R-Basics

Session 2: Principles of Programming in R

Dr. Ulrike Göbel

2024-05-22

Where We Left Off and Where We Go

- Debasish has introduced you to R and RStudio as tools.
  What is going on behind the scenes was deliberately left out.

Where We Left Off and Where We Go

- Debasish has introduced you to R and RStudio as tools.
  What is going on behind the scenes was deliberately left out.

- However **R is not simply a black box where data go in and results come out**. It is a full-blown programming language. And it is well suited for a first encounter with programming, because it is a **scripting language: little pieces of code can be directly run** from the RStudio console, or from a script, or from an Rmarkdown file.

Where We Left Off and Where We Go

- Debasish has introduced you to R and RStudio as tools.
  What is going on behind the scenes was deliberately left out.
- However **R is not simply a black box where data go in and results come out**. It is a full-blown programming language. And it is well suited for a first encounter with programming, because it is a **scripting language: little pieces of code can be directly run** from the RStudio console, or from a script, or from an Rmarkdown file.
- **In this session I will introduce the basic principles of actual programming with R**.

Where We Left Off and Where We Go

- Debasish has introduced you to R and RStudio as tools.
  What is going on behind the scenes was deliberately left out.
- However **R is not simply a black box where data go in and results come out**. It is a full-blown programming language. And it is well suited for a first encounter with programming, because it is a **scripting language: little pieces of code can be directly run** from the RStudio console, or from a script, or from an Rmarkdown file.
- **In this session I will introduce the basic principles of actual programming with R**.
- Even if you are never going to write complex code yourself, **knowing the basic concepts allows you to see patterns in existing code** and hence better understand what it is doing, and it also allows you to **make minor useful modifications yourself**.

Programming is Like ...

... orchestrating an invisible world from a script!

Programming is Like ...

... orchestrating an invisible world from a script!

`object`

Programming is Like ...

... orchestrating an invisible world from a script!

`object`

- "something" **inside the computer's memory**

Programming is Like ...

... orchestrating an invisible world from a script!

`object`

- "something" **inside the computer's memory**
- by a **specific structure** it can play a **defined role in specific computations** (a matrix can be transposed, inverted ...)

Programming is Like ...

... orchestrating an invisible world from a script!

`object`

- "something" **inside the computer's memory**
- by a **specific structure** it can play a **defined role in specific computations** (a matrix can be transposed, inverted ...)
- R can recognize and manipulate the structure,
  but how can **you??**

Using a Remote Handle: Assignment

```
symbol <- object
```

Using a Remote Handle: Assignment

```
symbol <- object
```

A **piece of data** stored inside the computer's **memory**

Using a Remote Handle: Assignment

```
symbol <- object
```

A **name**(=**variable**) in your R code

A **piece of data** stored inside the computer's **memory**

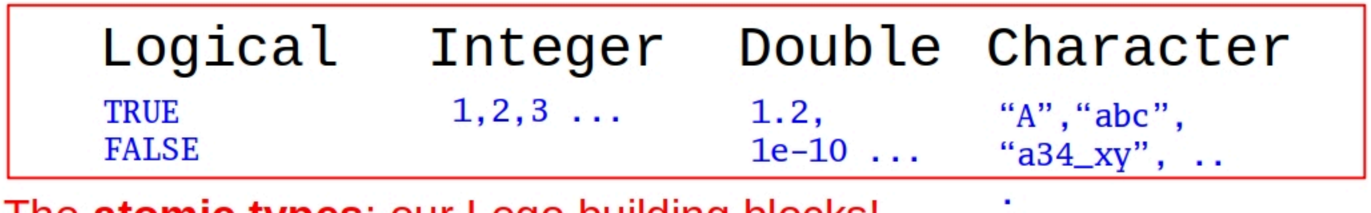
Using a Remote Handle: Assignment

```
symbol <- object
```

A **name**(=**variable**)
in your R code

A **piece of data** stored inside the computer's **memory**

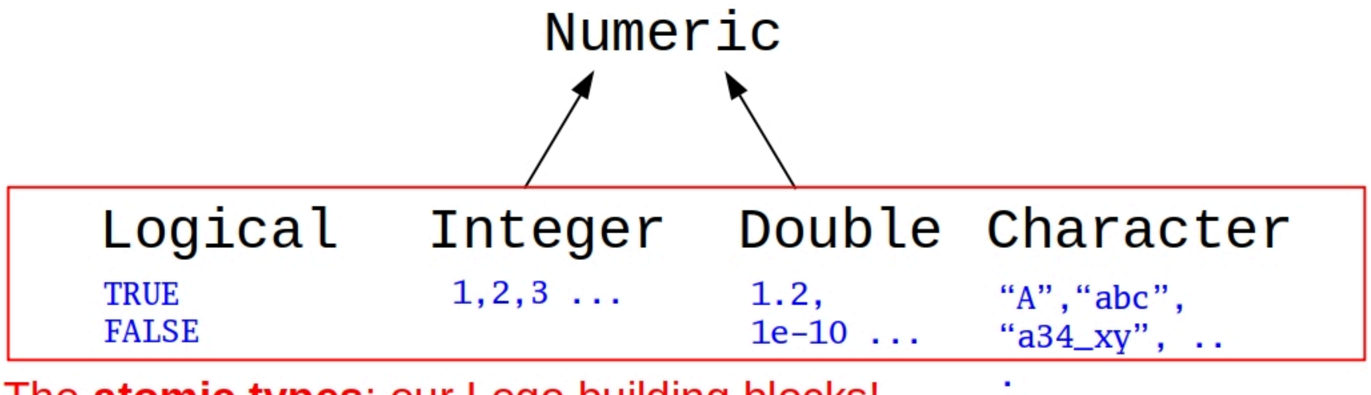
Data Structures



Logical	Integer	Double	Character
TRUE FALSE	1,2,3 ...	1.2, 1e-10 ...	"A","abc", "a34_xy", ...

