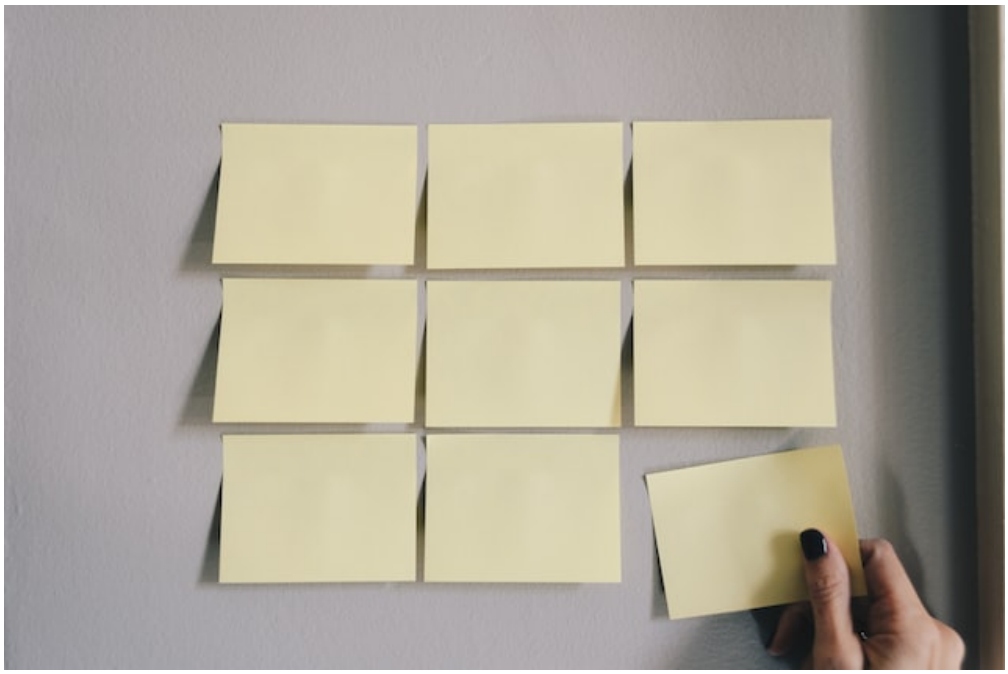
Lists in R

Introduction to data science (DSC 105) Fall 2022

What will you learn?



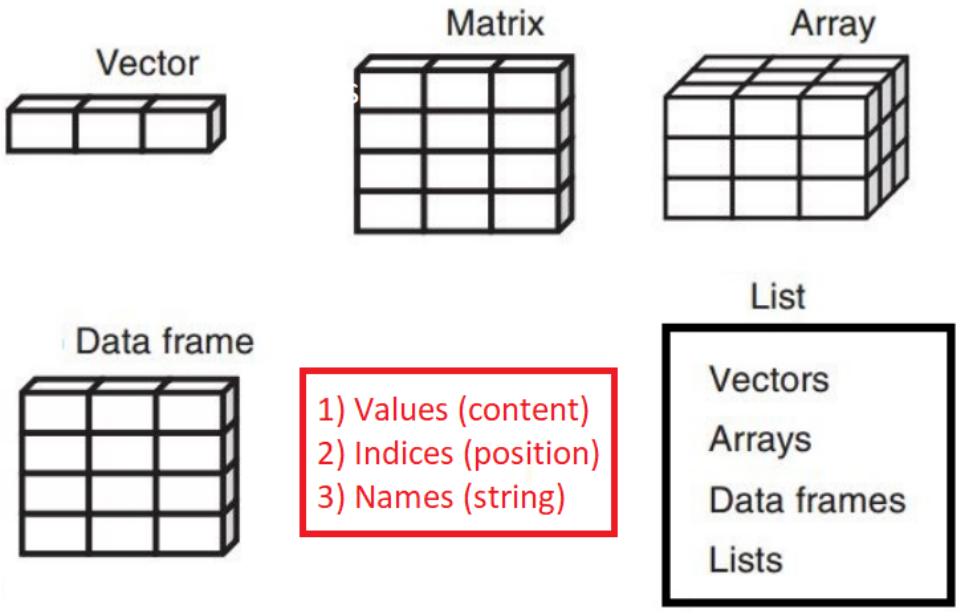
- Why are lists important?
- What are the list equivalents in other program languages?
- How do you create a (named) list?
- How can you subset lists?

