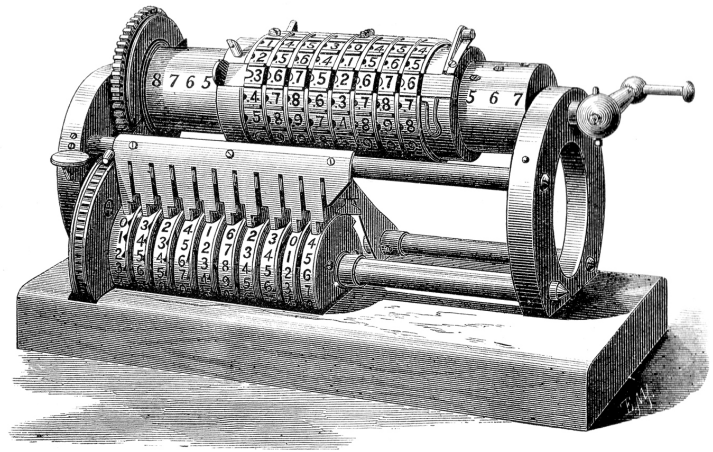
LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

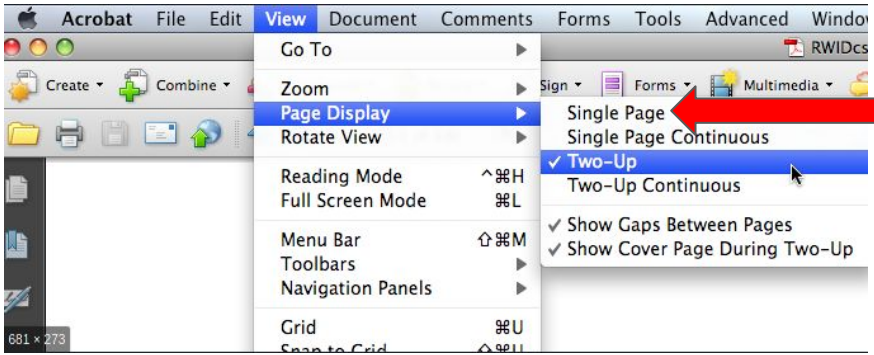
Programming in R

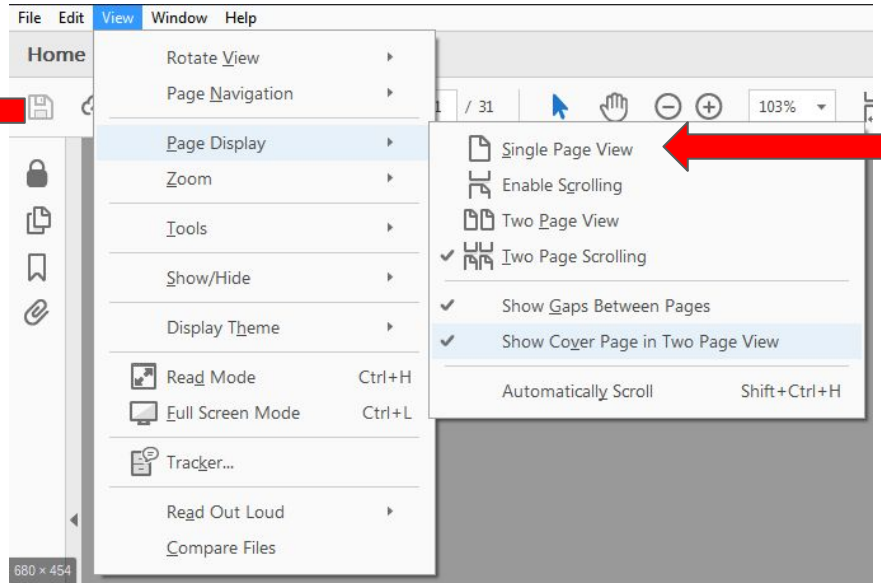


Unit 9: Tabular Data

About these Slides

The best way to view these slides (and to get the most out of the animations) is to view it in "presentation mode" or "single page view". See the images below where to set up single page view in your system.





Tabular Data

Remember, this course has multiple goals:

- Learn things about the R language: "R"
- Get to know nice tools to use: "Tools"
- Learn things about software development in general: "Dev"

This unit:

- "Tools" Track: The `data.table` Package

Tools Track

Tabular Data with `data.table`

Tabular Data: R's Strength

- R has its strengths and weaknesses
- Working with tables is one of its great strengths
  - (other strengths: plotting w/ ggplot2, obscure statistical methods only implemented in R)
- getting fluid with tabular data handling will make you very productive
- Mainly three ways to handle tabular data in R:
  - Base-R: built-in `data.frame`, together with `subset()`, `within()`, `aggregate()`, ...
  - `dplyr` / `tidyverse`: `tibble`, together with `filter()`, `select()`, `mutate()`, ...
  - `data.table`
- You already learned the others, now we do `data.table`:
  - faster, better at handling large data
  - more principled, simpler syntax; less to learn (-- which is not the same as "easier to learn"!)
  - (to some degree this is subjective / use-case dependent)
  - A new and different way of looking at data!

Tabular Data in R

- R native representation of *tabular data*: data.frame

```
> df <- data.frame(a = c(1, 2, 3), b = c("a", "b", "c"), stringsAsFactors=FALSE)
> df
  a b
1 1 a
2 2 b
3 3 c
> df[[1]]
[1] 1 2 3
> df[[2]]
[1] "a" "b" "c"
> df$a
[1] 1 2 3
> df$b
[1] "a" "b" "c"
```

Tabular Data in R

- R native representation of *tabular data*: data.frame

```
> df <- data.frame(a = c(1, 2, 3), b = c("a", "b", "c"), stringsAsFactors=FALSE)
> df
  a b
1 1 a
2 2 b
3 3 c
> df[[1]]
[1] 1 2 3
> df[[2]]
[1] "a" "b" "c"
> df$a
[1] 1 2 3
> df$b
[1] "a" "b" "c"
```

Familiar? This works just like a List...

