General form: `DT[i, j, by]` ⟶ ⟶ "Take DT, subset rows using `i`, then calculate `j` grouped by `by`"

CREATE A DATA TABLE

Create a `data.table` and call it **DT**.

```r
library(data.table)
set.seed(45L)
DT <- data.table(V1=c(1L,2L),
                 V2=LETTERS[1:3],
                 V3=round(rnorm(4),4),
                 V4=1:12)
```

```
> DT
    V1 V2      V3 V4
 1:  1  A -1.1727  1
 2:  2  B -0.3825  2
 3:  1  C -1.0604  3
 4:  2  A  0.6651  4
 5:  1  B -1.1727  5
 6:  2  C -0.3825  6
 7:  1  A -1.0604  7
 8:  2  B  0.6651  8
 9:  1  C -1.1727  9
10:  2  A -0.3825 10
11:  1  B -1.0604 11
12:  2  C  0.6651 12
```

SUBSETTING ROWS USING `i`

What?	*Example*	*Notes*	*Output*

Subsetting rows by numbers.	`DT[3:5,] #or DT[3:5]`	Selects third to fifth row.	`   V1 V2      V3 V4`
			`1:  1  C -1.0604  3`
			`2:  2  A  0.6651  4`
			`3:  1  B -1.1727  5`

Use column names to select rows in **`i`** based on a condition using fast automatic indexing. Or for selecting on multiple values: `DT[column %in% c("value1","value2")]`, which selects all rows that have **value1** or **value2** in **column**.	`DT[ V2 == "A"]`	Selects all rows that have value **A** in column **V2**.	`   V1 V2      V3 V4`
			`1:  1  A -1.1727  1`
			`2:  2  A  0.6651  4`
			`3:  1  A -1.0604  7`
			`4:  2  A -0.3825 10`

	`DT[ V2 %in% c("A","C")]`	Select all rows that have the value **A** or **C** in column **V2**.	`   V1 V2      V3 V4`
			`1:  1  A -1.1727  1`
			`2:  1  C -1.0604  3`
			`                 ...`
			`7:  2  A -0.3825 10`
			`8:  2  C  0.6651 12`

MANIPULATING ON COLUMNS IN J

What?	*Example*	*Notes*	*Output*

Select 1 column in `j`.	`DT[,V2]`	Column **V2** is returned as a vector.	`[1] "A" "B" "C" "A"`
			`"B" "C" ...`

Select several columns in `j`.	`DT[,.(V2,V3)]`	Columns **V2** and **V3** are returned as a `data.table`.	`   V2      V3`
			`1: A -1.1727`
			`2: B -0.3825`
			`3: C -1.0604`
			`…`

`.()` is an alias to `list()`. If `.()` is used, the returned value is a `data.table`. If `.()` is not used, the result is a vector.			
Call functions in `j`.	`DT[,sum(V1)]`	Returns the sum of all elements of column **V1** in a vector.	`[1] 18`

Computing on several columns.	`DT[,.(sum(V1),sd(V3))]`	Returns the sum of all elements of column **V1** and the standard deviation of **V3** in a `data.table`.	`   V1        V2`
			`1: 18 0.7634655`

Assigning column names to computed columns.	`DT[,.(Aggregate = sum(V1),`
	`Sd.V3 = sd(V3))]`
		The same as above, but with new names.	`   Aggregate     Sd.V3`
			`1:        18 0.7634655`

Columns get recycled if different length.	`DT[,.(V1, Sd.V3 = sd(V3))]`	Selects column **V1**, and compute std. dev. of **V3**, which returns a single value and gets recycled.	`    V1     Sd.V3`
			` 1:  1 0.7634655`
			` 2:  2 0.7634655`
			`...`
			`11:  1 0.7634655`
			`12:  2 0.7634655`

Multiple expressions can be wrapped in curly braces.	`DT[,{print(V2)`
	`     plot(V3)`
	`     NULL}]`
		`Print` column **V2** and `plot` **V3**.	`[1] "A" "B" "C" "A"`
			`"B" "C" ...`
			`#And a plot`

DOING j BY GROUP

What?	*Example*	*Notes*	*Output*

Doing `j` **by** group.	`DT[,.(V4.Sum = sum(V4)),by=V1]`	Calculates the sum of **V4**, for every group in **V1**.	`   V1 V4.Sum`
			`1:  1     36`