Practice file

1. Open a new practice file `lists.org` or re-use an old practice file, e.g. by opening Emacs on the cmd line: `emacs --file lists.org`
2. Start your Emacs Org-mode file with these two **exact** lines:

```
#+STARTUP: overview hideblocks indent
#+PROPERTY: header-args:R :session *R* :results output
```

Why are lists important?



- Data structures we met so far:

1. Vectors (including scalars) like `c(1,100,-55.44)`

```
c(1,100,-55.44)     # a numeric vector
letters[1:3]        # a character vector
c(TRUE,FALSE,TRUE)  # a logical vector
```

```
[1]   1.00 100.00 -55.44
[1] "a" "b" "c"
[1]  TRUE FALSE  TRUE
```

2. Matrices like `matrix(1:9, nrow=3)`

```
matrix(1:9,nrow=3)
```

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

3. Data frames like `ToothGrowth`

```
str(ToothGrowth)  # two numeric, one factor vectors
```

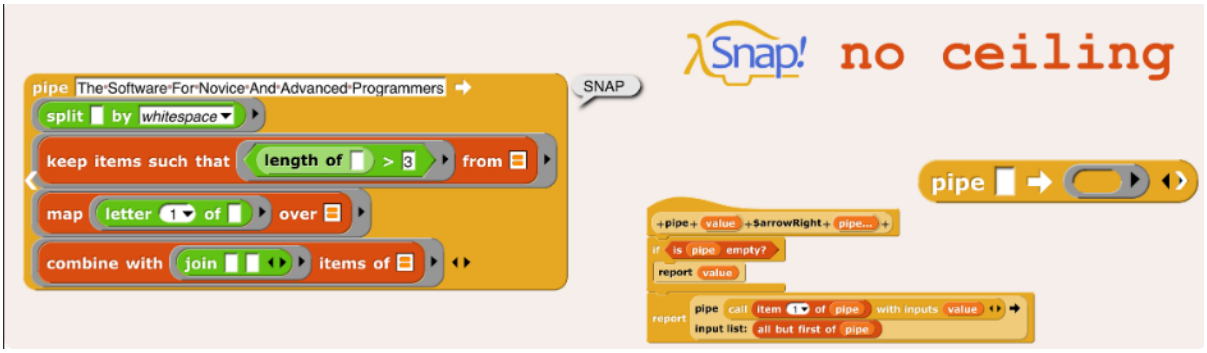
```
'data.frame':   60 obs. of  3 variables:
 $ len : num  4.2 11.5 7.3 5.8 6.4 10 11.2 11.2 5.2 7 ...
```

```
$ supp: Factor w/ 2 levels "OJ","VC": 2 2 2 2 2 2 2 2 2 2 ...
$ dose: num  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 ...
```

4. Lists

- Other R objects we've met:
  1. Operators like `$`, `[]` and `:`
  2. Functions like `c()`, `seq()`, `rep()`
  3. Functions like `hist()`, `subset()`
  4. Utility functions like `options()`, `par()`
  5. Variables like `foo`
  6. Constants like `LETTERS`
  7. Environments like `getwd()`
- In short:
  1. **Everything that exists in R is an object**
  2. **Everything that happens in R is a function call**
  3. **Interfaces to other software are a part of R**[1]
- You can group any mix of R structures and objects in a `list`

- You can even have a `list` as a component of another `list` - this separates languages suitable for data science from others. Scratch e.g. cannot do this, but Snap!, its advanced cousin, can



- Equivalents of lists in other programming languages: *recursive* structures, like *dictionary* in Python, *containers* in C++, or LIST in SQL

Creating (named) lists

- Creating a list is much like creating a vector with the `c` function
- You supply the elements to the `list` function, separated by commata

- Let's define a list `foo` and print it:

```
foo <- list(
  matrix(data = 1:4,
         nrow = 2,
         ncol = 2),
  c(TRUE,FALSE,TRUE,TRUE),
  "hello")
foo
```

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

[[2]]
[1]  TRUE FALSE  TRUE  TRUE

[[3]]
[1] "hello"
```

- List elements are indexed with the `[[` operator. Within each element, the elements are indexed according to their data structure
- What is the `length` of the list `foo`?

