

# Introduction to R - Day 1

Lidiya Mishieva

22 January, 2026

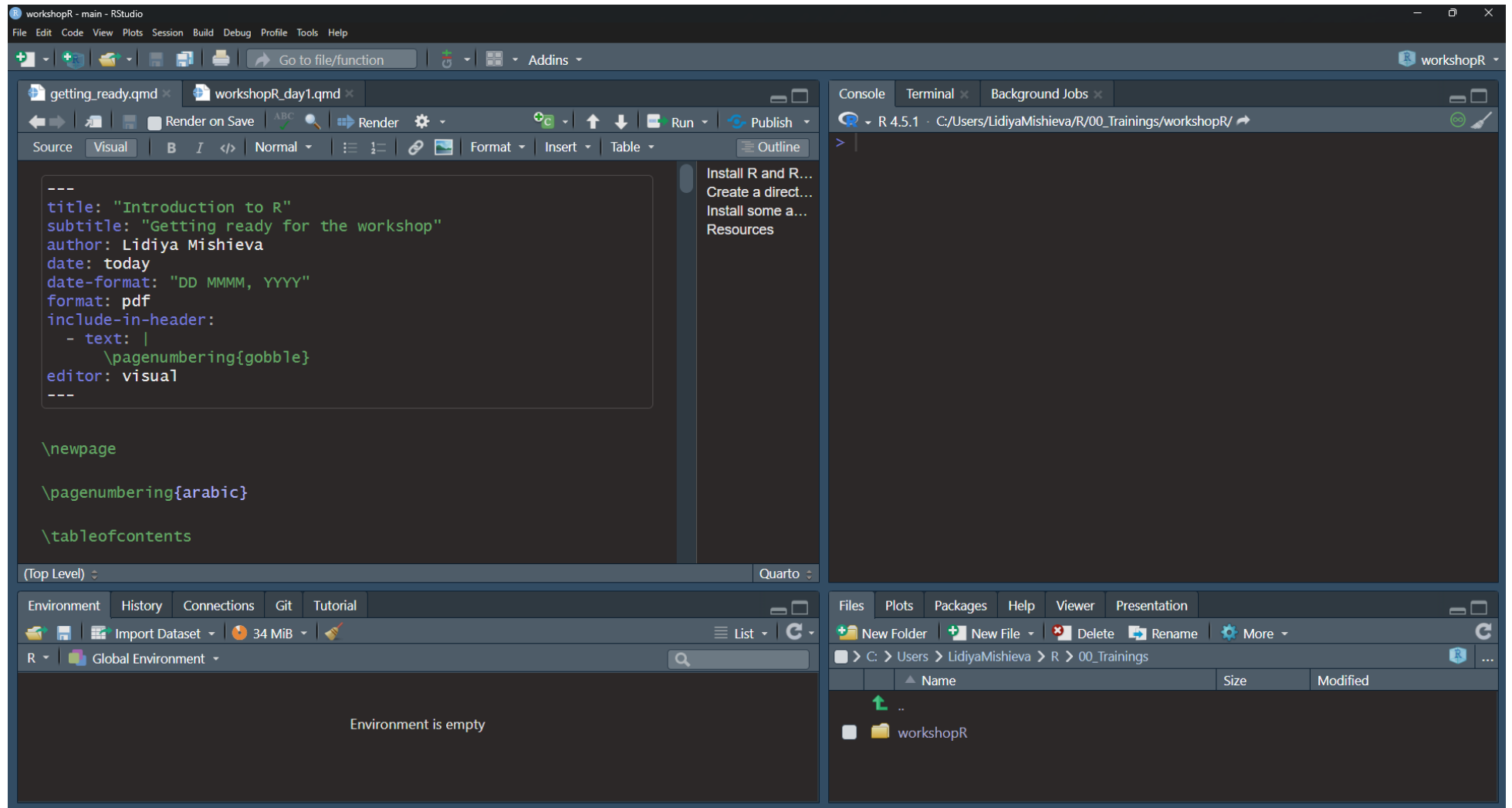
# Intro

- Day 1: basic syntax, classes, objects, functions
- Day 2: base package, tidy programming & tidyverse
- Day 3: RMarkdown / Quarto
- Day 4: Git & RStudio

# RStudio

- IDE with User interface for R
- Programming & dynamic reporting
- R package development