Doing `j` **by** several groups using `.()`.	`DT[,.(V4.Sum = sum(V4)),by=.(V1,V2)]`	The same as above, but for every group in **V1** <u>and</u> **V2**.	`   V1 V2 V4.Sum`
			`1:  1  A      8`
			`2:  2  B     10`
			`3:  1  C     12`
			`4:  2  A     14`
			`5:  1  B     16`
			`6:  2  C     18`

Call functions in **by**.	`DT[,.(V4.Sum = sum(V4)),by=sign(V1-1)]`	Calculates the sum of **V4**, for every group in **`sign(V1-1)`**.	`   sign V4.Sum`
			`1:    0     36`
			`2:    1     42`

Assigning new column names in **by**.	`DT[,.(V4.Sum = sum(V4)),`
	`        by=.(V1.01 = sign(V1-1))]`
		Same as above, but with a new name for the variable we are grouping by.	`   V1.01 V4.Sum`
			`1:     0     36`
			`2:     1     42`

Grouping only on a subset by specifying `i`.	`DT[1:5,.(V4.Sum = sum(V4)),by=V1]`	Calculates the sum of **V4**, for every group in **V1**, after subsetting on the first five rows.	`   V1 V4.Sum`
			`1:  1      9`
			`2:  2      6`

Using `.N` to get the total number of observations of each group.	`DT[,.N,by=V1]`	Count the number of rows for every group in **V1**.	`   V1 N`
			`1:  1 6`
			`2:  2 6`

ADDING/UPDATING COLUMNS BY REFERENCE IN USING :=

What?	*Example*	*Notes*	*Output*
Adding/updating a column by reference using `:=` in one line. Watch out: extra assignment (`DT <- DT[...]`) is redundant.	`DT[, V1 := round(exp(V1),2)]`	Column **V1** is updated by what is after `:=`.	Returns the result invisibly. Column **V1** went from: `[1] 1 2 1 2 …` to `[1] 2.72 7.39 2.72 7.39 …`
Adding/updating several columns by reference using `:=`.	`DT[, c("V1","V2") := list (round(exp(V1),2), LETTERS [4:6])]`	Column **V1** and **V2** are updated by what is after `:=`.	Returns the result invisibly. Column **V1** changed as above. Column **V2** went from: `[1] "A" "B" "C" "A" "B" "C"` … to: `[1] "D" "E" "F" "D" "E" "F"` …
Using functional `:=`.	`DT[, ':=' (V1 = round(exp(V1),2), V2 = LETTERS[4:6])][]`	Another way to write the same line as above this one, but easier to write comments side-by-side. Also, when `[]` is added the result is printed to the screen.	Same changes as line above this one, but the result is printed to the screen because of the `[]` at the end of the statement.
Remove a column instantly using `:=`.	`DT[, V1 := NULL]`	Removes column **V1**.	Returns the result invisibly. Column **V1** became **`NULL`**.
Remove several columns instantly using `:=`.	`DT[, c("V1","V2") := NULL]`	Removes columns **V1** and **V2**.	Returns the result invisibly. Column **V1** and **V2** became **`NULL`**.
Wrap the name of a variable which contains column names in parenthesis to pass the contents of that variable to be deleted.	`Cols.chosen = c("A","B")`		
	`DT[, Cols.chosen := NULL]`	Watch out: this deletes the column with column name `Cols.chosen`.	Returns the result invisibly. Column with name `Cols.chosen` became **`NULL`**.
	`DT[, (Cols.chosen) := NULL]`	Deletes the columns specified in the variable `Cols.chosen` (**V1** and **V2**).	Returns the result invisibly. Columns **V1** and **V2** became **`NULL`**.

INDEXING AND KEYS

What?	*Example*	*Notes*	*Output*
Use `setkey()` to set a key on a **DT**. The data is sorted on the column we specified by reference.	`setkey(DT,V2)`	A key is set on column **V2**.	Returns results invisibly.

