R basics

Daniel E. Weeks

Department of Human Genetics
School of Public Health
University of Pittsburgh

Location

This slide set is in the Lectures repository in the 03_R_basics folder.

Acknowledgment

- Many of these slides were originally created by Stephen Eglen, and are used by permission.
- The original slide set was published as a supplement to Elgen (2009) with these permissions: "This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited."
- Reference: Eglen SJ. A quick guide to teaching R programming to computational biology students. PLoS Comput Biol (2009) vol. 5 (8) pp. e1000482
- http://www.ploscompbiol.org/doi/pcbi.1000482

What is R?

- Computing environment, similar to matlab.
- Very popular in many areas of statistics, computational biology.
- “Programming with data” (Chambers)
- Approach:
  - command-line for one-liners.
  - write scripts/functions for larger work (edit/run cycle).

Modified from original slide by Eglen (2009).

History

- S language came from Bell Labs (Becker, Chambers and Wilks). Commercial version S-plus (1988).
- R emerged as a combination of S and Scheme: Ross Ihaka and Robert Gentleman (NZ).
- 1993: first announcement.
- 1995: 0.60 release, now under GPL.
- Major release typically Apr/Oct with fixes between.

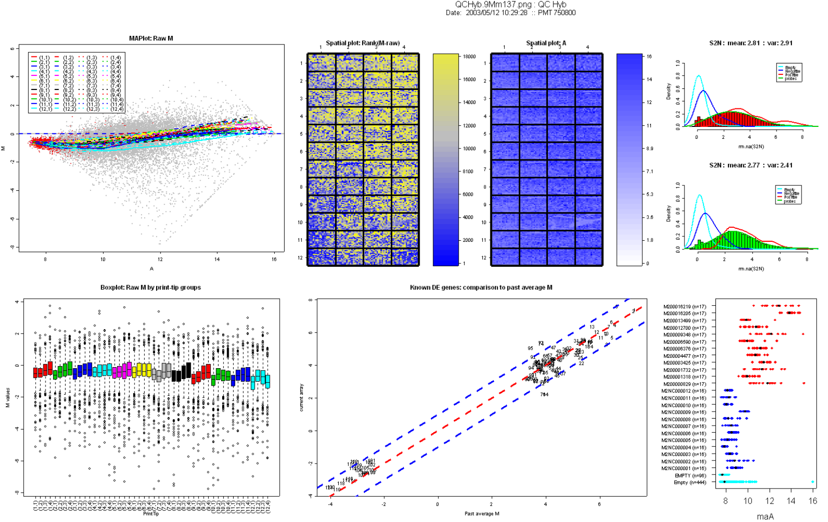
Modified from original slide by Eglen (2009).

Strengths of R

- GPL'd, available on many platforms.
- Excellent development team with Apr/Oct release cycle.
- Source always available to examine/edit.
- Fast for vectorized calculations.
- Foreign-language interface (C/Fortran) when speed crucial, or for interfacing with existing code.
- Good collection of numerical/statistical routines.
- Comprehensive R Archive Network (CRAN) - lots of R packages.
- On-line doc, with examples.
- High-quality graphics (pdf, postscript, quartz, x11, bitmaps). Often used just for plotting . . .

Modified from original slide by Eglen (2009).

Graphics example



Jean YH Yang; gpQuality
http://bioinf.wehi.edu.au/marray/ibc2004/lect1b-quality.pdf
Modified from original slide by Eglen (2009).

Weaknesses of R

- Loops are slow.
    - Learn how to vectorize solutions or use `apply` family of functions.
- No compiler yet, and unlikely to happen due to nature of language.

Modified from original slide by Eglen (2009).

Ignore your fear

- Patrick Burns found that “the biggest stumbling block in learning R was thinking that R was hard”.
- Hint number one when beginning R:
    - **Ignore your fear**.
- http://www.burns-stat.com/pages/Tutor/hints_R_begin.html

Using R

- Start-up: type ‘R’ at command line.
- Type commands interactively, and get results.
- Type commands into a file; source(‘myfile.R’); edit file . . .
- All platforms have a command-line interface
- Many external editors have support for R, including Emacs (http://ess.r-project.org).
- RStudio is now commonly used.

Modified from original slide by Eglen (2009).

