Attributes are arbitrary labels attached to R objects.

Can be set by hand, but should rarely be done by hand.

Example of attributes:

- names
- class
- dim (every object has an attribute with its dimension)

In [2]:
```
x <- runif(10)
names(x) <- LETTERS[1:10] # or letters for small letters
x
```

```
A   0.57487219828181
B   0.0770643802825361
C   0.0355405795853585
D   0.642795492196456
E   0.928615199634805
F   0.598092422354966
G   0.560900748008862
H   0.526027723914012
I   0.985095223877579
J   0.507641822332516
```

mode = simplified version of type. -> everything that is a number (e.g. integer, double, etc.) is called numeric, everything that is a function (e.g. builtin, closure, special) a function -> can be useful to perform check (e.g. to test if the variable is a number and you don't care if it is integer or double) closure = function in R

In [3]:
```
# mode as simplified version of type (upper layer of type)
mode
```

```
function (x)
{
    if (is.expression(x))
        return("expression")
    if (is.call(x))
        return(switch(deparse(x[[1L]])[1L], `(` = "(", "call"))
    if (is.name(x))
        "name"
    else switch(tx <- typeof(x), double = , integer = "numeric",
        closure = , builtin = , special = "function", tx)
}
```

vectors can only take values from the same type, will convert data from different types to the upper type (e.g. integer converted to numeric)

In [8]:
```
# numeric and sttring
c(1,"a",2)
#=> convert to character

# numeric, string and function
c(1, "a", c)
# => convert to list, as it allows elements of different types
```

```
'1'  'a'  '2'
```

1. 1
2. 'a'
3. .Primitive("c")

In [5]:
```
data <- rnorm(1000000)
# how many are about 0 ?
sum(data > 0)
sum(data)/length(data)
# but can also be done with logical
# => for counting the proportion -> mean of logical
mean(data > 0)
```

In [ ]:
```
In contrast to other languages, logical and numeric cannot be freely exc
hanged
```

In [6]:
```
vect <- 1:10
vect[c(0,1)]
# => with numeric, just retrieves element of corresponding index
vect[c(TRUE, FALSE)]
# => with logical, recycling
```

```
1

1  3  5  7  9
```

In [9]:
```
Main data type = vector, one of the only data types

Matrices and arrays are extension of vectors, where 2 or more dimension
are specified
```

```
Error in parse(text = x, srcfile = src): <text>:1:6: unexpected symbol
1: Main data
         ^
Traceback:
```

In [10]:
```
m <- matrix(1:30, ncol=6)
# the square brackets are function => we can add arguments after the com
a
m[11]
m[1,3]
dim(m)
length(m)
```

```
11

11

5  6

30
```

In [ ]:
```
a <- matrix(1:30)
# stores it as a vector
# class and dimension that differ
```

In [ ]:
```
mylist[1]
# => return a sublist, is still a list; returns part of the list

mylist[[1]]
# => get what is inside the list; returns what is inside this element
```

More flexible than matrices, can have different types in it the class is data frame, but typeof (=> low-level of the data) is list

As long as unambiguous, can use name shortcut, but don't do that !

summary() => summary of what the object contains; but the function str() is in most cases more useful

anova with numeric -> will do a linear regression instead of an ANOVA output with only 1 row => just 2 groups or linear regression

factors = vectors but constraints on the values they can contain

internally: integer numbers, 1st category = 1, 2nd category = 2

concatenating factors will not work -> will create a vector with numberic values

In [11]:
```
hair1 <- factor(c("red", "brown",  "blue"), levels = c("red", "brown", "
blue"))
hair2 <- factor(c("white", "green",  "grey"), levels = c("white", "gre
y", "green"))

c(hair1, hair2)

# for concatenating factors:
unlist(list(hair1, hair2))
# ... or:
c(as.character(hair1), as.character(hair2))
```

```
1  2  3  1  3  2

red   brown   blue   white   green   grey

'red'   'brown'   'blue'   'white'   'green'   'grey'
```

In [13]:
```
#ordered = TRUE => can be used for comparison
time <- factor(c("sometimes", "always", "never"), levels=c("never", "som
etimes", "always"), ordered=T)
time[2] < time[3]
```

```
FALSE
```

In previous versions of R: more memory efficient to use factors (e.g. store only 1, 2, etc. instead of the long string). But is not more a reason, because now stores only once each occurence of a string in a vector.

Object-oriented programming in R

S3 => what is used in the base R packages

S4 => have to specifically say that we create a new object (everything in bioconductor uses S4)

Objects and function: example of summary(): summary() is defined as a generic function, what the function does depends on the class of the object.

