SISG & SISMID

Module 3: Introduction to R

Ken Rice
Tim Thornton

University of Washington

Seattle, July 2013

Introduction: Course Aims

This is a first course in R; we aim to cover;

- Reading in, summarizing & managing data
- Use of functions in R – doing jobs by programming, not by using drop-down menus (much)
- *Some* standard functions for statistical analysis – but minimal statistics in this module
- How to use other people's code, how to get help, what to learn next

We assume no previous use of R, also non-extensive programming skills in other languages. If this is *not* your level, please consider switching to a later module.

Introduction: Resources

Most importantly, the class site is

http://faculty.washington.edu/kenrice/rintro

Contains (or will contain);

- PDF copies of slides (in color, and contains a few hyperlinks)
- All datasets needed for exercises
- Exercises for you to try
- Our solutions to exercises
- Links to other software, other courses, book, and places to get R help
- Links to a few helpful websites/email list archives

Of course, any search engine will find much more than this, and can be useful first step in finding how to tackle a problem in R.

Introduction: About Tim

- Assistant Prof, UW Biostat
- A useR and an instructoR
- Research in Genetic Epidemiology for Complex Human Traits
- Spare time: teaching my 2 and 4 year old how to swim!

Introduction: About Ken

- Associate Prof, UW Biostat
- AuthoR of a few R packages, useR, teacheR
- Genetic/Genomic research in Cardiovascular Epidemiology
- Sings bass (July 17 & 18!)

... and you?

(who are you, what genetics/infectious disease you work on)

Introduction: Course structure

10 sessions over 2.5 days

- Day 1; (Mostly RStudio) Data management, using functions
- Day 2; (Standard R) More about programming
- Day 2.5; More advanced ideas

Web page: `http://faculty.washington.edu/kenrice/rintro/`

Introduction: Session structure

What to expect in a typical session;

- 45 mins teaching (please interrupt!)
- 30 mins hands-on; please work in pairs
- 15 mins summary, discussion (interrupt again!)

There will also be one ‘take-home’ exercise, on Day 2; the final session will include in-depth discussion/evaluation.

1. Reading in data

Ken Rice
Tim Thornton

University of Washington

Seattle, July 2013

What is R?

R is a ‘programming environment for statistics and graphics’

- Does *basically* everything, can also be extended
- Default when statisticians implement new methods
- Free, open-source

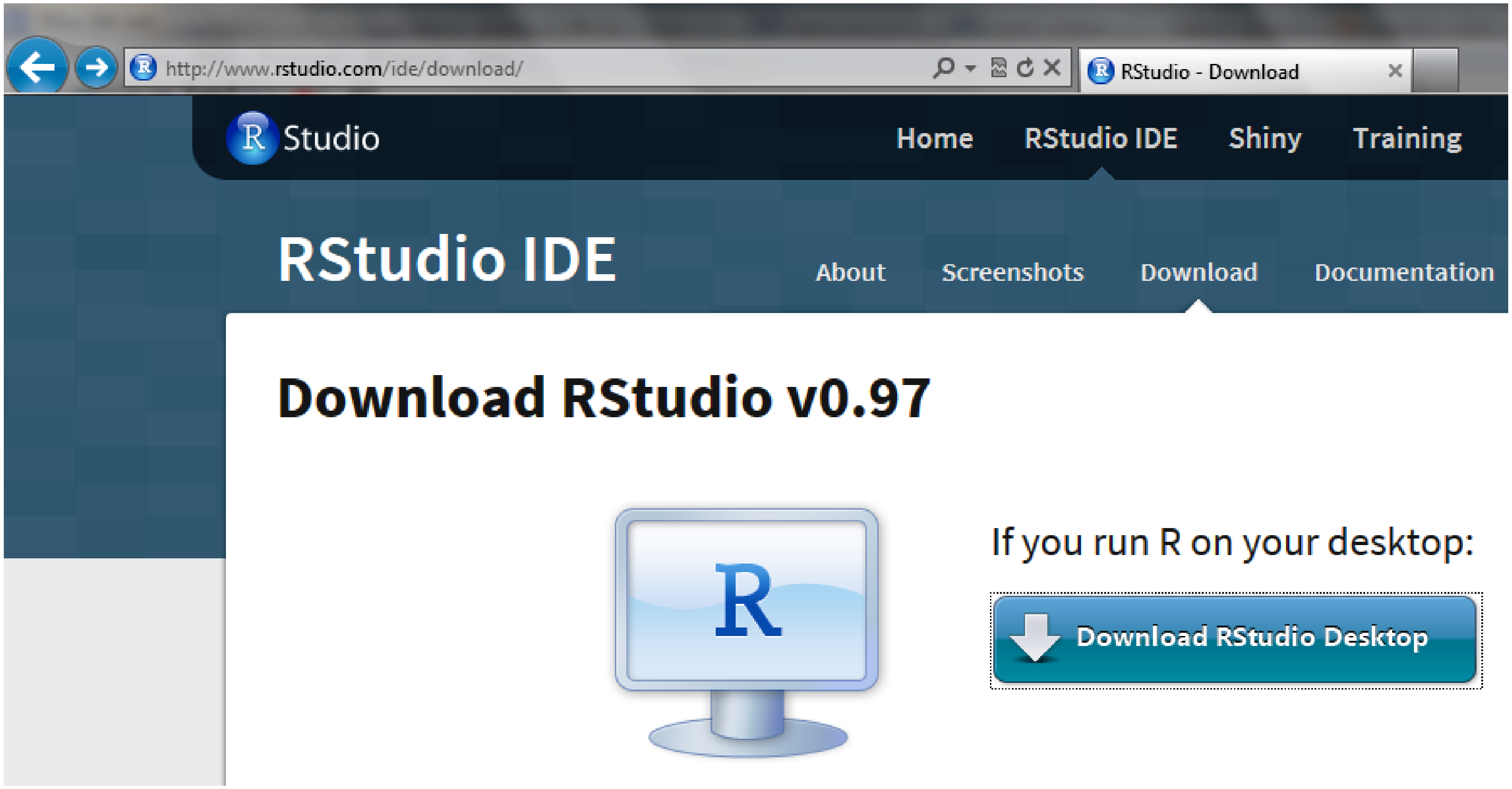
But;

- Steeper learning curve than e.g. Excel, Stata
- Command-line driven (programming, not drop-down menus)
- Gives only what you ask for!

To help with these difficulties, we will begin with RStudio, a front-end for R that is slightly more user-friendly than the default.

RStudio

In your favorite web browser, download from rstudio.com;



- Select, download & install version for your (desktop) system
- Default installation is fine
- Working in pairs *highly* recommended

RStudio

We'll use the 'Console' window first – as a (fancy!) calculator

```
> 2+2
[1] 4
> 2^5+7
[1] 39
> 2^(5+7)
[1] 4096
> exp(pi)-pi
[1] 19.9991
> log(20+pi)
[1] 3.141632
> 0.05/1E6   # a comment; note 1E6 = 1,000,000
[1] 5e-08
```

- All common math functions are available; parentheses (round brackets) work as per high school math
- Try to get used to bracket matching. A '+' prompt means the line isn't finished – hit Escape to get out, then try again.

RStudio

R stores data (and everything else) as *objects*. New objects are created when we *assign* them values;

```
> x <- 3
> y <- 2 # now check the Workspace window
> x+y
[1] 5
```

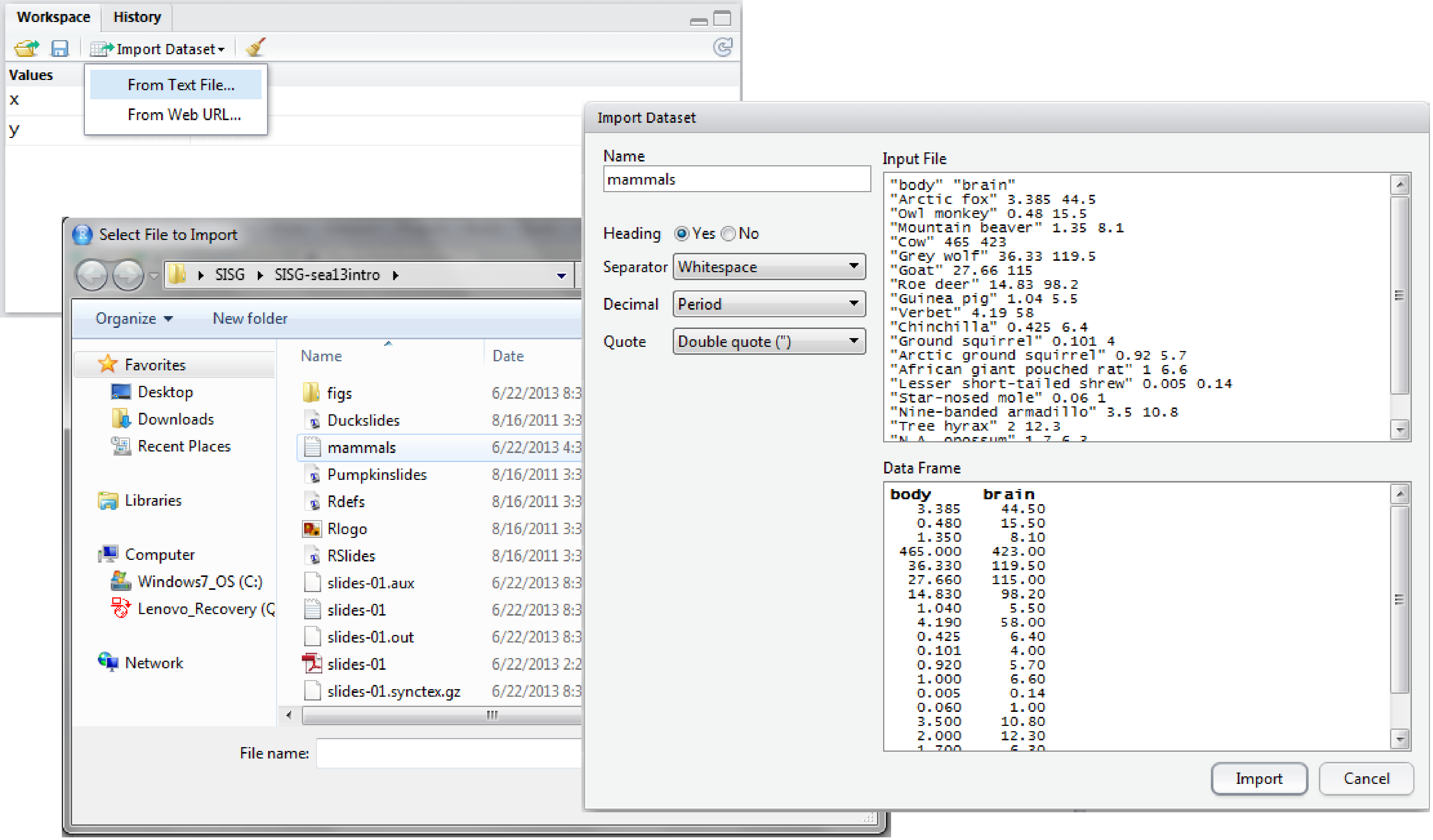
Assigning new values to existing objects over-writes the old version – and be aware there is no ‘undo’;

```
> y <- 17.4 # check the Workspace window again
> x+y
[1] 20.4
```

- Anything after a hash (#) is ignored – e.g. comments
- Spaces don’t matter
- Capital letters *do* matter

RStudio: Reading in data

To import a dataset, follow pop-ups from the Workspace tab;



Name	Date
figs	6/22/2013 8:3
Duckslides	8/16/2011 3:3
mammals	6/22/2013 4:3
Pumpkinslides	8/16/2011 3:3
Rdefs	8/16/2011 3:3
Rlogo	8/16/2011 3:3
RSlides	8/16/2011 3:3
slides-01.aux	6/22/2013 8:3
slides-01	6/22/2013 8:3
slides-01.out	6/22/2013 8:3
slides-01	6/22/2013 2:2
slides-01.synctex.gz	6/22/2013 8:3

Name: mammals

Heading ● Yes ○ No

Separator: Whitespace

Decimal: Period

Quote: Double quote (")

Input File

```
"body" "brain"
"Arctic fox" 3.385 44.5
"Owl monkey" 0.48 15.5
"Mountain beaver" 1.35 8.1
"Cow" 465 423
"Grey wolf" 36.33 119.5
"Goat" 27.66 115
"Roe deer" 14.83 98.2
"Guinea pig" 1.04 5.5
"Verbet" 4.19 58
"Chinchilla" 0.425 6.4
"Ground squirrel" 0.101 4
"Arctic ground squirrel" 0.92 5.7
"African giant pouched rat" 1 6.6
"Lesser short-tailed shrew" 0.005 0.14
"Star-nosed mole" 0.06 1
"Nine-banded armadillo" 3.5 10.8
"Tree hyrax" 2 12.3
"N.A. opossum" 1.7 6.3
```

Data Frame

```
body      brain
   3.385    44.50
   0.480    15.50
   1.350     8.10
 465.000   423.00
  36.330   119.50
  27.660   115.00
  14.830    98.20
   1.040     5.50
   4.190    58.00
   0.425     6.40
   0.101     4.00
   0.920     5.70
   1.000     6.60
   0.005     0.14
   0.060     1.00
   3.500    10.80
   2.000    12.30
   1.700     6.30
```

RStudio: Reading in data

More on those options;

- **Name**: Name of the data frame object that will store the whole dataset

- **Separator**: what's between items on a single line?

- **Decimal**: Usually a period (".")

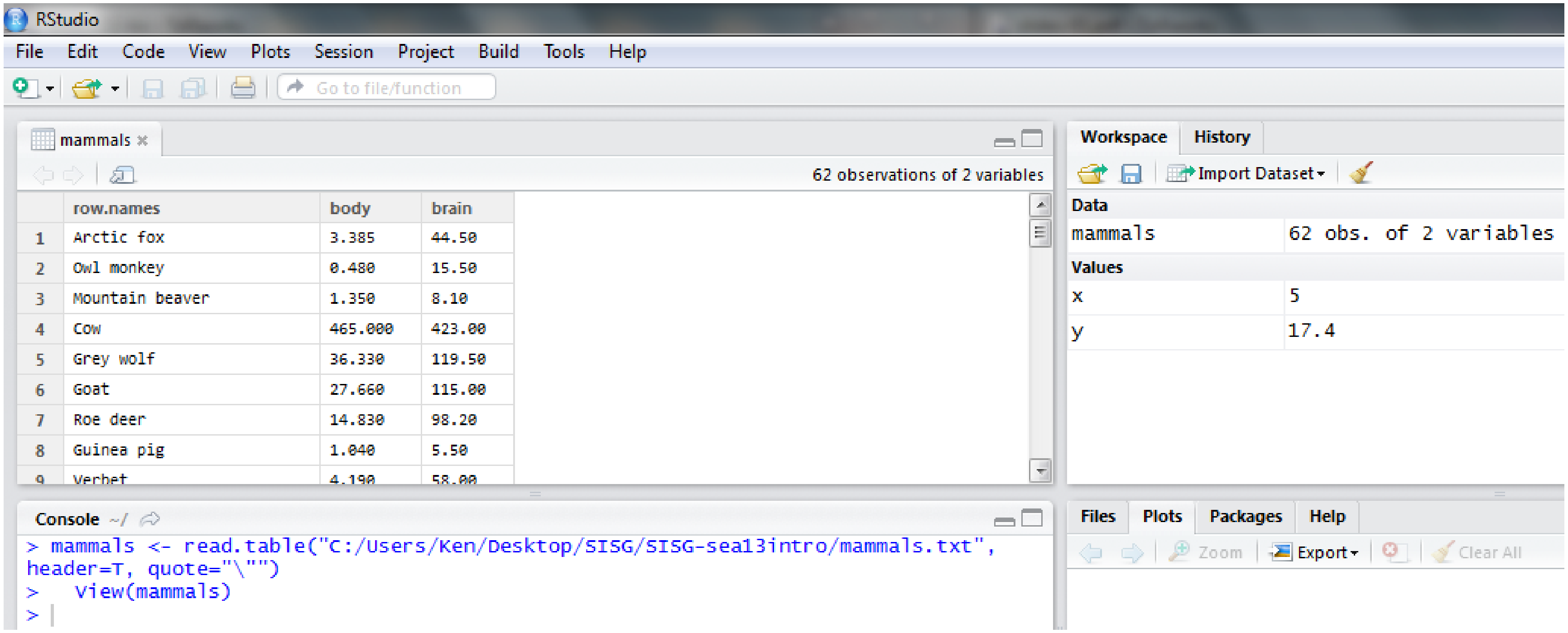
- **Quote**: Usually double – seldom critical

The defaults are sensible, but R assumes you *know* what your data *should* look like – and whether it has named columns, row names etc.

RStudio: Reading in data

After successfully reading in the data;

- The workspace now includes a `mammals` object – or whatever you called the data read from file
- A copy of the data can be examined in the Excel-like data viewer (below) – if it looks weird, find out why & fix it!


RStudio
File Edit Code View Plots Session Project Build Tools Help
Go to file/function
mammals
62 observations of 2 variables
Workspace History
Import Dataset
Console ~/
Files Plots Packages Help
Zoom
Export
Clear All

	row.names	body	brain
1	Arctic fox	3.385	44.50
2	Owl monkey	0.480	15.50
3	Mountain beaver	1.350	8.10
4	Cow	465.000	423.00
5	Grey wolf	36.330	119.50
6	Goat	27.660	115.00
7	Roe deer	14.830	98.20
8	Guinea pig	1.040	5.50
9	Vervet	4.190	58.00

Data	
mammals	62 obs. of 2 variables
Values	
x	5
y	17.4

```
> mammals <- read.table("C:/Users/Ken/Desktop/SISG/SISG-sea13intro/mammals.txt",
header=T, quote="\"")
>   View(mammals)
> |
```

... we'll return later, to `read.table` in the Console window

RStudio: Reading in data

What's a good name for my new object?

- Something memorable (!) and not easily-confused with other objects, e.g. `X` isn't a good choice if you already have `x`
- Names must start with a letter or period (". "), after that any letter, number or period is okay
- Avoid other characters; they get interpreted as math ("-","*") or are hard to read ("_") so should not be used in names
- Avoid names of existing functions – e.g. `summary`. Some one-letter choices (`c`, `C`, `F`, `t`, `T` and `S`) are already used by R as names of functions, it's best to avoid these too

Operating on data

To operate on data, type commands in the Console window, just like our earlier calculator-style approach;

```
> str(mammals)
'data.frame': 62 obs. of  2 variables:
 $ body : num  3.38 0.48 1.35 465 36.33 ...
 $ brain: num  44.5 15.5 8.1 423 119.5 ...
> summary(mammals)
      body               brain        
 Min.   :   0.005   Min.   :   0.14  
 1st Qu.:   0.600   1st Qu.:   4.25  
 Median :   3.150   Median :  17.25  
 Mean   : 198.738   Mean   : 283.13  
 3rd Qu.:  48.203   3rd Qu.: 166.00  
 Max.   :6654.000   Max.   :5712.00  
```

- `str()` tells us the `str`ucture of an object
- `summary()` summarizes the object

Can also use these commands on the single numbers we created earlier (try it!)

Operating on data: columns

Individual columns in data frames are identified using the $ symbol – just seen in the `str()` output.

```
> mammals$brain
 [1]   44.50   15.50    8.10  423.00  119.50  115.00   98.20    5.50   58.00
[10]    6.40    4.00    5.70    6.60    0.14    1.00   10.80   12.30    6.30
[19] 4603.00    0.30  419.00  655.00    3.50  115.00   25.60    5.00   17.50
[28]  680.00  406.00  325.00   12.30 1320.00 5712.00    3.90  179.00   56.00
[37]   17.00    1.00    0.40    0.25   12.50  490.00   12.10  175.00  157.00
[46]  440.00  179.50    2.40   81.00   21.00   39.20    1.90    1.20    3.00
[55]    0.33  180.00   25.00  169.00    2.60   11.40    2.50   50.40
> summary(mammals$brain)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   0.14    4.25   17.25  283.10  166.00 5712.00
```

Think of $ as 'apostrophe-S', i.e. `mammals`'S `brain`.

Unlike many other statistical packages, R can handle *multiple* datasets at the same time – helpful if your data are e.g. phenotypes & genotypes, or county & disease outbreak data. This isn't possible without $, or some similar bits of syntax.

Operating on data: columns

New columns are created when you assign their values – here containing the brain weights in kilograms;

```
> mammals$brainkg <- mammals$brain/1000
> str(mammals)
'data.frame': 62 obs. of  3 variables:
 $ body   : num  3.38 0.48 1.35 465 36.33 ...
 $ brain  : num  44.5 15.5 8.1 423 119.5 ...
 $ brainkg: num  0.0445 0.0155 0.0081 0.423 0.1195 ...
> summary(mammals$brainkg)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
0.00014 0.00425 0.01725 0.28310 0.16600 5.71200
```

- Assigning values to existing columns over-writes existing values – again, with no warning
- With e.g. `mammals$newcolumn <- 0`, the new column has every entry zero; R *recycles* this single value, for every entry
- It's unusual to delete columns... but if you *must*; `mammals$brainkg <- NULL`

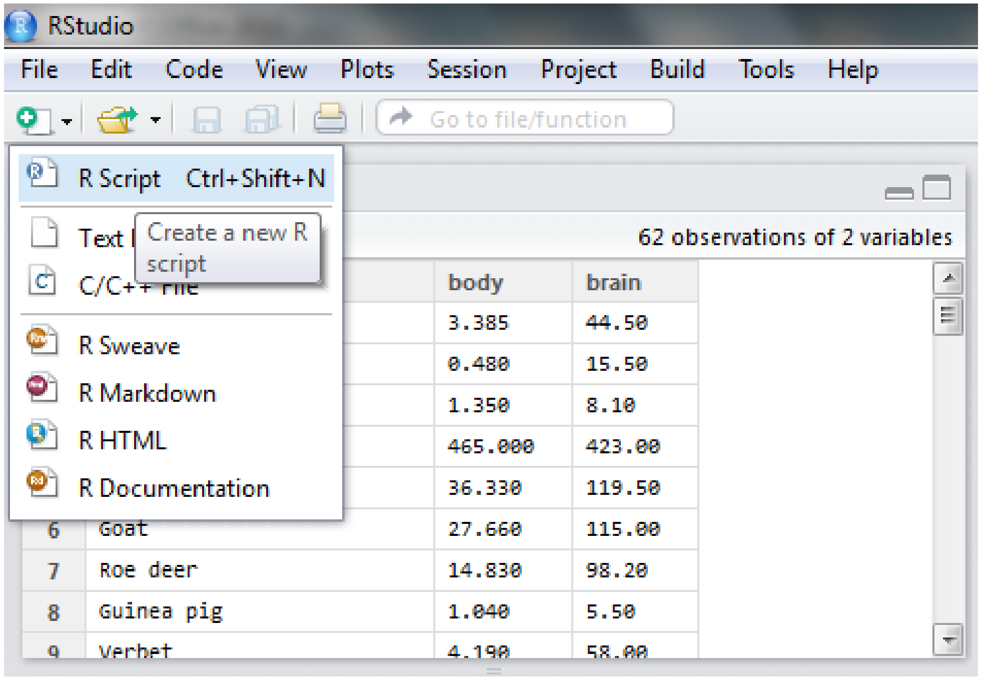
Operating on data: columns

Other functions useful for summarizing data frames, and their columns;

```
> names(mammals)
[1] "body"  "brain"
> dim(mammals)      # dim is short for dimension
[1] 62  2
> length(mammals$body)
[1] 62
> min(mammals$body)
[1] 0.005
> max(mammals$body)
[1] 6654
> range(mammals$body)
[1]    0.005 6654.000
> mean(mammals$brain)
[1] 283.1342
> sd(mammals$brain) # sd is short for standard deviation
[1] 930.2789
> median(mammals$brain)
[1] 17.25
> median(mammals$br) # uses pattern-matching (but hard to debug later)
[1] 17.25
```

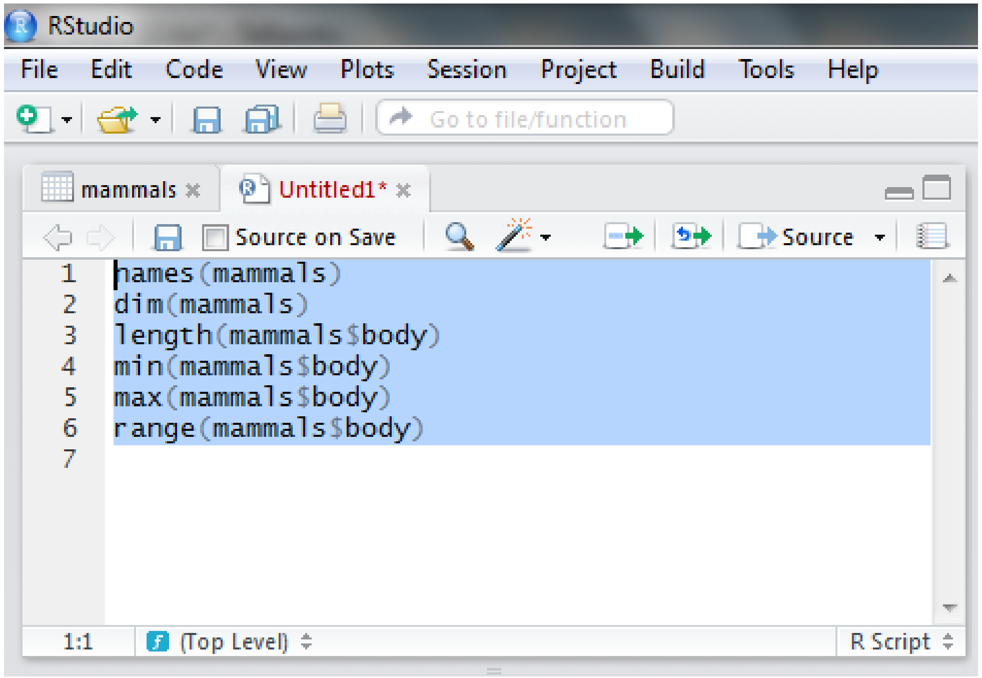
RStudio: the Script window

It's fine for occasional use, but entering *every* command 'by hand' is error-prone, and quickly gets tedious. A *much* better approach is to use a Script window – open one with Ctrl-Shift-N, or the drop-down menus;



62 observations of 2 variables

		body	brain
		3.385	44.50
		0.480	15.50
		1.350	8.10
		465.000	423.00
		36.330	119.50
6	Goat	27.660	115.00
7	Roe deer	14.830	98.20
8	Guinea pig	1.040	5.50
9	Verbet	4.190	58.00



```
names(mammals)
dim(mammals)
length(mammals$body)
min(mammals$body)
max(mammals$body)
range(mammals$body)
```

- Opens a nice editor, enables saving code (.R extension)
- Run current line (or selected lines) with Ctrl-Enter, or Ctrl-R

RStudio: the Script window

An important notice: from now on, we assume you are using a script editor.

- First-time users tend to be reluctant to switch! – but it's worth it, ask any experienced user
- Some code in slides may be formatted for cut-and-paste into scripts – it may not look exactly like what appears in the Console window
- Exercise 'solutions' given as .R files
- Scripts make it easy to run slightly modified code, without re-typing everything – remember to save them as you work
- Also remember the Escape key, if e.g. your bracket-matching goes wrong

For a very few jobs, we will still use drop-down menus, e.g. changing directories – though commands *are* available.

Operating on data: subsets

To identify general subsets – not just the columns selected by $ – R uses square brackets.

Selecting individuals elements;

```
> mammals$brain[32] # 32nd element of mammals$brain
[1] 1320
> row.names(mammals)[32]
[1] "Human"
> mammals$body[32]
[1] 62
> mammals[32,2] # 32nd row, 2nd column
[1] 62
```

Selecting entire columns (again!) or entire rows, blank entries indicate you want everything.

```
> mammals[32,]     # everything in the 32nd row
      body brain
Human   62  1320
> sum(mammals[32,])
[1] 1382
```

Operating on data: subsets

Suppose we were interested in the brain weight (i.e 2nd column) for mammals (i.e. rows) 14, 55, & 61. How to select these multiple elements?

```
> mammals[c(14,55,61),1]
[1] 0.005 0.048 0.104   # check these against data view
```

But what is `c(14,55,61)`? It's a *vector* of numbers – `c()` is for *combine*;

```
> length(c(14,55,61))
[1] 3
> str(c(14,55,61))
 num [1:3] 14 55 61
```

We can select these rows and all the columns;

```
> mammals[c(14,55,61),]
                           body brain
Lesser short-tailed shrew 0.005  0.14
Musk shrew                0.048  0.33
Tree shrew                0.104  2.50
```

Operating on data: subsets

A very useful special form of vector;

```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
> 6:2
[1] 6 5 4 3 2
> -1:-3
[1] -1 -2 -3
```

R expects you to know this shorthand – see e.g. its use of `1:3` in the output on the previous slide. For a ‘rectangular’ selection of rows and columns;

```
> mammals[20:22, 1:2]
                 body brain
Big brown bat   0.023   0.3
Donkey        187.100 419.0
Horse         521.000 655.0
```

Negative values correspond to dropping those rows/columns;

```
> mammals[-3:-62, 1:2] # everything but the first two rows, & columns 1:2
            body brain
Arctic fox 3.385  44.5
Owl monkey 0.480  15.5
```

Operating on data: subsets

As well as storing numbers and character strings (like `"Donkey"`, `"Big brown bat"`) R can also store *logicals* – TRUE and FALSE.

To make a new vector, with elements that are TRUE if body mass is above 500kg and FALSE otherwise;

```
> is.heavy <- mammals$body > 500
> table(is.heavy)  # another useful data summary command
is.heavy
FALSE  TRUE
   58     4
```

Which mammals were these? (And what were their masses?)

```
> mammals[is.heavy,]          # just the rows for which is.heavy is TRUE
                 body brain
Asian elephant   2547  4603
Horse             521   655
Giraffe           529   680
African elephant 6654  5712
> mammals[is.heavy,2]     # combining TRUE/FALSE (rows) and numbers (columns)
[1] 4603  655  680 5712
```

Operating on data: subsets

One final method... for now!

Instead of specifying rows/columns of interest by number, or through vectors of TRUEs/FALSEs, we can also just give the names – as *character strings*, or vectors of character strings.

```
> mammals[c("Cow","Goat","Human"),"body"]
[1] 465.00  27.66  62.00
> mammals[c("Cow","Goat","Human"),c("body","brain")]
        body brain
Cow   465.00   423
Goat   27.66   115
Human  62.00  1320
> mammals[c("Cow","Goat","Human"),2] # okay to mix & match
[1]  423  115 1320
```

– this is more typing than the other options, but much easier to debug/reuse.

Quitting time (almost)

When you're finished with RStudio;

- Ctrl-Q, or the drop-down menus, or entering `q()` at the command line all start the exit process
- You will be asked "Save workspace image to ~/.RData?"
  - No/Don't Save: nothing is saved, and is not available when you re-start. *This is recommended*, because you will do different things in each session
  - Yes: Everything in memory is stored in R's internal format (.Rdata) and will be available when you re-start RStudio
  - Cancel: don't quit, go back
- Writing about what you did (output from a script) often takes much longer than re-running that script's analyses

To get rid of *objects* in your current session, use `rm()`, e.g. `rm(is.heavy, mammals, x, y)` ... or RStudio's 'broom' button.

Summary

- In RStudio, read in data from pop-up menu in Workspace window

- Data frames store data; can have many of these objects, and others

- Identify vectors with $, subsets with square brackets

- Many useful summary functions are available, with sensible names

- Scripts are an important drudgery-avoidance tool!

2. More data summary & using functions

Ken Rice
Tim Thornton

University of Washington

Seattle, July 2013

In this session

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

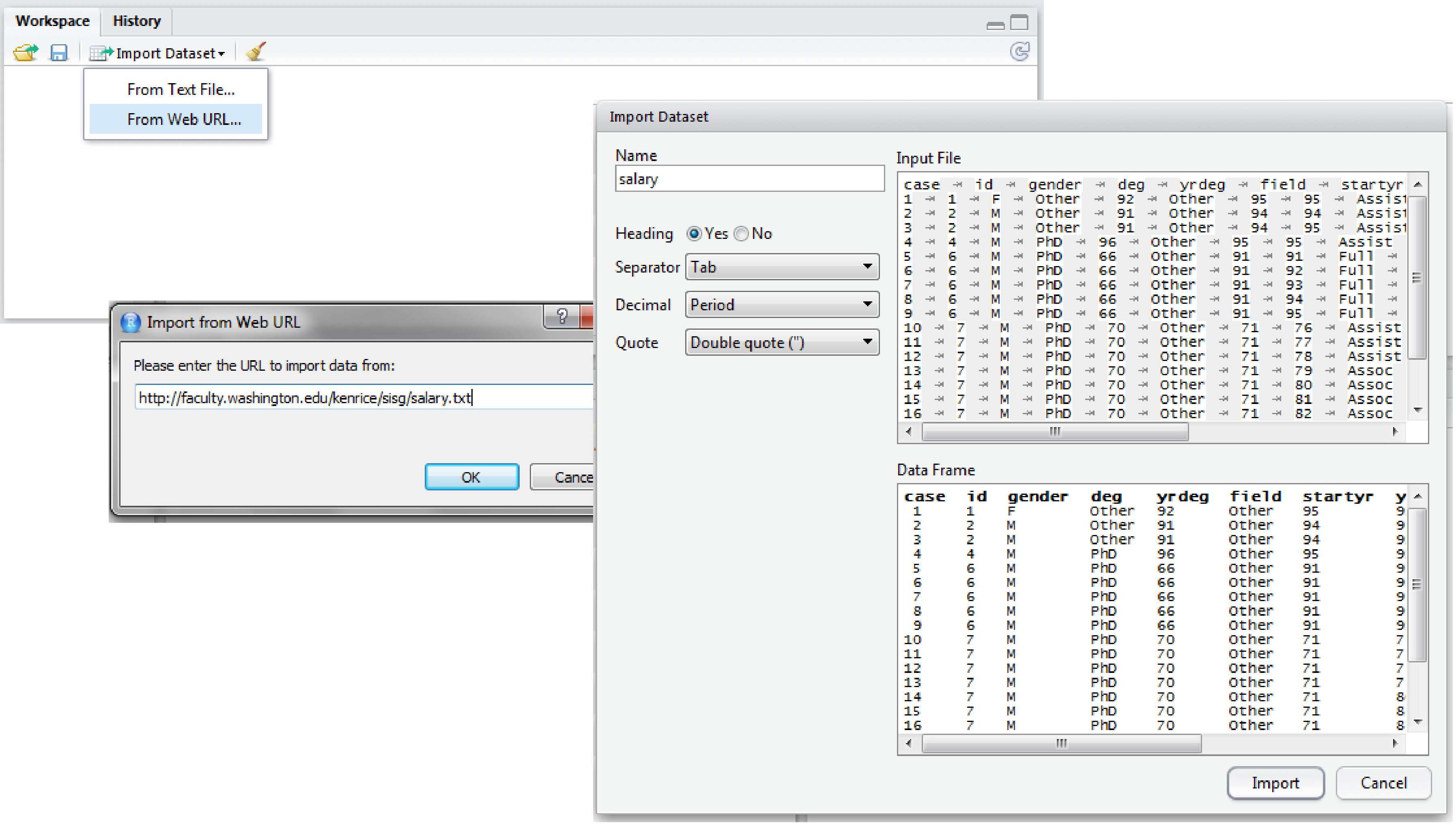
- Reading in data from the web
- More options for subsetting
- Using functions in a more flexible way
- Getting help!