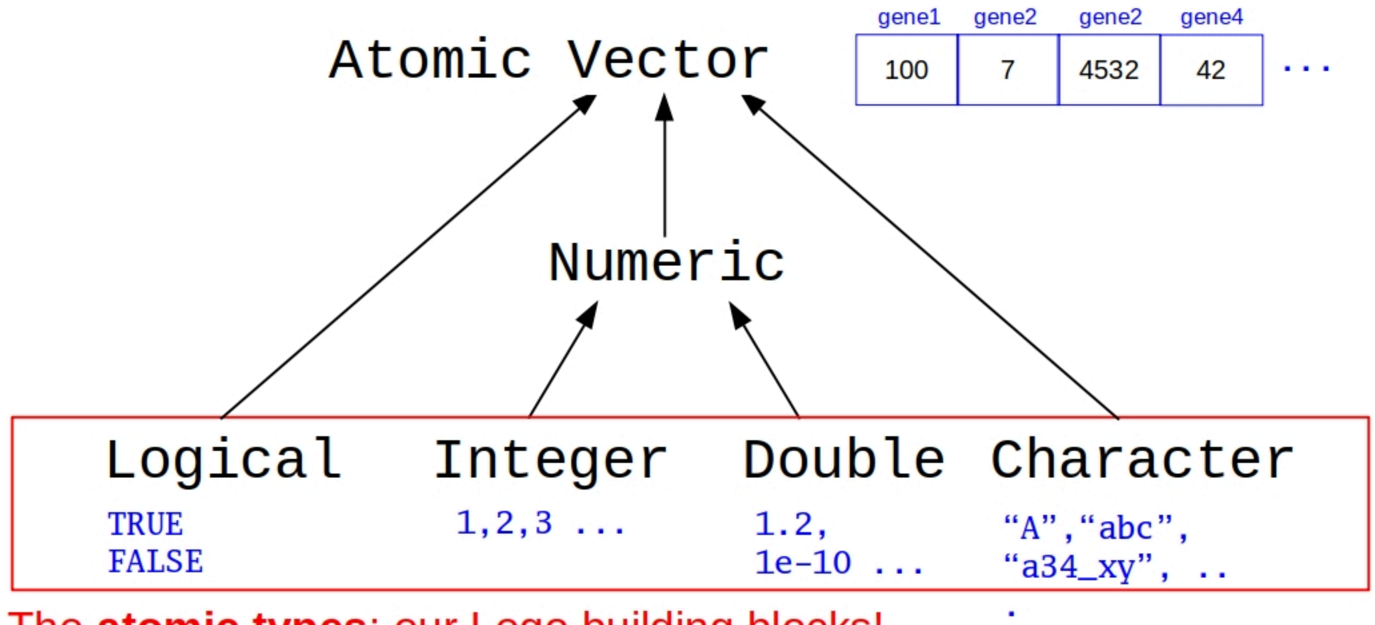
The **atomic types**: our Lego building blocks!

Data Structures



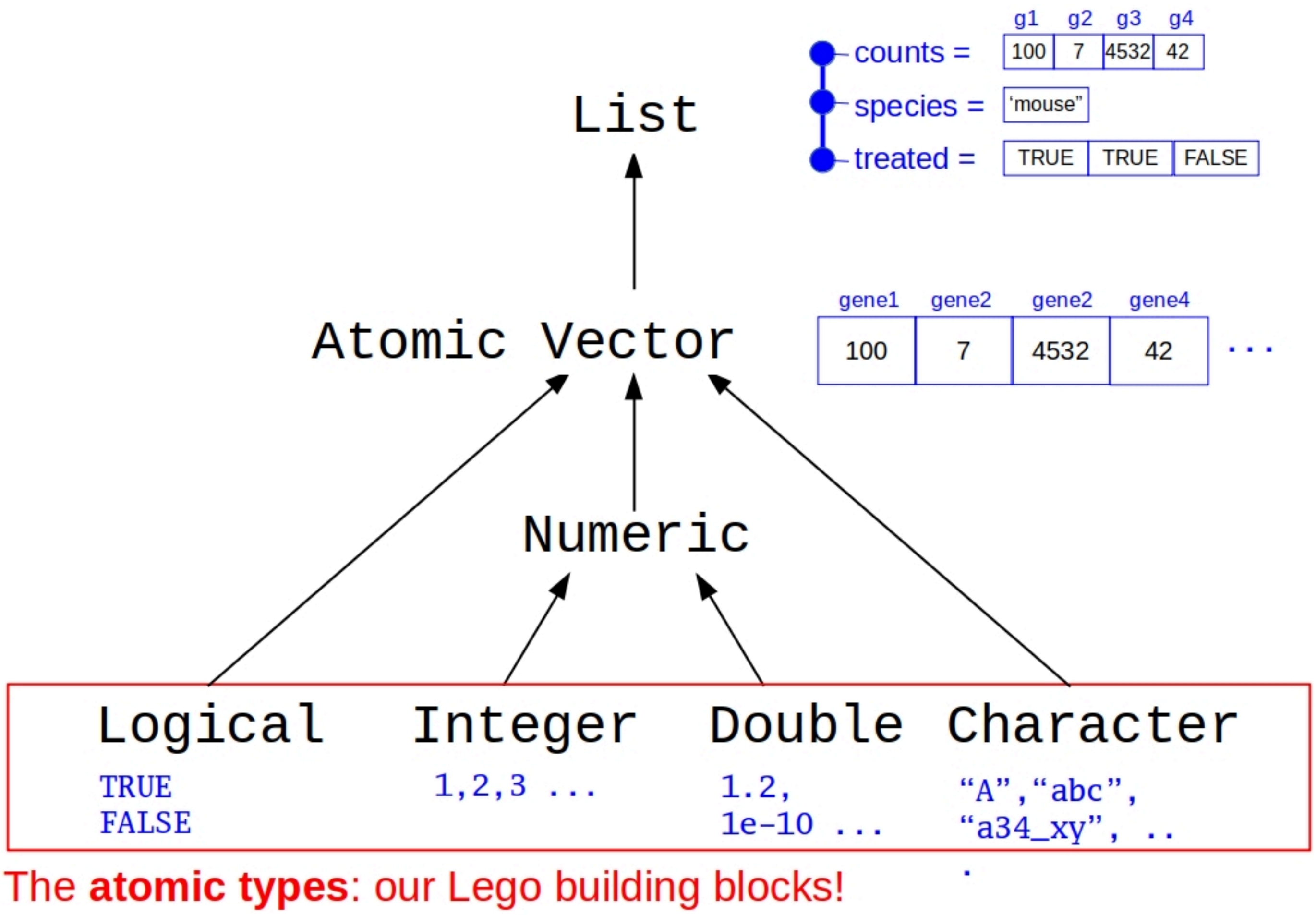
The **atomic types**: our Lego building blocks!

