Intro to Data Visualization and Statistics in R

Common data types in R

Genome Institute of Singapore

13th November 2019

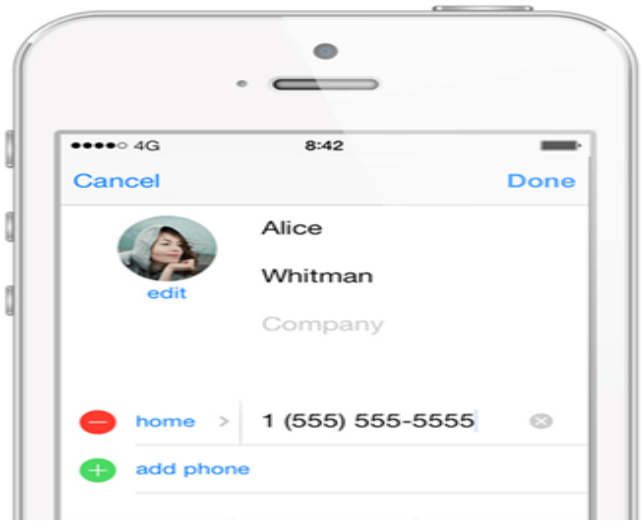
The need for different data types

How do you call someone? Do you

1. Dial someones phone number in full from memory into the dialpad?

or

2. Search the "Contacts" and then click it? If yes, you need to first assign/save the number into a Contact detail.



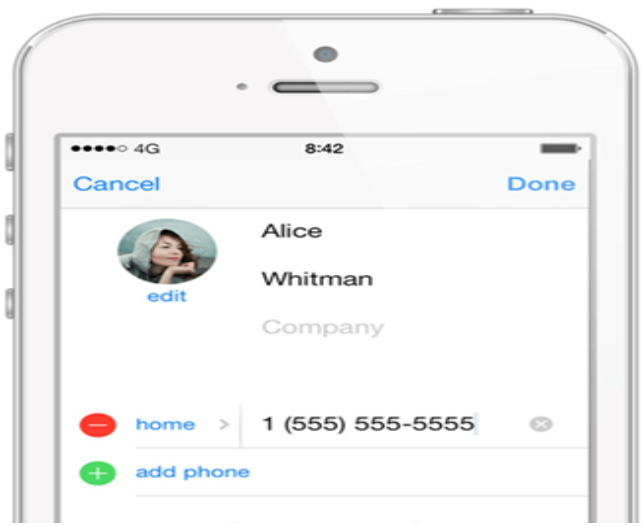
The need for different data types

How do you call someone? Do you

1. Dial someones phone number in full from memory into the dialpad?

or

2. Search the "Contacts" and then click it? If yes, you need to first assign/save the number into a Contact detail.



Similarly in R, you can assign data into a data type, depending on the structure and complexity, for later use.

Data types | Overview

Here are the generic data types in R.

- Scalar
- Vector
- Factor
- Matrix
- **Data Frames***
- Lists

* Data frames is one of the most common data types that we will encounter. The tidyverse grammar (select, rename, filter, pull, mutate, group_by, summarise, arrange) allows us to manipulate data frames in an elegant way.

In these slides, we will cover the data types listed above. We will use base R commands (tidyverse is only for data frames).

Data types | Overview

Every data type have the following features:

- way to create it
- special operations
- attributes
- way to extract elements from it

Use `class(object)` to determine the data type of the `object`.

SCALARS

Scalar | Definition and creation

Assign a value with a name so that it can be re-used it later.

```
x <- 100
(x + 1)^2 / x
## [1] 102.01
```

The assignment operator is "<-".

Scalar | Definition and creation

You can also assign a character using quotes:

```
x <- "100"
```

(Note: re-assigning to `x` destroys the previous value)

In R, you cannot apply a mathematical function on a character.

```
x + 1
# Error in y + 1 : non-numeric argument to binary operator
```

i.e. Numeric and character value are not identical.

VECTORS

Vectors | Definition

A vector is a collection of numeric values or character values (never a mix of both).

Three types:

- numeric vectors
- character vectors
- logical vectors

Optionally, a vector can have names.

Numeric vector | Creation

Let us generate a numeric vector:

```
x <- c(1, 2, 3, 4, 5)
x
## [1] 1 2 3 4 5
```

The small letter "c" is a function to ***c***ombine values into a vector.