* 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.txt



Please enter the URL to import data from:

http://faculty.washington.edu/kenrice/sisg/salary.txt

Name: salary

Heading ● Yes ○ No

Separator: Tab

Decimal: Period

Quote: Double quote (")

Input File

```
case ⇥ id ⇥ gender ⇥ deg ⇥ yrdeg ⇥ field ⇥ startyr
1 ⇥ 1 ⇥ F ⇥ Other ⇥ 92 ⇥ Other ⇥ 95 ⇥ 95 ⇥ Assist
2 ⇥ 2 ⇥ M ⇥ Other ⇥ 91 ⇥ Other ⇥ 94 ⇥ 94 ⇥ Assist
3 ⇥ 2 ⇥ M ⇥ Other ⇥ 91 ⇥ Other ⇥ 94 ⇥ 95 ⇥ Assist
4 ⇥ 4 ⇥ M ⇥ PhD ⇥ 96 ⇥ Other ⇥ 95 ⇥ 95 ⇥ Assist
5 ⇥ 6 ⇥ M ⇥ PhD ⇥ 66 ⇥ Other ⇥ 91 ⇥ 91 ⇥ Full ⇥
6 ⇥ 6 ⇥ M ⇥ PhD ⇥ 66 ⇥ Other ⇥ 91 ⇥ 92 ⇥ Full ⇥
7 ⇥ 6 ⇥ M ⇥ PhD ⇥ 66 ⇥ Other ⇥ 91 ⇥ 93 ⇥ Full ⇥
8 ⇥ 6 ⇥ M ⇥ PhD ⇥ 66 ⇥ Other ⇥ 91 ⇥ 94 ⇥ Full ⇥
9 ⇥ 6 ⇥ M ⇥ PhD ⇥ 66 ⇥ Other ⇥ 91 ⇥ 95 ⇥ Full ⇥
10 ⇥ 7 ⇥ M ⇥ PhD ⇥ 70 ⇥ Other ⇥ 71 ⇥ 76 ⇥ Assist
11 ⇥ 7 ⇥ M ⇥ PhD ⇥ 70 ⇥ Other ⇥ 71 ⇥ 77 ⇥ Assist
12 ⇥ 7 ⇥ M ⇥ PhD ⇥ 70 ⇥ Other ⇥ 71 ⇥ 78 ⇥ Assist
13 ⇥ 7 ⇥ M ⇥ PhD ⇥ 70 ⇥ Other ⇥ 71 ⇥ 79 ⇥ Assoc
14 ⇥ 7 ⇥ M ⇥ PhD ⇥ 70 ⇥ Other ⇥ 71 ⇥ 80 ⇥ Assoc
15 ⇥ 7 ⇥ M ⇥ PhD ⇥ 70 ⇥ Other ⇥ 71 ⇥ 81 ⇥ Assoc
16 ⇥ 7 ⇥ M ⇥ PhD ⇥ 70 ⇥ Other ⇥ 71 ⇥ 82 ⇥ Assoc
```

Data Frame

case	id	gender	deg	yrdeg	field	startyr	y
1	1	F	Other	92	Other	95	9
2	2	M	Other	91	Other	94	9
3	2	M	Other	91	Other	94	9
4	4	M	PhD	96	Other	95	9
5	6	M	PhD	66	Other	91	9
6	6	M	PhD	66	Other	91	9
7	6	M	PhD	66	Other	91	9
8	6	M	PhD	66	Other	91	9
9	6	M	PhD	66	Other	91	9
10	7	M	PhD	70	Other	71	7
11	7	M	PhD	70	Other	71	7
12	7	M	PhD	70	Other	71	7
13	7	M	PhD	70	Other	71	7
14	7	M	PhD	70	Other	71	8
15	7	M	PhD	70	Other	71	8
16	7	M	PhD	70	Other	71	8

Reading in data from the web

This online option is very convenient but;

- Make sure you are signed into UW's wifi system before trying to access the web
- Make a local copy if you anticipate loading the data through drop-down menus multiple times – this is quicker and more reliable
- Make a local copy if you have to cut off rows about 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 retricted to the drop-down version?

Reading in data from the web

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 ouput 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 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

Failing to supply arguments without defaults gives an error message – and no output.

Commonly-used arguments in commonly-used arguments 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=T)

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

… 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 disuss 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 the message (at best!) Either way, check the code does what you intended, before going further.

Operating on data: subsets again

In the previous session (mammals example) we saw how to subset 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 (somehow), 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

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`.

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

On 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.

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

3. Plotting functions and formulas

Ken Rice
Tim Thornton

University of Washington

Seattle, July 2013

In this session

R is known for having good graphics – good for data exploration and summary, as well as illustrating analyses.

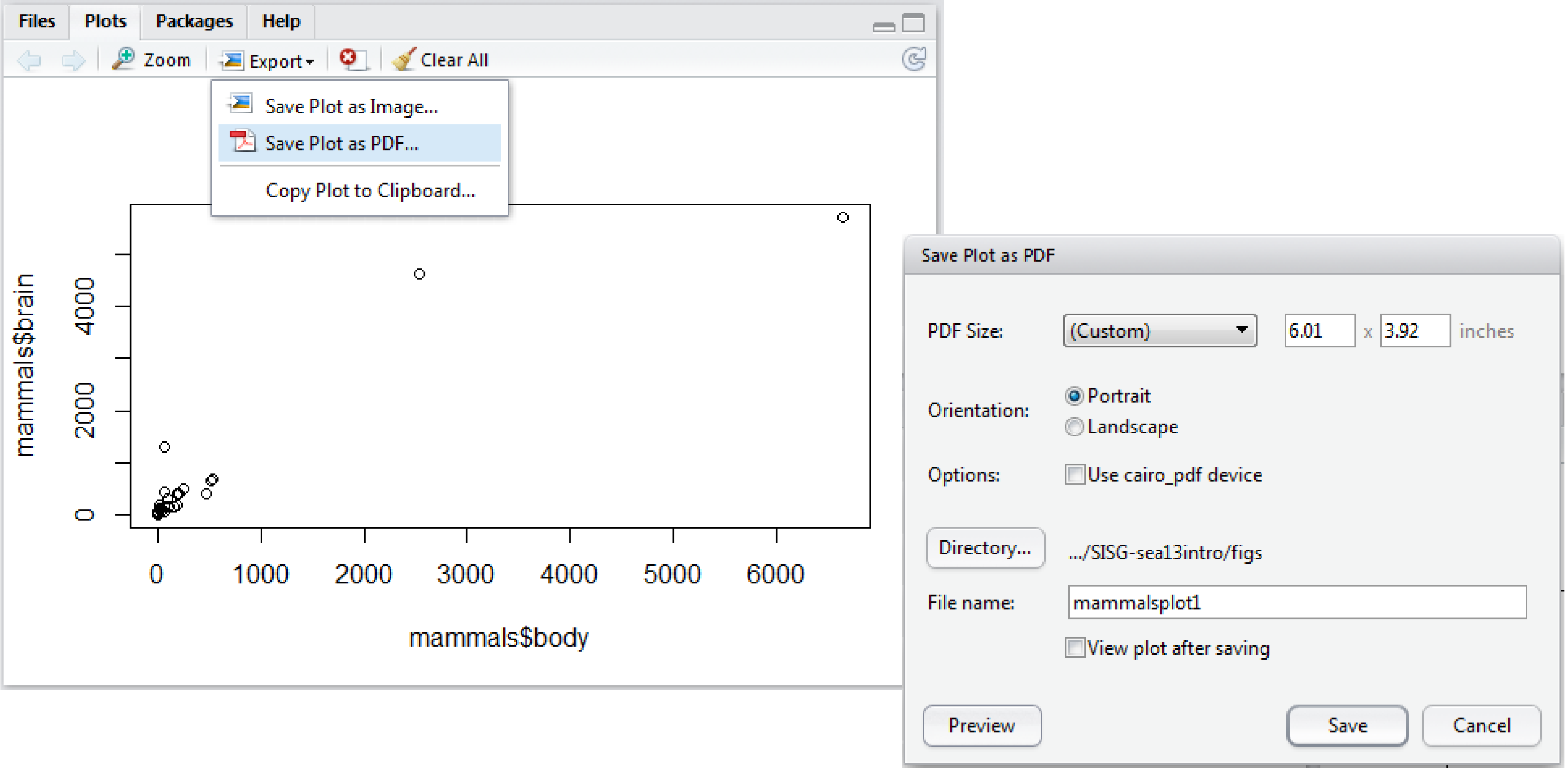
- Some *generic* plotting commands
- Making graphics files
- Fine-tuning your plots (and why not to do too much of this)
- The formula syntax

NB more graphics commands will follow, in the next session.

Making a scatterplot with `plot()`

A first example, using the `mammals` dataset – and its output in the Plot window; (The preview button is recommended)

```
plot(x=mammals$body, y=mammals$brain)
```



PDF Size: (Custom) 6.01 x 3.92 inches

Orientation: ● Portrait ○ Landscape

Options: □ Use cairo_pdf device

Directory... .../SISG-sea13intro/figs

File name: mammalsplot1

□ View plot after saving

Making a scatterplot with `plot()`

Some other options for exporting;

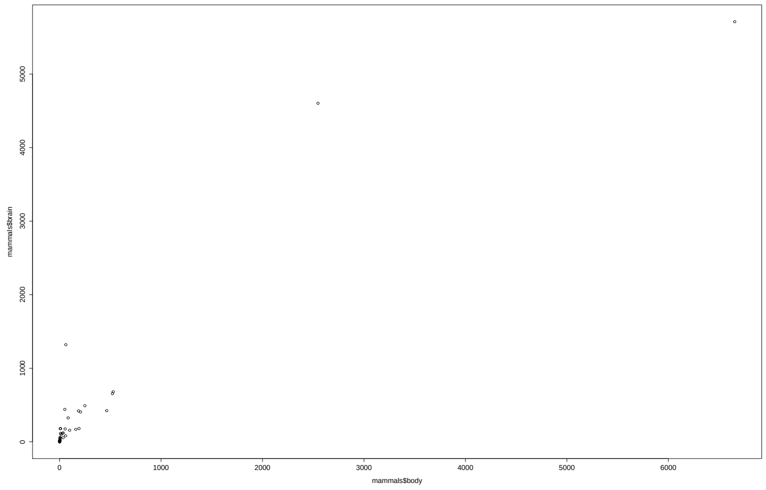
- Copy directly to clipboard as a bitmap or editable (Windows) metafile - then paste into e.g. your Powerpoint slides
- With ‘Save Plot as Image’, PNG is a (good) bitmap format, suitable for line art, i.e. graphs. JPEG is good for photos, not so good for graphs
- For PNG/JPEG, previews disappear if they get too large!
- Many of the options (TIFF, EPS) are seldom used, today
- Handy hint; if re-sizing runs into trouble, enter `dev.off()` and just start over

Making a scatterplot with `plot()`

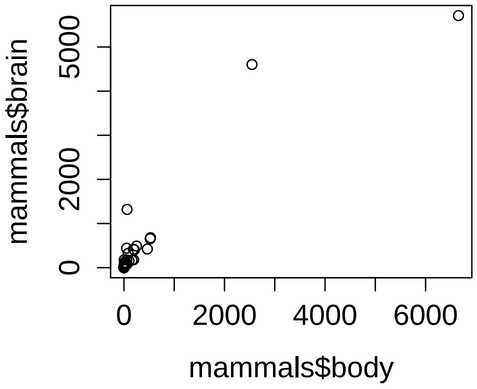
A golden rule for exporting; make the file the size it will be in the final document – because R is good at choosing font sizes

The same plot (aspect ratio 6:4), displayed as 4 × 2.67 inches;

Saved at 24 × 16 inches



Saved at 4 × 2.67 inches



- Not the same plot ‘blown up’ – note e.g. axes labels
- R likes white space – good in documents, less good in slides

Making a scatterplot with `plot()`

Better axes, better axis labels and a title would make the scatterplot better. But on looking up `?plot`...

For simple scatter plots, `plot.default` *will be used. However, there are* `plot` *methods for many R objects, including* `function`*s,* `data.frame`*s,* `density` *objects, etc. Use* `methods(plot)` *and the documentation for these.*

`plot()` is a *generic* function – it does different things given different input. For this job, the details we need are in `?plot.default`...

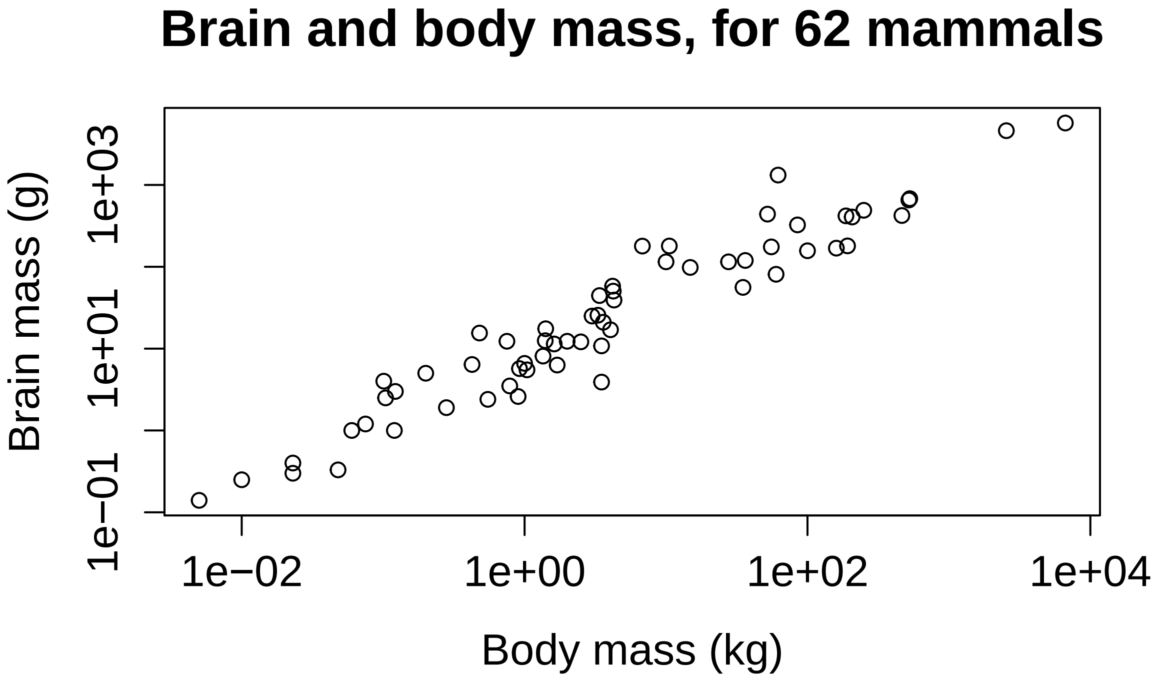
```
plot(x, y = NULL, type = "p",  xlim = NULL, ylim = NULL,
     log = "", main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
     ann = par("ann"), axes = TRUE, frame.plot = axes,
     panel.first = NULL, panel.last = NULL, asp = NA, ...)
```

Making a scatterplot with `plot()`

After checking the help page to see what these mean, we use;

- `xlab`, `ylab` for the axis labels
- `main` for the main title
- `log` to log the axes – `log="xy"`, to log them both

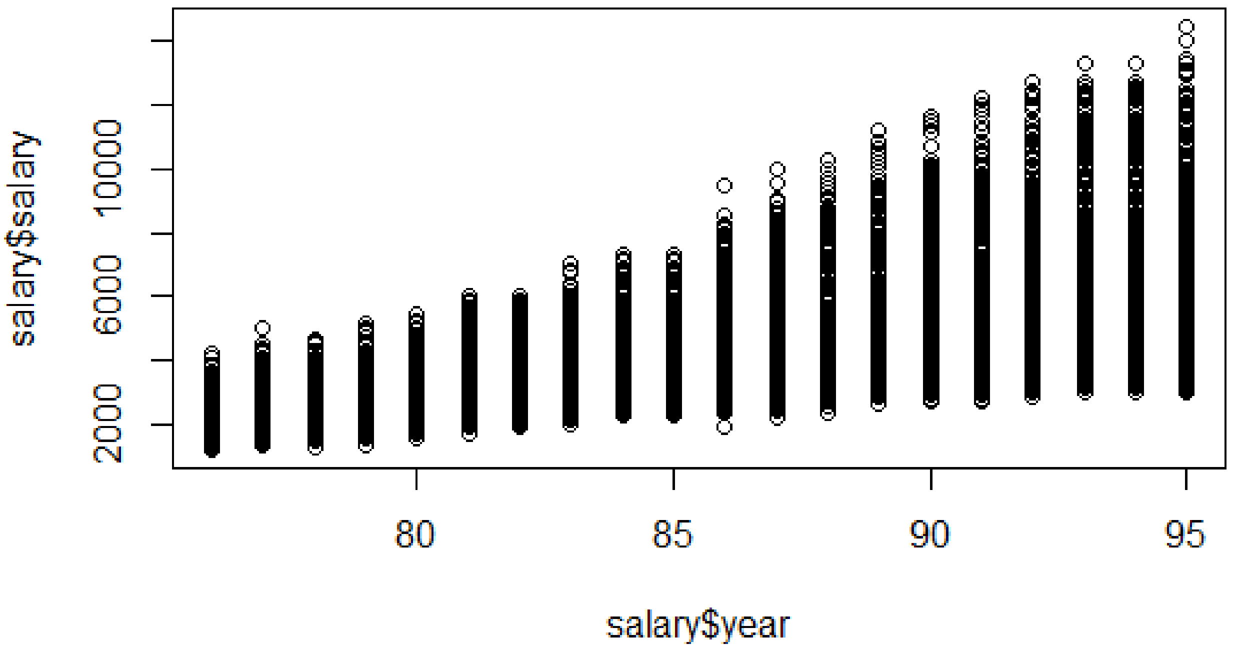
```
plot(x=mammals$body, y=mammals$brain, xlab="Body mass (kg)",
     ylab="Brain mass (g)", main="Brain and body mass, for 62 mammals",
     log="xy")
```



Other plots made with `plot()`

As the help file suggest, `plot()` gives different output for different types of input. First, another scatterplot;

```
plot(x=salary$year, y=salary$salary)
```

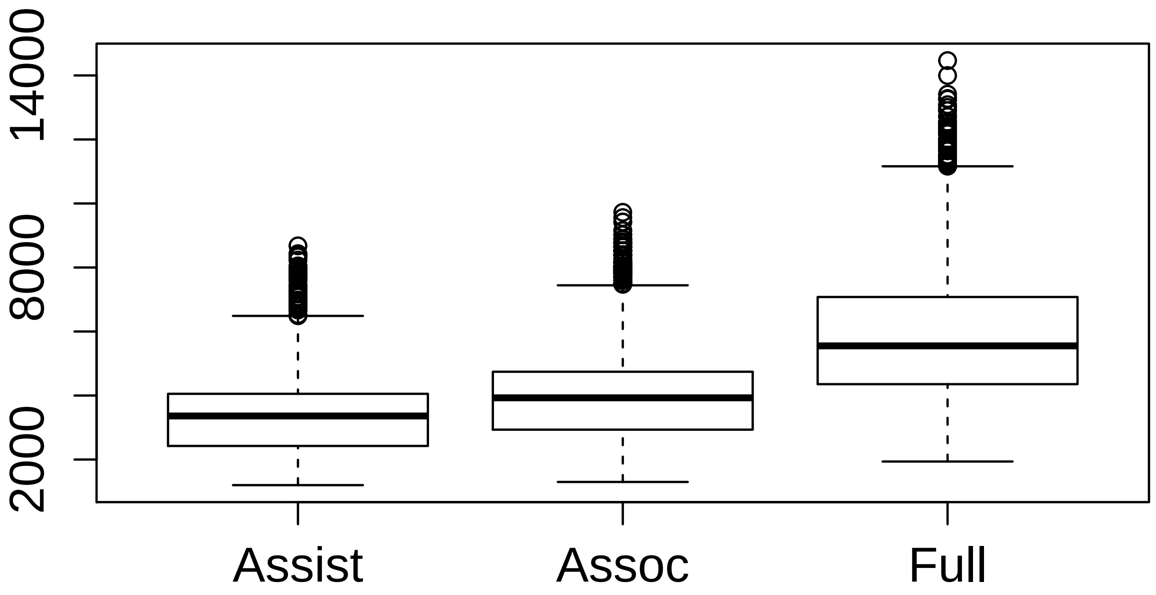


Handy hint: use PNG for graphs of large datasets like this.

Other plots made with `plot()`

Plotting one numeric variable against a factor;

```
plot(x=salary$rank, y=salary$salary)
```

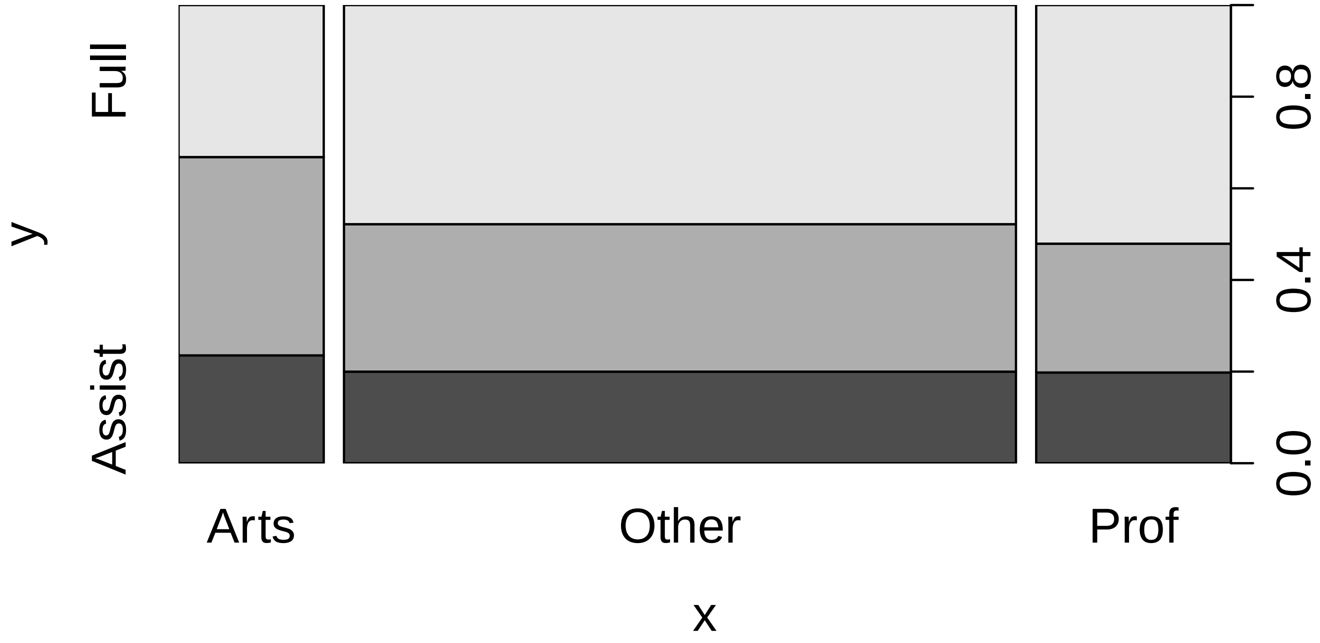


There is also a `boxplot()` function.

Other plots made with `plot()`

Plotting one factor variable against another;

```
plot(x=salary$field, y=salary$rank)
```

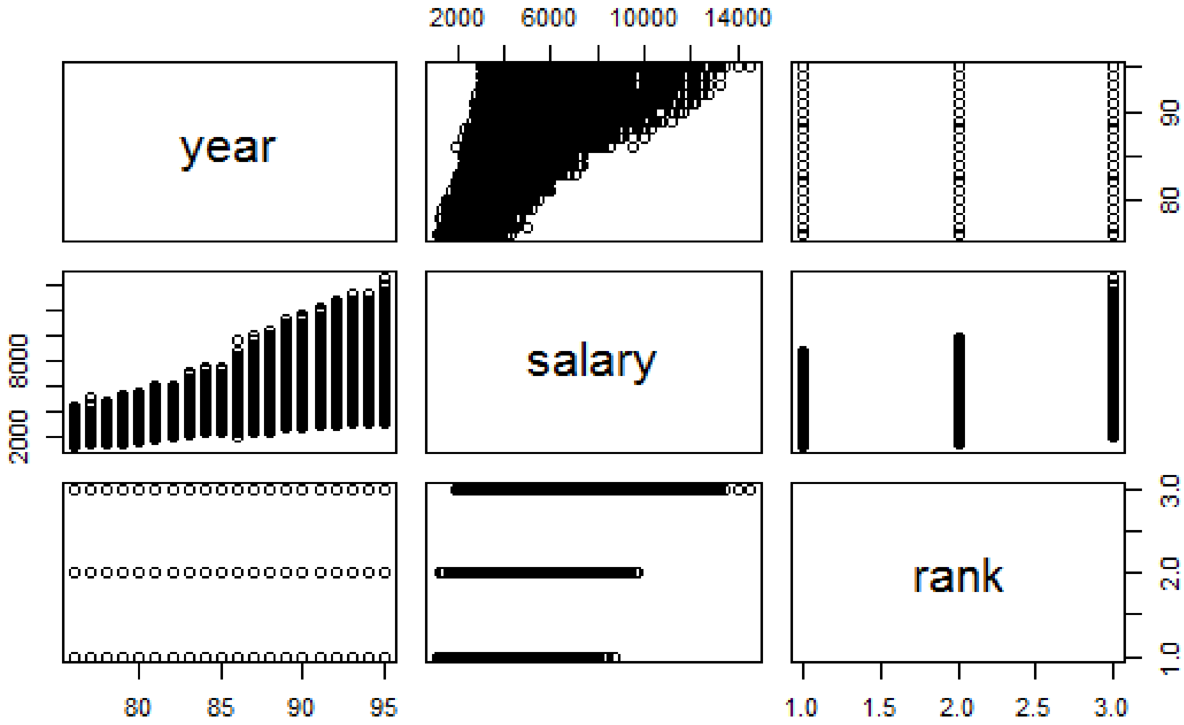


This is a *stacked barplot* – see also the `barplot()` function

Other plots made with `plot()`

Plotting an entire data frame (not too many columns)

```
smallsalary <- salary[,c("year","salary","rank")]
plot(smallsalary)
```

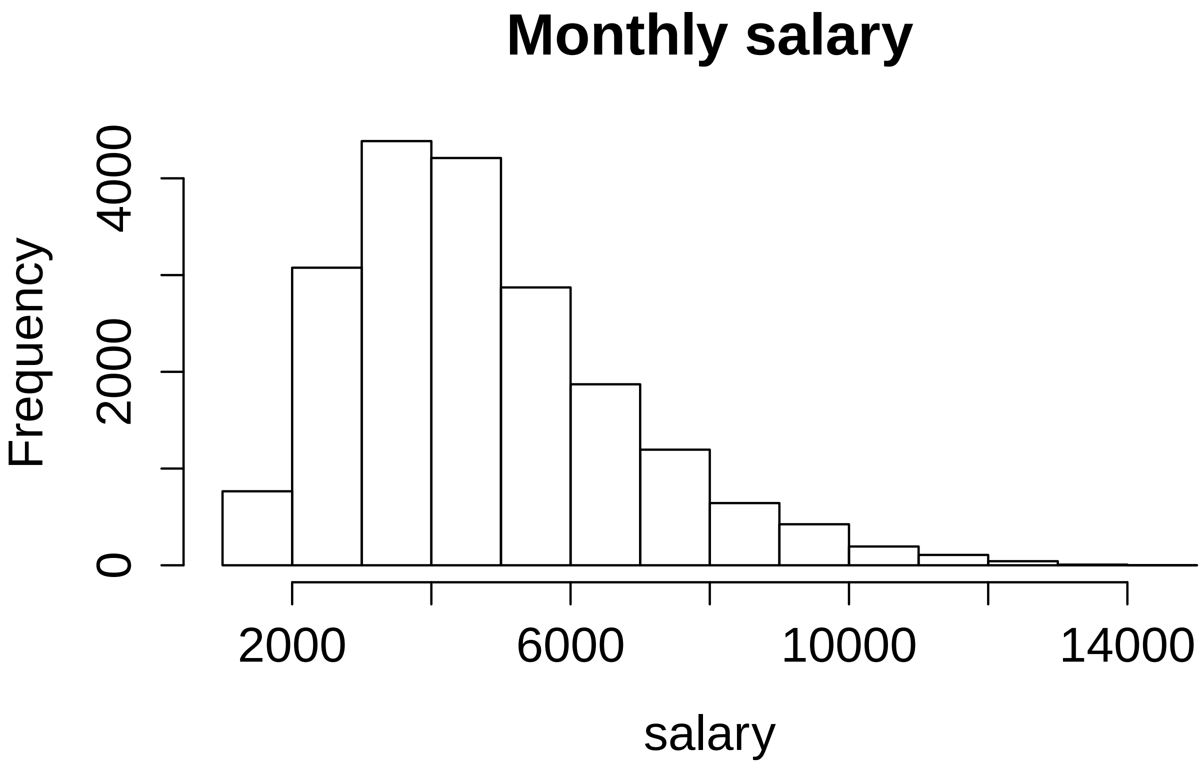


Not so clever! But quick, & okay if all numeric – see also `pairs()`.
NB Plotting functions for large datasets are in later sessions.

Other graphics commands

For histograms, use `hist()`;

```
hist(salary$salary, main="Monthly salary", xlab="salary")
```

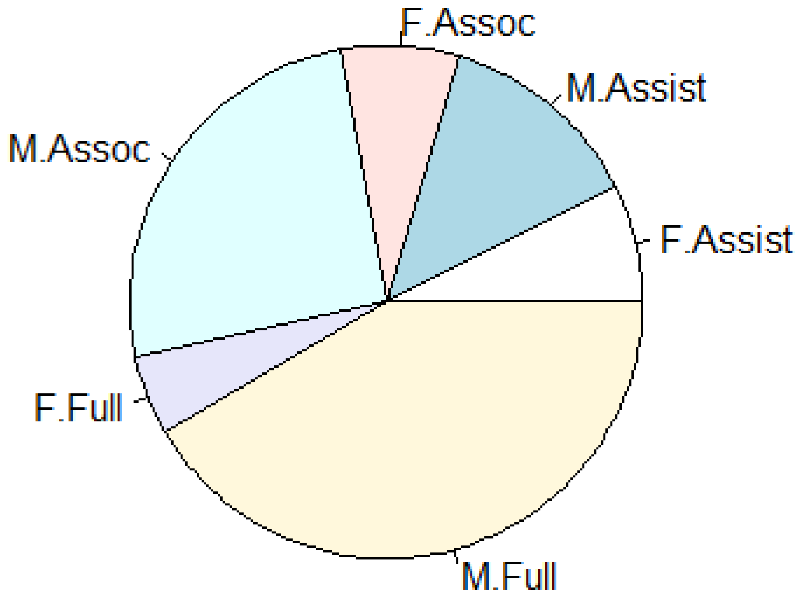


For more control, set argument `breaks` to *either* a number, or a vector of the breakpoints.

