‘frame’ an S4 class inheriting from data.frame

Find me on github

A class inheriting from data.frame featuring `drop=FALSE` as default, an ‘end’ keyword, 0 index for ‘all’ and a ‘desc’ field.

The problem

Despite very powerful, R language is mostly intended for interactive use. So in `x[i,j]` it is not straight to set such a value for `i` or `j` to emulate `x[,j]` or `x[i,]`.

Using `TRUE` (which will get recycled) is a trick to obtain this.

```
T=TRUE

x=matrix(1:12, ncol=3)
identical(x[,3], x[T,3])
```

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

```
identical(x[,3], `[`(x,T,3))
```

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

```
z=y=x
x[,3] =0
y[T,3]=0
identical(x,y)
```

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

```
z=`[<-`(z,T,3,0)
identical(x,z)
```

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

This works also for a data.frame.

```
x=data.frame(matrix(1:12, ncol=3))
identical(x[,3], x[T,3])
```

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

```
identical(x[,3], `[`(x,T,3))
```

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

```
z=y=x
x[,3] =0
y[T,3]=0
identical(x,y)
```

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

```
z=`[<-`(z,T,3,0)
identical(x,z)
```

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

Also the default `drop=TRUE` feature means breaking some programmatic code, like simple `nrow`, `ncol`, which will fail if the dimensions are dropped.

All these features are blessed time savers in interactive code, but oblige to many if/else check, prone to errors, in programmatic use.

The solution

I developed a new data.frame S4 class, `frame`, inheriting from data.frame but exploiting the `0`, unused in subsetting, to select all rows or columns, so `x[0,j]` and `x[i,0]` as `x[,j]` and `x[i,]`. Also `drop=FALSE` is used as the default value in place of `drop=TRUE`. Besides it uses a Matlab like `end` operator to get the last rows or columns item, e.g.: `x[3:end, 2:end]`. A `desc` field can be added to annotate any data description.

The old data.frame and the new one

This class inherits from the data.frame class: if a function works with a data.frame it is supposed to work with 'frame'. If your code requires a formal data.frame, you can obtain it from `x` frame with:

```
dfm(x)
```

This is a shortcut for `as.data.frame` which in turn will call the specialised function `as.data.frame.frame`.

What you get on the field

```
## Create a frame, from scratch (with column names)
frame(x, col.names=letters[1:3], desc="Hello frame")
```

```
## Frame 4x3
```

```
## : Hello frame
```

```
##   a b c
## 1 1 5 0
## 2 2 6 0
## 3 3 7 0
## 4 4 8 0
```

```
frame(1:4, 5:8, 9:12, desc="Hello frame")
```

```
## Frame 4x3: Hello frame
```

```
##   1:4 5:8 9:12
## 1   1   5    9
## 2   2   6   10
## 3   3   7   11
## 4   4   8   12
```

```
## or use a data.frame
x=data.frame(1:4, 5:8, 9:12)
frame(x, desc="Hello frame")
```

```
## Frame 4x3: Hello frame
```

```
##   X1.4 X5.8 X9.12
## 1    1    5     9
## 2    2    6    10
## 3    3    7    11
## 4    4    8    12
```

Note the option for frame only `col.names`. If you don't use it, data.frame by default checks column names, frame does not, as it gives a somewhat unpleasant view. If you need it, use `frame(1:4, 5:8, 9:12, desc="Hello frame", check.names = TRUE)`. Similarly a very annoying feature of data.frame is the automatic conversion of strings to factors. Here instead:

```
x=frame(a=1:26, b=letters)
class(x$b)
```

```
## [1] "character"
```

```
## ... while
x=data.frame(a=1:26, b=letters)
class(x$b) # unless you modified general R options
```

```
## [1] "factor"
```

You migh find convenient start from matrices:

```
frame(matrix(1:90, ncol=3), desc='Hello frame')
```

```
## Frame 30x3
```

```
## : Hello frame
##      1  2  3
## 1    1 31 61
## 2    2 32 62
## 3    3 33 63
## 4    4 34 64
## 5    5 35 65
## 6    6 36 66
## 7    7 37 67
## 8    8 38 68
## 9    9 39 69
## 10  10 40 70
## 11  11 41 71
## 12  12 42 72
## 13  13 43 73
## 14  14 44 74
## 15  15 45 75
## 16  16 46 76
## 17  17 47 77
## 18  18 48 78
## 19  19 49 79
## 20  20 50 80
```

```
## ...
```

```
as.frame(matrix(1:90, ncol=3), desc='Hello frame')
```

```
## Frame 30x3: Hello frame
```

```
##     V1 V2 V3
## 1    1 31 61
## 2    2 32 62
## 3    3 33 63
## 4    4 34 64
## 5    5 35 65
## 6    6 36 66
## 7    7 37 67
## 8    8 38 68
## 9    9 39 69
## 10  10 40 70
## 11  11 41 71
## 12  12 42 72
## 13  13 43 73
## 14  14 44 74
## 15  15 45 75
## 16  16 46 76
## 17  17 47 77
## 18  18 48 78
## 19  19 49 79
## 20  20 50 80
```

```
## ...
## use "col.names" argument, if you don't like the automatic naming
```