```
length(x = foo)
```

```
[1] 3
```

- What is the `class` of the list `foo` and of its elements?

```
class(foo)
class(foo[[1]])
class(foo[[2]])
class(foo[[3]])
```

```
[1] "list"
[1] "matrix" "array"
[1] "logical"
[1] "character"
```

Practice creating a `list`

Create a list that contains, in this order

1. a `seq`-uence of 20 evenly spread numbers between `-4` and `4`
2. a 3 x 3 `matrix` of the `logical` vector `c(F,T,T,T,F,T,T,F,F)` filled column-wise
3. a `character` vector with the two strings `"don"` and `"quixote"`

```
seq <- seq(from=-4, to=4, length.out=20)  #1
mat <- matrix(c(F,T,T,T,F,T,T,F,F),nrow=3,byrow=FALSE) #2
chr <- c("don","quixote")  #3
p <- list(seq, mat, chr)
p
```

```
[[1]]
 [1] -4.0000000 -3.5789474 -3.1578947 -2.7368421 -2.3157895 -1.8947368
 [7] -1.4736842 -1.0526316 -0.6315789 -0.2105263  0.2105263  0.6315789
[13]  1.0526316  1.4736842  1.8947368  2.3157895  2.7368421  3.1578947
```

```
[19]  3.5789474  4.0000000

[[2]]
      [,1]  [,2]  [,3]
[1,] FALSE  TRUE  TRUE
[2,]  TRUE FALSE FALSE
[3,]  TRUE  TRUE FALSE

[[3]]
[1] "don"     "quixote"
```

Subsetting lists

- Retrieve list members using indices in **double** square brackets
- Retrieve (or remove) list member elements using **single** square brackets

- Some examples:

    1. Retrieve the 1st list member of `foo` (def'd here)
    2. Retrieve the 3rd element of the 1st member of `foo`
    3. Retrieve the 2nd through 4th element of the 2nd member
    4. Retrieve the 3rd member
    5. What is the 2nd element of the 3rd member?

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

```
     [,1] [,2]
[1,]    1    3
[2,]    2    4
[1] 3
[1] FALSE  TRUE  TRUE
[1] "hello"
[1] NA
```

- Same for the matrix member of the list: use double indices

    1. retrieve the 2nd column of the 1st member of `foo`
    2. retrieve the 2nd row, 1st column of the 1st member of `foo`
    3. retrieve elements 1 through 4 of the 1st member of `foo`

```
foo[[1]]
foo[[1]][,2]   #1
foo[[1]][2,1]  #2
foo[[1]][1:4]  #3  this is.vector and not matrix!
foo[[1]][2]    # this works, too - remember byrow=FALSE
```

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

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

- Using double square brackets on a list is always interpreted with respect to a single member, for example:

```
foo[[c(2,3)]]
foo[[2]][3]
```

```
[1] TRUE
[1] TRUE
```

- Using the - operator inside the single index brackets to remove:

```
foo[[2]]
foo[[2]][-1]
```

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

- Preview: how would you extract the string member of `foo`?

