
  mutate(hp_per_cylinder = hp/cyl,
         sustainable = if_else(mpg > 25, 1, 0))

mtcars[c('hp_per_cylinder', 'sustainable')] ▷
  head(4)
```

```
##                 hp_per_cylinder sustainable
## Mazda RX4              18.33333           0
## Mazda RX4 Wag          18.33333           0
## Datsun 710             23.25000           0
## Hornet 4 Drive         18.33333           0
```

Recapitulation

Recapitulation

- We have seen the **basics** of R today
- We have seen various ways to accomplish several common-sensical tasks:
  - We have seen basic operations, ways of selecting and filtering in R
  - And we have also seen an arguably simpler variant of performing these same operations: the **tidyverse**