Advanced R Programming and Package Development

Robert Stojnic <rs550@cam.ac.uk>
Laurent Gatto <lg390@cam.ac.uk>

Course website:
http://logic.sysbiol.cam.ac.uk/teaching/advancedR/

Cambridge System Biology Centre
University of Cambridge

26th November 2012
Prerequisites

- good knowledge of R (data types, functions, scripting ...)
- basic knowledge of CLI
- some Latex knowledge helpful but not essential
- object-oriented programming knowledge helpful but not essential
Plan

1 Introduction
2 Revision of basic R
3 Object-oriented (OO) Programming
4 A few words about R packages
5 Package structure
6 Writing R documentation
7 Other advanced topics
   - Testing your package
   - Debugging
   - Profiling
   - Calling foreign languages
8 Distributing packages
Plan

1. **Introduction**
2. Revision of basic R
3. Object-oriented (OO) Programming
4. A few words about R packages
5. Package structure
6. Writing R documentation
7. Other advanced topics
   - Testing your package
   - Debugging
   - Profiling
   - Calling foreign languages
8. Distributing packages
Course introduction

**Course agenda**
- Object-oriented programming in R: S3 and S4 class systems
- Package development in R: creating and documenting packages
- Other advanced topics: testing, debugging, profiling, C interface
- This is an **intensive** course

**Objectives**

By the end of the course you should have created a working package written in the S4 class system.

You should be able to use the code as a template for your own work. Our example has been chosen for demonstrative purposes.
Course working example: "sequences" package

Working example

We will make a simple package to handle sequence data. This package will be able to load a FASTA file and based on sequence type do some operations, like finding the sequence length or reverse sequence. For simplicity we will manipulate single sequences only.

UML class diagram for the "sequences" package
Plan

1. Introduction

2. Revision of basic R

3. Object-oriented (OO) Programming

4. A few words about R packages

5. Package structure

6. Writing R documentation

7. Other advanced topics
   - Testing your package
   - Debugging
   - Profiling
   - Calling foreign languages

8. Distributing packages
Basic R functions (size reflects frequency of usage)
Defining functions in R

Simple function with 4 arguments:

```r
> # Function to calculate area of rectangle
> area <- function(x1, y1, x2, y2){
+ abs(x2-x1) * abs(y2-y1)
+ }
> area(0, 0, 5, 5)

[1] 25
```

Special argument "..." for any:

```r
> # Plot with a message before the plot
> plotMsg <- function(x, y, ...){
+ cat("Plotting", length(x), "data points!\n")
+ plot(x, y, ...)
+ }
> plotMsg(1:10, 1:10, main="My plot")

Plotting 10 data points!
```
Output of plotMsg()

> plotMsg(1:10, 1:10, main="My plot")

Plotting 10 data points!
Useful R function 1/2

- `readLines()` - reads raw lines of text from a file
- `nchar()` - gives number of characters in a string
  > `nchar("Some text")`
  
  ```
  [1] 9
  ```
- `strsplit()` - split a string by some separator
  > `strsplit("Some text", " ")`
  
  ```
  [[1]]
  [1] "Some" "text"
  ```
  > `strsplit("Some text", "")`
  
  ```
  [[1]]
  [1] "S" "o" "m" "e" "t" "e" "x" "t"
  ```
- `unique()` - unique elements of a vector
  > `unique(c(1, 1, 2, 2, 3))`
  
  ```
  [1] 1 2 3
  ```
  > `unique(c("a", "b", "a"))`
  
  ```
  [1] "a" "b"
  ```
Useful R function 2/2

- grep() - find which elements of vector match regular expression
  ```
  > grep("[AT]+", c("CGC", "TAT", "TATCATA"))
  [1] 2 3
  ```

- sub() - replace matches to regular expression
  ```
  > sub("[AT]+", ",", c("CGC", "TAT", "TATCATA"))
  [1] "CGC"", "", "-CATA"
  ```

- chartr() - translate a string by replacing individual characters
  ```
  > chartr("TA", "AT", "TATCTA")
  [1] "ATACAT"
  ```

- rev() - reverse ordering in a vector
  ```
  > rev(c("TAT", "ATT", "TTT"))
  [1] "TTT" "ATT" "TAT"
  ```

- paste() - concatenate variables into a string representation
  ```
  > paste(c("A", "T", "A"), collapse="")
  [1] "ATA"
  ```

Lists in R

List is a data structure that can hold a vector of any other variables.

```r
> x <- list(a=10, b="text")
> x

$a
[1] 10

$b
[1] "text"

> x$a
[1] 10

> x[['b']]
[1] "text"

> x[[1]]
[1] 10

> names(x)
[1] "a" "b"
```
Everything in R has a class

Everything in R has a type - in object oriented programming called a **class**.

```r
> class(10)
[1] "numeric"

> class(c(1, 2, 3))
[1] "numeric"

> class("Some text")
[1] "character"

> class(matrix(0, nrow=10, ncol=10))
[1] "matrix"

> class(plot)
[1] "function"

> class(table(1:4, 1:4))
[1] "table"
```
Recommended coding standards

**Coding standards**

- Use `<-` for assignment rather than `=`.
- Avoid long lines (80 characters).
- Use spaces for indentation (2 or 4).
- No semi-colons (unless you have several expression in a line).
- Start names with upper case for classes, lower for the rest.
- Use syntax highlighting
Plan

1. Introduction
2. Revision of basic R
3. **Object-oriented (OO) Programming**
4. A few words about R packages
5. Package structure
6. Writing R documentation
7. Other advanced topics
   - Testing your package
   - Debugging
   - Profiling
   - Calling foreign languages
8. Distributing packages
Object-oriented Programming (OOP)

Object-oriented vs Procedural programming

- OOP introduced in 1970s in Smalltalk but gained wider popularity in 1990s with programming languages like C++ and Delphi
- Traditional (procedural) programming - data and functions decoupled
- Object-oriented programming - data and functions tied together in objects

OOP concepts

- **Abstraction** - related data is stored and handled together
- **Inheritance** - code reuse by hierarchy of more-to-less general object types (classes)
- **Polymorphism** - the most appropriate function is called based on the dataset (e.g., various plot functions)
Procedural vs Object-oriented Programming

Procedural programming

```r
> area <- function(x1,y1,x2,y2){
+   abs(x2-x1)*abs(y2-y1)
+ }
> area(0, 0, 5, 5)
[1] 25
```

Object-oriented programming

```r
> setClass("Rectangle",
+   representation = representation(
+     x1 = "numeric",
+     y1 = "numeric",
+     x2 = "numeric",
+     y2 = "numeric")
+ )
> setGeneric("area", function(obj)
+       standardGeneric("area"))

[1] "area"

> setMethod("area", "Rectangle", function(obj){
+   abs(obj@x2 - obj@x1) * abs(obj@y2 - obj@y1)
+ })

[1] "area"

> rect = new("Rectangle", x1=0, y1=0, x2=5, y2=5)
> area(rect)
[1] 25
```
R has two object-oriented frameworks:

- **S3** - older and less formal (i.e. ad-hoc) framework with no explicit class definitions. Many parts of base R use S3, e.g. plotting, linear modelling, ...
  - limited introspection, single inheritance, single dispatch, instance-based

- **S4** - full-fledged object-oriented framework, de-facto standard for most modern packages and required for Bioconductor packages.
  - introspection, multiple inheritance, multiple dispatch (introduces a small overhead)
Course working example

Working example revisited

Working example for this course will be manipulating DNA/RNA sequence data.