Other graphics commands

Please tell no-one I told you this one;

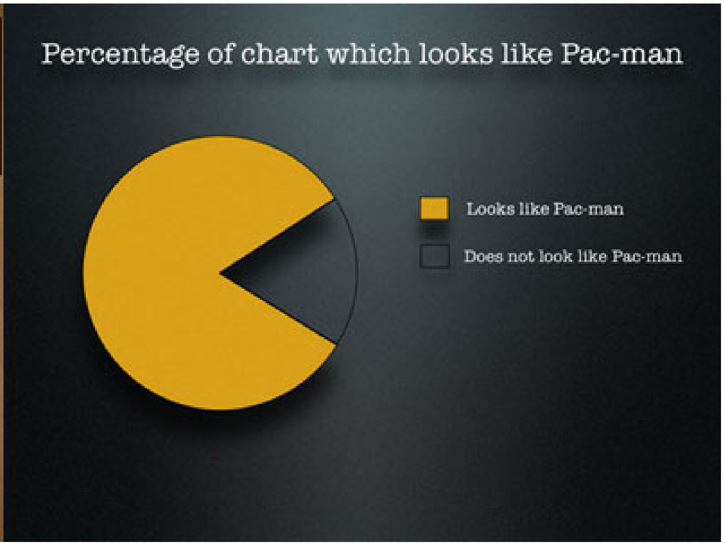
```
> table( interaction(salary$gender, salary$rank))
F.Assist M.Assist  F.Assoc  M.Assoc   F.Full   M.Full
    1460     2588     1465     5064     1001     8210
> pie(table( interaction(salary$gender, salary$rank) ) )
```

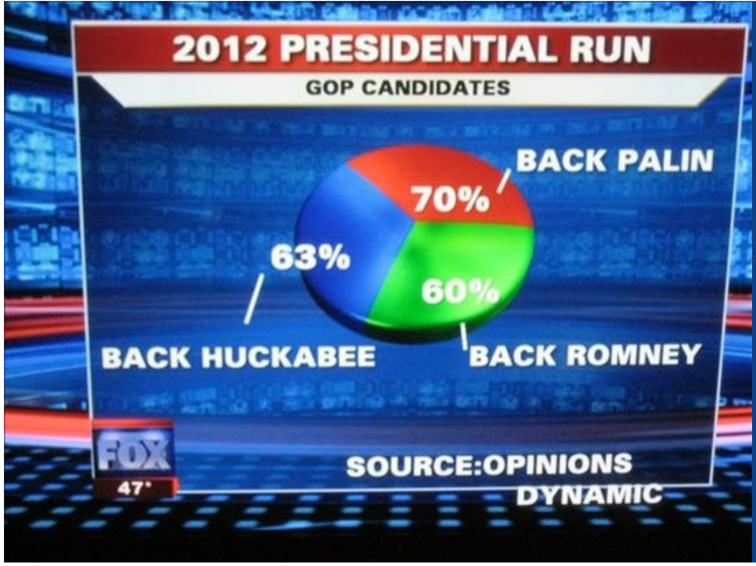


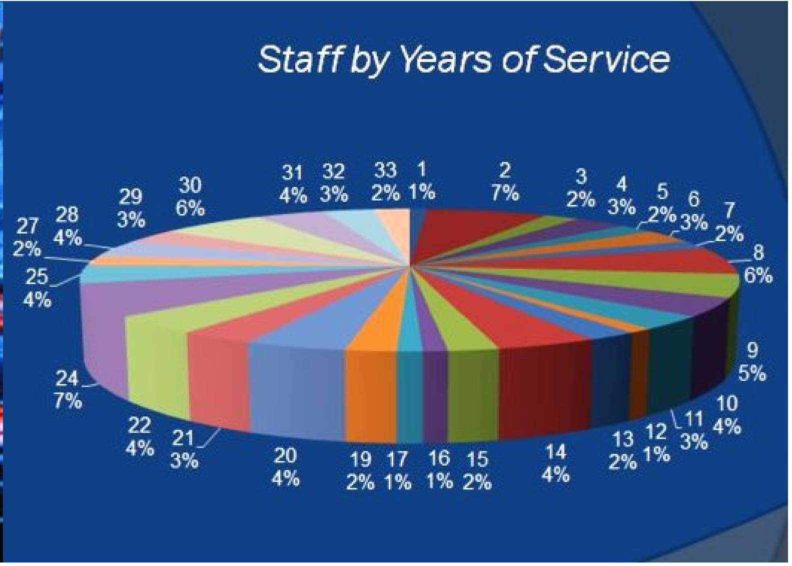
Why do statisticians hate pie charts with such passion?

Other graphics commands

... they really do!



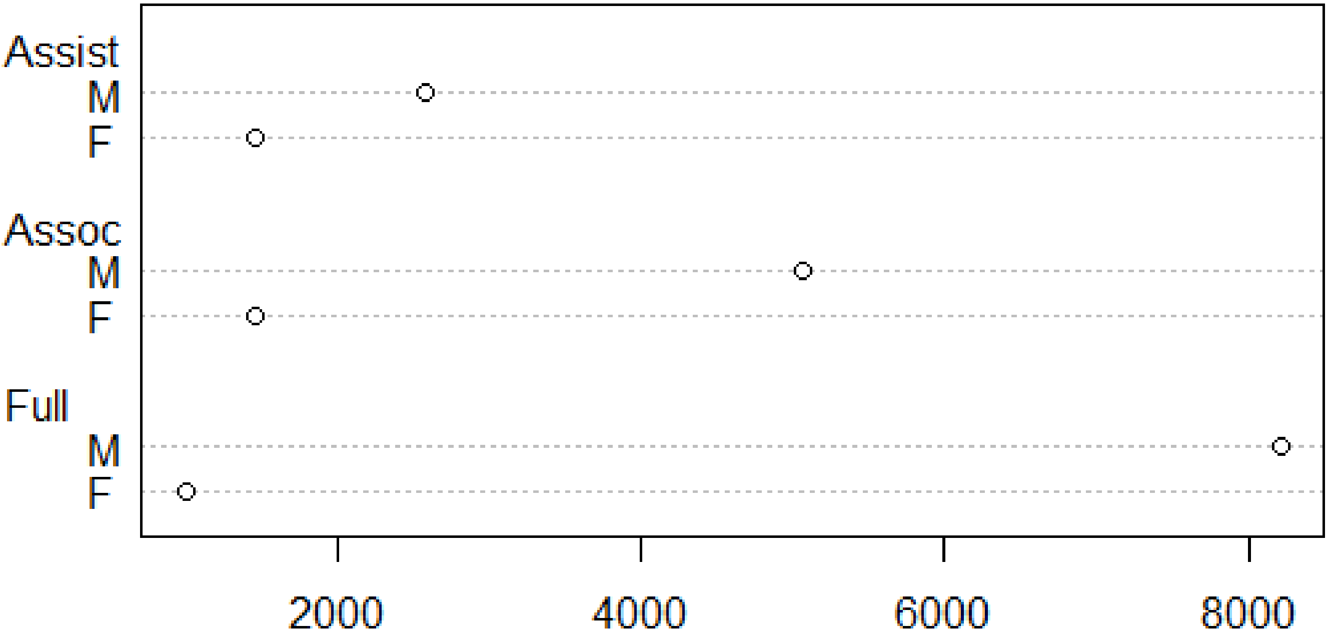




Other graphics commands

Because pie charts are a terrible way to present data. Dotcharts are *much* better – also easy to code;

```
dotchart(table( salary$gender, salary$rank ) )
```

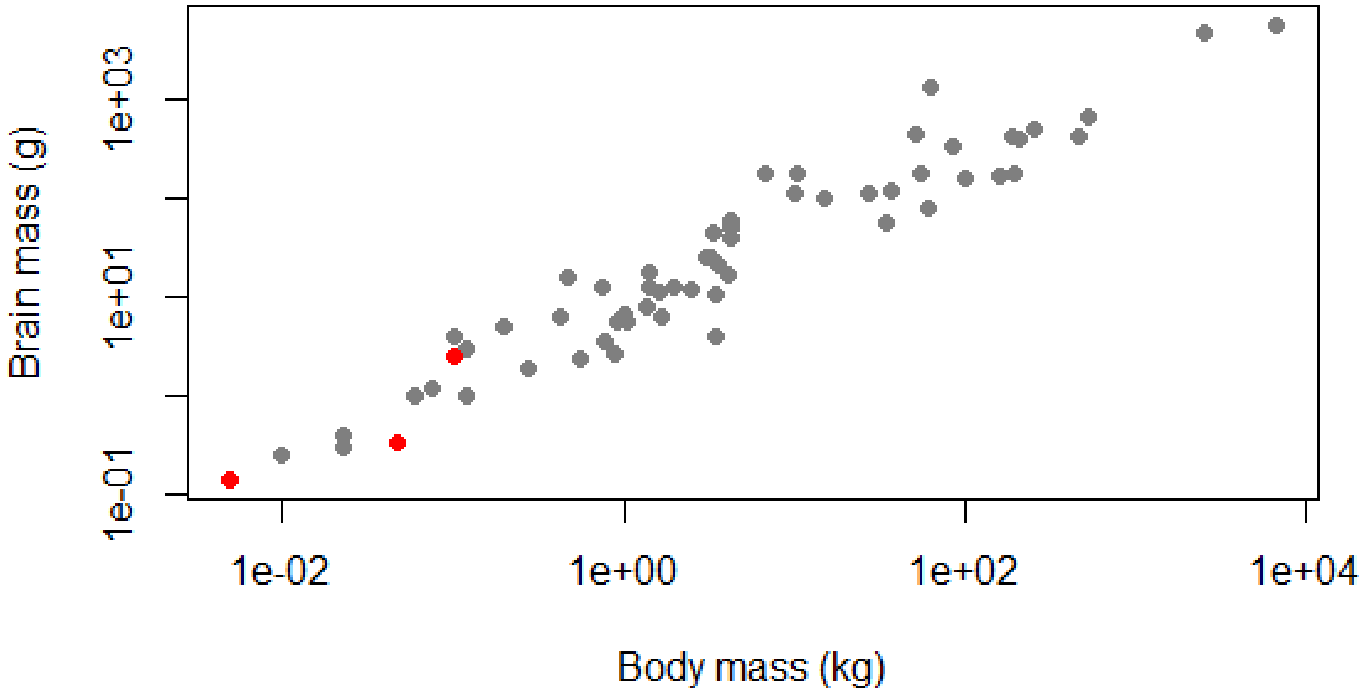


See also `stripchart()`; with multiple symbols per line, these are a good alternative to boxplots, for small samples.

Changing plotting symbols

Suppose you want to highlight certain points on a scatterplot; other options to the `plot()` command change point style & color;

```
> grep("shrew", row.names(mammals)) # or just look in Data viewer
[1] 14 55 61
> is.shrew <- 1:62 %in% c(14,55,61) # 3 TRUEs and 59 FALSEs
> plot(x=mammals$body, y=mammals$brain, xlab="Body mass (kg)",
+       ylab="Brain mass (g)",log="xy",
+       col=ifelse(is.shrew, "red", "gray50"), pch=19)
```



Changing plotting symbols

We used `col=ifelse(is.shrew, "red", "gray50")` – a vector of 3 `red`s and 59 `gray50`s.

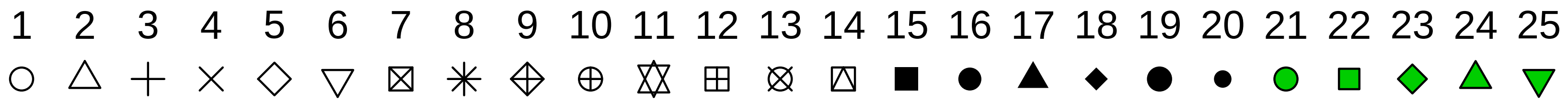
- If we supply fewer colors than datapoints, what we supplied is recycled
- You could probably guess `"red"`,`"green"`,`"purple"` etc, but not `"gray50"`. To find out the names of the (many) available R colors, use the `colors()` command – no arguments needed
- Can also specify colors by numbers; 1=black, 2=red, 3=green up to 8, then it repeats
- Or consult this online chart – or many others like it
- Can also supply colors by hexadecimal coding; #RRGGBB for red/green/blue – with #RRGGBBTT for transparency

NB legends will follow, in the next session.

Changing plotting symbols

We also used `pch=19` – to obtain the same non-default plotting symbol, a filled circle.

The full range;



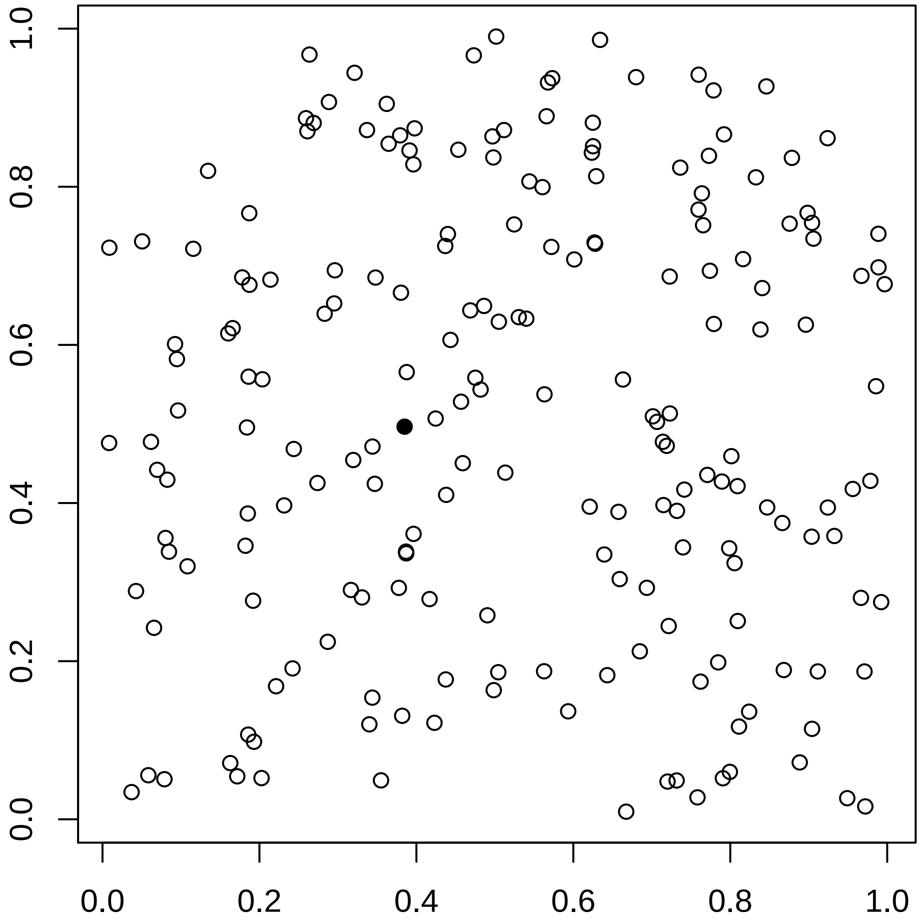
- Set the fill color for `21:25` with the `bg` argument
- The open circle (`pch=1`) is the default – because it makes it easiest to see points that nearly overlap. Change it only if you have a good reason to
- Filled symbols `15:20` work well with transparent colors, e.g. `col="#FF000033"` for translucent pink

For different size symbols, there is a `cex` option; `cex=1` is standard size, `cex=1.5` is 50% bigger, etc.

But beware! These options should be used *sparingly*...

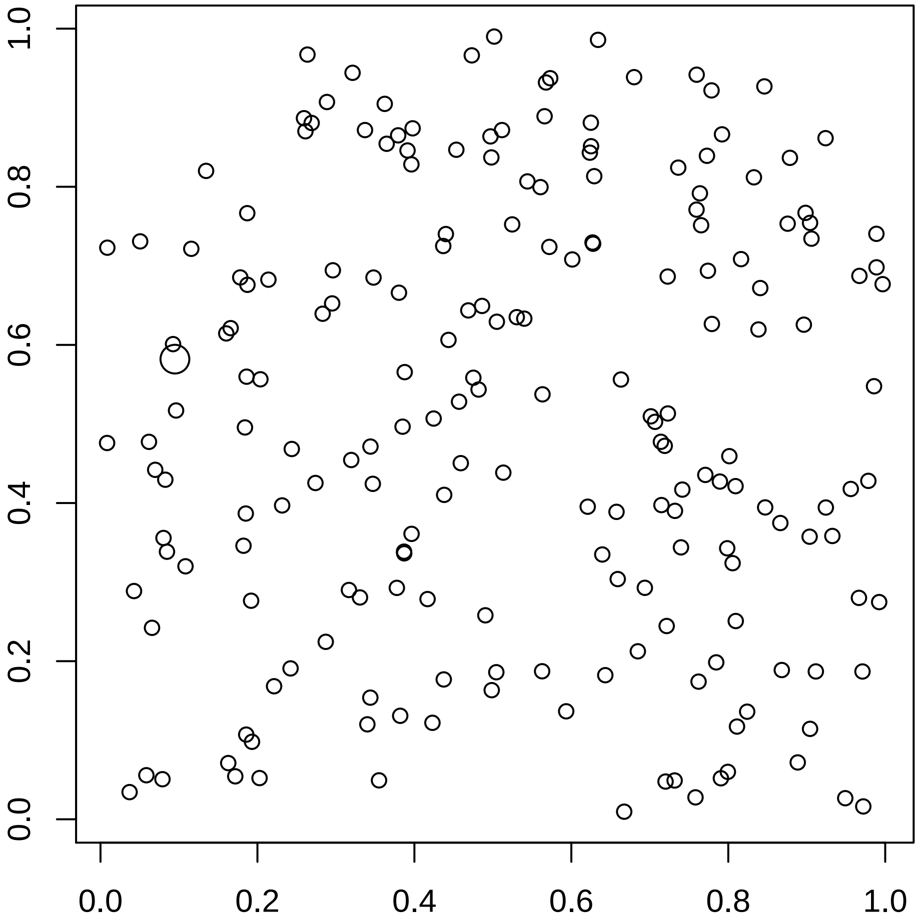
Changing plotting symbols

One of these points is not like the others...



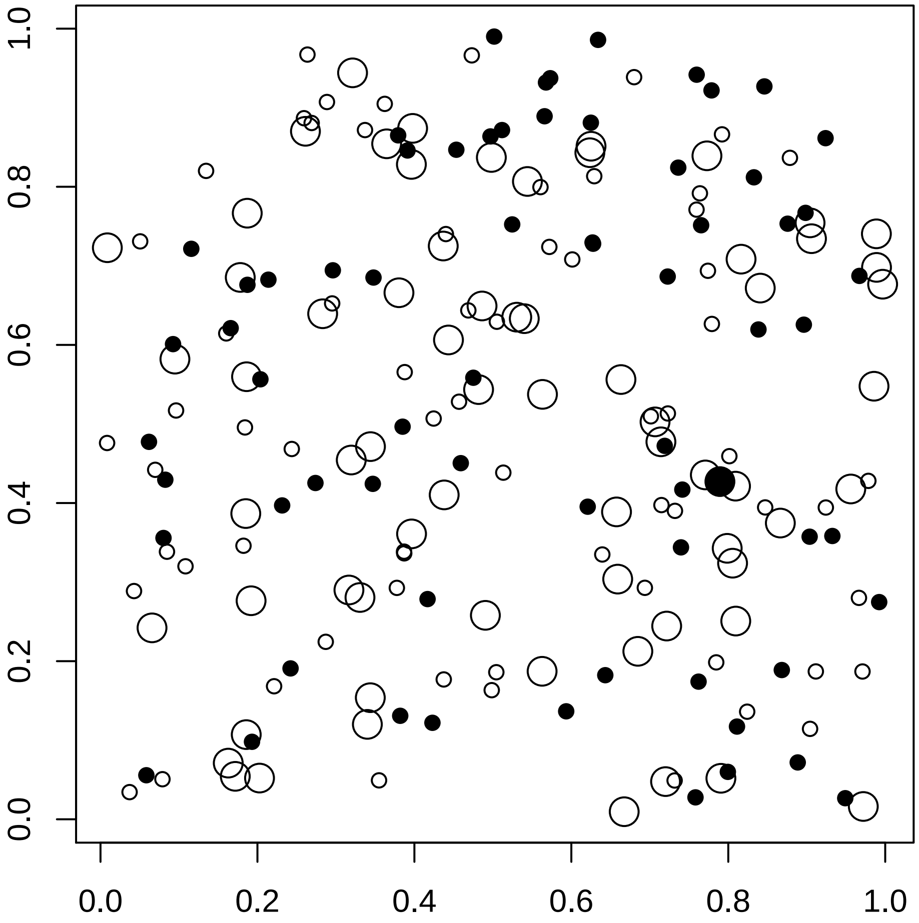
Changing plotting symbols

One of these points is not like the others...



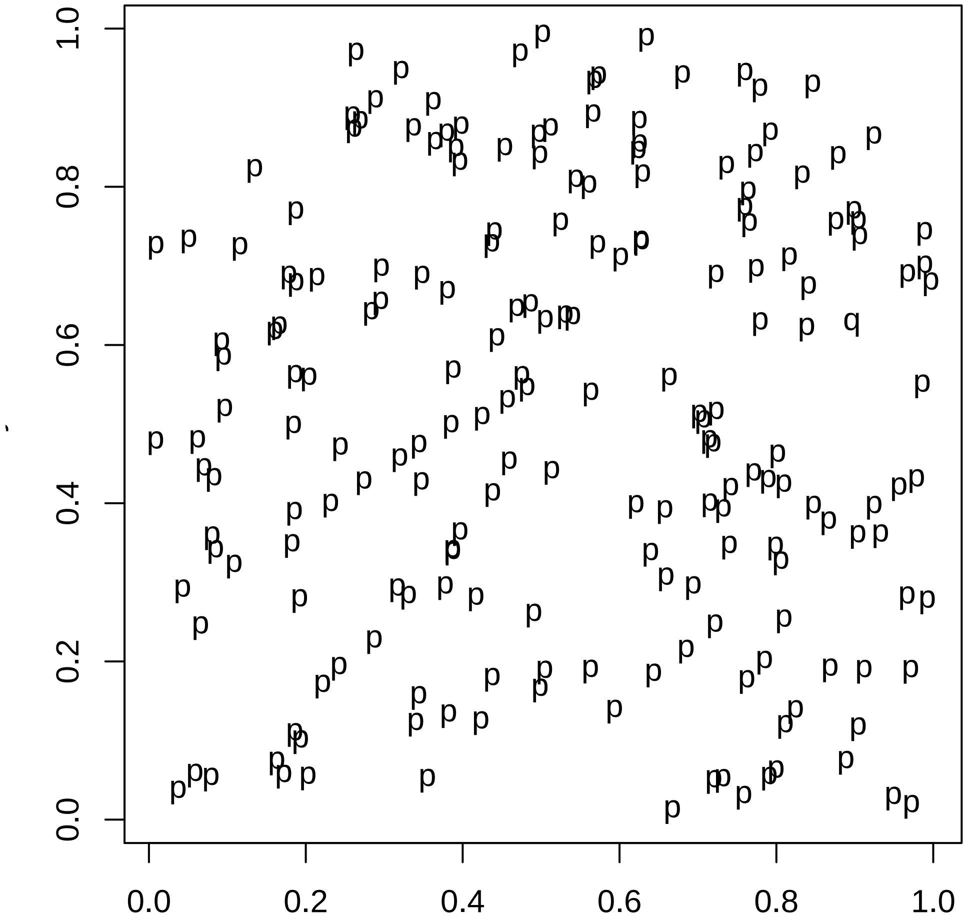
Changing plotting symbols

One of these points is not like the others...



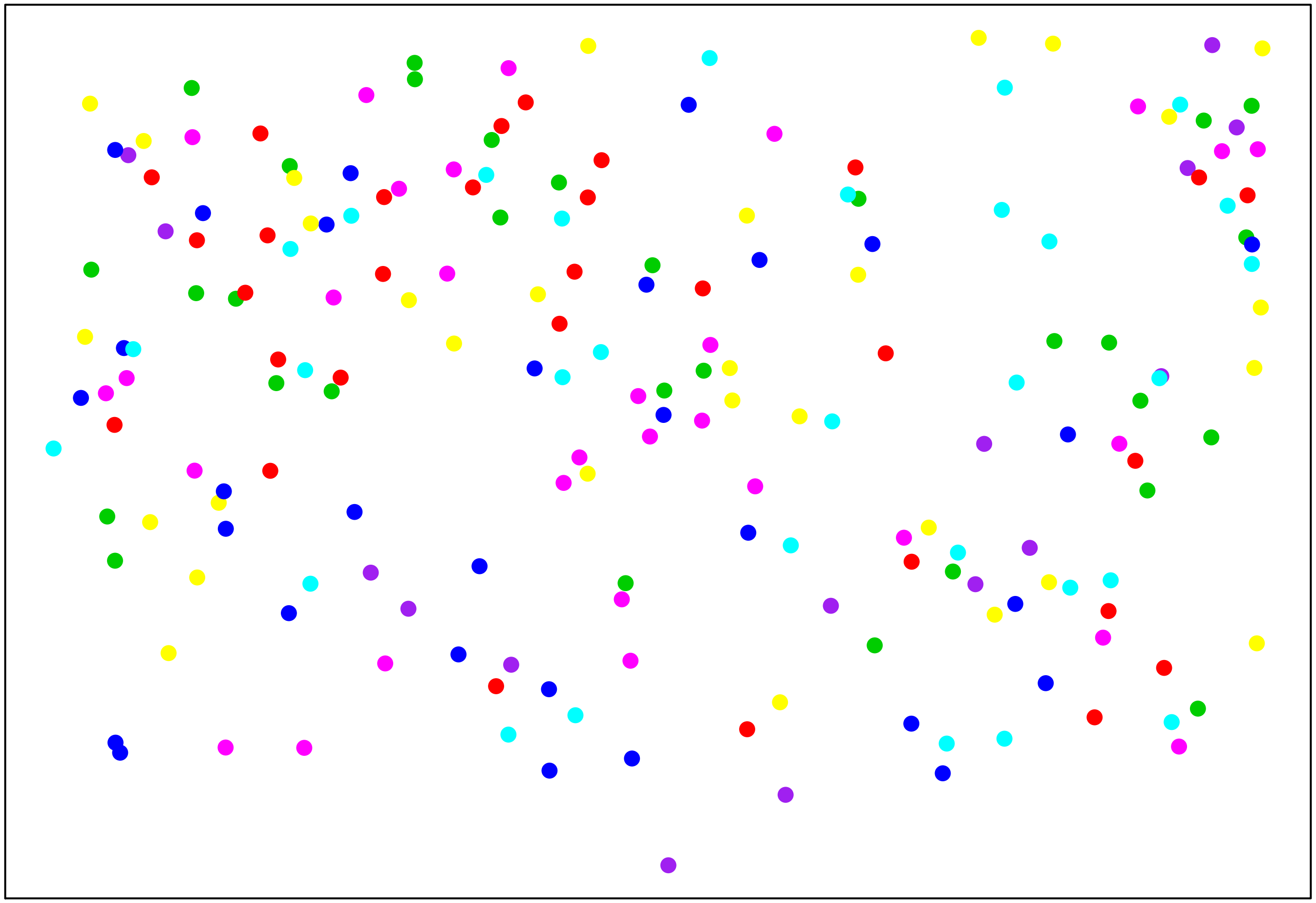
Changing plotting symbols

One of these points is not like the others... (`pch="p"`)



Changing plotting symbols

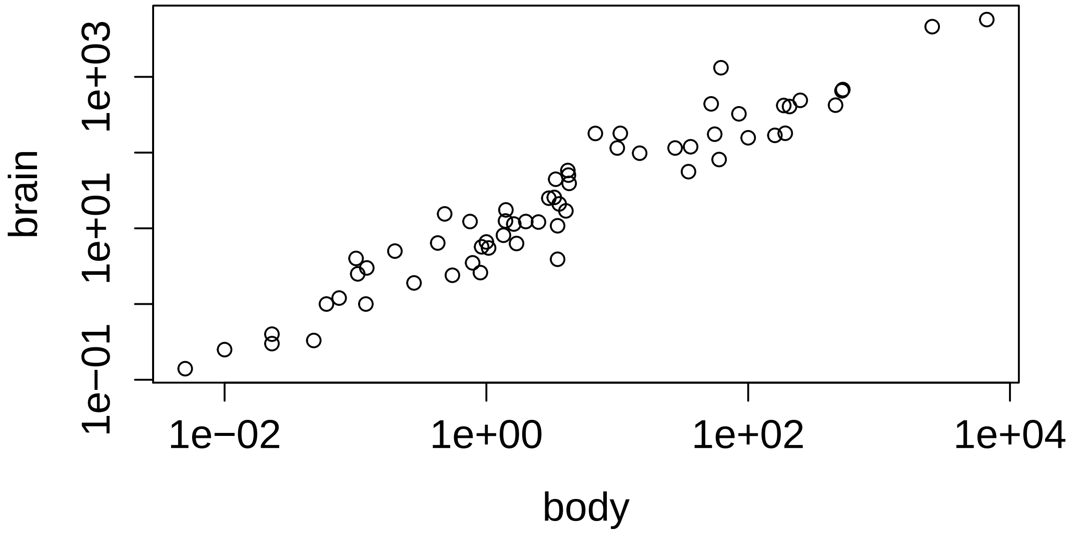
Too many colors (>4, say) requires too much attention; what pattern is illustrated here?



Plots via the formula syntax

To make plots, we've used arguments `x` (on the X-axis) and `y` (on the Y-axis). But another method makes a stronger connection to *why* we're making the plot;

```
plot(brain~body, data=mammals, log="xy")
```



"Plot how `brain` depends on `body`, using the `mammals` dataset"

Plots via the formula syntax

A few more examples, using the salary dataset;

```
plot(salary~year, data=mammals) # scatterplot
plot(salary~rank, data=mammals) # boxplot
plot(rank~field,  data=mammals) # stacked barplot
```

In all of these, $Y \sim X$ can be interpreted as Y *depends on* X – the ‘tilde’ symbol is R’s shorthand for ‘depends on’.

Statisticians (and scientists) like to think this way;

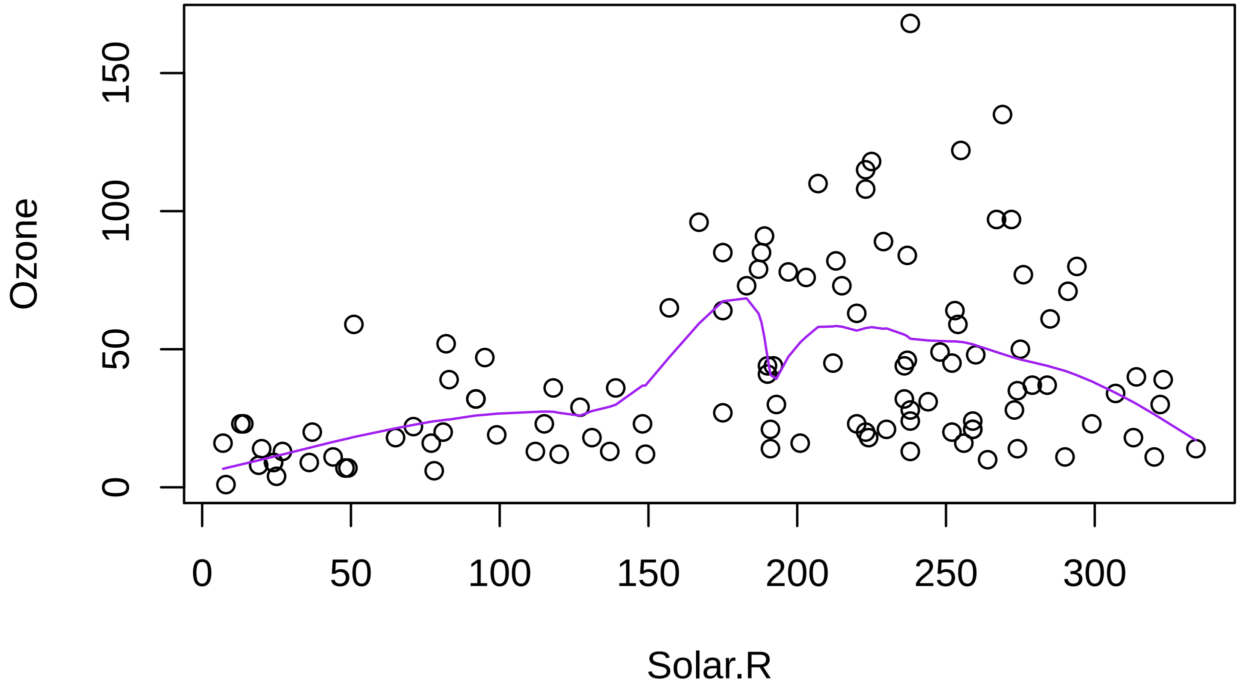
- How does some outcome (Y) depend on a covariate (X)? (a.k.a. a predictor)
- How does a dependent variable (Y) depend on an independent variable (X)?

And how does Y depend on X in observations with the same Z?

Plots via the formula syntax

To help us illustrate how scientists think, a bit of science;

Ozone is a *secondary pollutant*, produced from organic compounds and atmostpheric oxygen, in reactions catalyzed by nitrogen oxides and powered by sunlight. But for ozone concentrations in NY in summer (Y) we see a non-monotone relationship with sunlight (X) ...



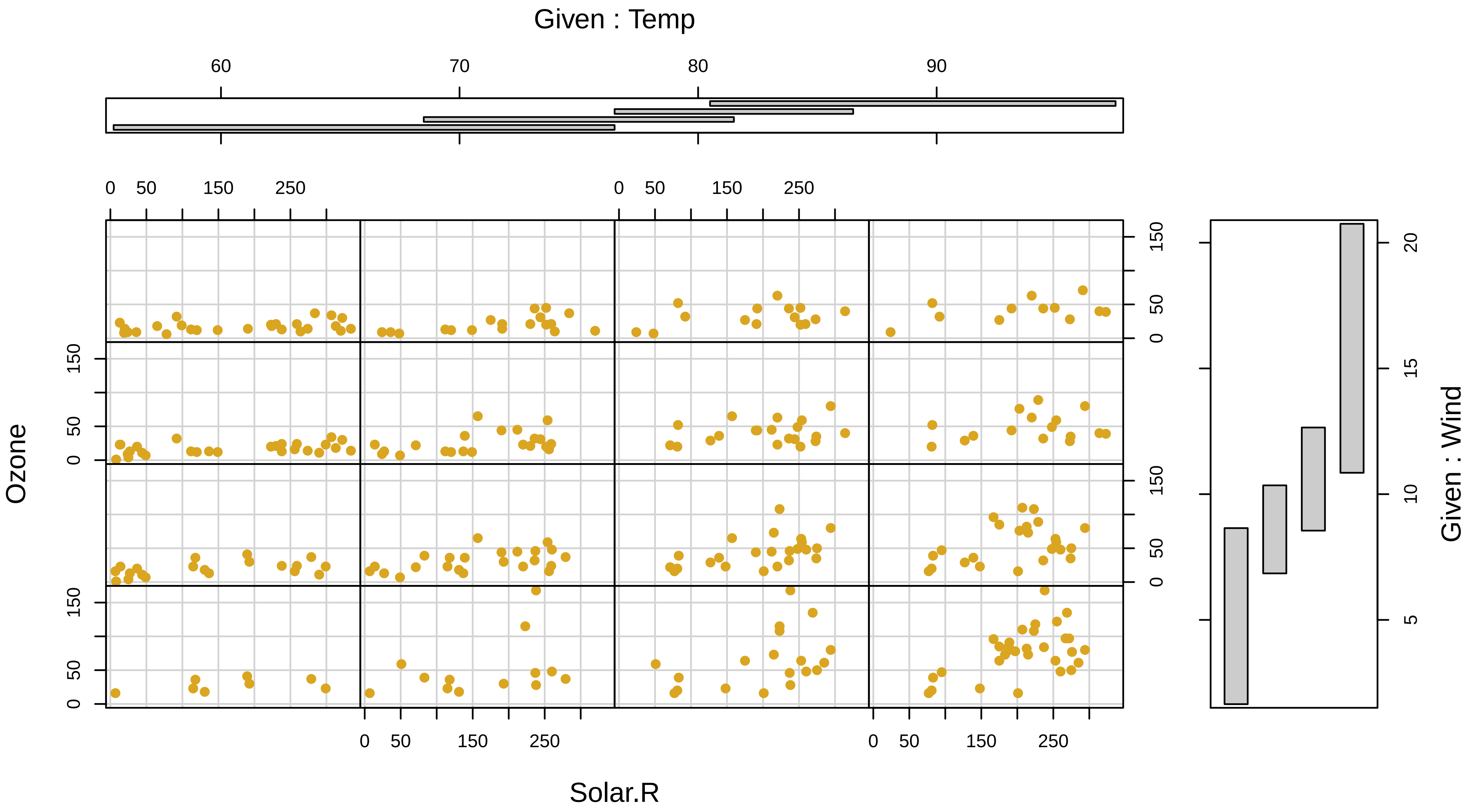
Plots via the formula syntax

Now draw a scatterplot of Ozone vs Solar.R for various subranges of Temp and Wind.

```
data("airquality") # using a dataset supplied with R
coplot(Ozone ~ Solar.R | Temp + Wind, number = c(4, 4),
       data = airquality,
       pch = 21, col = "goldenrod", bg = "goldenrod")
```

- The vertical dash ("|") means 'given particular values of', i.e. 'conditional on'
- Here, "+" means 'and', not 'plus' – see ?formula, and later sessions
- How does Ozone depend on Solar Radiation, on days with (roughly) the same Temperature *and* Wind Speed?
- ...using the airquality data, with a 4 × 4 layout, with solid dark yellow circular symbols

Plots via the formula syntax



Plots via the formula syntax

What does this show?