Note the ... signalling that only the first 20 rows were printed. Nothing is more boring than printing an object just to discover you get a useless time consuming gigantic print.

Let us see the standard subsetting.

```
(fr=as.frame(matrix(1:12, ncol=3)))
```

```
## Frame 4x3
```

```
##
##   V1 V2 V3
## 1  1  5  9
## 2  2  6 10
## 3  3  7 11
## 4  4  8 12
```

```
## Standard subsetting
fr[c(1,3),2:3]
```

```
## Frame 2x2
```

```
##
##   V2 V3
## 1  5  9
## 3  7 11
```

```
fr[2,]
```

```
## Frame 1x3
```

```
##
##   V1 V2 V3
## 2  2  6 10
```

```
fr[2,c('V1', 'V3')]
```

```
## Frame 1x2
```

```
##
##   V1 V3
## 2  2 10
```

```
fr[c(T,F,T,T),T]
```

```
## Frame 3x3
```

```
##
##   V1 V2 V3
## 1  1  5  9
## 3  3  7 11
## 4  4  8 12
```

```
# fr[2:3]  # error not supported
```

New features: subsetting without dropping dimensions by default!

```
## Non-standard behaviour:
## drop is false by default
fr[1,1]
fr[1,1, drop=T] #Traditional behaviour
fr[ ,1, drop=T] #Traditional behaviour

New features: `end` operator.
Inside brackets `end` is a keyword and it means the last row or column according to its position.
```

```
## Error: <text>:8:5: unexpected symbol
## 7:
## 8: New features
##        ^
```

```
(fr=as.frame(matrix(1:12, ncol=3)))
```

```
## Frame 4x3
```

```
##
##   V1 V2 V3
## 1  1  5  9
## 2  2  6 10
## 3  3  7 11
## 4  4  8 12
```

```
fr[end,end]  # 'end' is not quoted inside brackets!
```

```
## Frame 1x1
```

```
##
##   V3
## 4 12
```

```
## fr[end]   # error not supported
## fr[2:end] # error not supported
fr[2:end,end:1]
```

```
## Frame 3x3
```

```
##
##   V3 V2 V1
## 2 10  6  2
## 3 11  7  3
## 4 12  8  4
```

```
## 'end' is always a keyword inside brackets
end=100
fr[, end]
```

```
## Frame 4x1
```

```
##
##   V3
## 1  9
## 2 10
## 3 11
## 4 12
```

Let us examine subsetting with zero index. As an index zero means all (like `TRUE`).

```
## Zero means all
fr[0,1]
```

```
## Frame 4x1
```

```
##
##   V1
## 1  1
## 2  2
## 3  3
## 4  4
```

```
fr[1,0]
```

```
## Frame 1x3
```

```
##
##   V1 V2 V3
## 1  1  5  9
```

```
fr[0,0] # whole frame
```

```
## Frame 4x3
```

```
##
##   V1 V2 V3
## 1  1  5  9
## 2  2  6 10
## 3  3  7 11
## 4  4  8 12
```

Anyway sequences starting with zero are kept with the old meaning.

```
## Traditional zero behaviour
## For a DF [0:3,0:2] is like [1:3,1:2], and for 'frame' too
fr[0:3,0:2] # Your old code won't break
```

```
## Frame 3x2
```

```
##
##   V1 V2
## 1  1  5
## 2  2  6
## 3  3  7
```

Use in a programmatic environment

Consider this code snippet

```
size=function(x)
        if(nrow(x)>100) print("I am a long series") else
                         print("I am a short series")
```

It will break when a data.frame loses a dimension, due to the default `drop=TRUE`. Because `nrow` does not apply to atomic objects.

```
df=as.data.frame(matrix(1:12, ncol=3))
fr=frame(df)

## All columns
size(df)
```

```
## [1] "I am a short series"
```

```
## Take only the first col (we lose a dimension)
# size(df[,1])
# Error: argument is of length zero

## Take only the first col (now we don't lose original dimension)
size(fr[,1])
```

```
## [1] "I am a short series"
```

