Getting and Cleaning Data: Week 1

Video 1-1: Obtaining Data Motivation

About this course

Covers basic ideas behind getting data ready for analysis

- Finding and extracting raw data
- Tidy data principles and how to make data tidy
- Practical implementation through a range of R packages

What this course depends on

- The Data Scientist's Toolbox
- R Programming

What would be useful

- Exploratory analysis
- Reporting data and reproducible research

What you wished data looked like

(Screenshot of Excel spreadsheet)

Data is ideally in neat rows and columns, one observation per row, one variable per column

(My observation: I've seen some of the crappiest, dirtiest data imaginable in Excel)

What does data really look like?

(Screenshot of a flat file, with indecipherable field at start of each row, more gibberish at end of row)

You may have to parse data to extract the parts you care about.

(Screenshot of JSON file)

The data may be highly structured, but may not be in a row-column format that you're accustomed to. It may need to be reorganized into such a row-column form.

(Screenshot of medical data)

Sometimes the data is in a form like “Take one tablet by mouth, take one half-table with grapefruit juice”. We may need to extract pieces of that information, such as number of tablets.

(Screenshot of mySQL and MongoDB diagrams)

Data may be in structured databases, and you may need to be able to collect data from those databases in its raw form and process it.

Where is data?

Data may be on local file system, or it may be on the web, so methods are needed to obtain it no matter where it is.

One site that will be used in the course: http://open.baltimorecity.gov

The goal of this course

The pipeline for data analysis:

Raw data > Processing script > tidy data > data analysis > data communication

Many courses on data analysis tend to skip the first three steps. This course will focus on them.

Video 1-2: Raw and Processed Data

Definition of data

"Data are values of quantitative or qualitative variables, belonging to a set of items"

Qualitative: country of origin, sex, treatment Quantitative: height, weight, blood pressure

Raw versus processed data

Raw data

- The original source of the data
- Often hard to use for data analysis
- Data analysis *includes* processing
- Raw data may only need to be processed once

http://en.wikipedia.org/wiki/Raw_data

Processed data

- Data that is ready for analysis
- Processing can include merging, subsetting, transforming, etc.
- There may be standards for processing
- All steps should be recorded

http://en.wikipedia.org/wiki/Computer_data_processing

An example of a processing pipeline

(Screenshot of human genome processing machine)

Example discusses how DNA data is collected, and various samples are combined. Considering that all of this is done by a machine, I don't think this was a terribly relevant example. I guess the idea is that a data scientist had to set up the procedure.

Video 1-3: The Components of Tidy Data

The four things you should have

- The raw data
- A tidy data set
- A code book (the meta data) describing each variable, and its values in the tidy data set (including units)
- An explicit and exact recipe you used to go from 1 to 2 and 3

The raw data

- The strange binary file your measurement machine spits out
- The unformatted Excel file with 10 worksheets the company you contracted with sent you
- The complicated JSON data you got from scraping the Twitter API
- The hand-entered numbers you collected looking through a microscope

You know the raw data is in the right format if you

- Ran no software on the data
- Did not manipulate any of the numbers in the data
- Did not remove any data from the data set
- Did not summarize the data in any way

https://github.com/jtleek/datasharing

The tidy data

1. Each variable you measure should be in one column
2. Each different observation of that variable should be in a different row
3. There should be one table for each “kind” of variable
4. If you have multiple tables, they should include a column in the table that allows them to be linked

Some other important tips

- Include a row at the top of each file with variable names
- Make variable names human readable: AgeAtDiagnosis instead of AgeDx
- In general data should be saved in one file per table

https://github.com/jtleek/datasharing

The code book

1. Information about the variables (including units!) in the data set not contained in the tidy data
2. Information about the summary choices you made
3. Information about the experimental study you used

I found the phrase “not contained in the tidy data” confusing. It could refer to “information” or “variables” (maybe some variables in the raw data are dropped). I think it refers to “information”. For example, the tidy data may show numbers, but it won't show what unit those numbers are in.