Data Structures



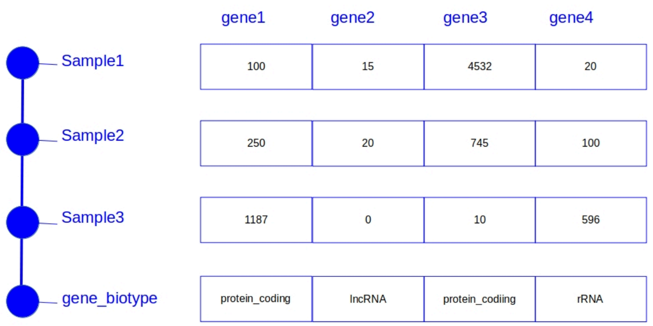
The **atomic types**: our Lego building blocks!

Data Structures



Aside: Two-dimensional Tables

Take a "list" of vectors of identical length …



	gene1	gene2	gene3	gene4
Sample1	100	15	4532	20
Sample2	250	20	745	100
Sample3	1187	0	10	596
gene_biotype	protein_coding	lncRNA	protein_codiing	rRNA

```
my_list <- list(Sample1=c(gene1=100, gene2=15,  gene3=4532, gene4=20),
                Sample2=c(gene1=250,  gene2=20, gene3=745,  gene4=100),
                Sample3=c(gene1=1187,  gene2=0, gene3=10,  gene4=596),
                bioc_type=c(gene1="protein_coding", gene2="lncRNA",gene3="protein_coding", gene4="rRNA"))
```

Aside: Two-dimensional Tables

… convert it to a **base R** table:

```
data.frame(my_list)
```

```
##       Sample1 Sample2 Sample3      bioc_type
## gene1     100     250    1187 protein_coding
## gene2      15      20       0         lncRNA
## gene3    4532     745      10 protein_coding
## gene4      20     100     596           rRNA
```

Aside: Two-dimensional Tables

... convert it to a **base R** table:

```
data.frame(my_list)
```

```
##       Sample1 Sample2 Sample3      bioc_type
## gene1     100     250    1187 protein_coding
## gene2      15      20       0         lncRNA
## gene3    4532     745      10 protein_coding
## gene4      20     100     596           rRNA
```

... convert it to a **tidyverse** "tibble":

```
my_list %>%
  as.data.frame() %>%
  tibble::rownames_to_column("gene_id")
```

```
##   gene_id Sample1 Sample2 Sample3      bioc_type
## 1   gene1     100     250    1187 protein_coding
## 2   gene2      15      20       0         lncRNA
## 3   gene3    4532     745      10 protein_coding
## 4   gene4      20     100     596           rRNA
```

Aside: Two-dimensional Tables

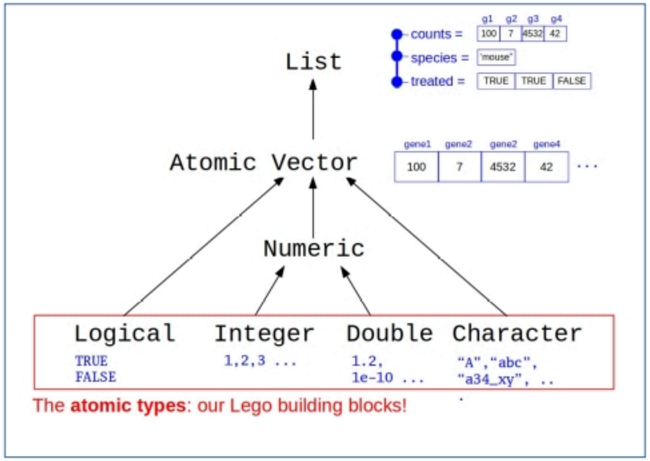
… convert **the numeric part of it** it to a **base R matrix**:

```
df <- data.frame(my_list)
m  <- as.matrix(df[,1:3]) ## select columns 1 to 3
m
```

```
##       Sample1 Sample2 Sample3
## gene1     100     250    1187
## gene2      15      20       0
## gene3    4532     745      10
## gene4      20     100     596
```