Functions we would like to have:

- readFasta() - read in a single sequence from a FASTA file
- id(), seq() - return the ID of sequence and the sequence (accessors)
- rev() - return reverse DNA/RNA sequence
- length() - return DNA/RNA sequence length
- comp() - return complementary DNA/RNA sequence
- transcribe() - return RNA sequence for DNA sequence

Goal

The final product should be an R package using S4 framework. But we need to get there, so let's start with a procedural and S3 implementation...
We will start with the implementation of `readFasta()`. This function should load the data from a FASTA file and somehow represent it in R.

A sample FASTA file:

```
> example dna sequence
agcatacgcacgactacgacacatcgacacatcgacactacacagacactacagactacagacatcagacactacatatttacatcatcatcagagattatatta
acatcagacatcgacacatcatcatcagcatcat
```

**Sequence description**

Notice that a sequence is described by the:

- name (example dna sequence)
- nucleotide sequence
- sequence alphabet (in case of DNA ATGC, for RNA AUGC)
Naive readFasta() implementation

readFasta() implementation

Read in a sequence from FASTA file and return the id, sequence and alphabet in a list:

```r
> readFasta <- function(infile){
+   lines <- readLines(infile)
+   header <- grep("^>" , lines)
+   
+   if ( length(header) > 1 ) {
+     warning("Reading first sequence only.")
+     lines <- lines[header[1]: (header[2]-1)]
+     header <- header[1]
+   }
+   
+   id <- sub("^> *" , "" , lines[header] , perl = TRUE)
+   sequence <- toupper(paste( lines[(header+1) : length(lines)] , collapse="" ))
+   alphabet <- unique(strsplit(sequence,"" )[[1]])
+   
+   return.value <- list(id=id, sequence=sequence, alphabet=alphabet)
+   class(return.value) <- "GenericSeq"
+   
+   return.value
+ }
```
S3 objects

```r
> s <- readFasta("aDnaSeq.fasta")
> s

$id
[1] "example dna sequence"

$sequence
[1] "AGCATACGACGACTACGACACTACGACATCAGACACTACAGACTACTACGACTACAGACATCAGACACTACATATTTACATCATCAGAGATTATATTAACATCAGACATCGACACATCATCATCAGCATCAT"

$alphabet
[1] "A" "G" "C" "T"

attr("class")
[1] "GenericSeq"

> names(s)

[1] "id"  "sequence" "alphabet"
```

**S3 object definition**

Any variable that has a "class" attribute is an S3 object. Now we can write class-specific functions - methods.
S3 methods and dispatch

Methods: class-specific functions

Lets write the `id()` method that will return the sequence id. There are two parts to defining a class-specific function (method):

- Defining a generic function
- Defining the class method

```r
> id <- function(x){ UseMethod("id") } # generic
> id.GenericSeq <- function(x){ x$id } # method
> id(s)

[1] "example dna sequence"
```

S3 methods mechanism

Generic function has the desired function name and contains only one command `UseMethod("functionName")` called a dispatch command. This command based on the first parameter’s class calls an appropriate function of format `functionName.className`. If such function doesn’t exist `functionName.default` is called.
Adding to existing S3 generics

**The seq() method**

Now consider the seq() function. This function already exists (try ?seq). We would like to retain this old function, but also add our seq() that return the DNA/RNA sequence.

The seq() function is already a generic. We don’t need to redefine it.

```r
> seq

function (...) 
UseMethod("seq")
<bytecode: 0x19e5510>
<environment: namespace:base>

> methods("seq")

[1] seq.Date  seq.default seq.POSIXt

> seq.GenericSeq = function(x){ x$sequence }
> seq(s)

[1] "AGCATACGACGACTACGACACTACGACATCAGACACTACAGACTACTACGACTACAGACATCAGACACTACATATTTACATCATCAGAGATTATATTAACATCAGACATCGACACATCATCATCAGCATCAT"
```
S3 methods exercises

Look at the code we have written so far, understand it, and then solve the following exercise.

Exercise 1:
Explore some of the built-in generics and methods. Try the following commands:

```
methods("summary")
methods(class="lm")
```


Both `length()` and `rev()` are already generic functions, but `alphabet()` is not. Add these methods for class `GenericSeq`:

- `length()` should return the length of the DNA/RNA sequence
- `alphabet()` should return the alphabet of the sequence
- `rev()` should return the sequence in reverse (Hint: try to use functions `strsplit()` and the existing base `rev()` function).
S3 inheritance

Reusing class methods

So far we have written methods for GenericSeq that work with any sequence type. Now let's introduce a new class DnaSeq. We want to **inherit** all methods from GenericSeq - to achieve this simply set the class attribute to all applicable class names.

```r
> setSeqSubtype <- function(s){
+   if (all( alphabet(s) %in% c("A","C","G","T") )) {
+     class(s) <- c("DnaSeq", "GenericSeq")
+   } else if (all( alphabet(s) %in% c("A","C","G","U") )) {
+     class(s) <- c("RnaSeq", "GenericSeq")
+   } else {
+     stop("Alphabet ", alphabet(s) ," is unknown.")
+   }
+   return(s)
+ }
> s.dna <- setSeqSubtype(s)
> class(s.dna)

[1] "DnaSeq" "GenericSeq"
```
DnaSeq methods

Define a DnaSeq method `complement()`. All GenericSeq methods still work with DnaSeq objects, but the `complement()` only works with DnaSeq.

```r
> complement <- function(x){ UseMethod("complement") }
> complement.DnaSeq = function(x) chartr("ACGT", "TGCA", seq(x))

> id(s) # works on GenericSeq
[1] "example dna sequence"

> id(s.dna) # works on DnaSeq, GenericSeq
[1] "example dna sequence"

> try({ complement(s) }) # fails with error
> complement(s.dna)

[1] "TCGTATGCTGCTGATGCTGTGATGCTGTAGTCTGTGATGTCTGATGATGCTGATGTCTGTAGTCTGTGATGTATAAATGTAGTCTGTAGCTGTGTAGTAGTAGTCGTAGTA"
```

S3 dispatch and inheritance

The dispatching will look for appropriate methods for all `x` (sub-)classes (in order in which they are set).
S3 inheritance exercise

Look at the inheritance code and understand how it works. Then solve the following exercise.

