R Basics

Laurent Gatto
`lg390@cam.ac.uk`

Computational Statistics for Genome Biology
2nd July 2012

Plan

Plan

Hello world

```
> x <- 1 ## a variable
> x

[1] 1

> x = 2 ## overwrite the value x
> x

[1] 2

> y <- length(x) ## calling a function
> y

[1] 1
```

Getting help

- ▶ Just ask!
- ▶ `help.start()` and the HTML help button in the Windows GUI.
- ▶ `help` and `?`: `help("data.frame")` or `?help`.
- ▶ `help.search`, `apropos`
- ▶ Online manuals and mailing lists

- ▶ Local R user groups

```
> ls()

[1] "txt" "x"    "y"

> rm(y)
> ls()

[1] "txt" "x"
```

Plan

```
> c(1,3,9,-1)
```

```
[1]   1   3   9  -1
```

A `vector` contains an indexed set of values

- ▶ index starts at 1;
- ▶ all items are of the same storage `mode`;
- ▶ one of `logical`, `numeric`, `complex` or `character`,

`numeric` can futher be broken into `integer`, `single` and `double` types (only important when passing these to `C` or `Fortran` code, though).

```
> mode(1)

[1] "numeric"

> typeof(1)

[1] "double"

> mode(1L)

[1] "numeric"

> typeof(1L)

[1] "integer"
```

```
> mode("1")
[1] "character"
> typeof("1")
[1] "character"
> mode(TRUE)
[1] "logical"
> typeof(FALSE)
[1] "logical"
> ## as we are talking about booleans...
> TRUE & TRUE
[1] TRUE
> TRUE | FALSE
[1] TRUE
```

The different modes an types can be retrieved and coerced with the `is.*` and `as.*` functions.

```
> x <- 1
> typeof(x)

[1] "double"

> y <- as.integer(x)
> typeof(y)

[1] "integer"

> is.integer(y)

[1] TRUE
```

Special values

```
> NULL; NA; NaN; Inf; -Inf
> is.null(); is.na(); is.infinite()
```

What are the mode and types of these?

All these are objects with a certain `class`.

```
> class(x)

[1] "numeric"

> class("a character")

[1] "character"
```

Creating vectors with functions

```
> vector(mode = "character", length = 3)

[1] "" "" ""

> vector(mode = "numeric", length = 4)

[1] 0 0 0 0

> numeric(4)

[1] 0 0 0 0
```

Creating vectors with functions (2)

```
> x <- c(1, 4, 7, 10) ## concatenate
> x

[1]  1  4  7 10

> y <- 1:5 ## integer sequence
> y

[1] 1 2 3 4 5

> z <- seq(from = 1, to = 10, by = 2)
> z

[1] 1 3 5 7 9
```

Arguments by position or name

```
> z1 <- seq(from = 1, to = 10, by = 2)
> z2 <- seq(1, 10, 2)
> z1 == z2

[1] TRUE TRUE TRUE TRUE TRUE

> all(z1 == z2)

[1] TRUE

> identical(z1, z2)

[1] TRUE
```

Vectorised arithmetic

```
> x <- 1:5; y <- 5:1
> x

[1] 1 2 3 4 5

> y

[1] 5 4 3 2 1

> x + y

[1] 6 6 6 6 6

> x^2

[1]  1  4  9 16 25
```

Matrices

are 2-dimensional vectors

```
> m <- matrix(1:12, nrow = 4, ncol = 3)
> m

     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12

> dim(m)

[1] 4 3
```

What if I don't get the data or dimensions right?

```
> matrix(1:11, 4, 3)

     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8    1

> matrix(1:12, 3, 3)

     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

```
> x <- 1:12
> class(x)

[1] "integer"

> dim(x)

NULL

> dim(x) <- c(4, 3)
> x

     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12

> class(x)

[1] "matrix"
```

Arrays

are n-dimensional vectors

```
> array(1:16, dim = c(2, 4, 2))

, , 1

     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8

, , 2

     [,1] [,2] [,3] [,4]
[1,]    9   11   13   15
[2,]   10   12   14   16
```

list

is an ordered set of elements that can be arbitrary R objects.

