Data Structures

- R has a rich set of *self-describing* data structures.

```r
> class(z)
[1] "character"
> class(x)
[1] "data.frame"
> x[1:2, ,]

<table>
<thead>
<tr>
<th>type</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>case</td>
<td>0.822737</td>
</tr>
<tr>
<td>case</td>
<td>1.964191</td>
</tr>
</tbody>
</table>
```

- There is no need to declare the types of the variables.
Data Structures (continued)

- **vector** - arrays of the same type
- **list** - can contain objects of different types
- **environment** - hashtable
- **data.frame** - table-like
- **factor** - categorical
- **Classes** - arbitrary record type
- **function**
Atomic Data Structures

• In R, vectors are the “base” type, not scalars.

• A vector contains an indexed set of values that are all of the same type:
  – logical
  – numeric
  – complex
  – character

• The numeric type can be further broken down into integer, single, and double types (but this is only important when making calls to foreign functions, eg. C or Fortran.)
Creating Vectors

There are two symbols that can be used for assignment: <- and =.

```r
> v <- 1
> v
[1] 1
> v <- c(1, 2, 3)
> v
[1] 1 2 3
> s <- "a string"
> t <- TRUE
> length(letters)
[1] 26
> letters
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p"
[17] "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
```
Creating Vectors with Functions

- **c** - concatenate

- **seq** (also :) and **rep** - patterns

- **vector** - new vector with default value.

```r
> seq(1, 3)
[1] 1 2 3
> 1:3
[1] 1 2 3
> rep(c(1, 2), 3)
[1] 1 2 1 2 1 2
> vector(mode = "character", length = 5)
[1] "" "" "" "" ""
Matrices and Arrays

- Can be created using `matrix` and `array`.
- Are represented as a vector with a dimension attribute.
- R is column oriented for matrices.
Matrix Examples

> x <- matrix(1:10, nrow = 2)
> dim(x)

[1] 2 5

> x

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

> as.vector(x)

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

The elements of a vector can (and often should) be given names. Names can be specified

- at creation time

  - using names, dimnames, rownames, colnames

```r
> x <- c(a = 0, b = 2)
> x
a b
0 2

> names(x) <- c("Australia", "Brazil")
> x
Australia    Brazil
    0     2
```
Naming (continued)

\[
\begin{align*}
> x & \leftarrow \text{matrix}(c(4, 8, 5, 6), nrow = 2) \\
> \text{dimnames}(x) & \leftarrow \text{list}(c("2005", "2006"), c("plane", "bus")) \\
> x \\
\end{align*}
\]

\[
\begin{array}{ccc}
\text{plane} & \text{bus} \\
2005 & 4 & 5 \\
2006 & 8 & 6 \\
\end{array}
\]
Subsetting

- One of the most powerful features of R is its ability to manipulate subsets of vectors and arrays.

- Subsetting is indicated by [, ].

- Note that [ is actually a function (try get("[")). The behavior can be customized for particular classes of objects.
Subsetting with Positive Indices

• A subscript consisting of a vector of positive integer values is taken to indicate a set of indices to be extracted.

> x <- 1:10
> x[2]
[1] 2
> x[1:3]
[1] 1 2 3

• A subscript which is larger than the length of the vector being subsetted produces an NA in the returned value.

> x[9:11]
[1] 9 10 NA
Subsetting with Positive Indices (continued)

- Subscripts which are zero are ignored and produce no corresponding values in the result.
  
  ```
  > x[0:1]
  [1] 1
  > x[c(0, 0, 0)]
  numeric(0)
  ```

- Subscripts which are NA produce an NA in the result.

  ```
  > x[c(10, 2, NA)]
  [1] 10 2 NA
  ```
Assignments with Positive Indices

- Subset expressions can appear on the left side of an assignment. In this case the given subset is assigned the values on the right (recycling the values if necessary).