Exercise 3: (code: 03_inherit.R, solution: 03_inherit_solution.R)

Write the complement() method for RnaSeq class. Since we don’t have a RNA FASTA file you will have to make a new RnaSeq object by hand and assign the right classes to test your code.

What do you notice about the S3 class system, is it easy to make mistakes? Could you also make your RNA sequence to be of class ”lm”?
S3 class system revision

- Classes are implicit (no formal class definition)
- Making new objects is done by simply setting the class attribute
- Making class methods is done by defining a generic function `functionName()` and a normal function `functionName.className()`. Methods can be retrieved using the `methods()` function.
- Objects can inherit multiple classes by setting the class attribute to a vector of class names
- Many functions in base R use the S3 system
- Easy to make new ad-hoc classes and objects, but also mistakes and inconsistencies

The S4 class system is designed to address some of these concerns.
Introduction to S4 class system

Differences of S4 class system to S3

- **Classes are explicit** - they have slots which describe what kind of data is stored
- **Improved introspection** - class, method and slot introspection
- **Consistency checking** - can no longer assign any class name, class hierarchy is explicitly checked and reinforced
- **Validity checking** - custom automatic checks of data consistency
- **Multiple inheritance, multiple dispatch, virtual classes**

S4 class system is the de-facto standard in Bioconductor.
Defining S4 classes

**Defining S4 class**

Each class in S4 needs to be defined before it can be used. At this stage data types and inheritance are specified.

```r
> setClass("GenericSeq",
  +   representation = representation(
  +     id = "character",
  +     alphabet = "character",
  +     sequence = "character"
  + ))
```

**S4 class slots**

Slots define the **names and types** of variables that are going to be stored in the object. Types can be any of the basic R type or S3/S4 classes. To inspect how basic R types are called use `class()` function, e.g.

```r
> class("hello")

[1] "character"
```
Creating S4 objects

Once we have a class definition, we can make an object by filling out the slots. We can directly access the slots using the @ notation although this is discouraged.

```r
> genseq <- new("GenericSeq", id="sequence name",
+       alphabet=c("A", "C", "G", "T"), sequence="AGATACCCCGAAACGA")
> genseq
An object of class "GenericSeq"
Slot "id":
[1] "sequence name"

Slot "alphabet":
[1] "A" "C" "G" "T"

Slot "sequence":
[1] "AGATACCCCGAAACGA"

> genseq@id
[1] "sequence name"

> slot(genseq, "id")
[1] "sequence name"
```
Creating S4 methods

Similar to S3 we define object methods in two steps: by defining a generic and the method.

```r
> setGeneric("rev", function(x) standardGeneric("rev"))

[1] "rev"

> setMethod("rev", "GenericSeq",
+ function(x) paste(rev(unlist(strsplit(x@sequence, ""))), collapse=""))

[1] "rev"

> rev(genseq)

[1] "AGCAAAGCCCCATAGA"

> showMethods("rev")

Function: rev (package .GlobalEnv)
  x="ANY"
  x="character"
    (inherited from: x="ANY")
  x="GenericSeq"
```
S4 accessor methods

It is considered bad practice to use @ in your code to access slots. It breaks the division between the internal class implementation and class usage.

Instead, create getter and setter methods for all slots you want to expose.

```r
> setGeneric("id", function(object) standardGeneric("id"))
[1] "id"
> setMethod("id", "GenericSeq", function(object) object@id)
[1] "id"
> setGeneric("id<-", function(object,value) standardGeneric("id<-"))
[1] "id<-
> setReplaceMethod("id", signature(object="GenericSeq",
+ value="character"),
+ function(object, value) {
+ object@id <- value
+ return(object)
+ })
[1] "id<-
> id(genseq) <- "new sequence name"
> id(genseq)
[1] "new sequence name"
```
**Exercise 4:** *(code: 04_basic_S4.R)*

Try the following introspection functions:

```r
showMethods("rev")
getClass("GenericSeq")
slotNames(genseq)
getMethod("rev", "GenericSeq")
findMethods("rev")
isGeneric("rev")
```

What do these function output? In some cases the result is an object. Use the introspection functions to find out more about the results (e.g. `class()`, `getClass()`,...).

**Exercise 5:** *(code as above, solution: 05_accessors_solution.R)*

Let's complete our `GenericSeq` implementation with some more methods. Implement getter/setter method `seq()` and getter only `alphabet()`. Then implement the method `length()` to return sequence length. First check if "length" is already a generic though.
Special methods - `show()`

You might have noticed that many object print a custom description instead of a plain list of slots. We can add this functionality by setting `show()` and `print()` methods.

```r
> setMethod("show",
+ "GenericSeq",
+ function(object) {
+   cat("Object of class",class(object),"\n")
+   cat(" Id:",id(object),"\n")
+   cat(" Length:",length(object),"\n")
+   cat(" Alphabet:",alphabet(object),"\n")
+   cat(" Sequence:",seq(object), "\n")
+ })

[1] "show"

> genseq

Object of class GenericSeq
  Id: new sequence name
  Length: 16
  Alphabet: A C G T
  Sequence: AGATACCCCGAAACGA
```
Special methods - `print()`

The `print()` function already exists, but is not an S4 generic.

```r
> setGeneric("print", function(x,...) standardGeneric("print"))
[1] "print"

> setMethod("print", "GenericSeq",
+   function(x) {
+     sq <- strsplit(seq(x),"")[1]
+     cat(">",id(x),"
"," 1\t")
+     for (i in 1:length(x)) {
+       if ((i %% 10)==0) {
+         cat("\n",i,"\t")
+       }
+     cat(sq[i])
+   })

[1] "print"

> print(genseq)

> new sequence name
  1 AGATACCCC
  10 GAAACGA
```
**Special methods - initialize()**

We might need to do some special processing on object creation. We can do this with a custom `initialize()` method.