Numeric vector | Creation

Generating a sequence of numbers is a fairly common task. You can use the colon operator (`:`) or `seq()` for this:

```
1:5                       ## increments of 1 only
## [1] 1 2 3 4 5

seq(1, 5, by=1)           ## same as 1:5
## [1] 1 2 3 4 5

seq(-100, 100, by=25) ## more generalized form
## [1] -100  -75  -50  -25    0   25   50   75  100
```

You can check if the two vectors are the same using:

```
x <- c(1, 2, 3, 4, 5)
y <- 1:5
x == y                    ## Check element by element
## [1] TRUE TRUE TRUE TRUE TRUE
```

Numeric vector | Operations

Many functions in R are vectorized. This means the function is applied to every element of a vector without an explicit `for()` loop.

```
x + 100
## [1] 101 102 103 104 105

y^2
## [1]  1  4  9 16 25

x + 100 + y^2          ## Adds element by element
## [1] 102 106 112 120 130

x < 3
## [1]  TRUE  TRUE FALSE FALSE FALSE
```

Vectorized operations are much more compact to code and usually computationally more efficient.

Numeric vector | Re-arranging

```
r <- c(3.3, 1.1, 4.4, 5.5, 2.2)

sort(r)                    ## Sort smallest to largest
## [1] 1.1 2.2 3.3 4.4 5.5

rank(r)                    ## Smallest value is ranked 1
## [1] 3 1 4 5 2

sort(r, decreasing=T) ## Sort largest to smallest
## [1] 5.5 4.4 3.3 2.2 1.1

rank(-r)                   ## Largest value is ranked 1
## [1] 3 5 2 1 4

rev(r)                     ## Reverse (N, N-1, ..., 2, 1)
## [1] 2.2 5.5 4.4 1.1 3.3
```

Numeric vector | Summary

You can summarize a numeric vector in different ways. E.g.

```
x <- seq(0, 500, by=100)
x
## [1]   0 100 200 300 400 500

length(x)
## [1] 6

mean(x)
## [1] 250

var(x)
## [1] 35000

summary(x)  ## Demo only. Not meaningful with few values.
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##       0     125     250     250     375     500
```

Character vector | Creation

You can create a character vector by quoting the values:

```
y <- c("G", "C", "T", "A")
y
## [1] "G" "C" "T" "A"
```

Arithmetic operations (sum, mean, var, summary) will not work. However, the sort and rank will work alphabetically.

```
length(y)
## [1] 4

sort(y)
## [1] "A" "C" "G" "T"

rank(y)
## [1] 3 2 4 1
```

Character vector | Comparison with a numeric vector

Quoting a number makes them characters which renders them not viable for arithmetic operations:

```
y <- c("1", "2", "3", "4", "5")
y
## [1] "1" "2" "3" "4" "5"
```

You can convert this back to a numeric vector

```
y <- as.numeric(y)
y + 100
## [1] 101 102 103 104 105
```

Character vector | Always character or always numeric

Creating a vector with a mix of numeric and character will turn it to a character vector instead.

```
y <- c(1, 2, 3, "D", "E")
y
## [1] "1" "2" "3" "D" "E"
```

Trying to convert this to a numeric vector will only convert the quoted numbers. The quoted characters will be turned into missing values (encoded as `NA` in R).

```
as.numeric(y)
## Warning: NAs introduced by coercion
## [1]  1  2  3 NA NA
```

Logical vectors | Definition and creation

This is a vector of TRUE and FALSE. The `which()` function pulls the **index** (or position) of the TRUE elements.

```
y <- c(TRUE, FALSE, TRUE, FALSE, TRUE)
y
## [1]  TRUE FALSE  TRUE FALSE  TRUE
which(y)
## [1] 1 3 5
```

Logical vectors can be created when you make a comparison.

```
x <- seq(-50, 50, by=25)
x
## [1] -50 -25   0  25  50
neg <- ( x < 0 )
neg
## [1]  TRUE  TRUE FALSE FALSE FALSE
which(neg)
## [1] 1 2
```

Vector | One element vector reduces to scalar

A vector with only one element reduces to a scalar

```
c(1)
## [1] 1

c("G")
## [1] "G"

c(TRUE)
## [1] TRUE
```

This concept of reduction is useful to know.