“Experimental study” is an unfortunate phrase, it may be an observational study

Some other important tips

- A common format for this is Word/text file
- There should be a section called “Study design” that has a thorough description of how you collected the data
- There must be a section called “Code book” that describes each variable and its units

The instruction list

- Ideally a computer script (in R or Python)
- The input for the script is the raw data
- The output is the processed tidy data
- There are no parameters to the script (it runs the same every time)

In some cases it will not be possible to script every step. In that case you should provide instructions like:

1. Step 1–Take the raw file, run version 3.1.2 of summarie software with parameters a=1, b=2, c=3
2. Step 2–Run the software separately for each sample
3. Step 3–take column 3 of outputfile.out for each sample and that is the corresponding row in the data output

set

I.e., be as detailed as possible–err on the side of too much detail rather than not enough.

Why is the instruction list important?

Example on study of austerity programs that had serious miscalculations, which were brought to light only because the steps were clearly described. So if you want to be mocked by Colbert, be sure to follow these guidelines!

Video 1-4: Downloading files

Get/set your working directory

- A basic component of working wtih data is knowing your working directory
- The two main commands are **getwd()** and **setwd()**
- Be aware of relative vs absolute paths
    - Relative: setwd("./data"), setwd("../")
    - Absolute: setwd("/Users/jtleek/data/")
- Important difference in Windows: setwd("C:\Users\Andrew\downloads")

Checking for and creating directories

- file.exists("directoryName") will check to see if directory exists
- dir.create("directoryName") will create a directory if it doesn't exist
- Here is an example checking for a "data" directory and creating it if it doesn't exist

```
if(!file.exists("data")) {
  dir.create("data")
}
```

Getting data from the internet: download.file()

- Downloads a file from the internet
- Even if you could download by hand, using this improves reproducibility
- Important parameters are *url*, *destfile*, *method*
- Useful for downloading tab-delimited, csv and other files

My note: last point is a red herring. It is useful to download any type of file.

Example: Baltimore camera data

You can go to any site that has data and copy the URL to the data file.

```
fileUrl <- "https://data.baltimorecity.gov/api/views/dz54-
2aru/rows.csv?accesstype=DOWNLOAD"
download.file(fileUrl,  destfile="./data/cameras.csv",
method="curl")
list.files("./data")
```

```
## [1] "cameras.csv"
```

```
dateDownloaded <- date()
dateDownloaded
```

```
## [1] "Sun Jan 12 21:37:44 2014"
```

It is important to capture the date that you downloaded the data, because the data might change from time to time, so the date will allow someone else to select the appropriate set of data if they are trying to reproduce your work.

Some notes about download.file()

- If the url starts with http you can use download.file()
- If the url starts with https on Windows you may be ok
- If the url starts with https on Mac (or Linux!) you may need to add parameter method="curl"
- If the file is big, this might take a while
- Be sure to record when you download

Video 1-5: Reading local flat files

Example - Baltimore camera data

https://data.baltimorecity.gov/Transportation/Baltimore-Fixed-Speed-Cameras/dz54-2aru

Download the file to load

```
if(!file.exists("data")) {
  dir.create("data")
}
fileUrl <- "https://data.baltimorecity.gov/api/views/dz54-
2aru/rows.csv?accessType=DOWNLOAD"
download.file(fileUrl, destfile="./data/cameras.csv", method="curl")
dateDownloaded <- date()
```

Note about the above: The video does not show "./data/" at the start of destfile, which is clearly an error. It also does not show any verification that the working directory is where we want it to be, i.e., where the /data directory should exist, or be created. A truly reproducible script would probably confirm this.

Loading flat files: read.table()

- This is the main function for reading data into R
- Flexible and robust but requires more parameters
- Reads the data into RAM–big data can cause problems (i.e., run out of memory)
- Important parameters are file, header, sep, row.names, nrows
- Related: read.csv, read.csv2()

Video says it can create problems with big data "unless you read it in chunks", but no indication given of how to do that.

This function defaults to sep="\t" (tab), header=FALSE.

