



2. More data summary & using functions

**Ken Rice
Tim Thornton**

University of Washington

Seattle, July 2018

In this session

In this session, using a larger dataset of faculty members' salaries at a **U**niformly **W**onderful institution*, we'll illustrate;

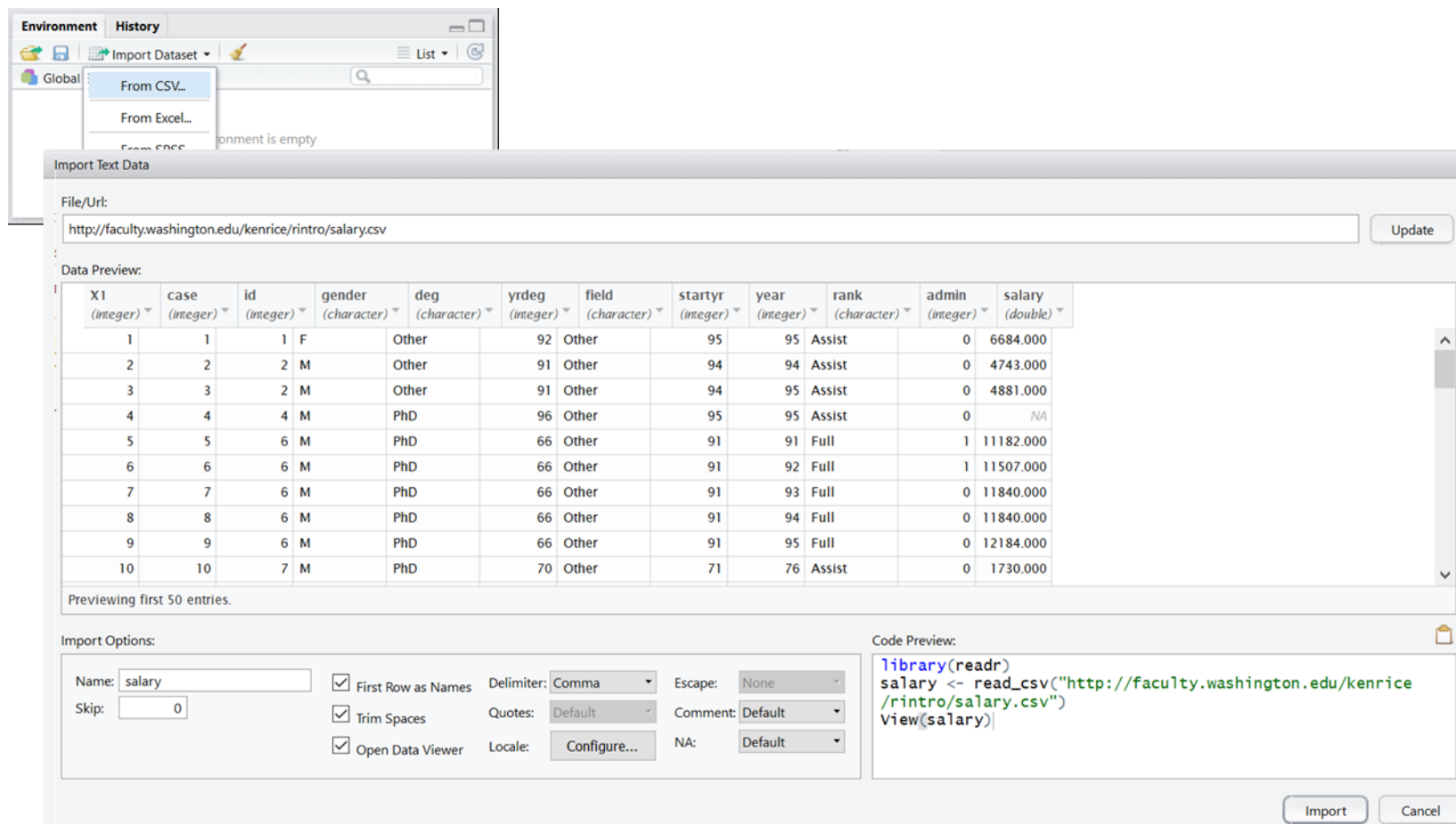
- Reading in data from the web
- More options for subsetting
- Using functions in a more flexible way
- Getting help!

* Data were collected from 1976–1995 on non-medical faculty, and include monthly salary, sex, highest degree attained, year of highest degree, field, year hired, rank, and administrative duties.