Startup

- At startup, R loads a number of packages, so the commands in those packages are available to you to use.

```
options(width=60)
search()
```

```
## [1] ".GlobalEnv"         "package:stats"
## [3] "package:graphics"   "package:grDevices"
## [5] "package:utils"      "package:datasets"
## [7] "package:methods"    "Autoloads"
## [9] "package:base"
```

- To quit R, use the command ‘q()’.

Blank Screen Syndrome

- So you have successfully started R on your machine. Here's where the trouble sometimes starts – there's a big, huge prompt daring you to do something.
- You don't need a mirror to know that you have that deer-in-the-headlights look on your face.
- The solution is,
  - first, to have something to do,
  - and then to break that task into steps.

By Patrick Burns - see http://www.burns-stat.com/pages/Tutor/hints_R_begin.html

I miss my menus

- You may be wondering why you should learn a language rather than have a package that just gives you menus.
- Do you carry a picture card around with you to communicate with other people?
    - Language is much more convenient than having a small number of choices to point at. Pointing at pictures on a menu is marginally workable at restaurants in foreign countries. Much beyond that it becomes useless.
- The computing world is not much different.
- While learning a language requires expending extra effort at first, ultimately it will most likely save a lot of effort.

By Patrick Burns - see
http://www.burns-stat.com/pages/Tutor/more_R_blankscreen.html

Write down the steps

- If you are not sure how to proceed with a task, write down the steps you need to do in order to achieve the task.
- You may have to break some steps into substeps. And substeps into subsubsteps.
- **Breaking a large task into bite-size steps** is really all that programming is.
  - Ultimately each step needs to be a command that the language understands.

By Patrick Burns - see
http://www.burns-stat.com/pages/Tutor/more_R_blankscreen.html

Do the steps

- Once you have the task broken down into steps, do the easy steps first.
- This violates my real-life motto of saving the best until last, but there are reasons for doing the easy parts first:
  - your brain will work on solving the hard steps while you do the easy steps. The hard steps may not be inherently hard, you might effortlessly twig on the solution given some time.
  - finishing a step might show you that the whole enterprise is misdirected
  - doing easy steps first might save you a lot of time in this regard.

By Patrick Burns - see
http://www.burns-stat.com/pages/Tutor/more_R_blankscreen.html

Make mistakes on purpose

- Make mistakes using R. That is, experiment. That's what the pros do.
- Two benefits of experimenting are:
  - You learn how things work (often reasonably efficiently).
  - You learn to maintain your equilibrium when something goes wrong.

By Patrick Burns - see http://www.burns-stat.com/pages/Tutor/hints_R_begin.html

Interacting with R

- Can use up/down arrow keys to go through command history.
  - Within a command, use left/right arrow keys to edit.
- History can be saved over sessions
  - ?history
- Multiple commands can be put onto one line, using ";" as separator
  between lines, e.g.
  - x<-10; y<-3; a <- 5.
- TAB can do object/file completion.

Modified from original slide by Eglen (2009).

Objects and Functions

- R manipulates objects.
- Each object has a name and a type (vector, matrix, list, . . .)
- Name of an object: letters (upper/lower case are distinct), digits, period. Start with a letter.
- Objects set by way of assignment.
  - Use the assignment operator '<-' rather than = wherever possible.
  - Does 'i = i+1' make sense?
  - HINT: Option + minus or Alt + minus types `<-` in one keystroke within RStudio within an R chunk.

Modified from original slide by Eglen (2009).

My first R session

```r
options(width=50) # Narrow output width for this slide.
x <- rnorm(n=45, mean=4)
round(x,2)
```

```
##  [1] 3.60 4.63 6.30 4.39 1.75 3.85 1.86 5.14 4.10
## [10] 3.60 5.50 4.65 2.82 5.86 3.24 3.80 4.33 4.79
## [19] 3.38 5.21 2.05 3.48 4.01 4.19 5.61 5.63 5.08
## [28] 3.10 4.15 3.11 5.63 3.87 4.42 3.48 2.85 3.62
## [37] 2.52 4.15 3.23 5.93 2.78 2.10 2.84 3.19 4.49
```

