Win-Vector Blog

The Win-Vector LLC data science blog

What can be in an R data.frame column?

April 9, 2015 John Mount Opinion, Programming, Rants, Statistics, Tutorials data frame, R, R is not your friend, R programming annoyances, types

As an R programmer have you every wondered what can be in a `data.frame` column?

The documentation is a bit vague, `help(data.frame)` returns some comforting text including:

> *Value*
>
> *A data frame, a matrix-like structure whose columns may be of differing types (numeric, logical, factor and character and so on).*

If you ask an R programmer the commonly depended upon properties of a `data.frame` columns are:

1. All columns in a data frame have the same length. (true, but with an asterisk)
2. All columns in a data frame are vectors with type (see `help(typeof)`) and class (see `help(class)`) deriving from one of the primitive types (such as: numeric, logical, factor and character and so on). (FALSE!)
3. (weakening from 2) All items in a single column in a data frame have the same number of items per row index. (FALSE)
4. (weakening from 3) All items in a single column in a data frame are at least homogeneous and have the type/class per row index. (FALSE, or that class/type can be `list` hiding heterogeneous nested entries)

Unfortunately only for first item is actually true. The `data.frame()` and `as.data.frame()` methods try to do some conversions to make more of the items in the above list are

usually true. We know `data.frame` is *implemented* as a list of columns, but the idea is the class `data.frame` overrides a lot of operators and should be able to maintain some useful invariants for us.

`data.frame` is one of R's flagship types. You would like it to have fairly regular and teachable observable behavior. (Though given the existence of the reviled `attach()` command I a beginning to wonder if `data.frame` was a late addition to `S`, the language R is based on.)

But if you are writing library code (like vtreat) you end up working with the data frames as they are, and not as you would like them to be.

Here is an example of the problem:

```
d <- data.frame(a=1:3)
d$v <- tapply(X=1:6,
              INDEX=c('a','a','b','b','c','c'),
              FUN=sum,
              simplify=TRUE)
print(class(d$a))
## [1] "integer"
print(class(d$v))
## [1] "array"
```

Even with the `simplify=TRUE` argument set, `tapply()` returns an array, and that array type survives when added to a `data.frame`. There is no implicit `as.numeric()` conversion to change from an array to a primitive vector class. Any code written under the assumption the columns of the data frame restrict themselves to simple classes and types will fail.

Case in point: earlier versions of `vtreat` would fail to recognize such a column as numeric (because the library was checking the class name, as I had falsely assumed the `is.numeric()` check was as fragile as the `is.vector()` checks) and treat the column as strings. And this is the cost of not having type strictness: there is no way to write concise correct code for dealing with other people's data. `vtreat` already had special case code for `POSIXlt` types (one way nested lists can get into data frames!), but I didn't have special code to check for lists and arrays in general. It isn't so much we used the wrong type-check (looking at `class()` instead of using `is.numeric()`, which can be debated) it is we failed to put in enough special case code to catch (or at least warn) on all the unexpected corner cases.

This is why I like type systems, they let you document (in a machine readable way, so you can also enforce!) the level of diversity of input you expect. If the inputs are not that diverse they you then have some chance that simple concise code can be correct. If the inputs are a diverse set of unrelated types that don't share common interfaces, then no concise code can be correct.

Many people say there is no great cost to R's loose type system, and I say there is. It isn't just my code. The loose types are why things like `ifelse()` are 30 lines of code instead of 5 lines of code (try `print(ifelse)`, you will notice the majority of the code is trying to strip off attributes and defend against types that are almost, but not quite what one would expect; only a minority of the code is doing the actual work). This drives up the expense of writing a fitter (such as: lm, glm, randomForest, gbm, rpart, ...) as to be correct the fitter may have to convert a number of odd types into primitives. And it drives up the cost of using fitters, as you have to double check the authors anticipated all types you end up sending. And you may not even know which types you are sending due to odd types entering through use of other libraries and functions (such as `tapply()`).

If your rule of code composition is Postel's law (instead of checkable types and behavior contracts) you are going to have very bloated code as each module is forced enumerate and correct a large number of "almost the same" behaviors and encodings. You will also have a large number of "rare" bugs as there is no way every library checks all corner cases, and each new programmer accidentally injects a different unexpected type into their work. When there are a large number of rare bugs lurking: then bugs are encountered often and diagnosing them is expensive (as each one feels unique).

When you work with systems that are full of special cases your code becomes infested with the need to handle special cases. Elegance and correctness become opposing goals instead of synergistic achievements.

Okay, I admit arrays are not that big a deal. But arrays are the least of your worries.

Columns of a data frame can be any the following types:

- `POSIXlt` a complicated list structure, making the column a nested list.
- Arbitrary lists (including ragged lists, and lists with different types in each row).
- Matrices.
- Data frames.