Use named arguments with default values (otherwise class checking might fail).

```r
> setMethod("initialize", "GenericSeq", 
+ function(.Object, ..., id="", sequence=""){
+ .Object@id <- id
+ .Object@sequence <- toupper(sequence)
+ callNextMethod(.Object, ...) # call parent class initialize()
+ })

[1] "initialize"

> show(new("GenericSeq", id="new seq.", alphabet=c("A", "T"), sequence="atatta"))

Object of class GenericSeq
  Id: new seq.
  Length: 6
  Alphabet: A T
  Sequence: ATATTA
Inheritance in S4 class system

Implementation of GenericSeq is finished. Now we want to re-use this code and add some extra functionality for DnaSeq and RnaSeq.

We start by defining the new classes that will inherit (contain) our GenericSeq class. It is good practise to provide some default (prototype) values.

> setClass("DnaSeq",
+   contains="GenericSeq",
+   prototype = prototype(
+     id = paste("my DNA sequence",date()),
+     alphabet = c("A","C","G","T"),
+     sequence = character()
+   )
> setClass("RnaSeq",
+   contains="GenericSeq",
+   prototype = prototype(
+     id = paste("my RNA sequence",date()),
+     alphabet = c("A","C","G","U"),
+     sequence = character()
+   )
Extending child classes with custom methods

**Custom `comp()` methods in two subclasses**

Now we can write the `comp()` method which is going to work differently for DNA and RNA sequences.

```r
> setGeneric("comp",function(object) standardGeneric("comp"))
[1] "comp"

> setMethod("comp","DnaSeq",
+ function(object) {
+     chartr("ACGT","TGCA",seq(object))
+ })
[1] "comp"

> setMethod("comp","RnaSeq",
+ function(object) {
+     chartr("ACGU","UGCA",seq(object))
+ })
[1] "comp"
```
Creating objects of appropriate class

We could use `new()` to create new object instances, but it is tedious and error prone. Instead, we should provide a function that reads in some data and sets the right class for the data.

```r
> readFasta <- function(infile){
+   lines <- readLines(infile)
+   header <- grep("^>" , lines)
+   if (length(header)>1) {
+     warning("Reading first sequence only.")
+     lines <- lines[header[1]: (header[2]-1)]
+     header <- header[1]
+   }
+   .id <- sub("^> *" , "", lines[header], perl=TRUE)
+   .sequence <- toupper(paste(lines[(header+1):length(lines)], collapse=""))
+   .alphabet <- toupper(unique(strsplit(.sequence,""))[1]))
+   if (all(.alphabet %in% c("A","C","G","T"))) {
+     newseq <- new("DnaSeq",
+                   id=.id,
+                   sequence=.sequence)
+   } else if (all(.alphabet %in% c("A","C","G","U"))) {
+     newseq <- new("RnaSeq",
+                   id=.id,
+                   sequence=.sequence)
+   } else {
+     stop("Alphabet ", .alphabet," is unknown.")
+   }
+   return(newseq)
+ }
```
Object validity tests

The user can still use `new` in an inconsistent way or change a consistent object in the way that will render it inconsistent (e.g. assign an RNA sequence to an object of class `DnaSeq`).

First let's make sure each new object is consistent, e.g. that alphabet matches sequence.

```r
> setClass("GenericSeq",
+    representation = representation(
+      id = "character",
+      alphabet = "character",
+      sequence = "character",
+      "VIRTUAL"),
+    validity = function(object) {
+      isValid <- TRUE
+      if (nchar(object@sequence)>0) {
+        chars <- casefold(unique(unlist(strsplit(object@sequence,"")))))
+        isValid <- all(chars %in% casefold(object@alphabet))
+      }
+      if (!isValid)
+        cat("Some characters are not defined in the alphabet.\n")
+      return(isValid)
+    })
```
Validity tests - setters

Now let's make sure the user cannot render the objects inconsistent by modifying the object.

```r
> setReplaceMethod("id",
+ signature(object="GenericSeq",
+ value="character"),
+ function(object, value) {
+ object@id <- value
+ if (validObject(object))
+ return(object)
+ })

[1] "id<-

> setReplaceMethod("seq",
+ signature(object="GenericSeq",
+ value="character"),
+ function(object, value) {
+ object@sequence <- value
+ if (validObject(object))
+ return(object)
+ })

[1] "seq<-
```
S4 exercises

Look at the code we wrote so far and understand it. Then solve the following exercise.

Exercise 6: (code: 06_S4_complete.R)

Try again reading the supplied fasta file using

```r
x <- readFasta("aDnaSeq.fasta")
```

Inspect the resulting object using object introspection tools. Try to break the resulting object by assigning invalid values to sequence. What happens if you do:

```r
seq(x) <- "!"
```

and what if:

```r
x@sequence <- "!"
```

Exercise 7: (code as above, solution: 07_transcribe_solution.R)

Implement a new method `transcribe()` of `DnaSeq`. This method should take a `DnaSeq`, replace the T’s with U’s and return a `RnaSeq` object.
Virtual classes

A class can be marked to be virtual so that no objects can be made, but it can only be inherited. In our case, we might want to mark GenericSeq as virtual, to do so just add parameter "VIRTUAL" into class representation.

Class unions

In some cases we might want a slot to contain an object from one of multiple unrelated classes. In that case we would create a "dummy" class to serve as a place holder. For this we can use class union, for example setClassUnion("AOrB", c("A", "B")) would create a new virtual class AOrB that is a parent class to both A and B.

Overriding operators

Operators in R can also be over-ridden. For instance setMethod("[", MyClass, ....) will override the subsetting operator [] for MyClass to give it custom functionality.
Reference classes

**Mutability**

R objects are not **mutable**; R has a **copy on modify** semantics: whenever you pass an object to a function, a copy is passed as argument. This is how things work for both S3 and S4 class systems.