My first R session

```r
mean(x)
```

```
## [1] 3.9626
```

```r
range(x)
```

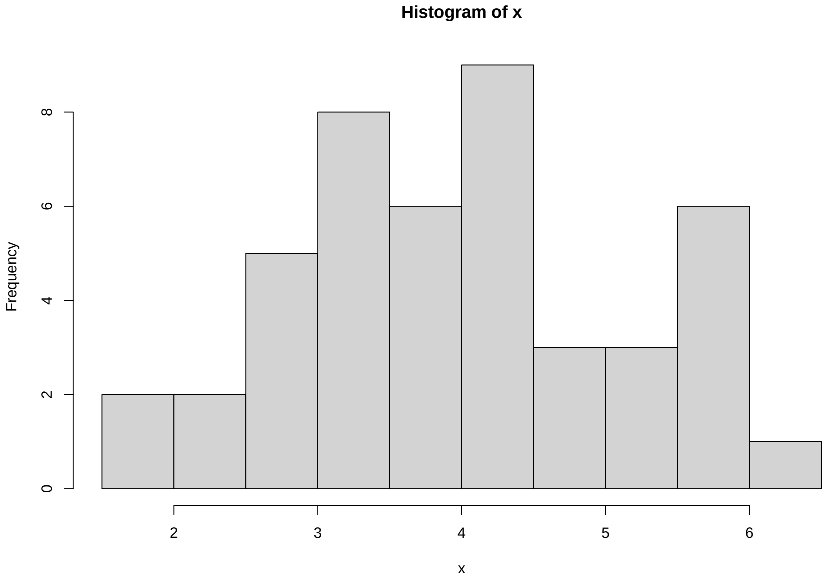
```
## [1] 1.745464 6.300039
```

```r
summary(x)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   1.745   3.186   3.874   3.963   4.651   6.300
```

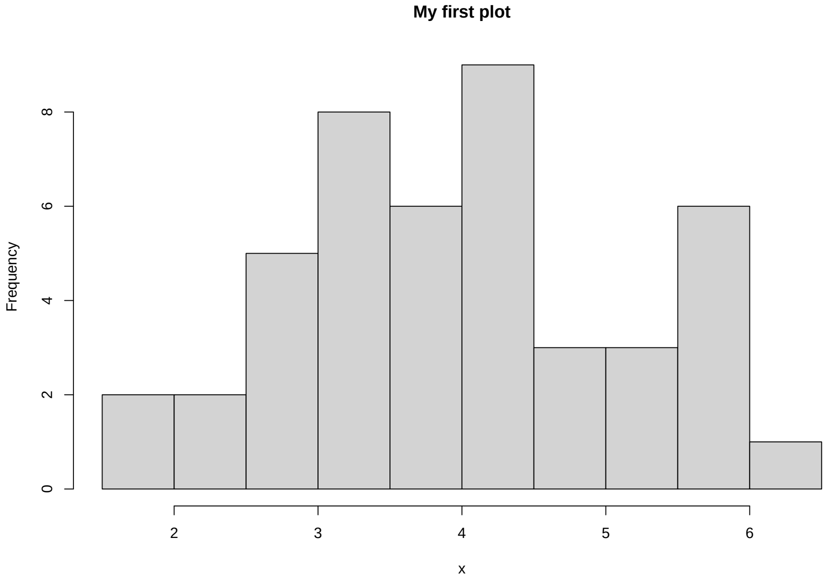
My first R session

```
hist(x)
```



My first R session

```
hist(x, main='My first plot')
```



Creating an object

- To create an R object, we use the assignment operator <-
- Here we create an R object named `x` that contains the single value 5:

```
x <- 5   # Assign 5 to x
x        # Print the object by typing its name
```

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

Object names

- Use understandable names
- Case sensitive
- Avoid using reserved names and function names
  - `TRUE`, `function`, `c`, `T`, `df`, `data`, etc.
- Use underscores (`my_data`) instead of dots (`my.data`)
- Use nouns for objects, verbs for functions.
- See these help pages for more information
  - `?make.names`
  - `?Reserved`

Functions

- Built-in or added via R packages or write your own.
- Input is specified via the **arguments**
  - Arguments may have default values.
- Often return/output a result and/or an R object
- Get help for a function by typing `?` followed by the function's name
  - `?sqrt`

```r
x <- 2
sqrt(2) # x is the input argument.
```

```
## [1] 1.414214
```

Functions

```
x <- sqrt(2)
x
```

```
## [1] 1.414214
```

```
round(x) # Default is to round to 0 digits
```

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

```
args(round) # What are the arguments of round()?
```

```
## function (x, digits = 0)
## NULL
```

```
round(x, digits = 3) # Use 'digits' argument
```

```
## [1] 1.414
```

Objects and functions

- Use [ ] for accessing elements of R objects.
- Use ( ) for calling functions.

```
age <- c(15, 19, 30)
age[2] ## [] for accessing elements.
```

```
## [1] 19
```

```
length(age) ## () for calling functions.
```

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

Modified from original slide by Eglen (2009).