- A 4-D relationship is illustrated; the Ozone/sunlight relationship changes in strength depending on both the Temperature and Wind
- The horizontal/vertical ‘shingles’ tell you which data appear in which plot. The overlap can be set to zero, if preferred
- `coplot()`’s default layout is a bit odd; try setting `rows`, `columns` to different values
- Almost any form of plot can be ‘conditioned’ in this way – but the commands are in the non-default `lattice` package

NB it is possible to produce ‘fake 3D’ plots in R – but (on 2D paper) conditioning plots work better!

Summary

- R makes publication-quality graphics, as well as graphics for data exploration and summary

- `plot()` is generic, and adapts to what you give it. There are (necessarily) lots of arguments to consider; colors, plotting symbols, labels, etc

- `hist()`, `boxplot()`, `dotplot()` and `coplot()` offer more functionality

- Graphics via the formula syntax make some scientific sense when the plot has a statistical intention

- Much more to come! In the next section we'll build up more complex plots

4. Adding Features to Plots

Ken Rice
Timothy Thornotn

University of Washington

Seattle, July 2013

In this session

R has very flexible built-in graphing capabilities to add a wide-range of features to a plot.

- Plotting options
- Adding points, lines, and segments to existing plots
- Creating a legend for a plot

Scatterplot Options

The command `plot(x,y)` will create a scatterplot when x and y are numeric. The default setting will plot points but one can graph lines or both (or neither):

- `plot(x,y,type="p")` is the default option that plots points
- `plot(x,y,type="l")` connects points by lines but does not plot point symbols
- `plot(x,y,type="b")` plots point symbols connected by lines
- `plot(x,y,type="o")` plots point symbols connected by lines, points on top of lines
- `plot(x,y,type="h"` will plot histogram like (or high-density) vertical lines
- `plot(x,y,type="n")` plots axes only, no symbols

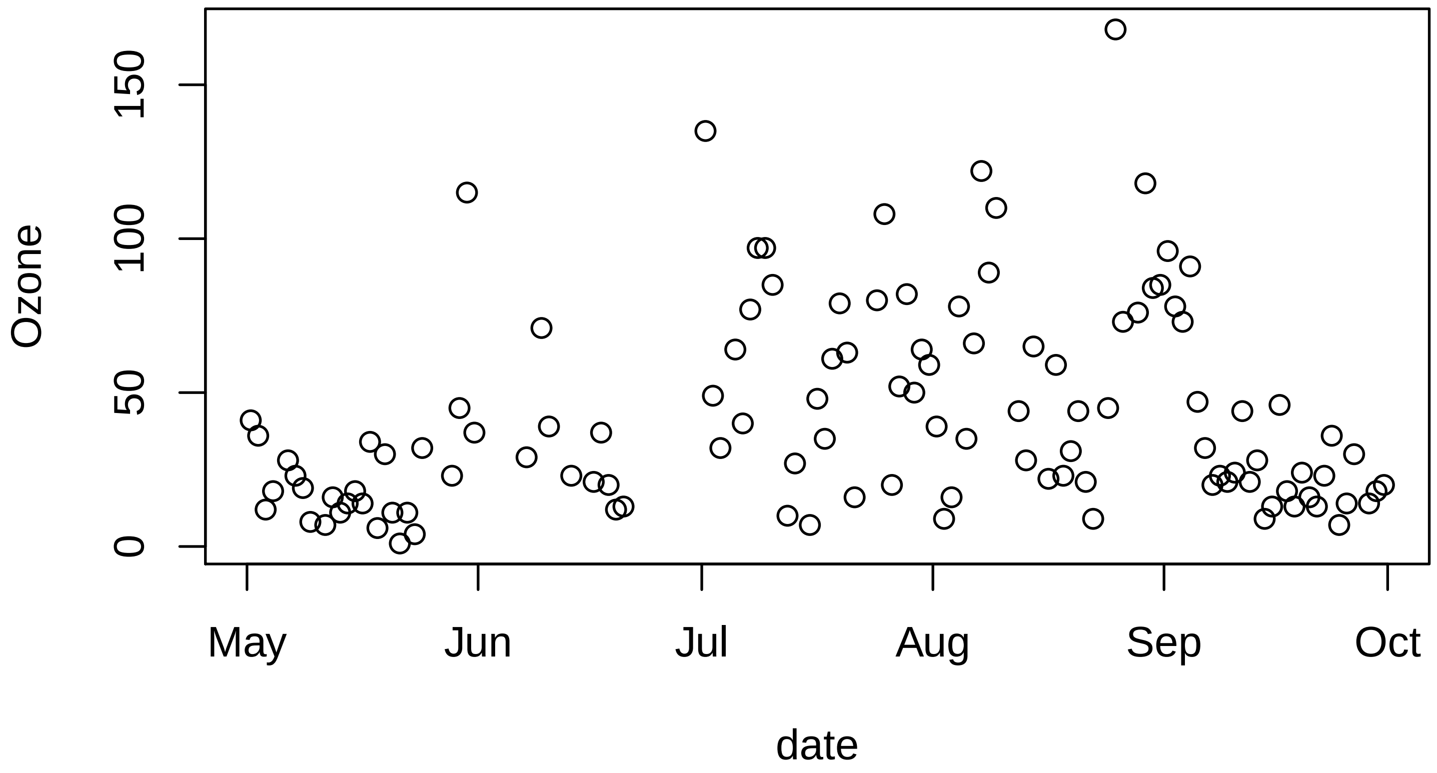
Examples: Plotting two variables

Let's consider the *airquality* dataset.

```
data(airquality)
names(airquality)
airquality$date<-with(airquality, ISOdate(1973,Month,Day))
```

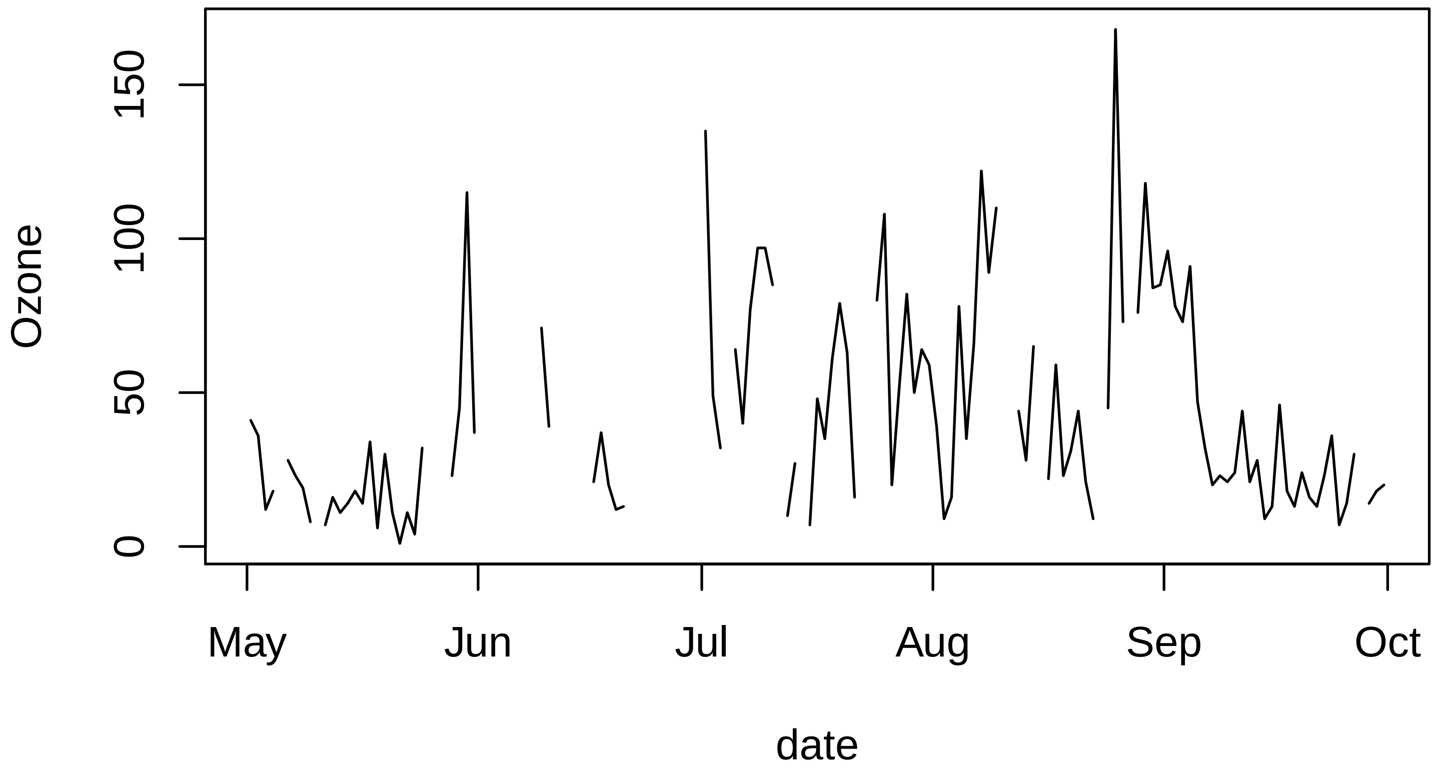
Examples: Plotting two variables

```
plot(Ozone~date, data=airquality)
```



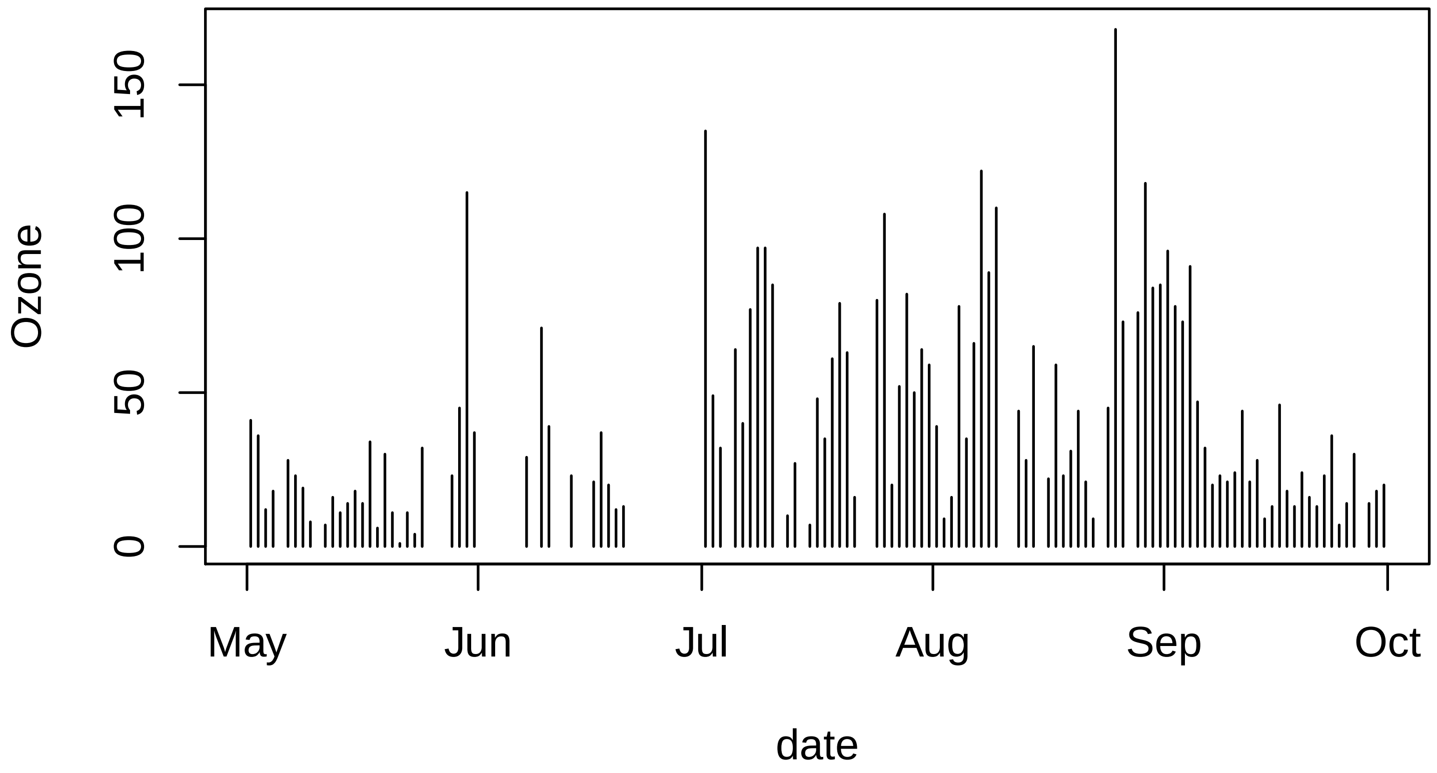
Examples: Plotting two variables

```
plot(Ozone~date, data=airquality,type="l")
```



Examples: Plotting two variables

```
plot(Ozone~date, data=airquality,type="h")
```



Adding points to a graph

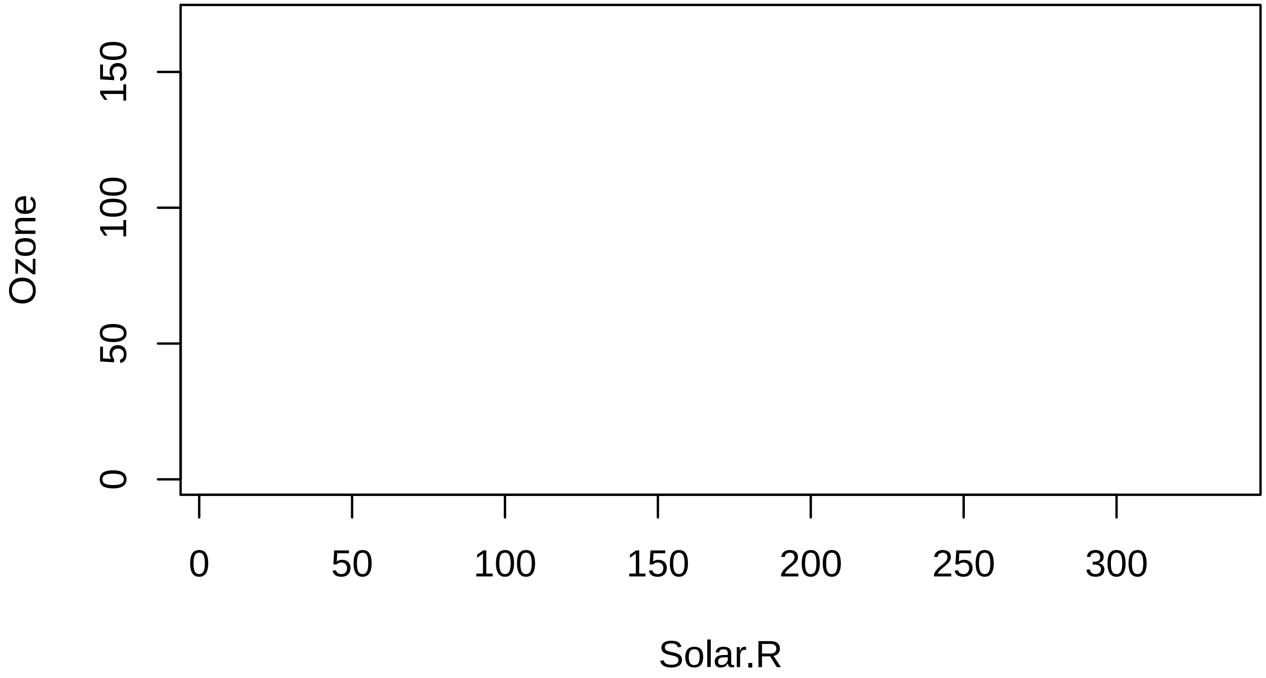
We can add points to an existing plot with the command `points(x,y)`

The `lines(x,y)` command can be used to add connected points by lines to an existing plot without symbols

Adding points to a graph

For example, create a graph that contains axes only.

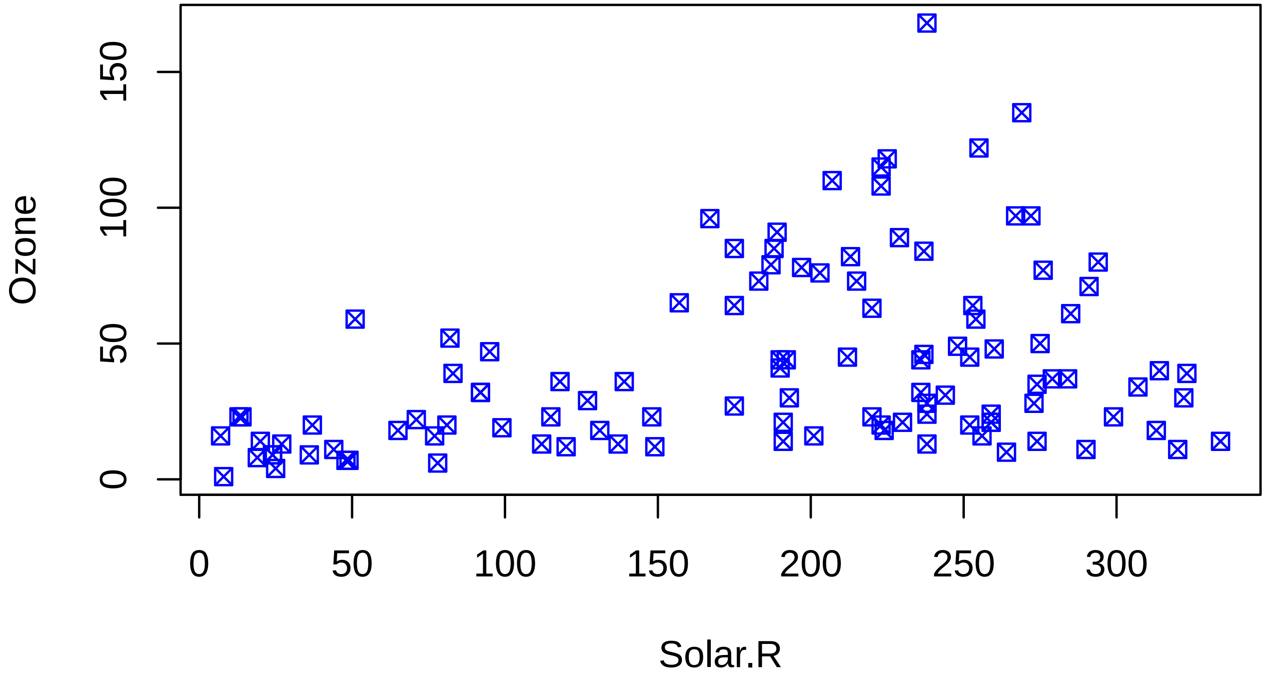
```
plot(Ozone~Solar.R, data=airquality,type="n")
```



Adding points to a graph

Now add the points to the graph:

```
points(airquality$Solar.R,airquality$Ozone,col="blue",pch=7)
```



Adding lines to plots

Horizontal, vertical, and sloped lines can be added to an existing plot with `abline()`:

- `abline(h=ycoordinate)` adds a horizontal line at the specified y-coordinate
- `abline(v=xcoordinate)` adds a vertical line at the specified x-coordinate
- `abline(intercept,slope)` adds a line with the specified intercept and slope

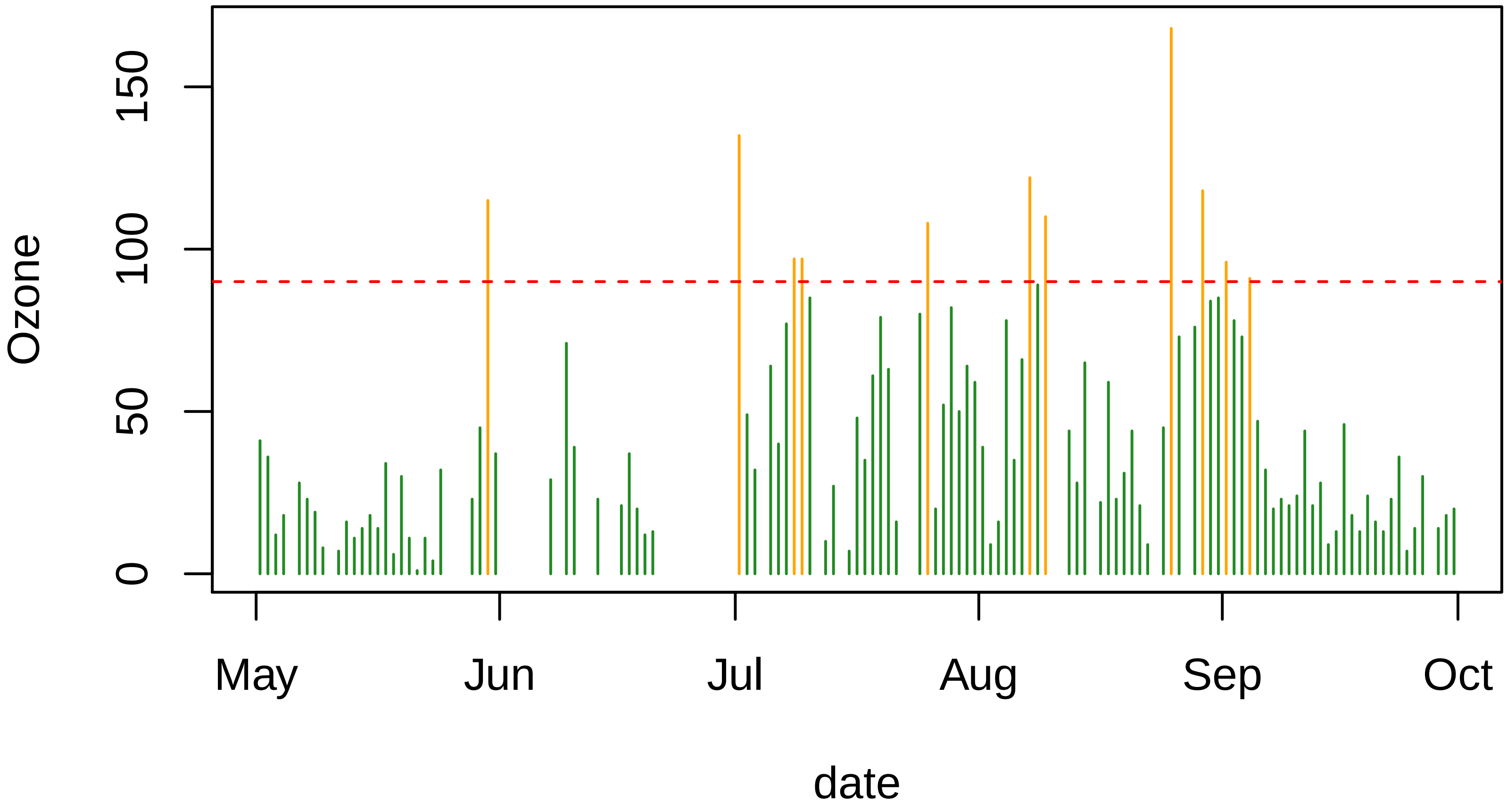
A line segment can be added to an existing plot with `segments()`:

- `segments(x0,y0,x1,y1)` adds a line segment from (x0,y0) to (x1,y1)

Adding lines to plots

```
bad <- ifelse(airquality$Ozone>=90, "orange","forestgreen")
plot(Ozone~date,data=airquality,type="h",col=bad)
abline(h=90,lty=2,col="red")
```

Adding lines to plots



Adding text to plots

Text labels can be added to a plot with the `text()` command:

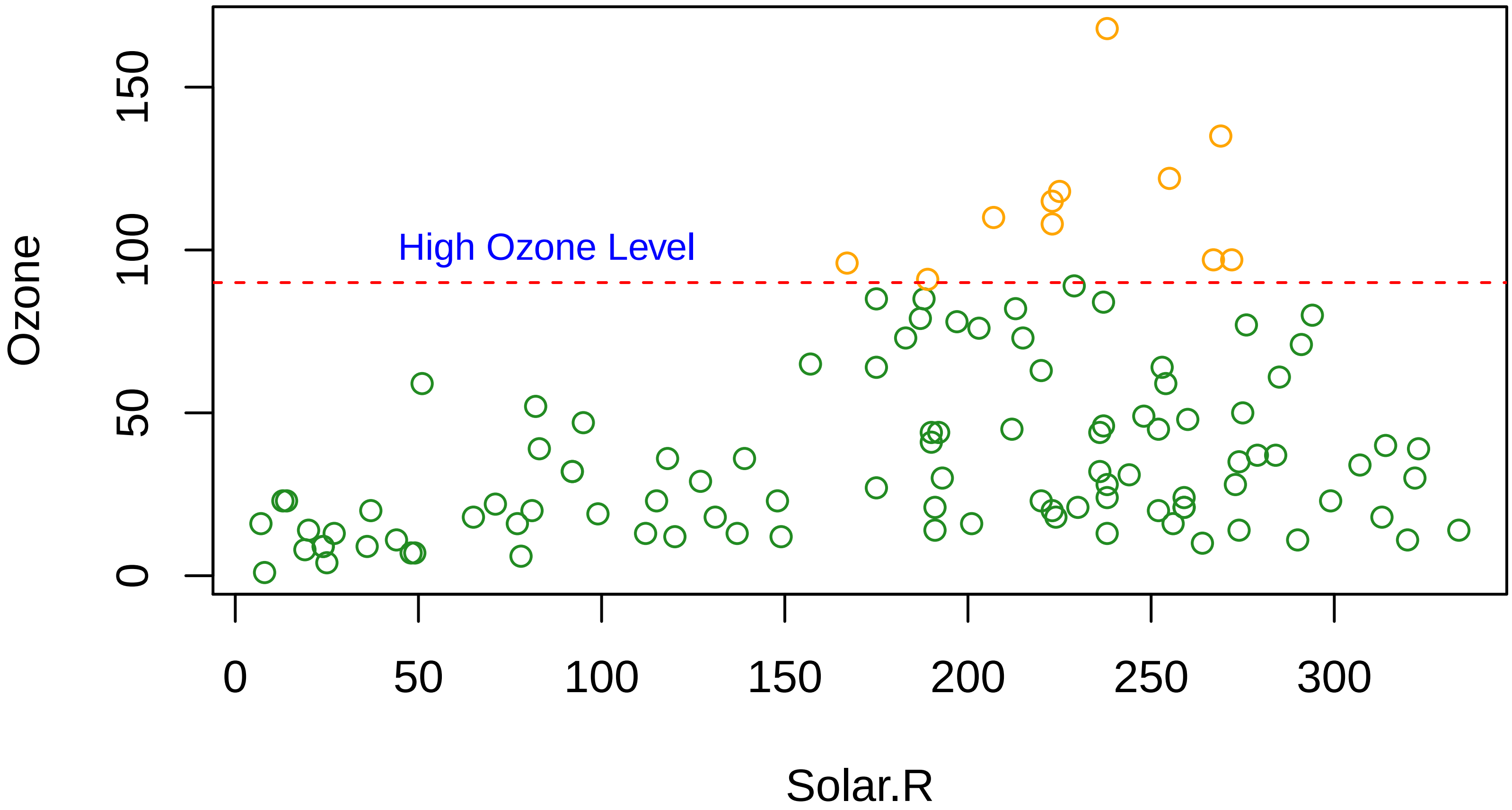
- `text(x,y,"Here is my text")` adds text centered at the specified (x,y) coordinates

Text colors and size can be specified with the options *col* and *cex*, respectively.

Adding text to plots

```
bad <- ifelse(airquality$Ozone>=90, "orange", "forestgreen")
plot(Ozone~Solar.R, data=airquality, col=bad)
abline(h=90, lty=2, col="red")
text(85,100,"High Ozone Level",cex=.8,col="blue")
```

Adding text to plots



Adding a legend to a plot

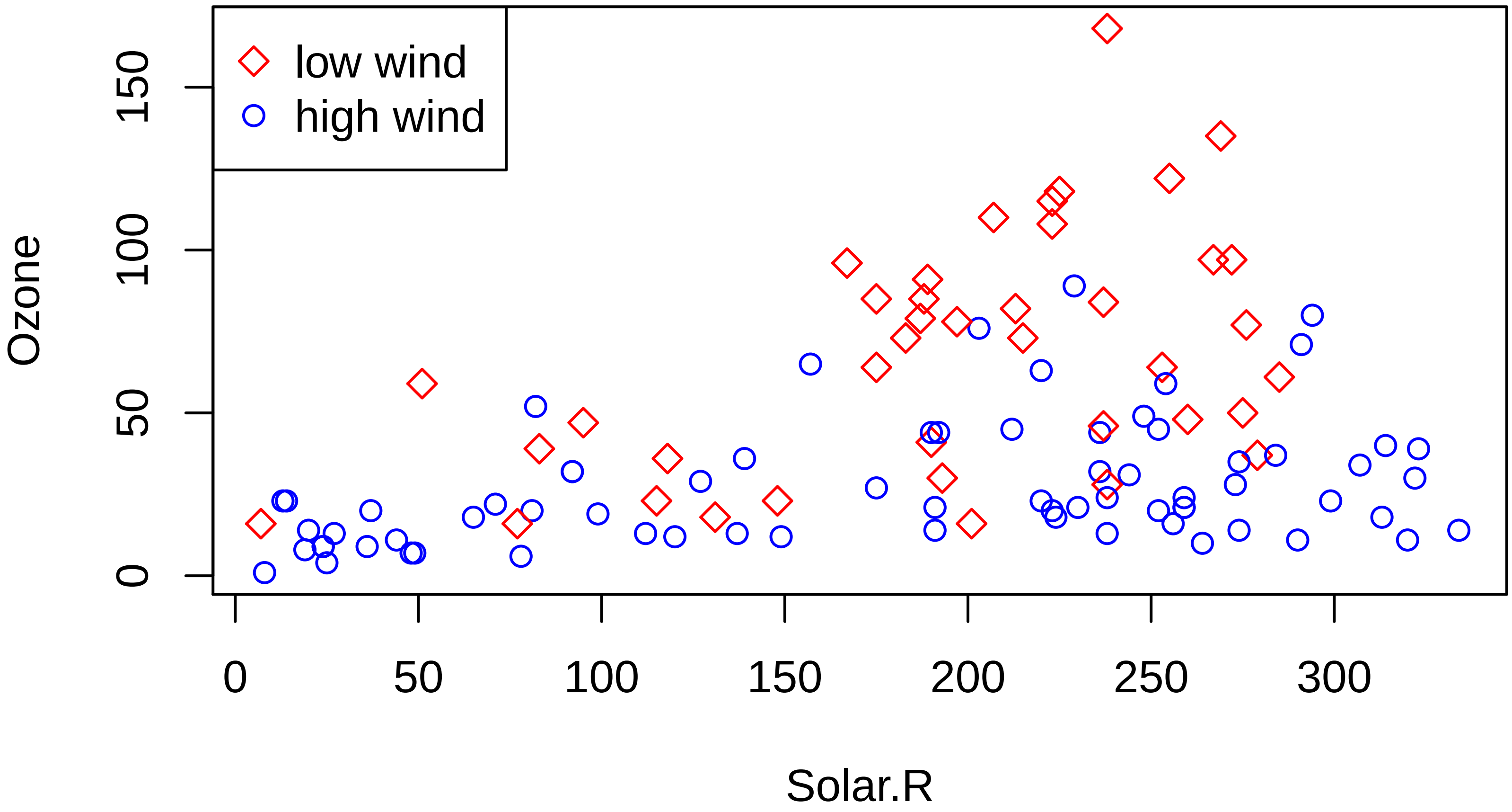
Including a legend is often essential for explaining symbols, colors, or line types used in a plot. The `legend()` command can be used to add a legend to an existing plot:

- The position of the legend can be specified by (x,y) coordinates or by using preset positions:
  - `legend(x,y,c("name1","name2"), pch=c(1,5)` adds a legend to the plot with its top-left corner at coordinate (x,y)
  - `legend("topright",c("name1","name2"),pch=c(1,5)` adds a legend in the top right corner of the plot. Also can use "bottom", "bottomleft", "left", "topleft", "top", "topright", "right" and "center".