Baltimore example

```
cameraData <- read.table("./data/cameras.csv")
## Error: line 1 did not have 13 elements
head(cameraData)
## Error: object 'cameraData' cannot be found
```

What happened: The read.table function by default expects records to be tab-delimited, but this file is comma-delimited. So the data is not loaded, and cameraData is not defined. Also, the file has a header row listing field names.

```
cameraData <- read.table("./data/cameras.csv", sep=",", header=TRUE)
head(cameraData)
##                      address direction      street crossStreet
## 1  S CATON AVE & BENSON AVE N/B         Caton Ave Benson Ave
## 2  S CATON AVE & BENSON AVE S/B         Caton Ave Benson Ave
```

Now the head command shows the first several rows of the contents of cameraData (I include only two here).

You could also use read.csv here. It defaults to sep="," and header=TRUE, so you don't need to include those parameters (although there is no harm if you do).

```
cameraData <- read.csv("./data/cameras.csv")
head(cameraData)
##                      address direction      street crossStreet
## 1  S CATON AVE & BENSON AVE N/B         Caton Ave Benson Ave
## 2  S CATON AVE & BENSON AVE S/B         Caton Ave Benson Ave
```

Some more important parameters

- quote: you can tell R whether there are any quoted values; quote="" means no quotes (my understanding is that this can speed loading)
- na.strings: set the character that represents missing values, defaults to "NA"
- nrows: specifies how many rows to load (not couning a header row)
- skip: number of rows to skip before starting to read (not counting a header row)

In instructor's experience, the biggest trouble with reading flat files are quotation marks ' or " placed in data values; setting quote="" often resolves these. (What I think this means is that any such quote marks are just considered to be data, rather than delimiters).

Video 1-6: Reading Excel files

Excel files are still probably the most widely used format for sharing data

Download the file to load

```
if(!file.exists("data")){dir.create("data")}
fileUrl <- "https://data.baltimorecity.gov/api/views/dz54-
2aru/rows.xlsx?accessType=DOWNLOAD"
download.file(fileUrl, destfile="./data/cameras.xlsx",
method="curl")
dateDownloaded <- date()
```

read.xlsx(), read.xlsx2() {xlsx package}

(Note that first you need to install the xlsx package. There are other packages that can read Excel data.)

```
library(xlsx)
cameraData <- read.xlsx("./data/cameras.xlsx", sheetIndex=1,
header=TRUE)
head(cameraData)
                     address direction      street crossStreet
intersection
1       S CATON & BENSON AVE       N/B  Caton Ave   Benson Ave
Caton Ave & Benson Ave
2       S CATON & BENSON AVE       S/B  Caton Ave   Benson Ave
Caton Ave & Benson Ave

1 (39.2693779962, -76.66881825297)
2 (39.2693779962, -76.66881825297)
```

Note that in the above, we see two rows of results, but they can't fit on a single line, so they are each split into two.

Also note that we must specify a sheetIndex, because an Excel file can contain many sheets. header=TRUE means that the first row in the sheet will contain field names.

Reading specific rows and columns

```
colIndex <- 2:3
rowIndex <- 1:4
cameraData <- read.xlsx("./data/cameras.xlsx", sheetIndex=1,
colIndex=colIndex, rowIndex=rowIndex)
```

The above reads data from the first worksheet, only from rows 2 and 3, and columns 1 through 4. Sometimes the data in a worksheet may not be in the upper left corner (in fact there may be two "tables" in different parts of a single worksheet); sometimes we want to read only part of the data in any case.

Further notes

- The write.xlsx function will write out data to an Excel file with similar arguments
- read.xlsx2 is much faster than read.xlsx but for reading subsets of rows may be slightly unstable
- The XLConnect package has more options for writing and manipulating Excel files
- The XLConnect vignette is a good place to start for that package
- In general it is advised to store your data in either a database or in comma-separated files (.csv) or tab separated files (.tab/.txt) as they are easier to distribute.

My note: I don't see how a database is easy to distribute!