```
char <- lapply(X=foo,FUN=is.character) # test each member
idx <- which(char==TRUE) # get the index
foo[[idx]]  # index list
```

```
[1] "hello"
```

- The `apply` family of functions will be taught in advanced data science, including `lapply` (apply `FUN`-ctions across a whole `list`)

Practice extracting from a `list`

Solve the following extraction problems:

1. Extract the 2nd member of `p` (def'd here)
2. Extract the 2nd column of the 2nd member of `p`
3. Extract the first and the last element of the 1st member of `p`

```
p[[2]]      #1
p[[2]][,2]  #2
p[[1]][c(1,length(p[[1]]))] #3
```

```
      [,1]  [,2]  [,3]
[1,] FALSE  TRUE  TRUE
[2,]  TRUE FALSE FALSE
[3,]  TRUE  TRUE FALSE
```

```
[1]  TRUE FALSE  TRUE
[1] -4  4
```

Removing, overwriting and slicing a `list`

- To overwrite a list member, use the assignment operator `<-`

```
foo[[3]]
bar <- foo # safety copy
bar[[3]] <- paste(foo[[3]], "world!")
bar[[3]]
```

```
[1] "hello"
[1] "hello world!"
```

- Here, `paste` concatenates strings but can also be used for output:

```
a <- "10,000"
paste("a is", a)

x <- 10000
paste("x is", x)
```

```
[1] "a is 10,000"
[1] "x is 10000"
```

- To remove a list member, overwrite it with `NULL` (like `names`)

```
baz <- foo  # safety copy
baz[[1]] <- NULL
baz
```

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

[[2]]
[1] "hello"
```

- *List slicing* means selecting multiple list items at once:

```
foo[c(2,3)] # select list members 2 and 3
```

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

[[2]]
[1] "hello"
```

- Note that the sliced list is itself a `list`

Naming lists

- List members can be *named* just like vector or data frame elements
- A name is an R *attribute*. An unnamed list has none:

```
attributes(foo)
```

```
NULL
```

- Name the members of `foo` using `names`, then print `str(foo)`:

```
names(foo) <- c(
  "mymatrix",
  "mylogicals",
  "mystring")
str(foo)
```

```
List of 3
 $ mymatrix  : int [1:2, 1:2] 1 2 3 4
 $ mylogicals: logi [1:4] TRUE FALSE TRUE TRUE
 $ mystring  : chr "hello"
```

- You can now use the names to subset the list as usual:

  1. Print the `matrix` member of `foo` (def'd here)
  2. Print the 2nd column of the `matrix` member
  3. Print the 2nd through 4th element of the `logical` member

```
foo$mymatrix      #1
foo$mymatrix[,2]  #2
foo$mylogicals[2:4]
```

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

- Note that the `names` are stored as a `character` vector but not used with quotes. Also, you cannot use the names inside double brackets

```
vec <- c("a"=1,"b"=2)   # vector with two named elements
names(vec)              # names of vector elements
vec["a"]                # extracting element with name
vec[1]                  # extracting element with index
vec[c("a","b")]         # extracting elements with names vector
```

```
vec[c(1,2)]       # extracting elements with index vector
n <- names(vec)   # storing names vector
vec[n]            # extracting elements with names vector
```

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

- You can also name the list when creating it with `list`:

```
q <- list(
  "my name"="Adam",
  "my sons"= c("Kain", "Abel"))
str(q)
```

```
List of 2
 $ my name: chr "Adam"
 $ my sons: chr [1:2] "Kain" "Abel"
```

TODO Practice naming lists

1. Make a safety copy `np` of `p`
2. Name the elements of `np` in this order: `num`, `logmat`, `char`
3. Display the structure of the named list `p`
4. Remove the 2nd string of the 3rd member using its name

```
np <- p                                #1
names(np) <- c("num","logmat","char") #2
str(np)                                #3
np$char[-2]
```

```
List of 3
 $ num   : num [1:20] -4 -3.58 -3.16 -2.74 -2.32 ...
 $ logmat: logi [1:3, 1:3] FALSE TRUE TRUE TRUE FALSE TRUE ...
 $ char  : chr [1:2] "don" "quixote"
[1] "don"
```

TODO Nesting lists

TODO Concept summary

- Lists of lists

TODO Code summary

TERM	MEANING

Footnotes:

[1] This last tenet is the basis of extending R beyond its own data structures, e.g. in the direction of C++ (Rcpp), or databases (RSQLite). This is subject of a planned DSC 482 special topics (fall '23) on software development methods, "Extending R" (title taken from a book by one of the creators of R, John Chambers).

Author: Marcus Birkenkrahe
Created: 2022-11-16 Wed 13:07