





Win-Vector Blog

R bracket is a bit irregular

 January 17, 2015  John Mount  [Coding](#), [Expository Writing](#), [Rants](#), [Statistics](#)
 [R](#), [R programming annoyances](#), [selection](#), [subset](#)

While skimming Professor Hadley Wickham's [Advanced R](#) I got to thinking about nature of the square-bracket or extract operator in [R](#). It turns out “[,]” is a bit more irregular than I remembered.

The [subsetting section](#) of [Advanced R](#) has a very good discussion on the subsetting and selection operators found in R. In particular it raises the important distinction of two simultaneously valuable but incompatible desiderata: simplification of results versus preservation of results.

The issue is: when you pull a single row or column out of R's most important structure (the data frame) do you get a data frame, a list, or a vector? Not all code that works on one of these types works equivalently across all of these types, so this can be a serious issue. We have written about this before (see [selection in R](#)). But it wasn't until we got more into teaching (and co-authored the book [Practical Data Science with R](#)) that we really appreciated how confusing this can be for the beginner.

Let's start with an example.

```
> d <- data.frame(x=c(1,2),y=c(3,4))
> print(d)
  x y
1 1 3
2 2 4
> print(d[1,])
  x y
1 1 3
> print(d[,1])
[1] 1 2
```

What we see is: when using the two-argument `[,]` extract operator on a simple data frame.

- Extracting a single row returns a data frame (confirm with the `class()` method).
- Extracting a single column returns a vector (instead of a data frame).

And this is pretty much what a user sitting in front of an interactive system would want: simplification on columns and preservation on rows. And this is compatible with R's history as an interactive analysis system (versus as a batch programming language, as outlined [here](#)).

Where we run into trouble is when we are writing code that we expect to run correctly in all situations (even when we are not watching). Consider the following example.

```
> selector1 <- c(TRUE, FALSE)
> selector2 <- c(TRUE, TRUE)
> print(d[, selector1])
[1] 1 2
> print(d[, selector2])
  x y
1 1 3
2 2 4
```

In the first case our boolean selection vector returned a vector, and in the second case it returned a data frame. Believe it or not this is problem. If we were reading this code and the values of `selector1` and `selector2` were set somewhere else (say as the result of a complicated calculation) we would have no way of knowing what type would be returned by `d[, selector1]`. This even if we were lucky enough to have documentation asserting `selector1` and `selector2` are logical vectors of the correct length.

At runtime we can see how many positions of `selector1` are set to `TRUE`. But we can't reliably infer this count from looking at just an isolated code snippet. So we would not know at coding time what code would be safe to apply to the result `d[, selector1]`. The changing of the return type based on mere variation of argument value (not argument type) is very bad thing in terms of readability. A code reader can't set simple (non data-dependent) expectations on the code. Or they can't use assumed pre-conditions known about the inputs (such as documented type) to establish useful post-conditions (guaranteed behavior of the code).

Why should we care about prior expectations? Can't we just consider those uniformed presumptions and teach past them? To my mind this violates some concepts of efficient

learning and teaching. In my opinion there is no such thing as passive learning (or completely pure teaching). Students learn by thinking and base their expectations for new material by generalizing and regularizing lessons from older material. The more effective students can be at this the faster they learn.

Also, pity the student who makes a mistake while trying to learn about the square-bracket extraction operator through the R help system. If they accidentally type `help('[')` instead of `help('[.data.frame')`, then they see the following confusing help.



Figure: `help('[')`.

Instead of seeing the relevant definition, which is as follows.

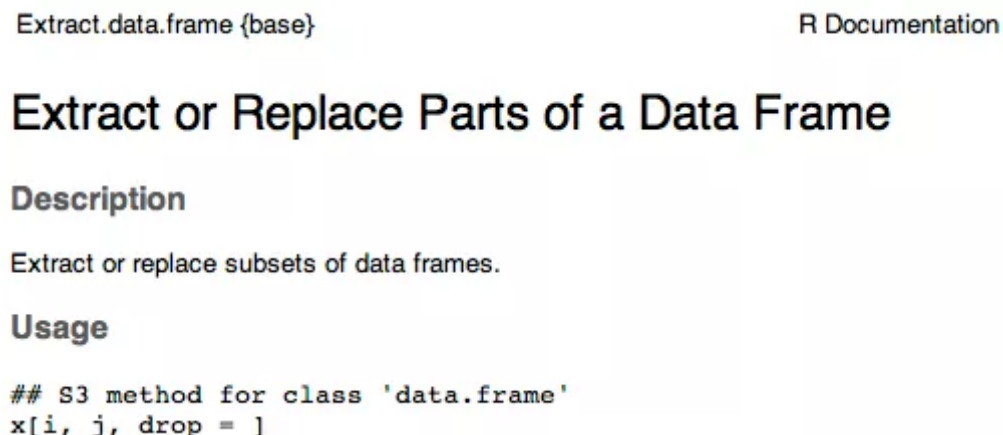


Figure: `help('[.data.frame')`.