```r
> a <- new("DnaSeq", sequence="ACGTaa")
> seq(a)
[1] "ACGTAA"
> comp(a)
[1] "TGCATT"
> seq(a)
[1] "ACGTAA"
```
Reference classes

A recent OO system, based on S4 classes, that implements a pass by reference semantic. See ?ReferenceClasses for details.

Example

```r
## here, you would have
> a$seq ## equivalent of seq(a)
[1] "AGCATG"
> a$comp()
> a$seq
[1] "TCGTAC"
```
Plan

1 Introduction
2 Revision of basic R
3 Object-oriented (OO) Programming
4 A few words about R packages
5 Package structure
6 Writing R documentation
7 Other advanced topics
   • Testing your package
   • Debugging
   • Profiling
   • Calling foreign languages
8 Distributing packages
References

- R Installation and Administration [R-admin], R Core team
- Writing R Extensions [R-ext], R Core team

Use `help.start()` to access them from your local installation, or http://cran.r-project.org/manuals.html from the web.

Terminology

A **package** is loaded from a **library** by the function `library()`. Thus a library is a directory containing installed packages.

Calling `library("foo", lib.loc = "/path/to/bar")` loads the package (book) `foo` from the library `bar` located at `/path/to/bar`. 
Packages

One of the aspects that make R appealing:

**CRAN** package repository features 2868 available packages.

**R-forge** 986 packages.

**Bioconductor** 517 reviewed packages in latest release (version 2.9).

Numbers checked on 2\textsuperscript{nd} March 2011
**Why packages**

*Packages provide a mechanism for loading optional code and attached documentation as needed.*

There is more to it – packages are a means to

- logically group your own functions
- keep code and documentation together and consistent
- keep code and data together
- keep track of changes in code
- summarise all packages used for a analysis (see `sessionInfo()`)
- make a reproducible research compendium (container for code, text, data as a means for distributing, managing and updating)
- optionally test your code
- … project management

even if you do not plan to distribute them.
Building packages

R CMD build myPackage – the R package builder builds R package (and vignettes if available).

Checking packages

R CMD check myPackage_0.1.1.tar.gz or R CMD check myPackage – the R package checker tests whether the package or source work correctly.

- The package is installed (checks missing cross-references and duplicate aliases in help files).
- File names validity, permissions.
- Package DESCRIPTION file is checked for completeness, and some of its entries for correctness.
- R and .Rd files are checked for syntax errors.
- A check is made for missing documentation entries.
- Codoc checking
- Examples provided by the package’s documentation are run.
- If available, package tests are run and vignettes are executed and compiled.
Installing packages

R CMD INSTALL myPackage_0.1.1.tar.gz or install.packages("myPackage_0.1.1.tar.gz") – installs the package in the default library. Other libraries can be specified with the -l option or lib argument.

Loading

Use library() or require().

On Windows

R is very much Unix centric. To build from source on Windows, you will need Rtools. See the *The Windows toolset* in R-Admin for more details.

http://www.murdoch-sutherlan.com/Rtools/
Plan

1. Introduction
2. Revision of basic R
3. Object-oriented (OO) Programming
4. A few words about R packages
5. **Package structure**
6. Writing R documentation
7. Other advanced topics
   - Testing your package
   - Debugging
   - Profiling
   - Calling foreign languages
8. Distributing packages
A minimal package

Function package.skeleton() automates some of the setup for a new source package. Calling it with minimal arguments produces the following hierarchy:

```r
> foo <- function(x) x
> package.skeleton(name="myRpackage",list="foo")
```

produces

myRpackage/
|-- DESCRIPTION
|-- man
| |-- foo.Rd
| `-- myRpackage-package.Rd
|-- R
| `-- foo.R
`-- Read-and-delete-me

2 directories, 5 files
Package: myRpackage ## mandatory (*)
Type: Package ## optional, 'Package' is default type
Title: What the package does (short line) ## mandatory
Version: 1.0 ## *
Date: 2011-03-01 ## release date of the current version
Author: Who wrote it ## *
Maintainer: Who to complain to <yourfault@somewhere.net> ## *
Description: More about what it does (maybe more than one line) ## *
License: What license is it under? ## *
LazyLoad: yes
Lazy loading

A mechanism used to defer initialization of an object until the point at which it is needed. The individual objects in the package’s environment are indirect references to the actual objects until, for example a function is called or an object loaded.

The LazyLoad and LazyData fields control whether the R objects and the datasets (respectively) use lazy-loading. LazyLoad must be set if the methods package is used.

LazyLoad is now on by default.
R uses *Lazy evaluation*, which delays the evaluation of an expression (here the argument) until its value is actually required \[^a\] :

```r
> f <- function(x) { 10 }
> system.time(f(Sys.sleep(3)))

user  system elapsed
 0.000   0.000   0.000

> f <- function(x) { force(x); 10 }
> system.time(f(Sys.sleep(3)))

user  system elapsed
 0.000   0.000   3.003
```

\[^a\] example from Hadley Wickham’s devtools
Lazy Example

```r
> suppressWarnings(dump("dnaseq","",evaluate=FALSE))
dnaseq <-
<promise: lazyLoadDBfetch(c(0L, 195L), datafile, compressed, envhook)>
```
### Other important fields

**Depends**  
A comma-separated list of package names (optionally with versions) which this package depends on.

**Suggests**  
Packages that are not necessarily needed: used only in examples, tests or vignettes, packages loaded in the body of functions (see `require()`).

**Imports**  
Packages whose name spaces are imported from (as specified in the `NAMESPACE` file) but which do not need to be attached to the search path.

**Collate**  
Controls the collation order for the R code files in a package. If `filed` is present, all source files must be listed.

**URL**  
A list of URLs separated by commas or whitespace.

...
The NAMESPACE file

Stored in the package directory. Restrict the symbols that are exported and imports functionality from other packages. Only the exported symbols will have to be documented.

Note: NAMESPACE is now required (since R 2.14).

```r
export(f, g) ## exports f and g
exportPattern("^[^\./]"")
import(foo) ## imports all symbols from package foo
importFrom(foo, f, g) ## imports f and g from foo
```

It is possible to explicitly use symbol `s` from package `foo` with `foo:::s` or `foo:::s` if `s` is not exported.
Attach and load

Packages are attached to the search path with library or require.

**Attach**  When a package is attached, then all of its dependencies (see Depends field in its DESCRIPTION file) are also attached. Such packages are part of the evaluation environment and will be searched.

**Load**  One can also use the Imports field in the NAMESPACE file. Imported packages are loaded but are not attached: they do not appear on the search path and are available only to the package that imported them.
Package subdirectories

R

Contains `source()`able R source code to be installed. Files must start with an ASCII (lower or upper case) letter or digit and have one of the extensions `.R` (recommended), `.S`, `.q`, `.r`, or `.s`. File order is important if code relies on earlier code – order use `Collate` filed in DESCRIPTION file.

Example