# RStudio user interface



RStudio screen shot

# Basic Syntax

# Basic Syntax - assignment

- Assigning a value to an object

```
1 # assigning a value to an object
2 x <- sqrt(4)
3 y = sqrt(4)
```

- Using `<-` and `=` gives the same result, but `<-` is better for consistency

```
1 x # print x
```

```
[1] 2
```

```
1 y # print y
```

```
[1] 2
```

# Basic Syntax - assignment

- In R objects get overwritten without warning!

```
1 x <- sqrt(9)
2 x
```

[1] 3

- Case-sensitivity

```
1 x <- sqrt(9) # lowercase x
2 X <- sqrt(16) # uppercase x
```

```
1 x # lowercase x
```

[1] 3

```
1 X # uppercase x
```

[1] 4

# Basic Syntax - assignment

- Some symbols are predefined and should not be used as object names, for example:
- T & F

```
1 T # logical operator TRUE
```

```
[1] TRUE
```

```
1 F # logical operator FALSE
```

```
[1] FALSE
```

- t

```
1 # function for transposing a matrix
2 mat <- matrix(c(1,1,1,2,2,2), nrow = 3, ncol = 2)
3 mat
```

```
      [,1] [,2]
[1,]     1     2
[2,]     1     2
[3,]     1     2
```

```
1 t(mat)
```

```
      [,1] [,2] [,3]
[1,]     1     1     1
[2,]     2     2     2
```

# Basic Syntax - assignment

- You can overwrite some them (bad practice)

```
1 T <- 1  
2 T
```

```
[1] 1
```

- Tip: always use TRUE and FALSE instead of T and F

```
> TRUE  
[1] TRUE  
> TRUE <- 1  
  
Fehler in TRUE <- 1 : ungültige (do_set) linke Seite in Zuweisung
```

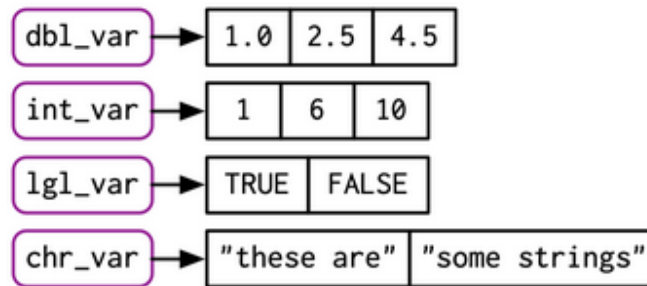
# Objects

- Objects vs. functions
- Object-oriented programming
- Objects are stored in the environment unless explicitly deleted using `rm()`
- Objects stored in the environment can be saved using `save.image()`
- Always start a new script by cleaning up your workspace `rm(list=ls())`

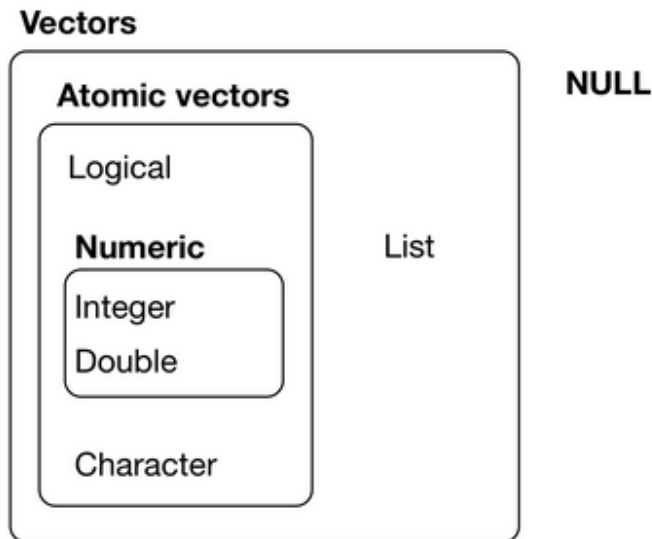
# Objects - vectors

- There are two types of vectors:
  - Atomic vectors
    - logical vectors
    - integer & double (numeric) vectors
    - character vectors
  - Lists or recursive vectors

# Objects - vectors



<https://adv-r.hadley.nz/vectors-chap.html>



The hierarchy of R's vector types (<https://r4ds.had.co.nz/vectors.html>)