Question

- What does ‘a <- 9; a < - 9’ do?

Answer

```
a <- 9
a
```

```
## [1] 9
```

```
a < - 9
```

```
## [1] FALSE
```

Assignment

- Note also that assignments return values:

```
y <- 1 + (x <- 9)
a <- b <- 0
y
```

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

```
x
```

```
## [1] 9
```

```
a
```

```
## [1] 0
```

```
b
```

```
## [1] 0
```

Modified from original slide by Eglen (2009).

Data types in R

- `integer`
- `character`
- `numeric`
- `logical`
    - `TRUE`, `FALSE`
- `date`
    - `Date`, `POSIXct`, `difftime`
- `complex`
    - complex numbers
- `raw`
    - raw bytes

Special values: `NA`, `NaN`, `Inf`

- `NA`: 'not applicable', the missing value code
    - Most operations that involve an `NA` return an `NA`.
- `NaN`: 'not a number', created by invalid mathematical operations.
- `Inf`: infinity

```r
max(c(1,2,NA))
```

```
## [1] NA
```

```r
max(c(1,2,NA), na.rm = TRUE)
```

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

```r
sqrt(-1)
```

```
## Warning in sqrt(-1): NaNs produced

## [1] NaN
```

Data structures in R

- Homogeneous - contains all the same type of data
  - **vectors (1 dimension)**
  - matrices (2 dimensions)
  - arrays (n dimensions)
  - factors
- Heterogeneous - can contain mixtures of data
  - lists
  - data frame
    - tibbles

Vectors

- Vectors are a fundamental object for R.
- Scalars are treated as vector of length 1.
- Construct vectors with the `c()` (combine) function.

```
y <- c(10, 20, 40)
y[2]  # Second element of the y vector
```

```
## [1] 20
```

```
length(y)
```

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

```
x <- 5
length(x)
```

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

Modified from original slide by Eglen (2009).

Vectors

Some operations work element by element, others on the whole vector, compare:

```
y <- c(20, 49, 16, 60, 100)
min(y)
```

```
## [1] 16
```

```
range(y)
```

```
## [1]  16 100
```

```
sqrt(y)
```

```
## [1]  4.472136  7.000000  4.000000  7.745967
## [5] 10.000000
```

```
log(y)
```

```
## [1] 2.995732 3.891820 2.772589 4.094345 4.605170
```

Generating vectors

Many short hand methods for regular sequences; c() for irregular.

```
x <- seq(from=1, to=9, by=2)
x
```

```
## [1] 1 3 5 7 9
```

```
y <- seq(from=2, by=7, length=3)
y
```

```
## [1]  2  9 16
```

```
z <- 4:8
z
```

```
## [1] 4 5 6 7 8
```

```
a <- seq.int(5) ## fast for integers
a
```

```
## [1] 1 2 3 4 5
```

Generating vectors

```
b <- c(3, 9, 2)
b
```

```
## [1] 3 9 2
```

```
d <- c(a, 10, b)
d
```

```
## [1]  1  2  3  4  5 10  3  9  2
```

```
e <- rep( c(1,2), 3)
e
```

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

```
f <- integer(7)
f
```

```
## [1] 0 0 0 0 0 0 0
```

Modified from original slide by Eglen (2009).

Accessing and setting elements

```r
x <- seq(from=100, by=1, length=8)
x
```

```
## [1] 100 101 102 103 104 105 106 107
```

```r
x[3] ## just element 3.
```

```
## [1] 102
```

```r
x[c(2,4)] ## element 2 and 4
```

```
## [1] 101 103
```

```r
x[1:5]
```

```
## [1] 100 101 102 103 104
```

```r
bad <- 1:4
x[-bad] ## negative indicies exclude elements
```

```
## [1] 104 105 106 107
```

Accessing and setting elements

Can also provide a logical vector of same length as vector (logical values explained later).

```
x <- c(5, 2, 9, 4)
v <- c(T, F, F, T)
x[v]
```

```
## [1] 5 4
```

Modified from original slide by Eglen (2009).