Tabular Data in R

- R native representation of *tabular data*: data.frame

```
> df <- data.frame(a = c(1, 2, 3), b = c("a", "b", "c"), stringsAsFactors=FALSE)
> df
  a b
1 1 a
2 2 b
3 3 c
> df[[1]]
[1] 1 2 3
> df[[2]]
[1] "a" "b" "c"
> df$a
[1] 1 2 3
> df$b
[1] "a" "b" "c"
```

Familiar? This works just like a List...

```
> df <- list(a = c(1, 2, 3), b = c("a", "b", "c"))
> df[[1]]
[1] 1 2 3
> df[[2]]
[1] "a" "b" "c"
> df$a
[1] 1 2 3
> df$b
[1] "a" "b" "c"
```

Tabular Data in R

- R native representation of *tabular data*: data.frame
  - What is a *data.frame*? Just a list.
    - with nice printer
    - with requirement that all columns have the same length
    - with matrix behaviour when accessed by `[ ]` and `[ , ]`

Tabular Data in R

- R native representation of *tabular data*: data.frame
  - What is a *data.frame*? Just a list.
    - with nice printer
    - with requirement that all columns have the same length
    - with matrix behaviour when accessed by `[ ]` and `[ , ]`
  - Limitations
    - Minor: Printer is a bit annoying when there are a lot of rows...
    - Major: A bit slow for large datasets

Tabular Data in R

- R native representation of *tabular data*: data.frame
  - What is a *data.frame*? Just a list.
    - with nice printer
    - with requirement that all columns have the same length
    - with matrix behaviour when accessed by `[ ]` and `[ , ]`
  - Limitations
    - Minor: Printer is a bit annoying when there are a lot of rows...
    - Major: A bit slow for large datasets
- Enter data.table
  - representation similar to data.frame
  - easy to convert from / to data.frame
  - different usage (!) of `[ ]` and `[ , ]`
  - more features
  - faster

data.table

the basics

data.table Setup

```
> library("data.table")
> dt <- data.table(a = 1:4, b = letters[1:4])
> dt
   a b
1: 1 a
2: 2 b
3: 3 c
4: 4 d
> options(datatable.print.class = TRUE)
> options(datatable.print.keys = TRUE)
> options(datatable.print.trunc.cols = TRUE)
> dt
       a      b
   <int> <char>
1:     1      a
2:     2      b
3:     3      c
4:     4      d
>
```

Construct a data.table

```
> data.table(a = 1:4, b = letters[1:4])
       a      b
   <int> <char>
1:     1      a
2:     2      b
3:     3      c
4:     4      d
```

```
> df <- data.frame(a = 1:4, b = letters[1:4])
> setDT(df)
> df
       a      b
   <int> <char>
1:     1      a
2:     2      b
3:     3      c
4:     4      d
```

"in-place"
also: `setDF()`

```
> as.data.table(iris)
    Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
           <num>       <num>        <num>       <num>    <fctr>
 1:          5.1         3.5          1.4         0.2    setosa
 2:          4.9         3.0          1.4         0.2    setosa
 3:          4.7         3.2          1.3         0.2    setosa
 4:          4.6         3.1          1.5         0.2    setosa
 5:          5.0         3.6          1.4         0.2    setosa
---
```

The [-Operator

```
> dt
       a      b
   <int> <char>
1:     1      a
2:     2      b
3:     3      c
4:     4      d
```

- No surprises, so far:

```
> dt[c(2, 4), ]
       a      b
   <int> <char>
1:     2      b
2:     4      d
> dt[, "b"]
        b
   <char>
1:      a
2:      b
3:      c
4:      d
```

The [-Operator

```
> dt
       a      b
   <int> <char>
1:     1      a
2:     2      b
3:     3      c
4:     4      d
```

- But:

```
> col <- "b"
> dt[, col]
Error in `[.data.table`(dt, , col) :
  j (the 2nd argument inside [...]) is a single
symbol but column name 'col' is not found. Perhaps
you intended DT[, ..col]. This difference to
data.frame is deliberate and explained in FAQ 1.1.
```

Something is up here!

The `[`-Operator

How does it actually work?

- "`dt[i, j]`"
  - `i`: selects the row(s)?
  - `j`: selects the columns(s)?

The `[`-Operator

How does it actually work?

- "`dt[i, j]`"
  - `i`: selects the row(s) ✓
    expression that is **evaluated in the data.table** and selects the rows!
  - `j`: selects the columns(s)?

The [-Operator

How does it actually work?

- "dt[i, j]"
  - i: selects the row(s) ✓
    expression that is **evaluated in the data.table** and selects the rows!
  - j: ~~selects the columns(s)~~
    expression that is **evaluated in the data.table** and **constructs** values!

```
> dt
       a      b
   <int> <char>
1:     1      a
2:     2      b
3:     3      c
4:     4      d
```

```
> dt[a %% 2 == 0, ]
       a      b
   <int> <char>
1:     2      b
2:     4      d
```

```
> dt[, .(a, x = 2 ^ a)]
       a     x
   <int> <num>
1:     1     2
2:     2     4
3:     3     8
4:     4    16
```

The `[`-Operator -- `i`

The `[`-Operator -- `i`

"`dt[i, j]`"

- integer-valued `numeric()` -- as usual
- `logical()` -- as usual

```
> dt
       a      b
   <int> <char>
1:     1      a
2:     2      b
3:     3      c
4:     4      d
```

```
> dt[c(3, 1), ]
       a      b
   <int> <char>
1:     3      c
2:     1      a
```

```
> l <-c(TRUE, FALSE, FALSE, TRUE)
> dt[l, ]
       a      b
   <int> <char>
1:     1      a
2:     4      d
```

- An expression that results in either of these!
- ... and joins (see later)

The [-Operator -- i

```
> iris.dt <- as.data.table(iris)
> iris.dt
     Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
            <num>       <num>        <num>       <num>    <fctr>
  1:          5.1         3.5          1.4         0.2    setosa
  2:          4.9         3.0          1.4         0.2    setosa
  3:          4.7         3.2          1.3         0.2    setosa
  4:          4.6         3.1          1.5         0.2    setosa
  5:          5.0         3.6          1.4         0.2    setosa
 ---
146:          6.7         3.0          5.2         2.3 virginica
147:          6.3         2.5          5.0         1.9 virginica
148:          6.5         3.0          5.2         2.0 virginica
149:          6.2         3.4          5.4         2.3 virginica
150:          5.9         3.0          5.1         1.8 virginica
```

```
> iris.dt[Species %like% '^v', ]
     Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
            <num>       <num>        <num>       <num>     <fctr>
  1:          7.0         3.2          4.7         1.4 versicolor
  2:          6.4         3.2          4.5         1.5 versicolor
  3:          6.9         3.1          4.9         1.5 versicolor
  4:          5.5         2.3          4.0         1.3 versicolor
  5:          6.5         2.8          4.6         1.5 versicolor
 ---
 96:          6.7         3.0          5.2         2.3  virginica
 97:          6.3         2.5          5.0         1.9  virginica
 98:          6.5         3.0          5.2         2.0  virginica
 99:          6.2         3.4          5.4         2.3  virginica
100:          5.9         3.0          5.1         1.8  virginica
```