# Objects - lists, matrices, dataframes, arrays

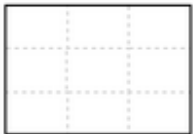
Vector



List



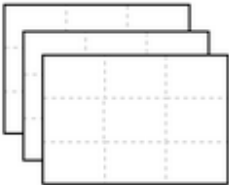
Matrix



Data frame



Array



# Objects - lists, matrices, dataframes, arrays

- Defining different vectors:

```
1 x <- c(1, 2, 3, 4)
2 y <- c("a", "b", "c", "d")
3 z <- 5
4 a <- c("Jayan will be", 25, "years old in", 2028)
```

- Output:

```
1 2 3 4 : numeric
```

```
a b c d : character
```

```
5 : numeric
```

```
Jayan will be 25 years old in 2028 : character
```

# Objects - lists, matrices, dataframes, arrays

- Defining different lists:

```
1 ls1 <- list(c(1, 2, 3, 4))
2 ls2 <- list("a", "b", "c")
3 ls3 <- c(ls1, ls2)
```

# Objects - lists, matrices, dataframes, arrays

- Output:

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

```
[[1]]  
[1] "a"
```

```
[[2]]  
[1] "b"
```

```
[[3]]  
[1] "c"
```

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

```
[[2]]  
[1] "a"
```

```
[[3]]  
[1] "b"
```

```
[[4]]  
[1] "c"
```

# Objects - lists, matrices, dataframes, arrays

- Defining a matrix:

```
1 x <- c(1, 2, 3, 4, 5, 6, 7, 8, 9)
2
3 mat <- matrix(x, nrow = 3)
```

- Output:

	[,1]	[,2]	[,3]
[1,]	1	4	7
[2,]	2	5	8
[3,]	3	6	9

# Objects - lists, matrices, dataframes, arrays

- Defining a data.frame:

```
1 x <- 1:10
2 y <- LETTERS[1:10]
3 z <- c("Ansh", "Raj", "Sahil", "Vikram", "Ravi", "Kareena", "Anita", "Jayan", "Priya", "Sunil")
4
5 df <- data.frame(x, y, z)
```

- Output:

	x	y	z
1	1	A	Ansh
2	2	B	Raj
3	3	C	Sahil
4	4	D	Vikram
5	5	E	Ravi
6	6	F	Kareena
7	7	G	Anita
8	8	H	Jayan
9	9	I	Priya
10	10	J	Sunil

# Functions

# Functions - syntax

- Basic syntax of a function

```
1 my_function <- function(arguments){  
2   statements  
3   return(value)  
4 }
```

## Note

In R you can drop the return statement in most of the cases. Last evaluated object is returned automatically.

- Looking for a function within a package:

```
1 # package::function()  
2 dplyr::add_count()
```

# Functions - custom functions

- In R, you can write your own functions. Here is an example:

```
1 # define a function that produces a sum of two numbers
2 my_sum <- function(a, b) {
3   result <- a + b
4   return(result)
5 }
6
7 # call a function
8 my_sum(a=1, b=2)
```

[1] 3

- Assigning the returned value to an object:

```
1 x <- my_sum(a=1, b=2)
2 x
```

[1] 3

- As long as we remember the order of the arguments, we can just enter the values of the parameters:

```
1 my_sum(1, 2)
```

[1] 3

# Functions - custom functions

- Removing the return() statement - the result is still returned, but not printed if the function is called without assignment:

```
1 my_sum <- function(a, b) {  
2   result <- a + b  
3 }  
4  
5 my_sum(a=1, b=2)  
6 x <- my_sum(a=1, b=2)  
7 x
```

[1] 3

- We can embed function within functions, like we have done here with a for-loop within a custom function:

```
1 # define a function that produces a sum from a list of numbers  
2 my_sum <- function(x) {  
3   total <- 0  
4   for (i in x) {  
5     total <- total + i  
6   }  
7   return(total)  
8 }  
9  
10 # call a function  
11 my_sum(x=list(1, 2, 3))
```

[1] 6

# Functions - custom functions

- Of course the `sum()` function already exists, but often we code specific analyses that we want to apply a number of times - a custom function can be helpful here
- **If you have to copy the code at least 3 times - write a function for it!**