Video 1-7: Reading XML

XML

- Extensible markup language
- Frequently used to store structured data
- Particularly widely used in internet applications
- Extracting XML is the basis for most web scraping
- Components
  - Markup: labels that give the text structure
  - Content: the actual text of the document

http://en/wikipedia.org/wiki/XML

Tags, elements and attributes

- Tags correspond to general labels
  - Start tags, e.g. <section>
  - End tags, e.g., </section>
  - Empty tags, e.g., <line-break />
- Elements are specific examples of tags
  - <Greeting> Hello, world </Greeting>
- Attributes are components of the label
  - <img src="jeff.jpg" alt="instructor">
  - <step number="3"> Connect A to B. </step>

Example XML file

From: http://www.w3schools.com/xml/simple.xml

```
<!-- Edited by XMLSpy -->
<breakfast_menu>
  <food>
    <name>Belgian Waffles</name>
    <price>$5.95</price>
    <description>Two of our famous Belgian Waffles with plenty of
real maple syrup</description>
    <calories>650</calories>
  </food>
  <food>
    <name>Strawberry Belgian Waffles</name>
    <price>$7.95</price>
    <description>Light Belgian waffles covered with strawberries and
whipped cream</description>
    <calories>900</calories>
  </food>
  <food>
    <name>Berry-Berry Belgian Waffles</name>
    <price>$8.95</price>
    <description>Light Belgian waffles covered with an assortment of
fresh berries and whipped cream</description>
    <calories>900</calories>
  </food>
  ...
```

Read the file into R

```
library(XML)
fileUrl <- "http://www.w3schools.com/xml/simple.xml"
doc <- xmlTreeParse(fileUrl,useInteral=TRUE)
rootNode <- xmlRoot(doc)
xmlName(rootNode) # outputs the tag name of the document root
element
[1] "breakfast_menu"
names(rootNode) # outputs tag names of direct children of root node
  food   food  food    food   food
"food" "food" "food" "food" "food"
```

Directly access parts of the XML document

```
rootNode[[1]]
<food>
  <name>Belgian Waffles</name>
  <price>$5.95</price>
  <description>Two of our famous Belgian Waffles with plenty of real
maple syrup</description>
  <calories>650</calories>
</food>

rootNode[[1]][[1]]
<name>Belgian Waffles</name>
```

Programmatically extract parts of the file

```
xmlSApply(rootNode,xmlValue) # uses SApply recursively on children
of rootNode
"Belgian Waffles$5.95Two of our famous Belgian Waffles with plenty
of real maple syrup"
"Strawberry Belgian Waffles$7.95Light Belgian waffles covered with
strawberries and whipped cream"
...
```

So the xmlValue function returns only the text value of an XML node, not the tag name or attributes.

XPath

- /node: Top level node
- //node: Node at any level, e.g., //name finds all <name> nodes, no matter where in the document
- node[@attr-name]: Node with an attribute name, e.g., name[@length] returns <name> nodes that have an attribute named length
- node[@att-name='bob']: Node with an attriubte with a given value, e.g., name[@length='7'] returns <name> nodes with attribute length=“7”

http://www.stat.berkeley.edu/~statcur/Workshop2/Presentations/XML.pdf

My tip: If you want to learn a lot about XPath, get the O'Reilly book *XSLT* (Doug Tidwell).

Get the items on the menu and prices

```
# gets all <name> nodes within rootNode, sends them to function
xmlValue
xpathSApply(rootNode, "//name", xmlValue)
[1] "Belgian Waffles"  "Strawberry Belgain Waffles"  "Berry-Berry
Belgian Waffles"
[4] "French Toast"  "Homestyle Breakfast"

xpathSApply(rootNode, "//price", xmlValue)

[1] "$5.95" "$7.95" "$8.95" "$4.50" "$6.95"
```

Another example

http://espn.go.com/nfl/team/_/name/bal/baltimore-ravens

This is HTML, which is similar to XML (tags, attributes).

Viewing the source

Right-click on page and view source. (Screenshot of HTML from the Ravens page.)