Accessing and setting elements

Elements can be set in several ways

```
x <- rep(0,10)
x
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0
```

```
x[1:3] <- 2
x
```

```
##  [1] 2 2 2 0 0 0 0 0 0 0
```

```
x[5:6] <- c(-5, NA)
x
```

```
##  [1]  2  2  2  0 -5 NA  0  0  0  0
```

```
x[7:10] <- c(1,9) ## recycling.
x
```

```
##  [1]  2  2  2  0 -5 NA  1  9  1  9
```

Recycling rule

Recycling is convenient, but dangerous; when vectors are of different lengths, the shorter one is often recycled to make a vector of the same length.

```
a <- c(1,5) + 2
a
```

```
## [1] 3 7
```

```
x <- c(1,2); y <- c(5,3,9,2)
x
```

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

```
y
```

```
## [1] 5 3 9 2
```

```
x + y
```

```
## [1]  6  5 10  4
```

Recycling rule

```
x
```

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

```
y
```

```
## [1] 5 3 9 2
```

```
c(y,1)
```

```
## [1] 5 3 9 2 1
```

```
x + c(y,1) ## odd recycling, warning.
```

```
## Warning in x + c(y, 1): longer object length is
## not a multiple of shorter object length
```

```
## [1]  6  5 10  4  2
```

Be aware of the recycling rule; an easy place to make subtle mistakes.

Naming indexes of a vector

```
joe <- c(24, 1.70)
joe
```

```
## [1] 24.0  1.7
```

```
names(joe)
```

```
## NULL
```

```
names(joe) <- c('age', 'height')
joe
```

```
##    age height
##   24.0    1.7
```

```
joe['height']
```

```
## height
##    1.7
```

Modified from original slide by Eglen (2009)

Naming indexes of a vector

```
names(joe) <- c('age', 'height')
joe['age']
```

```
## age
##  24
```

```
joe['height']
```

```
## height
##    1.7
```

Referring to index by name rather than by position can make code more readable, and flexible. Cannot do things like x[1:4] easily though, since you need to name all four elements you want. Note: in second use of names() above, we are actually using the *replacement function* names <-, see later

Modified from original slide by Eglen (2009).

Common functions for vectors

- length()
- rev()
- sum(), cumsum(), prod(), cumprod()
- mean(), sd(), var(), median()
- min(), max(), range(), summary()
- exp(), log(), sin(), cos(), tan() [radians, not degrees]
- round(), ceil(), floor(), signif()
- sort(), order(), rank()
- which(), which.max()
- any(), all()

Modified from original slide by Eglen (2009).

Common functions for vectors

Functions can be called within function calls; the following are equivalent:

```
x <- c(3, 2, 9, 4)
(y <- exp(x))
```

```
## [1]   20.085537    7.389056 8103.083928
## [4]   54.598150
```

```
(z1 <- which(y > 20)) ## case 1
```

```
## [1] 1 3 4
```

```
(z2 <- which ( exp(x) > 20)) ## case 2
```

```
## [1] 1 3 4
```

```
all.equal(z1, z2)
```

```
## [1] TRUE
```

Classes

Need to be aware of the class of the vector

```
(x <- c(3, 2))
```

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

```
class(x)
```

```
## [1] "numeric"
```

```
(y <- c(1, 'dog') )
```

```
## [1] "1"   "dog"
```

```
class(y)
```

```
## [1] "character"
```

Classes

The class of a combined vector may differ from that of its parts. **All elements of a vector are of the *same* type.**

```
(z <- c(4,TRUE))
```

```
## [1] 4 1
```

```
class(z)
```

```
## [1] "numeric"
```

```
(tg <- c(x,y,z))
```

```
## [1] "3"   "2"   "1"   "dog" "4"   "1"
```

```
class(tg)
```

```
## [1] "character"
```

Coercion

If you construct a vector with mixed data types, all elements will be coerced to the most flexible type. From least to most flexible: logical, integer, numeric, and character

```
class(c(TRUE, FALSE, 1L))
```

```
## [1] "integer"
```

```
class(c(1L, 2.3))
```

```
## [1] "numeric"
```

```
class(c(1L, 2.3, "A"))
```

```
## [1] "character"
```

Names

Can be useful to assign names to the vector elements

```
(z <- c(4,5,1))
```

```
## [1] 4 5 1
```

```
names(z) <- c('four','five','one')
z
```

```
## four five  one
##    4    5    1
```

```
(x <- c(four=4,five=5,one=1))
```

```
## four five  one
##    4    5    1
```

Questions

- When vectors are used in a mathematical expression, how are the operations performed?
- How would you remove the third element of a vector of length 10?
- How would you find all elements of a numerical vector that are greater than 2?