Adding a legend to a plot

Options such as symbols (*pch*), colors (*col*), and line types(*lty*) can be specified in the legend command. See `?legend` for more details.

```
lowwinds <- ifelse(airquality$Wind<=8, "red", "blue")
symbols <- ifelse(airquality$Wind<=8, 5,1)
plot(Ozone~Solar.R,data=airquality,col=lowwinds,pch=symbol)
legend("topleft",c("low wind","high wind"),col=c("red","blue"),
              pch=c(5,1))
```


◇ low wind
○ high wind
Ozone
0
50
100
150
Solar.R
0
50
100
150
200
250
300

Smoothing

A straight line may not adequately represent the relationship between two variables.

Smoothing is a way of illustrating the local relationship between two variables over parts of their ranges, which may differ from their global relationship.

Locally weighted scatterplot smoothing (LOWESS) can be performed in `R` with the `lowess()` function, which calculates a smooth curve that fits the relationship between y and x locally.

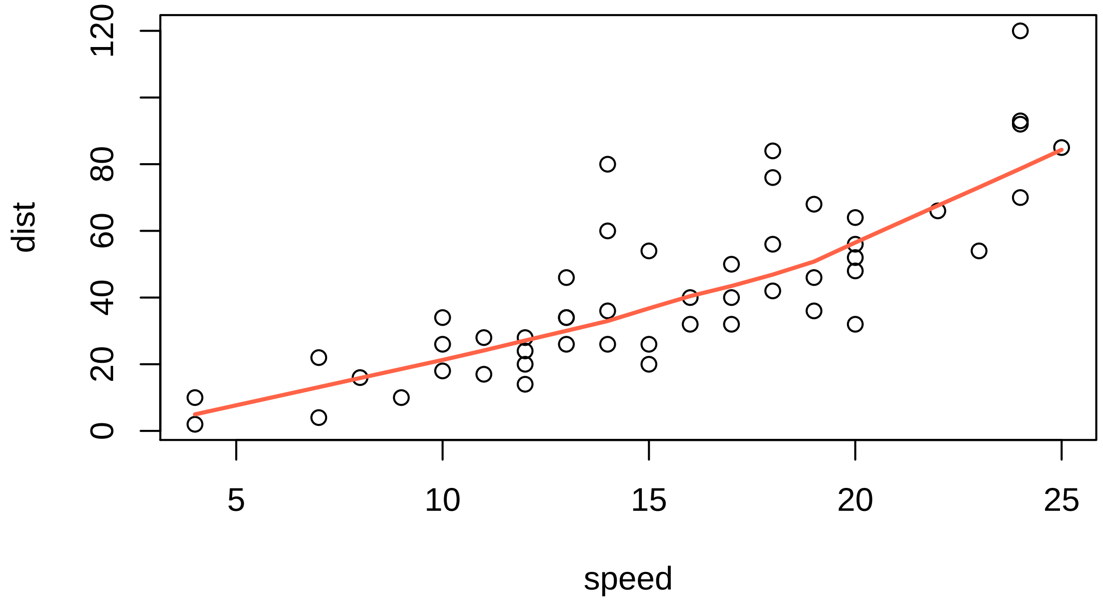
The `supsmu()` function can also be used for smoothing.

The output from both smoothing functions have attributes `$x` and `$y` that can be used with the generic plotting function `lines()`

Smoothing

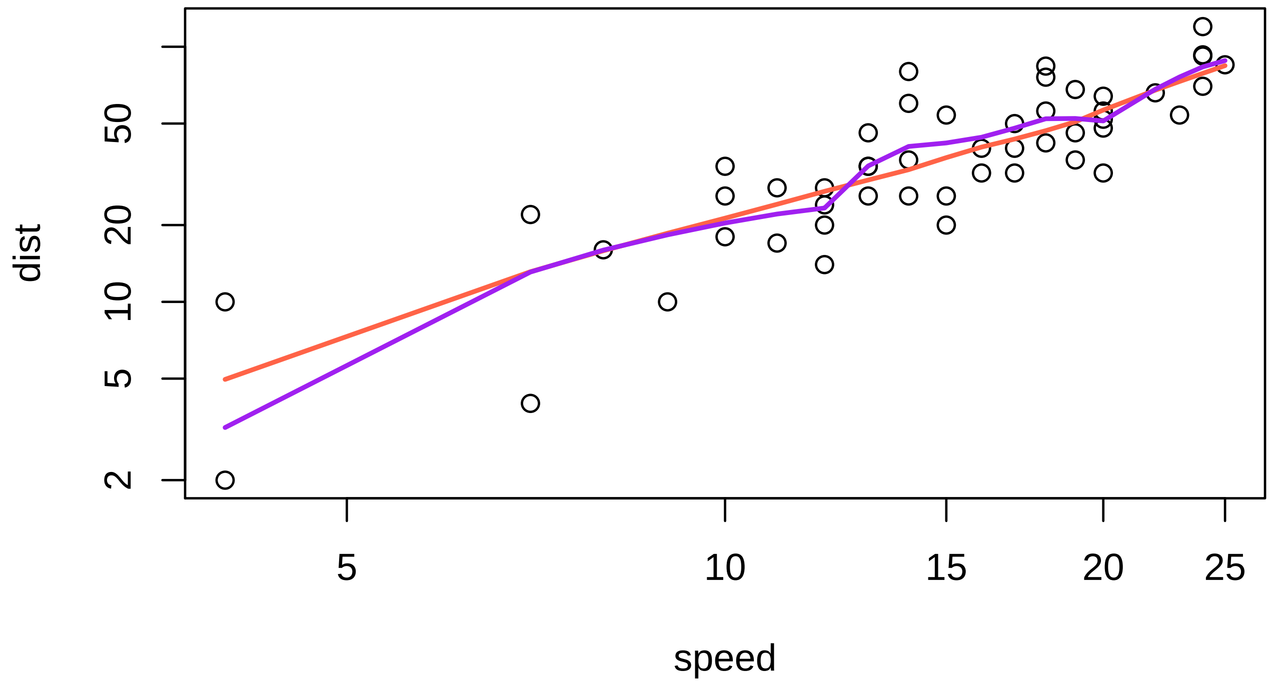
Consider the built-in dataset *cars*.

```
data(cars)
plot(dist~speed,data=cars)
with(cars, lines(lowess(speed, dist), col="tomato", lwd=2))
```



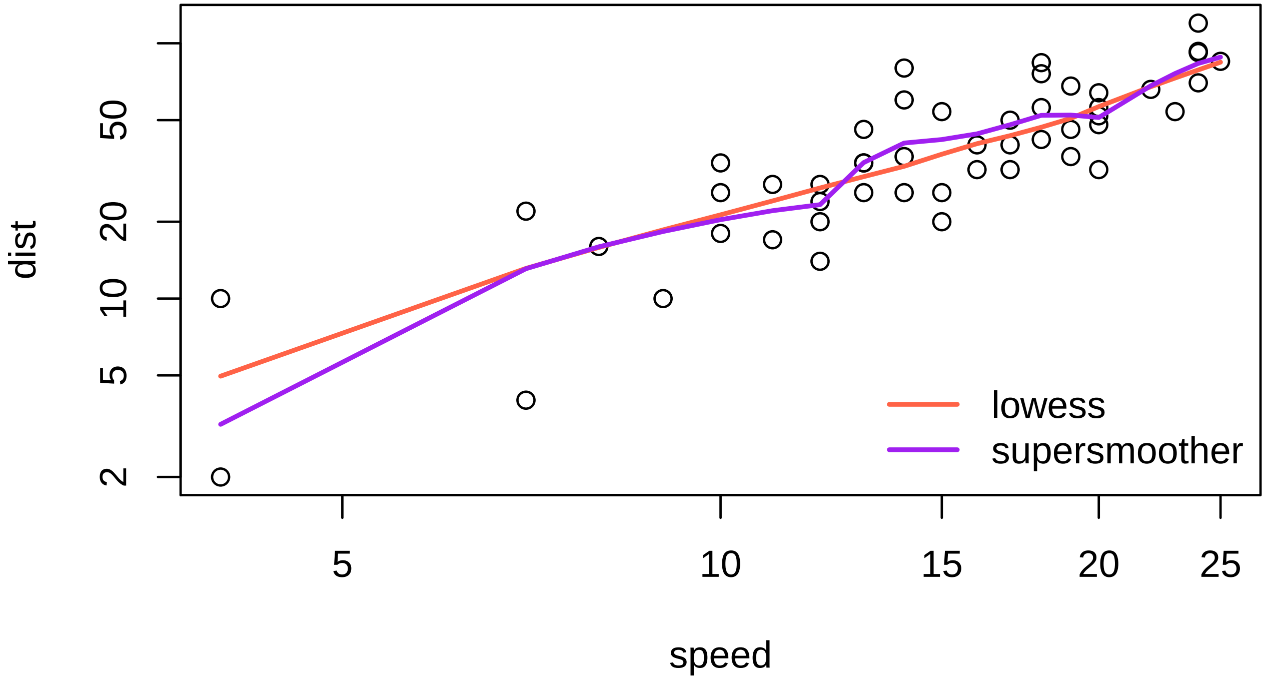
Smoothing

```
plot(dist~speed,data=cars, log="xy")
with(cars, lines(lowess(speed, dist), col="tomato", lwd=2))
with(cars, lines(supsmu(speed, dist), col="purple", lwd=2))
```



Smoothing

```
legend("bottomright", legend=c("lowess","supersmoother"),bty="n",
  lwd=2, col=c("tomato","purple"))
```



Multiple plots in a single figure

The `par()` and `layout()` functions can be used for drawing several plots in one figure.

`par()` with the option **mfrow**=c(*nrows*,*ncols*) creates a matrix of *nrows* × *ncols* plots that are filled in by row. Using the option **mfcol**=c(*nrows*,*ncols*) fills in the matrix by columns.

`layout(mat)` allows for a more customized panel with multiple plots, where *mat* is a matrix object that specifies the locations of the plots in the figure.

Multiple plots in a single figure

The *ToothGrowth* dataset, supplied with R, contains data from a study on the the effect of vitamin C on tooth growth in guinea pigs.

There are two treatments/supplement types: orange juice and ascorbic acid

There are three vitamin C dose levels for each of the two treatments: 0.5, 1, and 2mg

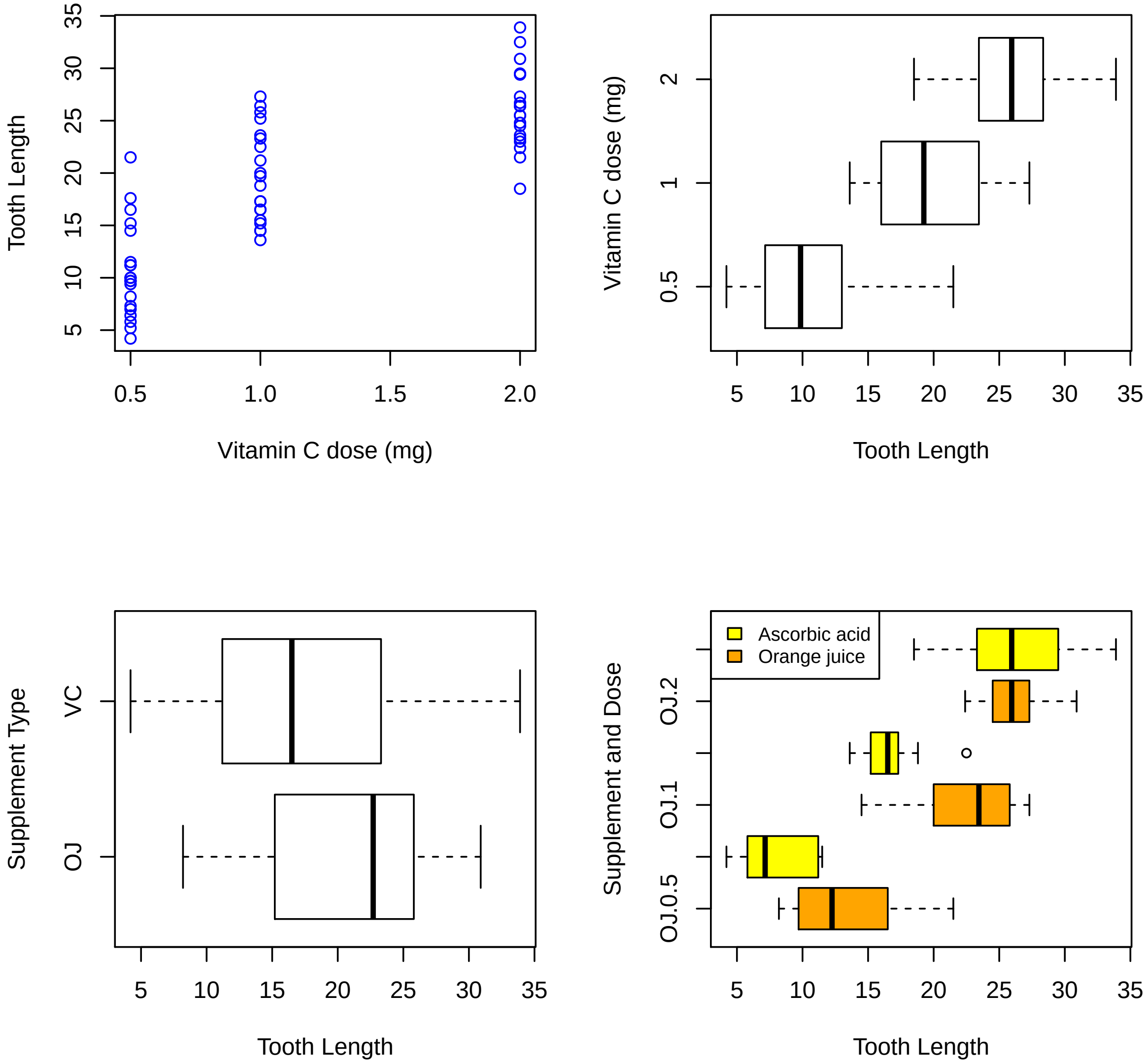
The response is the length of odontoblasts (teeth)

Multiple plots in a single figure

Below are commands for plotting multiple figures with the *ToothGrowth* dataset using `par()`

```
data(ToothGrowth)

par(mfrow=c(2,2))

plot(len~dose,data=ToothGrowth,xlab="Vitamin C dose (mg)",ylab="Tooth Length",
col="blue",cex.main=.8)

boxplot(len~dose,data=ToothGrowth,horizontal=TRUE,ylab="Vitamin C dose (mg)",
xlab="Tooth Length",cex.main=.8)

boxplot(len~supp,data=ToothGrowth, horizontal=TRUE,ylab="Supplement Type",
xlab="Tooth Length",cex.main=.8)

boxplot(len~supp*dose, data=ToothGrowth,horizontal=TRUE,
col=(c("orange","yellow")),  ylab="Supplement and Dose",xlab="Tooth Length")

legend("topleft",c("Ascorbic acid", "Orange juice"),fill = c("yellow", "orange"))
```

Multiple plots in a single figure

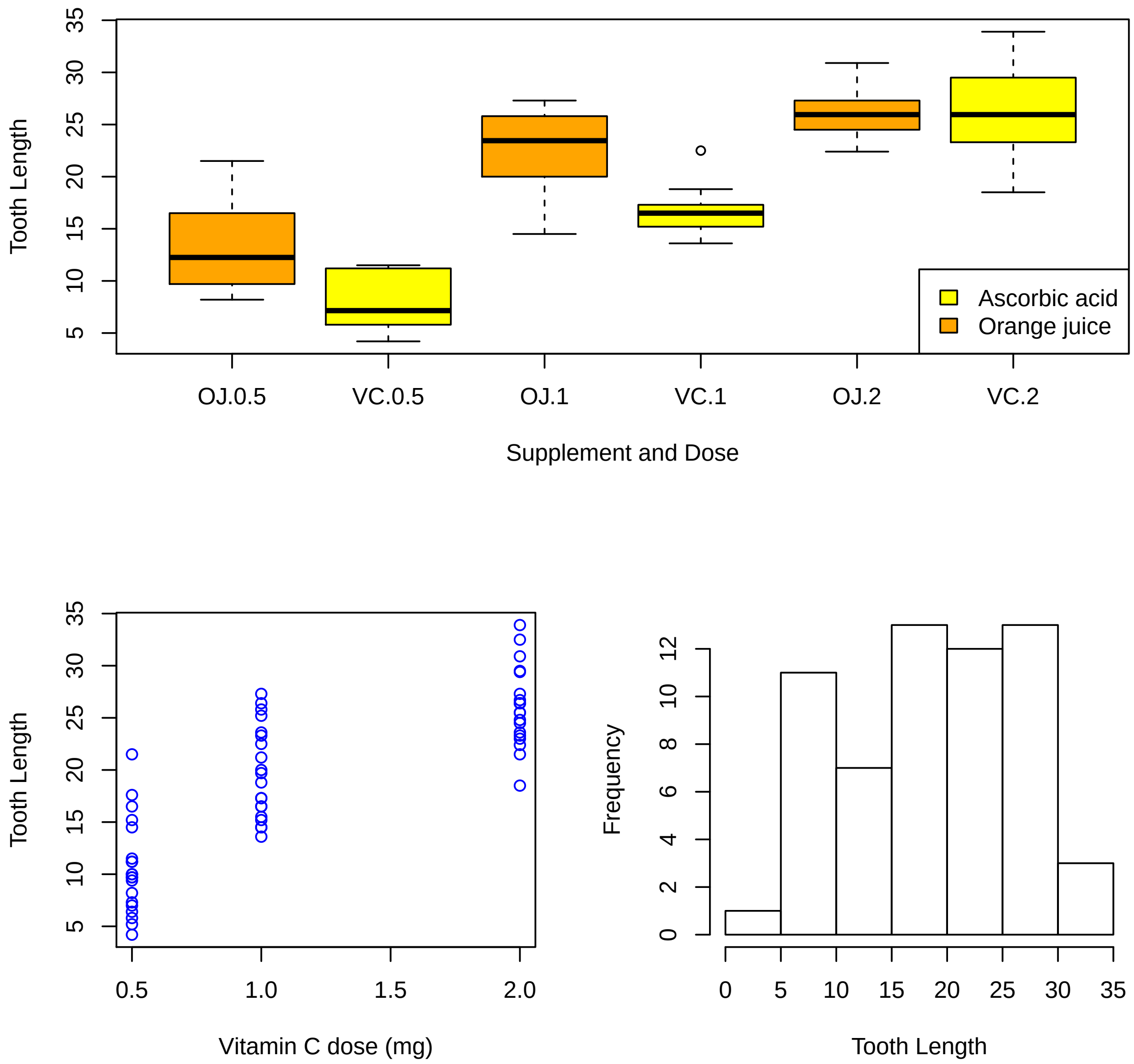


Multiple plots in a single figure

Below are commands for a more customized multiple-plot figure using `layout()`

```
layout(matrix(c(1,1,2,3), 2, 2, byrow = TRUE))

boxplot(len~supp*dose, data=ToothGrowth,col=(c("orange","yellow")),
xlab="Supplement and Dose",ylab="Tooth Length")

legend("bottomright",c("Ascorbic acid", "Orange juice"),
fill = c("yellow", "orange"))

plot(len~dose,data=ToothGrowth,xlab="Vitamin C dose (mg)",
ylab="Tooth Length",col="blue",cex.main=.8)

hist(ToothGrowth$len,xlab="Tooth Length",main="",cex.main=.8)
```

Multiple plots in a single figure



Summary

- R has a variety of plotting options
- `points()` adds points to an existing plot and `lines()` adds connected points by lines to an existing plot without symbols
- `abline()` draws a single straight line on a plot
- `lowess()` and `supsmu()` are scatterplot smoothers
- `legend()` adds a legend to a plot
- `par()` and `layout()` can be used for multi-panel plotting

5. Over and over

Ken Rice
Tim Thornton

University of Washington

Seattle, July 2013

In this session

In Sessions 1–4, we completed tasks by breaking them down, into one line of an R script at a time. In principle, we could do *everything* this way. But;

- Repeating the same job many times (i.e. once for each person in the dataset) the typing gets slow & tedious, and is error prone
- For iterative methods, we don't know how much code will be needed before starting the task

This session, and the next, introduce writing loops, so we can re-use the same code in a script, without re-typing it.

NB This module does *not* cover every R tool for looping.

A very first `for()` loop

Many people's first computer program looks like this;

```
> for(i in 1:5){
+   print("hello world!")
+   print(i^2)
+ }
[1] "hello world!"
[1] 1
[1] "hello world!"
[1] 4
[1] "hello world!"
[1] 9
[1] "hello world!"
[1] 16
[1] "hello world!"
[1] 25
```

Two fundamental ideas;

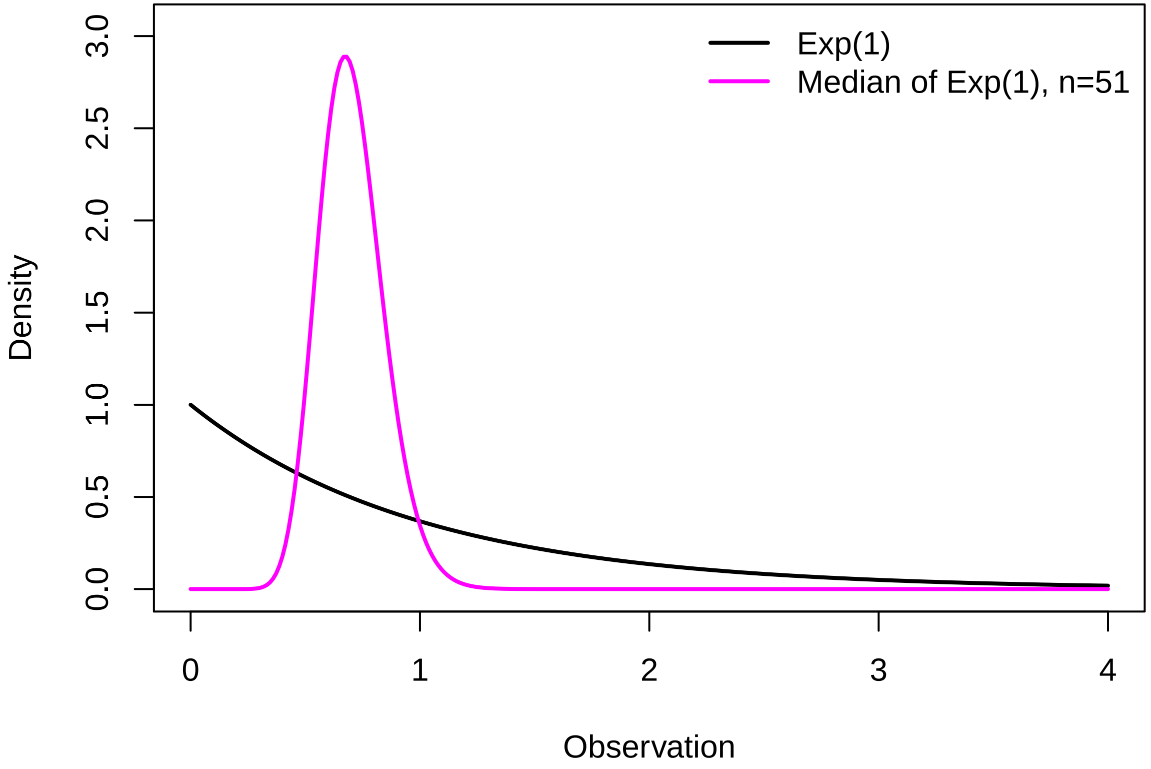
- Go round the loop 5 times
- Each time, do something that may (or may not) depend on which 'go round' it is

Of course, `for()` loops also have more practical uses...

Example: hard math made easy

A question from analysis of survival traits – and its answer!

What is the expected value of the median of a sample, size $n = 51$*, of independent data from* $Exp(1)$*? What is its variance?*



Example: hard math made easy

If the picture didn't make it obvious enough, here are the *exact* answers;

$$\begin{aligned}\mathbb{E}[\,\text{Median}_{51}\,] &= \frac{21781789365391086741 53}{30990445042459967064 00}\\ \mathbb{E}[\,\text{Median}^2_{51}\,] &= \frac{246728231606366796745923323213925797680195 9}{480203841964865774900127881537982390048000 0}\end{aligned}$$

These are 0.70286 and 0.51380 to 5 d.p. – so the variance is $0.51380 - 0.70286^2 = 0.01978$.

- Yes, there are 'pretty' answers here
- In general there aren't – but the 'expectation' ($\mathbb{E}[\ldots]$) terms just mean averaging over lots of datasets – which is easy, with a computer
- We can get a good-enough answer very quickly

Example: hard math made easy

We'll write code that;

1. Generates samples of size $n = 51$ from $Exp(1)$
2. Calculates their median and stores this number
3. Does steps 1 and 2 many times, then works out the mean and variance of the stored numbers

Here are steps 1 and 2 – run them and see what's created;

```
many.medians <- vector(10000, mode="numeric") # or just rep(NA, 10000)
set.seed(4)
for(i in 1:10000){
   mysample <- rexp(n=51, rate=1)        # take a sample, size 51
   many.medians[i] <- median(mysample)  # calcuate & store its  median
   }
```

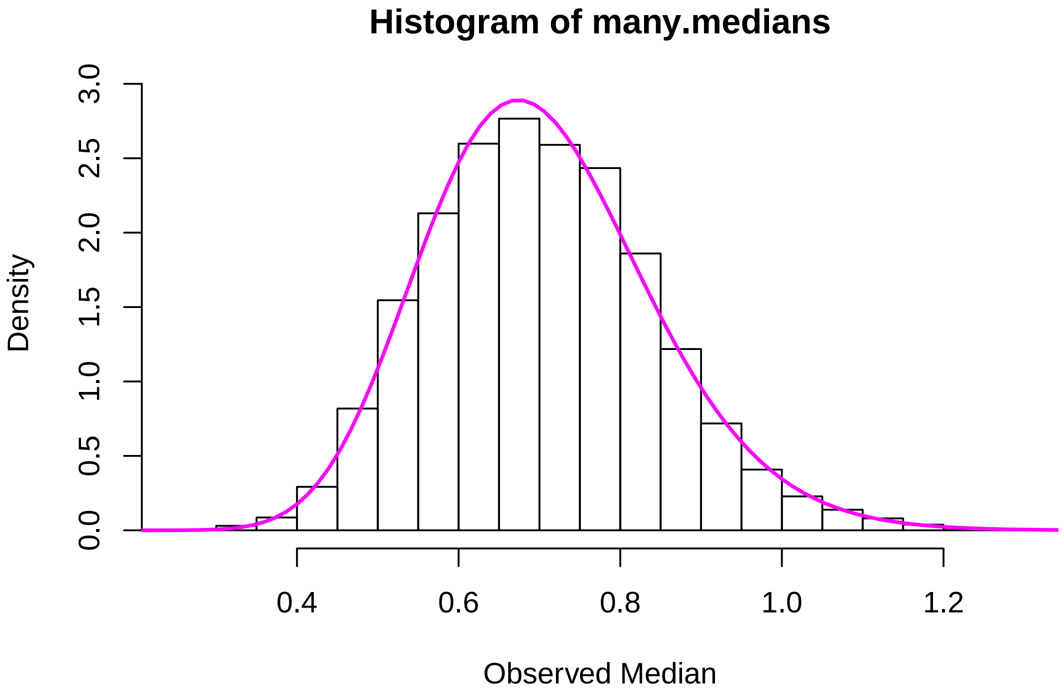
The function `set.seed()` tells R where to start its random-number generator – this is important, as it means we can repeat the code and get the same answers. Choose any 'seed' you like.

Example: hard math made easy

And the answers, from 10,000 simulations?

```
> mean(many.medians)
[1] 0.702171          # exact answer is 0.70286
> var(many.medians)
[1] 0.01955728        # exact answer is 0.01978
```

NB: for large-enough values of 10000, we could work basically *anything* about the sample median, with little extra work;



Example: hard math made easy

Notes on the coding; (NB see `?Control` for the help page on `for()`, `?for` won't work as `for` is 'restricted')

- `for( [iteration] in [vector of iteration values] )` – the vector of iteration values can be of anything, not just `1:n`
- The expression between the curly brackets { } is evaluated each 'go round' the loop, substituting `i` for 1,2, ... 10,000 in turn
- Very important – create an object to store the output first (but no need to create `i` first). To do this, you'll need to know how big the output is going to be.
- Last-used version of objects used (`i, mysample`) are available when the loop terminates – which is very helpful, if (when!) an error occurs
- We used `rexp()`, but there are *many* built-in distributions; `rnorm()`, `rgamma()`, `rbinom()`, `rpois()` etc

Example: data manipulation

Recall the salary example – on faculty measured over several years. Suppose we were interested in the *final* observation for each person – how to construct that dataset?

- Different numbers of observations per person – so can't just look at e.g. rows 1,5,11,15, ... (but see `seq()` if you *do* want to do this)
- Different entry and exit years
- `subset()` won't work, neither will use of square brackets

Instead, we can go through every `id` number, pull out the rows with that `id` and record the one for which `year` is highest. Or, if the data is sorted first (by `id` and `time`) pull out the *last* row for each `id` number.

As before, it's very important that we prepare an object for the results ('pulled out' data, here) *before* running any loops.

Example: data manipulation

First sort the data, and make the empty object ready to take output;

```
salary <- salary[ order(salary$id, salary$year) , ]
View(salary) # check we know what we should get from subsetting

n <- length(unique(salary$id))
finalsalary <- salary[0,] # take just column names from salary
finalsalary[1:n,] <- NA   # fill in with missing values
str(finalsalary)          # check the structure we made
```

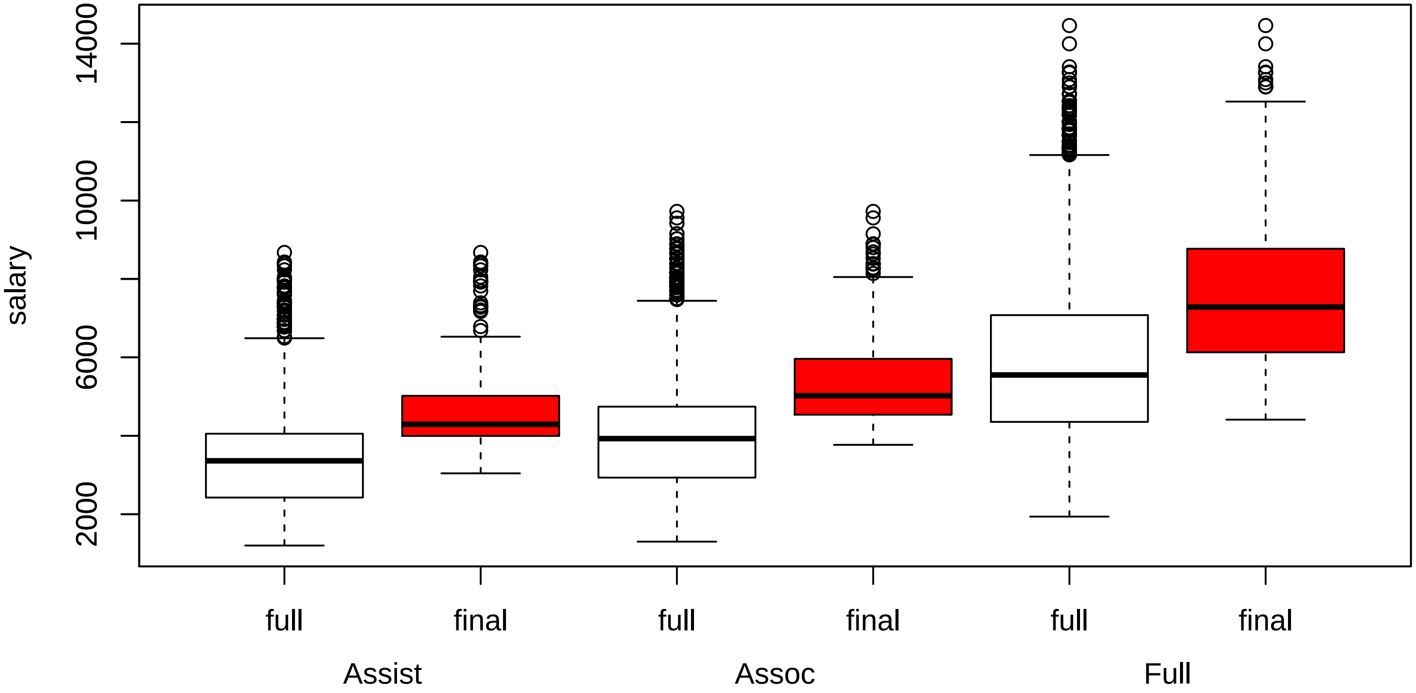
- `order()` returns the vector that puts objects in order. There is a `sort()` function, but it accepts only vectors and not data frames
- A less-sneaky way to make a new empty data frame uses e.g. `data.frame(id=NULL, age=NULL, sex=NULL)`
- In RStudio, `View()` operates in the Source window; in vanilla R it opens up a new window. Neither refreshes automatically

Example: data manipulation

Now for the loop;

```
for(i in 1:n){
   id.i             <- unique(salary$id)[i]
   salary.i         <- subset(salary, id==id.i)
   n.i              <- dim(salary.i)[1]        # dim() for dimension
   finalsalary[i,] <- salary.i[n.i,]          # i.e. just the last row
   } # View(finalsalary) a good idea, to check it worked
```

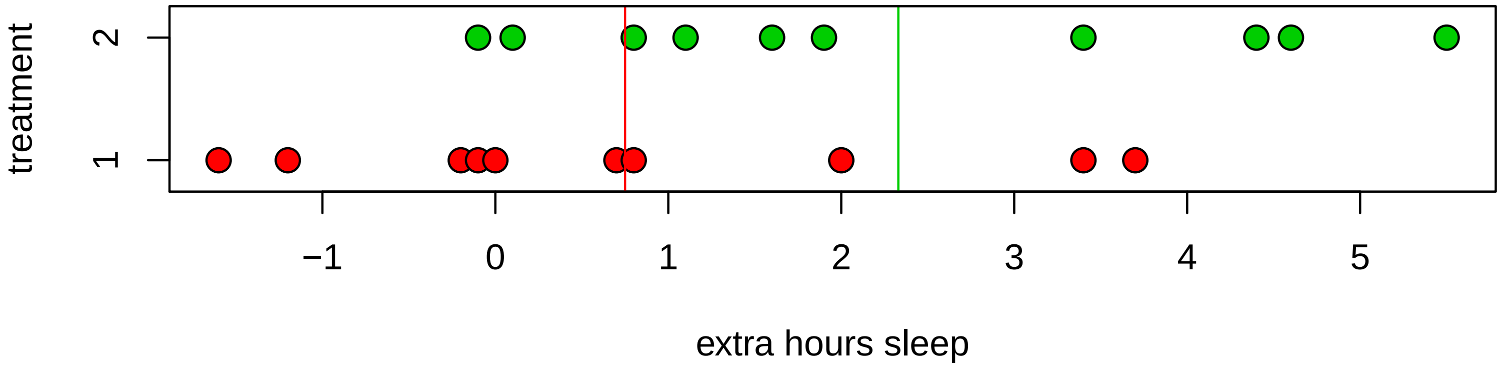
Compare the full dataset (white) and final-only version (red);



Example: permutation test

A classical statistical question: are the data we've observed *unexpected*, if there's nothing going on?