```
> ll <- list(a = 1:3, c = length)
> ll

$a
[1] 1 2 3

$c
function (x)   .Primitive("length")

> ll[[1]]

[1] 1 2 3

> ll$c(ll)

[1] 2
```

data.frame

is a 2-dimensional `list`.

```
> dfr <- data.frame(type = c(
+                          rep("case", 2),
+                          rep("ctrl", 2)),
+                        time = rnorm(4))
> dfr

  type      time
1 case 1.1035332
2 case 0.4608016
3 ctrl 1.3241386
4 ctrl 0.5920115
```

```
> dfr[1,]

  type     time
1 case 1.103533

> dfr[1, "time"]

[1] 1.103533

> dfr$time

[1] 1.1035332 0.4608016 1.3241386 0.5920115
```

environment

is an unordered collection of objects.

```
> e <- new.env()
> e[["a"]] <- 1:3
> assign("b", "CSAMA", envir = e)
> ls(e)

[1] "a" "b"

> e$a

[1] 1 2 3

> get("b", e)

[1] "CSAMA"
```

Names

We have seen that function arguments have names, and named our `data.frame` columns. We can also name `matrix`/`data.frame` columns and rows, dimensions, and vector items.

```
> x <- c(a = 1, b = 2)
> x

a b
1 2

> names(x)

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

```
> M <- matrix(c(4, 8, 5, 6, 4, 2, 1, 5, 7), nrow=3)
> dimnames(M) <- list(year =
+                      c(2005, 2006, 2007),
+                      "mode of transport" =
+                      c("plane", "bus", "boat"))
> M

      mode of transport
year   plane bus boat
  2005     4   6    1
  2006     8   4    5
  2007     5   2    7
```

factor

for categorical data

```
> sample.ExpressionSet$type

 [1] Control Case    Control Case    Case    Control Case
[10] Control Case    Control Case    Case    Case    Contro
[19] Case    Case    Control Control Control Control Case
Levels: Case Control
```

Higher order objects

When the data to be stored is more complex, special objects are created to store and handle it in a specialised manner. These higher order objects are constructed using the data types we have seen so far as building blocks.

Let's look at how microarray data is handled in Bioconductor. The eSet model has been re-used for other technologies.

```
> library(Biobase)
> data(sample.ExpressionSet)
> sample.ExpressionSet

ExpressionSet (storageMode: lockedEnvironment)
assayData: 500 features, 26 samples
  element names: exprs, se.exprs
protocolData: none
phenoData
  sampleNames: A B ... Z (26 total)
  varLabels: sex type score
  varMetadata: labelDescription
featureData: none
experimentData: use 'experimentData(object)'
Annotation: hgu95av2
```

```
> class(sample.ExpressionSet)

[1] "ExpressionSet"
attr(,"package")
[1] "Biobase"

> slotNames(sample.ExpressionSet)

[1] "experimentData"    "assayData"         "phenoData"
[4] "featureData"       "annotation"        "protocolData"
[7] ".__classVersion__"

> class?ExpressionSet
```

assayData expression values in identical sized matrices.

phenoData sample annotation in `AnnotatedDataFrame`.

featureData feature annotation in `AnnotatedDataFrame`.

experimentData description of the experiment as a `MIAME` object (see `?MIAME` for more details).

annotation type of chip as a `character`.

protocolData scan dates as a `character`.

The `assayData` **slot**

Stored the expression data of the assay.

```
> exprs(sample.ExpressionSet)[1:4, 1:3]

                      A         B        C
AFFX-MurIL2_at   192.7420  85.75330 176.7570
AFFX-MurIL10_at   97.1370 126.19600  77.9216
AFFX-MurIL4_at    45.8192   8.83135  33.0632
AFFX-MurFAS_at    22.5445   3.60093  14.6883

> dim(exprs(sample.ExpressionSet)) ## or
dim(sample.ExpressionSet)

[1] 500  26
```

The `phenoData` **slot**

stores the meta data about the samples.

```
> phenoData(sample.ExpressionSet)

An object of class "AnnotatedDataFrame"
  sampleNames: A B ... Z (26 total)
  varLabels: sex type score
  varMetadata: labelDescription
```

The `featureData` slot
stores the meta data about the feautres.

```
> featureData(sample.ExpressionSet)

An object of class "AnnotatedDataFrame": none
```

AnnotatedDataFrame

consists of a collection of samples and the values of variables measured on those samples. There is also a description of each variable measured. `AnnotatedDataFrame` associates a `data.frame` with its metadata.

```
> head(pData(sample.ExpressionSet))

     sex    type score
A Female Control  0.75
B   Male    Case  0.40
C   Male Control  0.73
D   Male    Case  0.42
E Female    Case  0.93
F   Male Control  0.22
```

Plan

- One of the most powerful features of R is its ability to manipulate subsets of vectors and arrays.
- As seen, subsetting is done with, [], [,], . . .