```r
## works fine without Collate field
AllGenerics.R    DataClasses.R
methods-ClassA.R methods-ClassB.R
functions-ClassA.R ...
```

`zzz.R` is generally used to define special functions used to initialize (called after a package is loaded and attached) and clean up (just before the package is detached). See `help(".onLoad")`, `?.First.Lib` and `?.Last.Lib` for more details.
Package subdirectories

**man**

Manuals for the objects (package, functions, generics, methods, classes and data sets) in the package in R documentation (.Rd) format. The filenames must start with an ASCII (lower or upper case) letter or digit and have the extension .Rd or .rd and should be URL compatible. If you use a NAMESPACE, only exported symbols need to be documented. Without NAMESPACE, internal use only objects should be documented in pkg-internal.Rd.
Package subdirectories

**data**

Contains data files, made available via *lazy-loading* or for loading using `data()`. Data types that are allowed are

- **R code** self-sufficient plain R code (`.R` or `.r`),
- **Tables** possibly compressed tables (`.tab`, `.txt`, or `.csv`, see `?data` for the file formats)
- **Objects** created using `save()` (`.RData` or `.rda`).

**Example**

There is a `DnaSeq` object in `sequences/data`. 
Package subdirectories

**inst**
Content is copied recursively to the installation directory, for example

* CITATION file (see `citation()` function),
* doc directory for additional documents (see vignettes, later).
* extdata directory for other data files, not belonging in data.
* tests code for unit tests (see later).

**Example**
In our `sequences` package, there is a fasta sequence in `sequences/inst/extdata` used to illustrate the `readFasta` function.
Package subdirectories

**tests**
Contains additional package-specific test code. We will talk about unit tests later.

**src**
Contains sources and headers for the compiled code, plus optionally a file Makevars or Makefile.

**demo**
R scripts runned via demo() that demonstrate some of the functionality of the package. Execution of these scripts is not checked.
Exercise 8: Let’s create a package

So far, you have defined a set of classes, methods and functions ... Create the required directory structure and files using `package.skeleton(name="sequences")` or manually. For the former, you can use different arguments:

- **list** to specify the R objects by their names.
- **code_files** to specify R code files.
- **environment** to specify an environment where objects are looked for.

See `?package.skeleton` for more details.
Exercise 9: Let’s build/check it

Do you expect the package to build/check/INSTALL:

R CMD build sequences
R CMD check sequences_1.0.tar.gz
R CMD INSTALL sequences_1.0.tar.gz

Why? Have a look at R CMD build|check --help.
**Plan**

1. Introduction
2. Revision of basic R
3. Object-oriented (OO) Programming
4. A few words about R packages
5. Package structure
6. **Writing R documentation**
7. Other advanced topics
   - Testing your package
   - Debugging
   - Profiling
   - Calling foreign languages
8. Distributing packages
R documentation format

R objects are documented in files written in R documentation (Rd) format, a simple markup language much of which closely resembles \LaTeX, which can be processed into a variety of formats, including \LaTeX, HTML, pdf and plain text.
Rd format

An Rd file consists of

- **Header** provides basic information about the name of the file, the topics documented, a title, a short textual description and R usage information – mandatory.

- **Body** gives further information defined within sections (for example, on the function’s arguments and return value, as in the example)

- **Footer** with keyword information – optional.

Every (exported) object must be documented. Package documentation is optional.
Reload Saved Datasets

Reload the datasets written to a file with the function \code{save}.

\usage{load(file, envir = parent.frame())}

\arguments{
  \item{file}{a connection or a character string giving the name of the file to load.}
  \item{envir}{the environment where the data should be loaded.}
}

\seealso{\code{\link{save}}.}

\examples{
## save all data
save(list = ls(), file= "all.RData")

## restore the saved values to the current environment
load("all.RData")

## restore the saved values to the workspace
load("all.RData", .GlobalEnv)
}

\keyword{file}
**General comments**

- Different objects are documented with different types of Rd files, as defined by the `\docType{}` tag.
- Different object documentation require or are advised to contain different sections.
- One `.Rd` file can document several objects by defining multiple `\alias{}`'es.
Guidelines for Rd files

These are suggested guidelines for the system help files (in .Rd format) that are intended for core developers but may also be useful for package writers. (see http://developer.r-project.org/Rds.html)

There are many different sections and marking text (for mathematical notation, tables, cross-references, ...), that will look very familiar to \LaTeX users. All are described in Writing R documentation files (section 2) of the R-ext manual.

Fortunately, the prompt(object) et. al. functions will inspect the object to be documented and create a specific documentation skeleton for us to be completed.
Package documentation

Provides an short and optional overview of a package.

Example

promptPackage("sequences")

Exercise 10:
Create a sequences-package.Rd and document your package.
Data sets documentation

**Example**

\name{rivers}
\docType{data}
\alias{rivers}
\title{Lengths of Major North American Rivers}
\description{
   This data set gives the lengths (in miles) of 141 \dQuote{major} rivers in North America, as compiled by the US Geological Survey.
}
\usage{rivers}
\format{A vector containing 141 observations.}
\source{World Almanac and Book of Facts, 1975, page 406.}
\references{
}
\keyword{datasets}

**Example**

prompt(myDataFrame) or promptData(myDataObject)

**Exercise 11:**

document the dnaseq object.
Many markup command, including \usage{fun(arg1, arg2, ...)}, \arguments{...}, \section{Warning}{...} and \examples{...}, which are executed!

**Example**

prompt(object=myFunction) or prompt(name="myFunction")

**Exercise 12:**

Choose one of the functions and document it.
Documenting S4 classes and methods

Documentation is 'similar' than for functions. Note that aliases are of the form MyClass-class or MyGeneric,signature_list-method. Additionnal aliases should be added to refer to MyGeneric, MyGeneric-method, ... and the manuals are accessed with class?topic and method?topic. Overall documentation for methods should be aliased with MyGeneric-methods
See help("Documentation", package = "methods") for more details.

Example

promptClass("MyClass") and promptMethods("myMethod")

Exercise 13:

Document one class and one method of the package.
NB: we have used aliases for the methods to refer to the class documentation.
Roxygen2

What is it?
Roxygen is a Doxygen-like documentation system for R; allowing in-source specification of Rd files, collation and namespace directives.

See https://github.com/klutometis/roxygen.

Install with `install.packages("roxygen2")`.

Use R CMD roxygen myPackage to generate manuals and NAMESPACE.
## Read fasta files.

This function reads DNA and RNA fasta files and generates valid `DnaSeq` and `RnaSeq` instances.

### Examples

```r
f <- dir(system.file("extdata", package="sequences"), pattern="fasta", full.names=TRUE)

f <- readFasta(f)