Later, you will see that pulling a column from matrix or dataframe will reduce it to a vector.

Vector | Naming

Let's create a vector to hold the number of days in a non-leap year.

```
nDays <- c(31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)
nDays
##  [1] 31 28 31 30 31 30 31 31 30 31 30 31
```

It is difficult to see which month has how many days in this numeric vector. Let's set the `names()` attribute.

Vector | Naming

Let's create a vector to hold the number of days in a non-leap year.

```
nDays <- c(31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)
nDays
##  [1] 31 28 31 30 31 30 31 31 30 31 30 31
```

It is difficult to see which month has how many days in this numeric vector. Let's set the `names()` attribute.

```
mths <- c("Jan", "Feb", "Mar", "Apr", "May", "Jun",
          "Jul", "Aug", "Sep", "Oct", "Nov", "Dec")
names(nDays) <- mths
nDays
## Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
##  31  28  31  30  31  30  31  31  30  31  30  31
```

Using names makes it easier to read and extract specific elements using their name (coming up in the next few slides).

Vector | Naming and creating

You can choose to create and name a vector simultaneously:

```
nDays <- c(Jan=31, Feb=28, Mar=31, Apr=30, May=31, Jun=30,
           Jul=31, Aug=31, Sep=30, Oct=31, Nov=30, Dec=31)

nDays
## Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
##  31  28  31  30  31  30  31  31  30  31  30  31
```

The quote around the month names is not necessary when creating named vectors unless it contains special characters (e.g. "Jan-2019", "Jan 2019"). Also, you can extract the names via:

```
names(nDays)
## [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun"
## [7] "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
```

Vector | Extraction

Several ways of extracting element(s) from a vector using a square bracket and one of the following:

1) position index. e.g. x[1]

2) position index but with a negative sign to drop elements. e.g. x[-1]

3) name if the vector is named. e.g. nDays[“Jan”]

You cannot combine a negative with a name. i.e. nDays[-“Jan”] will give an error.

Vector | Extraction of a single element

```
nDays
## Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
##  31  28  31  30  31  30  31  31  30  31  30  31

nDays[ 1 ]   ## Select only the first element
## Jan
##  31
nDays[ -1 ]  ## Select everything but the first element
## Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
##  28  31  30  31  30  31  31  30  31  30  31

nDays[ "Jan" ]
## Jan
##  31

keep <- "Jan";  nDays[ keep ]  ## using variable
## Jan
##  31
```

Vector | Extraction of multiple elements

You can use a vector of positions to extract specific elements:

```
x <- c("e", "h", "l", "o", "w", "r", "d")
x[ c(4, 1) ]  # Extract the 4th and 1st element
## [1] "o" "e"

y <- c(2, 1, 3, 3, 4)
x[y]          # Extract using another vector
## [1] "h" "e" "l" "l" "o"

paste(x[y], collapse="") # collapse to a scalar string
## [1] "hello"
```

Note:

1) The elements are returned in order requested
2) You can request the same index several times (e.g. here we requested the 3rd element twice to get “ll”)

Vector | Extraction of multiple elements

You can use the negative sign to drop some elements from the vector:

```
x
## [1] "e" "h" "l" "o" "w" "r" "d"

y
## [1] 2 1 3 3 4

x[-y]
## [1] "w" "r" "d"
```

Vector | Extraction using logic operators

```
nDays
# Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
#  31  28  31  30  31  30  31  31  30  31  30  31

nDays==30
##   Jan   Feb   Mar   Apr   May  Jun
## FALSE FALSE FALSE  TRUE FALSE TRUE
##   Jul   Aug   Sep   Oct   Nov   Dec
## FALSE FALSE  TRUE FALSE  TRUE FALSE

which(nDays==30)              # position of matches
## Apr Jun Sep Nov
##   4   6   9  11

names(which(nDays==30))     # get the names instead
## [1] "Apr" "Jun" "Sep" "Nov"

## or longer: names(nDays)[ which(nDays==30) ]
```

Vector | Extraction elements that do not exist

You get missing value (NA) if you request something non-existent.

```
nDays
## Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
##  31  28  31  30  31  30  31  31  30  31  30  31

nDays[ "June" ]    # Should be Jun not June
## <NA>
##   NA

nDays[ 13 ]        # ghost month?
## <NA>
##   NA

nDays[ c("May", "June", "July") ]  # May is found
##  May <NA> <NA>
##   31   NA   NA
```