Use keys like supercharged rownames to select rows.	`DT["A"]`	Returns all the rows where the key column (set to column **V2** in the line above) has the value **A**.	`   V1 V2      V3 V4`
			`1:  1  A -1.1727  1`
			`2:  2  A  0.6651  4`
			`3:  1  A -1.0604  7`
			`4:  2  A -0.3825 10`

	`DT[c("A","C")]`	Returns all the rows where the key column (**V2**) has the value **A** or **C**.	`   V1 V2      V3 V4`
			`1:  1  A -1.1727  1`
			`2:  2  A  0.6651  4`
			`                 ...`
			`7:  1  C -1.1727  9`
			`8:  2  C  0.6651 12`

The `mult` argument is used to control which row that **i** matches to is returned, default is all.	`DT["A", mult ="first"]`	Returns first row of all rows that match the value **A** in the key column (**V2**).	`   V1 V2      V3 V4`
			`1:  1  A -1.1727  1`

	`DT["A", mult = "last"]`	Returns last row of all rows that match the value **A** in the key column (**V2**).	`   V1 V2      V3 V4`
			`1:  2  A -0.3825 10`

The `nomatch` argument is used to control what happens when a value specified in **i** has no match in the rows of the **DT**. Default is NA, but can be changed to 0.
0 means no rows will be returned for that non-matched row of **i**.
	`DT[c("A","D")]`	Returns all the rows where the key column (**V2**) has the value **A** or **D**. **A** is found, **D** is not so **NA** is returned for **D**.	`   V1 V2      V3 V4`
			`1:  1  A -1.1727  1`
			`2:  2  A  0.6651  4`
			`3:  1  A -1.0604  7`
			`4:  2  A -0.3825 10`
			`5: NA  D      NA NA`

	`DT[c("A","D"), nomatch = 0]`	Returns all the rows where the key column (**V2**) has the value **A** or **D**. Value **D** is not found and not returned because of the `nomatch` argument.	`   V1 V2      V3 V4`
			`1:  1  A -1.1727  1`
			`2:  2  A  0.6651  4`
			`3:  1  A -1.0604  7`
			`4:  2  A -0.3825 10`

`by=.EACHI` allows to group by each subset of known groups in **i**. A key needs to be set to use `by=.EACHI`.	`DT[c("A","C"), sum(V4)]`	Returns one total sum of column **V4**, for the rows of the key column (**V2**) that have values **A** or **C**.	`[1] 52`

	`DT[c("A","C"), sum(V4), by=.EACHI]`	Returns one sum of column **V4** for the rows of column **V2** that have value **A**, and another sum for the rows of column **V2** that have value **C**.	`   V2 V1`
			`1:  A 22`
			`2:  C 30`

Any number of columns can be set as key using `setkey()`. This way rows can be selected on 2 keys which is an equijoin.	`setkey(DT,V1,V2)`	Sorts by column **V1** and then by column **V2** within each group of column **V1**.	Returns results invisibly.

	`DT[.(2,"C")]`	Selects the rows that have the value **2** for the first key (column **V1**) and the value **C** for the second key (column **V2**).	`   V1 V2      V3 V4`
			`1:  2  C -0.3825  6`
			`2:  2  C  0.6651 12`

	`DT[.(2, c("A","C"))]`	Selects the rows that have the value 2 for the first key (column **V1**) and within those rows the value **A** or **C** for the second key (column **V2**).	`   V1 V2      V3 V4`
			`1:  2  A  0.6651  4`
			`2:  2  A -0.3825 10`
			`3:  2  C -0.3825  6`
			`4:  2  C  0.6651 12`

ADVANCED DATA TABLE OPERATIONS

What?	*Example*	*Notes*	*Output*

`.N` contains the number of rows or the last row.	Usable in **i**: `DT[.N-1]`	Returns the penultimate row of the `data.table`.	`   V1 V2      V3 V4`
			`1:  1  B -1.0604 11`

	Usable in **j**: `DT[,.N]`	Returns the number of rows.	`[1] 12`

`.()` is an alias to `list()` and means the same. The `.()` notation is not needed when there is only one item in **by** or **j**.	Usable in **j**: `DT[,.(V2,V3)] #or DT[,list(V2,V3)]`	Columns **V2** and **V3** are returned as a `data.table`.	`   V2      V3`
			`1:  A -1.1727`
			`2:  B -0.3825`
			`3:  C -1.0604`
			`...`

	Usable in **by**: `DT[, mean(V3), by=.(V1,V2)]`	Returns the result of **j**, grouped by all possible combinations of groups specified in **by**.	`   V1 V2       V1`
			`1:  1  A -1.11655`
			`2:  2  B  0.14130`
			`3:  1  C -1.11655`
			`4:  2  A  0.14130`
			`5:  1  B -1.11655`
			`6:  2  C  0.14130`

