

# Package ‘data.table’

March 26, 2012

**Version** 1.8.0

**Title** Extension of data.frame for fast indexing, fast ordered joins, fast assignment, fast grouping and list columns.

**Author** M Dowle, T Short, S Lianoglou

**Maintainer** M Dowle <datatable-help@lists.r-forge.r-project.org>

**Depends** R (>= 2.12.0)

**Imports** methods

**Suggests** chron, ggplot2 (>= 0.9.0), plyr, reshape, testthat (>= 0.4), hexbin, fastmatch, nlme

**Description** Enhanced data.frame. Fast indexing, fast ordered joins, fast assignment, fast grouping and list columns in a short and flexible syntax. i and j may be expressions of column names directly, for faster development. Example: X[Y] is a fast join for large data.

**License** GPL (>= 2)

**URL** <http://datatable.r-forge.r-project.org/>,  
<http://rwiki.sciviews.org/doku.php?id=packages:cran:data.table>,  
<http://crantastic.org/packages/data.table>

**BugReports** [https://r-forge.r-project.org/tracker/?group\\_id=240](https://r-forge.r-project.org/tracker/?group_id=240)

**Repository** CRAN

**Date/Publication** 2012-03-26 07:52:40

## R topics documented:

:=	2
all.equal	4
between	5
chmatch	6
data.table	8
data.table-class	14

duplicated . . . . .	15
IDateTime . . . . .	16
J . . . . .	19
last . . . . .	20
like . . . . .	21
merge . . . . .	22
setkey . . . . .	24
subset.data.table . . . . .	26
tables . . . . .	27
test.data.table . . . . .	28
timetaken . . . . .	29
transform.data.table . . . . .	30
truelength . . . . .	31

<b>Index</b>	<b>33</b>
--------------	-----------

---

:= *Assignment by reference*

---

## Description

Fast add, remove and modify subsets of columns, by reference.

## Usage

```
LHS := RHS          # in j only i.e. DT[i,LHS:=RHS]
```

## Arguments

LHS	A single column name. Or, when with=FALSE, a vector of column names or numeric positions (or a variable that evaluates as such). If the column doesn't exist, it is added, by reference.
RHS	A vector of replacement values. It is recycled in the usual way to fill the number of rows satisfying i, if any. Or, when with=FALSE, a list of replacement vectors which are applied (the list is recycled if necessary) to each column of LHS. To remove a column use NULL.

## Details

:= is defined for use in j only. This syntax *updates* the column(s) by reference. It makes no copies of any part of memory at all. Typical usages are :

```
DT[i,colname:=value]
DT[i,"colname" :=value,with=FALSE]
DT[i,(3:6):=value,with=FALSE]
DT[i,colnamevector:=value,with=FALSE]
```

The following all result in a friendly error (by design) :

```

x := 1L                                # friendly error
DT[i,colname] := value                 # friendly error
DT[i]$colname := value                 # friendly error

```

:= in `j` can be combined with all types of `i`, such as binary search.

When the LHS is a factor column and the RHS is a character vector with items missing from the factor levels, the new level(s) are automatically added (by reference, efficiently), unlike base methods.

Unlike `<-` for `data.frame`, the (potentially large) LHS is not coerced to match the type of the (often small) RHS. Instead the RHS is coerced to match the type of the LHS, if necessary. Where this involves double precision values being coerced to an integer column, a warning is given (whether or not fractional data is truncated). The motivation for this is efficiency. It is best to get the column types correct up front and stick to them. Changing a column type is possible but deliberately harder: provide a whole column as the RHS. This RHS is then *plonked* into that column slot and we call this *plonk syntax*, or *replace column syntax* if you prefer. By needing to construct a full length vector of a new type, you as the user are more aware of what is happening, and it's clearer to readers of your code that you really do intend to change the column type.

`data.tables` are *not* copied-on-write by `setkey`, `key<-` or `:=`. See [copy](#).

Additional resources: search for "==" in the [FAQs vignette](#) (3 FAQs mention :=), search Stack Overflow's [data.table tag for "reference"](#) (6 questions) and search `data.table`'s [wiki](#).

Advanced (internals) : sub assigning to columns is easy to see how that is done internally. Removing columns by reference is also straightforward by modifying the vector of column pointers only (using `memmove` in C). Adding columns is more tricky to see how that can be grown by reference: the list vector of column pointers is over-allocated, see [truelength](#). By defining `:= in j` we believe update syntax is natural, and scales, but also it bypasses `[<-` dispatch via `*tmp*` and allows `:=` to update by reference with no copies of any part of memory at all.

## Value

DT is modified by reference and the new value is returned. If you require a copy, take a copy first (using `DT2=copy(DT)`). Recall that this package is for large data (of mixed column types, with multi-column keys) where updates by reference can be many orders of magnitude faster than copying the entire table.

## See Also

[data.table](#), [copy](#), [alloc.col](#), [truelength](#)

**Examples**

```

DT = data.table(a=LETTERS[c(1,1:3)],b=4:7,key="a")
DT[,c:=8]      # add a numeric column, 7 for all rows
DT[,d:=9L]     # add an integer column, 8L for all rows
DT[,c:=NULL]   # remove the c column
DT[2,d:=10L]   # subassign by reference to column d
DT             # DT changed by reference

DT[b>4,b:=d*2L] # subassign to b using d, where b>4
DT["A",b:=0L]  # binary search for group "A" and set column b

## Not run:
DT[,newcol:=sum(v),by=group] # like fast transform() by group (not yet implemented)
## End(Not run)

# Speed example ...

m = matrix(1,nrow=100000,ncol=100)
DF = as.data.frame(m)
DT = as.data.table(m)
## Not run:
system.time(for (i in 1:1000) DF[i,1] <- i)
# 591 seconds
## End(Not run)
system.time(for (i in 1:1000) DT[i,V1:=i])
# 1.16 seconds ( 509 times faster )

```

---

all.equal

*Equality Test Between Two Data Tables*


---

**Description**

Performs some factor level “stripping” and other operations to allow for a convenient test of data equality between `data.table` objects.

**Usage**

```

## S3 method for class 'data.table'
all.equal(target, current, trim.levels = TRUE, ...)

```

**Arguments**

`target`, `current` data.tables to compare

`trim.levels` A logical indicating whether or not to remove all unused levels in columns that are factors before running equality check.

... Passed down to internal call of `all.equal.list`

**Details**

This function is used primarily to make life easy with a testing harness built around `test_that`. A call to `test_that::(expect_equal|equal)` will ultimately dispatch to this method when making an "equality" check.

**Value**

Either TRUE or a vector of mode "character" describing the differences between target and current.

**See Also**

[all.equal.list](#)

**Examples**

```
dt1 <- data.table(A = letters[1:10], X = 1:10, key = "A")
dt2 <- data.table(A = letters[5:14], Y = 1:10, key = "A")
identical(all.equal(dt1, dt1), TRUE)
is.character(all.equal(dt1, dt2))
```

---

between

*Convenience function for range subset logic.*

---

**Description**

Intended for use in `[.data.table i]`.

**Usage**

```
between(x, lower, upper, incbounds=TRUE)
x
```

**Arguments**

x	Any vector e.g. numeric, character, date, ...
lower	Lower range bound.
upper	Upper range bound.
incbounds	TRUE means inclusive bounds i.e. <code>[lower,upper]</code> . FALSE means exclusive bounds i.e. <code>(lower,upper)</code> .

**Value**

Integer vector containing the locations of x which lie within the range `[lower,upper]` or `(lower,upper)`.

**Note**

Current implementation does not make use of ordered keys.

**See Also**

[data.table](#), [like](#)

**Examples**

```
DT = data.table(a=1:5, b=6:10)
DT[b %between% c(7,9)]
```

---

chmatch

*Faster match of character vectors*

---

**Description**

chmatch returns a vector of the positions of (first) matches of its first argument in its second. Both arguments must be character vectors.