Subsetting with positive indices

```
> x <- 1:10
> x[3:7]

[1] 3 4 5 6 7

> x[9:11]

[1]  9 10 NA

> x[0:1]

[1] 1

> x[c(1, 7, 2, NA)]

[1]  1  7  2 NA
```

Assignments with positive indices

```
> x[2] <- 20
> x[4:5] <- x[4:5] * 100
> x[1:6]

[1]   1  20   3 400 500   6
```

Subsetting with negative indices

```
> x <- 1:10
> x[-c(3:7)]

[1]  1  2  8  9 10
```

Subsetting with logical predicates

```
> x[c(TRUE, TRUE, rep(FALSE, 8))]

[1] 1 2

> x > 5

 [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE

> x[x > 5]

[1]  6  7  8  9 10

> x[c(TRUE, FALSE)] ## recycled

[1] 1 3 5 7 9
```

Subsetting by names

```
> x <- c(a = 1, b = 2, c = 2)
> x[c("a", "c")]

a c
1 2

> x[c("a", "d")]

   a <NA>
   1   NA
```

Subsetting matrices

```
> M <- matrix(1:12, 3)
> M[2,3] <- 0
> M

     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    0   11
[3,]    3    6    9   12
```

Subsetting matrices (2)

```
> M < 9

     [,1] [,2]  [,3]  [,4]
[1,] TRUE TRUE  TRUE FALSE
[2,] TRUE TRUE  TRUE FALSE
[3,] TRUE TRUE FALSE FALSE

> M[M < 9] <- -1
> M

     [,1] [,2] [,3] [,4]
[1,]   -1   -1   -1   10
[2,]   -1   -1   -1   11
[3,]   -1   -1    9   12
```

Subsetting lists

```
> ll <- list(a = 1:3, b = "CSAMA", c = length)
> ll[1] ## still a list

$a
[1] 1 2 3

> ll[[1]] ## first element of the list

[1] 1 2 3
```

Subsetting `ExpressionSet` instances

It is reasonable to expect that subsetting operations work also for higher order objects.

```
> sample.ExpressionSet[1:10, 1:2]

ExpressionSet (storageMode: lockedEnvironment)
assayData: 10 features, 2 samples
  element names: exprs, se.exprs
protocolData: none
phenoData
  sampleNames: A B
  varLabels: sex type score
  varMetadata: labelDescription
featureData: none
experimentData: use 'experimentData(object)'
Annotation: hgu95av2
```

Plan

Data IO

read.table creates a `data.frame` from a spreadsheet file.

write.table writes a `data.frame`/`matrix` to a spreadsheet (tsv, csv).

save writes an binary representation of `R` objects to a file (cross-platform).

load load a binary `R` file from disk.

Specialised data formats often have specific i/o functionality (microarray CEL files, XML, ...)

```
> read.table("./Data/data.csv", sep = ",",
+            header = TRUE, row.names = 1)

         Ctrl1 Ctrl2 Case1 Case2
31340_at 4.837 4.057 5.037 5.217
31341_at 8.304 8.123 8.273 8.658
31342_at 7.123 7.406 7.051 6.778
31343_at 3.088 4.404 2.251 3.367
31344_at 5.926 4.988 6.422 6.074
31345_at 5.237 4.044 5.367 5.568
31346_at 5.154 5.266 5.990 5.804
31347_at 7.591 6.303 6.587 6.768
31348_at 4.640 1.633 3.754 3.805
31349_at 4.882 4.912 5.323 5.432
```

```
> read.csv("./Data/data.csv", row.names = 1)

         Ctrl1 Ctrl2 Case1 Case2
31340_at 4.837 4.057 5.037 5.217
31341_at 8.304 8.123 8.273 8.658
31342_at 7.123 7.406 7.051 6.778
31343_at 3.088 4.404 2.251 3.367
31344_at 5.926 4.988 6.422 6.074
31345_at 5.237 4.044 5.367 5.568
31346_at 5.154 5.266 5.990 5.804
31347_at 7.591 6.303 6.587 6.768
31348_at 4.640 1.633 3.754 3.805
31349_at 4.882 4.912 5.323 5.432
```

```
> x <- read.csv("./Data/data.csv", row.names = 1)
> save(x, file = "./Data/data.rda")
> rm(x)
> load("./Data/data.rda")
> x[1:3, ]

         Ctrl1 Ctrl2 Case1 Case2
31340_at 4.837 4.057 5.037 5.217
31341_at 8.304 8.123 8.273 8.658
31342_at 7.123 7.406 7.051 6.778
```