Vector | Aside: set operators

Set operators can be useful when finding elements that are common/unique to each set. Example:

```r
left  <- c("A", "A", "B", "C")
right <- c("B", "C", "C", "C", "E", "D")

setdiff(left, right)    # Repeated items in a set ignored
## [1] "A"

setdiff(right, left)
## [1] "E" "D"

intersect(left, right) # same as intersect(right, left)
## [1] "B" "C"

union(left, right)      # same as union(right, left)
## [1] "A" "B" "C" "E" "D"
```

FACTOR

Factor | Definition

A factor is simply a vector where the value of the elements are limited and have an inherent order (set by the levels argument).

There are two main use cases for factors:

- order the output of a table or graph (e.g. axis order, facet)
- in a linear model, the effect of a categorical variable is reported with respect to a reference value

Factor | Motivation

Here are the WHO classification of BMI for 5 people:

```
BMI_class <- c("over", "normal", "obese", "over", "over")
```

Table gives the counts but in alphabetical order which is messy.

```
table(BMI_class)
## BMI_class
## normal  obese   over
##      1      1      3
```

Factor | Creation

Let's convert it to a factor and add in an extra level for future use.

```
levs <- c("under", "normal", "over", "obese")

BMI_class <- factor(BMI_class, levels=levs)

table(BMI_class)     ## Columns are ordered correctly
## BMI_class
##  under normal   over  obese
##      0      1      3      1

levels(BMI_class)   ## Levels of the factor
## [1] "under"  "normal" "over"   "obese"

nlevels(BMI_class)  ## Number of levels
## [1] 4
```

MATRIX

Matrix | Definition

A matrix is a collection of data elements

- arranged in a two-dimensional rectangular layout with rows and columns.
- each column is a vector.
- all vectors will be either numeric or character vector (never a mix of both).

Optionally, a matrix can have row names and/or column names.

Matrix | Creation

When you import data into R (e.g. read_csv, read_excel), it will be typically stored in a data frame. Then you can convert a dataframe to a matrix with `data.matrix()`.

```
iris_num <- iris %>% select(-Species) %>% data.matrix()
iris_num %>% head(2)
##      Sepal.Length Sepal.Width Petal.Length Petal.Width
## [1,]          5.1         3.5          1.4         0.2
## [2,]          4.9         3.0          1.4         0.2
```

You can also explicitly create a matrix manually from a vector. E.g.

```
m <- matrix(1:8, nrow=2)
m
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5    7
## [2,]    2    4    6    8
```

The data is filled columnwise by default unless you set `byrow=T`.

Matrix | Operations

Row and column names are optional but can be useful. The names are inherited from the dataframe if they are present. You can manually set the names or overwrite existing names via:

```
rownames(m) <- paste0("row", 1:nrow(m))
colnames(m) <- paste0("col", 1:ncol(m))
m
##      col1 col2 col3 col4
## row1    1    3    5    7
## row2    2    4    6    8
```

`nrow()` and `ncol()` gives the number of rows and columns.

```
dim(m)       ## number of rows by number of columns
## [1] 2 4
rownames(m)  ## extract rownames
## [1] "row1" "row2"
colnames(m)  ## extract colnames
## [1] "col1" "col2" "col3" "col4"
```

Matrix | Operations

Just like a vector, arithmetic operations work on every element

```
1 + m/100
##      col1 col2 col3 col4
## row1 1.01 1.03 1.05 1.07
## row2 1.02 1.04 1.06 1.08
```

Special functions exist to calculate column/row sums and means:

```
colSums(iris_num)
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##        876.5        458.6        563.7        179.9
colMeans(iris_num)
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##     5.843333     3.057333     3.758000     1.199333
rowSums(m)
## row1 row2
##   16   20
rowMeans(m)
```

Matrix | Operations

Sometimes it is useful to combine two different matrix together. We will use the same object here for example:

```
cbind(m, m)    ## Column binding
##      col1 col2 col3 col4 col1 col2 col3 col4
## row1    1    3    5    7    1    3    5    7
## row2    2    4    6    8    2    4    6    8

rbind(m, m)    ## Row binding
##      col1 col2 col3 col4
## row1    1    3    5    7
## row2    2    4    6    8
## row1    1    3    5    7
## row2    2    4    6    8
```

You need to be careful or the order of the objects with `rbind()` and column names of output with `cbind()`. We will later cover `bind_rows()` and `joins` which we feel are safer alternatives.