%chin% is like %in%, but for character vectors.

**Usage**

```
chmatch(x, table, nomatch=NA_integer_)
x %chin% table
chorder(x)
chgroup(x)
```

**Arguments**

x	character vector: the values to be matched, or the values to be ordered or grouped
table	character vector: the values to be matched against.
nomatch	the value to be returned in the case when no match is found. Note that it is coerced to integer.

**Details**

Fast versions of match, %in% and order, optimised for character vectors. chgroup groups together duplicated values but retains the group order (according the first appearance order of each group), efficiently. They have been primarily developed for internal use by data.table, but have been exposed since that seemed appropriate.

Strings are already cached internally by R (CHARSXP) and that is utilised by these functions. No hash table is built or cached, so the first call is the same speed as subsequent calls. Essentially, a counting sort (similar to `base::sort.list(x, method="radix")`, see [setkey](#)) is implemented using the (almost) unused truelength of CHARSXP as the counter. *Where R has used truelength of CHARSXP (where a character value is shared by a variable name), the non zero truelengths are stored first and reinstated afterwards.* Each of the ch\* functions implements a variation on this theme. Remember that internally in R, length of a CHARSXP is the nchar of the string and DATAPTR is the string itself.

Methods that do build and cache a hash table (such as the [fastmatch package](#)) are *much* faster on subsequent calls (almost instant) but a little slower on the first. Therefore `chmatch` may be particularly suitable for ephemeral vectors (such as local variables in functions) or tasks that are only done once. Much depends on the length of `x` and `table`, how many unique strings each contains, and whether the position of the first match is all that is required.

It may be possible to speed up `fastmatch`'s hash table build time by using the technique in `data.table`, and we have suggested this to its author. If successful, `fastmatch` would then be fastest in all cases.

### Value

As `match` and `%in%`. `chorder` and `chgroup` return an integer index vector.

### Note

The name `chmatch` was taken by [charmatch](#), hence `chmatch`.

### See Also

[match](#), [%in%](#), [fmatch](#)

### Examples

```
# Please type 'example(chmatch)' to run this and see timings on your machine

u = as.character(as.hexmode(1:10000))
y = sample(u,1e7,replace=TRUE)
x = sample(u)

system.time(a <- match(x,y))           # 4.8s
system.time(b <- chmatch(x,y))        # 0.9s  Faster than 1st fmatch
identical(a,b)
if (fastmatchloaded<-suppressWarnings(require(fastmatch))) {
  print(system.time(c <- fmatch(x,y)))  # 2.1s  Builds and caches hash
  print(system.time(c <- fmatch(x,y)))  # 0.00s  Uses hash
  identical(a,c)
}

system.time(a <- x %in% y)             # 4.8s
system.time(b <- x %chin% y)          # 0.9s
identical(a,b)
if (fastmatchloaded) {
  match <- fmatch                      # fmatch is drop in replacement
  print(system.time(c <- match(x,y)))   # 0.00s
  print(system.time(c <- x %in% y))    # 4.8s  %in% still prefers base::match
  # Anyone know how to get %in% to use fmatch (without masking %in% too)?
  rm(match)
  identical(a,c)
}

# Different example with more unique strings ...
u = as.character(as.hexmode(1:1e6))
y = sample(u,1e7,replace=TRUE)
```

```
x = sample(u,1e7,replace=TRUE)
system.time(a <- match(x,y))           # 34.0s
system.time(b <- chmatch(x,y))         # 6.4s
identical(a,b)
if (fastmatchloaded) {
  print(system.time(c <- fmatch(x,y)))   # 7.9s
  print(system.time(c <- fmatch(x,y)))   # 4.0s
  identical(a,c)
}
```

---

data.table

*Enhanced data.frame*


---

## Description

data.table *inherits* from data.frame. It offers fast subset, fast grouping, fast update, fast ordered joins and list columns in a short and flexible syntax, for faster development. It is inspired by A[B] syntax in R where A is a matrix and B is a 2-column matrix. Since a data.table is a data.frame, it is compatible with R functions and packages that *only* accept data.frame.

The 10 minute quick start guide to data.table may be a good place to start: [vignette\("datatable-intro"\)](#). Or, the first section of FAQs is intended to be read from start to finish and is considered core documentation: [vignette\("datatable-faq"\)](#). If you have read and searched these documents and the help page below, please feel free to ask questions on [datatable-help](#) or the Stack Overflow [data.table tag](#). To report a bug please type: `bug.report(package="data.table")`.

Please check the [homepage](#) for up to the minute [news](#).

Tip: one of the quickest ways to learn the features is to type `example(data.table)` and study the output at the prompt.

\*NEW\* : help page for `:=` keyby argument

## Usage

```
data.table(..., keep.rownames=FALSE, check.names=FALSE, key=NULL)
```

```
## S3 method for class 'data.table'
x[i, j, by, keyby, with=TRUE,
  nomatch = getOption("datatable.nomatch"), # default: NA_integer_
  mult = "all", roll = FALSE, rolltolast = FALSE,
  which = FALSE, .SDcols,
  verbose=getOption("datatable.verbose"), # default: FALSE
  drop=NULL]
```

## Arguments

... Just as ... in [data.frame](#). Usual recycling rules are applied to vectors of different lengths to create a list of equal length vectors.

keep.rownames If ... is a matrix or data.frame, TRUE will retain the rownames of that object in a column named rn.

check.names	Just as check.names in <code>data.frame</code> .
key	Character vector of one or more column names which is passed to <code>setkey</code> . It may be a single comma separated string such as <code>key="x,y,z"</code> , or a vector of names such as <code>key=c("x","y","z")</code> .
x	A <code>data.table</code> .
i	Integer, logical or character vector, expression of column names, <code>list</code> or <code>data.table</code> . integer and logical vectors work the same way they do in <code>[.data.frame]</code> . Other than NAs in logical <code>i</code> are treated as FALSE and a single NA logical is not recycled to match the number of rows, as it is in <code>[.data.frame]</code> . character is matched to the first column of <code>x</code> 's key. expression is evaluated within the frame of the <code>data.table</code> (i.e. it sees column names as if they are variables) and can evaluate to any of the other types. When <code>i</code> is a <code>data.table</code> , <code>x</code> must have a key. <code>i</code> is <i>joined</i> to <code>x</code> using the key and the rows in <code>x</code> that match are returned. An equi-join is performed between each column in <code>i</code> to each column in <code>x</code> 's key. The match is a binary search in compiled C in $O(\log n)$ time. If <code>i</code> has less columns than <code>x</code> 's key then many rows of <code>x</code> may match to each row of <code>i</code> . If <code>i</code> has more columns than <code>x</code> 's key, the columns of <code>i</code> not involved in the join are included in the result. If <code>i</code> also has a key, it is <code>i</code> 's key columns that are used to match to <code>x</code> 's key columns and a binary merge of the two tables is carried out. Advanced: When <code>i</code> is an expression of column names that evaluates to <code>data.table</code> or <code>list</code> , a join is performed. We call this a <i>self join</i> . Advanced: When <code>i</code> is a single variable name, it is not considered an expression of column names and is instead evaluated in calling scope. Advanced: When <code>i</code> is a regular <code>list</code> (such as <code>.BY</code> ), it is automatically converted to <code>data.table</code> .
j	A single column name, single expression of column names, <code>list()</code> of expressions of column names, an expression or function call that evaluates to <code>list</code> (including <code>data.frame</code> and <code>data.table</code> which are <code>lists</code> , too), or (when <code>with=FALSE</code> ) same as <code>j</code> in <code>[.data.frame]</code> . <code>j</code> is evaluated within the frame of the <code>data.table</code> ; i.e., it sees column names as if they are variables. Use <code>j=list(...)</code> to return multiple columns and/or expressions of columns. A single column or single expression returns that type, usually a vector. See the examples.
by	A single unquoted column name, <code>list()</code> of expressions of column names, or a single character string containing comma separated column names, or a character vector of column names. The <code>list()</code> of expressions is evaluated within the frame of the <code>data.table</code> (i.e. it sees column names as if they are variables). The <code>data.table</code> is then grouped by the <code>by</code> and <code>j</code> is evaluated within each group. The order of the rows within each group is preserved, as is the order of the groups. <code>j=list(...)</code> may be omitted when there is just one expression, for convenience, typically a single expression such as <code>sum(colB)</code> ; e.g., <code>DT[, sum(colB), by=colA]</code> . When <code>by</code> contains the first <code>n</code> columns of <code>x</code> 's key, we call this a <i>keyed by</i> . In a keyed <code>by</code> the groups appear contiguously in RAM and memory is copied in bulk internally, for extra speed. Otherwise, we call it an <i>ad hoc by</i> . <code>Ad hoc by</code> is