`.SD` is a `data.table` and holds all the values of all columns, except the one specified in **by**. It reduces programming time but keeps readability. `.SD` is only accessible in **j**.	`DT[, print(.SD), by=V2]`	To look at what `.SD` contains.	`#All of .SD (output too long to display here)`

	`DT[,.SD[c(1,.N)], by=V2]`	Selects the first and last row grouped by column **V2**.	`   V2 V1      V3 V4`
			`1:  A  1 -1.1727  1`
			`2:  A  2 -0.3825 10`
			`3:  B  2 -0.3825  2`
			`4:  B  1 -1.0604 11`
			`5:  C  1 -1.0604  3`
			`6:  C  2  0.6651 12`

	`DT[, lapply(.SD, sum), by=V2]`	Calculates the sum of all columns in `.SD` grouped by **V2**.	`   V2 V1      V3 V4`
			`1:  A  6 -1.9505 22`
			`2:  B  6 -1.9505 26`
			`3:  C  6 -1.9505 30`

`.SDcols` is used together with `.SD`, to specify a subset of the columns of `.SD` to be used in **j**.	`DT[, lapply(.SD,sum), by=V2, .SDcols = c("V3","V4")]`	Same as above, but only for columns **V3** and **V4** of `.SD`.	`   V2      V3 V4`
			`1:  A -1.9505 22`
			`2:  B -1.9505 26`
			`3:  C -1.9505 30`

`.SDcols` can be the result of a function call.	`DT[, lapply(.SD,sum), by=V2, .SDcols = paste0("V",3:4)]`	Same result as the line above.	

CHAINING HELPS TACK EXPRESSIONS TOGETHER AND AVOID (UNNECESSARY) INTERMEDIATE ASSIGNMENTS

What?	*Example*	*Notes*	*Output*
Do 2 (or more) sets of statements at once by chaining them in one statement. This corresponds to *having* in SQL.	`DT<-DT[, .(V4.Sum = sum(V4)),by=V1]` `DT[V4.Sum > 40] #no chaining`	First calculates sum of **V4**, grouped by **V1**. Then selects that group of which the sum is > 40 without chaining.	`V1 V4.Sum` `1: 1 36` `2: 2 42`
	`DT[, .(V4.Sum = sum(V4)),` `by=V1][V4.Sum > 40 ]`	Same as above, but with chaining.	`V1 V4.Sum` `1: 2 42`
Order the results by chaining.	`DT[, .(V4.Sum = sum(V4)),` `by=V1][order(-V1)]`	Calculates sum of **V4**, grouped `by` **V1**, and then `orders` the result on **V1**.	`V1 V4.Sum` `1: 2 42` `2: 1 36`

USING THE set()-FAMILY

What?	*Example*	*Notes*	*Output*
`set()` is used to repeatedly update rows and columns by reference. `Set()` is a loopable low overhead version of `:=`. Watch out: It can not handle grouping operations.	Syntax of `set()`: `for (i in from:to) set(DT, row, column, new value).`		
	`rows = list(3:4,5:6)` `cols = 1:2` `for (i in seq_along(rows))` `{ set(DT,` `i=rows[[i]],` `j = cols[i],` `value = NA) }`	Sequence along the values of `rows`, and for the values of `cols`, set the values of those elements equal to **NA**.	Returns the result invisibly. `> DT` `V1 V2 V3 V4` `1: 1 A -1.1727 1` `2: 2 B -0.3825 2` `3: NA C -1.0604 3` `4: NA A 0.6651 4` `5: 1 NA -1.1727 5` `6: 2 NA -0.3825 6` `7: 1 A -1.0604 7` `8: 2 B 0.6651 8`
`setnames()` is used to create or update column names by reference.	Syntax of `setnames()`: `setnames(DT,"old","new")[]`	Changes (`set`) the name of column **old** to **new**. Also, when `[]` is added at the end of any `set()` function the result is printed to the screen.	
	`setnames(DT,"V2","Rating")`	Sets the name of column **V2** to **Rating**.	Returns the result invisibly.
	`setnames(DT,c("V2","V3"),` `c("V2.rating","V3.DataCamp"))`	Changes two column names.	Returns the result invisibly.
`setcolorder()` is used to reorder columns by reference.	`setcolorder(DT, "neworder")`	**neworder** is a character vector of the new column name ordering.	
	`setcolorder(DT,` `c("V2","V1","V4","V3"))`	Changes the column ordering to the contents of the vector.	Returns the result invisibly. The new column order is now `[1] "V2" "V1" "V4" "V3"`