Extract content by attributes

Up to this point, I have included no actual R code in these notes. The following is “live” and will actually run. You need to have the XML library installed for it to work.

```
library(XML)
fileUrl <- "http://espn.go.com/nfl/team/_/name/bal/baltimore-ravens"
doc <- htmlTreeParse(fileUrl, useInternal = TRUE)
scores <- xpathSApply(doc, "//li[@class='score']", xmlValue)
teams <- xpathSApply(doc, "//li[@class='team-name']", xmlValue)
scores
```

```
## list()
```

```
teams
```

```
##  [1] "San Francisco" "Dallas"        "Washington"    "New
Orleans"
##  [5] "Cincinnati"    "Pittsburgh"    "Cleveland"     "Carolina"
##  [9] "Indianapolis"  "Tampa Bay"     "Atlanta"       "Cincinnati"
## [13] "Pittsburgh"    "Tennessee"     "New Orleans"   "San Diego"
## [17] "Miami"         "Jacksonville"  "Houston"       "Cleveland"
```

Note: The scores show “list()” above because the list is empty; as of now, there are no scores on the site. Instead they're listing the 2014 schedule. An example of how things can change!

Notes and further resources

- Offical XML tutorials
  - Short: http://www.omegahat.org/RSXML/shortIntro.pdf
  - Long: http://www.omegahat.org/RSXML/Tour.pdf
- An outstanding guide to the XML package: http://www.stat.berkeley.edu/~statcur/Workshop2/Presentations/XML.pdf

Video 1-8: Reading JSON

JSON

- Javascript Object Notation
- Lightweight data storage
- Common format for data from application programming interfaces (APIs)
- Similar structure to XML but different syntax/format
- Data stored as
  - Numbers (double)
  - Strings (double quoted)
  - Boolean (true or false)
  - Array (ordered, comma separated list enclosed in square brackets [])
  - Object (unordered, commad separated collection of key:value pairs in curly brackets {})

http://en.wikipedia.org/wiki/JSON

Example JSON file

https://api.github.com/users/jtleek/repos

```
library(jsonlite)
jsonData <- fromJSON("https://api.github.com/users/jtleek/repos")
names(jsonData)
```

```
##  [1] "id"                "name"              "full_name"
##  [4] "owner"             "private"           "html_url"
##  [7] "description"       "fork"              "url"
## [10] "forks_url"         "keys_url"          "collaborators_url"
## [13] "teams_url"         "hooks_url"         "issue_events_url"
## [16] "events_url"        "assignees_url"     "branches_url"
## [19] "tags_url"          "blobs_url"         "git_tags_url"
## [22] "git_refs_url"      "trees_url"         "statuses_url"
## [25] "languages_url"     "stargazers_url"    "contributors_url"
## [28] "subscribers_url"   "subscription_url"  "commits_url"
## [31] "git_commits_url"   "comments_url"      "issue_comment_url"
## [34] "contents_url"      "compare_url"       "merges_url"
## [37] "archive_url"       "downloads_url"     "issues_url"
## [40] "pulls_url"         "milestones_url"    "notifications_url"
## [43] "labels_url"        "releases_url"      "created_at"
## [46] "updated_at"        "pushed_at"         "git_url"
## [49] "ssh_url"           "clone_url"         "svn_url"
## [52] "homepage"          "size"              "stargazers_count"
## [55] "watchers_count"    "language"          "has_issues"
## [58] "has_downloads"     "has_wiki"          "forks_count"
## [61] "mirror_url"        "open_issues_count" "forks"
## [64] "open_issues"       "watchers"          "default_branch"
```

```
names(jsonData$owner)  # shows the names in the owner node
```

```
##  [1] "login"               "id"                  "avatar_url"
##  [4] "gravatar_id"         "url"                 "html_url"
##  [7] "followers_url"       "following_url"       "gists_url"
## [10] "starred_url"         "subscriptions_url"
"organizations_url"
## [13] "repos_url"           "events_url"
"received_events_url"
## [16] "type"                "site_admin"
```

```
jsonData$owner$login  # shows the values for all nodes named owner
```

```
##  [1] "jtleek" "jtleek" "jtleek" "jtleek" "jtleek" "jtleek"
"jtleek"
##  [8] "jtleek" "jtleek" "jtleek" "jtleek" "jtleek" "jtleek"
"jtleek"
## [15] "jtleek" "jtleek" "jtleek" "jtleek" "jtleek" "jtleek"
"jtleek"
## [22] "jtleek" "jtleek" "jtleek" "jtleek" "jtleek" "jtleek"
"jtleek"
## [29] "jtleek" "jtleek"
```