Answers

- When vectors are used in a mathematical expression, the operations are applied to each element, one by one.
- To remove third element of a vector x of length 10, do this: `x[-3]`
- To list all elements of a numerical vector that are greater than 2, do this: `x[x>2]`

Question

- What are the three ways to select elements?

Answer

1. numerical index
2. logical index
3. names

```r
x <- c(A="a",B="b") # Create a named vector
x[2]
```

```
##   B
## "b"
```

```r
x[c(TRUE, FALSE)]
```

```
##   A
## "a"
```

```r
x["B"]
```

```
##   B
## "b"
```

Part I: Important points

- Programming is breaking a large task into bite-size steps.
- R objects are created using the assignment operator `<-`
- Function behavior can be changed by using different values for their arguments.
- R objects can be examined with `class` and `str` commands.
- Vectors are used a lot in R.
- There are three different ways to select elements from a vector.
- With vectors, be careful about coercion to characters and recycling.
- All elements of a vector are of the *same* type.

Data structures in R

- Homogeneous - contains all the same type of data
  - vectors (1 dimension)
  - **matrices (2 dimensions)**
  - arrays (n dimensions)
  - factors
- Heterogeneous - can contain mixtures of data
  - lists
  - data frame
    - tibbles

Matrices

A matrix is just a vector with some additional mark-up to reformat it. Matrix stored in column-major order (like Fortran, unlike C).

```
x <- 1:6
is.matrix(x)
```

```
## [1] FALSE
```

```
dim(x) <- c(2,3)
is.matrix(x)
```

```
## [1] TRUE
```

```
x
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

Matrices

```r
x
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```r
class(x)
```

```
## [1] "matrix" "array"
```

```r
dim(x)
```

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

Matrices

```
x
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
x[2,2]
```

```
## [1] 4
```

```
x[1,] ## extracting values.
```

```
## [1] 1 3 5
```

```
x[1:2, 2:3]
```

```
##      [,1] [,2]
## [1,]    3    5
## [2,]    4    6
```

Modified from original slide by Eglen (2009).

Matrices

```
x
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
x[,2] ## not column vector!
```

```
## [1] 3 4
```

```
x[,2,drop=F] ## gotcha!
```

```
##      [,1]
## [1,]    3
## [2,]    4
```

Note that everything in a matrix is of the same type.

Modified from original slide by Eglen (2009).

Typical matrix construction methods

- matrix()
- cbind() or bind_cols()
- rbind() or bind_rows()

```
(m <- matrix( floor(runif(6, max=50)), nrow=3)) ##ncol=2
```

```
##      [,1] [,2]
## [1,]   38   18
## [2,]   15   22
## [3,]   49   42
```

```
(x <- rbind( c(1,4,9), c(2,6,8), c(3,2,1)))
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    9
## [2,]    2    6    8
## [3,]    3    2    1
```

Modified from original slide by Eglen (2009).

Typical matrix construction methods

Recycling:

```
(y <- cbind( c(1,2,3), 5, c(4,5,6))) # recycling again
```

```
##      [,1] [,2] [,3]
## [1,]    1    5    4
## [2,]    2    5    5
## [3,]    3    5    6
```

Modified from original slide by Eglen (2009).

Typical matrix construction methods

Note that matrix indices can also be named:

```
dimnames(m) <- list(student=c('ann', 'bob', 'joe'),
                    exam=c('math', 'french'))
m
```

```
##        exam
## student math french
##     ann   38     18
##     bob   15     22
##     joe   49     42
```

```
m['bob',] ## get bob's scores
```

```
##   math french
##     15     22
```

Modified from original slide by Eglen (2009).

Common matrix operations

- diagonal: diag(x) ## watch if x matrix or scalar.
- matrix multiplication: %*% vs * (element-wise)

```
(x <- matrix(1:4, 2,2))
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```
(i <- diag(2)) ## 2x2 identity matrix x
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```

```
x * i ## not x!
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    4
```