R will look for a summary.() function. If the function does not exist, call summary.default()

methods("summary") => list of all the summary functions implemented for the different classes

S3 classes

In [14]:
```
methods("summary")
```

```
 [1] summary.aov                    summary.aovlist*
 [3] summary.aspell*                summary.check_packages_in_dir*
 [5] summary.connection             summary.data.frame
 [7] summary.Date                   summary.default
 [9] summary.ecdf*                  summary.factor
[11] summary.glm                    summary.infl*
[13] summary.lm                     summary.loess*
[15] summary.manova                 summary.matrix
[17] summary.mlm*                   summary.nls*
[19] summary.packageStatus*         summary.PDF_Dictionary*
[21] summary.PDF_Stream*            summary.POSIXct
[23] summary.POSIXlt                summary.ppr*
[25] summary.prcomp*                summary.princomp*
[27] summary.proc_time              summary.srcfile
[29] summary.srcref                 summary.stepfun
[31] summary.stl*                   summary.table
[33] summary.tukeysmooth*
see '?methods' for accessing help and source code
```

In [15]:
```
# to see the body of non-visible functions:
getS3method("summary", "princomp")
getAnywhere("summary.princomp")
```

```
function (object, loadings = FALSE, cutoff = 0.1, ...)
{
    object$cutoff <- cutoff
    object$print.loadings <- loadings
    class(object) <- "summary.princomp"
    object
}
```

```
A single object matching ‘summary.princomp’ was found
It was found in the following places
  registered S3 method for summary from namespace stats
  namespace:stats
with value

function (object, loadings = FALSE, cutoff = 0.1, ...)
{
    object$cutoff <- cutoff
    object$print.loadings <- loadings
    class(object) <- "summary.princomp"
    object
}
<bytecode: 0x5202d50>
<environment: namespace:stats>
```

Another exemple of generic functions: print(): if not implemented, just the content of the object that is printed

```
In [ ]: # create a list containing all the attributes your object need
        # (any R object could be used, but lists are almost always used)
        mygsea2 <- function(small.list, big.list) {
            #[...]
            #z <- list(ks.pos = res, [...])
            z <- list()
            class(z) <- "gsea"
        }

        # lists most often used for their flexibility


        # an object can have multiple classes !!!
        # => test with
        # if(!any(class(object) == "gsea"))
        # => ... and not
        # if(class(object) == "gsea"))

        # create a function to print information about the object
        print.gsea <- function(){}

        # the dispatch is not needed if the function already exists in R (e.g. p
        rint, plot, etc.)

        # for print() -> dispatch not needed
        reduce.gsea <- function()

        # Dispatch: tell R the function is implemented for this class
        reduce <- function(object) UseMethod("reduce")

        # !!! to access object elements -> use implemented functions, not direct
        ly list elements !!!
```

If the method we create for the class does not already exist in R, we need to do the dispatch. If the function already exists (print, plot, etc.), the dispatch is not needed.

```
In [16]: # to get all the generic methods implemented for a class:
         methods(class="lm")

 [1] add1           alias          anova          case.names     coerce
 [6] confint        cooks.distance deviance       dfbeta         dfbetas
[11] drop1          dummy.coef     effects        extractAIC     family
[16] formula        hatvalues      influence      initialize     kappa
[21] labels         logLik         model.frame    model.matrix   nobs
[26] plot           predict        print          proj           qr
[31] residuals      rstandard      rstudent       show           simulate
[36] slotsFromS3    summary        variable.names vcov
see '?methods' for accessing help and source code
```

S4 classes

We have to explicitly say what the class should contain.

Validation function that will refuse to create an object if not fullfilled. -> useful to create object to do quality control of the data we are using !

Slots = attributes of the object

showMethods() => to see the methods defined for an object

! do not access object attributes with the @ but use functions that provide access to the slots !

In [19]:
```
showMethods("show")
```

```
Function: show (package methods)
object="ANY"
object="classGeneratorFunction"
object="classRepresentation"
object="envRefClass"
object="externalRefMethod"
object="genericFunction"
object="genericFunctionWithTrace"
object="MethodDefinition"
object="MethodDefinitionWithTrace"
object="MethodSelectionReport"
object="MethodWithNext"
object="MethodWithNextWithTrace"
object="namedList"
object="ObjectsWithPackage"
object="oldClass"
object="refClassRepresentation"
object="refMethodDef"
object="refObjectGenerator"
object="signature"
object="sourceEnvironment"
object="traceable"
```

In [20]:
```
showMethods(class="GSEA")
```

```
Function "asJSON":
 <not an S4 generic function>
```

S3 to interface with base functions (e.g. linear models, etc.) For bioconductor, use S4. S4 format fits with the format of C++.

names(object) => empty for S4 object issS4(object) => TRUE for S4 object

methods(class = "class") => for S3 showMethods(class="class") => for S4

method.class => for S3 getMethods("method", "class") => for S4

$ => for S3 @ => for S4

Another class: ReferenceClasses (R5)