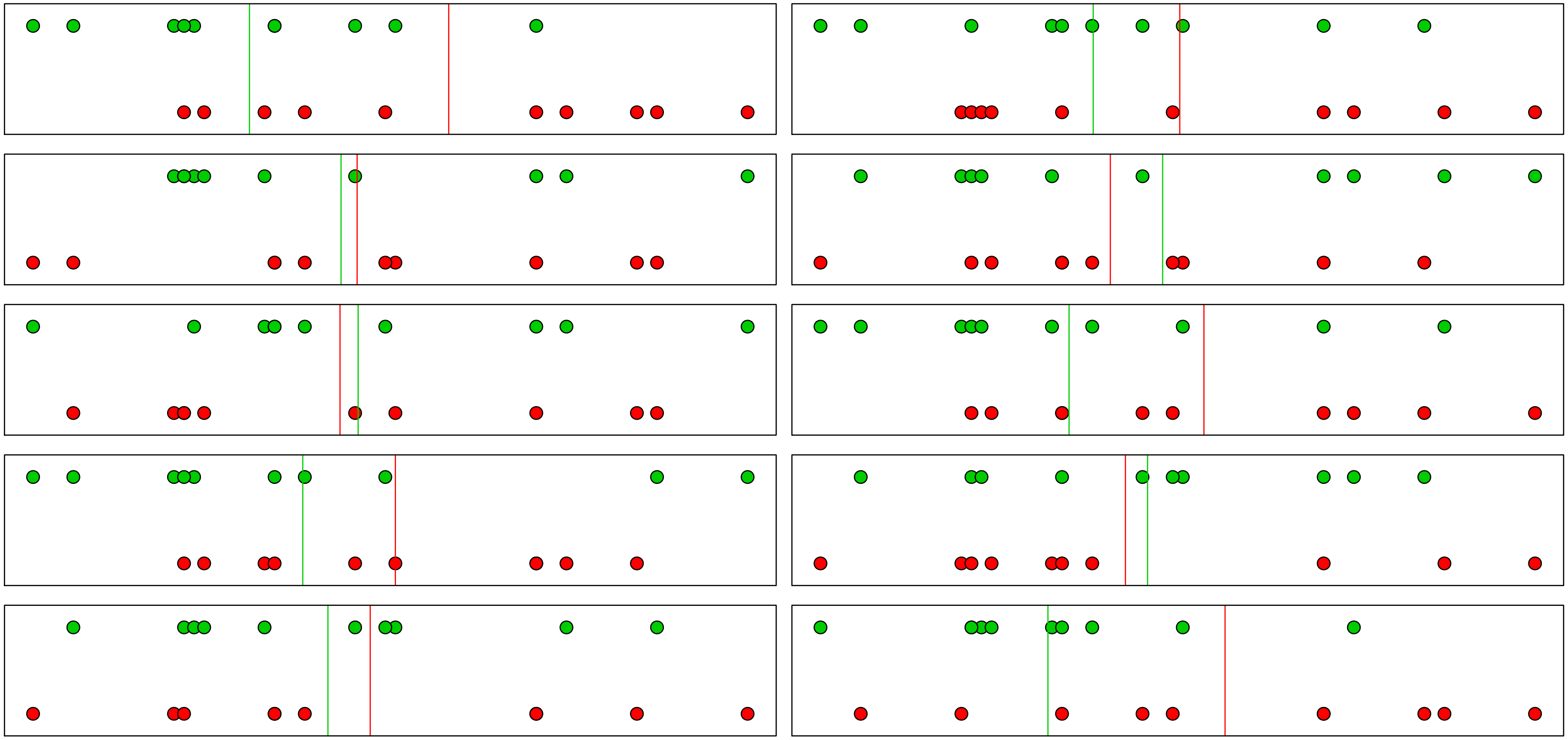
An example where we can answer this is R's `sleep` data;



- 10 subjects per group
- Groups receive different treatments, we record how many hours sleep they get, compared to baseline
- Mean extra hours sleep is higher in group 2 (2.33 hrs vs 0.75 hrs, so difference is 1.58 hrs)

Example: permutation test

What if there were nothing going on*, i.e. what if any differences in mean were just chance? If so, the data we saw would be just as likely as that obtained assigning the group labels *at random*;



[*] Formally, what if the *null hypothesis* of equal means held, in the population from which this data has been sampled?

Example: permutation test

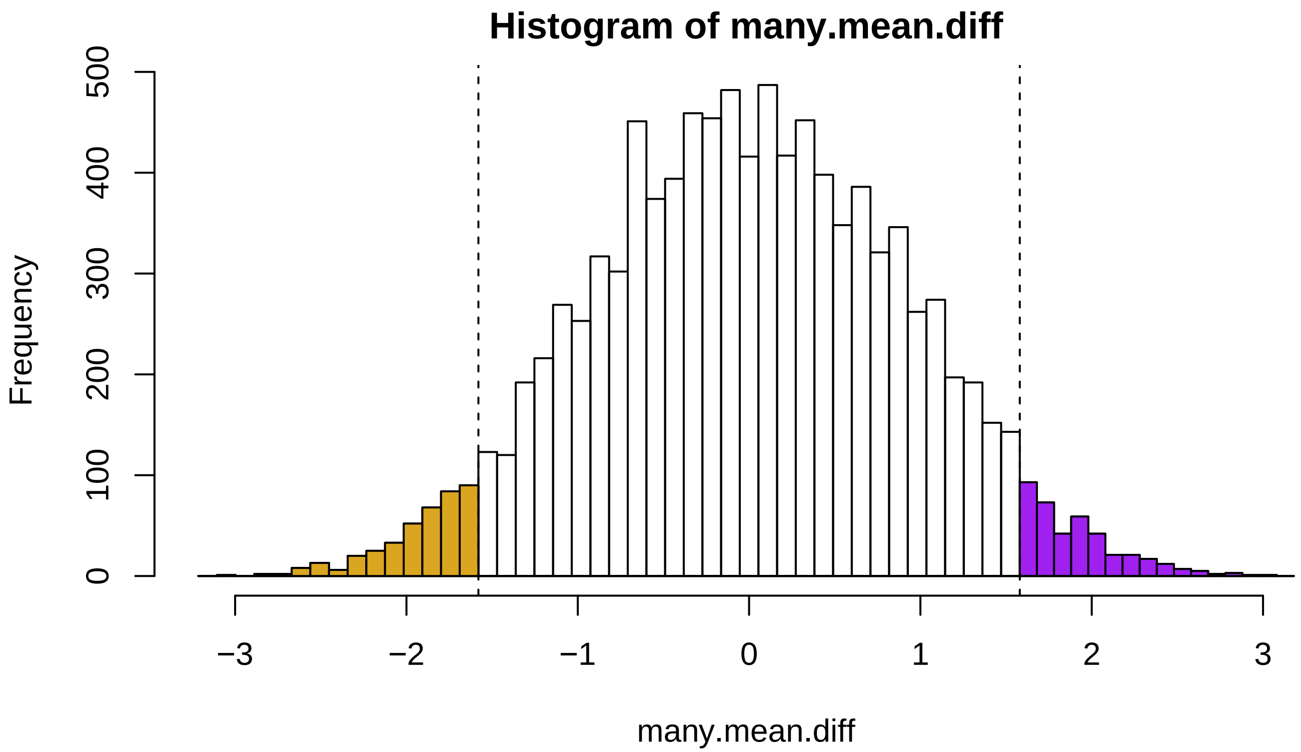
To measure how unexpected our data is, we compute the red/green difference in means for many of these *permutations*, and see how the *observed* data compares.

```
orig.mean.diff <- with(sleep,
      mean(extra[group==2]) - mean(extra[group==1])
      )
orig.mean.diff

many.mean.diff <- vector(10000, mode="numeric") # set up place to put output
set.seed(4)                                     # set the random seed

for(i in 1:10000){                              # do this bit 10,000 times
  group.shuffle <- sample(sleep$group)
  many.mean.diff[i] <- with(sleep,
    mean(extra[group.shuffle==2]) - mean(extra[group.shuffle==1])
    )
}
```

- `sample()` returns a random shuffle of a vector
- The same calculation is made, for the original data and the shuffled version; the difference in means is called the *test statistic*

Example: permutation test

How does original data (w/ mean diff=1.58) compare to these?



```
> table(many.mean.diff>orig.mean.diff)
FALSE  TRUE
 9601   399
> mean(many.mean.diff>orig.mean.diff)
[1] 0.0399
> mean(abs(many.mean.diff)>abs(orig.mean.diff))
[1] 0.0789
```

Example: permutation test

- The proportion of sample in the RH tail is a (valid) p-value for a one-tailed test, where the alternative is that green $>$ red. $p = 0.04$, here
- The proportion in both tails is the p-value for a two-tail test; $p = 0.079$
- There is some ‘Monte Carlo’ error in these p-values; roughly ±0.004 here, i.e. 2 decimal places in p. If that’s not good enough, use more permutations. (Here, could use all 184,756 – but in larger samples it’s not possible)

To get a quicker (but approximate) version of the same thing;

```
> t.test(extra~group, data=sleep) # recall extra ‘depends on’ group
Welch Two Sample t-test
data:  extra by group
t = -1.8608, df = 17.776, p-value = 0.07939
alternative hypothesis: true difference in means is not equal to 0
```

The t test makes fewer assumptions than most people think!

Notes on timing, and speed

Doing a lot of calculations can take a long time – it's useful to know how long. Try out the `system.time()` command on a smaller version of the problem, i.e.

```
system.time({
  for(i in 1:1000){                       # just 1000, not 10000
    group.shuffle <- sample(sleep$group)
    many.mean.diff[i] <- with(sleep,
      mean(extra[group.shuffle==2]) - mean(extra[group.shuffle==1])
    )
  }
})
```

This returns the time taken to run the outer curly brackets;

```
  user  system elapsed
   0.57    0.00    0.60
```

... so running 100,000 permutations would take `100*0.6/60` = 1 minute, roughly. (NB this is much less time than it took to write the code!)

If RStudio hangs, there is a 'STOP' button on the Console window; in vanilla R hit Escape, or Ctrl-D.

Notes on timing, and speed

Throughout, we have stressed the importance of setting up empty objects for the loop's output. Why? Let's code the permutation test without doing this;

```
many.mean.diff <- NULL          # this will ‘grow’, in the loop
system.time({
  for(i in 1:100000){
    group.shuffle <- sample(sleep$group)
    mean.diff <- with(sleep,
       mean(extra[group.shuffle==2]) - mean(extra[group.shuffle==1]))
    many.mean.diff <- c(many.mean.diff, mean.diff) # ‘grow’ the dataset
  }})

   user  system elapsed # CPU/child process/total
 115.53    4.93  122.07
```

- This works, but at half the speed of the other version
- The extra time is *all* spend copying vector `many.mean.diff` – R is slow at copying objects
- The slowdown is worse for larger objects, i.e. gets worse with more permutations, i.e. when speed really matters

Notes on timing, and speed

Compared to using a single R command (when available) to do the job, `for()` loops can be inefficient.

- Add two vectors (`x <- y + z`) don't add them element by element (`for(i in 1:n){ x[i] <- y[i] + z[i]}`)
- Recall `ifelse()` earlier, rather than looping over a vector.

```
many.samples <- matrix(data=NA, nrow=100000,ncol=20)

for(i in 1:100000){
    many.samples[i,] <- sample(sleep$extra)
}
many.mean.diff <- rowMeans(many.samples[,1:10]) - rowMeans(many.samples[,11:20])
```

- Shuffling the outcomes is equivalent to shuffling the groups
- A `matrix` has all entries of the same type – less flexible than a `data.frame`, but faster to work with
- This version takes 4.3s, i.e. it's ×14 faster than the loop.
- Not available for every task – also uses more memory

Summary

- Writing loops saves a lot of typing – essential for serious computing jobs, but helpful for data management too

- `for()` loops offer enough flexibility for several jobs – more to come in the next session!

- As with all programming; break the job into lots of small pieces, and do each one in turn

- Never never *never* grow the output

- Other methods exist in R – but not in this module

6. More Loops, Control Structures, and Bootstrapping

Ken Rice
Timothy Thornotn

University of Washington

Seattle, July 2013

In this session

We will introduce additional looping procedures as well as control structures that are useful in R. We also provide applications to bootstrapping.

- Repeat and While loops,
- If-Then and If-Then-Else structures
- Introduction to the bootstrap, with examples

Repeat loops

The repeat loop is an infinite loop that is often used in conjunction with a **break** statement that terminates the loop when a specified condition is satisfied. The basic structure of the repeat loop is:

```
repeat {
expression
expression
expression
if(condition)
break
}
```

Repeat loops

Below is a repeat loop for printing the square of integers from 1 to 10.

```
i <- 1
repeat {
print(i^2)
i <- i+1
if(i > 10)
   break
}
```

While loops

The while loop is used when you want to keep iterating as long as a specific condition is satisfied. The basic structure of the while loop is:

```
while (condition) {
    commands
}
```

While loops

Below is a while loop for printing out the first few Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13,..., where each number is the sum of the previous two numbers in the sequence.

```
a = 0
b = 1
print(a)
while (b < 50) {
    print(b)
    temp = a + b
    a = b
    b = temp
}
```

While loops

Below is a while loop that creates a vector containing the first 20 numbers in the Fibonacci sequence

```
x = c(0,1)
n=20
while (length(x) < n) {
    position = length(x)
    new = x[position] + x[position-1]
    x = c(x,new)
}
```

If-Then and If-Then-Else structures

Sometimes a block of code in a program should only be executed if a certain condition is satisfied. For these situations, *if-then* and *if-then-else* structures can be used:

The *if-then* structure has the following general form:

```
if (condition)

{expression
 expression}
```

The *if-then-else* structure has the following general form:

```
if (condition)
{expression
 expression} else
 {expression
  expression}
```

If-Then and If-Then-Else structures

Below is an *if-then-else* statement that takes the square root of the product of two numbers x and y if the product is positive:

```
x=3
y=7
  if( (x<0 & y<0)| (x>0 & y>0))
 {myval=sqrt(x*y)} else
 {myval=NA}
```

And the value of myval when x=3 and y=7 is:

```
> myval
[1] 4.582576
```

What is myval if x=2 and y=-10?

```
> myval
[1] NA
```

Introduction to bootstrapping

Bootstrapping is a very useful tool when the distribution of a statistic is unknown or very complex.

Bootstrapping is a non-parametric resampling method that allows for the computation of standard errors and confidence intervals, as well as hypothesis testing.

The method is often used when sample sizes are small and asymptotic distribution assumptions, such as normality, may not be appropriate.

“The bootstrap is a computer-based method for assigning measures of accuracy to sample estimates.” [B. Efron and R. J. Tibshirani, An Introduction to the Bootstrap, Boca Raton, FL: CRC Press, 1994.]

Introduction to bootstrapping

Bootstrapping generally has the following three steps:

- Resample a given data set **with replacement** a specified number of time, where each ”bootstrap sample” is the same size as the original sample

- Calculate a statistic of interest for each of the bootstrap samples.

- The distribution of the statistic from the bootstrap samples can then be used to obtain estimated standard errors, create confidence intervals, and to perform hypothesis testing with the statistic.

Example: bootstrapping the median

Bootstrapping can be easily implemented in R using loops.

The `sample(x, size, replace, prob)` function is very useful for resampling a given data set in `R`:

- The first argument of `sample` is a vector containing the data set to be resampled or the **indices of the data** to be resampled.

- The *size* option specifies the sample size, with the default being the same size as the data set being resampled.

- The *replace* option determines if the sample will be drawn with or without replacement where the default value is FALSE, i.e., sampling is performed without replacement.

Example: bootstrapping the median

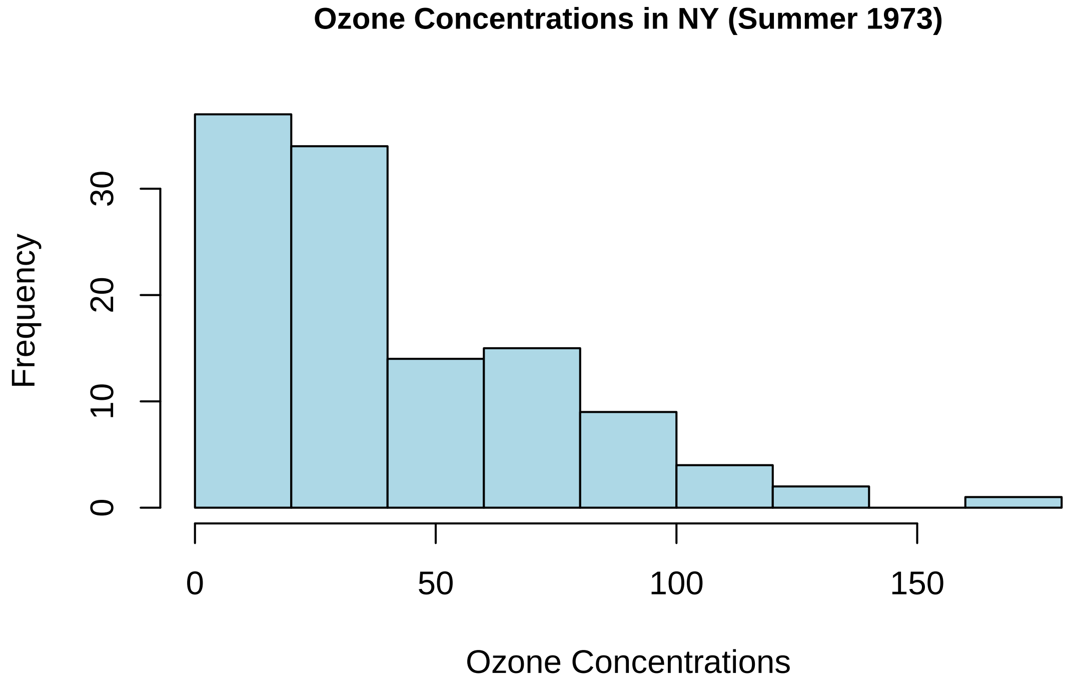
- The *prob* option takes a vector of length equal to the data set given in the first argument containing the probability of selection for each element of x. The default setting has each element with equal probability of being sampled.

In a typical bootstrapping situation the bootstrapping samples will be the same size as the data set being sampled and sampling will be done with replacement.

Example: bootstrapping the median

Let's consider the *airquality* dataset again. Below is a histogram of the daily ozone concentrations in New York, summer 1973.

```
hist(airquality$Ozone,col="lightblue",xlab="Ozone Concentrations",
main="Ozone Concentrations in NY (Summer 1973)")
```



What is the median ozone concentration level?

Example: bootstrapping the median

```
> median(airquality$Ozone)
[1] NA    # There are missing  daily ozone concentration values

> median(airquality$Ozone,na.rm=TRUE)
[1] 31.5
```

So the median is estimated to be 31.5.

Can we obtain a 95% confidence interval for the median? What is the distribution of the median ozone concentration?

Bootstrapping can be implemented for this!

Example: bootstrapping the median

We first obtain a vector of the ozone concentrations with missing values excluded:

```
ozone=airquality$Ozone[!is.na(airquality$Ozone)]
```

Using a `for()` loop, we can create 10,000 bootstrap samples and calculate the median for each sample:

```
nboot <-10000    #number of bootstrap samples
bootstrap.medians <-rep(NA, nboot)
set.seed(10)
for(i in 1:nboot){
bootstrap.medians[i]<-median(sample(ozone,replace=TRUE))
}
```

Example: bootstrapping the median

From the bootstrap medians we can obtain the .025 and .975 quantiles:

```
alpha=.05
sort(bootstrap.medians)[nboot*alpha/2]
sort(bootstrap.medians)[nboot*(1-alpha/2)]
```

```
> sort(bootstrap.medians)[nboot*alpha/2]
[1] 23.5
> sort(bootstrap.medians)[nboot*(1-alpha/2)]
[1] 39
```

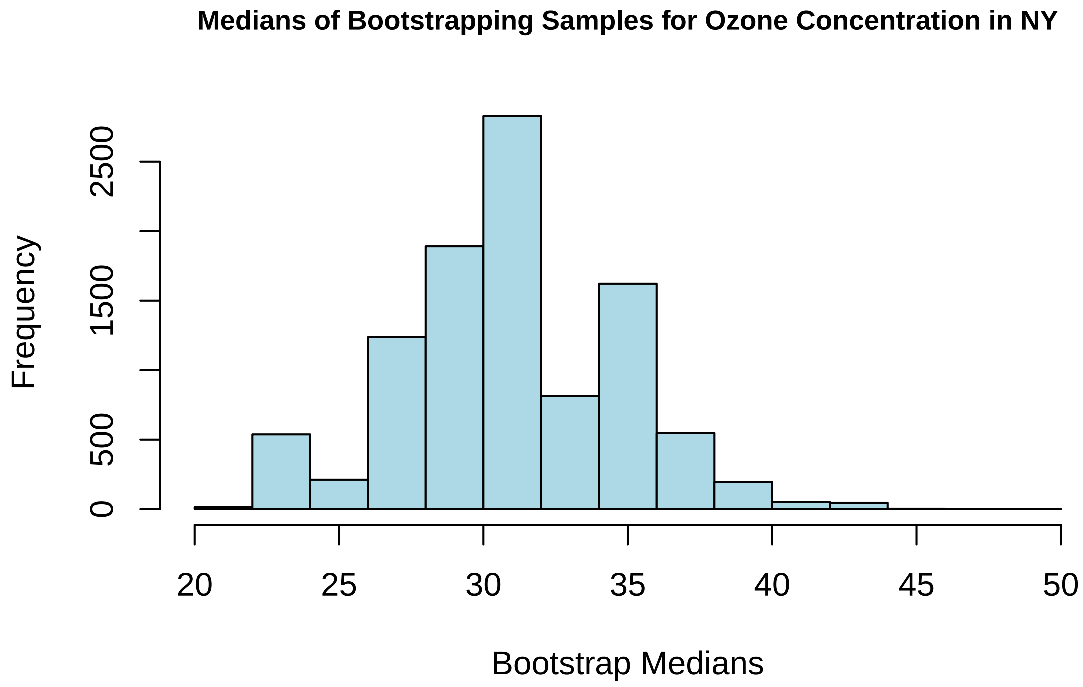
So our bootstrap 95% confidence interval for the median ozone concentration levels is **(23.5,39.0)**.

NB could also use `quantile(bootstrap.medians, c(0.025, 0.975))`

Example: bootstrapping the median

Below is a histogram of the medians from the 10,000 bootstrap samples;

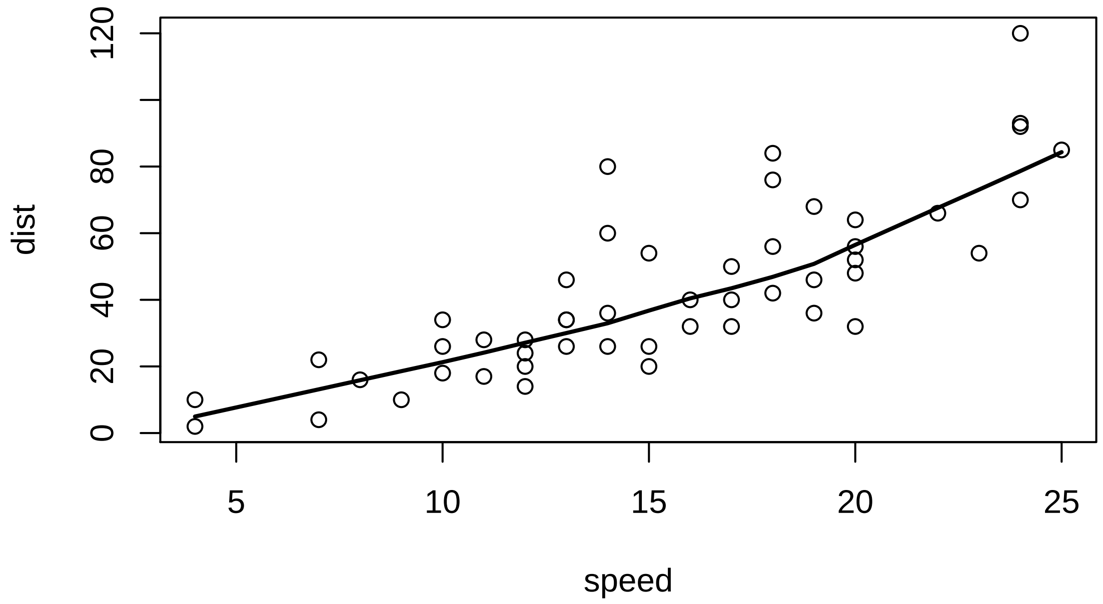
```
hist(bootstrap.medians,col="lightblue",xlab="Bootstrap Medians",
main="Bootstrap Medians for Ozone Concentrations in NY",cex.main=.8)
```



Example: bootstrap for lowess curve

Recall the `cars` data, and the line we put through it;

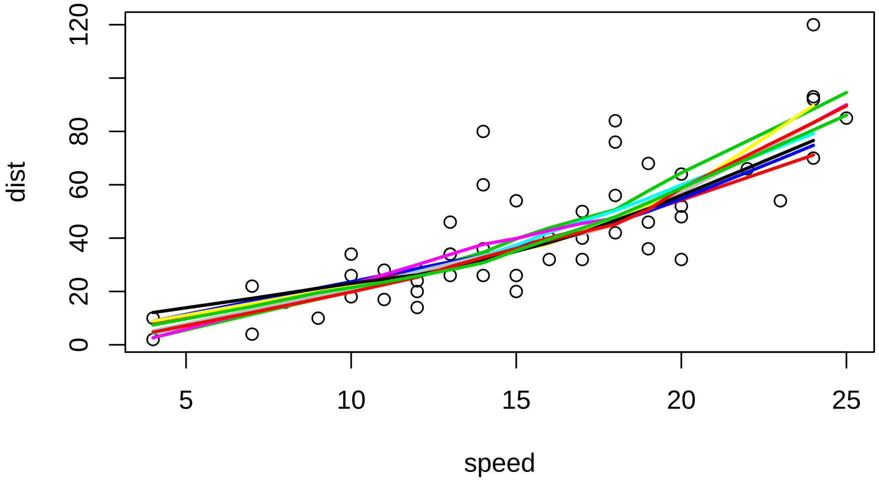
```
data(cars)
plot(dist~speed,data=cars)
with(cars, lines(lowess(speed, dist), col="tomato", lwd=2))
```



Example: bootstrap for lowess curve

To bootstrap the curve, a first step is to resample entire observations;

```
m=dim(cars)[1]     # obtain the sample size
nboot=20
for(i in 1:nboot){
   mysample <- sample(1:m,replace=T)
   with(cars, lines(lowess(speed[mysample], dist[mysample]),
                    col=(i+1), lwd=2)
        )}
```



Example: bootstrap for lowess curve

`lowess()` only produces output at the sampled points – so we extrapolate to the others using `approx()`;

```
nboot <- 1000
boot.speed <- matrix(NA, 1000,m)
set.seed(1314)
for(i in 1:nboot){
mysample <- sample(1:m,replace=T)
low1 <- with(cars, lowess(speed[mysample], dist[mysample]))
low.all <- approx(low1$x, low1$y, xout=cars$speed, rule=2)
boot.speed[i,] <- low.all$y
}
```

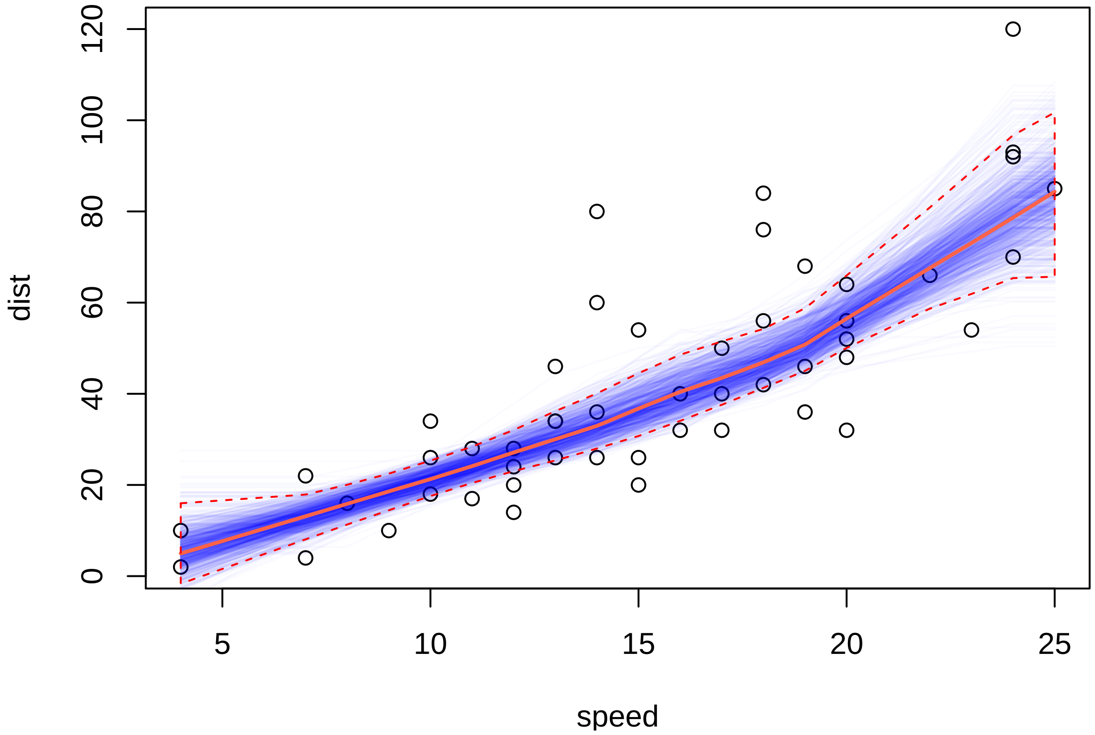
Now work out the lower and upper ranges of the lines, at each values of `speed`;

```
upper <- rep(NA, m)
lower <- rep(NA, m)
for(j in 1:m){
upper[j] <- quantile(boot.speed[,j], 0.975)
lower[j] <- quantile(boot.speed[,j], 0.025)}
```

Example: bootstrap for lowess curve

And make a nice picture;

```
plot(dist~speed,data=cars)
for(i in 1:nboot){
   lines(x=cars$speed, y=boot.speed[i,], col="#0000FF05")}
with(cars, lines(lowess(speed, dist), col="tomato", lwd=2))
polygon(x=c(cars$speed, rev(cars$speed)), y=c(upper, rev(lower)),
        density=0, col="red", lty=2)
```



Summary

- `while{}` and `repeat{}` are useful tools for looping until a condition is satisfied

- *if-then* and *if-then-else* structures allow blocks of code to be executed under different specified conditions

- bootstrapping is a powerful statistical technique when the distribution of a statistic is uknown

- bootstrapping can be very easily implemented in R using loops and the `sample()` function

7. Fitting models

Ken Rice
Tim Thornton

University of Washington

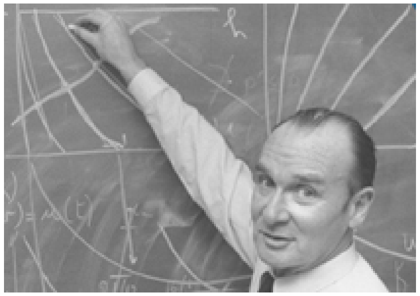
Seattle, July 2013

Disclaimer/Warning

In statistics, as in fashion, a model is an idealization of reality.

Peter McCullagh
JRSSD (1999) 48:1

Models basically play the same role in economics as in fashion: they provide an articulated frame on which to show off your material to advantage ...; a useful role, but fraught with the dangers that the designer may get carried away by his personal inclination for the model, while the customers may forget that the model is more streamlined than reality.