Assume that `fr` represents real observations.

```
fr=as.frame(matrix(rnorm(16), ncol=4))
```

Calculate the covariance matrix of some or all the `fr` columns. Generate a random frame like `fr`, calculate the same covariance matrix and obtain the differences.

```
dcov=function(fr,i,j){
        v1=fr[i,j]
        m=matrix(rnorm(nrow(fr) * ncol(fr)), ncol=ncol(fr))
        v2=as.frame(m)[i,j]
        var(v1)-var(v2)
}
```

Get delta-cov for all columns in `fr`:

```
dcov(fr, 0, 0)
```

```
##              V1          V2          V3          V4
## V1 -1.20717271  0.33114372 -0.80376393 -0.04201971
## V2  0.33114372  0.24550832 -0.06600963 -0.19319950
## V3 -0.80376393 -0.06600963  0.12442672  0.73472394
## V4 -0.04201971 -0.19319950  0.73472394 -0.19146884
```

Only columns starting from second:

```
dcov(fr, 0, '2:end') #Quoting necessary!!
```

```
##            V2         V3         V4
## V2 -2.7786743 -0.8103209 -0.2197724
## V3 -0.8103209  1.0137194  1.4579046
## V4 -0.2197724  1.4579046 -2.8973433
```

Note: Without quoting, if you have a variable `end=3`, you are sending `2:3`.

All columns, but remove last row:

```
dcov(fr, '-end', 0) #Quoting necessary!!
```

```
##            V1         V2         V3         V4
## V1  0.1870858 -0.2862340  0.3589954  0.2551383
## V2 -0.2862340 -2.4014519 -0.6133402  0.7794532
## V3  0.3589954 -0.6133402 -0.1734055  0.1492068
## V4  0.2551383  0.7794532  0.1492068 -0.2049813
```

‘end’ quotation

Inside brackets you *do not quote* ‘end’ to use it as a keyword for last element.

```
fr=as.frame(matrix(1:16, nrow=4))
end=100
fr[2:end,] #'end' is the last row (4 not 100)
```

```
## Frame 3x4
```

```
##
##    V1  V2  V3  V4
## 2   2   6  10  14
## 3   3   7  11  15
## 4   4   8  12  16
```

Outside brackets, that is in a programmatic environment when using ‘end’ keyword indirectly in a variable assignment, *quote* end!

```
end=1
a=end
b='end'

## Without quoting, variable value 'end' is 1
fr[1:2,a]
```

```
## Frame 2x1
```

```
##
##    V1
## 1   1
## 2   2
```

```
## By quoting, variable value 'end' is a keyword
fr[1:2,b]
```

```
## Frame 2x1
```

```
##
##    V4
## 1  13
## 2  14
```

If the assignment is an *expression* contains ‘end’ the same rule applies to get it as a keyword: *quote too*.

```
a='3:end'
b=0
fr[a,b]
```

```
## Frame 2x4
```

```
##
##    V1  V2  V3  V4
## 3   3   7  11  15
## 4   4   8  12  16
```

In assignments, without quoting ‘end’, it will get the current value of end, if any and valid, or raise an error. Inside brackets it will be passed unevaluated and after converted to a keyword. Quoting in brackets means querying for the row/column named “end”.

Bracket alternatives. These rules keep for bracket alternatives ‘[‘(x,i,j) and ‘[<-‘(x,i,j).

```
`[`(fr,end,end)
```

```
## Frame 1x1
```

```
##
##   V4
## 4 16
```

```
a='3:end'
`[`(fr,a,a)
```

```
## Frame 2x2
```

```
##
##   V3 V4
## 3 11 15
## 4 12 16
```

What if my column is named ‘end’?

```
fr=as.frame(matrix(1:16, nrow=4))
names(fr)[3]='end'
fr[1,]
```

```
## Frame 1x4
```

```
##
##   V1 V2 end V4
## 1  1  5   9 13
```

Inside brackets nothing special happens. As a keyword ‘end’ is unquoted, while as a name:

```
fr[, c('V1','end')]
```

```
## Frame 4x2
```

```
##
##   V1 end
## 1  1   9
## 2  2  10
## 3  3  11
## 4  4  12
```

For assignments to a variable, outside brackets, quote twice.

```
a="'end'"
b="c('V1', 'end')"

fr[1:2,a]
```

```
## Frame 2x1
```

```
##
##   end
## 1   9
## 2  10
```

```
fr[1:2,b]
```

```
## Frame 2x2
```

```
##
##   V1 end
## 1  1   9
## 2  2  10
```

Obviously you can quote twice also with `sQuote`, `dQuote` or escaping inner quotation marks with "\".