still many times faster than `tapply`, for example, but just not as fast as keyed by when datasets are very large, in particular when the size of *each group* is large.

Advanced: Aggregation for a subset of known groups is particularly efficient when passing those groups in `i`. When `i` is a `data.table`, `DT[i, j]` evaluates `j` for each row of `i`. We call this *by without by* or *grouping by i*. Hence, the self join `DT[data.table(unique(colA)), j]` is identical to `DT[, j, by=colA]`.

Advanced: Objects `.SD`, `.BY` and `.N` may be used in the `j` expression. `.SD` is a `data.table` containing the **S**ubset of `x`'s **D**ata for each group, excluding the group column(s). It can be used when grouping by `i`, when grouping by `by`, keyed by, and ad hoc by. `.BY` is a `list` containing a length 1 vector for each item in `by`. This can be useful when `by` is not known in advance. The `by` variables are also available to `j` directly by name; useful for example for titles of graphs if `j` is a plot command, or to branch with `if()` depending on the value of a group variable. `.N` is an integer, length 1, containing the number of rows in the group. This may be useful when the column names are not known in advance and for convenience generally. When grouping by `i`, `.N` is the number of rows in `x` matched to, for each row of `i`, regardless of whether `nomatch` is `NA` or `0`. `.SD`, `.BY` and `.N` are *read only*. Their bindings are locked and attempting to assign to them will generate an error. If you wish to manipulate `.SD` before returning it, take a copy(`.SD`) first (see FAQ 4.5). Using `:=` in the `j` of `.SD` may appear to change a copy of `.SD`, currently, but in future is intended to be a (tortuously flexible) way to update `DT` by reference by group (even when groups are not contiguous in an ad hoc by).

Advanced: In the `X[Y, j]` form of grouping, the `j` expression sees variables in `X` first, then `Y`. We call this *join inherited scope*. If the variable is not in `X` or `Y` then the calling frame is searched, its calling frame, and so on in the usual way up to and including the global environment.

<code>keyby</code>	An <i>ad hoc by</i> just as <code>by</code> but with an additional <code>setkey()</code> on the <code>by</code> columns of the result, for convenience. Not to be confused with a <i>keyed by</i> as defined above.
<code>with</code>	By default <code>with=TRUE</code> and <code>j</code> is evaluated within the frame of <code>x</code> . The column names can be used as variables. When <code>with=FALSE</code> , <code>j</code> works as it does in <code>[.data.frame]</code> .
<code>nomatch</code>	Same as <code>nomatch</code> in <code>match</code> . When a row in <code>i</code> has no match to <code>x</code> 's key, <code>nomatch=NA</code> (default) means <code>NA</code> is returned for <code>x</code> 's non-join columns for that row of <code>i</code> . <code>0</code> means no rows will be returned for that row of <code>i</code> . The default value (used when <code>nomatch</code> is not supplied) can be changed from <code>NA</code> to <code>0</code> using <code>options(datatable.nomatch=0)</code> .
<code>mult</code>	When <i>multiple</i> rows in <code>x</code> match to the row in <code>i</code> , <code>mult</code> controls which are returned: "all" (default), "first" or "last".
<code>roll</code>	Applies to the last join column, generally a date but can be any ordered variable, irregular and including gaps. If <code>roll=TRUE</code> and <code>i</code> 's row matches to all but the last <code>x</code> join column, and its value in the last <code>i</code> join column falls in a gap (including after the last observation in <code>x</code> for that group), then the <i>prevailing</i> value in <code>x</code> is <i>rolled</i> forward. This operation is particularly fast using a modified binary search. The operation is also known as last observation carried forward (LOCF). Usually, there should be no duplicates in <code>x</code> 's key, the last key column is a date (or time, or <code>datetime</code> ) and all the columns of <code>x</code> 's key are joined to. A common idiom is to select a contemporaneous regular time series (dts) across a set of

	identifiers (ids): DT[CJ(ids,dts),roll=TRUE] where DT has a 2-column key (id,date) and CJ stands for <i>cross join</i> .
rolltolast	Like roll but the data is not rolled forward past the <i>last</i> observation. The value of i must fall in a gap in x but not after the end of the data for that group defined by all but the last join column. roll and rolltolast may not both be TRUE.
which	TRUE returns the integer row numbers of x that i matches to.
.SDcols	Advanced. Specifies the columns of x included in .SD. May be character column names or numeric positions. This is useful for speed when applying a function through a subset of (possible very many) columns; e.g., DT[,lapply(.SD,sum),by="x,y",.SDcols=301
verbose	TRUE turns on status and information messages to the console. Turn this on by default using options(datatable.verbose=TRUE). The quantity and types of verbosity may be expanded in future.
drop	Never used by data.table. Do not use. It needs to be here because data.table inherits from data.frame. See vignette("datatable-faq").

## Details

data.table builds on base R functionality to reduce 2 types of time :

1. programming time (easier to write, read, debug and maintain)
2. compute time

It combines database like operations such as `subset`, `with` and `by` and provides similar joins that `merge` provides but faster. This is achieved by using R's column based ordered in-memory data.frame structure, eval within the environment of a list, the `[\code{data.table}]` mechanism to condense the features, and compiled C to make certain operations fast.

The package can be used just for rapid programming (compact syntax). Largest compute time benefits are on 64bit platforms with plentiful RAM, or when smaller datasets are repeatedly queried within a loop, or when other methods use so much working memory that they fail with an out of memory error.

As with `[\code{data.frame}]`, *compound queries* can be concatenated on one line; e.g.,

```
DT[,sum(v),by=colA][V1<300][tail(order(V1))]  
# sum(v) by colA then return the 6 largest which are under 300
```

The `j` expression does not have to return data; e.g.,

```
DT[,plot(colB,colC),by=colA]  
# produce a set of plots (likely to pdf) returning no data
```

Multiple data.tables (e.g. X, Y and Z) can be joined in many ways; e.g.,

```
X[Y][Z]  
X[Z][Y]  
X[Y[Z]]  
X[Z[Y]]
```

A data.table is a list of vectors, just like a data.frame. However :

1. it never has rownames. Instead it may have one *key* of one or more columns. This key can be used for row indexing instead of rownames.
2. it has enhanced functionality in [.data.table for fast joins of keyed tables, fast aggregation, and fast last observation carried forward (LOCF).

Since a *list* is a vector, data.table columns may be type *list*. Columns of type *list* can contain mixed types. Each item in a column of type *list* may be different lengths. This is true of data.frame, too.

Several *methods* are provided for data.table, including is.na, na.omit, t, rbind, cbind, merge and others.