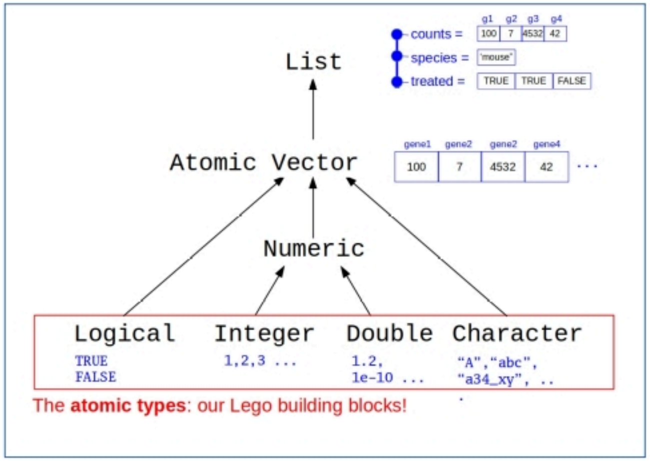
Data Structures

pre-defined data structures

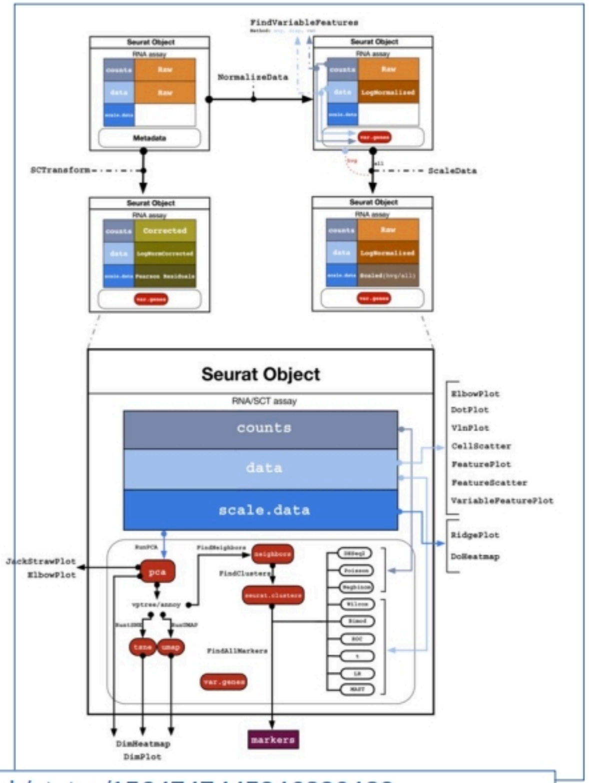


Data Structures

pre-defined data structures

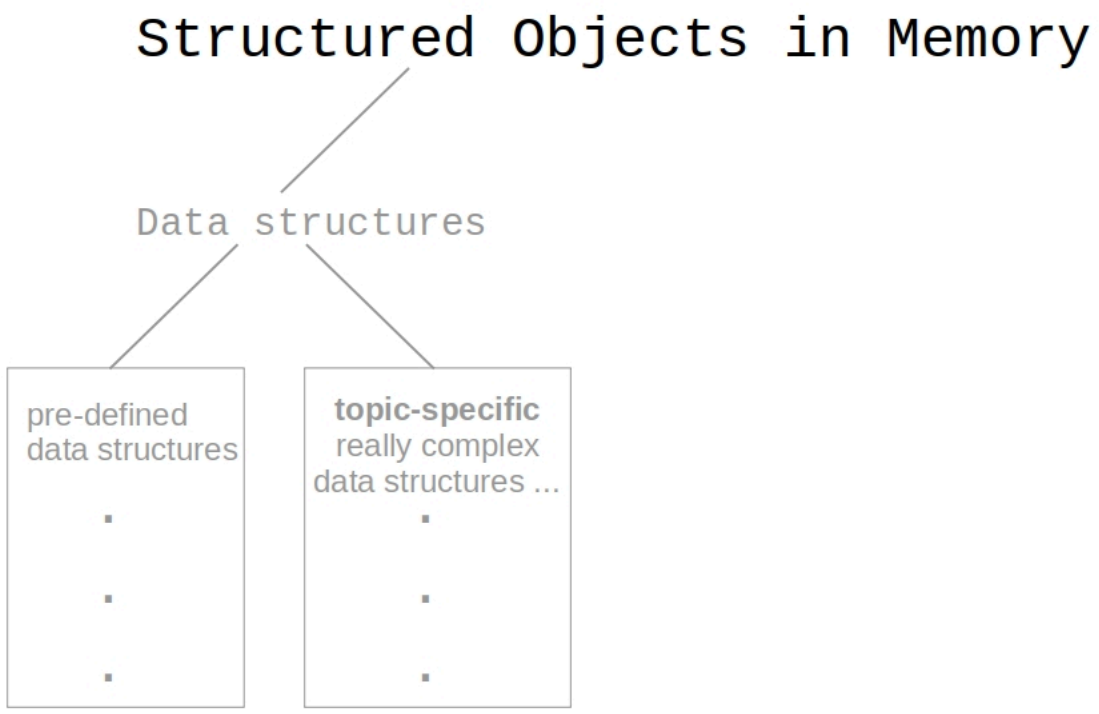


topic-specific really complex data structures ...