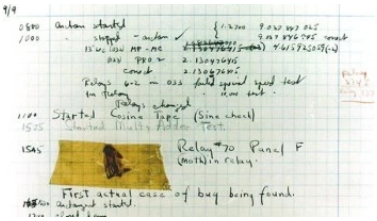
Below is an example of a pretty nasty data frame. Try `code()` and `typeof()` on various columns; try `str()` on various entries; and definitely try the `print(unclass(d[1,'xPOSIXlt']))` as it looks like `str()` hides the awful details in this case (perhaps it or something it depends on is overridden).

```
d <- data.frame(xInteger=1:3,
                xNumeric=0,
                xCharacter='a',
                xFactor=as.factor('b'),
                xPOSIXct=Sys.time(),
                xRaw=raw(3),
                xLogical=TRUE,
                xArrayNull=as.array(list(NULL,NULL,NULL)),
                stringsAsFactors=FALSE)
d$xPOSIXlt <- as.POSIXlt(Sys.time())
d$xArray <- as.array(c(7,7,7))
d$xMatrix <- matrix(data=-1,nrow=3,ncol=2)
d$xListH <- list(10,20,'thirty')
d$xListR <- list(list(),list('a'),list('a','b'))
d$xData.Frame <- data.frame(xData.FrameA=6:8,xData.FrameB=11:13)

print(colnames(d))
##  [1] "xInteger"    "xNumeric"    "xCharacter"  "xFactor"     "xPOSIXct"
##  [6] "xRaw"        "xLogical"    "xArrayNull"  "xPOSIXlt"    "xArray"
## [11] "xMatrix"     "xListH"      "xListR"      "xData.Frame"

print(d)
##   xInteger xNumeric xCharacter xFactor            xPOSIXct xRaw xLogical
## 1        1        0          a       b 2015-04-09 10:40:26   00     TRUE
## 2        2        0          a       b 2015-04-09 10:40:26   00     TRUE
## 3        3        0          a       b 2015-04-09 10:40:26   00     TRUE
##   xArrayNull            xPOSIXlt xArray xMatrix.1 xMatrix.2 xListH xListR
## 1       NULL 2015-04-09 10:40:26      7        -1        -1     10   NULL
## 2       NULL 2015-04-09 10:40:26      7        -1        -1     20      a
## 3       NULL 2015-04-09 10:40:26      7        -1        -1 thirty   a, b
##   xData.Frame.xData.FrameA xData.Frame.xData.FrameB
## 1                        6                       11
## 2                        7                       12
## 3                        8                       13

print(unclass(d[1,'xPOSIXct']))
## [1] 1428601226

print(unclass(d[1,'xPOSIXlt']))
...
```

(Note: neither `is.numeric(d$xPOSIXct)` or `is.numeric(d$xPOSIXlt)` is true, though both pass nicely through `as.numeric()`. So even `is.numeric()` doesn't signal everything we need to know about the ability to use a column as a numeric quantity.)

(Also notice `length(d$xData.Frame)` is 2: the number of columns of the sub-data frame. And it is not 3 or `nrow(d$xData.Frame)`. So even the statement "all columns have the same length" needs a bit of an asterisk by it. The columns all have the same length- but not the length returned by the `length()` method. Also note `nrow(c(1,2,3))` return `NULL` so you can't use that function everywhere either.)

RELATED



Factors are not first-class citizens in R
In "Computer Science"

R bracket is a bit irregular
In "Coding"

What is new in the vtreat library?
The Win-Vector LLC vtreat library is a library we supply (under a GPL license) for automating the simple domain independent part of variable
In "Practical Data Science"

April 9, 2015 John Mount Opinion, Programming, Rants, Statistics, Tutorials data frame, R, R is not your friend, R programming annoyances, types

3 thoughts on "What can be in an R data.frame column?"

kablag

April 9, 2015 at 10:46 pm

I think you can add another interesting method with I() function.
For example :
d$xListR2 <- I(list(NULL,'a',c('a','b')))

jmount

April 10, 2015 at 7:03 am

kablag,

Great example, thanks! I played with it a bit more and broke `data.frame`'s built in print function!

```
d <- data.frame(xInteger=1:3)
d$xData.Frame <- I(data.frame(xData.FrameA=6:8,xData.FrameB=11:13))
tryCatch(print(d),
         error=function(e) print(e))
## <simpleError in format.AsIs(x[[i]], ..., justify = justify): dims [product
6] do not match the length of object [2]>
for(i in seq_len(nrow(d))) {
  print(d[i,])
}
##   xInteger xData.Frame.xData.FrameA xData.Frame.xData.FrameB
## 1        1                        6                       11
##   xInteger xData.Frame.xData.FrameA xData.Frame.xData.FrameB
## 2        2                        7                       12
##   xInteger xData.Frame.xData.FrameA xData.Frame.xData.FrameB
## 3        3                        8                       13
```

So obviously this `data.frame` is so bad it violates assumptions in `print.data.frame`. Not even base-lang can anticipate all the unintended consequences of having so many corner cases.

---

kablag

April 11, 2015 at 2:36 am

Yes, using of I() breaks almost all functions :) and can't be used in real world. It's just an example.

Comments are closed.