Writing data frames to JSON

```
myjson <- toJSON(iris[1:3, ], pretty = TRUE)  # just show the JSON for first 3 rows
# pretty=TRUE means the file is indented for easy reading
cat(myjson)
```

```
## [
##  {
##      "Sepal.Length" : 5.1,
##      "Sepal.Width" : 3.5,
##      "Petal.Length" : 1.4,
##      "Petal.Width" : 0.2,
##      "Species" : "setosa"
##  },
##  {
##      "Sepal.Length" : 4.9,
##      "Sepal.Width" : 3,
##      "Petal.Length" : 1.4,
##      "Petal.Width" : 0.2,
##      "Species" : "setosa"
##  },
##  {
##      "Sepal.Length" : 4.7,
##      "Sepal.Width" : 3.2,
##      "Petal.Length" : 1.3,
##      "Petal.Width" : 0.2,
##      "Species" : "setosa"
##  }
## ]
```

Convert back to JSON

Convert the JSON back to data frame

```
iris2 <- fromJSON(myjson)
head(iris2)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 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
```

http://www.r-bloggers.com/new-package-jsonlite-a-smarter-json-encoderdecoder/

Further resources

- http://www.json.org/
- A good tutorial on jsonlite: http://www.r-bloggers.com/new-package-jsonlite-a-smarter-json-encoderdecoder/
- jsonlite vignette: http://cran.r-project.org/web/packages/jsonlite/vignettes/json-mapping.pdf

My note: JSON can be a difficult topic. The www.json.org site is the authoritative source in my opinion.

Video 1-9: Using data.table

data.table

- Inherits from data.frame
  - All functions that accept data.frame work on data.table
- Written in C so much faster
- Much faster at subsetting, grouping and updating

Creating data tables just like data frames

```
library(data.table)
DF = data.frame(x = rnorm(9), y = rep(c("a", "b", "c"), each = 3), z = rnorm(9))
head(DF, 3)
```

```
##           x y        z
## 1  0.324544 a -1.07157
## 2 -0.121757 a  2.05970
## 3  0.006384 a -0.08257
```

```
DT = data.table(x = rnorm(9), y = rep(c("a", "b", "c"), each = 3), z = rnorm(9))
head(DT, 3)
```

```
##           x y       z
## 1:  0.41135 a  0.9590
## 2: -0.01255 a -0.4972
## 3: -2.40613 a -0.3190
```

See all the data tables in memory

```
tables()
```

```
##      NAME NROW MB COLS  KEY
## [1,] DT      9 1  x,y,z
## Total: 1MB
```

Subsetting rows

Subsetting rows in a data table is the same as in a data frame

```
DT[2, ]
```

```
##           x y       z
## 1: -0.01255 a -0.4972
```

```
DT[DT$y == "a", ]
```

```
##           x y       z
## 1:  0.41135 a  0.9590
## 2: -0.01255 a -0.4972
## 3: -2.40613 a -0.3190
```

But with a data table, you can subset with only a single index, in which case it is assumed you are subsetting by row.

```
DT[c(2, 3)]  # return rows 2 and 3 of DT
```

```
##           x y       z
## 1: -0.01255 a -0.4972
## 2: -2.40613 a -0.3190
```

Subsetting columns!?

This is where data tables are very different from data frames.

```
DT[, c(2, 3)]
```

```
## [1] 2 3
```

(The video gives no hint why the above returns a vector with the numbers 2,3.)