String manipulation (1)

```
> paste("abc", "def", sep = "-")

[1] "abc-def"

> paste0("abc", "def")

[1] "abcdef"
```

String manipulation (2)

```
> month.name[1:4]

[1] "January"  "February" "March"    "April"

> grep("Feb", month.name)

[1] 2

> grep("Feb", month.name, value = TRUE)

[1] "February"

> grepl("Feb", month.name)

 [1] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

String manipulation (3)

```
> month.name[1]

[1] "January"

> length(month.name[1])

[1] 1

> nchar(month.name[1])

[1] 7
```

String manipulation (4)

```
> strsplit("abc-def", "-")

[[1]]
[1] "abc" "def"
```

Comparing and matching (1)

```
> set.seed(1)
> x <- sample(letters[1:10], 6)
> y <- sample(letters[1:10], 6)
> x

[1] "c" "d" "e" "g" "b" "h"

> y

[1] "j" "f" "i" "a" "b" "g"
```

Comparing and matching (2)

```
> intersect(x, y)

[1] "g" "b"

> setdiff(x, y)

[1] "c" "d" "e" "h"

> union(x, y)

 [1] "c" "d" "e" "g" "b" "h" "j" "f" "i" "a"
```

Comparing and matching (3)

```
> x %in% y

[1] FALSE FALSE FALSE  TRUE  TRUE FALSE

> x == y

[1] FALSE FALSE FALSE FALSE  TRUE FALSE

> match(x, y)

[1] NA NA NA  6  5 NA
```

Generating data (1)

```
> seq(1,7,3)

[1] 1 4 7

> rep(1:2, 2)

[1] 1 2 1 2

> rep(1:2, each = 2)

[1] 1 1 2 2
```

Generating data (2)

```
> runif(5)

[1] 0.6870228 0.3841037 0.7698414 0.4976992 0.7176185

> rnorm(5)

[1]  2.4046534  0.7635935 -0.7990092 -1.1476570 -0.2894616
```

About the data

```
> table(sample(letters, 100, replace = TRUE))

a b c d e f g h i j k l m n o p q r s t u v w x y z
2 2 4 4 2 2 4 2 6 4 5 7 9 3 1 3 5 3 5 5 6 4 5 2 2 3

> summary(rnorm(100))

     Min.    1st Qu.     Median       Mean    3rd Qu.       Max.
-1.680000  -0.827700  -0.008126  -0.008863   0.608900   2.659000

> head(x)

[1] "c" "d" "e" "g" "b" "h"

> tail(x)

[1] "c" "d" "e" "g" "b" "h"
```

```
> M <- matrix(rnorm(1000), ncol=4)
> head(M)