Select rows based on truth-values:

- dplyr `filter()`
  SQL "WHERE"
  base-R `subset()`
- Combine multiple conditions using `&`, `|` (single operators!)
- data.table provides some helpers:
  - `%like%` (i.e. `grepl()`)
  - `%between%`
- `NA` is treated as `FALSE`

The [-Operator -- i

```
> iris.dt <- as.data.table(iris)
> iris.dt
     Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
            <num>       <num>        <num>       <num>    <fctr>
  1:          5.1         3.5          1.4         0.2    setosa
  2:          4.9         3.0          1.4         0.2    setosa
  3:          4.7         3.2          1.3         0.2    setosa
  4:          4.6         3.1          1.5         0.2    setosa
  5:          5.0         3.6          1.4         0.2    setosa
 ---
146:          6.7         3.0          5.2         2.3 virginica
147:          6.3         2.5          5.0         1.9 virginica
148:          6.5         3.0          5.2         2.0 virginica
149:          6.2         3.4          5.4         2.3 virginica
150:          5.9         3.0          5.1         1.8 virginica
```

```
> iris.dt[order(Sepal.Length + Sepal.Width), ]
     Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
            <num>       <num>        <num>       <num>     <fctr>
  1:          4.5         2.3          1.3         0.3     setosa
  2:          5.0         2.0          3.5         1.0 versicolor
  3:          4.3         3.0          1.1         0.1     setosa
  4:          5.0         2.3          3.3         1.0 versicolor
  5:          4.4         2.9          1.4         0.2     setosa
 ---
146:          7.6         3.0          6.6         2.1  virginica
147:          7.7         3.0          6.1         2.3  virginica
148:          7.2         3.6          6.1         2.5  virginica
149:          7.7         3.8          6.7         2.2  virginica
150:          7.9         3.8          6.4         2.0  virginica
```

Select rows based on indices:

- dplyr `slice()`
  base-R `[ , ]`
- Useful with
  - `order()`
  - `which.min()`, `which.max()`
    - (also `== min()` works, but could give multiple rows)
  - `sample.int()`

The [-Operator -- i

```
> iris.dt <- as.data.table(iris)
> iris.dt
     Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
            <num>       <num>        <num>       <num>    <fctr>
  1:          5.1         3.5          1.4         0.2    setosa
  2:          4.9         3.0          1.4         0.2    setosa
  3:          4.7         3.2          1.3         0.2    setosa
  4:          4.6         3.1          1.5         0.2    setosa
  5:          5.0         3.6          1.4         0.2    setosa
 ---
146:          6.7         3.0          5.2         2.3 virginica
147:          6.3         2.5          5.0         1.9 virginica
148:          6.5         3.0          5.2         2.0 virginica
149:          6.2         3.4          5.4         2.3 virginica
150:          5.9         3.0          5.1         1.8 virginica
```

```
> iris.dt[sample.int(150, 3), ]
   Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
          <num>       <num>        <num>       <num>     <fctr>
1:          5.8         2.7          4.1         1.0 versicolor
2:          5.1         3.5          1.4         0.2     setosa
3:          4.7         3.2          1.3         0.2     setosa
```

Select rows based on indices:

- dplyr `slice()`
  base-R `[ , ]`
- Useful with
  - `order()`
  - `which.min()`, `which.max()`
    - (also `== min()` works, but could give multiple rows)
  - `sample.int()`

The `[`-Operator -- `i`

- Select rows
  - based on `logical()` values -- variable or expression
  - based on integer `numeric()` values -- variable or expression
- Things to consider:
  - Expressions are evaluated in data.table first, and take other variables second

```
> sepalval <- 5
> iris.dt[Sepal.Length == sepalval, ]
    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
           <num>       <num>        <num>       <num>     <fctr>
 1:            5         3.6          1.4         0.2     setosa
 2:            5         3.4          1.5         0.2     setosa
 3:            5         3.0          1.6         0.2     setosa
 4:            5         3.4          1.6         0.4     setosa
 5:            5         3.2          1.2         0.2     setosa
 6:            5         3.5          1.3         0.3     setosa
 7:            5         3.5          1.6         0.6     setosa
 8:            5         3.3          1.4         0.2     setosa
 9:            5         2.0          3.5         1.0 versicolor
10:            5         2.3          3.3         1.0 versicolor
```

The `[`-Operator -- `i`

- Select rows
  - based on `logical()` values -- variable or expression
  - based on integer `numeric()` values -- variable or expression
- Things to consider:
  - Expressions are evaluated in data.table first, and take other variables second
  - Single symbols are interpreted as (external) variables

```
> dt <- data.table(a = 1:4, b = c(TRUE, FALSE, TRUE, FALSE))
> dt
       a      b
   <int> <lgcl>
1:     1   TRUE
2:     2  FALSE
3:     3   TRUE
4:     4  FALSE
> dt[b, ]
Error in `[.data.table`(dt, b, ) :
  b is not found in calling scope but it is a column of type logical. If you wish to select
rows where that column contains TRUE, or perhaps that column contains row numbers of itself
to select, try DT[(col)], DT[DT$col], or DT[col==TRUE] is particularly clear and is
optimized. When the first argument inside DT[...] is a single symbol (e.g. DT[var]),
data.table looks for var in calling scope.
```

The `[`-Operator -- `i`

- Select rows
  - based on `logical()` values -- variable or expression
  - based on integer `numeric()` values -- variable or expression
- Things to consider:
  - Expressions are evaluated in data.table first, and take other variables second
  - Single symbols are interpreted as (external) variables

```
> dt <- data.table(a = 1:4, b = c(TRUE, FALSE, TRUE, FALSE))
> dt
       a      b
   <int> <lgcl>
1:     1   TRUE
2:     2  FALSE
3:     3   TRUE
4:     4  FALSE
> dt[b == TRUE, ]
       a      b
   <int> <lgcl>
1:     1   TRUE
2:     3   TRUE
```

```
> dt[(b), ]
       a      b
   <int> <lgcl>
1:     1   TRUE
2:     3   TRUE
```

The `[`-Operator -- `i`

- Select rows
  - based on `logical()` values -- variable or expression
  - based on integer `numeric()` values -- variable or expression
- Things to consider:
  - Expressions are evaluated in data.table first, and take other variables second
  - Single symbols are interpreted as (external) variables
  - Unlike data.frame, using `[ ]` without a comma selects the row
    -- this is like giving the first of several arguments to the `[, ]`-function

```
> iris.dt[c(1, 3)]
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
          <num>       <num>        <num>       <num>  <fctr>
1:          5.1         3.5          1.4         0.2  setosa
2:          4.7         3.2          1.3         0.2  setosa
```