@title Read fasta files.
@param infile the name of the fasta file which the data are to be read from.
@return an instance of `DnaSeq` or `RnaSeq`.
@seealso `GenericSeq`, `DnaSeq` and `RnaSeq`.
@keywords IO, file

```
**Good points**
Makes (1) to get from code to full package straightforward and also (2) maintenance much easier.

**Since roxygen2**
S4 support (classes, generics, methods).

**See also**
Rd2roxygen – Convert Rd to roxygen documentation and utilities to improve documentation
http://cran.r-project.org/web/packages/Rd2roxygen/index.html
**Package vignette**

These *executable* documents are in Sweave format (.Rnw extension), which is an extended \(\text{\LaTeX}\) document that includes code chunks. These are executed and the output (variable, but also tables and graphs) are displayed in the document. These dynamic reports, are updated automatically if data or analysis change.

The package vignettes are compiled at build time and are the preferred place for more extensive package documentation and use-cases.

Reference: http://www.stat.uni-muenchen.de/~leisch/Sweave/
\begin{Schunk}
\begin{Sinput}
<<label=myCode,echo=TRUE,fig=TRUE>>= 
\begin{Schunk}
x <- sort(rnorm(100))
y <- sort(rnorm(100,2,2))
plot(x,y,pch=19,col="#0000BB80")
abline(lm(y~x))
@ 
\end{Schunk}
\end{Sinput}
\end{Schunk}
> x <- sort(rnorm(100)); y <- sort(rnorm(100,2,2))
> plot(x,y,pch=19,col="#0000BB80"); abline(lm(y~x))
Example

Have a look at the `sequences` package vignette in `sequences/inst/doc`.

Exercise 14:

The vignette is very basic. Try to add some code chunks to improve it. You can also embed code in-line with `\Sexpr{}`.
sessionInfo()

Prints version information about R and attached or loaded packages.

Example

> sessionInfo()

R version 2.15.1 (2012-06-22)
Platform: x86_64-pc-linux-gnu (64-bit)

locale:
[1] LC_CTYPE=en_GB.UTF-8 LC_NUMERIC=C
[3] LC_TIME=en_GB.UTF-8 LC_COLLATE=en_GB.UTF-8
[5] LC_MONETARY=en_GB.UTF-8 LC_MESSAGES=en_GB.UTF-8
[7] LC_PAPER=C LC_NAME=C
[9] LC_ADDRESS=C LC_TELEPHONE=C

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

loaded via a namespace (and not attached):
[1] tools_2.15.1
sessionInfo() in vignettes

Example

\begin{verbatim}
> toLatex(sessionInfo())
\end{verbatim}

- R version 2.15.1 (2012-06-22), x86_64-pc-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, utils

- Loaded via a namespace (and not attached): tools 2.15.1
Plan

1. Introduction
2. Revision of basic R
3. Object-oriented (OO) Programming
4. A few words about R packages
5. Package structure
6. Writing R documentation
7. Other advanced topics
   - Testing your package
   - Debugging
   - Profiling
   - Calling foreign languages
8. Distributing packages
How to test the code in your package?

Or how to make sure that changes in your code do not break existing functionality?

- Implicitely, documentation examples and a vignette do some tests.
- Using R’s build-in testing, that runs some code and compares the output to a saved template.
- Specific packages for unit testing: RUnit\(^a\) or testthat\(^b\).

\(^a\)http://cran.r-project.org/web/packages/RUnit/index.html
\(^b\)http://cran.r-project.org/web/packages/testthat/index.html
Using an `.Rout.save` file

In `package/tests/`

Create

- `mytest.R` with code to be tested
- `mytest.Rout.save` with the reference output

When checking your package R will

1. execute the code in `mytest.R`
2. save the output to `mytest.Rout`
3. compare `mytest.Rout` to `mytest.Rout.save`
4. report any differences
Using testthat

**Test individual expression**

\[
\text{expect\_that}\text{(object\_or\_expression, condition)} \text{ with conditions}
\]

- **equals**  
\[
\text{expect\_that}(1+2, \text{equals}(3)) \text{ or expect\_equal}(1+2, 3)
\]

- **gives\_warning**  
\[
\text{expect\_that}(\text{warning}("a"), \text{gives\_warning}())
\]

  - **is\_a**  
\[
\text{expect\_that}(1, \text{is\_a}("numeric")) \text{ or} \\
\text{expect\_is}(1,"numeric")
\]

  - **is\_true**  
\[
\text{expect\_that}(2 == 2, \text{is\_true}()) \text{ or} \\
\text{expect\_true}(2==2)
\]

  - **matches**  
\[
\text{expect\_that}("Testing is fun", \text{matches}("fun")) \text{ or} \\
\text{expect\_match}("Testing is fun", "f.n")
\]

- **takes\_less\_than**  
\[
\text{expect\_that}(\text{Sys.sleep}(1), \text{takes\_less\_than}(3))
\]

...
Using testthat

Example

```r
> library(testthat)
> test_that("ok test", {
+   expect_equal(length(a),6)
+   expect_true(seq(a)=="ACGTAA")
+   expect_is(a,"DnaSeq")
+ })
> try(expect_true(seq(a)=="ACGTaa")) ## fails with
> ## Error: seq(aa) == "ACGTaa" isn't true
```
Exercise 15:


Hint: check the updated package and look in `sequences.Rcheck/tests/`
Using R’s tools

- Call traceback() after error to print the sequence of calls that lead to the error.
- Use debug(faultyFunction) to register faultyFunction for debugging, so that browser() will be called on entry. In browser mode, the execution of an expression is interrupted and it is possible to inspect the environment (with ls()). Use undebug(faultyFunction) to revert to normal usage.
- Use trace() to insert code into functions, start the browser or recover() from error.
- Set options(error=recover) to get the call stack and browse in any of the function calls.

Good reference: An Introduction to the Interactive Debugging Tools in R

http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf
Exercise 16:

Let’s debug sequences’ readFasta function:

1. Prepare for debugging: `debug(readFasta)`
2. Let get a *fasta* file from the package:
   ```r
dir(system.file(dir="extdata", package="sequences"), full.names=TRUE)
```
3. Call the function to be debugged: `readFasta(fasta)`
4. Debug!

Hint: when debugging, use `n` (or an empty line) to advance to the next step, `c` to continue to the end of the current context (to the end of a loop for instance), `w` to print the stack trace of all active function calls and `Q` to exit the browser.

Other hint: use `ls(all.names=TRUE)` to see all objects, also those that start with a ‘.’.
Profiling time

Measuring time

```r
> m <- matrix(runif(1e4), nrow=1000)
> system.time(apply(m, 1, sum))

    user  system elapsed
   0.004   0.000   0.002
```

Replicate

```r
> replicate(5, system.time(apply(m, 1, sum))[[1]])

[1] 0.000 0.000 0.004 0.004 0.004
```
Profiling time

Execution time

```r
> Rprof("rprof")
> res <- apply(m,1,mean,trim=.3)
> Rprof(NULL); summaryRprof("rprof")