### Note

If keep.rownames or check.names are supplied they must be written in full because R does not allow partial argument names after '...'. For example, data.table(DF, keep=TRUE) will create a column called "keep" containing TRUE and this is correct behaviour; data.table(DF, keep.rownames=TRUE) was intended.

POSIXlt is not supported as a column type because it uses 40 bytes to store a single datetime. Unexpected errors may occur if you manage to create a column of type POSIXlt. Please see [NEWS](#) for 1.6.3, and [IDateTime](#) instead. IDateTime has methods to convert to and from POSIXlt.

### References

data.table homepage: <http://datatable.r-forge.r-project.org/>  
 User reviews: <http://crantastic.org/packages/data-table>  
[http://en.wikipedia.org/wiki/Binary\\_search](http://en.wikipedia.org/wiki/Binary_search)  
[http://en.wikipedia.org/wiki/Radix\\_sort](http://en.wikipedia.org/wiki/Radix_sort)

### See Also

[data.frame](#), [\[.data.frame](#), [as.data.table](#), [setkey](#), [J](#), [SJ](#), [CJ](#), [merge.data.table](#), [tables](#), [test.data.table](#), [IDateTime](#), [unique.data.table](#), [copy](#), [:=](#), [alloc.col](#), [truelength](#)

### Examples

```
## Not run:
example(data.table) # to run these examples at the prompt
## End(Not run)

DF = data.frame(x=rep(c("a","b","c"),each=3), y=c(1,3,6), v=1:9)
DT = data.table(x=rep(c("a","b","c"),each=3), y=c(1,3,6), v=1:9)
DF
DT
identical(dim(DT),dim(DF)) # TRUE
identical(DF$a, DT$a)     # TRUE
is.list(DF)               # TRUE
is.list(DT)               # TRUE

is.data.frame(DT)        # TRUE
```

```

tables()

DT[2]           # 2nd row
DT[,v]         # v column (as vector)
DT[,list(v)]   # v column (as data.table)
DT[2:3,sum(v)] # sum(v) over rows 2 and 3
DT[2:5,cat(v,"\\n")] # just for j's side effect
DT[c(FALSE,TRUE)] # even rows (usual recycling)

DT[,2,with=FALSE] # 2nd column
colNum = 2
DT[,colNum,with=FALSE] # same

setkey(DT,x)    # set a 1-column key. No quotes, for convenience.
setkeyv(DT,"x") # same (v in setkeyv stands for vector)
v="x"
setkeyv(DT,v)   # same
# key(DT)<-"x"  # copies whole table, please use set* functions instead

DT["a"]        # binary search (fast)
DT[x=="a"]     # vector scan (slow)

DT[,sum(v),by=x] # keyed by
DT[,sum(v),by=key(DT)] # same
DT[,sum(v),by=y] # ad hoc by

DT["a",sum(v)] # j for one group
DT[c("a","b"),sum(v)] # j for two groups

X = data.table(c("b","c"),foo=c(4,2))
X

DT[X]          # join
DT[X,sum(v)]   # join and eval j for each row in i
DT[X,mult="first"] # first row of each group
DT[X,mult="last"] # last row of each group
DT[X,sum(v)*foo] # join inherited scope

J("a",2)      # J() is alias for data.table()
data.table("a",2) # same

setkey(DT,x,y) # 2-column key
setkeyv(DT,c("x","y")) # same

DT["a"]       # join to 1st column of key
DT[J("a")]    # same
DT[J("a",3)]  # join to 2 columns
DT[J("a",3:6)] # join 4 rows (2 missing)
DT[J("a",3:6),nomatch=0] # remove missing
DT[J("a",3:6),roll=TRUE] # rolling join (locf)

DT[,sum(v),by=list(y%2)] # by expression
DT[,.SD[2],by=x]        # 2nd row of each group

```

```

DT[,tail(.SD,2),by=x]      # last 2 rows of each group
DT[,lapply(.SD,sum),by=x] # applying through columns by group

DT[,list(MySum=sum(v),
        MyMin=min(v),
        MyMax=max(v)),
     by=list(x,y%2)]      # by 2 expressions

DT[,sum(v),x][V1<20]      # compound query
DT[,sum(v),x][order(-V1)] # ordering results

DT[,z:=42L]               # add new column by reference
DT[,z:=NULL]              # remove column
DT["a",v:=42L]            # subassign v by reference

DT[,transform(.SD,m=mean(v)),by=x]
DT[,.SD[which.min(v)],by=x]

# Follow posting guide, support is here (not r-help) :
maintainer("data.table")

## Not run:
vignette("datatable-intro")
vignette("datatable-faq")
vignette("datatable-timings")

test.data.table()        # over 300 low level tests

update.packages()        # keep up to date

## End(Not run)

```

---

data.table-class      *S4 Definition for data.table*

---

## Description

A data.table can be used in S4 class definitions as either a parent class (inside a contains argument of setClass), or as an element of an S4 slot.

## Author(s)

Steve Lianoglou

## See Also

[data.table](#)

**Examples**

```
## Used in inheritance.
setClass('SuperDataTable', contains='data.table')

## Used in a slot
setClass('Something', representation(x='character', dt='data.table'))
x <- new("Something", x='check', dt=data.table(a=1:10, b=11:20))
```

duplicated

*Determine Duplicate Rows***Description**

`duplicated` returns a logical vector indicating which rows of a `data.table` have duplicate rows (by key).

`unique` returns a data table with duplicated rows (by key) removed, or (when no key) duplicated rows by all columns removed.

**Usage**

```
## S3 method for class 'data.table'
duplicated(x, incomparables=FALSE, tolerance=.Machine$double.eps ^ 0.5, ...)

## S3 method for class 'data.table'
unique(x, incomparables=FALSE, tolerance=.Machine$double.eps ^ 0.5, ...)
```

**Arguments**

<code>x</code>	A <code>data.table</code> .
<code>...</code>	Not used at this time.
<code>incomparables</code>	Not used. Here for S3 method consistency.
<code>tolerance</code>	Double precision values are considered equal if they are within this tolerance. Same default as <a href="#">all.equal</a> .

**Details**

Because `data.tables` are usually sorted by key, tests for duplication are especially quick. Unlike [unique.data.frame](#), `paste` is not used to ensure equality of floating point data. This is done directly (for speed) whilst still respecting tolerance in the same spirit as [all.equal](#).

When `x` has a key, only key columns are checked for duplication; non-key columns are not checked. When `x` has no key, all columns are checked.

**Value**

`duplicated` returns a logical vector of length `nrow(x)` indicating which rows are duplicates.

`unique` returns a data table with duplicated rows removed.

**See Also**

[data.table](#), [duplicated](#), [unique](#), [all.equal](#)

**Examples**

```
DT <- data.table(A = rep(1:3, each=4), B = rep(1:4, each=3), C = rep(1:2, 6), key = "A,B")
duplicated(DT)
unique(DT)

DT = data.table(a=c(2L,1L,2L), b=c(1L,2L,1L)) # no key
unique(DT) # rows 1 and 2 (row 3 is a duplicate of row 1)

DT = data.table(a=c(3.142, 4.2, 4.2, 3.142, 1.223, 1.223), b=rep(1,6))
unique(DT) # rows 1,2 and 5

DT = data.table(a=tan(pi*(1/4 + 1:10)), b=rep(1,10)) # example from ?all.equal
length(unique(DT$a)) # 10 strictly unique floating point values
all.equal(DT$a,rep(1,10)) # TRUE, all within tolerance of 1.0
DT[,which.min(a)] # row 10, the strictly smallest floating point value
identical(unique(DT),DT[1]) # TRUE, stable within tolerance
identical(unique(DT),DT[10]) # FALSE
```

---

IDateTime

*Integer based date class*


---

**Description**

Date and time classes with integer storage for fast sorting and grouping. Still experimental!