The [-Operator -- j

The [-Operator -- j

"dt[i, j]"

```
> dt
       a      b
   <int> <char>
1:     1      a
2:     2      b
3:     3      c
4:     4      d
```

- integer-valued numeric(), logical(), character() **constants**

```
> dt[, c(TRUE, FALSE)]
       a
   <int>
1:     1
2:     2
3:     3
4:     4
```

```
> dt[, c(2, 2)]
       b      b
   <char> <char>
1:     a      a
2:     b      b
3:     c      c
4:     d      d
```

```
> dt[, "b"]
       b
   <char>
1:     a
2:     b
3:     c
4:     d
```

- numeric(), logical(), character() values when with = FALSE
- expressions giving *atomic* results
- expressions giving *list* results
- :=-assignments

The [-Operator -- j

Select column names / indices with argument `with = FALSE`

```
> dt
       a      b
   <int> <char>
1:     1      a
2:     2      b
3:     3      c
4:     4      d
```

```
> selecting <- "a"
> dt[, selecting, with = FALSE]
       a
   <int>
1:     1
2:     2
3:     3
4:     4
```

```
> selecting <- 2
> dt[, selecting, with = FALSE]
       b
   <char>
1:     a
2:     b
3:     c
4:     d
```

The [-Operator -- j

Most expressions are evaluated inside the data.table

- can take outside variables
  - potentially dangerous when you don't know the table!
- expression generates *atomic* value -> returns *vector*

```
> dt
        a       b
   <int> <char>
1:     1       a
2:     2       b
3:     3       c
4:     4       d
```

```
> dt[, a]
[1] 1 2 3 4
> dt[, a + 1]
[1] 2 3 4 5
> x <- 2
> dt[, a ^ x]
[1]  1  4  9 16
> b <- 2
> dt[, a ^ b]
Error in a^b : non-numeric argument to binary operator
```

b-column takes precedence before b variable!

The [-Operator -- j

Most expressions are evaluated inside the data.table

- can take outside variables
  - potentially dangerous when you don't know the table!
- expression generates *atomic* value -> returns *vector*
- expression generates *list* -> returns *data.table*

```
> dt
       a      b
   <int> <char>
1:     1      a
2:     2      b
3:     3      c
4:     4      d
```

```
> dt[, list(a + 1)]
      V1
   <num>
1:     2
2:     3
3:     4
4:     5
> dt[, list(x = a + 1)]
       x
   <num>
1:     2
2:     3
3:     4
4:     5
```

The `[`-Operator -- `j`

Most expressions are evaluated inside the data.table

- can take outside variables
  - potentially dangerous when you don't know the table!
- expression generates *atomic* value -> returns *vector*
- expression generates *list* -> returns *data.table*
  - `.()` as shortcut for `list()`

```
> dt
       a      b
   <int> <char>
1:     1      a
2:     2      b
3:     3      c
4:     4      d
```

```
> dt[, list(a + 1)]
      V1
   <num>
1:     2
2:     3
3:     4
4:     5
> dt[, list(x = a + 1)]
       x
   <num>
1:     2
2:     3
3:     4
4:     5
```

```
> dt[, .(a + 1)]
      V1
   <num>
1:     2
2:     3
3:     4
4:     5
> dt[, .(x = a + 1)]
       x
   <num>
1:     2
2:     3
3:     4
4:     5
```

The `[`-Operator -- `j`

Most expressions are evaluated inside the data.table

- can take outside variables
  - potentially dangerous when you don't know the table!
- expression generates *atomic* value -> returns *vector*
- expression generates *list* -> returns *data.table*
  - `.()` as shortcut for `list()`
  - no-one said this must have the appropriate length!

```
> dt
        a       b
   <int> <char>
1:      1       a
2:      2       b
3:      3       c
4:      4       d
```

```
> dt[, .(x = sum(a))]
       x
   <int>
1:    10
```

```
> dt[, .(a = head(a, 2), b = tail(b, 2))]
       a      b
   <int> <char>
1:     1      c
2:     2      d
```

The [-Operator -- j

Most expressions are evaluated inside the data.table

- can take outside variables
  - potentially dangerous when you don't know the table!
- expression generates *atomic* value -> returns *vector*
- expression generates *list* -> returns *data.table*
  - `.()` as shortcut for `list()`
  - no-one said this must have the appropriate length!
  - no-one said it must involve the data.table at all!

```
> dt
       a      b
   <int> <char>
1:     1      a
2:     2      b
3:     3      c
4:     4      d
```

```
> dt[, .(x = runif(2), y = c("Hi", "Bye"))]
          x      y
      <num> <char>
1: 0.7448941     Hi
2: 0.4114382    Bye
```

The `[`-Operator -- `j`

Most expressions are evaluated inside the data.table

- can take outside variables
  - potentially dangerous when you don't know the table!
- expression generates *atomic* value -> returns *vector*
- expression generates *list* -> returns *data.table*
  - `.()` as shortcut for `list()`
  - no-one said this must have the appropriate length!
  - no-one said it must involve the data.table at all!
  - --> Just imagine the `.()` as a new construction, calling `data.table()`

```
> dt
       a      b
   <int> <char>
1:     1      a
2:     2      b
3:     3      c
4:     4      d
```

```
> dt[, .(x = runif(2), y = c("Hi", "Bye"))]
          x      y
      <num> <char>
1: 0.7448941     Hi
2: 0.4114382    Bye
```

```
> data.table(x = runif(2), y = c("Hi", "Bye"))
          x      y
      <num> <char>
1: 0.9489363     Hi
2: 0.9164093    Bye
```

The `[`-Operator -- `j` -- `:=`-assignment

Use `:=` to assign inside a `data.table`

- similar to `within()` (base-R) or `mutate()` (dplyr)
- create new columns

```
> dt
       a      b
   <int> <char>
1:     1      a
2:     2      b
3:     3      c
4:     4      d
```

```
> dt[, c := LETTERS[1:4]]
> dt
       a      b      c
   <int> <char> <char>
1:     1      a      A
2:     2      b      B
3:     3      c      C
4:     4      d      D
```

The [-Operator -- j -- :=-assignment

Use `:=` to assign inside a `data.table`

- similar to `within()` (base-R) or `mutate()` (dplyr)
- create new columns
- ... and update existing columns

```
> dt
       a      b
   <int> <char>
1:     1      a
2:     2      b
3:     3      c
4:     4      d
```

```
> dt[, c := LETTERS[1:4]]
> dt
       a      b      c
   <int> <char> <char>
1:     1      a      A
2:     2      b      B
3:     3      c      C
4:     4      d      D
```

```
> dt[, a := a + 1]
> dt
       a      b
   <num> <char>
1:     2      a
2:     3      b
3:     4      c
4:     5      d
```