Notice the first help implies there is an argument called `drop` that defaults to `TRUE`. This is true for matrices (what the help is talking about), but false for data frames (the

central class of R, nobody should choose R for the matrix operations). You could (informally) think of `[.data.frame]` as being a specialization of the base `[` in the sense of object-oriented inheritance. Except, it is considered very bad form to change the semantics or rules when extending types and operators. The expectations set in the base class (and especially those set in the base-class documentation) should hold in derived classes and methods.

We can confirm `[.data.frame,]` does not act like either of `[.data.frame, , drop=TRUE]` or `[.data.frame, , drop=FALSE]`. It picks its own behavior depending on if you end up with a single column or not (note: I didn't say "if you picked a single column or not"). The code below shows some of the variations in behavior.

```
> print(d[1,])
  x y
1 1 3
> print(d[,1])
[1] 1 2

> print(d[1,,drop=TRUE])
$x
[1] 1
$y
[1] 3
> print(d[,1,drop=TRUE])
[1] 1 2

> print(d[1,,drop=FALSE])
  x y
1 1 3
> print(d[,1,drop=FALSE])
  x
1 1
2 2
```

Notice how none of the complete results of these three experiments (running without the drop argument, running with it set to `TRUE`, and running with it set to `FALSE`) entirely match any of the others.

Also you can trigger the "only one column causes type conversion" issue even when you are not selecting on columns (in fact even when selecting the entire data frame!):

```
> d1 <- data.frame(x=c(1,2))
> print(d1)
```

```

      x
1 1
2 2
> print(d1[c(TRUE,TRUE),])
[1] 1 2

```

This is a good point to return to the article about the historic context and influences of R, which gives us the following quote:

Pat begins with how R began as an experimental offshoot from S (there's an adorable 1990's-era photo of R's creators Ross Ihaka and Robert Gentleman in Auckland on page 23, reproduced below), and then evolved into a language used first interactively, and then for programming. The tensions between the two modes of use led to some of the quirkier aspects of R. (Pat's moral: "if you want to create a beautiful language, for god's sake don't make it useful".)

How would I like R to behave if it evolved anew and didn't have to support older code? I'd like (but know I can't have) the following:

- `[,]` is reserved to select sets of rows and columns and by default guarantees "preserving" behavior in all cases (i.e. all variations of `[,]` default to `drop=FALSE`).
- `[[[]]` is reserved for extracting a single item and is "simplifying".
- To extract a single column as a vector from a data frame you must use the single argument list operator `[[[]]`.
- In all cases `[[[]]` signals an error if you do not select exactly one element.

When I say I want these things: understand this means both I already known this is not the way they are and I know (for practical reasons) they can not be changed to be so. The fact that none of the above statements as currently true will come as a surprise to many R users. For example it is widely thought that `[[[]]` behaves everywhere as it behaves on lists: properly signaling errors if you try to select more than one element. Notice this does not turn out to be the case. For vectors and lists we have good error-indicating behaviors:

```

> c(1,2,3)[[c(1,2)]]
Error in c(1, 2, 3)[[c(1, 2)]] : attempt to select more than one element
> list(1,2,3)[[c(1,2)]]
Error in list(1, 2, 3)[[c(1, 2)]] : subscript out of bounds

```

```
> list(1,2,3)[[2]]
```

For data frames we have a less desirable “anything goes” situation:

```
> d[[c(1,2)]]
[1] 2
```

Remember: a situation that should have signaled an error and did not is worse than a situation with a signaling error. (Note: `subset(d1, x==1, select=c('x'))` seems to reliably avoid unwanted simplification, but is not advised as it invokes non-standard evaluation issues. Look at `getS3method('subset', 'data.frame')` for details.)