**Usage**

```
as.IDate(x, ...)
## Default S3 method:
as.IDate(x, ...)
## S3 method for class 'Date'
as.IDate(x, ...)
## S3 method for class 'IDate'
as.Date(x, ...)
## S3 method for class 'IDate'
as.POSIXct(x, tz = "UTC", time = 0, ...)
## S3 method for class 'IDate'
as.chron(x, time = NULL, ...)
## S3 method for class 'IDate'
round(x, digits = c("weeks", "months", "quarters", "years"), ...)

as.ITime(x, ...)
## Default S3 method:
as.ITime(x, ...)
```

```

## S3 method for class 'ITime'
as.POSIXct(x, tz = "UTC", date = as.Date(Sys.time()), ...)
## S3 method for class 'ITime'
as.chron(x, date = NULL, ...)
## S3 method for class 'ITime'
as.character(x, ...)
## S3 method for class 'ITime'
format(x, ...)

IDateTime(x, ...)
## Default S3 method:
IDateTime(x, ...)

hour(x)
yday(x)
wday(x)
mday(x)
week(x)
month(x)
quarter(x)
year(x)

```

### Arguments

<code>x</code>	an object
<code>...</code>	arguments to be passed to or from other methods. For <code>as.IDate.default</code> , arguments are passed to <code>as.Date</code> . For <code>as.ITime.default</code> , arguments are passed to <code>as.POSIXlt</code> .
<code>tz</code>	time zone (see <code>strptime</code> ).
<code>date</code>	date object convertible with <code>as.IDate</code> .
<code>time</code>	time-of-day object convertible with <code>as.ITime</code> .
<code>digits</code>	really units; one of the units listed for rounding. May be abbreviated.

### Details

`IDate` is a date class derived from `Date`. It has the same internal representation as the `Date` class, except the storage mode is integer. `IDate` is a relatively simple wrapper, and it should work in almost all situations as a replacement for `Date`.

Functions that use `Date` objects generally work for `IDate` objects. This package provides specific methods for `IDate` objects for `mean`, `cut`, `seq`, `c`, `rep`, and `split` to return an `IDate` object.

`ITime` is a time-of-day class stored as the integer number of seconds in the day. `as.ITime` does not allow days longer than 24 hours. Because `ITime` is stored in seconds, you can add it to a `POSIXct` object, but you should not add it to a `Date` object.

Conversions to and from `Date`, `POSIXct`, and `chron` formats are provided.

`ITime` does not account for time zones. When converting `ITime` and `IDate` to `POSIXct` with `as.POSIXct`, a time zone may be specified.

In `as.POSIXct` methods for `ITime` and `IDate`, the second argument is required to be `tz` based on the generic template, but to make converting easier, the second argument is interpreted as a date instead of a time zone if it is of type `IDate` or `ITime`. Therefore, you can use either of the following: `as.POSIXct(time, date)` or `as.POSIXct(date, time)`.

`IDateTime` takes a date-time input and returns a data table with columns `date` and `time`.

Using integer storage allows dates and/or times to be used as data table keys. With positive integers with a range less than 100,000, grouping and sorting is fast because radix sorting can be used (see `sort.list`).

Several convenience functions like `hour` and `quarter` are provided to group or extract by hour, month, and other date-time intervals. `as.POSIXlt` is also useful. For example, `as.POSIXlt(x)$mon` is the integer month. The R base convenience functions `weekdays`, `months`, and `quarters` can also be used, but these return character values, so they must be converted to factors for use with `data.table`.

The `round` method for `IDate`'s is useful for grouping and plotting. It can round to weeks, months, quarters, and years.

## Value

For `as.IDate`, a class of `IDate` and `Date` with the date stored as the number of days since some origin.

For `as.ITime`, a class of `ITime` stored as the number of seconds in the day.

For `IDateTime`, a data table with columns `idate` and `itime` in `IDate` and `ITime` format.

`hour`, `codeyday`, `wday`, `mday`, `week`, `month`, `quarter`, and `year` return integer values for hour, day of year, day of week, day of month, week, month, quarter, and year.

## Author(s)

Tom Short, [t.short@ieee.org](mailto:t.short@ieee.org)

## References

- G. Grothendieck and T. Petzoldt, "Date and Time Classes in R," R News, vol. 4, no. 1, June 2004.
- H. Wickham, <http://gist.github.com/10238>.

## See Also

[as.Date](#), [as.POSIXct](#), [strptime](#), [DateTimeClasses](#)

## Examples

```
# create IDate:
(d <- as.IDate("2001-01-01"))

# S4 coercion also works
identical(as.IDate("2001-01-01"), as("2001-01-01", "IDate"))

# create ITime:
```

```

(t <- as.ITime("10:45"))

# S4 coercion also works
identical(as.ITime("10:45"), as("10:45", "ITime"))

(t <- as.ITime("10:45:04"))

(t <- as.ITime("10:45:04", format = "%H:%M:%S"))

as.POSIXct("2001-01-01") + as.ITime("10:45")

datetime <- seq(as.POSIXct("2001-01-01"), as.POSIXct("2001-01-03"), by = "5 hour")
(af <- data.table(IDateTime(datetime), a = rep(1:2, 5), key = "a, idate, itime"))

af[, mean(a), by = "itime"]
af[, mean(a), by = list(hour = hour(itime))]
af[, mean(a), by = list(wday = factor(weekdays(idate)))]
af[, mean(a), by = list(wday = wday(idate))]

as.POSIXct(af$idate)
as.POSIXct(af$idate, time = af$itime)
as.POSIXct(af$idate, af$itime)
as.POSIXct(af$idate, time = af$itime, tz = "GMT")

as.POSIXct(af$itime, af$idate)
as.POSIXct(af$itime) # uses today's date

(seqdates <- seq(as.IDate("2001-01-01"), as.IDate("2001-08-03"), by = "3 weeks"))
round(seqdates, "months")

if (require(chron)) {
  as.chron(as.IDate("2000-01-01"))
  as.chron(as.ITime("10:45"))
  as.chron(as.IDate("2000-01-01"), as.ITime("10:45"))
  as.chron(as.ITime("10:45"), as.IDate("2000-01-01"))
  as.ITime(chron(times = "11:01:01"))
  IDateTime(chron("12/31/98", "10:45:00"))
}

```

---

J

*Creates a Join data table*


---

### Description

Creates a `data.table` to be passed in as the `i` to a `[.data.table]` join.

### Usage

```
J(..., SORTFIRST=FALSE)
```

```
SJ(...)
CJ(...)
```

### Arguments

... Each argument is a vector. Generally each vector is the same length but if they are not then usual silent repetition is applied.

SORTFIRST Internal argument. SJ sets this to TRUE

### Details

J, SJ and CJ are convenience functions for creating a `data.table` in the context of a `data.table` 'query' on `x`. `x[data.table(id)]` is the same as `x[J(id)]` but the latter is more readable. `x` must have a key when passing in a join table as the `i`. See [[.data.table](#)]

### Value

J : the same result as calling `data.table`. J is a direct alias for `data.table` but results in clearer more readable code. SJ : (S)orted (J)oin. The same value as J() but additionally `setkey()` is called on all the columns in the order they were passed in to SJ. For efficiency. CJ : (C)ross (J)oin. A `data.table` is formed from the cross product of the vectors. For example, 10 ids, and 100 dates, CJ returns a 1000 row table containing all the dates for all the ids.

### See Also

[data.table](#), [test.data.table](#)

### Examples

```
DT = data.table(A=5:1,B=letters[5:1])
setkey(DT,B) # re-orders table and marks it sorted.
DT[J("b")] # returns the 2nd row
```

---

last

*Last item of an object*

---

### Description

Returns last item of a vector, or last row of a matrix, `data.frame` or `data.table`.