```r
> x[2] <- 200
> x[8:10] <- 10
> x

[1] 1 200 3 4 5 6 7 10 10 10
```

- If a zero or NA occurs as a subscript in this situation, it is ignored.
Subsetting with Negative Indexes

- A subscript consisting of a vector of negative integer values is taken to indicate the indices which are not to be extracted.
  > x[-(1:3)]
  [1] 4 5 6 7 10 10 10

- Subscripts which are zero are ignored and produce no corresponding values in the result.

- NA subscripts are not allowed.

- Positive and negative subscripts cannot be mixed.
Assignments with Negative Indexes

- Negative subscripts can appear on the left side of an assignment. In this case the given subset is assigned the values on the right (recycling the values if necessary).

```r
> x = 1:10
> x[-(8:10)] = 10
> x

[1] 10 10 10 10 10 10 10 8 9 10
```

- Zero subscripts are ignored.

- NA subscripts are not permitted.
Subsetting by Logical Predicates

- Vector subsets can also be specified by a logical vector of TRUEs and FALSEs.

```r
> x = 1:10
> x > 5

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

- NA values used as logical subscripts produce NA values in the output.

- The subscript vector can be shorter than the vector being subsetted. The subscripts are recycled in this case.

- The subscript vector can be longer than the vector being subsetted. Values selected beyond the end of the vector produce NAs.
Subsetting by Name

- If a vector has named elements, it is possible to extract subsets by specifying the names of the desired elements.

```r
> x <- c(a = 1, b = 2, c = 3)
> x[c("c", "a", "foo")]
  c   a <NA>
  3   1   NA
```

- If several elements have the same name, only the first of them will be returned.

- Specifying a non-existent name produces an NA in the result.
Exercises

1. Determine (precisely) how R handles non-integer subscripts (e.g. 1.2). How might this produce problems?

2. What value do the following expressions produce.

\[ x = 1:10 \]
\[ x[-11] \]

3. How could you choose all elements of a vector which have odd subscripts? Even subscripts?

4. How are complex subscripts treated?
Subsetting matrices

- when subsetting a matrix, missing subscripts are treated as if all elements are named; so $x[1,]$ corresponds to the first row and $x[,3]$ to the third column.

- for arrays, the treatment is similar, for example $y[,,1]$.  

- these can also be used for assignment, $x[1,]=20$
Subsetting Arrays

- Rectangular subsets of arrays obey similar rules to those which apply to vectors.

- One point to note is that arrays can be treated as either matrices or vectors. This can be quite useful.

```r
> x = matrix(1:9, ncol = 3)
> x[x > 6]
[1] 7 8 9
> x[row(x) > col(x)] = 0
> x
   [,1] [,2] [,3]
[1,] 1 4 7
[2,] 0 5 8
[3,] 0 0 9
```
Custom Subsetting Example

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

> class(sample.ExpressionSet)
[1] "ExpressionSet"
attr(,"package")
[1] "Biobase"

> dim(sample.ExpressionSet)

    Rows Samples
     500     26

> slotNames(sample.ExpressionSet)

[1] "assayData" "phenoData" "experimentData"
[4] "annotation"
Custom Subsetting Example

> sample.ExpressionSet

Instance of ExpressionSet

assayData

  Storage mode: lockedEnvironment
  featureNames: AFFX-MurIL2_at, AFFX-MurIL10_at, AFFX-MurIL4_at,
  Dimensions:
    se.exprs exprs
    Rows  500  500
    Samples  26  26

phenoData

  sampleNames: A, B, C, ..., Y, Z (26 total)
  varLabels:
    sex: Female/Male
type: Case/Control
score: Testing Score

Experiment data
Experimenter name: Pierre Fermat
Laboratory: Francis Galton Lab
Contact information: pfermat@lab.not.exist
Title: Smoking-Cancer Experiment
URL: www.lab.not.exist
PMIDs:

Abstract: A 8 word abstract is available. Use 'abstract' method.