Jacques Drèze
Economic Journal (1985) 95:380

In this session

... we will not attempt to teach `all` of statistical modeling. Instead, we'll cover;

- More about the formula syntax $(Y \sim X)$, and some functions that use it to fit models

- *Some* explanation of what these functions are doing, and why it might be useful

- Some 'helper' functions, used when fitting models

Example: the t-test

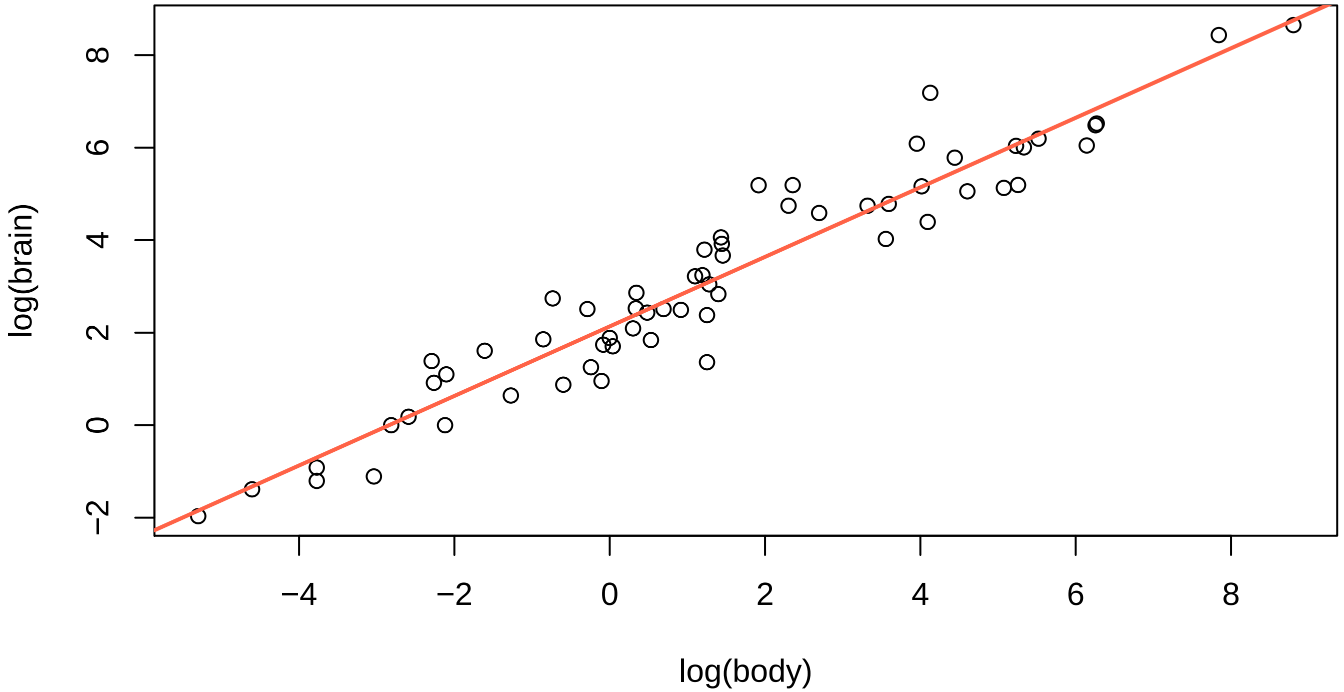
Recall the `sleep` example from Session 5. We want to compare mean levels of extra sleep, in Group 1 and 2. The full version of the code and output;

```
> t.test(extra˜group, data=sleep)
    Welch Two Sample t-test
data:  extra by group
t = -1.8608, df = 17.776, p-value = 0.07939
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -3.3654832  0.2054832
sample estimates:
mean in group 1 mean in group 2
           0.75            2.33
```

- `extra` is the outcome, it depends on `group` – for an analogous graphical comparison use `plot(extra∼group, data=sleep)`
- Confidence interval is for difference in means
- p-value: null hypothesis is of equal means (2-sided test)

Example: linear regression

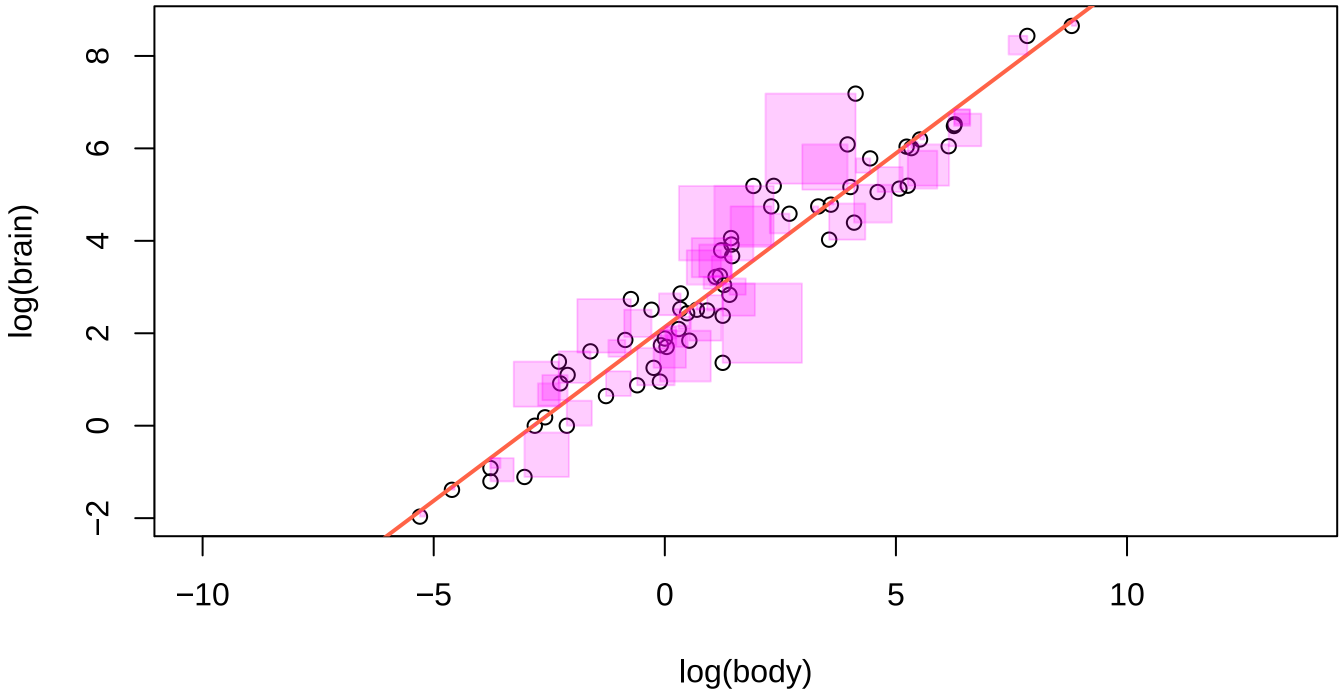
Another favorite example;



Straight line suggests `log(brain)` higher by ≈0.75 units, per 1-unit difference in `log(body)` – i.e. a power law, brain $\propto$ body$^{0.75}$.

Example: linear regression

Where does the straight line come from? One way* to justify it is as the *least squares* fit;



Any other choice of line would use more purple ink. (NB see `asp` in `?par` for aspect ratios)

* there are several – too many to discuss here!

Example: linear regression

Finding the least-squares fit is known as ‘simple’ *linear regression*, or fitting a *linear model*. In R;

```
> mammals.reg <- lm(log(brain)~log(body), data=mammals)
> summary(mammals.reg)

Call:
lm(formula = log(brain) ~ log(body), data = mammals)

Residuals:
     Min       1Q   Median       3Q      Max
-1.71550 -0.49228 -0.06162  0.43597  1.94829

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  2.13479    0.09604   22.23   <2e-16 ***
log(body)    0.75169    0.02846   26.41   <2e-16 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 0.6943 on 60 degrees of freedom
Multiple R-squared:  0.9208,Adjusted R-squared:  0.9195
F-statistic: 697.4 on 1 and 60 DF,  p-value: < 2.2e-16
```

Example: linear regression

The `summary()` is fairly verbose; (SAS is a lot worse!)

- Function `lm()` make an `lm.object`, containing the output of the regression; try `str(mammals.reg)` to see that `summary()` picks out the most important bits
- `Call` restates the formula, `Residuals` summarizes how small our 'least' square edges are
- `Coefficient`; the fitted line is

  $$\log(\texttt{brain}) = 2.13 + 0.75 \times \log(\texttt{body})$$

  The intercept (2.13) is (sensibly) added by default.
- `Std. Error` describes the noise in each estimate – smaller when you have more data
- `Pr(>|t|)` is a two sided p-value, for the null hypothesis that the relevant coefficient is zero
- Other terms describe remaining 'noise'

Example: linear regression

The next-most useful summary;

```
> confint(mammals.reg, parm="log(body)", level=0.95)
              2.5 %    97.5 %
log(body) 0.6947503 0.8086215
> confint(mammals.reg, parm=1:2, level=0.975)
               1.25 %  98.75 %
(Intercept) 1.9139805 2.355597
log(body)   0.6862469 0.817125
```

- For `lm.objects`, `confint()` gives intervals based on point estimate ± `Std. Error` × the appropriate quantile of the appropriate t distribution
- `confint.default()` uses Normal quantiles instead
- `level` is the confidence level, default is 95%
- `parm` can be a vector of coefficient names, or a vector of numbers; the default gives intervals for all terms
- Like most software, R gives an insane number of decimal places – but don't `round()` until final write-up

Example: linear regression

To ‘extract’ other parts of an `lm.object`, you can use the `$` (apostrophe-S) symbol, e.g. `mammals.reg$coef` is the point estimates. But R’s regression functions also have generic *extractor* functions, helpful for common jobs;

- `coef(mammals.reg)` – gives the fitted coefficients
- `fitted(mammals.reg)` returns the fitted `log(brain)` values (i.e. Y), for each data point (i.e. each X)
- `residuals(mammals.reg)` returns `log(body)` minus the fitted value – that we minimized the sum of, when squared
- `predict(mammals.reg, new.data.frame)` predicts the mean `log(brain)` (i.e. Y) for which you supply `log(body)`

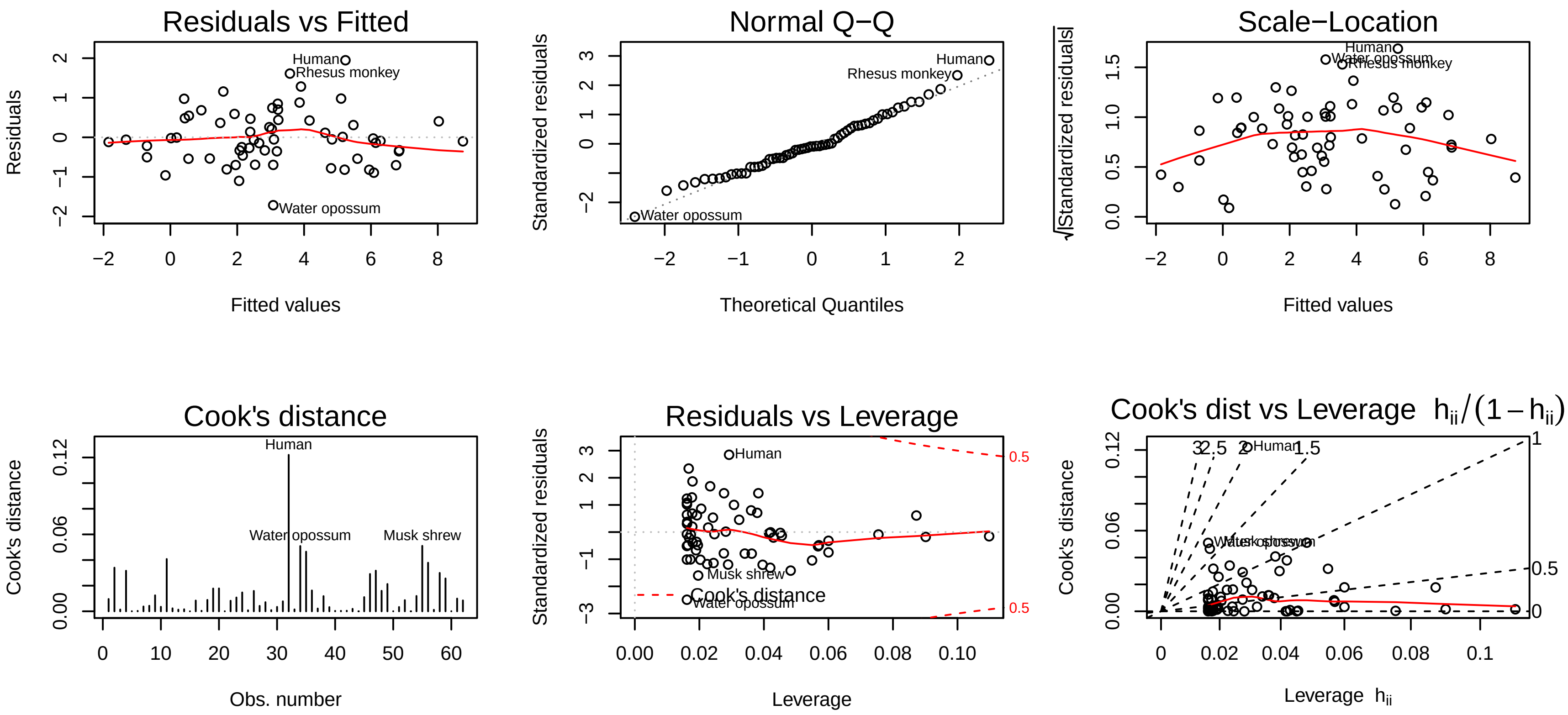
Experts: `vcov()` gives the variance-covariance matrix, describing the statistical noise in the coefficients; `sqrt(diag(vcov(mammals.reg)))` is the same as `Std. Error` column in `summary()` output.

For more of these (some fairly esoteric) use `methods(class="lm")`.

Example: linear regression

Experts again: `plot()` has a method for `lm.objects`;

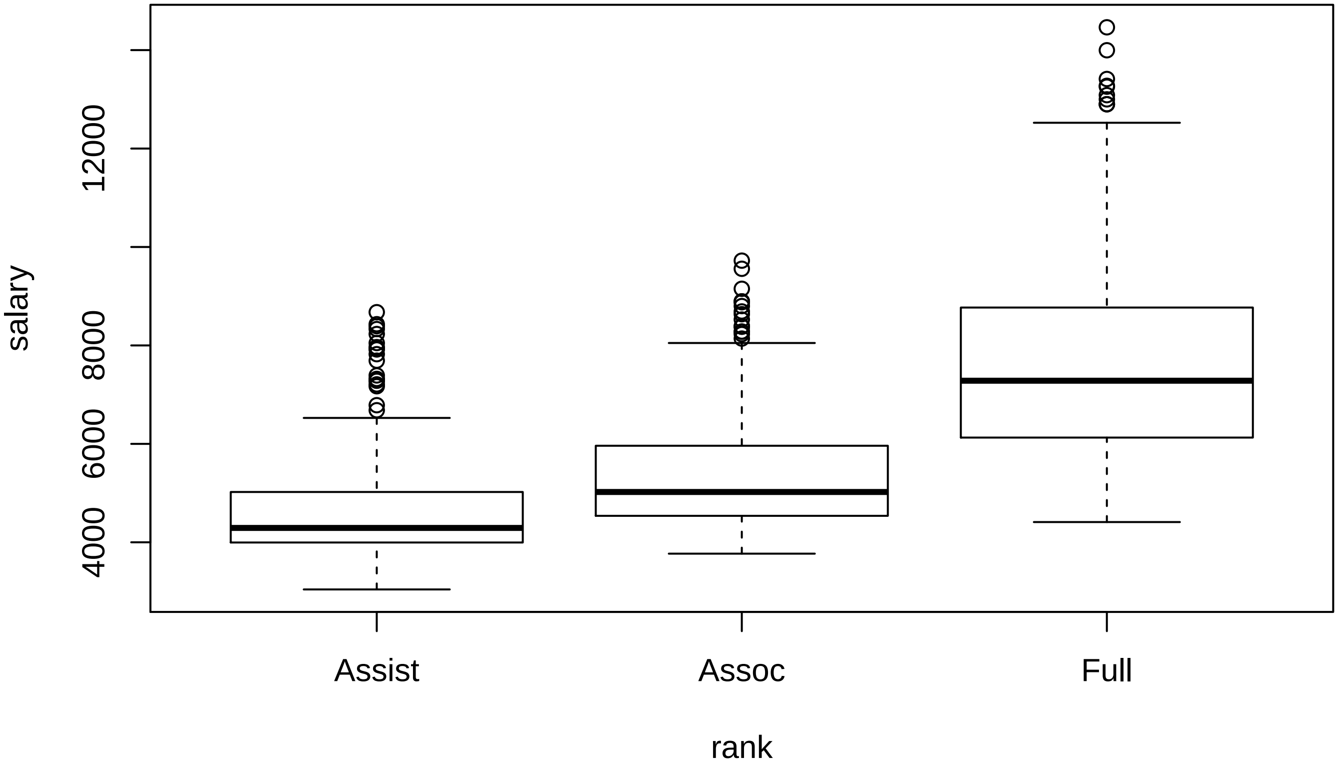
```
par(mfrow=c(2,3))
plot(mammals.lm, which=1:6)
```



Example: salaries again

Another familiar example; how does salary depend on rank?

```
plot(salary~rank, data=finalsalary)
```



As a regression, we could ask whether the mean salary is different at different ranks. NB With one final salary per person, it's reasonable to assumed independent observations.

Example: salaries again

Comparison of means is exactly what analysis of variance does …despite the name!

```
> salary.aov <- aov(salary~rank, data=finalsalary)
> summary( salary.aov )
              Df    Sum Sq   Mean Sq F value Pr(>F)
rank           2 2.642e+09 1.321e+09   529.6 <2e-16 ***
Residuals   1593 3.974e+09 2.495e+06
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
1 observation deleted due to missingness
> model.tables(salary.aov, type="means")
Tables of means
Grand mean
         6391.161
 rank
    Assist Assoc Full
      4650  5335 7584
rep    314   437  845
> table(finalsalary$rank)
Assist  Assoc   Full
   315    437    845      # spot the difference
```

Example: salaries again

‘Under the hood’, `aov()` runs group-specific linear regressions with just an intercept (`salary`$\sim$`1`, in the formula syntax) and recombines them. Here, least-squares $\equiv$ take each group’s mean.

A simpler approach* uses regression directly; (edited output)

```
> salary.lm <- lm(salary~rank, data=finalsalary)
> summary(salary.lm)
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  4650.43      89.13  52.173  < 2e-16 ***
rankAssoc     684.31     116.85   5.856 5.74e-09 ***
rankFull     2933.93     104.39  28.106  < 2e-16 ***
  (1 observation deleted due to missingness)
F-statistic: 529.6 on 2 and 1593 DF,  p-value: < 2.2e-16
```

- Same F statistic, and p-value, *equivalent* point estimates
- `Intercept` describes mean salary in Assist Profs (again)
- Other coefficients describe *differences* – so e.g. $p = 5.74 \times 10^{-9}$ is for testing Assist=Assoc. `Assist` is ‘reference’ level

* preferred by most statisticians, though not all

Multiple regression

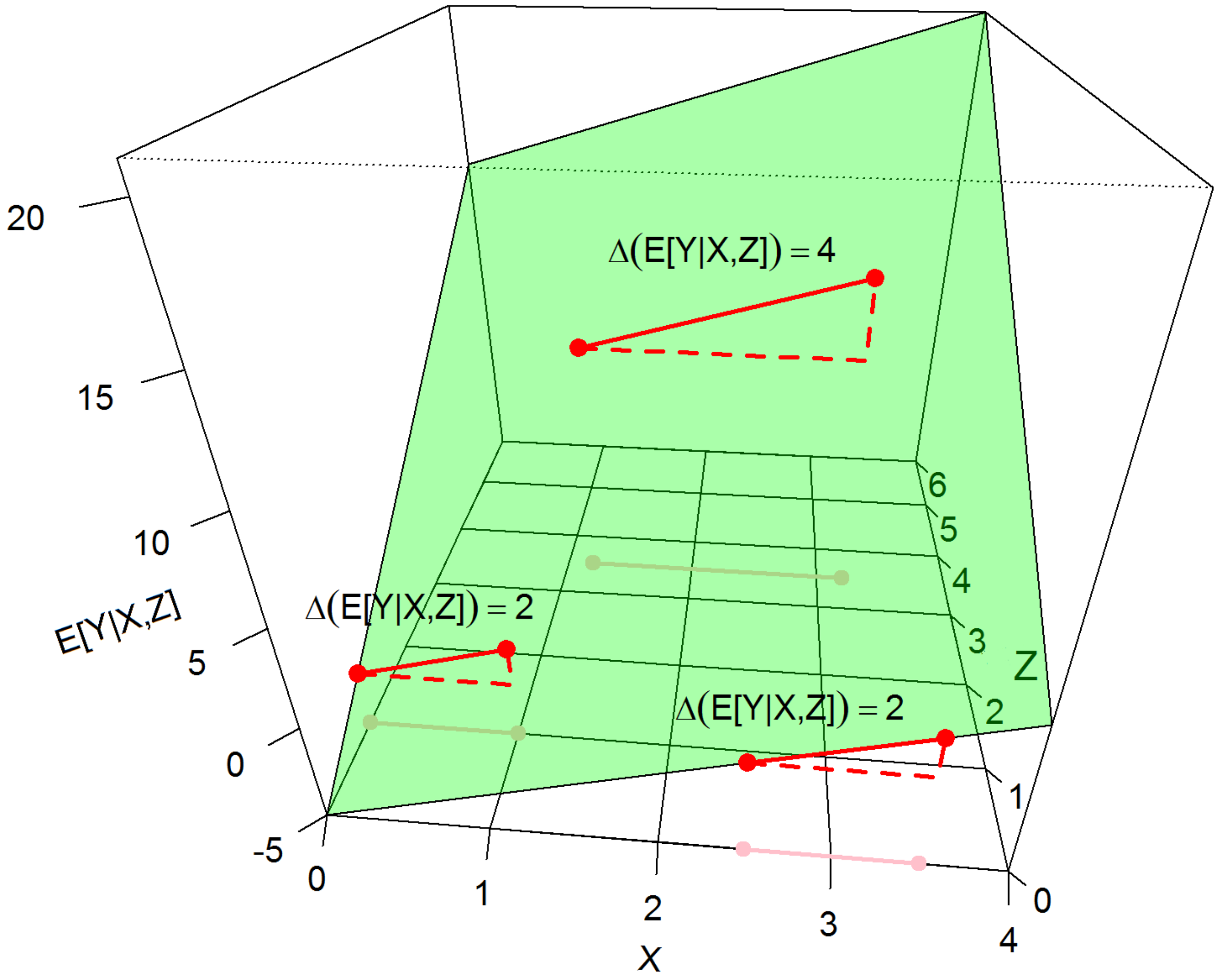
Say you wanted to know how salary depended on start year at UW, and on year of final degree (≈ age, here)

```
> mreg <- lm(salary~ yrdeg + startyr, data=finalsalary)
> summary(mreg)
Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 13583.596    375.936  36.133  < 2e-16 ***
yrdeg        -118.455      7.380 -16.051  < 2e-16 ***
startyr        22.438      7.275   3.084  0.00208 **
```

- Starting later associated with greater salary (+22.44) in people with same year of degree
- Getting a degree earlier associated with less salary (-118.46) in those who started in same year
- In the formula, '+' means 'and'. To regress on multiple covariates, use `y ~ x + z + u + v + ...`
- To use 'plus' in a formula (or minus) `I()` is for insulate; `y ~ x + I(z + u)` regresses Y on X and the sum of Z & U

Multiple regression

Regressing Y on X and Z fits a plane;



Multiple regression

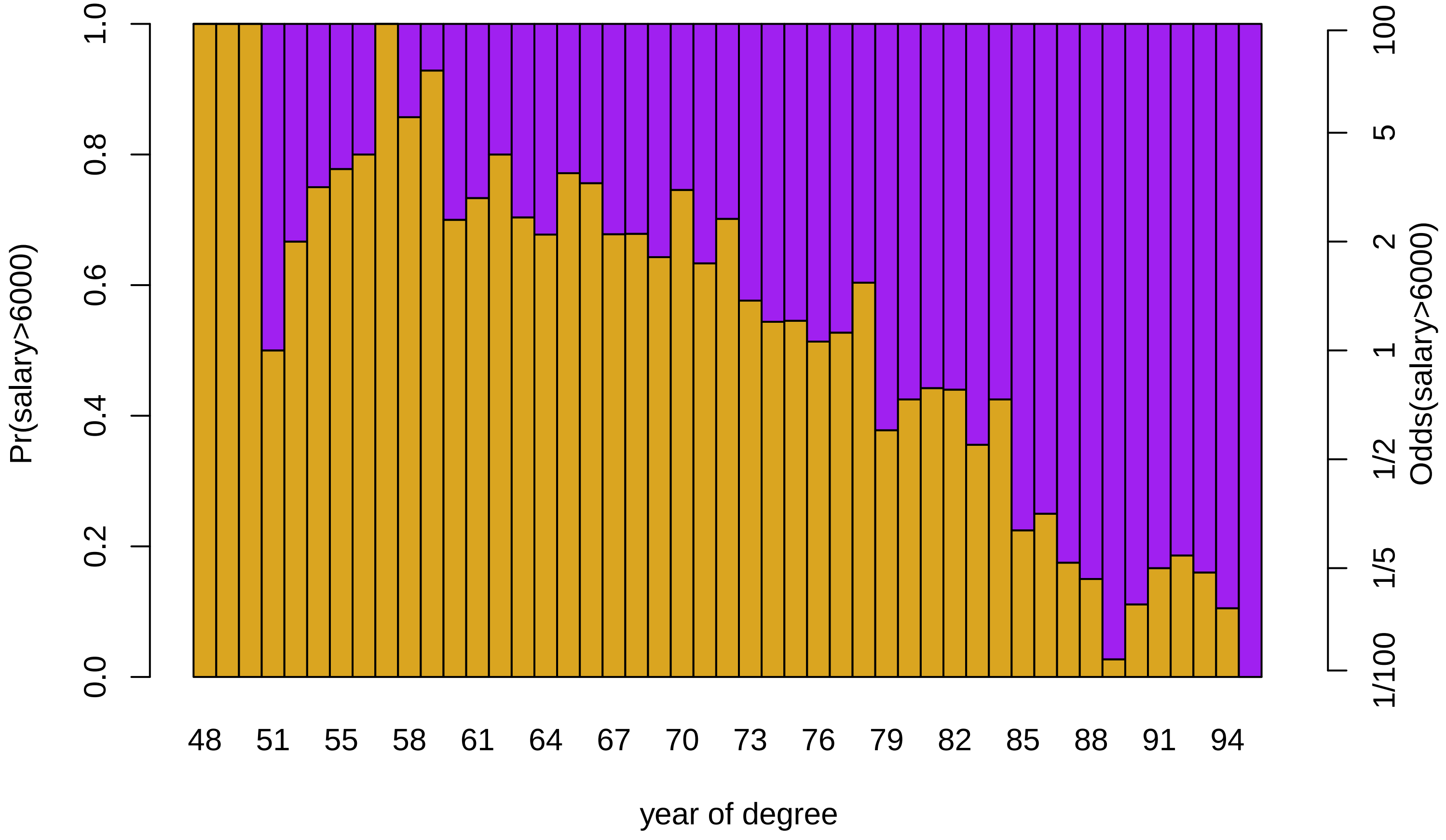
To test hypotheses involving more than one parameter at a time, use `anova()` to compare the fitted models with and without those parameters;

```
> mreg  <- lm(salary~ yrdeg + startyr, data=finalsalary)
> mreg0 <- lm(salary~ yrdeg + startyr + rank, data=finalsalary)
> anova(mreg, mreg0)
Analysis of Variance Table

Model 1: salary ~ yrdeg + startyr
Model 2: salary ~ yrdeg + startyr + rank
  Res.Df        RSS Df  Sum of Sq      F    Pr(>F)
1   1593 5025366914
2   1591 3834775234  2 1190591680 246.98 < 2.2e-16 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
```

- Here testing any difference between ranks, adjusted for the other two variables – order doesn't matter
- Not the same as `aov()`!
- With only one model, `anova()` tests each coefficient, in order of appearance – order *does* matter

Logistic regression

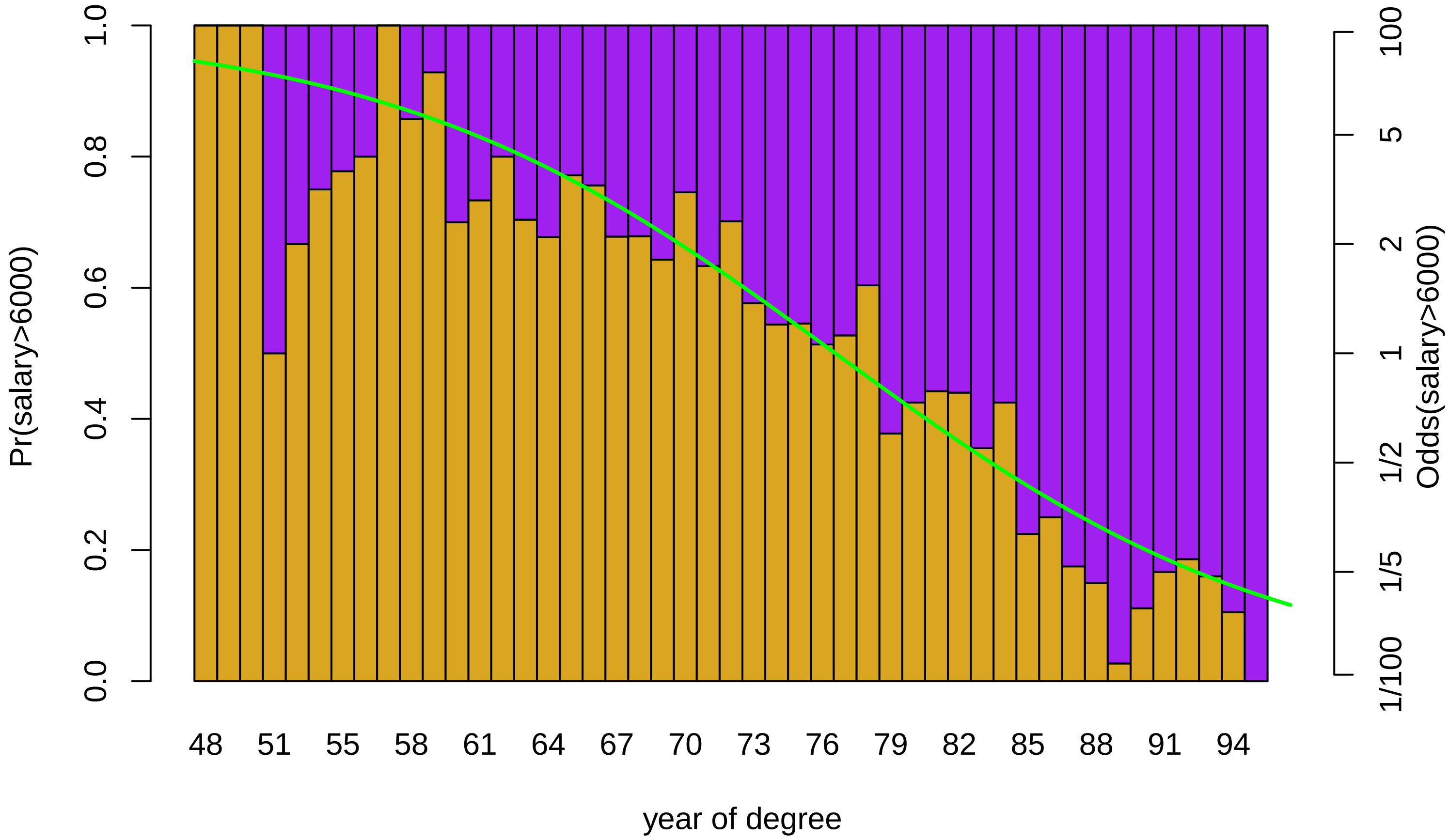
When Y is binary (e.g. 1/0, yes/no, dead/alive) the expected value of Y is the probability that $Y = 1$.



Linear regression's straight line might give a poor summary.

Logistic regression

Instead of a straight line, *logistic regression* fits a curve through the data;



The fitted odds shrink by ≈10%, for each extra year.

Logistic regression

The `glm()` command does this (close relative of `lm()`)

```
> glm1 <- glm(salary>6000 ~ yrdeg, data=finalsalary, family=binomial)
> summary(glm1)
Call:
glm(formula = salary > 6000 ~ yrdeg, family = binomial, data = finalsalary)
Deviance Residuals:
    Min       1Q   Median       3Q      Max
-2.2923  -0.9342  -0.5215   0.9674   1.9871
Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept)  7.743524   0.490335   15.79   <2e-16 ***
yrdeg       -0.101791   0.006403  -15.90   <2e-16 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
(Dispersion parameter for binomial family taken to be 1)
    Null deviance: 2212.5  on 1595  degrees of freedom
Residual deviance: 1895.9  on 1594  degrees of freedom
  (1 observation deleted due to missingness)
AIC: 1899.9
Number of Fisher Scoring iterations: 4
```

NB turn those #&*%ing stars off! `options(show.signif.stars=FALSE)`

Logistic regression

The coefficients here are *log* odds (for the Intercept) and *log* odds ratios. So, for a confidence interval around the '10% smaller' result;

```
> confint(glm1, "yrdeg", level=0.95)
Waiting for profiling to be done...
      2.5 %      97.5 %
-0.11454628 -0.08943648
> confint.default(glm1, "yrdeg", level=0.95)
           2.5 %     97.5 %
yrdeg -0.1143396 -0.0892416
> round(exp(confint.default(glm1, "yrdeg", level=0.95)), 3)
      2.5 % 97.5 %
yrdeg 0.892  0.915
```

- The default is fairly sophisticated; for typical symmetric intervals use `confint.default()`
- ... then exponentiate to get interval for the odds ratio

All the extractor functions we saw before are available – and use the formula syntax to regress on multiple covariates.

Other regressions, other tests

In `glm()`, other `family` arguments provide other forms of regressions – too many for our course. Some other tests;

```
> (tab1 <- with(droplevels(subset(finalsalary, yrdeg>87 & rank!="Full")),
+ table( salary>6000, rank) ))
       rank
        Assist Assoc
  FALSE    199    24
  TRUE      25     8
> chisq.test( tab1 )
Pearson's Chi-squared test with Yates' continuity correction
X-squared = 3.6229, df = 1, p-value = 0.05699
Warning message:
In chisq.test(tab1) : Chi-squared approximation may be incorrect
> fisher.test(tab1)
Fisher's Exact Test for Count Data
p-value = 0.04413
alternative hypothesis: true odds ratio is not equal to 1
95 percent confidence interval:
 0.9243447 6.9357450
sample estimates:
odds ratio
  2.640338
```

Summary

- There are R implementation of almost every regression method

- Most use the formula syntax, also used for plotting – naturally, because both describe how outcome Y depends on some covariates

- The default in `lm()` and `glm()` is to drop cases with NAs – without warning you

- Extractor functions save time, and make code easier to read

There are many more regression methods available, beyond what 'plain vanilla' R provides – in the next session we'll discuss use of 'packages', to extend R.

8. Introduction to R Packages

Ken Rice
Timothy Thornotn

University of Washington

Seattle, July 2013

In this session

Base R includes pre-installed packages that allow for a fully functioning statistical environment in which a variety of analyses can be conducted. Thousands of user-contributed extension packages are available that provide enhanced functionality with R.