### Usage

```
last(x)
```

### Arguments

x A vector, matrix, `data.frame` or `data.table`

**Value**

The last item. If `x` is a `data.table`, the last row is returned at a one row `data.table`.

**See Also**

[NROW](#)

---

like	<i>Convenience function for calling <code>regexpr</code>.</i>
------	---------------------------------------------------------------

---

**Description**

Intended for use in `[.data.table i]`.

**Usage**

```
like(vector, pattern)
vector
```

**Arguments**

vector	Either a character vector or a factor. A factor is faster.
pattern	Passed on to <a href="#">grepl</a> .

**Value**

Logical vector, TRUE for items that match pattern.

**Note**

Current implementation does not make use of sorted keys.

**See Also**

[data.table](#), [grepl](#)

**Examples**

```
DT = data.table(Name=c("Mary", "George", "Martha"), Salary=c(2, 3, 4))
DT[Name %like% "^Mar"]
```

merge

*Merge Two Data Tables***Description**

Relatively quick merge of data tables based on common keys (by default).

This function is meant to act very similarly to the `merge.data.frame` function, with the major exception being that the default columns used to merge two data.tables are the shared key columns, and not the shared columns with the same names.

For a more data.table-centric (and faster) way of merging two data.tables, take a look at [`.data.table`; e.g., `x[y, ...]`].

**Usage**

```
## S3 method for class 'data.table'
merge(x, y, by = NULL, all = FALSE, all.x = all, all.y = all, suffixes = c(".x", ".y"), ...)
```

**Arguments**

<code>x, y</code>	data tables. <code>y</code> is coerced to a data.table if it isn't one already
<code>by</code>	A vector of shared column names in <code>x</code> and <code>y</code> to merge on. This defaults to the shared key columns between the two tables. If <code>y</code> has no key columns, this defaults to the keys set for <code>x</code> . Note that if the specified values in <code>by</code> are not the keys (or prefixes of keys) for <code>x, y</code> , then they are first set as the keys prior to performing the merge – this might make this function perform slower than you are expecting.
<code>all</code>	logical; <code>all = L</code> is shorthand for <code>all.x = L</code> and <code>all.y = L</code> .
<code>all.x</code>	logical; if TRUE, then extra rows will be added to the output, one for each row in <code>x</code> that has no matching row in <code>y</code> . These rows will have 'NA's in those columns that are usually filled with values from <code>y</code> . The default is FALSE, so that only rows with data from both <code>x</code> and <code>y</code> are included in the output.
<code>all.y</code>	logical; analogous to <code>all.x</code> above.
<code>suffixes</code>	A character(2) specifying the suffixes to be used for making non-by column names unique. The suffix behavior works in a similar fashion as <code>merge.data.frame</code> does.
<code>...</code>	Not used at this time.

**Details**

Keys for each data.table are reshuffled to ensure that the columns identified in the `by` parameter are prefixes of the keys set for data.tables `x` and `y` – this may cause the function to run slower than expected.

**Value**

A new `data.table` based on the merged data tables, sorted by the columns set (or inferred for) the `by` argument.

**See Also**

[data.table](#), [\[.data.table](#), [merge.data.frame](#)

**Examples**

```
(dt1 <- data.table(A = letters[1:10], X = 1:10, key = "A"))
(dt2 <- data.table(A = letters[5:14], Y = 1:10, key = "A"))
merge(dt1, dt2)
merge(dt1, dt2, all = TRUE)

(dt1 <- data.table(A = letters[rep(1:3, 2)], X = 1:6, key = "A"))
(dt2 <- data.table(A = letters[rep(2:4, 2)], Y = 6:1, key = "A"))
merge(dt1, dt2)

(dt1 <- data.table(A = c(rep(1L, 5), 2L), B = letters[rep(1:3, 2)], X = 1:6, key = "A,B"))
(dt2 <- data.table(A = c(rep(1L, 5), 2L), B = letters[rep(2:4, 2)], Y = 6:1, key = "A,B"))
merge(dt1, dt2)
merge(dt1, dt2, by="B")

# test it more:
d1 <- data.table(a=rep(1:2,each=3), b=1:6, key="a,b")
d2 <- J(a=0:1, bb=10:11, key="a")
d3 <- J(a=0:1, key="a")
d4 <- J(a=0:1, b=0:1, key="a,b")

merge(d1, d2)
merge(d2, d1)
merge(d1, d2, all=TRUE)
merge(d2, d1, all=TRUE)

merge(d3, d1)
merge(d1, d3)
merge(d1, d3, all=TRUE)
merge(d3, d1, all=TRUE)

merge(d1, d4)
merge(d1, d4, by="a", suffixes=c(".d1", ".d4"))
merge(d4, d1)
merge(d1, d4, all=TRUE)
merge(d4, d1, all=TRUE)
```

---

 setkey

*Create key on a data table*


---

### Description

Sorts a `data.table` and marks it as sorted. The sorted columns are the key. The key can be any columns in any order. The columns are sorted in ascending order always. The table is changed *by reference*. No copy is made at all, other than temporary working memory as large as one column.

### Usage

```
setkey(x, ..., verbose=getOption("datatable.verbose"))
setkeyv(x, cols, verbose=getOption("datatable.verbose"))
key(x)
haskey(x)
copy(x)
setattr(x,name,value)
setnames(x,old,new)
setcolorder(x,neworder)
set(x,i,j,value) # should be documented in "?":=" , perhaps.
key(x) <- value # deprecated, please use setkey or setkeyv instead.
```

### Arguments

<code>x</code>	A <code>data.table</code> .
<code>...</code>	The columns to sort by. Do not quote the column names. If <code>...</code> is missing (i.e. <code>setkey(DT)</code> ), all the columns are used.
<code>cols</code>	A character vector (only) of column names.
<code>value</code>	In (deprecated) <code>key&lt;-</code> , a character vector (only) of column names. In <code>setattr</code> , the value to assign to the attribute or <code>NULL</code> removes the attribute, if present.
<code>name</code>	The character attribute name.
<code>verbose</code>	Output status and information.
<code>old</code>	When <code>new</code> is provided, character names or numeric positions of column names to change. When <code>new</code> is not provided, the new column names, which must be the same length as the number of columns. See examples.
<code>new</code>	Optional. New column names, the same length as <code>old</code> .
<code>neworder</code>	Character vector of the new column name ordering. May also be column numbers.
<code>i</code>	Integer row numbers to be assigned value.
<code>j</code>	Integer column number to be assigned value.

## Details

The sort is attempted with the very fast "radix" method in `sort.list`. If that fails, the sort reverts to the default method in `order`. That logic is repeated column by column.

The sort is *stable*; i.e., the order of ties (if any) is preserved.

If `v=NULL`, the key is removed.

In v1.7.8, the `key<-` syntax was deprecated. The `<-` method copies the whole table and we know of no way to avoid that copy without a change in `R` itself. Please use the `set*` functions instead, which make no copy at all. `setkey` accepts unquoted column names for convenience, whilst `setkeyv` accepts one vector of column names.

The problem (for `data.table`) with the copy by `key<-` (other than being slower) is that `R` doesn't maintain the over allocated `truelength`, but it looks as though it has. Adding a column by reference using `:=` after a `key<-` was therefore a memory overwrite and eventually a seg fault; the over allocated memory wasn't really there after `key<-`'s copy. `data.tables` have a new attribute `.internal.selfref` to catch and warn about such copies in future. This attribute has been implemented in way that is friendly with `identical()` and `object.size()`.

For the same reason, please use `setattr()` rather than `attr(x,name)<-value`, `setnames()` rather than `names(x)<-value` or `colnames(x)<-value`, and `setcolorder()` rather than `DT<-DT[,neworder,with=FALSE]`.