The `[`-Operator -- `j` -- `:=`-assignment

Use `:=` to assign inside a `data.table`

- similar to `within()` (base-R) or `mutate()` (dplyr)
- create new columns
- ... and update existing columns
- This happens **in-place!** -- the `dt`-variable changes

```
> dt
       a      b
   <int> <char>
1:     1      a
2:     2      b
3:     3      c
4:     4      d
```

```
> dt[, c := LETTERS[1:4]]
> dt
       a      b      c
   <int> <char> <char>
1:     1      a      A
2:     2      b      B
3:     3      c      C
4:     4      d      D
```

```
> dt[, a := a + 1]
> dt
       a      b
   <num> <char>
1:     2      a
2:     3      b
3:     4      c
4:     5      d
```

The [-Operator -- j -- :=-assignment

```
> dt
        a       b
   <int> <char>
1:      1       a
2:      2       b
3:      3       c
4:      4       d
```

Use `:=` to assign inside a `data.table`

- similar to `within()` (base-R) or `mutate()` (dplyr)
- create new columns
- ... and update existing columns
- This happens **in-place!** -- the `dt`-variable changes
- use `(var)` to assign to variables dynamically

```
> target <- "x"
> dt[, (target) := sqrt(a)]
> dt
        a       b        x
   <int> <char>    <num>
1:      1       a 1.000000
2:      2       b 1.414214
3:      3       c 1.732051
4:      4       d 2.000000
```

```
> targets <- c("x", "y")
> dt[, (targets) := .(sqrt(a), a^2)]
> dt
        a       b        x     y
   <int> <char>    <num> <num>
1:      1       a 1.000000     1
2:      2       b 1.414214     4
3:      3       c 1.732051     9
4:      4       d 2.000000    16
```

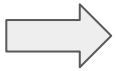
The [-Operator -- j -- :=-assignment

```
> dt
       a      b
   <int> <char>
1:     1      a
2:     2      b
3:     3      c
4:     4      d
```

Use `:=` to assign inside a `data.table`

- similar to `within()` (base-R) or `mutate()` (dplyr)
- create new columns
- ... and update existing columns
- This happens **in-place!** -- the `dt`-variable changes
- use `(var)` to assign to variables dynamically
- Two ways to assign to create multiple columns: `` `:=()` ``, and non-scalars on both sides

```
> dt[, c("x", "y") := .(sqrt(a), a^2)]
> dt[, `:=`(x = sqrt(a), y = a^2)]
```



```
> dt
       a      b        x     y
   <int> <char>    <num> <num>
1:     1      a 1.000000     1
2:     2      b 1.414214     4
3:     3      c 1.732051     9
4:     4      d 2.000000    16
```

The `[`-Operator -- `j` -- `:=`-assignment

```
> dt
        a       b
   <int> <char>
1:     1       a
2:     2       b
3:     3       c
4:     4       d
```

Use `:=` to assign inside a `data.table`

- similar to `within()` (base-R) or `mutate()` (dplyr)
- create new columns
- ... and update existing columns
- This happens **in-place!** -- the `dt`-variable changes
- use `(var)` to assign to variables dynamically
- Two ways to assign to create multiple columns: `` `:=()` ``, and non-scalars on both sides
- Delete columns by assigning `NULL`

```
> dt[, a := NULL]
> dt
        b
   <char>
1:      a
2:      b
3:      c
4:      d
```

The `[`-Operator -- `j` -- `:=`-assignment

Use `:=` to assign inside a `data.table`

- similar to `within()` (base-R) or `mutate()` (dplyr)
- create new columns
- ... and update existing columns
- This happens **in-place!** -- the `dt`-variable changes
- use `(var)` to assign to variables dynamically
- Two ways to assign to create multiple columns: `` `:=()` ``, and non-scalars on both sides
- Delete columns by assigning `NULL`
- You can change column-types

```
> dt
       a      b
   <int> <char>
1:     1      a
2:     2      b
3:     3      c
4:     4      d
```

```
> dt[, a := as.character(a)]
> dt
       a      b
   <char> <char>
1:      1      a
2:      2      b
3:      3      c
4:      4      d
```

The `[`-Operator -- `j` -- `:=`-assignment

```
> dt
        a       b
   <int> <char>
1:     1       a
2:     2       b
3:     3       c
4:     4       d
```

Use `:=` to assign inside a `data.table`

- similar to `within()` (base-R) or `mutate()` (dplyr)
- create new columns
- ... and update existing columns
- This happens **in-place!** -- the `dt`-variable changes
- use `(var)` to assign to variables dynamically
- Two ways to assign to create multiple columns: `` `:=()` ``, and non-scalars on both sides
- Delete columns by assigning `NULL`
- You can change column-types
- Add another `[]` at the end to print result immediately

```
> dt[, a := -a][]
        a       b
   <int> <char>
1:    -1       a
2:    -2       b
3:    -3       c
4:    -4       d
```

The `[`-Operator -- `j`

- Select columns with constants or `with = FALSE` -- least interesting part
- expressions giving *atomic* results results in *vectors*
- expressions giving *list* results results in data.tables
  - This very much feels like subsetting most of the time!

```
> dt[, .(a, b, B = toupper(b))]
       a      b      B
   <int> <char> <char>
1:     1      a      A
2:     2      b      B
3:     3      c      C
4:     4      d      D
```

The `[`-Operator -- `j`

- Select columns with constants or `with = FALSE` -- least interesting part
- expressions giving *atomic* results results in *vectors*
- expressions giving *list* results results in data.tables
  - This very much feels like subsetting most of the time!

```
> dt[, .(a, b, B = toupper(b))]
       a      b      B
   <int> <char> <char>
1:     1      a      A
2:     2      b      B
3:     3      c      C
4:     4      d      D
```

- "expression" can mean a lot!

```
> dt[, {
+   x <- sqrt(a)
+   y <- match(b, letters)
+   x ^ y
+ }]
[1]  1.000000  2.000000  5.196152 16.000000
```

The `[`-Operator -- `j`

- Select columns with constants or `with = FALSE` -- least interesting part
- expressions giving *atomic* results results in *vectors*
- expressions giving *list* results results in data.tables
  - This very much feels like subsetting most of the time!

```
> dt[, .(a, b, B = toupper(b))]
       a      b      B
   <int> <char> <char>
1:     1      a      A
2:     2      b      B
3:     3      c      C
4:     4      d      D
```