Annotation [1] "hgu95av2"
Custom Subsetting Example

```r
> sample.ExpressionSet[1:2, 2:5]
```

Instance of ExpressionSet

**assayData**
- Storage mode: lockedEnvironment
- featureNames: AFFX-MurIL2_at, AFFX-MurIL10_at
- Dimensions:
  - `se.exprs exprs`
  - Rows: 2 2
  - Samples: 4 4

**phenoData**
- sampleNames: B, C, D, E
- varLabels:
  - sex: Female/Male
type: Case/Control
score: Testing Score

Experiment data
Experimenter name: Pierre Fermat
Laboratory: Francis Galton Lab
Contact information: pfermat@lab.not.exist
Title: Smoking-Cancer Experiment
URL: www.lab.not.exist
PMIDs:

Abstract: A 8 word abstract is available. Use 'abstract' method.

Annotation [1] "hgu95av2"
Vectorized Arithmetic

- Most arithmetic operations in the R language are *vectorized*. That means that the operation is applied element-wise.
  
  ```
  > 1:3 + 10:12
  [1] 11 13 15
  ```

- In cases where one operand is shorter than the other the short operand is recycled, until it is the same length as the longer operand.
  
  ```
  > 1 + 1:5
  [1] 2 3 4 5 6
  > paste(1:5, "A", sep = "")
  [1] "1A" "2A" "3A" "4A" "5A"
  ```

- Many operations which need to have explicit loops in other languages do not need them with R. You should vectorize any functions you write.
Lists

- In addition to atomic vectors, R has a number of recursive data structures. Among the important members of this class are lists and environments.

- A list is a vector which can contain vectors and other lists (in fact arbitrary R objects) as elements. In contrast to atomic vectors, whose elements are homogeneous, lists and environments contain heterogeneous elements.

```r
> lst = list(a = 1:3, b = "a list")
> lst

$a
[1] 1 2 3

$b
[1] "a list"
```
Environments

- One difference between lists and environments is that there is no concept of ordering in an environment. All objects are stored and retrieved by name.

\[ e1 = \text{new.env(hash = TRUE)} \]
\[ e1["a"] \leftarrow 1:3 \]
\[ \text{assign("b", "a list", e1)} \]
\[ \text{ls(e1)} \]

[1] "a" "b"

- Another difference is that for lists partial matching of names is used, for environments it is not.
Subsetting and Lists

- Lists are useful as containers for grouping related things together (many R functions return lists as their values).
- Because lists are a recursive structure it is useful to have two ways of extracting subsets.
- The `[ ]` form of subsetting produces a sub-list of the list being subsetted.
- The `[[ ]]` form of subsetting can be used to extract a single element from a list.
List Subsetting Examples

- Using the [ ] operator to extract a sublist.
  
  ```r
  lst[1]
  ```

  ```r
  $a
  [1] 1 2 3
  ```

- Using the [[ ]] operator to extract a list element.
  
  ```r
  lst[[1]]
  ```

  ```r
  [1] 1 2 3
  ```

- As with vectors, indexing using logical expressions and names are also possible.
List Subsetting by Name

- The dollar operator provides a short-hand way of accessing list elements by name. This operator is different from all other operators in R, it does not evaluate its second operand (the string).

```r
> lst$a
[1] 1 2 3
> lst[["a"]]
[1] 1 2 3
```

- For these accessors partial matching (!) is used.
Environment Accessing Elements

- Access to elements in environments can be through, get, assign, mget.

- You can also use the dollar operator and the \[[\ ]]\ operator, with character arguments only. No partial matching is done.

```r
> e1$a
[1] 1 2 3
> e1[["b"]]
[1] "a list"
```
Assigning values in Lists and Environments

- Items in lists and environments can be replaced in much the same way as items in vectors are replaced.

```r
> lst[[1]] = list(2, 3)
> lst[[1]]
[[1]]
[1] 2

[[2]]
[1] 3

> e1$b = 1:10
> e1$b
[1] 1 2 3 4 5 6 7 8 9 10
```
Data Frames

- Data frames are a special R structure used to hold a set of related variables. They are the R representation of a statistical data matrix. In a `data.frame`, the observations are the rows and the covariates are the columns.