Data frames are guaranteed to be lists of columns (a publicly exposed implementation detail, a bit obscured by the fact that the derived two-argument operator `[,]` superficially appears to be row-oriented). So we would expect `d[[c(1,2)]]` to properly error-out as it does for lists. However, it appears to behaving more like a two-dimensional index operator. Probably some code is using this, but it is a pretty clear violation of exceptions (especially for a new student). Repeating: data frames are lists of columns (you can check this with `unclass(d)`) and this is not a hidden implementation detail (it is commonly discussed and expected). ~~However the `[[.data.frame]` operator has extended or overridden behavior that is different than any notional base `[[]` method/operator.~~ (Please see comments below for corrections on `d[[c(1,2)]]`.)

One of the reasons we need two extraction operators (`[]` and `[[[]]`) is: R does not expose true scalar types (even the number 3 is in length-1 vector) so we have no convenient way to signal (even using runtime types) if we thought we were coding a set-based extraction (through a set/vector of indices or a vector of booleans) or a scalar based extraction (through a single index, the case where simplification is most likely to be desirable). It is likely the designers understood that return types changing on mere change in values of arguments (and not in more fundamental changes of types of arguments) is confusing and undesirable (as it eliminates any chance at pure type to type reasoning) that led to S/R having so many extraction/selection operators. They saw the need to isolate and document different behaviors. However these abstractions turn out to be a bit leaky.

For my part I teach designing your code assuming you had simple regular versions of the above operators, and then implementing defensively (specifying `drop`, and preferring `subset()` and `[[[]]` to `[]`) to ensure you get good regular behavior.

SHARE THIS:



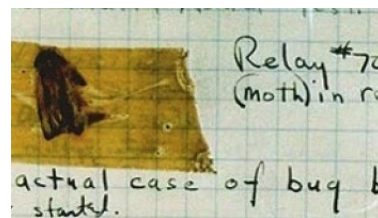
RELATED

Selection in R

The design of the statistical programming language R sits in a slightly uncomfortable place between the functional programming and object In "Pragmatic Machine Learning"

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"

**My favorite R bug**

In "Expository Writing"

📅 January 17, 2015 👤 John Mount 📁 Coding, Expository Writing, Rants, Statistics 🔖 R, R programming annoyances, selection, subset

5 thoughts on “R bracket is a bit irregular”

Radford Neal

January 18, 2015 at 4:03 pm

You're mistaken in thinking that `d[[c(1,2)]]` has been given a special meaning by the `data.frame` code. You get the same result with `unclass(d)[[c(1,2)]]`. A vector subscript for `[[` is used to subscript recursively, using the first subscript first, then using the second to subscript what the first got, etc.

Possibly the inconsistency with subscripting matrices should resolved by making `M[[c(1,2)]]` give `M[,1][2]`, though I'd want to think about it more first.

jmount

January 18, 2015 at 8:16 pm

You are right, thanks for the correction. I definitely should have confirmed that with `unclass(d)` as you did. I am going to add a note to the article so I don't mislead others.

I don't think it would be easy or desirable to get matching behavior from matrices. `as.matrix(d)[[c(1,2)]]` errors out and `as.matrix(d)[[1]]` reaches into the underlying

one-dimensional vector that stores the matrix entries.

Nick

January 20, 2015 at 1:15 pm

This is a bit clunky, but since data.frames are lists, and extracting list elements only involves one dimension, I thought that extracting one or more columns from a data.frame could be accomplished by:

```
as.data.frame(as.list(d)[selector1])
as.data.frame(as.list(d)[selector2])
```

This seems to give the same results as:

```
d[, selector1, drop = FALSE]
d[, selector2, drop = FALSE]
```

On the other hand, these give slightly different results:

```
selector0 <- c(FALSE, FALSE)
as.data.frame(as.list(d)[selector0])
d[, selector0, drop = FALSE]
```

In the first case, the result is a data frame with zero rows and zero columns, but in the second case, the data frame has 2 rows and zero columns.

jmount

January 26, 2015 at 5:42 pm

Oh man, I think I know what is going on in the following- but it is really hard to feel “good” about it:

```
> data.frame(a=c(1,2),b=c(3,4),
  c=c(5,6),d=c(7,8))[ , c(TRUE, FALSE)]
  a c
1 1 5
2 2 6
```

David Winsemius

February 16, 2015 at 9:39 am

@jmount: Regarding using '[.data.frame]' with logical vectors: "It's a feature!" Logical substituting in either the row or column position gets recycled. I realize that recycling seems irregular and surprises many newcomers to R but it is a well-described feature of the language. Makes it easy to select every n-th row or every n-th column.

Comments are closed.

Proudly powered by WordPress