- Finding and installing packages
- Loading packages
- Examples using available packages on CRAN

CRAN Packages

The programming environment of R has facilitated rapid development of packages by numerous authors.

R packages are collections of functions, data, and compiled code in a well-defined format.

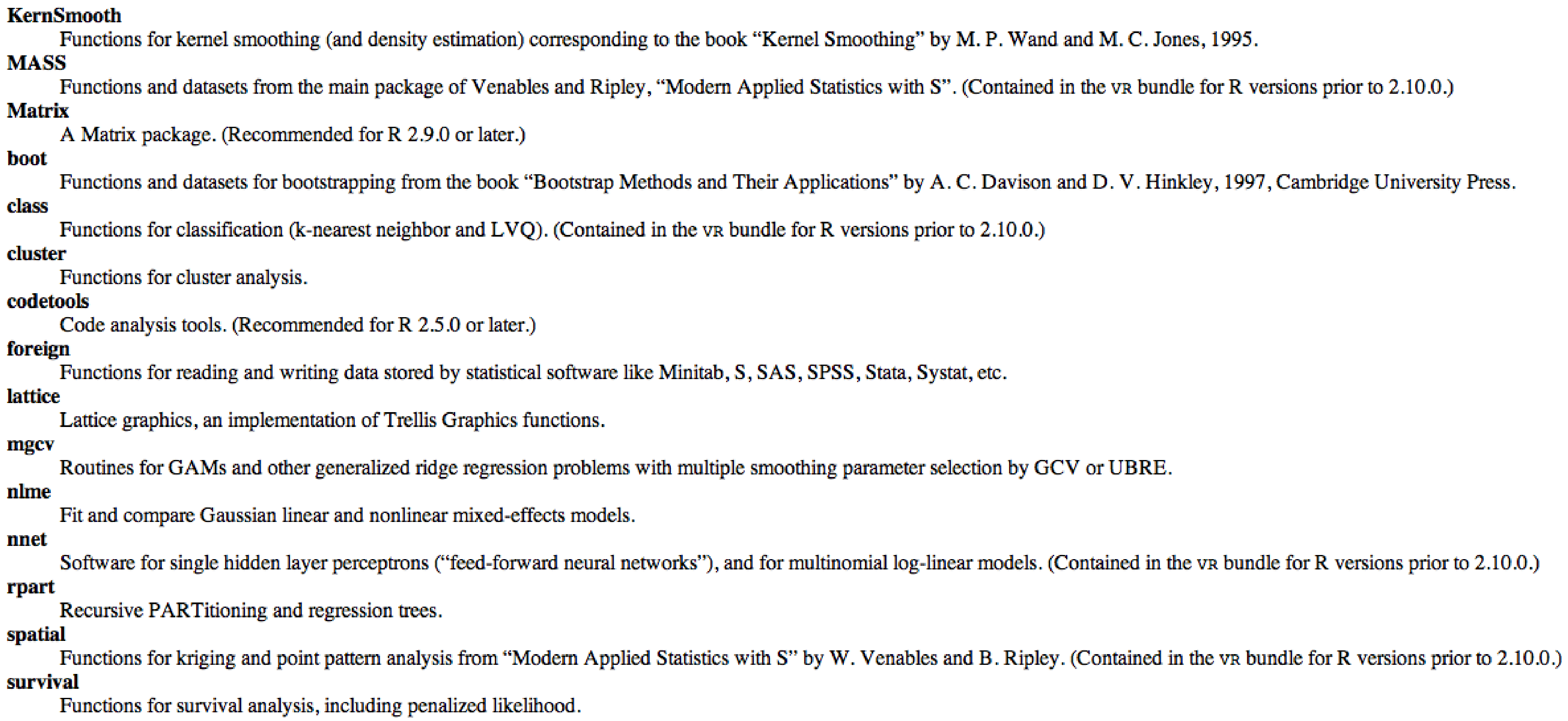
Thousands of packages are available for download from the Comprehensive R Archive Network (CRAN):
`http://cran.r-project.org`

There are more than 4,500 packages available on CRAN!

Standard or *base* R comes with some standard packages installed for basic data management, analysis, and graphical tools.

CRAN Packages

Below are some of the CRAN packages that come with base R:



KernSmooth
Functions for kernel smoothing (and density estimation) corresponding to the book "Kernel Smoothing" by M. P. Wand and M. C. Jones, 1995.

MASS
Functions and datasets from the main package of Venables and Ripley, "Modern Applied Statistics with S". (Contained in the VR bundle for R versions prior to 2.10.0.)

Matrix
A Matrix package. (Recommended for R 2.9.0 or later.)

boot
Functions and datasets for bootstrapping from the book "Bootstrap Methods and Their Applications" by A. C. Davison and D. V. Hinkley, 1997, Cambridge University Press.

class
Functions for classification (k-nearest neighbor and LVQ). (Contained in the VR bundle for R versions prior to 2.10.0.)

cluster
Functions for cluster analysis.

codetools
Code analysis tools. (Recommended for R 2.5.0 or later.)

foreign
Functions for reading and writing data stored by statistical software like Minitab, S, SAS, SPSS, Stata, Systat, etc.

lattice
Lattice graphics, an implementation of Trellis Graphics functions.

mgcv
Routines for GAMs and other generalized ridge regression problems with multiple smoothing parameter selection by GCV or UBRE.

nlme
Fit and compare Gaussian linear and nonlinear mixed-effects models.

nnet
Software for single hidden layer perceptrons ("feed-forward neural networks"), and for multinomial log-linear models. (Contained in the VR bundle for R versions prior to 2.10.0.)

rpart
Recursive PARTitioning and regression trees.

spatial
Functions for kriging and point pattern analysis from "Modern Applied Statistics with S" by W. Venables and B. Ripley. (Contained in the VR bundle for R versions prior to 2.10.0.)

survival
Functions for survival analysis, including penalized likelihood.

Installing packages

We often need additional functionality beyond the packages offered by the base core R library.

In order to install an extension package, we can invoke the `install.packages()` function which typically downloads the package from CRAN and installs it for use.

For example, the following command can be used to install the hexbin package:

```
install.packages("hexbin")
```

Installing packages

Alternatively, an R package can be installed by

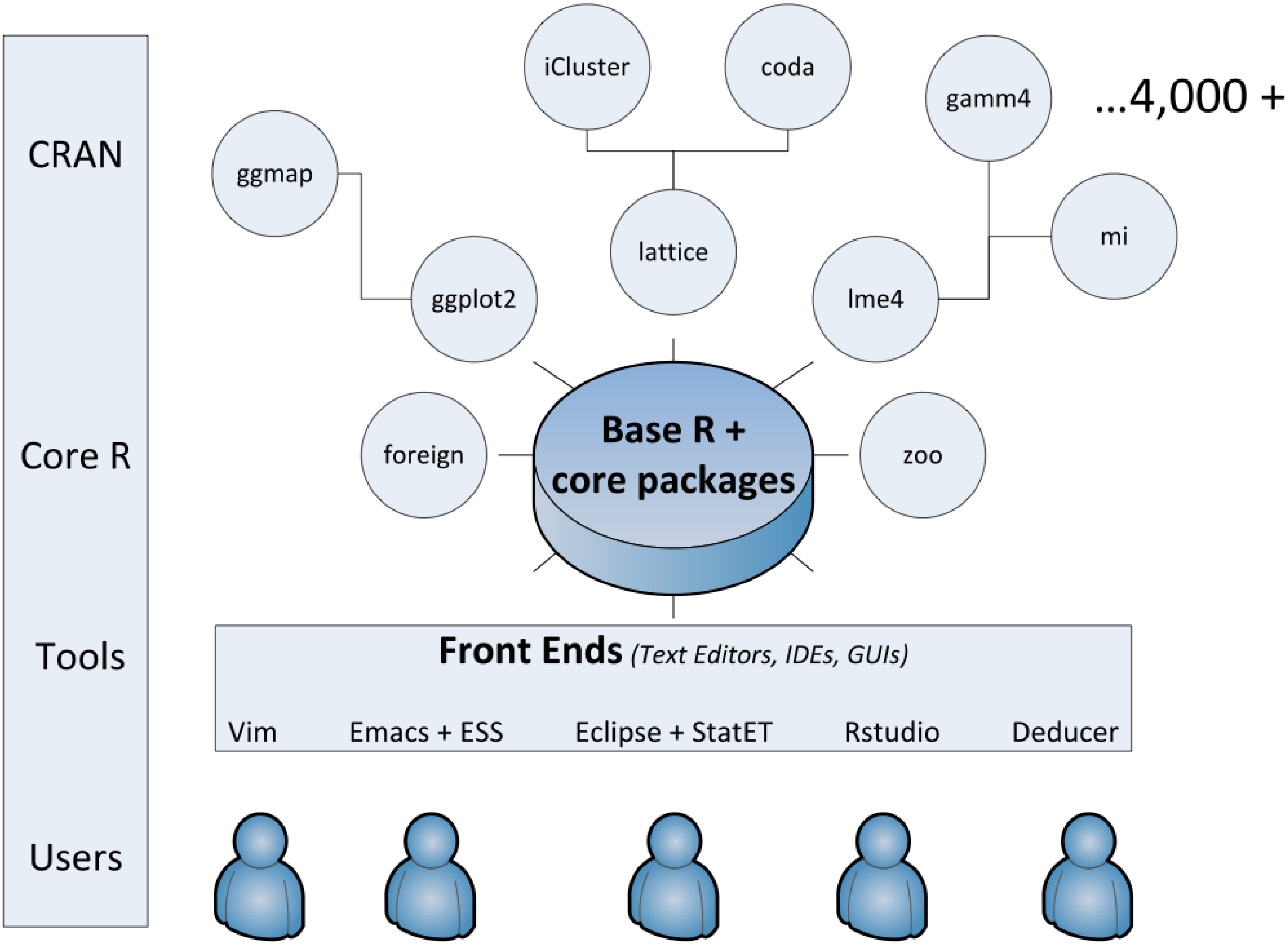
- using the command `install.packages()` without any arguments

- selecting “Install package(s)” from the “Packages” menu (for Windows versions) or “Packages and Data” menu (for Apple versions) near the top of the R console.

This will bring up a list of packages that are available for installation.

You can then choose the package that you would like to install from that list.

Installing packages



Loading Packages

To use an installed package for an R session, the package must first be loaded.

The `library` function can be used to load a package. For example, to load the foreign package for an R session, use the following command:

```
library("foreign")
```

To see which packages are installed for your session, issue the command `library()` without any arguments.

Users connected to the Internet can use the `update.packages()` function to update packages.

Vignettes

Many packages include ”vignettes”. A package vignette gives an overview of the package and sometimes includes examples.

Package vignettes are not a required component of an `R` package, so some packages will not have them.

For those packages which contain vignettes, you can find them by the `browseVignettes()` function. For example, the vignette for the hexbin package can be found using the following command:

```
browseVignettes("hexbin")
```

The `vignette()` function can also be used for finding package vignettes.

Info about your R Session

To get a description of the version of R and packages that have been loaded for the current session, the sessionInfo function can be used:

```
> sessionInfo()
R version 2.13.0 (2011-04-13)
Platform: x86_64-apple-darwin9.8.0/x86_64 (64-bit)

locale:
[1] en_US.UTF-8/en_US.UTF-8/C/C/en_US.UTF-8/en_US.UTF-8

attached base packages:
[1] grid      stats     graphics  grDevices utils
datasets  methods   base
```

Info about your R Session

```
other attached packages:
[1] hexbin_1.26.0   lattice_0.19-23 foreign_0.8-43

loaded via a namespace (and not attached):
[1] tools_2.13.0
```

The command `search()`, without any arguments, can also be used to for information on which packages are currently loaded.

Example: the foreign package

The foreign package comes with base R and is quite useful for importing and exporting data.

To use the functions that are available in this package, we must first load foreign:

```
library("foreign")
```

Datasets from other statistical analysis software using functions in the foreign package.

The function `read.spss` can be used to read in an SPSS data file:

```
dat.spss <- read.spss("http://faculty.washington.edu/tathornt/sisg2013/hsb2.sav",
to.data.frame=TRUE)
```

The function `read.dta` can be used to read in a Stata data file:

```
dat.dta <- read.dta("http://faculty.washington.edu/tathornt/sisg2013/hsb2.dta")
```

Example: the survey package

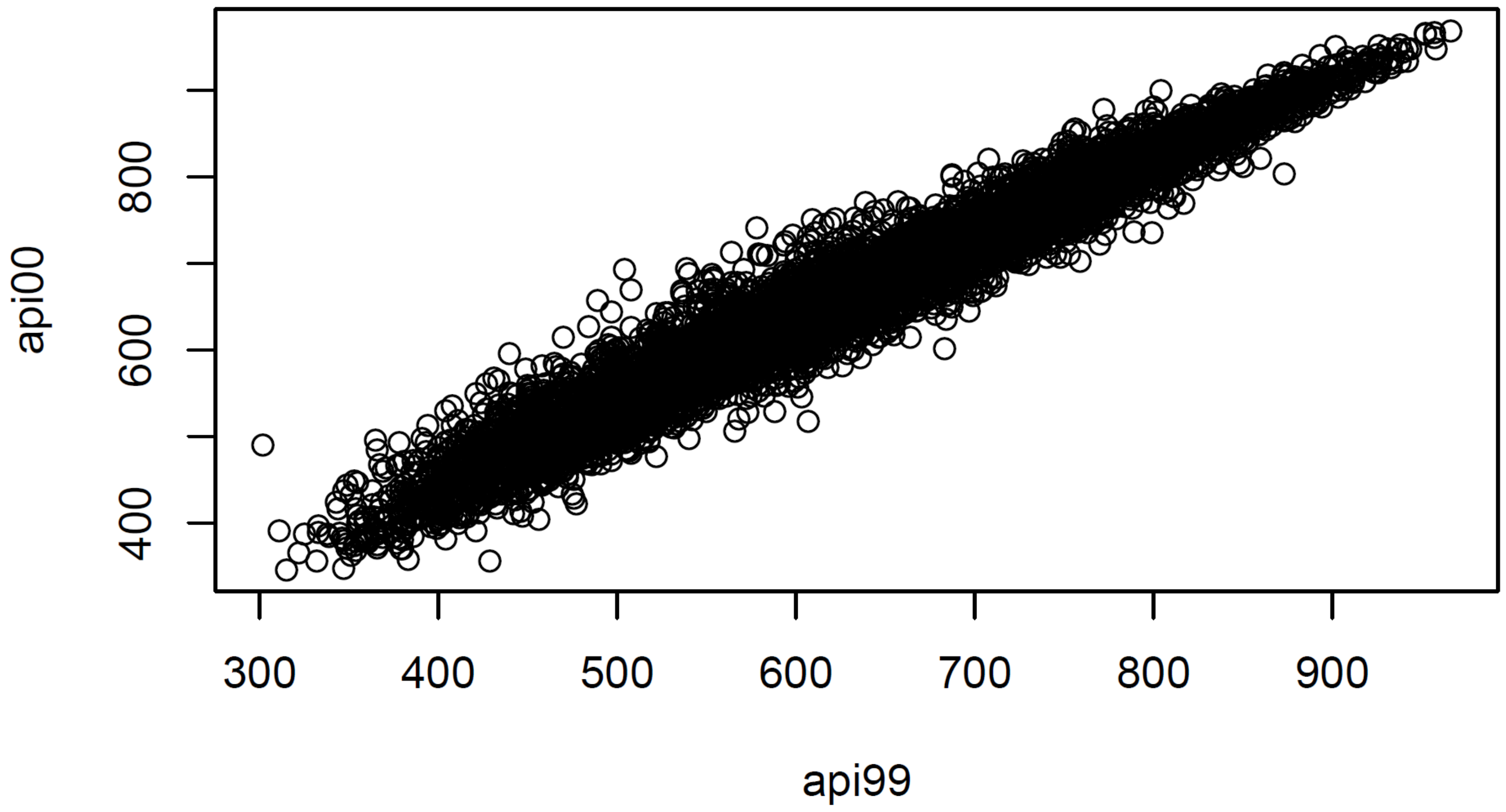
The survey package includes a data set named ”api” containing California Academic Performance Index is reported on 6194 schools

```
library("survey")
data(api, package="survey")
```

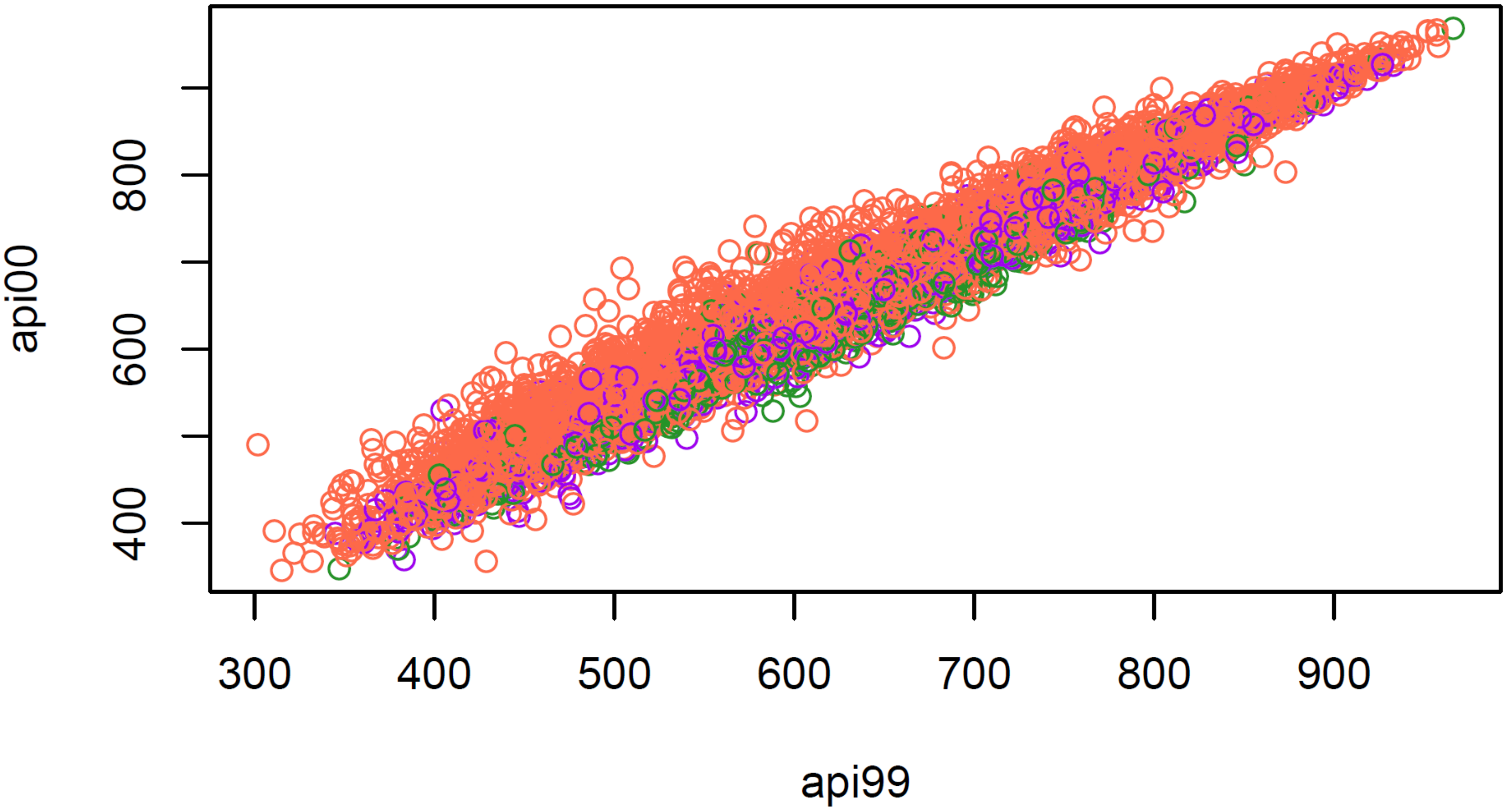
Below are scatterplot commands for this data. The scatterplots illustrate how crowded these plots can be with large data sets.

```
plot(api00~api99,data=apipop)
colors<-c("tomato","forestgreen","purple")[apipop$stype]
plot(api00~api99,data=apipop,col=colors)
```

Example: the survey package



Example: the survey package



Example: the hexbin package

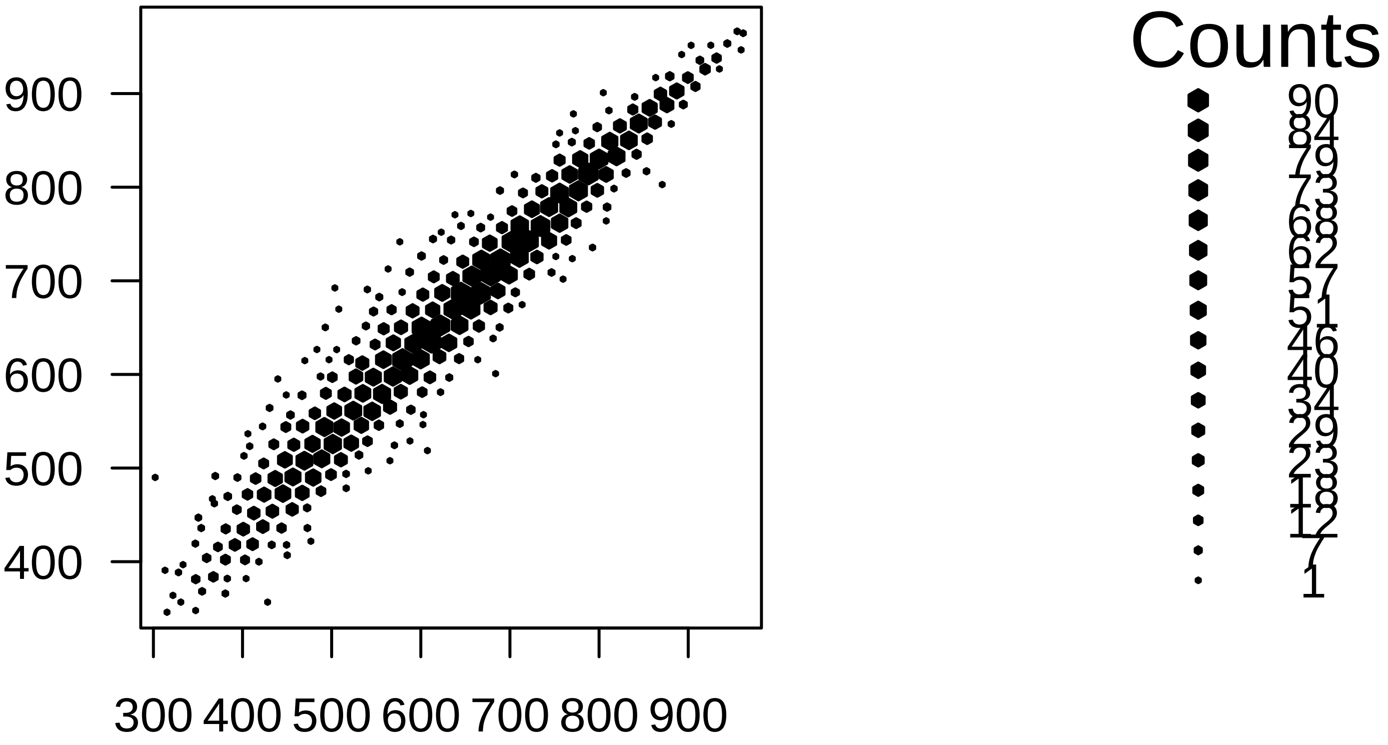
When there are many data points and significant overlap, scatterplots become less useful.

The `hexbin()` function in the hexbin package provides a way to aggregate the points in a scatterplot. It computes the number of points in each hexagonal bin.

```
library("hexbin")
with(apipop, plot(hexbin(api99,api00), style="centroids"))
```

The `style="centroids"` option plots filled hexagons, at the centroid of each bin. The sizes of the plotted hexagons are proportional to the number of points in each bin.

Example: the hexbin package



Summary

- Many functions in R live in optional packages.
- Thousands of packages are available on CRAN for downloading.
- The `install.packages()` function is used for installing an extension package
- The `library()` function lists packages, shows help, or loads packages from the package library.
- The foreign package is in the standard distribution. It handles import and export of data.

9. Writing Functions

Ken Rice
Tim Thornton

University of Washington

Seattle, July 2013

In this session

One of the most powerful features of R is the user's ability to expand existing functions and write custom functions. We will give an introduction to writing functions in R.

- Structure of a function
- Creating your own function
- Examples and applications of functions

Introduction

Functions are an important part of R because they allow the user to customize and extend the language.

- Functions allow for reproducible code without copious/error prone retyping

- Organizing code into functions for performing specified tasks makes complex programs tractable

- Often necessary to develop your own algorithms or take existing functions and modify them to meet your needs

Structure of a function

Functions are created using the function() directive and are stored as R objects.

Functions are defined by

1. a function name with assignment to the function() directive (function names can be almost anything. However, the usage of names of existing functions should be avoided.)

2. the declaration of arguments/variables

3. and the definition of operations (the function body) that perform computations on the provided arguments.

Structure of a function

The basic structure of a function is:

```
my.func <- function(arg1,arg2,arg3, ...) {
 commands;
 return(output);
}
```

Function arguments (arg1, arg2, ...) are what is passed to the function and used by the function's code to perform calculations.

Calling a function

Functions are called by their name followed by parentheses containing possible argument names.

A call to the function generally takes the form

```
my.func(arg1=expr1,arg2=expr2,arg3=exp3, ...)
```

or

```
my.func(expr1,expr2,exp3, ...)
```

Arguments can be "matched" by name or by position.

A function can also take no arguments, and in this case the function is called with the name of the function with empty parenthesis () after the function name.

Typing just the function name without parentheses will print the definition of a function.

Function body

The actual expressions (commands/operations) are defined in the body of the function.

The function body appears within {curly brackets}. The brackets {} are not required for functions with just one expression.

Individual commands/operations are separated by new lines or semicolons.

An object is returned by a function with the `return()` command, where the object to be returned appears inside the parentheses.

If the end of a function is reached without calling `return`, the value of the last evaluated expression will be returned by the function.

Variables that are created inside the function body exist only for the lifetime of the function. Thus, they are not accessible outside of the function in an R session.

Example: returning a single value

Below is a function for calculating the coefficient of variation (the ratio of the standard deviation to the mean) for a vector.

```
coef.of.var <- function(x){
   meanval <- mean(x,na.rm=TRUE)
   sdval   <- sd(x,na.rm=TRUE)
   return(sdval/meanval)
   }
```

We can apply this function to obtain the coefficient of variation for the daily ozone concentrations in New York, summer 1973:

```
data(airquality)
coef.of.var(airquality$Ozone)
```

```
> coef.of.var(airquality$Ozone)
[1] 0.7830151
```

Example: returning multiple values

A function can return multiple objects/values by using `list()` – which collects objects of (potentially) different types.

The function below calculates and returns the maximum likelihood estimates for the mean and standard deviation for a numeric vector under a normal distribution assumption

```
gaussian.mle <- function(x) {
n <- length(x)
mean.est <- mean(x,na.rm=TRUE)
var.est <- var(x,na.rm=TRUE)*(n-1)/n
est <- list(mean=mean.est, sd=sqrt(var.est))
return(est)}
```

Example: returning multiple values

We can apply the gaussian.mle function to the daily ozone concentrations in New York data:

```
> results<-gaussian.mle(airquality$Ozone)
> attributes(results)  #list the attributes of the object returned
$names
[1] "mean" "sd"
> results$mean
[1] 42.12931
> results$sd
[1] 32.8799
```

Elements of lists can also be obtained using *double* square brackets, i.e. `results[[1]]`.

Declaring functions within functions

Functions can be declared and used inside a function.

```
square.plus.cube <- function(y) {
   square <- function(x) { return(x*x) }
   cube <- function(x) { return(x^3) }
   return(square(y) + cube(y))
 }
```

```
> square.plus.cube(4)
[1] 80
```

Example: function returning a function

A function can also return another function as the return object.

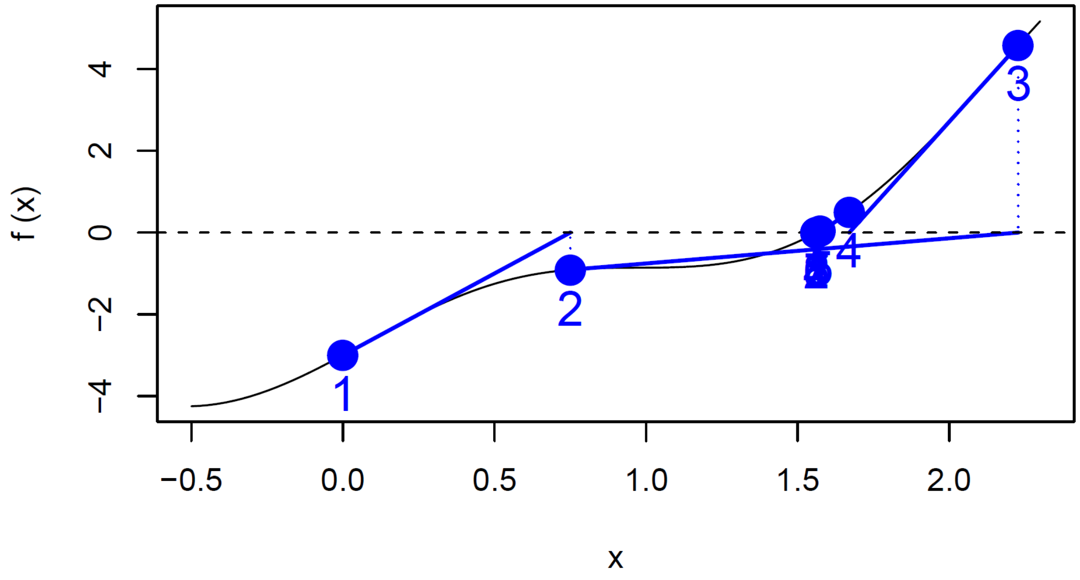
```
make.power <- function(n){
   pow <- function(x) {x^n}
   pow
   }

cube <- make.power(3)
square <- make.power(2)

> cube(3)
[1] 27
> square(3)
[1] 9
```

Example: functions as arguments

Functions can take other functions as arguments. This is helpful with finding *roots* of a function; values of x such that $f(x) = 0$.

With the Newton-Raphson method, a root can be found by the following iteration procedure until convergence:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$



Example: functions as arguments

This function implements the Newton-Raphson method, given input of arguments, a place to start, and convergence tolerance:

```
newton.raphson <- function(f,fprime,x0,thresh){
   myabsdiff <- Inf
   xold      <- x0
   while(myabsdiff>thresh){ # have we converged yet? If no, move;
      xnew      <- xold-f(xold)/(fprime(xold))
      myabsdiff <- abs(xnew-xold)
      xold      <- xnew
   }
return(xnew)
}
```

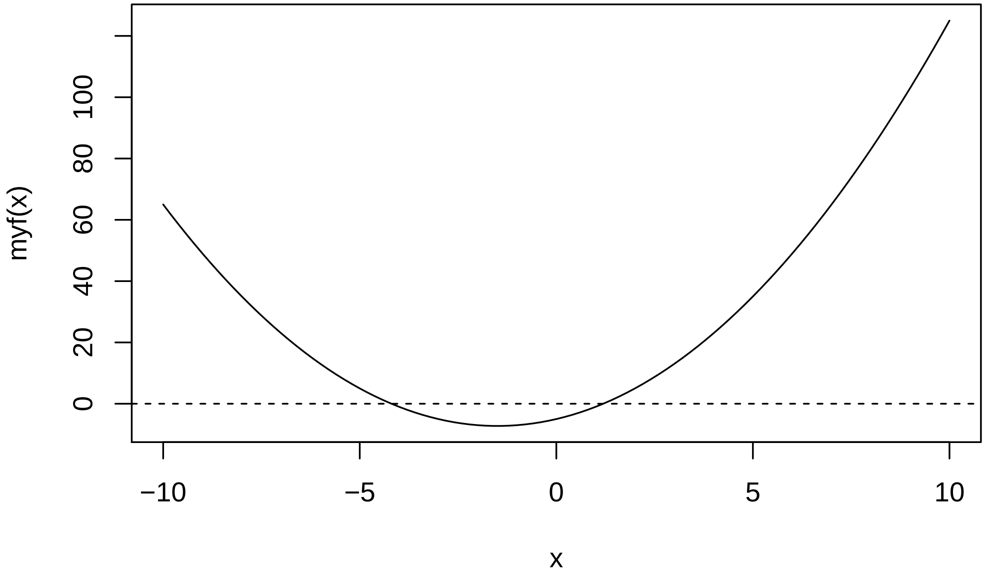
Example: functions as arguments

We'll find the roots of $f(x) = x^2+3x-5$, using Newton-Raphson. We need the derivative of $f(x)$: $f'(x) = 2x+3$

```
myf <- function(x){ x^2+3*x-5 }
myfprime<-function(x){ 2*x+3 }
```

We use the `newton.raphson()` function with initial value of 10 and a convergence threshold of 0.0001 to obtain a root:

```
> newton.raphson(f=myf,fprime=myfprime,x0=10,thresh=0.0001)
[1] 1.192582
```



Tips for writing functions

- Avoid rewriting the same code...use functions!

- Modularize as much as possible: functions calling other functions

- Provide documentation, including detailed comments describing the procedures being conducted by the functions, especially for large, complex programs

- Test your functions: use data/arguments for which you know the results to verify that your functions are working properly

- Use meaningful variable and function names

Summary

- User-defined functions can easily be created in `R` with function(argument list)

- Arguments of a function are allowed to be practically any `R` object including lists, numeric vectors, data frames, and functions

- In functions calls, arguments are matched by name or by position

- An object can be returned by a function with `return()`. If `return()` is not invoked, the last evaluated expression in the body of a function will be returned.

- `list()` can be used for returning multiple values

10. The End

Ken Rice
Tim Thornton

University of Washington

Seattle, July 2013

In this session

- Review of the HW exercise
- Some other packages, that interact with the wider world
- Questions/Answers, advertising of other modules

[Slides not printed for this session]

NB The course site remains ‘up’, if you need to review it in future.