Matrix | Operations

Another useful function is to transpose the rows and columns

```
t(m)
##      row1 row2
## col1    1    2
## col2    3    4
## col3    5    6
## col4    7    8
```

This can be useful in at least two ways:

1) When you are using functions that act on rows. E.g. the principal component command assumes samples are in rows. Therefore you need to transpose the expression data.
2) When you are combining two objects. E.g. samples are represented by columns in the expression data and by rows in the clinical data. So you need to transpose the expression data before joining with the clinical data.

Matrix | Extraction

The rules for extracting elements from a vector also applies to a matrix. However, you need to specify the row **and** column indexes (i.e. `mat[ row_indexs, column_indexes ]`).

```
m[ 1:2, c(4,2) ]  ## First two rows. Columns 4 & 2.
##      col4 col2
## row1    7    3
## row2    8    4
m[    , c(4,2) ]  ## Can omit the row index for all rows
##      col4 col2
## row1    7    3
## row2    8    4

iris_num[ c(1, 51, 101), c("Sepal.Length", "Sepal.Width")]
##      Sepal.Length Sepal.Width
## [1,]          5.1         3.5
## [2,]          7.0         3.2
## [3,]          6.3         3.3
```

Matrix | Extraction of single column/row

The output will be coerced into a vector if you extract only one row or only one column.

```
m[  , 2 ]
## row1 row2
##    3    4
m[ 2,   ]
## col1 col2 col3 col4
##    2    4    6    8
```

You can avoid this coercion with `drop=FALSE`. Output is a matrix.

```
m[  , 2, drop=FALSE ]
##      col2
## row1    3
## row2    4
m[ 2,  , drop=FALSE ]
##      col1 col2 col3 col4
## row2    2    4    6    8
```

DATA FRAMES

Data frames | Definition

A data frame is a more generalized version of matrix.

It has the same two-dimensional shape as a matrix, but it can hold numeric columns along with character or factor columns.

Data frames | Creation

Most data import functions will store the data in a data frame.

You can also explicitly create a data.frame from a vector. E.g.

```
vowels <- c("A", "E", "I", "O", "U")
pos    <- which(LETTERS %in% vowels)

df <- data.frame(letter=LETTERS[pos], position=pos)
df
##   letter position
## 1      A        1
## 2      E        5
## 3      I        9
## 4      O       15
## 5      U       21
```

Data frames | Creation

The default behaviour when creating a data frame in this manner is to convert strings to factors.

```
df[ , "letter"]
## [1] A E I O U
## Levels: A E I O U
```

You can override this behaviour with `stringsAsFactors=FALSE`.

```
df <- data.frame(letter=LETTERS[pos], position=pos,
                 stringsAsFactors=FALSE)
df[ , "letter"]
## [1] "A" "E" "I" "O" "U"
```

If you wish to use non-standard names (e.g. Jan-2019, 2019-01), you can quote them with `check.names=FALSE` in `data.frame()`. Also, can use `check.names=FALSE` in `read.csv()`.

Data frames | Extraction

The extraction methods for a matrix also works for a data frame.

```
iris[ c(1, 51, 101), c("Sepal.Length", "Species")]
##     Sepal.Length    Species
## 1            5.1     setosa
## 51           7.0 versicolor
## 101          6.3  virginica
```

However, we recommend using tidyverse grammar to extract or manipulate data frames wherever possible for readability. For example, the code above can be rewritten as:

```
iris %>%
  slice(1, 51, 101) %>%
  select( Sepal.Length, Species )
##   Sepal.Length    Species
## 1          5.1     setosa
## 2          7.0 versicolor
## 3          6.3  virginica
```

Data frames | Extraction

You can also use "$" to pull, create and delete a column.

```
df$letter              ## pull "letter" column as a vector
## [1] "A" "E" "I" "O" "U"

df$junk <- rnorm(5)  ## Add a column with 5 random numbers
df
##   letter position       junk
## 1      A        1  0.8379214
## 2      E        5  1.6718361
## 3      I        9 -0.3434184
## 4      O       15 -0.1905589
## 5      U       21 -0.4483125

df$junk <- NULL      ## Delete the "junk" column
df
##   letter position
## 1      A        1
## 2      E        5
```

LIST

List | Definition

A list contain elements of other data types.