           [,1]       [,2]        [,3]       [,4]
[1,]  0.7795840 -0.3398806 -1.44688974 -0.1657650
[2,]  0.7132405  0.6062646  1.01951283  0.5571036
[3,] -0.5428819  1.3411303  1.17854698  1.4443344
[4,]  0.8857784  0.7672873 -0.01025877  0.9013571
[5,] -0.3485947  0.1937257  0.26862487 -0.2220350
[6,] -1.0080546  1.1405667  1.34202887  0.1061913
```

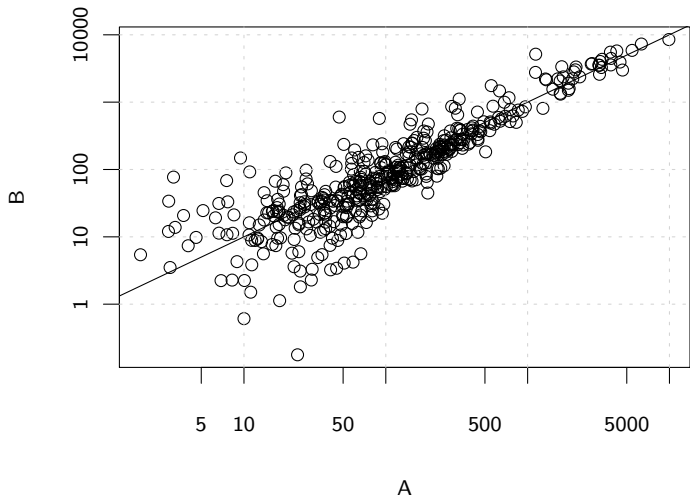# Plan

- Scatterplots with `plot`
- Boxplots with `boxplot`
- Barplots with `barplot`
- Histograms with `hist`
- `smoothScatter`

```
> plot(exprs(sample.ExpressionSet[, 1]),
+      exprs(sample.ExpressionSet[, 2]),
+      log = "xy",
+      xlab = sampleNames(sample.ExpressionSet)[1],
+      ylab = sampleNames(sample.ExpressionSet)[2])
> abline(0, 1)
> grid()
```
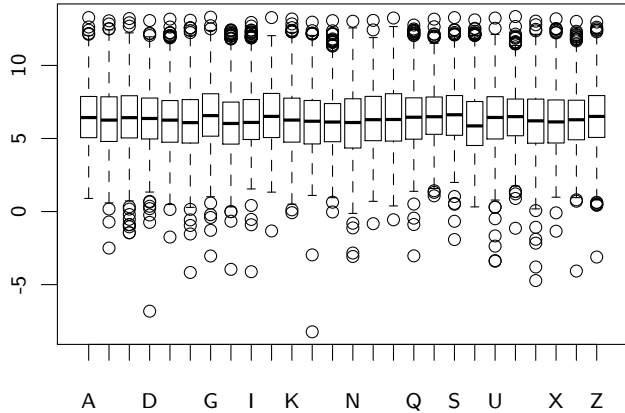
To create subplots, one can use `par(mfrow = c(2,2))`, `layout`, or (for scatterplots)
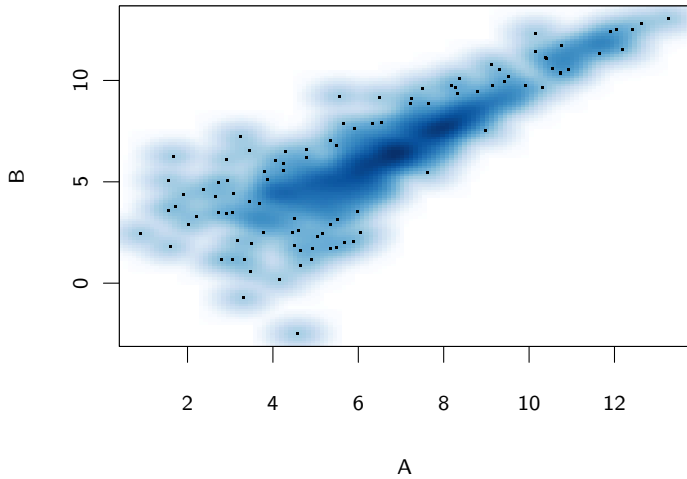
```
> pairs(log2(exprs(sample.ExpressionSet)[, 1:4]),
+       pch = 19,
+       col = "#0000FF20")
```
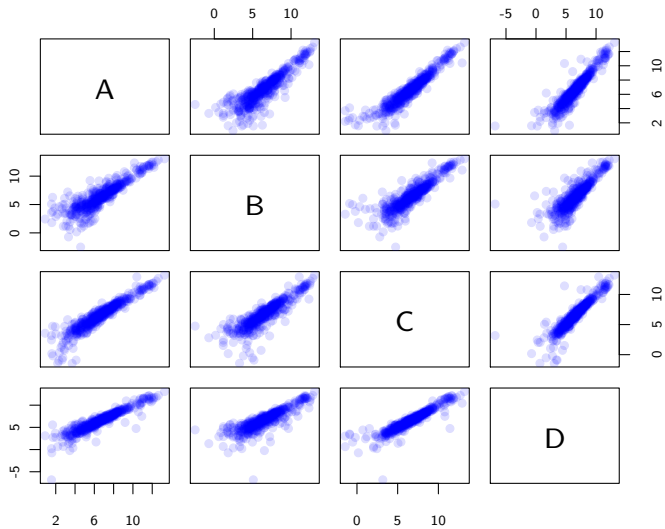
```
> boxplot(log2(exprs(sample.ExpressionSet)))
```