- Data frames can be treated like matrices, and indexed with two subscripts. The first subscript refers to the observation, the second to the variable.

- Data frames are really lists, and list subsetting can also be used on them.
Data Frames (continued)

```r
> df <- data.frame(type = rep(c("case", "control"), c(2, + 3)), time = rexp(5))
> df

       type  time
1    case 0.5388906
2    case 1.8223102
3 control 1.2595001
4  control 0.4415987
5  control 1.8353047

> df$time
[1] 0.5388906 1.8223102 1.2595001 0.4415987 1.8353047

> names(df)
[1] "type" "time"

> rn <- paste("id", 1:5, sep = "")
> rownames(df) <- rn
> df[1:2, ]

       type  time
id1    case 0.5388906
id2    case 1.8223102
```
Classes

- A class consists of a set of *slots* each containing a specific type (character, numeric, etc.).

- *methods* can be defined for classes. A rectangle class that has slots for length and width could have an *area* method.

- Slots are accessed using @, but accessor methods are preferred.
Classes (example)

```r
> setClass("Person", representation(name = "character",
+       height = "numeric", country = "character"))

[1] "Person"

> p <- new("Person", name = "Alice", height = 5, country = "UK")
> p

An object of class "Person"
Slot "name":
[1] "Alice"

Slot "height":
[1] 5

Slot "country":
[1] "UK"

> p@name

[1] "Alice"
```
Getting Help There are a number of ways of getting help:

- `help` and `?:`: `help("data.frame")`
- `help.search`, `apropos`
- `RSiteSearch` (requires internet connection)
- `help.start`
- `sessionInfo`
- Online manuals
- Mailing lists (`sessionInfo`)
Packages

- In R one of primary mechanisms for distributing software is via *packages*

- CRAN is the major repository for getting packages.

- You can either download packages manually or use `install.packages` or `update.packages` to install and update packages.

- In addition, on Windows and in some other GUIs, there are menu items that facilitate package downloading and updating.

- It is important that you use the R package installation facilities. You cannot simply unpack the archive in some directory and expect it to work.
Packages - Bioconductor

- Bioconductor packages are hosted in CRAN-style repositories and are accessible using `install.packages`.

- The most reliable way to install Bioconductor packages (and their dependencies) is to use `biocLite`.

- Bioconductor has both a release branch and a development branch. Each Bioconductor release is compatible with a specific R release.

- Bioconductor packages all have vignettes.
Packages

- Having, and needing many more packages can cause some problems.
- When packages are loaded into R, they are essentially attached to the `search` list, see `search`.
- This greatly increases the probabilities of variable masking, that is one package provides a function that has the same name as a different function in another package.
- Name spaces were introduced in R 1.7.0 to provide tools that would help alleviate some of the problems.
Control-Flow R has a standard set of control flow functions:

- Looping: `for`, `while` and `repeat`.
- Conditional evaluation: `if` and `switch`.
Two Useful String Functions

1. Concatenate strings: paste
2. Search strings: grep
Example: paste

```r
> s <- c("a", "b", "c")
> paste(s, "X", sep = "_")
[1] "a_X" "b_X" "c_X"
> paste(s, collapse = ", ")
[1] "a, b, c"
```
Example: grep

> library("ALL")
> data(ALL)
> class(ALL$mol.biol)

[1] "factor"

> negIdx <- grep("NEG", ALL$mol.biol)
> negIdx[1:10]

[1] NA NA NA NA NA NA NA NA NA NA
The apply Family

- A natural programming construct in R is to *apply* the same function to elements of a list, of a vector, rows of a matrix, or elements of an environment.

- The members of this family of functions are different with regard to the data structures they work on and how the answers are dealt with.

- Some examples, `apply`, `sapply`, `lapply`, `mapply`, `eapply`. 
Using `apply`

- `apply` applies a function over the margins of an array.