It isn't good programming practice, in general, to use column numbers rather than names. This is why `setkey` and `setkeyv` only accept column names, and why `old` in `setnames()` is recommended to be names. If you use column numbers then bugs (possibly silent) can more easily creep into your code as time progresses if changes are made elsewhere in your code; e.g., if you add, remove or reorder columns in a few months time, a `setkey` by column number will then refer to a different column, possibly returning incorrect results with no warning. (A similar concept exists in `SQL`, where "select \* from ..." is considered poor programming style when a robust, maintainable system is required.) If you wish to use column numbers, it's possible but a little harder; e.g., `setkeyv(DT,colnames(DT)[1:2])`.

## Value

The `data.table` is modified by reference, and returned (invisibly) so it can be used in compound statements; e.g., `setkey(DT,a)[J("foo")]`. If you require a copy, take a copy first (using `DT2=copy(DT)`). `copy()` may also sometimes be useful before `:=` is used to subassign to a column by reference. See `?copy`.

## Note

`base::sort.list(x,method="radix")` actually invokes a *counting sort*, not a radix sort. See `do_radixsort` in `src/main/sort.c`. A counting sort, however, is particularly suitable for sorting integers and factors, and we like it. Anyway, this is one reason `data.table` 'likes' integers and factors.

## References

[http://en.wikipedia.org/wiki/Radix\\_sort](http://en.wikipedia.org/wiki/Radix_sort)  
[http://en.wikipedia.org/wiki/Counting\\_sort](http://en.wikipedia.org/wiki/Counting_sort)

**See Also**

[data.table](#), [tables](#), [J](#), [sort.list](#), [copy](#), [:=](#)

**Examples**

```
# Type 'example(setkey)' to run these at prompt and browse output

DT = data.table(A=5:1,B=letters[5:1])
DT # before
setkey(DT,B)           # re-orders table and marks it sorted.
DT # after
tables()               # KEY column reports the key'd columns
key(DT)
keycols = c("A","B")
setkeyv(DT,keycols)   # rather than key(DT)<-keycols (which copies entire table)

DT = data.table(A=5:1,B=letters[5:1])
DT2 = DT               # does not copy
setkey(DT2,B)         # does not copy-on-write to DT2
identical(DT,DT2)     # TRUE. DT and DT2 are two names for the same keyed table

DT = data.table(A=5:1,B=letters[5:1])
DT2 = copy(DT)        # explicit copy() needed to copy a data.table
setkey(DT2,B)         # now just changes DT2
identical(DT,DT2)     # FALSE. DT and DT2 are now different tables

DF = data.frame(a=1:2,b=3:4)      # base data.frame to demo copies
try(tracemem(DF))                 # try() for non-Windows where R is faster without memory profiling
colnames(DF)[1] <- "A"           # 4 copies of entire object
names(DF)[1] <- "A"              # 3 copies of entire object
'names<-'(DF,c("A","b"))        # 1 copy of entire object
x='names<-'(DF,c("A","b"))       # still 1 copy (so not print method)
# What if DF is large, say 10GB in RAM. Copy 10GB just to change a column name?

DT = data.table(a=1:2,b=3:4,c=5:6)
try(tracemem(DT))
setnames(DT,"b","B")             # by name; no match() needed
setnames(DT,3,"C")               # by position
setnames(DT,2:3,c("D","E"))     # multiple
setnames(DT,c("a","E"),c("A","F")) # multiple by name
setnames(DT,c("X","Y","Z"))     # replace all
# And, no copy of DT was made by setnames() at all
```

---

subset.data.table

*Subsetting data.tables*

---

**Description**

Retruns subsets of a data.table.

**Usage**

```
## S3 method for class 'data.table'
subset(x, subset, select, ...)
```

**Arguments**

x	data.table to subset.
subset	logical expression indicating elements or rows to keep
select	expression indicating columns to select from data.table
...	further arguments to be passed to or from other methods

**Details**

The subset argument works on the rows and will be evaluated in the data.table so columns can be referred to (by name) as variables in the expression.

The data.table that is returned will maintain the original keys as long as they are not select-ed out.

**Value**

A data.table containing the subset of rows and columns that are selected.

**See Also**

[subset](#)

**Examples**

```
dt <- data.table(a=sample(c('a', 'b', 'c'), 20, replace=TRUE),
                b=sample(c('a', 'b', 'c'), 20, replace=TRUE),
                c=sample(20, key=c('a', 'b')))

sub <- subset(dt, a == 'a')
all.equal(key(sub), key(dt))
```

---

tables

*Display all objects of class 'data.table'*

---

**Description**

Lists all data.table's in memory, including number of rows, column names and any keys.

**Usage**

```
tables(mb = TRUE, order.col = "NAME", width = 80, env=parent.frame(), silent=FALSE)
```

**Arguments**

mb	TRUE adds size of the data.table in MB to the output (slow in older versions of R).
order.col	Quoted column name to sort the output by
width	Number of characters to truncate the COLS output
env	Usually tables() is executed at the prompt where parent.frame() returns .GlobalEnv. tables() may also be useful inside functions where parent.frame() is the local scope of the function, or set it to .GlobalEnv
silent	By default tables() is expected to be called at the prompt for its compact print output. silent=TRUE prints nothing. The data statistics are returned as a data.table, silently, whether silent is TRUE or FALSE

**Value**

A data.table containing the information printed.

**See Also**

[data.table](#), [setkey](#), [ls](#), [objects](#), [object.size](#)

**Examples**

```
DT = data.table(A=1:10,B=letters[1:10])
DT2 = data.table(A=1:10000,ColB=10000:1)
setkey(DT,B)
tables()
```

---

test.data.table	<i>Runs a set of tests.</i>
-----------------	-----------------------------

---

**Description**

Runs a set of tests to check data.table is working correctly.

**Usage**

```
test.data.table(echo=FALSE)
```

**Arguments**

echo	Displays each test on the console as it runs.
------	-----------------------------------------------

**Details**

Runs a series of tests. These can be used to see features and examples of usage, too. Running test.data.table will tell you the full location of the test file(s) to open.

**Value**

TRUE if all tests were successful. FALSE otherwise.

**See Also**

[data.table](#)

**Examples**

```
## Not run:  
test.data.table()  
  
## End(Not run)
```

---

timetaken	<i>Pretty print of time taken</i>
-----------	-----------------------------------

---

**Description**

Pretty print of time taken since last started.at.

**Usage**

```
timetaken(started.at)
```

**Arguments**

started.at      The result of `proc.time()` taken some time earlier.

**Value**

A character vector of the form hh:mm:ss, or ss.mmm if under 60 seconds.

**Examples**

```
started.at=proc.time()  
Sys.sleep(1)  
cat("Finished in",timetaken(started.at),"\n")
```

---

transform.data.table *Data table utilities*

---

### Description

Utilities for data.table transformation.

### Usage

```
## S3 method for class 'data.table'  
transform('_data', ...)  
## S3 method for class 'data.table'  
within(data, expr, ...)
```

### Arguments

data, _data	data.table to be transformed.
...	for transform, Further arguments of the form tag=value. Ignored for within.
expr	expression to be evaluated within the data.table.

### Details

within is like with, but modifications (columns changed, added, or removed) are updated in the returned data.table.

Note that transform will keep the key of the data.table provided the “targets” of the transform (i.e. the columns that appear in ...) are not in the key of the data.table. within also retains the key provided the key columns are not “touched”.

### Value

The modified value of data.

### See Also

[transform](#) and [within](#)

### Examples

```
dt <- data.table(a=rep(1:5, 1), b=1:10)  
  
transform(dt, c = a^2)  
  
#within(dt, {  
# b <- rev(b)  
# c <- a^2  
# rm(a)  
#})
```

```
# dt[, transform, c = max(b), by="a"] # like "ave"
```

---

truelength	<i>Over-allocation access</i>
------------	-------------------------------

---

## Description

These functions are experimental and somewhat advanced. By *experimental* we mean their names might change and perhaps the syntax, argument names and types (so if you write a lot of code using them, you have been warned!). They should work and be stable, though, so please report problems with them.