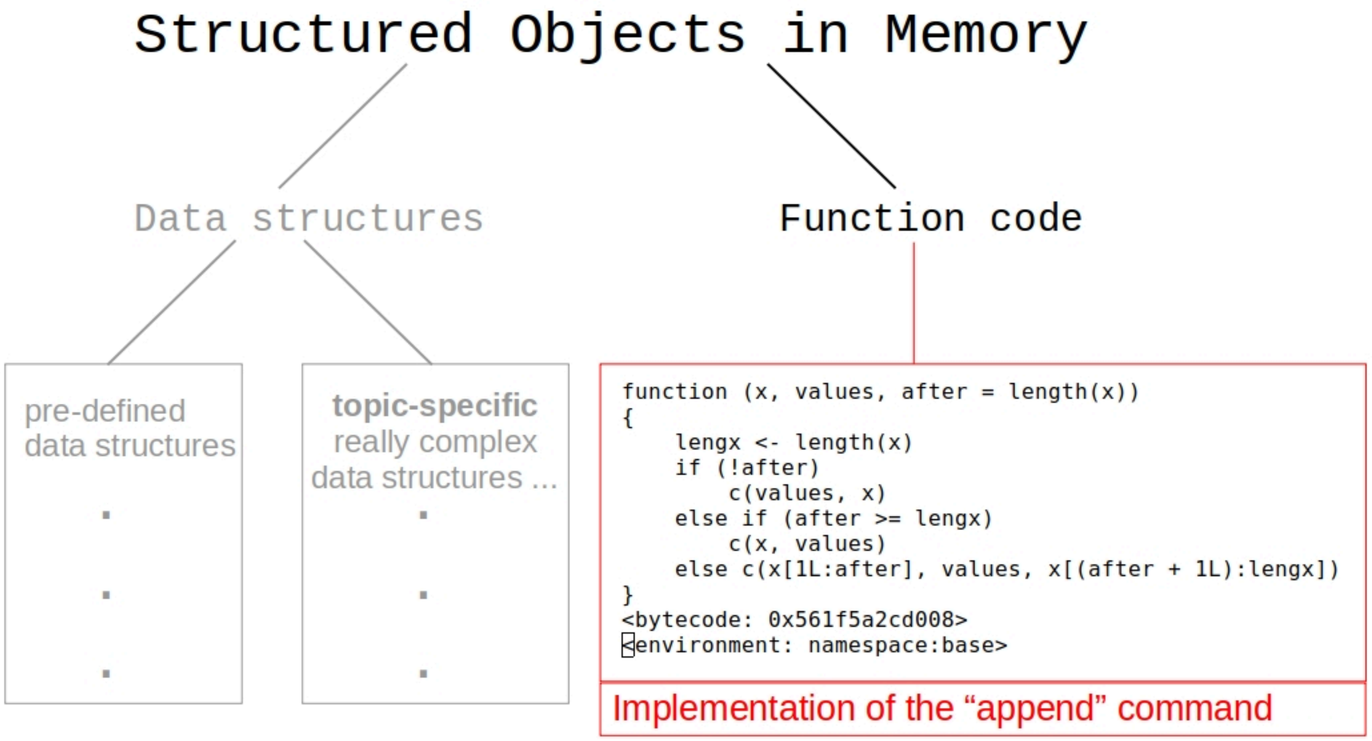


https://twitter.com/theHumanBorch/status/1524747445346836482

Data Structures -- Not the Only Objects!


Structured Objects in Memory
Data structures
pre-defined data structures
topic-specific really complex data structures ...
.
.
.

Data Structures -- Not the Only Objects!


Structured Objects in Memory
Data structures
Function code
pre-defined data structures
topic-specific really complex data structures ...
.
.
.
Implementation of the “append” command

```
function (x, values, after = length(x))
{
    lengx <- length(x)
    if (!after)
        c(values, x)
    else if (after >= lengx)
        c(x, values)
    else c(x[1L:after], values, x[(after + 1L):lengx])
}
<bytecode: 0x561f5a2cd008>
<environment: namespace:base>
```

Control Flow in Program Code

Control Flow in Program Code

-- conditional execution of code

Control Flow in Program Code

-- conditional execution of code

```r
weather <- "rainy"

## Execute (or not) a specific portion of the code,
## depending on the values of existing variables:
if(weather == "rainy") {
  bathing <- "brr..."
} else if (weather == "sunny") {
  bathing <- "great!"
} else {
  bathing <- "hm, don't know ..."
}

paste("Is bathing a good idea today?", bathing)
```

```
## [1] "Is bathing a good idea today? brr..."
```

Control Flow in Program Code

-- the "for" loop: iterate over a vector

```r
weather_options <- c("rainy",
                     "cloudy",
                     "sunny")

## Iterate over the elements of a vector,
## option 1: by element
 for( weather in weather_options) {
   if(weather == "rainy") {
     bathing <- "brr..."
   } else if (weather == "sunny") {
     bathing <- "great!"
   } else {
     bathing <- "hm, don't know ..."
   }

   ## NOTE that output within a loop must be
   ## explicitly printed to be visible!
   print(paste("Today the weather is:", weather))
   print(paste("Is bathing a good idea today?", bathing))
 }
```