- "expression" can mean a lot!

```
> dt[, {
+   cat(sprintf("The value of a[[2]] is: %s\n", a[[2]]))
+   NULL
+ }]
The value of a[[2]] is: 2
NULL
```

The `[`-Operator -- `j`

- Select columns with constants or `with = FALSE` -- least interesting part
- expressions giving *atomic* results results in *vectors*
- expressions giving *list* results results in data.tables
  - This very much feels like subsetting most of the time!

```
> dt[, .(a, b, B = toupper(b))]
       a      b      B
   <int> <char> <char>
1:     1      a      A
2:     2      b      B
3:     3      c      C
4:     4      d      D
```

- "expression" can mean a lot!
- Modify columns in-place using `:=`-assignments

```
> dt[, {
+   cat(sprintf("The value of a[[2]] is: %s\n", a[[2]]))
+   NULL
+ }]
The value of a[[2]] is: 2
NULL
```

The `[`-Operator -- `i` and `j`

The [-Operator -- i and j

"dt[i, j]"

- i selects rows before expression in j is applied
- i.e., it affects the variables that the j-expression has access to
- this usually has exactly the results you'd expect

```
> dt
       a      b
   <int> <char>
1:     1      a
2:     2      b
3:     3      c
4:     4      d
```

```
> dt[c(1, 3), .(a, b, B = toupper(b))]
       a      b      B
   <int> <char> <char>
1:     1      a      A
2:     3      c      C
```

The [-Operator -- i and j

"dt[i, j]"

```
> dt
       a      b
   <int> <char>
1:     1      a
2:     2      b
3:     3      c
4:     4      d
```

- i selects rows before expression in j is applied
- i.e., it affects the variables that the j-expression has access to
- this usually has exactly the results you'd expect
- but remember:
  > no-one said this must have the appropriate length!

```
> dt[c(1, 3), .(x = 1:4)]
       x
   <int>
1:     1
2:     2
3:     3
4:     4
```

more usefully:

```
> dt[c(1, 3), .(asum = sum(a))]
    asum
   <int>
1:     4
```

The [-Operator -- i and j

"dt[i, j]"

```
> dt
       a      b
   <int> <char>
1:     1      a
2:     2      b
3:     3      c
4:     4      d
```

- i selects rows before expression in j is applied
- i.e., it affects the variables that the j-expression has access to
- this usually has exactly the results you'd expect
- but remember:

  > no-one said this must have the appropriate length!
- Updates only specific rows when using :=-assignment

```
> dt[c(1, 2), b := toupper(b)]
> dt
       a      b
   <int> <char>
1:     1      A
2:     2      B
3:     3      c
4:     4      d
```

```
> dt[a %% 2 == 1, b := toupper(b)]
> dt
       a      b
   <int> <char>
1:     1      A
2:     2      b
3:     3      C
4:     4      d
```

The [-Operator -- i and j

"dt[i, j]"

```
> dt
       a      b
   <int> <char>
1:     1      a
2:     2      b
3:     3      c
4:     4      d
```

- i selects rows before expression in j is applied
- i.e., it affects the variables that the j-expression has access to
- this usually has exactly the results you'd expect
- but remember:

  > no-one said this must have the appropriate length!
- Updates only specific rows when using :=-assignment
  - Fill new columns with NA otherwise

```
> dt[c(1, 2), c := toupper(b)]
> dt
       a      b      c
   <int> <char> <char>
1:     1      a      A
2:     2      b      B
3:     3      c   <NA>
4:     4      d   <NA>
```

The [-Operator -- i and j

"dt[i, j]"

- i selects rows before expression in j is applied
- i.e., it affects the variables that the j-expression has access to
- this usually has exactly the results you'd expect
- but remember:

  > no-one said this must have the appropriate length!
- Updates only specific rows when using :=-assignment
  - Fill new columns with NA otherwise
  - Refuses to change column types

```
> dt
       a      b
   <int> <char>
1:     1      a
2:     2      b
3:     3      c
4:     4      d
```

```
> dt[c(1, 2), a := b]
Warning messages:
1: In `[.data.table`(dt, c(1, 2), `:=`(a, b)) :
  Coercing 'character' RHS to 'integer' to match the type of the target column (column 1
named 'a').
2: In `[.data.table`(dt, c(1, 2), `:=`(a, b)) : NAs introduced by coercion
```

It calls as.integer(b), which gives NAs

```
> dt
      a      b      c
  <int> <char> <char>
1:   NA      a      A
2:   NA      b      B
3:    3      c   <NA>
4:    4      d   <NA>
```

Wrapping Up

What We Expect You to Know

- Construct with (`as.`)`data.table()` or `setDT()`
- `dt[i, j]`
- Both `i` and `j` are evaluated inside the data.table
  - exceptions: `i` with single symbols, `j` with constants
- `i` selects rows `logical()` or `numeric()` values
- `j` creates a new data.table out of lists (use `.()` as shortcut) or returns atomics
  - use `with = FALSE` to select columns directly
- Use `:=` in `j` to create or change columns
  - use `` `:=`() `` to assign multiple columns at once
  - use `(...) := ...` to assign to columns dynamically; possibly multiple at once
- Give both `i` and `j` to have the `j`-expression evaluated on a subset of the original table
- This all "adds up to normality" most of the time
- But you can get very elaborate in your expressions!
- For more useful printing, set

```
options(datatable.print.class = TRUE,
        datatable.print.keys = TRUE,
        datatable.print.trunc.cols = TRUE)
```

Exercise

Try to do the following steps in *many different ways* each

-- how many ways can you come up with?

1. Get a data.table out of the `cars` dataset
   (Note that `setDT()` won't work on built-in datasets directly)
2. Create a column `speed.2`, containing the squared "speed"-values
3. Remove all rows that have `speed` either below 10 or above 20
   - Try this using `%between%`
4. Remove all rows where `speed.2` is smaller than 4 times the `dist`
5. Add the columns `speed.3` and `speed.4` (as `speed^3` and `speed^4`) with a single command
6. Remove the `speed` column

Intermission:
Copy Semantics, Reference Semantics

Copy / Reference Semantics

- One major new thing about data.table is *reference semantics*, because data.table functions may change values *in place*
- Remember that this doesn't work as intended:

```
> f <- function(x) {
+   x <- 3
+ }
```

Copy / Reference Semantics

- One major new thing about data.table is *reference semantics*, because data.table functions may change values *in place*
- Remember that this doesn't work as intended:

```
> f <- function(x) {
+   x <- 3
+ }
>
> val <- 0
> f(val)
> val
[1] 0
```

Not changed!