# Functions - conditional execution

- Here is a syntax for a simple if-else function:

```
1 if (condition) {  
2   # code executed when condition is TRUE  
3 } else {  
4   # code executed when condition is FALSE  
5 }
```

- Defining multiple conditions:

```
1 if (this) {  
2   # do that  
3 } else if (that) {  
4   # do something else  
5 } else {  
6   #  
7 }
```

- There is also a function `ifelse()` that is useful when there are not too many conditions, e.g. creating dummy variables:

```
1 var <- 1:10  
2  
3 ifelse(var < 5, 0, 1)
```

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

# Functions - conditional execution

- Still, you can define more conditions here:

```
1 var <- 1:10
2
3 ifelse(var < 5, 0, ifelse( var > 7, 2, 1))
[1] 0 0 0 0 1 1 1 2 2 2
```

# Functions - standard loops: a for-loop

- General syntax:

```
1 for (var in vector)
2 {
3   statement(s)
4 }
```

- Here is another example:

```
1 for (i in 1:4){
2   print(i ^ 2)
3 }
```

```
[1] 1
[1] 4
[1] 9
[1] 16
```

- A break statement:

```
1 for (i in c(1, 2, 3, 0, 9, 7))
2 {
3   if (i == 0)
4   {
5     break
6   }
7   print(i)
8 }
```

```
[1] 1
[1] 2
[1] 3
```

# Functions - standard loops: a for-loop

- A next statement:

```
1 for (i in c(1, 2, 3, 0, 9, 7))
2 {
3   if (i == 0)
4   {
5     next
6   }
7   print(i)
8 }
```

```
[1] 1
[1] 2
[1] 3
[1] 9
[1] 7
```

# Functions - standard loops: a while-loop

- An example:

```
1 # a while-loop
2 i <- 1
3 while (i < 6) {
4   print(i)
5   i <- i + 1
6 }
```

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

- Using the break-statement

```
1 i <- 1
2 while (i < 6) {
3   print(i)
4   i <- i + 1
5   if (i == 4) {
6     break
7   }
8 }
```

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

# Functions - 'apply' family

- Functions from the 'apply' family are essentially loops
- But they allow to run loops in one line (in contrast to for a while loops that require a multi-line syntax)
- There are multiple apply functions:
  - `lapply()`
    - iterates over each element of a list
    - applies a *function* to each element of the list
    - returns a list

# Functions - 'apply' family (Example 1)

- Define a list

```
1 x <- list(a = 1:5)
2 x
```

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

- Change each element iteratively

```
1 # add number 1 to each element of the list
2 lapply(x, function(x){x+1})
```

```
$a
[1] 2 3 4 5 6
```

# Functions - 'apply' family (Example 2)

- Define a list containing 2 subvectors

```
1 x <- list(a = 1:5, b = rnorm(10))
2 x
```

\$a

[1] 1 2 3 4 5

\$b

```
[1] -0.11267553 -0.17569006 0.06637363 0.60727481 1.76045670 0.96760655
[7] -0.39130344 1.91461452 1.04171526 1.70004904
```

- Apply mean computation by looping through a list

```
1 # the result is a list containing vectors
2 lapply(x, mean)
```

\$a

[1] 3

\$b

[1] 0.7378421

# Functions - 'apply' family

- There are some more:
  - `sapply()` essentially the same as `lapply()`, bit simplifying the output
  - `apply()`: Apply a function over the margins of an array
  - `tapply()`: Apply a function over subsets of a vector
  - `mapply()`: Multivariate version of `lapply`