- For example,
  ```r
  > apply(x, 2, mean)
  ```
  computes the column means of a matrix `x`, while
  ```r
  > apply(x, 1, median)
  ```
  computes the row medians.

- `(apply)` is implemented in a way which avoids the overhead associated with looping. (But it is still slow and you might use `rowSums` or `colSums`).
Writing Functions

- Writing R functions provides a means of adding new functionality to the language.

- Functions that a user writes have the same status as those which are provided with R.

- Reading the functions provided with the R system is a good way to learn how to write functions.

- If a user chooses she can make modifications to the system functions and use her modified ones, in preference to the system ones.
A Simple Function

- Here is a function that computes the square of its argument.
  
  ```
  > square = function(x) x * x
  > square(10)
  [1] 100
  ```

- Because the underlying arithmetic is vectorized, so is this function.
  
  ```
  > square(1:4)
  [1]  1  4  9 16
  ```
Composition of Functions

- Once a function is defined, it is possible to call it from other functions.

> sumsq = function(x) sum(square(x))
> sumsq(1:10)

[1] 385
Returning Values

- Any single R object can be returned as the value of a function; including a function.

- If you want to return more than one object, you should put them in a list (usually with names), or an S4 object, and return that.

- The value returned by a function is either the value of the last statement executed, or the value of an explicit call to `return`.

- `return` takes a single argument, and can be called from any where in a function.

- `return` is lexically scoped, and can be passed out to other functions, to effect non-local returns.
Control of Evaluation

- In some cases you want to evaluate a function that may fail, but you do not want to exit from the middle of an evaluation.
- In these cases the function `try` can be used.
- `try(expr)` will either return the value of the expression `expr`, or an object of class `try-error`.
- `tryCatch` provides a much more substantial mechanism for condition handling and error recovery.
Name Spaces

- Name spaces were introduced in R 1.7.0, see R News, Vol 3/1 for more details.

- They provide a mechanism that allows package writers to control what functions they import (and hence use) and export (and hence let others use).

- Related functions: `loadedNamespaces` and `:::`.
Object Oriented Programming

- Object oriented programming is a style of programming where one attempts to have software reflections of real-world objects and to write functions (methods) that operate on these objects.

- The R language has two different object oriented paradigms, one S3 is older and should not be used for new projects. The second, S4 is newer and is currently under active development.

- These objects systems are more like OOP in Scheme, Lisp or Dylan than they are like OOP in Java or C++.
Classes

• In OOP there are two basic ingredients, objects and methods.

• An object is an instance of a class, and most OOP implementations have mechanisms to ensure that all objects of a particular class have some common characteristics.

• In most implementations there is some notion of inheritance or class extension. Class B is said to extend class A if a member of B has all the attributes that a member of A does, plus some other attributes.
Generic Functions

- A *generic function* is an interface, or a dispatcher, that examines the type or class of its arguments and invokes the most appropriate method.

- A method is registered with a generic function, by indicating its existence together with the number and types (classes) of its arguments.

- In the previous example, if a generic function is called with an instance of class B and there is no class B method, a class A method could be used.
S3

- S3 OOP has no real mechanism for defining classes or for creating objects from a specific class.

- One can make any object an instance of class *foo*, by assigning a class attribute, `class(x) = "foo"`.

- S3 handles inheritance by setting several different class attributes (but these are not always handled correctly).

- S3 is not suitable for the development of large scale complex systems.
S3 Generic Functions

- The relationship between a generic function and its methods is done by a naming convention. The generic function must have a call to `UseMethod` and the method must have a name that is the name of the generic function concatenated with the name of the class, with the two names separated by a dot.

```r
> mean

function (x, 
UseMethod("mean")
<environment: namespace:base>

> methods("mean")

[1] mean.Date   mean.POSIXct  mean.POSIXlt  mean.data.frame
[5] mean.default mean.difftime
```
References