```
> smoothScatter(log2(exprs(sample.ExpressionSet)[,
1:2]))
```

Plan

Flow control

```
> for (var in seq) expr
> while (cond) expr
> repeat expr
> break
```

```
> for (i in 1:4) { ## bad
+   print(i^2)
+ }

[1] 1
[1] 4
[1] 9
[1] 16

> (1:4)^2 ## good

[1]  1  4  9 16
```

The apply family and friends

- Applies a function to each element of an input, being a list or a vector (`sapply`, `lapply`), a matrix or a data frame (`apply`) or an environment (`eapply`).
- Same functionality than an explicit `for` loop, but often more elegant, function-centric, **not** faster.

```
> M <- matrix(1:9, ncol = 3)
> M

     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

> apply(M, 1, max)

[1] 7 8 9

> apply(M, 2, max)

[1] 3 6 9

> 2

[1] 2
```

```
> sapply(month.name[1:2], paste0, "_2012")

        January          February
"January_2012" "February_2012"

> lapply(month.name[1:2], paste0, "_2012")

[[1]]
[1] "January_2012"

[[2]]
[1] "February_2012"
```

```
> mean(rnorm(100))

[1] -0.003273181

> replicate(3, mean(rnorm(100)))

[1] -0.01014122 -0.05703239 -0.10293522

> replicate(2, rnorm(3))

            [,1]         [,2]
[1,] -0.008159868 -0.49746967
[2,]  2.477377478 -0.05147703
[3,]  0.645612638  1.31791727
```

Conditionals

```
> if (cond) expr1 else expr2
> ifelse(cond, expr1, expr2)
> switch
```

```
> x <- 2
> if (x > 0) { ## bad
+     log2(x)
+ } else {
+     log2(-x)
+ }

[1] 1

> log2(abs(x)) ## better

[1] 1
```

Exception handling

try(exprs) will either return the value of the expression expr, or an object of class try-error.

tryCatch provides a more configurable mechanism for condition handling and error recovery.

Writing functions

```
> myFun <- function(param1, param2, ...) {
+    ## function body
+    ## acting on copies of the params
+    ans <- param1 + param2
+    return(ans)
+ }
```

Function facts

- Single `return` value.
- To return multiple items, use a `list` or a proper `object` (see OO programming).
- The return value is either the last statement, or explicit return using `return` (can be called from any where in a function)

Function facts (cont.)

- Functions act on a pass-by-copy semantic.

```
> x <- 1
> f <- function(x) { x <- x + 10; x }
> f(x)
[1] 11
> x
[1] 1
```

Function facts (cont.)

▶ Functions live/act in their own environment and have access to *global* variables.

```
> x <- 1
> f <- function() { x <- x + 10; x }
> f()
[1] 11
> x
[1] 1
```

Anonymous function

```
> M <- matrix(rnorm(50), ncol = 5)
> M[sample(50, 10)] <- NA
> sum(is.na(M))

[1] 10

> apply(M, 1, function(x) sum(is.na(x)))

 [1] 0 3 1 1 1 0 1 0 1 2

> apply(M, 2, function(x) sum(is.na(x)))