Control Flow in Program Code

-- the "for" loop: iterate over a vector

```r
weather_options <- c("rainy",
                     "cloudy",
                     "sunny")

## Iterate over the elements of a vector,
## option 1: by element
for(weather in weather_options) {
  if(weather == "rainy") {
    bathing <- "brr..."
  } else if (weather == "sunny") {
    bathing <- "great!"
  } else {
    bathing <- "hm, don't know ..."
  }

  ## NOTE that output within a loop must be
  ## explicitly printed to be visible!
  print(paste("Today the weather is:", weather))
  print(paste("Is bathing a good idea today?", bathing))
}
```

```
## [1] "Today the weather is: rainy"
## [1] "Is bathing a good idea today? brr..."
## [1] "Today the weather is: cloudy"
## [1] "Is bathing a good idea today? hm, don't know ..."
## [1] "Today the weather is: sunny"
## [1] "Is bathing a good idea today? great!"
```

Control Flow in Program Code

-- the "for" loop: iterate over a vector

```
weather_options <- c("rainy",
                     "cloudy",
                     "sunny")

## Iterate over the elements of a vector,
## option 2: by vector position
for(i in seq_along(weather_options)) {
  if(weather_options[i] == "rainy") {
    bathing <- "brr..."
  } else if (weather_options[i] == "sunny") {
    bathing <- "great!"
  } else {
    bathing <- "hm, don't know ..."
  }

  ## NOTE that output within a loop must be
  ## explicitly printed to be visible!
  print(paste("Today the weather is:", weather_options[i]))
  print(paste("Is bathing a good idea today?", bathing))
}
```

Control Flow in Program Code

-- the "for" loop: iterate over a vector

```r
weather_options <- c("rainy",
                     "cloudy",
                     "sunny")

## Iterate over the elements of a vector,
## option 2: by vector position
for(i in seq_along(weather_options)) {
  if(weather_options[i] == "rainy") {
    bathing <- "brr..."
  } else if (weather_options[i] == "sunny") {
    bathing <- "great!"
  } else {
    bathing <- "hm, don't know ..."
  }

  ## NOTE that output within a loop must be
  ## explicitly printed to be visible!
  print(paste("Today the weather is:", weather_options[i]))
  print(paste("Is bathing a good idea today?", bathing))
}
```

Control Flow in Program Code

-- the "for" loop: iterate over a vector

```r
weather_options <- c("rainy",
                     "cloudy",
                     "sunny")

## Iterate over the elements of a vector,
## option 2: by vector position
for(i in seq_along(weather_options)) {
  if(weather_options[i] == "rainy") {
    bathing <- "brr..."
  } else if (weather_options[i] == "sunny") {
    bathing <- "great!"
  } else {
    bathing <- "hm, don't know ..."
  }

  ## NOTE that output within a loop must be
  ## explicitly printed to be visible!
  print(paste("Today the weather is:", weather_options[i]))
  print(paste("Is bathing a good idea today?", bathing))
}
```

```
## [1] "Today the weather is: rainy"
## [1] "Is bathing a good idea today? brr..."
## [1] "Today the weather is: cloudy"
## [1] "Is bathing a good idea today? hm, don't know ..."
## [1] "Today the weather is: sunny"
## [1] "Is bathing a good idea today? great!"
```

Control Flow in Program Code

-- the "while" loop: repeat until TRUE

```
 ## generate a new random number, until the number is > 900
num <- sample(1:1000, size=1)
while(num <= 900) {
   print(num) ## print old number
   num <- sample(1:1000, size=1) ## try again
}
```

Control Flow in Program Code

-- the "while" loop: repeat Until TRUE

```r
## generate a new random number, until the number is > 900
num <- sample(1:1000, size=1)
while(num <= 900) {
  print(num) ## print old number
  num <- sample(1:1000, size=1) ## try again
}
```

```
## [1] 204
## [1] 191
```

Functions

- **each R "command" is a function**
- a function is an object which contains R code
- the symbol bound to a function object is the function name
- accordingly, **the underlying object (the code) is printed when the name is typed** (however the code is not always available)

```
append
```

```
## function (x, values, after = length(x))
## {
##     lengx <- length(x)
##     if (!after)
##         c(values, x)
##     else if (after >= lengx)
##         c(x, values)
##     else c(x[1L:after], values, x[(after + 1L):lengx])
## }
## <bytecode: 0x55f1914fe198>
## <environment: namespace:base>
```

Functions

```r
c
```

```
## function (...)  .Primitive("c")
```

Functions

- **calling** a function executes the code

```
append(x = c("first","second","third"), ## the input vector
       values = c("NEW1","NEW2"), ## that which we want to insert
       after = 2 ## insert elements after original position 2
       )
```

- A **function call** consists of
  - The function name
  - followed by a comma-separated list of "a=b" pairs in parentheses,
  - where "a" is an internal symbol of the function (a **parameter**)
  - and "b" is a symbol or object in your workspace
- "a=b" assigns the value of "b" to the internal symbol "a" for the time of function execution

Functions

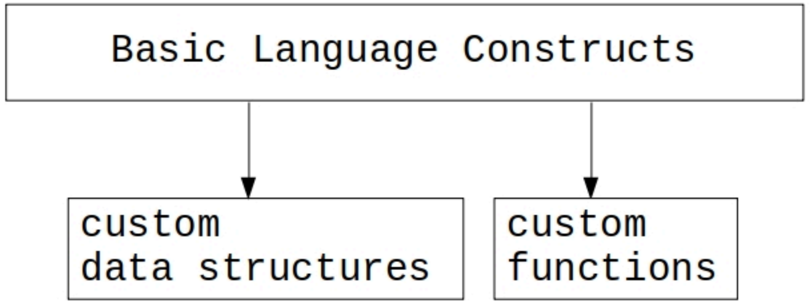
- typing **?** in the console followed by the function name opens a **help page**