Reading in data from the web

The data live at;

<http://faculty.washington.edu/kenrice/rintro/salary.csv>



Environment History

Global

Import Dataset

From CSV...

From Excel...

From SFFS...

Environment is empty

Import Text Data

File/Url:

<http://faculty.washington.edu/kenrice/rintro/salary.csv> Update

Data Preview:

X1 (integer)	case (integer)	id (integer)	gender (character)	deg (character)	yrdeg (integer)	field (character)	startyr (integer)	year (integer)	rank (character)	admin (integer)	salary (double)
1	1	1	F	Other	92	Other	95	95	Assist	0	6684.000
2	2	2	M	Other	91	Other	94	94	Assist	0	4743.000
3	3	2	M	Other	91	Other	94	95	Assist	0	4881.000
4	4	4	M	PhD	96	Other	95	95	Assist	0	NA
5	5	6	M	PhD	66	Other	91	91	Full	1	11182.000
6	6	6	M	PhD	66	Other	91	92	Full	1	11507.000
7	7	6	M	PhD	66	Other	91	93	Full	0	11840.000
8	8	6	M	PhD	66	Other	91	94	Full	0	11840.000
9	9	6	M	PhD	66	Other	91	95	Full	0	12184.000
10	10	7	M	PhD	70	Other	71	76	Assist	0	1730.000

Previewing first 50 entries.

Import Options:

Name: salary

Skip: 0

First Row as Names

Trim Spaces

Open Data Viewer

Delimiter: Comma

Quotes: Default

Locale: Configure...

Escape: None

Comment: Default

NA: Default

Code Preview:

```
library(readr)
salary <- read_csv("http://faculty.washington.edu/kenrice/rintro/salary.csv")
View(salary)
```

Import Cancel

... note using the Update button gives the helpful preview

Reading in data from the web

This online option is very convenient but;

- Gives you a *tibble* and not a standard data frame. These are almost identical, but not quite
- Make sure you are signed into the wifi system *before* trying to access the data
- Make a local copy if you anticipate loading the data through drop-down menus multiple times – doing this is quicker and more reliable than downloading every time
- Make a local copy if you have to cut off rows above the headings – some sources put a short version of the documentation there, which the drop-down version cannot cope with
- Keep your local copy up to date!

But what if you're not using drop-down menus?

Reading in data from the web

One way to import data from the command line (or a script);

```
salary <- read.table("http://faculty.washington.edu/kenrice/rintro/salary.txt"  
                    , header=TRUE)
```

Let's break this down;

- `read.table()` is a function, that returns output and stores it in new object `salary`. (Earlier we assigned other output to new object `is.heavy`)
- `read.table()` takes *arguments*; the first is a character string giving the location of the file – the URL here, could also give the file name (in your working directory, see Session/Set Working Directory in the drop-down menus)
- The second argument (`header=TRUE`) tells R to expect a row giving the column names
- Getting either of these wrong (i.e. non-interpretable to R) will result in error messages, and no data being read in.

Functions: help!

So how do we know which arguments to provide? The help system is a huge ... help!

```
> ?read.table # then look in Help window
read.table(file, header = FALSE, sep = "", quote = "\"'",
  dec = ".", row.names, col.names,
  as.is = !stringsAsFactors,
  na.strings = "NA", colClasses = NA, nrows = -1,
  skip = 0, check.names = TRUE, fill = !blank.lines.skip,
  strip.white = FALSE, blank.lines.skip = TRUE,
  comment.char = "#",
  allowEscapes = FALSE, flush = FALSE,
  stringsAsFactors = default.stringsAsFactors(),
  fileEncoding = "", encoding = "unknown", text)
```

The arguments are also described further down the help page;

- **file**: the name of the file which the data are to be read from... can also be a complete URL
- **header**: a logical value indicating whether the file contains the names of the variables as its first line

Functions: help!

Rules for supplying arguments to functions;

- Arguments must be objects of the correct form, e.g. a data frame, or a vector, or a character string
- R assumes unnamed arguments – as in e.g. `summary(mammals)` – refer to those at the start of the help page's list
- Named arguments that follow can be from anywhere in the list
- Arguments you don't supply are assumed to follow the default value – which is *usually* sensible

Failing to supply arguments that has no defaults gives an error message – and no output.

Commonly-used arguments in commonly-used functions quickly become familiar. But because R can do so much, even experts refer to the help system all the time when coding; no-one learns every detail of every function.