## Usage

```
truelength(x)
alloc.col(DT,
  n = getOption("datatable.alloccol"), # default: quote(max(100,2*ncol(DT)))
  verbose = getOption("datatable.verbose")) # default: FALSE
```

## Arguments

x	Any type of vector, including <code>data.table</code> which is a list vector of column pointers.
DT	A <code>data.table</code> .
n	The number of column pointer slots to reserve in memory, including existing columns. May be a numeric, or a <code>quote()</code> -ed expression (see default). If DT is a 10 column <code>data.table</code> , <code>n=1000</code> means grow the spare slots from 90 to 990, assuming the default of 100 has not been changed.
verbose	Output status and information.

## Details

When adding columns by reference using `:=`, we *could* simply create a new column list vector (one longer) and `memcpy` over the old vector, with no copy of the column vectors themselves. That requires negligible use of space and time, and is what v1.7.2 did. However, that copy of the list vector of column pointers only (but not the columns themselves), a *shallow copy*, resulted in inconsistent behaviour in some circumstances. So, as from v1.7.3 `data.table` over allocates the list vector of column pointers so that columns can be added fully by reference, consistently.

When the allocated column pointer slots are used up, to add a new column `data.table` must reallocate that vector. If two or more variables are bound to the same `data.table` this shallow copy may or may not be desirable, but we don't think this will be a problem very often (more discussion may be required on `datatable-help`). Setting `options(datatable.verbose=TRUE)` includes messages if and when a shallow copy is taken. To avoid shallow copies there are several options: use [copy](#) to make a deep copy first, use `alloc.col` to reallocate in advance, or, change the default allocation rule (perhaps in your `.Rprofile`); e.g., `options(datatable.alloccol=1000)`.

Please note : over allocation of the column pointer vector is not for efficiency per se, it's so that `:=` can add columns by reference without a shallow copy.

**Value**

`truelength(x)` returns the length of the vector allocated in memory. `length(x)` of those items are in use. Currently, it's just the list vector of column pointers that is over-allocated (i.e. `truelength(DT)`), not the column vectors themselves, which would in future allow fast row `insert()`. For tables loaded from disk however, `truelength` is 0 in R 2.14.0 and random in R <= 2.13.2; i.e., in both cases perhaps unexpected. `data.table` detects this state and over-allocates the loaded `data.table` when the next column addition or deletion occurs. All other operations on `data.table` (such as fast grouping and joins) do not need `truelength`.

`alloc.col` *reallocates* `DT` by reference. This may be useful for efficiency if you know you are about to going to add a lot of columns in a loop. It also returns the new `DT`, for convenience in compound queries.

**See Also**

[copy](#)

**Examples**

```
DT = data.table(a=1:3,b=4:6)
length(DT)           # 2 column pointer slots used
truelength(DT)       # 100 column pointer slots allocated
alloc.col(DT,200)
length(DT)           # 2 used
truelength(DT)       # 200 allocated
DT[,c:=7L]           # add new column
truelength(DT)-length(DT) # 197 slots spare
```

# Index

- \*Topic **chron**
  - IDateTime, 16
- \*Topic **classes**
  - data.table-class, 14
- \*Topic **data**
  - :=, 2
  - between, 5
  - chmatch, 6
  - data.table, 8
  - duplicated, 15
  - J, 19
  - last, 20
  - like, 21
  - merge, 22
  - setkey, 24
  - subset.data.table, 26
  - tables, 27
  - test.data.table, 28
  - timetaken, 29
  - transform.data.table, 30
  - truelength, 31
- \*Topic **methods**
  - data.table-class, 14
- \*Topic **utilities**
  - IDateTime, 16
  - :=, 2, 8, 12, 26
  - [.data.frame, 9, 12
  - [.data.table, 20, 22, 23
  - [.data.table(data.table), 8
  - %between%(between), 5
  - %chin%(chmatch), 6
  - %like%(like), 21
  - %in%, 7
  
  - all.equal, 4, 15, 16
  - all.equal.list, 4, 5
  - alloc.col, 3, 12
  - alloc.col(truelength), 31
  - as.character.ITime(IDateTime), 16
  - as.chron.IDate(IDateTime), 16
  - as.chron.ITime(IDateTime), 16
  - as.data.table, 12
  - as.data.table(data.table), 8
  - as.Date, 18
  - as.Date.IDate(IDateTime), 16
  - as.IDate(IDateTime), 16
  - as.ITime(IDateTime), 16
  - as.list.IDate(IDateTime), 16
  - as.POSIXct, 18
  - as.POSIXct.IDate(IDateTime), 16
  - as.POSIXct.ITime(IDateTime), 16
  - as.POSIXlt.ITime(IDateTime), 16
  
  - between, 5
  - by, 11
  
  - c.IDate(IDateTime), 16
  - charmatch, 7
  - chgroup(chmatch), 6
  - chmatch, 6
  - chorder(chmatch), 6
  - CJ, 11, 12
  - CJ(J), 19
  - class:data.table(data.table-class), 14
  - copy, 3, 12, 26, 31, 32
  - copy(setkey), 24
  - cut.IDate(IDateTime), 16
  
  - data.frame, 8, 9, 12
  - data.table, 3, 6, 8, 14, 16, 20, 21, 23, 26, 28, 29
  - data.table-class, 14
  - DateTimeClasses, 18
  - duplicated, 15, 16
  
  - fmatch, 7
  - format.ITime(IDateTime), 16
  
  - grep1, 21
  
  - haskey(setkey), 24

hour (IDateTime), 16  
 IDate (IDateTime), 16  
 IDate-class (IDateTime), 16  
 IDate**Time**, 12, 16  
 is.data.table (data.table), 8  
 is.na.data.table (data.table), 8  
 ITime (IDateTime), 16  
 ITime-class (IDateTime), 16  
  
 J, 12, 19, 26  
  
 key (setkey), 24  
 key<- (setkey), 24  
  
 last, 20  
 like, 6, 21  
 ls, 28  
  
 match, 7, 10  
 mday (IDateTime), 16  
 mean.IDate (IDateTime), 16  
 merge, 11, 22  
 merge.data.frame, 22, 23  
 merge.data.table, 12  
 month (IDateTime), 16  
  
 na.omit.data.table (data.table), 8  
 NROW, 21  
  
 object.size, 28  
 objects, 28  
 Ops.data.table (data.table), 8  
 order, 25  
  
 print.ITime (IDateTime), 16  
  
 quarter (IDateTime), 16  
  
 rep.IDate (IDateTime), 16  
 rep.ITime (IDateTime), 16  
 round.IDate (IDateTime), 16  
  
 seq.IDate (IDateTime), 16  
 set (setkey), 24  
 setattr (setkey), 24  
 setcolororder (setkey), 24  
 setkey, 6, 9, 12, 24, 28  
 setkeyv (setkey), 24  
 setnames (setkey), 24  
 SJ, 12  
  
 SJ (J), 19  
 sort.list, 25, 26  
 split.IDate (IDateTime), 16  
 strptime, 18  
 subset, 11, 27  
 subset.data.table, 26  
  
 tables, 12, 26, 27  
 test.data.table, 12, 20, 28  
 timetaken, 29  
 transform, 30  
 transform.data.table, 30  
 truelength, 3, 12, 31  
  
 unique, 16  
 unique.data.frame, 15  
 unique.data.table, 12  
 unique.data.table (duplicated), 15  
  
 wday (IDateTime), 16  
 week (IDateTime), 16  
 with, 11  
 within, 30  
 within.data.table  
     (transform.data.table), 30  
  
 yday (IDateTime), 16  
 year (IDateTime), 16