Writing Your Own Function

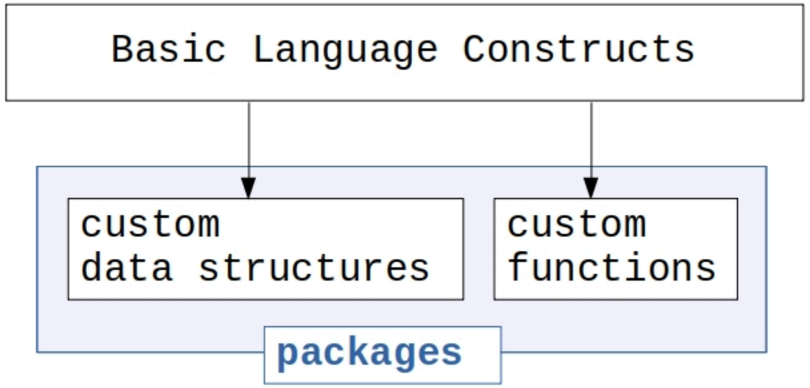
```
number_dart <- function(full_range=c(0,1000),
                        aimed_at_range=c(10,20)
                        ) {
  ## try to capture an integer number in sub-range "aimed_at_range"
  ## within range "full_range"

  ## Draw a number:
  this_number <- sample(full_range[1]:full_range[2],
                        size=1)

  ## Is it a hit?
  ## Note that a function returns the result of the last statement.
  if(this_number %in%  aimed_at_range[1]:aimed_at_range[2]) {
    ":-)"
  } else {
    ":-("
  }
}

## Aiming at the entire range is of course a cheat ...
number_dart(aimed_at_range=c(0,1000))
```

```
## [1] ":-)"
```

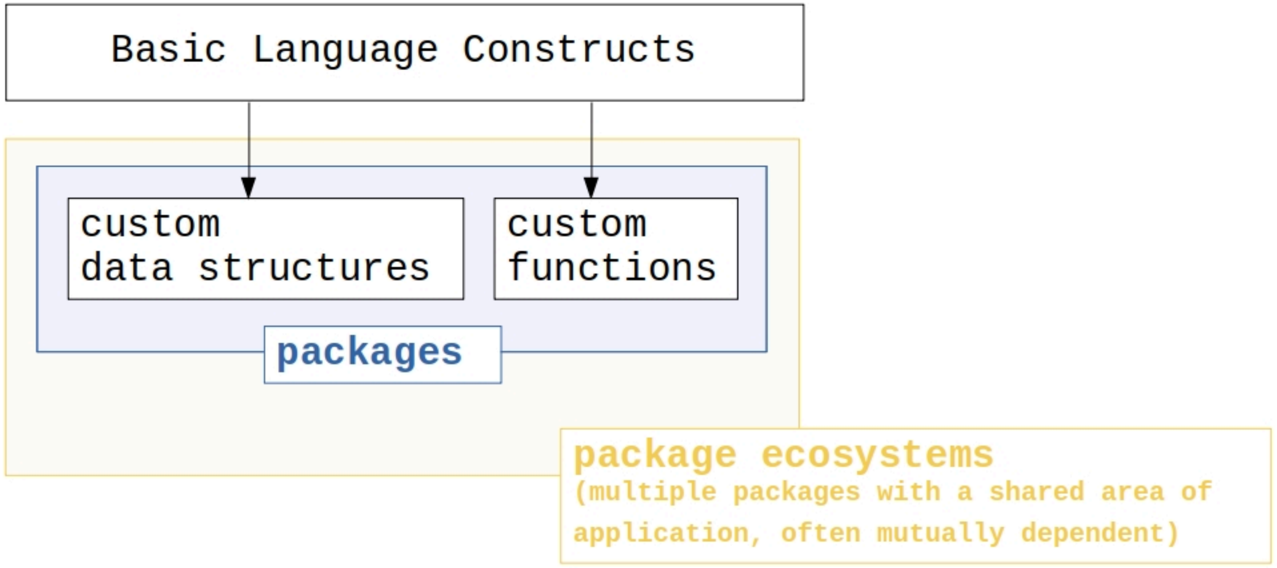
R Is Infinitely Extensible!