Functions: help!

Two command-line ways to read in a local copy of the salary dataset;

```
myfile <- file.choose() # a function with no arguments
salary <- read.table(myfile, header=TRUE)
```

```
salary <- read.table(file.choose(), header=TRUE)
```

... the second is essentially what the GUI does. The result;

```
> str(salary)
'data.frame': 19792 obs. of 11 variables:
 $ case   : int  1 2 3 4 5 6 7 8 9 10 ...
 $ id     : int  1 2 2 4 6 6 6 6 6 7 ...
 $ gender : Factor w/ 2 levels "F","M": 1 2 2 2 2 2 2 2 2 2 ...
 $ deg    : Factor w/ 3 levels "Other","PhD",...: 1 1 1 2 2 2 2 2 2 2 ...
<some rows omitted>
 $ salary : num  6684 4743 4881 4231 11182 ...
```

R is a *language* – and like any language, it provides multiple valid ways to say the same thing. None is ‘best’, so use the way you find easiest. (We’ll discuss speed & efficiency later)

Functions: help!

Other (useful!) parts of the help system;

- **Value:** What output the function is going to return
- **Examples:** Short bits of code showing the function in action – either cut and paste or use e.g. `example("read.table")`
- **See Also:** other functions that perform related tasks

R has too big a vocabulary to list every function – which can be a problem for new users unsure what to use. We'll mention many common functions, but to find others;

- `?fn` or `help("fn")` for help on `fn`
- `help.search("topic")` for help pages related to "topic"
- `apropos("tab")` for functions whose names contain "tab"
- `RSiteSearch("FDR")` searches the R Project website (if online!)
- Your favorite search engine and/or reference book

Factors

The case and id variables are integers, i.e. whole numbers. As we saw with the mammals' numeric data, these can be added, multiplied, exponentiated, compared etc.

The gender and deg columns are columns of *Factor* variables – this is R's term for categorical variables (e.g. hair color, nationality, soprano/alto/tenor/bass)

```
> table(salary$deg)
Other  PhD  Prof
 1640 16806 1346
> table(salary$gender, salary$deg)
      Other  PhD  Prof
F    569  3220  137
M   1071 13586 1209
> table(salary$deg == "Prof")
FALSE  TRUE
18446  1346
> (salary$deg == "Prof")[1:10]
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
> levels(salary$rank)          # default alpha-numeric ordering
[1] "Assist" "Assoc"  "Full"
```

Factors

What did that show?

- `table()` crosstabulates all the variables you pass as arguments
- The 'double equals' `==` indicates equality (*exact* equality)
- Used to compare `salary$deg` (length 19,792) and `"Prof"` (length 1), the second vector gets recycled until it's as long as the first
- Factors have *levels* – and you can see what they are

As you might imagine, factors can't be added;

```
> salary$deg[1:10] + 4.2
[1] NA NA NA NA NA NA NA NA NA NA
Warning message:
In Ops.factor(salary$deg[1:10], 4.2) : + not meaningful for factors
```

This is a Warning – R produces output, unlike an Error which gives just a message (at best!) Either way, check the code does what you intended, before going any further.

Operating on data: subsets again

In the previous session (the mammals example) we saw how to make subsets by;

- Selecting numbered rows/columns of interest
- Selecting rows/columns corresponding to TRUEs, in vector(s) of logicals
- Selecting rows/columns by their names

There is also a `subset()` command, that returns a new data frame object – with elements that are a subset of the old one;

```
> oldprofdata <- subset(salary, rank=="Full" & year<83)
> table(oldprofdata$gender)
  F    M
99 1542
```

The first line translates as ‘make a subset of the salary data frame, using the rows where evaluating `rank=="Full"` AND `year<83` in the salary data frame returns TRUE’.

Operating on data: subsets again

Having made this subset (however!), you might be surprised at this;

```
> summary(oldprofdata$rank)
Assist  Assoc  Full
      0      0  1641
> table(oldprofdata$rank)
Assist  Assoc  Full
      0      0  1641
> levels(oldprofdata$rank)
[1] "Assist" "Assoc"  "Full"
```

If you want to drop unused factor levels;

```
> oldprofdata <- droplevels(oldprofdata) # overwrites original version
> levels(oldprofdata$rank)
[1] "Full"
```

You can also change level names with e.g.

```
> levels(oldprofdata$field)
[1] "Arts"  "Other" "Prof"
> levels(oldprofdata$field) <- c("Arts", "Other", "Law'n'Med")
```