[1] 3 1 3 2 1
```

Plan

- Primary mechanism to distribute `R` software is via <u>packages</u>.
- <u>Packages</u> are installed in <u>libraries</u> (directories) on your had disk, and they are loaded with the `library` function.
- There are <u>software</u>, <u>data</u> and <u>annotation</u> packages.
- The Comprehensive R Archive Network (`CRAN`) is the main package repository. It provides an automatic build framework for package authors.
- The Bioconductor project manages its own CRAN-style repository.
- R-forge – `https://r-forge.r-project.org/`

Package installation

- From within R , using `install.packages` - takes care of dependencies.
- Update all installed packages with `update.packages`.
- For Bioconductor packages, use `biocLite`:
  ```
  > source("http://www.bioconductor.org/biocLite.R")
  > ## or, if you have already done so in the past
  > library("BiocInstaller")
  > biocLite("packageName")
  ```

Getting information about packages

- ▶ CRAN/Bioconductor/R-forge web pages
- ▶ Documentation
  ```
  > help(package = "Biobase")
  ```
- ▶ Vignettes (mandatory for Bioconductor packages)
  ```
  > vignette(package = "Biobase")
  > vignette("Bioconductor", package = "Biobase")
  ```
- ▶ Demos
  ```
  > demo("lattice", package = "lattice")
  ```

```
> packageDescription("Biobase")

Package: Biobase
Title: Biobase: Base functions for Bioconductor
Version: 2.17.6
Author: R. Gentleman, V. Carey, M. Morgan, S. Falcon
Description: Functions that are needed by many other packages or which
        replace R functions.
Suggests: tools, tkWidgets, ALL
Depends: R (>= 2.10), utils, BiocGenerics (>= 0.1.0)
Imports: methods, BiocGenerics
Maintainer: Bioconductor Package Maintainer
        <maintainer@bioconductor.org>
License: Artistic-2.0
Collate: tools.R strings.R environment.R vignettes.R packages.R
        AllGenerics.R .....
LazyLoad: yes
biocViews: Infrastructure, Bioinformatics
Packaged: 2012-06-12 00:54:21 UTC; biocbuild
Built: R 2.16.0; x86_64-unknown-linux-gnu; 2012-06-12 18:03:44 UTC;
        unix

-- File: /home/lgatto/R/x86_64-unknown-linux-gnu-library/2.16/Biobase/Meta/pack
```

Package versions
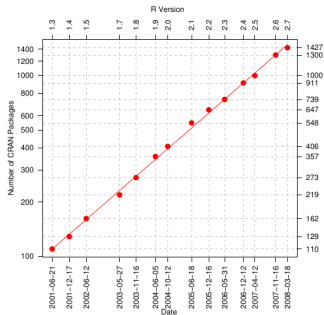
- ▶ New Bioconductor releases appear twice a year. Bioconductor versions are tied to R versions.
- ▶ Stable packages versions are x.y.z where x $>=$ 1 and y is even
- ▶ Devel packages have z odd.
- ▶ New (devel) packages have 0.y.z (y odd); if y $==$ 99, then the package will become 1.0.0 in the next release.

Bioconductor 636 reviewed packages

CRAN 3889 packages

R-forge 1313 projects

(19th June 2012)

Finding packages

- BiocViews – `http://bioconductor.org/packages/release/BiocViews.html`.
- CRAN Task Views – `http://cran.r-project.org/web/views/`.
- sos to search inside contributed R packages.

References

- W. N. Venables, D. M. Smith and the R Development Core Team, An Introduction to R (get it with `help.start()`)
- R. Gentleman, R Programming for Bioinformatics, CRC Press, 2008

```
> toLatex(sessionInfo())
```

- R Under development (unstable) (2012-06-21 r59599), x86_64-unknown-linux-gnu
- Locale: LC_CTYPE=en_GB.UTF-8, LC_NUMERIC=C, LC_TIME=en_GB.UTF-8, LC_COLLATE=en_GB.UTF-8, LC_MONETARY=en_GB.UTF-8, LC_MESSAGES=en_GB.UTF-8, LC_PAPER=C, LC_NAME=C, LC_ADDRESS=C, LC_TELEPHONE=C, LC_MEASUREMENT=en_GB.UTF-8, LC_IDENTIFICATION=C
- Base packages: base, datasets, graphics, grDevices, methods, stats, tools, utils
- Other packages: Biobase 2.17.6, BiocGenerics 0.3.0, cacheSweave 0.6-1, codetools 0.2-8, filehash 2.2-1, formatR 0.5, highlight 0.3.2, parser 0.0-15, pgfSweave 1.3.0, Rcpp 0.9.12, stashR 0.3-5
- Loaded via a namespace (and not attached): digest 0.5.2, tikzDevice 0.6.2

Thank you for your attention