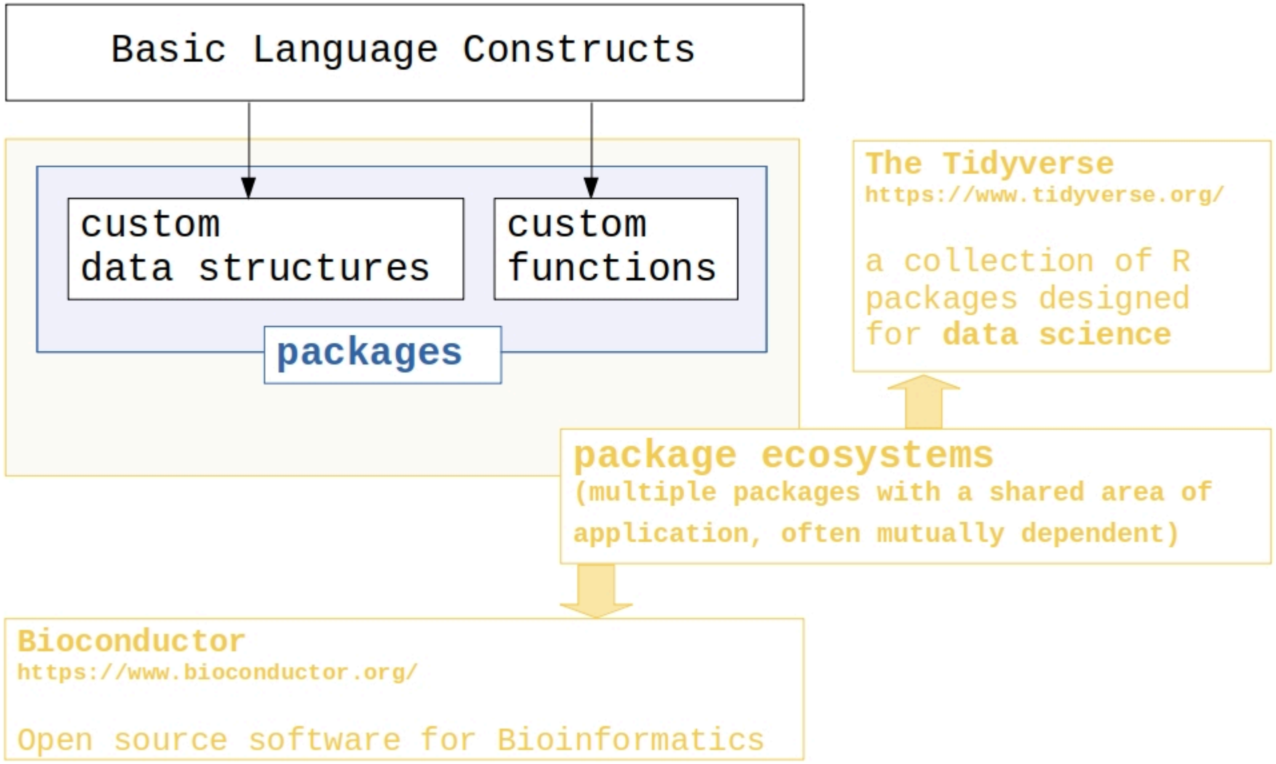
R Is Infinitely Extensible!



R Is Infinitely Extensible!


Basic Language Constructs
custom data structures
custom functions
packages
package ecosystems
(multiple packages with a shared area of application, often mutually dependent)

R Is Infinitely Extensible!


Basic Language Constructs
custom data structures
custom functions
packages
The Tidyverse
https://www.tidyverse.org/
a collection of R packages designed for data science
package ecosystems
(multiple packages with a shared area of application, often mutually dependent)
Bioconductor
https://www.bioconductor.org/
Open source software for Bioinformatics

Packages

- A package **bundles functions and possibly data which together implement a specific type of analysis** and usually depend on each other
- Example: The Seurat package and its associated packages SeuratObject and SeuratData (actually a small "package ecosystem")

Packages

- A package **bundles functions and possibly data which together implement a specific type of analysis** and usually depend on each other
- Example: The Seurat package and its associated packages SeuratObject and SeuratData (actually a small "package ecosystem")
- A package is **loaded with the `library()` function**, e.g. `library(Seurat)`. Once loaded, the exported symbols of a package (notably the function names) are visible in the user's workspace and can be used. To use a specific function of a package without loading it, prepend the function name with the package name, e.g. Seurat::DimPlot().

Packages

- A package **bundles functions and possibly data which together implement a specific type of analysis** and usually depend on each other
- Example: The Seurat package and its associated packages SeuratObject and SeuratData (actually a small "package ecosystem")
- A package is **loaded with the `library()` function**, e.g. `library(Seurat)`. Once loaded, the exported symbols of a package (notably the function names) are visible in the user's workspace and can be used. To use a specific function of a package without loading it, prepend the function name with the package name, e.g. Seurat::DimPlot().
- Packages can be found on and installed from **public repositories (https://cran.r-project.org/mirrors.html, https://www.bioconductor.org/install/, ...)** or from the **GitHub repositories of the package authors**, e.g. https://github.com/satijalab/seurat)

Packages

- A package **bundles functions and possibly data which together implement a specific type of analysis** and usually depend on each other
- Example: The Seurat package and its associated packages SeuratObject and SeuratData (actually a small "package ecosystem")
- A package is **loaded with the `library()` function**, e.g. `library(Seurat)`. Once loaded, the exported symbols of a package (notably the function names) are visible in the user's workspace and can be used. To use a specific function of a package without loading it, prepend the function name with the package name, e.g. Seurat::DimPlot().
- Packages can be found on and installed from **public repositories (https://cran.r-project.org/mirrors.html, https://www.bioconductor.org/install/, ...)** or from the **GitHub repositories of the package authors**, e.g. https://github.com/satijalab/seurat)
- See **rstudio->Tools->Install Packages** for instructions on installation and repository configuration. See also **`?.libPath`**.

Packages

- A package **bundles functions and possibly data which together implement a specific type of analysis** and usually depend on each other
- Example: The Seurat package and its associated packages SeuratObject and SeuratData (actually a small "package ecosystem")
- A package is **loaded with the `library()` function**, e.g. `library(Seurat)`. Once loaded, the exported symbols of a package (notably the function names) are visible in the user's workspace and can be used. To use a specific function of a package without loading it, prepend the function name with the package name, e.g. Seurat::DimPlot().
- Packages can be found on and installed from **public repositories (https://cran.r-project.org/mirrors.html, https://www.bioconductor.org/install/, ...)** or from the **GitHub repositories of the package authors**, e.g. https://github.com/satijalab/seurat)
- See **rstudio->Tools->Install Packages** for instructions on installation and repository configuration. See also **`?.libPath`**.
- Mind: Packages usually have regular **updates**, which may alter function behavior!