Column subsetting in data.table

- The subsetting function is modified for data.table
- The argument you pass after the comma is called an “expression”
- In R an expression is a collection of statements enclosed in curly brackets

```
{
    x = 1
    y = 2
}
k = {
    print(10)
    5
}  # this prints 10, but sets k to 5 because 5 is last statement in expression
```

```
## [1] 10
```

```
print(k)  # this prints 5
```

```
## [1] 5
```

Calculating values for variables with expressions

```
DT[, list(mean(x), sum(z))]  # we pass a list of functions and specify against which
column to apply each
```

```
##          V1    V2
## 1: -0.1398 4.923
```

```
DT[, table(y)]  # this applies the table function to the column y
```

```
## y
## a b c
## 3 3 3
```

Therefore, if you pass a list as the second index, it is assumed to be a list of functions to be applied.

So how do you simply get a column? Specify its name.

```
DT[, y]
```

```
## [1] "a" "a" "a" "b" "b" "b" "c" "c" "c"
```

But what if you need the second column of the data table but don't know it's name? I don't know if this is the best technique but it works:

```
DT[, DT[[1]]]
```

```
## [1]  0.41135 -0.01255 -2.40613 -1.35985  0.60222  0.45792
-0.86639  1.73805
## [9]  0.17702
```

Adding new columns

Use := to add a new column to a data table.

```
DT[, `:=`(w, z^2)]
```

```
##            x y       z      w
## 1:  0.41135 a  0.9590 0.9197
## 2: -0.01255 a -0.4972 0.2472
## 3: -2.40613 a -0.3190 0.1018
## 4: -1.35985 b -0.8536 0.7286
## 5:  0.60222 b  0.5701 0.3250
## 6:  0.45792 b  1.5983 2.5547
## 7: -0.86639 c  0.4750 0.2256
## 8:  1.73805 c  2.0440 4.1780
## 9:  0.17702 c  0.9463 0.8955
```

Note that the above outputs DT at the same time.

When you add a new column to a data frame, R creates an entirely new copy of it. This is not the case with data table, which makes it more memory-efficient.

Also note that assigning an existing data table to a new variable does not create a new copy of the data table, as it would with data frame. Therefore, in the code below, the y column is set to 2 for all rows in both the original DT data table and the DT2 data table. In fact, they both refer to only a single data table.

```
DT2 <- DT
DT[, `:=`(y, 2)]
```

```
## Warning: Coerced 'double' RHS to 'character' to match the
column's type;
## may have truncated precision. Either change the target column to
'double'
## first (by creating a new 'double' vector length 9 (nrows of
entire table)
## and assign that; i.e. 'replace' column), or coerce RHS to
'character'
## (e.g. 1L, NA_[real|integer]_, as.*, etc) to make your intent
clear and for
## speed. Or, set the column type correctly up front when you create
the
## table and stick to it, please.
##           x y       z      w
## 1:  0.41135 2  0.9590 0.9197
## 2: -0.01255 2 -0.4972 0.2472
## 3: -2.40613 2 -0.3190 0.1018
## 4: -1.35985 2 -0.8536 0.7286
## 5:  0.60222 2  0.5701 0.3250
## 6:  0.45792 2  1.5983 2.5547
## 7: -0.86639 2  0.4750 0.2256
## 8:  1.73805 2  2.0440 4.1780
## 9:  0.17702 2  0.9463 0.8955
```

```
head(DT2, n = 3)
```

```
##           x y       z      w
## 1:  0.41135 2  0.9590 0.9197
## 2: -0.01255 2 -0.4972 0.2472
## 3: -2.40613 2 -0.3190 0.1018
```