Operating on data: subsets again

Yet another way to operate on data frames – or subsets of them;

```
> with(salary, table(gender, rank))
```

```
      rank
gender Assist Assoc Full
  F      1460   1465 1001
  M      2588   5064 8210
```

```
> with( subset(salary, rank=="Full" & year<83), table(gender, rank))
```

```
      rank
gender Assist Assoc Full
  F         0     0   99
  M         0     0 1542
```

```
> with( droplevels(subset(salary, rank=="Full" & year<83)), table(gender, rank))
```

```
      rank
gender Full
  F     99
  M 1542
```

`with()` temporarily sets up a data frame as the default place to look up variables. This means you can then execute commands (like `table(gender, rank)`) without having to tell R where to find `gender` and `rank`. It's also easier to read code without `$`'s everywhere.

Operating on data: with Logic!

To make the subset, we used `&` as a logical AND. Similarly;

- `|` denotes logical OR
- `!` denotes negation; `!TRUE` is `FALSE` and `!FALSE` is `TRUE`
- `==` denotes exact equality (as before)
- `!=` Not equal to
- `>=` Greater than or equal to; see also `>`, `<`, `<=`
- `%in%` Are elements of the first vector in the second?

An example of `%in%`; (for details on the others, see `?Logic`)

```
> letters %in% c("t","i","m")
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE
[13]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
[25] FALSE FALSE
> (1:26)[letters %in% c("t","i","m")]
[1]  9 13 20
```

You *can* use a single-equals sign (`=`) to denote assignment (previously `<-`), e.g. `one.to.ten = 1:10`. But it's *far* too easy to mix this up with `==`, and the help system uses the arrow

Operating on data: missing values

Here's the last line of the summary of the full dataset;

```
      salary
Min.   : 1200
1st Qu.: 3287
Median : 4353
Mean   : 4722
3rd Qu.: 5794
Max.   :14464
NA's   :4
```

- NA is R's code for missing data - so there are 4 entries here where the monthly salary is missing
- Missing data is important for analysis!
- If your *data* doesn't use NA, see the `na.strings` argument in `read.table()` to tell R this
- ... or re-assign elements of vectors, e.g.

```
salary$salary <- ifelse(salary$salary == -99, NA, salary$salary)
```


Operating on data: missing values

Formally NA is short for 'Not Available', but it's better to think of it as "Don't Know". Try it in the following situations;

- `42 + NA`: What's 42 plus a number you don't know?
- `TRUE & NA`: Are TRUE & an unknown logical both TRUE?
- `FALSE & NA`: Are FALSE & an unknown logical TRUE?
- `mean(c(1,2,75,NA))`: What's the mean of 1,2,75 and a number you don't know?
- `x == NA`: Is x equal to a number you don't know?

So how did we get the mean earlier? R's mean for a `summary()` of a data frame is *slightly* different from 'plain vanilla' `mean()`;

```
> mean(salary$salary)
[1] NA
> mean(salary$salary, na.rm=TRUE) # na.rm's default is FALSE, in many functions
[1] 4721.712
```

R distinguishes NA from NaN ('not a number', e.g. `sqrt(-1)`) and Infinity (e.g. `1/0`). Also note `is.na(x)` returns TRUE/FALSE.

But what about tibbles? (optional)

Data frames are standard for most users – and the help system – while RStudio’s ‘tibbles’ are not. To read in using RStudio’s menu (and preview) and then convert to standard data frames;

```
> library("readr") # loads a package - more on this later
> saltib <- read_csv("http://faculty.washington.edu/kenrice/rintro/salary.csv")
> saltib[1:2,1:4]
# A tibble: 2 x 4
   X1 case id gender
<int> <int> <int> <chr>
1     1     1     1     F
2     2     2     2     M
> saldf <- as.data.frame(saltib)
> saldf[ 1:2,1:4]
  X1 case id gender
1  1     1  1     F
2  2     2  2     M # or use class() to check more directly
```

- You can convert the other way with `as.tibble()`, in the `tibble` package.
- Most users start with drop-down menus, then quickly get used to scripts – making this headache go away

Summary

- Data can live on the web too
- R uses functions; these have arguments, which have names and (often) default values
- The help system is essential to use arguments correctly – but there are multiple correct ways to code individual tasks
- Factors are treated slightly differently from numbers
- Remember NA is 'Don't Know', to understand what will happen with missing values