$by.self

<table>
<thead>
<tr>
<th>Function</th>
<th>Self Time</th>
<th>Self Pct</th>
<th>Total Time</th>
<th>Total Pct</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;mean.default&quot;</td>
<td>0.02</td>
<td>33.33</td>
<td>0.06</td>
<td>100.00</td>
</tr>
<tr>
<td>&quot;any&quot;</td>
<td>0.02</td>
<td>33.33</td>
<td>0.02</td>
<td>33.33</td>
</tr>
<tr>
<td>&quot;unique.default&quot;</td>
<td>0.02</td>
<td>33.33</td>
<td>0.02</td>
<td>33.33</td>
</tr>
</tbody>
</table>

$by.total

<table>
<thead>
<tr>
<th>Function</th>
<th>Total Time</th>
<th>Total Pct</th>
<th>Self Time</th>
<th>Self Pct</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;mean.default&quot;</td>
<td>0.06</td>
<td>100.00</td>
<td>0.02</td>
<td>33.33</td>
</tr>
<tr>
<td>&quot;apply&quot;</td>
<td>0.06</td>
<td>100.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>&quot;FUN&quot;</td>
<td>0.06</td>
<td>100.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>&quot;any&quot;</td>
<td>0.02</td>
<td>33.33</td>
<td>0.02</td>
<td>33.33</td>
</tr>
<tr>
<td>&quot;unique.default&quot;</td>
<td>0.02</td>
<td>33.33</td>
<td>0.02</td>
<td>33.33</td>
</tr>
<tr>
<td>&quot;sort.int&quot;</td>
<td>0.02</td>
<td>33.33</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>&quot;unique&quot;</td>
<td>0.02</td>
<td>33.33</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

$sample.interval

[1] 0.02

$sampling.time

[1] 0.06
Memory usage using `tracemem`

> a
Object of class DnaSeq
  Id:
  Length: 6
  Alphabet: A C G T
  Sequence: GCATCA

> tracemem(a)
[1] "<0x20f0978>"

> seq(a) <- "GTGT"

* `tracemem` requires to build R with `--enable-memory-profiling`
Calling foreign languages

**The difficult route**

- When R is getting too slow or is not doing well in terms of memory management.
- Implement the heavy stuff in C, C++\(^a\), Fortran or Java\(^b\).

\(^a\)http://dirk.eddelbuettel.com/code/rcpp.html
\(^b\)http://www.rforge.net/rJava/

**Other scripting languages**

- R/Perl\(^a\) and R/Python\(^b\) bidirectional interfaces.
- There is also the `system()` function for direct access to OS functions.

\(^a\)http://www.omegahat.org/RSPerl/
\(^b\)http://www.omegahat.org/RSPython/
R’s build-in C interfaces

- Better know how to program in C.
- Documentation is not always easy to follow: R-Ext, R Internals as well as R and other package’s code.

.C

- Easy way
- Arguments and return values must be *primitive* (vectors of doubles or integers)

.Call

- Accepts R data structures as arguments and return values (SEXP and friends) (no type checking is done though).
- Memory management: memory allocated for R objects is garbage collected. Thus R objects in C code, you must be explicitly PROTECTed to avoid being gc()ed, and subsequently UNPROTECTed.
Example

```c
#include <R.h>
#include <Rdefines.h>

SEXP gc_count(SEXP inseq) {
    int i, l;
    SEXP ans, dnaseq;
    PROTECT(dnaseq = STRING_ELT(inseq, 0));
    l = LENGTH(dnaseq);
    printf("length %d\n", l);
    PROTECT(ans = NEW_NUMERIC(4));

    for (i = 0; i < 4; i++)
        REAL(ans)[i] = 0;

    for (i = 0; i < l; i++) {
        char p = CHAR(dnaseq)[i];
        if (p=='A')
            REAL(ans)[0]++;
        else if (p=='C')
            REAL(ans)[1]++;
        else if (p=='G')
            REAL(ans)[2]++;
        else if (p=='T')
            REAL(ans)[3]++;
        else
            error("Wrong alphabet");
    }
    UNPROTECT(2);
    return(ans);
}
```
Using your C code

**Directly**

1. Create a shared library: 
   ```
   R CMD SHLIB gccount.c
   ```

2. Load the shared object: 
   ```
   dyn.load("gccount.so")
   ```

3. Create an R function that uses it: 
   ```
   gccount <- function(inseq) .Call("gccount", inseq)
   ```

4. Use your C code: 
   ```
   gccount("GACAGCATCA")
   ```

**In a package**

- Document your function.
- Overwrite `.First.lib` to `dyn.load` your shared object.
- If you have a `NAMESPACE`, export the shared objects with `useDynLib`.
Example

In sequences, we have

- The gccount.c code in src.
- Defined a R function in R/functions.R
  ```r
  gccount <- function(inseq) {
    .Call("gccount",
          inseq,
          PACKAGE="sequences")
  }
  ```
- Written the man/gccount.Rd man page.
- Exported the function in NAMESPACE using export(gccount) and the shared library with useDynLib(sequences)
Example

```r
> s <- "GACTACGA"
> gccount

function (inseq)
{
  .Call("gccount", inseq, PACKAGE = "sequences")
}

<environment: namespace:sequences>

> gccount(s)

[1]  3  2  2  1

> table(strsplit(s,""))

A C G T
3 2 2 1
```
Plan

1. Introduction
2. Revision of basic R
3. Object-oriented (OO) Programming
4. A few words about R packages
5. Package structure
6. Writing R documentation
7. Other advanced topics
   - Testing your package
   - Debugging
   - Profiling
   - Calling foreign languages
8. Distributing packages
**Submission**

**CRAN**  Upload your checked myPackage_x.y.z.tar.gz to ftp://cran.R-project.org/incoming and optionally send a message to CRAN@R-project.org. Your package will be installable with `install.packages("myRpackage")`.

**R-forge**  Log in, register a project and wait for acceptance. Then commit your code to the svn repository. Your package will be installable with `install.packages` using `repos="http://R-Forge.R-project.org"`.

**Bioconductor**  Make sure to satisfy submission criteria (pass check, have a vignette, use S4, have a `NAMESPACE`, make use of appropriate existing infrastructure, include a NEWS file, must **not** already be on CRAN, ...) and submit by email. Your package will then be reviewed before acceptance. A svn account will then be created. Package will be installable with `biocLite("myPackage")`. 
Reference

**Further reading**

- R Installation and Administration, R Core team
- Writing R Extensions, R Core team
- R Programming for Bioinformatics, Robert Gentleman
- http://logic.sysbiol.cam.ac.uk/teaching/advancedR/
Acknowledgement

- Graduate School of Life Sciences
- Jenny Barna
- David Judge
- Paul Judge
Thank you for your attention.