Common matrix operations

- matrix multiplication: %*% vs * (element-wise)

```
x
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```
x * i # not x!
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    4
```

```
x %*% i  # Matrix multiplication
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

Modified from original slide by Eglen (2009).

Common matrix operations

- transpose: `t(x)`
- `dim`, `nrow`, `ncol`
- inverse: `solve(x)`, `x %*% solve(x) == diag(nrow(x))`
- Arrays as extension of matrices to multiple dimensions.
  - `x <- array(1:12, c(2,2,3))`.

Modified from original slide by Eglen (2009).

Question

- What would happen if we issued this R command: `matrix(c(1,2,"a","b"),nrow=2)`?

Answer

- If we issued this R command: `matrix(c(1,2,"a","b"),nrow=2)`, all the elements would be converted to characters.

```
matrix(c(1,2,"a","b"),nrow=2)
```

```
##      [,1] [,2]
## [1,] "1"  "a"
## [2,] "2"  "b"
```

Question

What is x[,2]?

```
x <- 1:6; dim(x) <- c(2,3); x
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

Answer

What is x[,2]?

```
x[,2] # not column vector!
```

```
## [1] 3 4
```

```
x[,2, drop = FALSE] # A column vector
```

```
##      [,1]
## [1,]    3
## [2,]    4
```

Data structures in R

- Homogeneous - contains all the same type of data
    - vectors (1 dimension)
    - matrices (2 dimensions)
    - arrays (n dimensions)
    - factors
- Heterogeneous - can contain mixtures of data
    - **lists**
    - data frame
        - tibbles

What is a list?

A list is used to collect a group of objects of different sizes and types. Very flexible. Often returned as the result of a complex function (e.g. model fit) to return all relevant information in one object.

```
l <- list(id='joe', height=1.70, dob=c(1960, 12, 1))
l
```

```
## $id
## [1] "joe"
##
## $height
## [1] 1.7
##
## $dob
## [1] 1960   12    1
```

What is a list?

```
length(l)
```

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

```
names(l) ## show components
```

```
## [1] "id"     "height" "dob"
```

```
l$height ## access an element.
```

```
## [1] 1.7
```

```
unlist(l) ## compact way of viewing it.
```

```
##     id height   dob1   dob2   dob3
##  "joe"  "1.7" "1960"   "12"    "1"
```

What is a list?

- Different components of a list can have different modes
- List elements can either be accessed
  - by name (e.g. l$height)
  - by position (l[[2]]).
- When using numbers to index list, compare l[2] (a list with one element) with l[[2]]. You can therefore do l[2:3] but not l[[2:3]].

Modified from original slide by Eglen (2009).

What is a list?

```r
unlist(l)
```

```
##      id height   dob1   dob2   dob3
##   "joe"  "1.7" "1960"   "12"    "1"
```

```r
l[1]
```

```
## $id
## [1] "joe"
```

```r
l[[1]]
```

```
## [1] "joe"
```

What is a list?

```
unlist(l)
```

```
##      id height   dob1   dob2   dob3
##   "joe"  "1.7" "1960"   "12"    "1"
```

```
str(l[1])
```

```
## List of 1
##  $ id: chr "joe"
```

```
str(l[[1]])
```

```
##  chr "joe"
```

What is a list?

The summary function will provide information about the top-level elements of a list:

```
unlist(l)
```

```
##      id height   dob1   dob2   dob3 
##   "joe"  "1.7" "1960"   "12"    "1"
```

```
summary(l)
```

```
##        Length Class  Mode     
## id     1      -none- character
## height 1      -none- numeric  
## dob    3      -none- numeric
```

Modifying lists

We can append new items to list either by making a new list from the old one (Ex1) , or directly by assigning new element (Ex2):

```
unlist(l1 <- list(who='fred'))
```

```
##    who
## "fred"
```

```
l1 <- c(l1, height=1.8) ## Ex1
unlist(l1)
```

```
##    who height
## "fred"  "1.8"
```

```
l1[['dob']] <- c(1965, 10, 17) ## Ex2
unlist(l1)
```

```
##    who height   dob1   dob2   dob3
## "fred"  "1.8" "1965"   "10"   "17"
```

Modifying lists

Deleting list items:

```
l1['height'] <- NULL
unlist(l1)
```

```
##     who   dob1   dob2   dob3
## "fred" "1965"   "10"   "17"
```

Modifying lists

Finally, for completeness, here is a way to predefine a list of given length and gradually fill it in:

```
empty <- vector('list', 3) ## Prealloc to given length.
names(empty) <- c('who', 'height', 'dob')
empty[['height']] <- 1.8
empty
```

```
## $who
## NULL
##
## $height
## [1] 1.8
##
## $dob
## NULL
```

Modified from original slide by Eglen (2009).