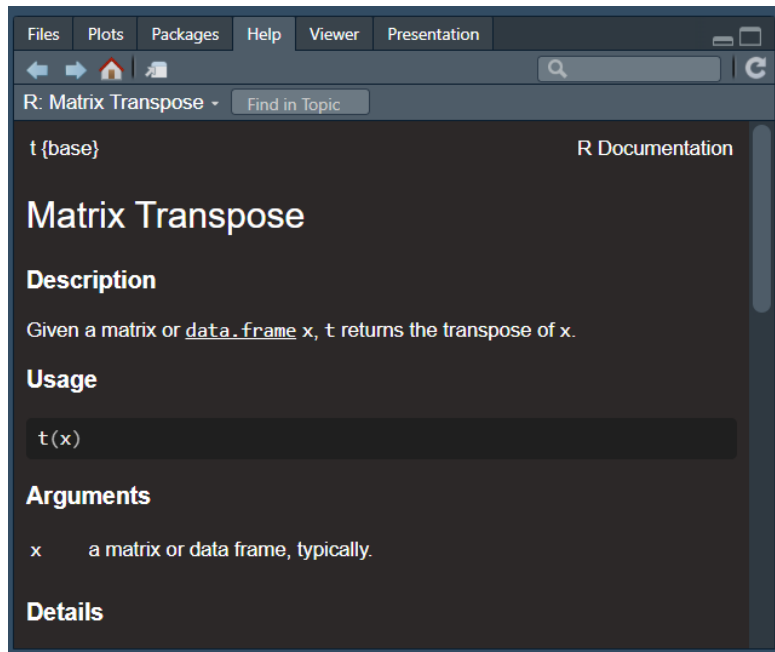
# Packages

- R packages are sets of multiple functions
- Most of them are produced using R - you can write your one function, if you want to combine all your custom functions (you can follow this [guide](#))
- CRAN
- Development versions and packages that are not stored at CRAN
- Need to be loaded into the environment in every session
- Documentation
- Searching for a function in a package
- Printing the code of a function
- Not all packages are maintained ‘forever’
- Versions and `sessionInfo()`
- Project environment using `renv` package

# Getting help

- You can use `?` or `help()` to call an R Documentation of a function or an operator

```
1 ?t
2 help(t)
```



# Getting help

- Some are organized in help topics, check it out

```
1 ?Syntax # operator syntax and precedence
2 ?Arithmetic # arithmetic operators
3 ?Comparison # relational operators
4 ?Extract # operators on vectors and arrays
5 ?Control # control flow
6 ?Logic # logical operators
```

# Getting help

- RSeek (<https://www.rseek.org/>)



how to run linear regression



[All results](#)

[R-project](#)

[Econometrics](#)

[Popular Package](#)

[Issues](#)

[Social Sciences](#)

[Blog](#)

[Documentation](#)

[Package](#)

[Finance](#)

[Vignette](#)

[Psychometrics](#)

About 990,000,000 results (0.59 seconds)

Sort by:

Relevance ▾

**R rookie: Package for Linear Regression : r/RStudio**

[Reddit](#) › [mknry](#) › [r\\_rookie\\_package\\_for\\_linear\\_regression](#)

5 Apr 2021 ... What R package should I install to run linear regressions please? I tried `install.packages("stats")` but I always get error message.

Labeled [Issues](#)

**DHARMa: residual diagnostics for hierarchical (multi-level/mixed ...**

[cran.r-project.org](#) › [web](#) › [packages](#) › [DHARMa](#) › [vignettes](#) › [DHARMa](#)



The 'DHARMa' package uses a simulation-based approach to create readily interpretable scaled (quantile) residuals for fitted (generalized) linear mixed models.

Labeled [Vignette](#) [Documentation](#) [R-project](#)

**How to perform a regression in Rstudio : r/RStudio**

[Reddit](#) › [RStudio](#) › [how\\_to\\_perform\\_a\\_regression\\_in\\_rstudio](#)

4 Oct 2022 ... If it's a linear regression use `m= lm( y~x , data = dataframe )` where m is the model name then use `summary(m)` For logistic use `glm` instead of `lm`.

Labeled [Issues](#)

**Fitting Linear Mixed-Effects Models using lme4**

[cran.r-project.org](#) › [vignettes](#) › [lmer](#)

File Format: PDF/Adobe Acrobat

As for most model-fitting functions in R, the model is described in an `lmer` call by a formula, in this case including both fixed- and random-effects terms. The ...

Labeled [Package](#) [R-project](#)

# Getting help

- Posit community (<https://forum.posit.co/>)
- Posit Cheatsheets (<https://rstudio.github.io/cheatsheets/>)
- Manuals on CRAN (<https://cran.r-project.org/manuals.html>)
- Documentations on CRAN ([https://cran.r-project.org/web/packages/available\\_packages\\_by\\_name.html](https://cran.r-project.org/web/packages/available_packages_by_name.html))
- Stack exchange (<https://stats.stackexchange.com>)
- Stack overflow (<https://stackoverflow.com>)