Copy / Reference Semantics

- One major new thing about data.table is *reference semantics*, because data.table functions may change values *in place*
- Remember that this doesn't work as intended:

```
> f <- function(x) {
+   x <- 3
+ }
>
> val <- 0
> f(val)
> val
[1] 0
```

Not changed!

- **But why is that so?**

Copy / Reference Semantics

How are objects (vectors, lists) handled in R?

- Create a vector / list: memory allocated and filled

```
> x <- list(1, 2, 3)
> data.table::address(x)
[1] "0x55b5c2e9c5a8"
```

- "copy" a vector to a different variable: at first the memory is *not* copied, the copied variable points to the same memory address!

```
> y <- x
> data.table::address(y)
[1] "0x55b5c2e9c5a8"
```

Copy / Reference Semantics

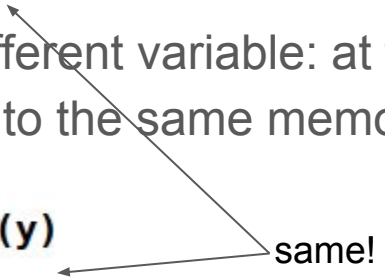
First: how are objects (vectors, lists) handled in R?

- Create a vector / list: memory allocated and filled

```
> x <- list(1, 2, 3)
> data.table::address(x)
[1] "0x55b5c2e9c5a8"
```

- "copy" a vector to a different variable: at first the memory is *not* copied, the copied variable points to the same memory address!

```
> y <- x
> data.table::address(y)
[1] "0x55b5c2e9c5a8"
```



Copy / Reference Semantics

First: how are objects (vectors, lists) handled in R?

- However: When I change **y**, then **x** won't get changed!

```
> x[[1]]
[1] 1
> y[[1]]
[1] 1
> y[[1]] <- 2
> x[[1]]
[1] 1
> y[[2]]
[1] 2
```

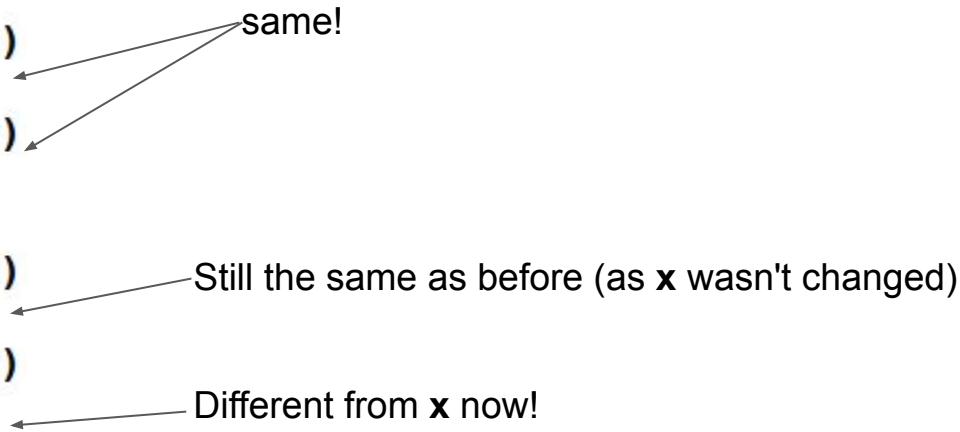
- This is what we call "copy semantics": y behaves as if it is a **copy** of x
- how can this be if they have the same address?

Copy / Reference Semantics

First: how are objects (vectors, lists) handled in R?

- What happens is, R does *copy on write*: Variables share the same address upon copy, until one value changed (upon which the addresses differ).

```
> x <- list(1, 2, 3)
> y <- x
> data.table::address(x)
[1] "0x55b5c2e9bc68"
> data.table::address(y)
[1] "0x55b5c2e9bc68"
> y[[1]] <- 2
> data.table::address(x)
[1] "0x55b5c2e9bc68"
> data.table::address(y)
[1] "0x55b5c2e9bdf8"
```



Copy / Reference Semantics

Recognize that this is very nice:

- it is relatively efficient (because values in memory only get copied when they need to)
- It is also very user-friendly: you don't need to worry that f(x) will change your x
- but it is sometimes a bit wasteful when handling large datasets: just changing one value in a list copies the whole list.
- data.table wants to be able to handle large datasets, so it does *in place operations*

Copy / Reference Semantics

In-Place Operations

- data.table has these kinds of functions:

```
> DT <- data.table(x = 1, y = 2)
> DT
   x y
1: 1 2
> setcolorder(DT, c("y", "x"))
```

Copy / Reference Semantics

In-Place Operations

- data.table has these kinds of functions:

```
> DT <- data.table(x = 1, y = 2)
> DT
   x y
1: 1 2
> setcolorder(DT, c("y", "x"))
> DT
   y x
1: 2 1
> DT[, z := x + y]
> DT
   y x z
1: 2 1 3
```

Value was changed by the function in-place

Value was changed by the [:=] assignment in-place

Copy / Reference Semantics

In-Place Operations

- data.table has in-place functions
- they lead to reference-semantics, in case when two variables point to the same address!

```
> DT <- data.table(x = 1, y = 2)
> DT2 <- DT
> data.table::address(DT)
[1] "0x55b5c908d0b0"
>  data.table::address(DT2)
[1] "0x55b5c908d0b0"
> DT2
   x y
1: 1 2
> setcolorder(DT, c("y", "x"))
> DT2
   y x
1: 2 1
```

DT2 was changed, because DT was changed, and DT and DT2 point to the same address

Copy / Reference Semantics

In-Place Operations

- data.table has in-place functions
- they lead to reference-semantics, in case when two variables point to the same address!
- we say here data.table uses *reference semantics*: variables behave like they reference the same value
- use the copy() function to make disconnected copies of data.table that will not get changed when other variables get changed
- check with data.table::address() whether two data.tables are the same place in memory, and therefore, if in-place operations on one will change the other.

Copy / Reference Semantics

In-Place Operations

- Note: operations that are not data.table::set*-functions, or [:=] column operations will lead to *actual copies* and will not operate in-place

```
> DT <- data.table(x = 1, y = 2)
> DT2 <- DT
> DT[1, 1] <- 100
> DT
     x y
1: 100 2
> DT2
   x y
1: 1 2
```

Not changed!