Think of list as your handbag which can contain all manners of wonderful things in all shapes and sizes. Even list of lists!

List | Creation

Here is how to create one:

```
experiment <- list(

  Requester = "Oguz, Gokce",

  Library_Kit = "Illumina TruSeq",

  Read = c(Length="150", type="paired-end"),

  out = data.frame(ID      = c("MUX1", "MUX2", "MUX3"),
                   Machine = c("HS08", "HS08", "HS06"),
                   Date    = c("1Jun", "2Jun", "1Jun"))
)
```

List | Creation

Here is how this list looks like:

```
experiment
## $Requester
## [1] "Oguz, Gokce"
##
## $Library_Kit
## [1] "Illumina TruSeq"
##
## $Read
##       Length          type
##        "150" "paired-end"
##
## $out
##     ID Machine Date
## 1 MUX1    HS08 1Jun
## 2 MUX2    HS08 2Jun
## 3 MUX3    HS06 1Jun
```

List | Operations

You can extract the length and names of the list:

```
length(experiment)
## [1] 4

names(experiment)
## [1] "Requester"   "Library_Kit" "Read"        "out"

str(experiment)
## List of 4
##  $ Requester  : chr "Oguz, Gokce"
##  $ Library_Kit: chr "Illumina TruSeq"
##  $ Read       : Named chr [1:2] "150" "paired-end"
##   ..- attr(*, "names")= chr [1:2] "Length" "type"
##  $ out        :'data.frame': 3 obs. of  3 variables:
##   ..$ ID     : Factor w/ 3 levels "MUX1","MUX2",..: 1 2
##   ..$ Machine: Factor w/ 2 levels "HS06","HS08": 2 2 1
##   ..$ Date   : Factor w/ 2 levels "1Jun","2Jun": 1 2 1
```

List | Extraction of multiple elements

You can use the same syntax you learned for vector extraction to extract or re-order elements in the list.

```
experiment[ c("Requester", "Library_Kit") ]
## $Requester
## [1] "Oguz, Gokce"
##
## $Library_Kit
## [1] "Illumina TruSeq"

experiment[ 1:2 ]    ## Using positions instead of names
## $Requester
## [1] "Oguz, Gokce"
##
## $Library_Kit
## [1] "Illumina TruSeq"
```

The output is always a list when using "[".

List | Extraction of a single element

You can also extract a single element from the list using "[[" or $.

```
experiment[[ "Read" ]]     ## Output is a vector
##       Length           type
##        "150" "paired-end"

experiment$Read            ## Output is a vector
##       Length           type
##        "150" "paired-end"
```

Note: The output is a list using "[". i.e.

```
experiment[ "Read" ]     ## Output is a list
## $Read
##       Length           type
##        "150" "paired-end"
```

List | Creation and destruction

You can also add a new element using the "[[" syntax.

```
experiment[[ "Completed" ]] <- TRUE

names(experiment)
## [1] "Requester" "Library_Kit" "Read" "out" "Completed"

str(experiment)
## List of 5
##  $ Requester  : chr "Oguz, Gokce"
##  $ Library_Kit: chr "Illumina TruSeq"
##  $ Read       : Named chr [1:2] "150" "paired-end"
##   ..- attr(*, "names")= chr [1:2] "Length" "type"
##  $ out        :'data.frame': 3 obs. of  3 variables:
##   ..$ ID     : Factor w/ 3 levels "MUX1","MUX2",..: 1 2
##   ..$ Machine: Factor w/ 2 levels "HS06","HS08": 2 2 1
##   ..$ Date   : Factor w/ 2 levels "1Jun","2Jun": 1 2 1
##  $ Completed  : logi TRUE
```

List | Creation and destruction

Likewise, you can remove an element by setting it to NULL

```
experiment[ c("Requester", "Completed") ] <- NULL

experiment
## $Library_Kit
## [1] "Illumina TruSeq"
##
## $Read
##       Length          type
##        "150" "paired-end"
##
## $out
##     ID Machine Date
## 1 MUX1    HS08 1Jun
## 2 MUX2    HS08 2Jun
## 3 MUX3    HS06 1Jun
```

THE END :)