Question

Describe differences between a list and a vector.

Answer

A list generalizes a vector, as a list's elements can be of different types and dimensions.

Data structures in R

- Homogeneous - contains all the same type of data
    - vectors (1 dimension)
    - matrices (2 dimensions)
    - arrays (n dimensions)
    - factors
- Heterogeneous - can contain mixtures of data
    - lists
    - **data frames**
        - tibbles

Data frames

- A data frame stores a table of data
- Each column can have a different mode (unlike a matrix)
- Each column must be the same length (less flexible than a list)
- Often created using read.table() or read.csv() to read in tabular data
  - Be careful about conversion to factors
  - Be careful about coercion of numerical to character

Data frames

```r
a <- c(1:3)
b <- c('A','B','C')
(d <- data.frame(a,b))
```

```
##   a b
## 1 1 A
## 2 2 B
## 3 3 C
```

```r
dim(d)
```

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

```r
str(d)
```

```
## 'data.frame':    3 obs. of  2 variables:
##  $ a: int  1 2 3
##  $ b: chr  "A" "B" "C"
```

`str` command

The `str` command compactly displays the internal structure of an R object

```r
str(d)
```

```
## 'data.frame':    3 obs. of  2 variables:
##  $ a: int  1 2 3
##  $ b: chr  "A" "B" "C"
```

A similar useful command is the `glimpse` command from the tidyverse.

Data frames: select rows and columns

```
names(d)
```

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

```
names(d) <-c('ID','Grade')
d
```

```
##   ID Grade
## 1  1     A
## 2  2     B
## 3  3     C
```

```
d[2,]     # Row two
```

```
##   ID Grade
## 2  2     B
```

```
d[,1]     # Column one
```

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

Data frames

Compare how a data frame and a list are printed:

```
d
```

```
##   ID Grade
## 1  1     A
## 2  2     B
## 3  3     C
```

```
f <- list(ID=a,Grade=b)
f
```

```
## $ID
## [1] 1 2 3
##
## $Grade
## [1] "A" "B" "C"
```

Data frames

To pull a column out of a data fame, you can use the $ operator followed by the name of the desired column:

```
d$ID
```

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

```
d$Grade
```

```
## [1] "A" "B" "C"
```

Data frames: create a new column

```
d
```

```
##   ID Grade
## 1  1     A
## 2  2     B
## 3  3     C
```

```
d$Name <- c("Bob","Jane","Dan")
d
```

```
##   ID Grade Name
## 1  1     A  Bob
## 2  2     B Jane
## 3  3     C  Dan
```

Tibbles

- Tibbles are an extension of data frames
  - Tibbles typically consist of named lists of vectors, all of the same length.
  - Tibbles can also contain list columns
    - A list column's elements can be lists or tibbles.
    - Nested data

Questions

- Describe differences between a data frame and a matrix.
- Describe differences between a data frame and a list.

Answers

- All elements of a matrix must be of the same type, while each column of a data frame can be its own type.
- A data frame is a list, with the restriction that every element of the list is of the same length.

Data structures in R

- Homogeneous - contains all the same type of data
  - vectors (1 dimension)
  - matrices (2 dimensions)
  - arrays (n dimensions)
  - factors
- Heterogeneous - can contain mixtures of data
  - lists
  - data frames
    - tibbles

How to get help

- If you know the command, then use the question mark
  - ?data.frame
- If you don't know the command, try help.search()
  - help.search("data frame")
- Google search appended with 'in R'
  - data.frame in R
- https://stackoverflow.com
  - Search with the tag `[r]`
- Other R search engines/links
  - http://lib.stat.cmu.edu/R/CRAN/search.html

R Basics Group Exercise

Please try out the `R Basics Group Exercise` in our online HuGen2071 book:

https://danieleweeks.github.io/HuGen2071/Rbasics.html

The End

- What questions do you have?