Multiple operations

The following adds a new column, but the values assigned to it are calculated in two statements (which are separated by semicolons). First, the value of the x and z columns are added together for each row; then log_2 is calculated for that sum plus 5. The last statement in the curly brackets is the one assigned to the m column.

```
DT[, `:=`(m, {
    tmp <- (x + z)
    log2(tmp + 5)
})]
```

```
##           x y       z      w     m
## 1:  0.41135 2  0.9590 0.9197 2.671
## 2: -0.01255 2 -0.4972 0.2472 2.167
## 3: -2.40613 2 -0.3190 0.1018 1.186
## 4: -1.35985 2 -0.8536 0.7286 1.479
## 5:  0.60222 2  0.5701 0.3250 2.626
## 6:  0.45792 2  1.5983 2.5547 2.819
## 7: -0.86639 2  0.4750 0.2256 2.204
## 8:  1.73805 2  2.0440 4.1780 3.135
## 9:  0.17702 2  0.9463 0.8955 2.614
```

plyr-like operations

The following sets a to TRUE for any row in which x is greater than zero; otherwise FALSE.

```
DT[, `:=`(a, x > 0)]
```

```
##            x y       z      w     m     a
## 1:  0.41135 2  0.9590 0.9197 2.671  TRUE
## 2: -0.01255 2 -0.4972 0.2472 2.167 FALSE
## 3: -2.40613 2 -0.3190 0.1018 1.186 FALSE
## 4: -1.35985 2 -0.8536 0.7286 1.479 FALSE
## 5:  0.60222 2  0.5701 0.3250 2.626  TRUE
## 6:  0.45792 2  1.5983 2.5547 2.819  TRUE
## 7: -0.86639 2  0.4750 0.2256 2.204 FALSE
## 8:  1.73805 2  2.0440 4.1780 3.135  TRUE
## 9:  0.17702 2  0.9463 0.8955 2.614  TRUE
```

This example adds a new column, b, and sets it to the mean of the sum of the x and w columns, but groups by the a column. So rows with a=TRUE will have one value, and rows with a=FALSE will have another.

Special variables

.N is an integer of length one, containing the number of matching elements

```
set.seed(123)
DT <- data.table(x = sample(letters[1:3], 1e+05, TRUE))  # create large number of
records
DT[, .N, by = x]  # show number of rows for each value of x
```

```
##    x     N
## 1: a 33387
## 2: c 33201
## 3: b 33412
```

Keys

```
DT <- data.table(x = rep(c("a", "b", "c"), each = 100), y = rnorm(300))
setkey(DT, x)
head(DT["a"])  # assumes a key is set, so this is the same as DT[x=='a']
```

```
##    x       y
## 1: a  0.2596
## 2: a  0.9175
## 3: a -0.7223
## 4: a -0.8083
## 5: a -0.1414
## 6: a  2.2570
```

```
head(DT[x == "a"])
```

```
##    x       y
## 1: a  0.2596
## 2: a  0.9175
## 3: a -0.7223
## 4: a -0.8083
## 5: a -0.1414
## 6: a  2.2570
```

Joins

Keys can be used to faciliate joins between data tables.

```
DT1 <- data.table(x = c("a", "a", "b", "dt1"), y = 1:4)
DT2 <- data.table(x = c("a", "b", "dt2"), z = 5:7)
setkey(DT1, x)
setkey(DT2, x)
merge(DT1, DT2)
```

```
##     x y z
## 1: a 1 5
## 2: a 2 5
## 3: b 3 6
```

Fast reading

fread is much faster than read.table.

```
big_df <- data.frame(x=rnorm(1E6, y=rnorm(1E6)))
file <- tempfile()
write.table(big_df, file=file, row.names=FALSE, col.names=TRUE,
sep="\t", quote=FALSE)
system.time(fread(file))

 user  system elapsed
0.312   0.015   0.326

system.time(read.table(file, header=TRUE, sep="\t"))

 user  system elapsed
5.702   0.048   5.755
```

Summary and further reading

- The latest development version contains new functions like melt and dcast for data.tables
  - https://r-forge.r-project.org/scm/viewvc.php/pkg/NEWS?view=markup&root=datatable
- Here is a list of differences between data.table and data.frame
  - http://stackoverflow.com/questions/13618488/what-you-can-do-with-data-frame-that-you-cant-in-data-table
- Notes based on Raphael Gottardo's notes: https://github.com/raphg/Biostat-578/blob/master/Advanced_data_manipulation.Rpres