End of Intermission

data.table

There are lots of resources on data.table, linked from the GitHub homework page for this week. Try to see which ones work well for you!

data.table is very useful because

- it is fast when working with large datasets (impress your dplyr-using coworkers during internships!)
  - **Q**: "Why would I care about speed with 100 GB of data in-memory, when my laptop only has 4 GB of working memory?"
  - **A**: When you work with *real* data (internship, master thesis, job, ...) you will most likely get access to a big machine.
    - E.g. at LMU: LRZ Compute Cloud (192 GB), LRZ RStudio Server (256 GB), LRZ Linux Cluster (up to 6 TB)
    - You can rent a 768 GB machine at amazon AWS for a bit more than 5 EUR / hr, *much less* than your hourly rate as a data scientist!

data.table

There are lots of resources on data.table, linked from the GitHub homework page for this week. Try to see which ones work well for you!

data.table is very useful because

- it is fast when working with large datasets (impress your dplyr-using coworkers during internships!)
- its concise syntax makes it possible to write quite complex data analysis pipelines in a very short time

We therefore encourage you to get to know it well. Look at the following two slides as an overview of what you can learn about data.table, and then use the ressources, as well as the R help(), to find out what each of the points mean.

What We Expect You to Know

`data.table`: know how to...

- tell if an object is a `data.table` or just a `data.frame`
- change how data.table objects are printed
  - by giving `datatable.print.xxx` arguments to `print()` as described in `?print.data.table`
  - by using `options(datatable.print.xxx = OPTION)` to set the option globally
- get a `DT` from `data.frame`s, matrices, `list`s of rows: `as.data.table`, `setDT()`, `rbindlist()`
- get individual rows, columns, elements from a `DT`
- subset a DT: specific columns, specific rows, rows by a condition
- modify or add new columns based on calculations done on old columns using `` `:=` `` or `set()`
  - single col at a time & `[, (<col group>) := .(<value list>)]`
- handle in-place functions (that start with set... in data.table) as well as the `` `:=` `` operator in `[ ]`), know about reference-semantics and use `copy()` if needed
- use `fread()`, `fwrite()` for fast reading / writing of large files
- do aggregation with `` `by =` ``
  - count subgroup sizes with `.N`
  - calculate aggregate values for subgroups
  - advanced aggregation control with `.SD` and `.SDcols`
  - other special values: `.BY`, `.GRP`, `.NGRP`, `.I`, `.EACHI`
- work with list columns that may contain different kinds of data on different rows
- merge / join data.tables
  - both with `merge()` as well as with `X[Y, ...]` (with `X`, `Y` both being a `DT`)
  - understand the difference between inner, left/right, outer, anti join and how to do them in `DT`
- reshape `DT`s between "wide" format and "long" format using `dcast()` and `melt()`
- use keys
  - what are keys useful for?
    - automatic sorting
    - fast row subsetting
    - row selection using X[<value>]
  - `key()`, `indices()`, `haskey()`
  - difference between `setkey()` and `setindex()`
  - difference between `setkey()`/`setindex()` and `setkeyv()`/`setindexv()`

What We Expect You to Know

`data.table`: know about... (grey: not that important at our level)

- using `[]` as a suffix to print `data.table` in-place operation results even when they are "invisible"
- functions that treat `DT`s like sets of rows to do set operations and sorting on them
  - `fintersect()`, `fsetdiff()`, `fsetequal()`, `funion()`: set-operations that treat data table rows as sets
  - `duplicated()`, `unique()`, `anyDuplicated()`: find duplicate rows / restrict to unique rows
    - `uniqueN()`: short for `nrow(unique(x))`
    - also note these have a "`by`" argument
  - `frank()`, `frankv()`: `rank()` on data.table
  - `split()`: split data.table into list of smaller tables (but it is usually better do do aggregate operations with '`by`' in `[ ]`.)
  - `na.omit()`: exclude rows with NAs
- Further `set...()` functions
  - `setattr()`, `setnames()` -- change attributes by reference
  - `setcolorder()` -- reorder columns
  - `setorder()`, `setorderv()` -- reorder rows, similar to `setkey()`/`setkeyv()`, but without setting a key
- helpful operators for the `i` (i.e. row-selector) argument
  - `between()`, `%between%` -- between to values
  - `inrange()`, `%inrange%` -- in any of multiple ranges
  - `like()`, `%like%`, `%flike%`, `%ilike%`: faster `grepl()`; `%flike%`: fixed (not regex), `%ilike%`: ignores case
- general helper functions
  - `first()`, `last()` -- like `head()`/`tail()`, but get just one item
  - `shift()` -- lead or lag a vector
  - `transpose()` -- transpose lists, `data.frames`, `data.tables`
  - `tstrsplit()` -- `transpose()` of `strsplit()`
  - `fcoalesce()`: vectorized: give first non-NA value
  - `nafill()`, `setnafill()` -- fill missing values
  - `CJ()` -- cross product DT
- System info functions and global settings
  - `address()` -- address of an object
  - `setDTthreads()`, `getDTthreads()` -- change cpu parallelization threads
  - `tables()` -- summarize metadata of all 'data.table' objects in memory
  - `getNumericRounding()`, `setNumericRounding()` -- rounding mode for equality checks
  - `timetaken()` -- time difference to result of call `proc.time()`

What We Don't Really Expect You To Know, But Include Here for Completeness Sake

- fast version of R function, optimized for character vectors
  - chgroup(): like order(), but only groups together duplicates instead of sorting
  - chmatch(): character version for match()
  - chorder(): character version of order()
  - %chin%: character version of %in%
- other fast / more robust versions of R functions
  - fifelse(): ifelse(), preserves attributes
  - frank(), frankv(): faster rank(), but also ranks lists, data.frames and data.tables
- Helpers for aggregation and joining
  - groupingsets(), rollup(), cube() -- aggregate by different columns
  - Id column generators
    - rowid(), rowidv(): unique rowid
    - rleid(), rleidv(): run-length encoding
    - SJ(), CJ(): Join helpers
- Experimental (usage of these functions might change)
  - foverlaps(): fast overlap join
  - truelength(), alloc.col(), setalloccol(): over-allocation of column memory
  - frollmean(), frollsum(), frollapply(): rolling window aggregates
  - fsort(): faster sort through multicore
  - (Experimental) date/time class -- mostly a wrapper for POSIXct and Date
    - IDate, ITime: classes
    - as.IDate(), as.ITime(), IDateTime(): conversion
    - year(), quarter(), month(), week(), isoweek(), yday(), mday(), wday(), hour(), minute(), second(): get specific aspect from object

data.table

Advanced Usage Note:

- Don't rely on by-reference updates when adding new columns with `:=`. If you add too many columns using `:=` (more than getOption("datatable.alloccol")), then the data.table is re-allocated and other variables referencing the same data.table are not changed. See `help("truelength")`